Algorithms and Data Structures

Part 1: Fundamentals

Department of Mathematics
Stockholm University

Part 1 focuses on the following topics.

1-1 What is an algorithm?

1-2 Correctness of algorithms

1-3 Runtime of algorithms & space complexity

The latter Parts **1-1**, **1-2**, **1-3** will, in particular, be examined on a sorting algorithm `Insertion_Sort`. Further examples will be provided. We then continue with

1-4 Elementary Data Structures

Part 1-1: What is an algorithm?

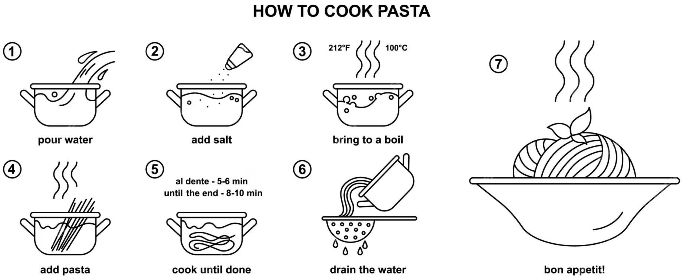
Part 1-1: What is an algorithm?

Informally, an algorithm is a step-by-step unambiguous instructions to solve a given problem by taking some some value(s) as input and by producing some value(s) as output.

Part 1-1: What is an algorithm?

Informally, an algorithm is a step-by-step unambiguous instructions to solve a given problem by taking some some value(s) as input and by producing some value(s) as output.

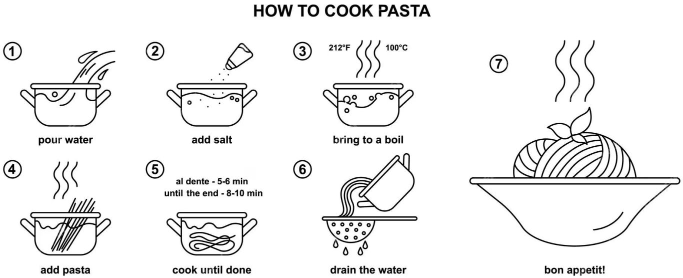
```
Cooking_Pasta(Water, Pasta, Salt)
    Add 1ℓ water to pot
    Add salt to pot
    Boil-up Water
    Add pasta to pot
    Cook until done
    Drain water
    RETURN Cooked delicious pasta
```



Part 1-1: What is an algorithm?

Informally, an algorithm is a step-by-step unambiguous instructions to solve a given problem by taking some some value(s) as input and by producing some value(s) as output.

```
Cooking_Pasta(Water, Pasta, Salt)
    Add 1ℓ water to pot
    Add salt to pot
    Boil-up Water
    Add pasta to pot
    Cook until done
    Drain water
    RETURN Cooked delicious pasta
```

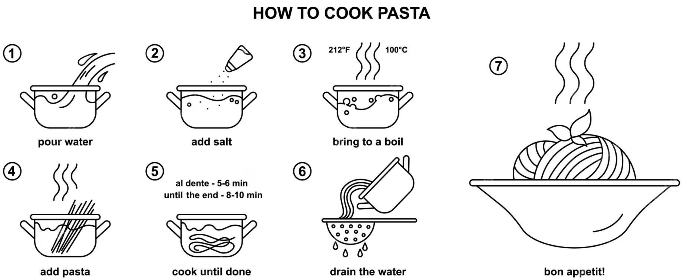


Question: unambiguous? executable by some machine/robot? . . .

Part 1-1: What is an algorithm?

Informally, an algorithm is a step-by-step unambiguous instructions to solve a given problem by taking some some value(s) as input and by producing some value(s) as output.

```
Cooking_Pasta(Water, Pasta, Salt)
    Add 1ℓ water to pot
    Add salt to pot
    Boil-up Water
    Add pasta to pot
    Cook until done
    Drain water
    RETURN Cooked delicious pasta
```



Question: unambiguous? executable by some machine/robot? ...

A human may know how to "boil-up" water by using a cooking plate (...or open fire ...?) but does a robot know this?

Part 1-1: What is an algorithm?

Informally, an algorithm is a step-by-step unambiguous instructions to solve a given problem by taking some some value(s) as input and by producing some value(s) as output.

`Bogo_Sort`* (n cards)

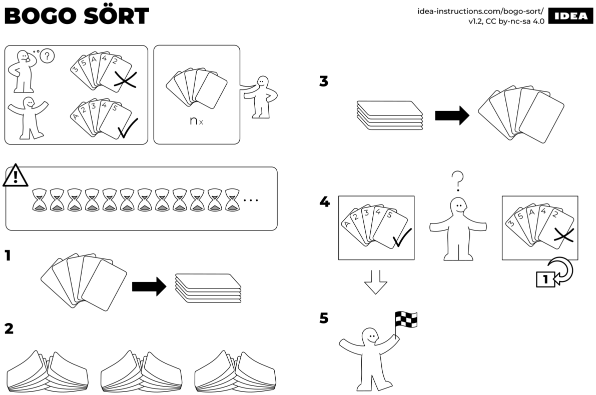
1 Align cards to a pack-of-cards

2 Shuffle cards 3 times

3 Spread cards

4 `IF` (*cards are ordered*) `THEN`
goto step 5
`ELSE` goto step 1

5 `RETURN` Sorted Cart Deck



---

* AKA stupid sort

Part 1-1: What is an algorithm?

Informally, an algorithm is a step-by-step unambiguous instructions to solve a given problem by taking some some value(s) as input and by producing some value(s) as output.

`Bogo_Sort`* (*n* cards)

1 Align cards to a pack-of-cards

2 Shuffle cards 3 times

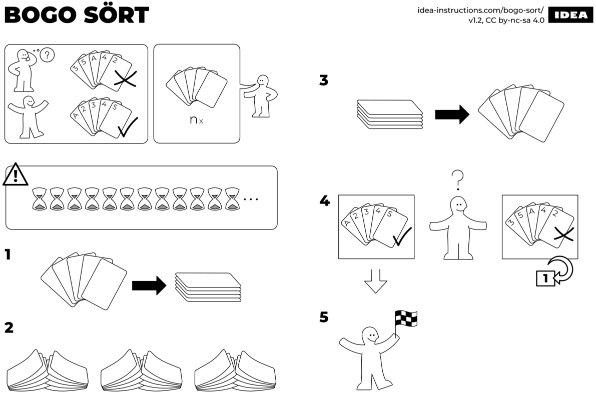
3 Spread cards

4 `IF` (*cards are ordered*) `THEN`

    goto step 5

  `ELSE` goto step 1

5 `RETURN` Sorted Cart Deck



Question: unambiguous (is "order" well-defined)? does it terminate (runtime)?...

---

* AKA stupid sort

Part 1-1: What is an algorithm?

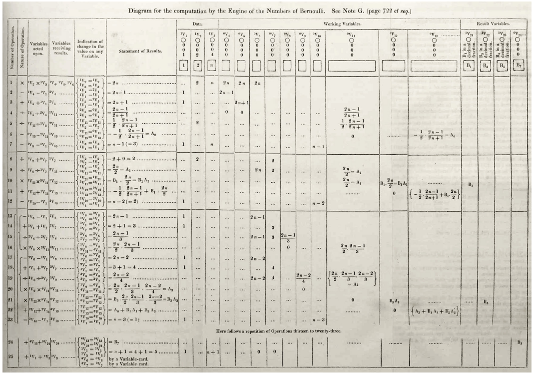
(780–850) Persian mathematician, astronomer, geographer, . . .

The word 'algorithm' has its roots in the name of Persian mathematician Muhammad ibn Musa **al-Khwarizmi**.

He wrote a fundamental treatise on the "Hindu–Arabic numeral system" which was translated into Latin during the 12th century.

Here: al-Khwarizmi was translated into Algorizmi

Part 1-1: What is an algorithm?



(1815-1852) English mathematician

The first computer program was written by Ada Lovelace for the “Analytical Engine” [design for a simple mechanical computer] by Charles Babbage to compute Bernoulli numbers.

Part 1-1: What is an algorithm?

The formal definition of algorithm goes back to Alan Turing who designed Turing machines as a theoretical concept to simulate the operating principles of a computer (central processing unit = CPU)

(1912-1954) English mathematician, computer scientist, logician, . . .

Excellent overview of Turing machines: https://plato.stanford.edu/entries/turing-machine/ https://www.youtube.com/watch?v=dNRDvLACg5Q

Part 1-1: What is an algorithm?

The formal definition of algorithm goes back to Alan Turing who designed Turing machines as a theoretical concept to simulate the operating principles of a computer (central processing unit = CPU)

A Turing machine is a mathematical model of computation describing an abstract machine that manipulates symbols on a strip of tape according to a table of rules.

(1912-1954) English mathematician, computer scientist, logician, . . .

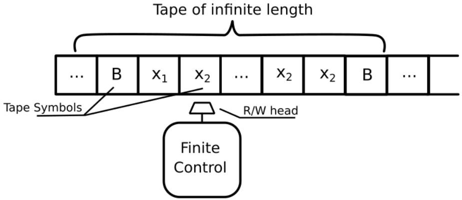
Excellent overview of Turing machines:

https://plato.stanford.edu/entries/turing-machine/
https://www.youtube.com/watch?v=dNRDvLACg5Q

Part 1-1: What is an algorithm?

The formal definition of algorithm goes back to Alan Turing who designed Turing machines as a theoretical concept to simulate the operating principles of a computer (central processing unit = CPU)

A Turing machine is a mathematical model of computation describing an abstract machine that manipulates symbols on a strip of tape according to a table of rules.



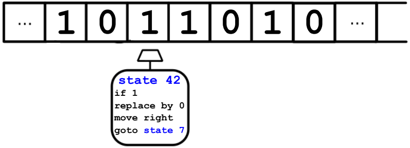
(1912-1954) English mathematician, computer scientist, logician, . . .

Excellent overview of Turing machines: https://plato.stanford.edu/entries/turing-machine/
https://www.youtube.com/watch?v=dNRDvLACg5Q

Part 1-1: What is an algorithm?

The formal definition of algorithm goes back to Alan Turing who designed Turing machines as a theoretical concept to simulate the operating principles of a computer (central processing unit = CPU)

A Turing machine is a mathematical model of computation describing an abstract machine that manipulates symbols on a strip of tape according to a table of rules.



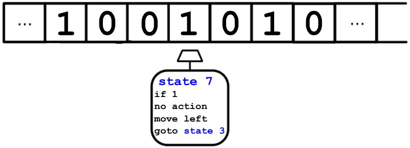
(1912-1954) English mathematician, computer scientist, logician, . . .

Excellent overview of Turing machines: https://plato.stanford.edu/entries/turing-machine/
https://www.youtube.com/watch?v=dNRDvLACg5Q

Part 1-1: What is an algorithm?

The formal definition of algorithm goes back to Alan Turing who designed Turing machines as a theoretical concept to simulate the operating principles of a computer (central processing unit = CPU)

A Turing machine is a mathematical model of computation describing an abstract machine that manipulates symbols on a strip of tape according to a table of rules.



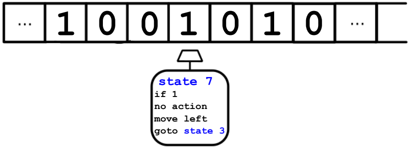
(1912-1954) English mathematician, computer scientist, logician, . . .

Excellent overview of Turing machines: https://plato.stanford.edu/entries/turing-machine/ https://www.youtube.com/watch?v=dNRDvLACg5Q

Part 1-1: What is an algorithm?

The formal definition of algorithm goes back to Alan Turing who designed Turing machines as a theoretical concept to simulate the operating principles of a computer (central processing unit = CPU)

A Turing machine is a mathematical model of computation describing an abstract machine that manipulates symbols on a strip of tape according to a table of rules.



(1912-1954) English mathematician, computer scientist, logician, . . .

Whatever any computer can do, can be done by a Turing machine !!

Excellent overview of Turing machines: https://plato.stanford.edu/entries/turing-machine/ https://www.youtube.com/watch?v=dNRDvLACg5Q

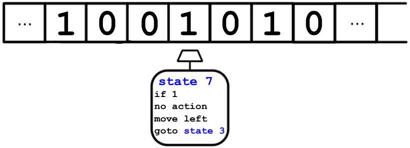
Part 1-1: What is an algorithm?

Definition (algorithm)

A calculation rule for a problem is called ***algorithm*** *if there is a Turing machine equivalent to this calculation rule which stops for every input that has a solution.*

The formal definition of algorithm goes back to Alan Turing who designed Turing machines as a theoretical concept to simulate the operating principles of a computer (central processing unit = CPU)

A Turing machine is a mathematical model of computation describing an abstract machine that manipulates symbols on a strip of tape according to a table of rules.



(1912-1954) English mathematician, computer scientist, logician, . . .

Whatever any computer can do, can be done by a Turing machine !!

Excellent overview of Turing machines: https://plato.stanford.edu/entries/turing-machine/ https://www.youtube.com/watch?v=dNRDvLACg5Q

Part 1-1: What is an algorithm?

Definition (algorithm)

A calculation rule for a problem is called ***algorithm*** *if there is a Turing machine equivalent to this calculation rule which stops for every input that has a solution.*

To check whether a "calculation rule / program" is in fact an algorithm, we must design a Turing machine mimicking these rules.

Part 1-1: What is an algorithm?

Definition (algorithm)

A calculation rule for a problem is called ***algorithm*** *if there is a Turing machine equivalent to this calculation rule which stops for every input that has a solution.*

To check whether a "calculation rule / program" is in fact an algorithm, we must design a Turing machine mimicking these rules.

This is beyond the scope of this course.

Part 1-1: What is an algorithm?

Definition (algorithm)

A calculation rule for a problem is called ***algorithm*** *if there is a Turing machine equivalent to this calculation rule which stops for every input that has a solution.*

To check whether a "calculation rule / program" is in fact an algorithm, we must design a Turing machine mimicking these rules.

This is beyond the scope of this course.

However, to establish algorithms and to analyze their costs, we need to have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs while still having a "correct" notion of algorithm (Turing machines).

Part 1-1: What is an algorithm?

Definition (algorithm)

*A calculation rule for a problem is called **algorithm** if there is a Turing machine equivalent to this calculation rule which stops for every input that has a solution.*

To check whether a "calculation rule / program" is in fact an algorithm, we must design a Turing machine mimicking these rules.

This is beyond the scope of this course.

However, to establish algorithms and to analyze their costs, we need to have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs while still having a "correct" notion of algorithm (Turing machines).

There are other computer-models that are "equivalent" to Turing machines and that are closer to the notion of computers that we know. This "equivalence" allows us to use these models instead.

Part 1-1: What is an algorithm?

Definition (algorithm)

A calculation rule for a problem is called ***algorithm*** *if there is a Turing machine equivalent to this calculation rule which stops for every input that has a solution.*

To check whether a "calculation rule / program" is in fact an algorithm, we must design a Turing machine mimicking these rules.

This is beyond the scope of this course.

However, to establish algorithms and to analyze their costs, we need to have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs while still having a "correct" notion of algorithm (Turing machines).

There are other computer-models that are "equivalent" to Turing machines and that are closer to the notion of computers that we know. This "equivalence" allows us to use these models instead.

Whatever any computer can do, can be done by those models and thus, by a Turing machine !!

Part 1-1: What is an algorithm?

"Equivalent" to Turing machines are register machines. One of them are random-access machines (RAM): an abstract model of computers that is "closest" to the common notion of a computer and where instructions are executed one after another, with no concurrent operations.

Part 1-1: What is an algorithm?

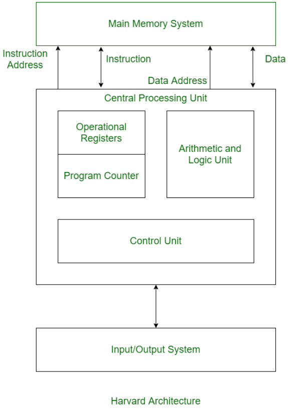
"Equivalent" to Turing machines are register machines. One of them are random-access machines (RAM): an abstract model of computers that is "closest" to the common notion of a computer and where instructions are executed one after another, with no concurrent operations.

(unlimited) memory $\mathrm{MEM}[0], \mathrm{MEM}[1], \mathrm{MEM}[2], \ldots$

fixed number of registers $R_1, \ldots, R_k$

[Registers are the memory locations that the CPU can access directly. The registers contain operands or the instructions that the processor is currently accessing.]

memory and registers store w-bit integers $n \in \{0, \ldots, 2^w - 1\}$



Harvard Architecture

Full Details about RAM in THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS; Aho et al. 1974

Part 1-1: What is an algorithm?

"Equivalent" to Turing machines are register machines. One of them are random-access machines (RAM): an abstract model of computers that is "closest" to the common notion of a computer and where instructions are executed one after another, with no concurrent operations.

(unlimited) memory $\mathrm{MEM}[0], \mathrm{MEM}[1], \mathrm{MEM}[2], \ldots$

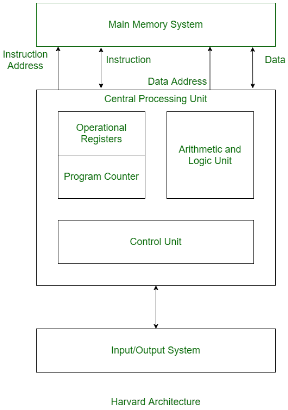
fixed number of registers $R_1, \ldots, R_k$

[Registers are the memory locations that the CPU can access directly. The registers contain operands or the instructions that the processor is currently accessing.]

memory and registers store w-bit integers $n \in \{0, \ldots, 2^w - 1\}$

instructions:

load/store $R_i = \mathrm{MEM}[j]$, $\mathrm{MEM}[j] = R_i$



Harvard Architecture

Full Details about RAM in THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS; Aho et al. 1974

Part 1-1: What is an algorithm?

"Equivalent" to Turing machines are register machines. One of them are random-access machines (RAM): an abstract model of computers that is "closest" to the common notion of a computer and where instructions are executed one after another, with no concurrent operations.

(unlimited) memory $\mathrm{MEM}[0], \mathrm{MEM}[1], \mathrm{MEM}[2], \ldots$

fixed number of registers $R_1, \ldots, R_k$

[Registers are the memory locations that the CPU can access directly. The registers contain operands or the instructions that the processor is currently accessing.]

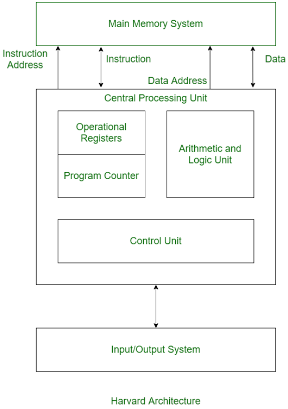
memory and registers store w-bit integers $n \in \{0, \ldots, 2^w - 1\}$

instructions:

load/store $R_i = \mathrm{MEM}[j]$, $\mathrm{MEM}[j] = R_i$

basic operations on registers:

$R_k = R_i + R_j$ (arithmetic is *modulo* 2^w!)
also $R_k = R_i - R_j$, $R_i * R_j$, $R_i \mathrm{div} R_j$, $R_i \mathrm{mod} R_j$
[these basic operations are "easy" to be implement on hardware]



Harvard Architecture

Full Details about RAM in THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS; Aho et al. 1974

Part 1-1: What is an algorithm?

"Equivalent" to Turing machines are register machines. One of them are random-access machines (RAM): an abstract model of computers that is "closest" to the common notion of a computer and where instructions are executed one after another, with no concurrent operations.

(unlimited) memory $\mathrm{MEM}[0], \mathrm{MEM}[1], \mathrm{MEM}[2], \ldots$

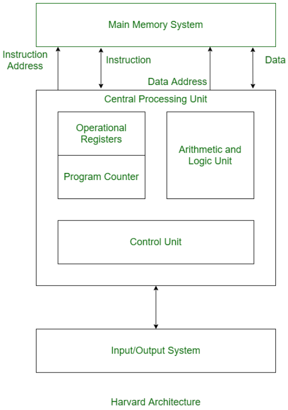
fixed number of registers $R_1, \ldots, R_k$

[Registers are the memory locations that the CPU can access directly. The registers contain operands or the instructions that the processor is currently accessing.]

memory and registers store w-bit integers $n \in \{0, \ldots, 2^w - 1\}$

instructions:

- load/store $R_i = \mathrm{MEM}[j]$, $\mathrm{MEM}[j] = R_i$
- basic operations on registers:
  - $R_k = R_i + R_j$ (arithmetic is *modulo* 2^w!)
  - also $R_k = R_i - R_j$, $R_i * R_j$, $R_i \mathrm{div} R_j$, $R_i \mathrm{mod} R_j$
  - *[these basic operations are "easy" to be implement on hardware]*
- conditional / unconditional jumps
  [algorithms are lines of instructions, jump back and forth to these lines]



Harvard Architecture

Full Details about RAM in THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS; Aho et al. 1974

Part 1-1: What is an algorithm?

"Equivalent" to Turing machines are register machines. One of them are random-access machines (RAM): an abstract model of computers that is "closest" to the common notion of a computer and where instructions are executed one after another, with no concurrent operations.

(unlimited) memory $\mathrm{MEM}[0], \mathrm{MEM}[1], \mathrm{MEM}[2], \ldots$

fixed number of registers $R_1, \ldots, R_k$

[Registers are the memory locations that the CPU can access directly. The registers contain operands or the instructions that the processor is currently accessing.]

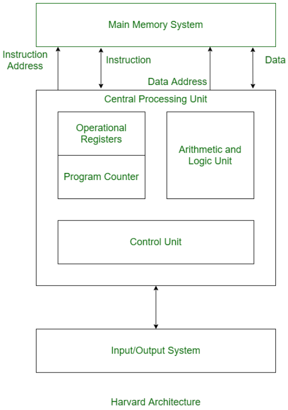
memory and registers store w-bit integers $n \in \{0, \ldots, 2^w - 1\}$

instructions:

- load/store $R_i = \mathrm{MEM}[j]$, $\mathrm{MEM}[j] = R_i$
- basic operations on registers:
  - $R_k = R_i + R_j$ (arithmetic is *modulo* 2^w!)
  - also $R_k = R_i - R_j$, $R_i * R_j$, $R_i \,\mathrm{div}\, R_j$, $R_i \,\mathrm{mod}\, R_j$
  - *[these basic operations are "easy" to be implement on hardware]*
- conditional / unconditional jumps
  [algorithms are lines of instructions, jump back and forth to these lines]

costs = number of executed step-by-step instructions (i.e., each instruction takes constant time)

Full Details about RAM in THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS; Aho et al. 1974



Harvard Architecture

Part 1-1: What is an algorithm?

"Equivalent" to Turing machines are register machines. One of them are random-access machines (RAM): an abstract model of computers that is "closest" to the common notion of a computer and where instructions are executed one after another, with no concurrent operations.

Already simplified, but typical RAM-code (here for computing $\sum_{i=1}^{n} i$)

```
READ n               # Read the value of n from input
SET R_sum = 0        # Initialize a register R_sum to store the sum
SET R_count = 1      # Initialize a register R_count to store the counter variable

LOOP_START:
    COMPARE R_count > n
    IF_TRUE jump to END_LOOP  # Check if R_count is greater than n

    # Add the current value of R_count to sum
    LOAD R_tmp, R_sum   # Load the current value of sum into a temporary register
    ADD R_tmp, R_count  # Add the value of counter variable to the temporary register "R_tmp += R_count"
    STORE R_sum, R_tmp  # Store the result back in the sum register

    # Increment the counter
    LOAD R_tmp2,  R_count     # Load the current value of R_count into another temporary register
    ADD R_tmp2, 1             # Increment the value in the temporary register "R_tmp2 += 1"
    STORE R_count, R_tmp2     # Store the result back in the R_count register

    jump to LOOP_START  # Go back to the beginning of the loop

END_LOOP:
PRINT R_sum            # Output the final sum
```

Example: Board

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

- . . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)
- . . . its input is a finite set of values
- . . . in every step only a finite amount of memory is used
- . . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,. . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n - 1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,. . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n - 1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,. . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n - 1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo, . . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n - 1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term "algorithm" and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,. . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n - 1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term "algorithm" and we provide here a "rough, approximate" definition:

A procedure is an algorithm if ...

...it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on ...)

...its input is a finite set of values

...in every step only a finite amount of memory is used

...has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,...
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n-1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,. . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n-1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,. . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n - 1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,. . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n - 1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,. . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n - 1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,. . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n - 1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (which is quite cumbersome!). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if ...

...it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on ...)

...its input is a finite set of values

...in every step only a finite amount of memory is used

...has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,...
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n-1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Part 1-1: What is an algorithm?

Classical ways to represent algorithms briefly explained on the "Sum-up" problem:

Compute $total\_sum = \sum_{i=1}^{n} i$ for a given integer n.

Verbal "*We define total_sum to be* 0 *and then add to total_sum the integer* 1*, then we add* 2*, . . . , then we add n.*"

for communication of ideas often sufficient, usually indicates only implicitly sequence of instructions
[must be careful here when it comes to checking costs!]

pseudocode

```
Sum(n)
  total_sum := 0
  FOR (i := 1 to n) DO
      total_sum := total_sum + i
  PRINT total_sum
```

Part 1-1: What is an algorithm?

Classical ways to represent algorithms briefly explained on the "Sum-up" problem:

Compute $total\_sum = \sum_{i=1}^{n} i$ for a given integer n.

Verbal "*We define total_sum to be* 0 *and then add to total_sum the integer* 1*, then we add* 2*, . . . , then we add n.*"

for communication of ideas often sufficient, usually indicates only implicitly sequence of instructions
[must be careful here when it comes to checking costs!]

pseudocode

```
Sum(n)
  total_sum := 0
  FOR (i := 1 to n) DO
      total_sum := total_sum + i
  PRINT total_sum
```

Part 1-1: What is an algorithm?

Classical ways to represent algorithms briefly explained on the "Sum-up" problem:

Compute $total\_sum = \sum_{i=1}^{n} i$ for a given integer n.

Verbal "*We define total_sum to be* 0 *and then add to total_sum the integer* 1*, then we add* 2*, ..., then we add n.*"

for communication of ideas often sufficient, usually indicates only implicitly sequence of instructions
[must be careful here when it comes to checking costs!]

pseudocode

```
Sum(n)
  total_sum := 0
  FOR (i := 1 to n) DO
      total_sum := total_sum + i
  PRINT total_sum
```

Part 1-1: What is an algorithm?

Classical ways to represent algorithms briefly explained on the "Sum-up" problem:

Compute $total\_sum = \sum_{i=1}^{n} i$ for a given integer n.

Verbal "*We define total_sum to be* 0 *and then add to total_sum the integer* 1*, then we add* 2*, . . . , then we add n.*"

for communication of ideas often sufficient, usually indicates only implicitly sequence of instructions
[must be careful here when it comes to checking costs!]

pseudocode

```
Sum(n)
  total_sum := 0
  FOR (i := 1 to n) DO
        total_sum := total_sum + i
  PRINT total_sum
```

The pseudo-code description is in a "free form", mixing English words and "programming-like" statements as long as it serves the purpose of conveying –without ambiguity– how our algorithm runs.

Part 1-1: What is an algorithm?

Classical ways to represent algorithms briefly explained on the "Sum-up" problem:
Compute $total\_sum = \sum_{i=1}^{n} i$ for a given integer n.

Verbal "*We define total_sum to be* 0 *and then add to total_sum the integer* 1, *then we add* 2, . . . , *then we add n.*"

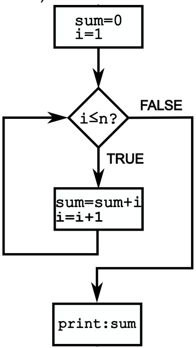
for communication of ideas often sufficient, usually indicates only implicitly sequence of instructions
[must be careful here when it comes to checking costs!]

pseudocode

```
Sum(n)
  total_sum := 0
  FOR (i := 1 to n) DO
        total_sum := total_sum + i
  PRINT total_sum
```

The pseudo-code description is in a "free form", mixing English words and "programming-like" statements as long as it serves the purpose of conveying –without ambiguity– how our algorithm runs.

diagram/flowchart (good to "see" the step-by-step instructions)



Part 1-1: What is an algorithm?

Classical ways to represent algorithms briefly explained on the "Sum-up" problem:

Compute $total\_sum = \sum_{i=1}^{n} i$ for a given integer n.

Verbal "*We define total_sum to be* 0 *and then add to total_sum the integer* 1, *then we add* 2, . . . , *then we add n.*"

for communication of ideas often sufficient, usually indicates only implicitly sequence of instructions
[must be careful here when it comes to checking costs!]

pseudocode

Sum(n)

$total\_sum := 0$
FOR ($i := 1$ to n) DO
$\quad total\_sum := total\_sum + i$
PRINT $total\_sum$

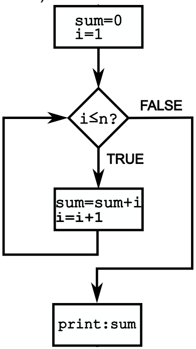
Some "real" programming language (here python)

```python
def sum_up_to_n(n):

 total_sum = 0
 for i in range(1, n + 1):
     total_sum += i

 print(f"The sum of integers from 1 to {n} is: {total_sum}")
```

diagram/flowchart (good to "see" the step-by-step instructions)



Part 1-2: Correctness of algorithms

Part 1-2: Correctness of algorithms

We are, in particular, interested in algorithms that solve problems:

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the previous "sum-up" problem are the n for a specific integer (e.g. $n = 3$)

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

[$\Longleftrightarrow$ an incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer]

Let's have a look to a specific problem, design an algorithm and show its correctness, i.e., it solves this problem.

Part 1-2: Correctness of algorithms

We are, in particular, interested in algorithms that solve problems:

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the previous "sum-up" problem are the n for a specific integer (e.g. $n = 3$)

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

[$\Longleftrightarrow$ an incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer]

Let's have a look to a specific problem, design an algorithm and show its correctness, i.e., it solves this problem.

Part 1-2: Correctness of algorithms

We are, in particular, interested in algorithms that solve problems:

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the previous "sum-up" problem are the n for a specific integer (e.g. $n = 3$)

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

[$\Longleftrightarrow$ an incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer]

Let's have a look to a specific problem, design an algorithm and show its correctness, i.e., it solves this problem.

Part 1-2: Correctness of algorithms

We are, in particular, interested in algorithms that solve problems:

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the previous "sum-up" problem are the n for a specific integer (e.g. $n = 3$)

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

[$\iff$ an incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer]

Let's have a look to a specific problem, design an algorithm and show its correctness, i.e., it solves this problem.

Part 1-2: Correctness of algorithms

We are, in particular, interested in algorithms that solve problems:

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the previous "sum-up" problem are the n for a specific integer (e.g. $n = 3$)

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

[$\Longleftrightarrow$ an incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer]

Let's have a look to a specific problem, design an algorithm and show its correctness, i.e., it solves this problem.

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

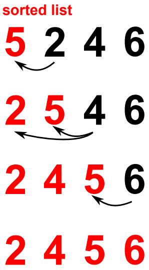
$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \dots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \dots, a'_n)$ such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$



A simple sorting "algorithm" idea:

We assume to have an order list (highlighted in red)

Then, subsequently insert the next element x into this sorted list by comparing x with the elements in sorted list from right to left

We put this into an algorithm, known as `Insertion_Sort`.

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 ... j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 ... j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j					
i					
A		5	2	4	6

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \dots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \dots, a'_n)$ such that $a'_1 \le a'_2 \le \dots \le a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j			×		
i					
A		5	2	4	6

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 ... j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j			×		
i					
A		5	2	4	6

$key = 2$

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 ... j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j			×		
i		×			
A		5	2	4	6

$key = 2$

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j			×		
i		×			
A		5	2	4	6

$A[0] = 5 > key = 2$

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j			×		
i		×			
A		5	5	4	6
			2		

$A[0] = 5 > key = 2$
$A[1] := A[0] = 5$

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 ... j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j			×		
i	×				
A		5	5	4	6
			2		

$A[0] = 5 > key = 2$
$A[1] := A[0] = 5$
$i := 0 - 1 = -1$

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 ... j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j			×		
i	×				
A		5	5	4	6
			2		

$A[0] = 5 > key = 2$
$A[1] := A[0] = 5$
$i := 0 - 1 = -1$

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \dots, a_n)$

Goal: A re-ordering $(a'_1, a'_2 \dots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j			×		
i	×				
A		2	5	4	6
			2		

$A[0] = 5 > key = 2$
$A[1] := A[0] = 5$
$i := 0 - 1 = -1$
$A[0] := key = 2$

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j				×	
i	×				
A		2	5	4	6

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j				×	
i	×				
A		2	5	4	6

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \dots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \dots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j				×	
i			×		
A		2	5	4	6

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j				×	
i			×		
A		2	5	4	6

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j				×	
i			×		
A		2	5	5	6
				4	

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 ... j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j				×	
i		×			
A		2	5	5	6
				4	

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j				×	
i					
A		2	5	5	6
				4	

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 ... j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j				×	
i					
A		2	4	5	6
				4	

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j					×
i					
A		2	4	5	6

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j					×
i					
A		2	4	5	6

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 ... j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j					×
i				×	
A		2	4	5	6

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \dots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \dots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//*A* is array of size *n* containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j					×
i				×	
A		2	4	5	6

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j					×
i				×	
A		2	4	5	6

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j					×
i				×	
A		2	4	5	6

Part 1-2: Correctness of algorithms

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$

//A is array of size n containing the integers, 1st entry A[0]

```
Insertion_Sort(A)
FOR (j := 1 to n − 1) DO
    key := A[j]
    //insert A[j] into the sorted sequence A[0 . . . j − 1]
    i := j − 1
    WHILE (i ≥ 0 and A[i] > key)
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := key
RETURN Sorted array A
```

key					
index	-1	0	1	2	3
j					×
i				×	
A		2	4	5	6

This shows that `Insertion_Sort` correctly works precisely for the input sequence $(5, 2, 4, 6)$.

Part 1-2: Correctness of algorithms

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the sorting-problem are all finite sequences of integers, e.g. $(1, 2, 3)$, $(1, 1, 1, \ldots, 1)$, $(5, 4, 5, 3)$, $\ldots$

An algorithm is **correct** or **solves** the problem if, for every input instance, it halts with the correct output w.r.t. the given problem.

So-far, we showed that `Insertion_Sort` correctly works only for the specific instance $(5, 2, 4, 6)$.

Let's proof its correctness, i.e., we show that `Insertion_Sort` terminates and correctly returns a sorted list for any finite input sequence of integers.

Part 1-2: Correctness of algorithms

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the sorting-problem are all finite sequences of integers, e.g. $(1, 2, 3)$, $(1, 1, 1, \ldots, 1)$, $(5, 4, 5, 3)$, $\ldots$

An algorithm is **correct** or **solves** the problem if, for every input instance, it halts with the correct output w.r.t. the given problem.

So-far, we showed that `Insertion_Sort` correctly works only for the specific instance $(5, 2, 4, 6)$.

Let's proof its correctness, i.e., we show that `Insertion_Sort` terminates and correctly returns a sorted list for any finite input sequence of integers.

Part 1-2: Correctness of algorithms

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the sorting-problem are all finite sequences of integers, e.g. $(1, 2, 3)$, $(1, 1, 1, \ldots, 1)$, $(5, 4, 5, 3)$, $\ldots$

An algorithm is **correct** or **solves** the problem if, for every input instance, it halts with the correct output w.r.t. the given problem.

So-far, we showed that `Insertion_Sort` correctly works only for the specific instance $(5, 2, 4, 6)$.

Let's proof its correctness, i.e., we show that `Insertion_Sort` terminates and correctly returns a sorted list for any finite input sequence of integers.

Part 1-2: Correctness of algorithms

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the sorting-problem are all finite sequences of integers, e.g. $(1, 2, 3)$, $(1, 1, 1, \ldots, 1)$, $(5, 4, 5, 3)$, $\ldots$

An algorithm is **correct** or **solves** the problem if, for every input instance, it halts with the correct output w.r.t. the given problem.

So-far, we showed that `Insertion_Sort` correctly works only for the specific instance $(5, 2, 4, 6)$.

Let's proof its correctness, i.e., we show that `Insertion_Sort` terminates and correctly returns a sorted list for any finite input sequence of integers.

Part 1-2: Correctness of algorithms

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the sorting-problem are all finite sequences of integers, e.g. $(1, 2, 3)$, $(1, 1, 1, \dots, 1)$, $(5, 4, 5, 3)$, $\dots$

An algorithm is **correct** or **solves** the problem if, for every input instance, it halts with the correct output w.r.t. the given problem.

So-far, we showed that `Insertion_Sort` correctly works only for the specific instance $(5, 2, 4, 6)$.

Let's proof its correctness, i.e., we show that `Insertion_Sort` terminates and correctly returns a sorted list for any finite input sequence of integers.

Part 1-2: Correctness of algorithms

Theorem 1

`Insertion_Sort` correctly sorts a given finite sequence A of integers.

Proof.

board via loop-invariants □

Part 1: Summary up to here

We have discussed so-far the topics

1-1 What is an algorithm?

We shortly explained that TMs and equivalent models (RAM) are used to define the term algorithm

Deeper details are beyond the scope of this course. Instead, we provided here a "rough, approximate" definition:

A procedure is an algorithm if ...

... it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on ...)
... its input is a finite set of values
... in every step only a finite amount of memory is used
... has some finite set of values as output (in case it terminates)

This finite linear sequence of instructions can be written using e.g. pseudo-code or flowcharts

1-2 Correctness of algorithms

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

Parts **1-1** and **1-2** have been examined on a sorting algorithm `Insertion_Sort`.

Let's now continue with Part **1-3** "Runtime"

Part 1: Summary up to here

We have discussed so-far the topics

1-1 What is an algorithm?

We shortly explained that TMs and equivalent models (RAM) are used to define the term algorithm

Deeper details are beyond the scope of this course. Instead, we provided here a "rough, approximate" definition:

A procedure is an algorithm if ...

... it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on ...)
... its input is a finite set of values
... in every step only a finite amount of memory is used
... has some finite set of values as output (in case it terminates)

This finite linear sequence of instructions can be written using e.g. pseudo-code or flowcharts

1-2 Correctness of algorithms

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

Parts **1-1** and **1-2** have been examined on a sorting algorithm `Insertion_Sort`.

Let's now continue with Part **1-3** "Runtime"

Part 1: Summary up to here

We have discussed so-far the topics

1-1 What is an algorithm?

We shortly explained that TMs and equivalent models (RAM) are used to define the term algorithm

Deeper details are beyond the scope of this course. Instead, we provided here a "rough, approximate" definition:

A procedure is an algorithm if . . .

- . . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)
- . . . its input is a finite set of values
- . . . in every step only a finite amount of memory is used
- . . . has some finite set of values as output (in case it terminates)

This finite linear sequence of instructions can be written using e.g. pseudo-code or flowcharts

1-2 Correctness of algorithms

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

Parts **1-1** and **1-2** have been examined on a sorting algorithm `Insertion_Sort`.

Let's now continue with Part **1-3** "Runtime"

Part 1: Summary up to here

We have discussed so-far the topics

1-1 What is an algorithm?

We shortly explained that TMs and equivalent models (RAM) are used to define the term algorithm

Deeper details are beyond the scope of this course. Instead, we provided here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)
. . . its input is a finite set of values
. . . in every step only a finite amount of memory is used
. . . has some finite set of values as output (in case it terminates)

This finite linear sequence of instructions can be written using e.g. pseudo-code or flowcharts

1-2 Correctness of algorithms

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

Parts **1-1** and **1-2** have been examined on a sorting algorithm `Insertion_Sort`.

Let's now continue with Part **1-3** "Runtime"

Part 1: Summary up to here

We have discussed so-far the topics

1-1 What is an algorithm?

We shortly explained that TMs and equivalent models (RAM) are used to define the term algorithm

Deeper details are beyond the scope of this course. Instead, we provided here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)
. . . its input is a finite set of values
. . . in every step only a finite amount of memory is used
. . . has some finite set of values as output (in case it terminates)

This finite linear sequence of instructions can be written using e.g. pseudo-code or flowcharts

1-2 Correctness of algorithms

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

Parts **1-1** and **1-2** have been examined on a sorting algorithm `Insertion_Sort`.

Let's now continue with Part **1-3** "Runtime"

Part 1: Summary up to here

We have discussed so-far the topics

1-1 What is an algorithm?

We shortly explained that TMs and equivalent models (RAM) are used to define the term algorithm

Deeper details are beyond the scope of this course. Instead, we provided here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)
. . . its input is a finite set of values
. . . in every step only a finite amount of memory is used
. . . has some finite set of values as output (in case it terminates)

This finite linear sequence of instructions can be written using e.g. pseudo-code or flowcharts

1-2 Correctness of algorithms

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

Parts **1-1** and **1-2** have been examined on a sorting algorithm `Insertion_Sort`.

Let's now continue with Part **1-3** "Runtime"

Part 1: Summary up to here

We have discussed so-far the topics

1-1 What is an algorithm?

We shortly explained that TMs and equivalent models (RAM) are used to define the term algorithm

Deeper details are beyond the scope of this course. Instead, we provided here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)
. . . its input is a finite set of values
. . . in every step only a finite amount of memory is used
. . . has some finite set of values as output (in case it terminates)

This finite linear sequence of instructions can be written using e.g. pseudo-code or flowcharts

1-2 Correctness of algorithms

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

Parts **1-1** and **1-2** have been examined on a sorting algorithm `Insertion_Sort`.

Let's now continue with Part **1-3** "Runtime"

Part 1: Summary up to here

We have discussed so-far the topics

1-1 What is an algorithm?

We shortly explained that TMs and equivalent models (RAM) are used to define the term algorithm

Deeper details are beyond the scope of this course. Instead, we provided here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)
. . . its input is a finite set of values
. . . in every step only a finite amount of memory is used
. . . has some finite set of values as output (in case it terminates)

This finite linear sequence of instructions can be written using e.g. pseudo-code or flowcharts

1-2 Correctness of algorithms

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

Parts **1-1** and **1-2** have been examined on a sorting algorithm `Insertion_Sort`.

Let's now continue with Part **1-3** "Runtime"

Part 1: Summary up to here

We have discussed so-far the topics

1-1 What is an algorithm?

We shortly explained that TMs and equivalent models (RAM) are used to define the term algorithm

Deeper details are beyond the scope of this course. Instead, we provided here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)
. . . its input is a finite set of values
. . . in every step only a finite amount of memory is used
. . . has some finite set of values as output (in case it terminates)

This finite linear sequence of instructions can be written using e.g. pseudo-code or flowcharts

1-2 Correctness of algorithms

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

Parts **1-1** and **1-2** have been examined on a sorting algorithm `Insertion_Sort`.

Let's now continue with Part **1-3** "Runtime"

Part 1-3: Runtime of algorithms

Part 1-3: Runtime of algorithms

Naive idea: measure the time from start to end in (milli)seconds

say we want to know for some input N how fast the algorithm is:

$N = 4000$ and runtime 6.3 seconds
$N = 8000$ and runtime 51.1 seconds
$N = 16000$ and runtime 410.8 seconds

Hypothesis: For arbitrary N runtime is $\sim 10^{-10}N^3$

not really comparable since this can differ on distinct computers.
we need a notation to classify "runtime" that is independent on the "performance" of computer.

Time complexity

- NOT: measure runtime on a specific computer
- BUT: determine effort for idealized computer model (e.g. RAM-model)
- We need abstract measure for time complexity to estimate asymptotic costs that depends on the size of the input

Part 1-3: Runtime of algorithms

Naive idea: measure the time from start to end in (milli)seconds

say we want to know for some input N how fast the algorithm is:

$N = 4000$ and runtime 6.3 seconds
$N = 8000$ and runtime 51.1 seconds
$N = 16000$ and runtime 410.8 seconds

Hypothesis: For arbitrary N runtime is $\sim 10^{-10} N^3$

not really comparable since this can differ on distinct computers.
we need a notation to classify “runtime” that is independent on the "performance" of computer.

Time complexity

- NOT: measure runtime on a specific computer
- BUT: determine effort for idealized computer model (e.g. RAM-model)
- We need abstract measure for time complexity to estimate asymptotic costs that depends on the size of the input

Part 1-3: Runtime of algorithms

Naive idea: measure the time from start to end in (milli)seconds

say we want to know for some input N how fast the algorithm is:

$N = 4000$ and runtime 6.3 seconds
$N = 8000$ and runtime 51.1 seconds
$N = 16000$ and runtime 410.8 seconds

Hypothesis: For arbitrary N runtime is $\sim 10^{-10}N^3$

not really comparable since this can differ on distinct computers.
we need a notation to classify “runtime” that is independent on the "performance" of computer.

Time complexity

NOT: measure runtime on a specific computer

BUT: determine effort for idealized computer model (e.g. RAM-model)

We need abstract measure for time complexity to estimate asymptotic costs that depends on the size of the input

Part 1-3: Runtime of algorithms

Naive idea: measure the time from start to end in (milli)seconds

say we want to know for some input N how fast the algorithm is:

$N = 4000$ and runtime 6.3 seconds
$N = 8000$ and runtime 51.1 seconds
$N = 16000$ and runtime 410.8 seconds

Hypothesis: For arbitrary N runtime is $\sim 10^{-10} N^3$

not really comparable since this can differ on distinct computers.
we need a notation to classify “runtime” that is independent on the "performance" of computer.

Time complexity

NOT: measure runtime on a specific computer

BUT: determine effort for idealized computer model (e.g. RAM-model)

We need abstract measure for time complexity to estimate asymptotic costs that depends on the size of the input

Add two numbers

	1	1	1	3	7
+		2	8	3	4
=	1	3	9	7	1

Takes 5 single additions.

Hence, addition needs $\max\{m, n\}$ operations (even slightly more if we consider "carryover") for two numbers having m, resp., n digits.

There two main types of cost models:

- the unit-cost model assigns a constant cost to every machine operation, regardless of the size of the numbers involved.
- the logarithmic-cost model, assigns a cost to every machine operation proportional to the number of bits involved [Integer $n \in \{0, \ldots, 2^w - 1\}$ needs w bits to be stored]
  not used in this course, however, important e.g. in cryptography

In this course **unit-cost model**:
The RAM-model contains instructions commonly found in real computers:

- arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling),
- data movement (load, store, copy), and
- control (conditional and unconditional branch, subroutine call and return).

Each such instruction is counted as one time-unit and thus, takes a constant amount of time.

Hence, we essentially count the number of execution of instructions (as the number of operations)

Part 1-3: Runtime of algorithms

Add two numbers

	1	1	1	3	7
+		2	8	3	4
=	1	3	9	7	1

Takes 5 single additions.

Hence, addition needs $\max\{m, n\}$ operations (even slightly more if we consider "carryover") for two numbers having m, resp., n digits.

There two main types of cost models:

- the unit-cost model assigns a constant cost to every machine operation, regardless of the size of the numbers involved.
- the logarithmic-cost model, assigns a cost to every machine operation proportional to the number of bits involved [Integer $n \in \{0, \ldots, 2^w - 1\}$ needs w bits to be stored]
  not used in this course, however, important e.g. in cryptography

In this course **unit-cost model**:
The RAM-model contains instructions commonly found in real computers:

- arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling),
- data movement (load, store, copy), and
- control (conditional and unconditional branch, subroutine call and return).

Each such instruction is counted as one time-unit and thus, takes a constant amount of time.

Hence, we essentially count the number of execution of instructions (as the number of operations)

Add two numbers

	1	1	1	3	7
+		2	8	3	4
=	1	3	9	7	1

Takes 5 single additions.

Hence, addition needs $\max\{m, n\}$ operations (even slightly more if we consider "carryover") for two numbers having m, resp., n digits.

There two main types of cost models:

- the unit-cost model assigns a constant cost to every machine operation, regardless of the size of the numbers involved.
- the logarithmic-cost model, assigns a cost to every machine operation proportional to the number of bits involved [Integer $n \in \{0, \ldots, 2^w - 1\}$ needs w bits to be stored]
  not used in this course, however, important e.g. in cryptography

In this course **unit-cost model**:
The RAM-model contains instructions commonly found in real computers:

- arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling),
- data movement (load, store, copy), and
- control (conditional and unconditional branch, subroutine call and return).

Each such instruction is counted as one time-unit and thus, takes a constant amount of time.

Hence, we essentially count the number of execution of instructions (as the number of operations)

Part 1-3: Runtime of algorithms

Add two numbers

	1	1	1	3	7
+		2	8	3	4
=	1	3	9	7	1

Takes 5 single additions.

Hence, addition needs $\max\{m, n\}$ operations (even slightly more if we consider "carryover") for two numbers having m, resp., n digits.

There two main types of cost models:

- the unit-cost model assigns a constant cost to every machine operation, regardless of the size of the numbers involved.
- the logarithmic-cost model, assigns a cost to every machine operation proportional to the number of bits involved [Integer $n \in \{0, \ldots, 2^w - 1\}$ needs w bits to be stored]
  not used in this course, however, important e.g. in cryptography

In this course **unit-cost model**:
The RAM-model contains instructions commonly found in real computers:

- arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling),
- data movement (load, store, copy), and
- control (conditional and unconditional branch, subroutine call and return).

Each such instruction is counted as one time-unit and thus, takes a constant amount of time.

Hence, we essentially count the number of execution of instructions (as the number of operations)

Part 1-3: Runtime of algorithms

Add two numbers

	1	1	1	3	7
+		2	8	3	4
=	1	3	9	7	1

Takes 5 single additions.

Hence, addition needs $\max\{m, n\}$ operations (even slightly more if we consider "carryover") for two numbers having m, resp., n digits.

There two main types of cost models:

- the unit-cost model assigns a constant cost to every machine operation, regardless of the size of the numbers involved.
- the logarithmic-cost model, assigns a cost to every machine operation proportional to the number of bits involved [Integer $n \in \{0, \ldots, 2^w - 1\}$ needs w bits to be stored]
  not used in this course, however, important e.g. in cryptography

In this course **unit-cost model**:
The RAM-model contains instructions commonly found in real computers:

- arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling),
- data movement (load, store, copy), and
- control (conditional and unconditional branch, subroutine call and return).

Each such instruction is counted as one time-unit and thus, takes a constant amount of time.

Hence, we essentially count the number of execution of instructions (as the number of operations)

Part 1-3: Runtime of algorithms

Add two numbers

	1	1	1	3	7
+		2	8	3	4
=	1	3	9	7	1

Takes 5 single additions.

Hence, addition needs $\max\{m, n\}$ operations (even slightly more if we consider "carryover") for two numbers having m, resp., n digits.

There two main types of cost models:

- the unit-cost model assigns a constant cost to every machine operation, regardless of the size of the numbers involved.
- the logarithmic-cost model, assigns a cost to every machine operation proportional to the number of bits involved [Integer $n \in \{0, \ldots, 2^w - 1\}$ needs w bits to be stored]
  not used in this course, however, important e.g. in cryptography

In this course **unit-cost model**:
The RAM-model contains instructions commonly found in real computers:

- arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling),
- data movement (load, store, copy), and
- control (conditional and unconditional branch, subroutine call and return).

Each such instruction is counted as one time-unit and thus, takes a constant amount of time.

Hence, we essentially count the number of execution of instructions (as the number of operations)

Part 1-3: Runtime of algorithms

We denote with $T(|I|)$ the runtime of an algorithm with input I. Here, $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A$ with n entries: $|I| = n$

```
Count_Zeros(array A)
    int i, count
    count := 0
    FOR(i := 0 to n − 1)
        IF(A[i] == 0) DO count++
```

variable declaration (e.g. `int` i, $count$):	2
assignment statement (e.g. $i := 0$):	2
increment (i and $count$)	$n + n$
compare "$A[i] == 0$"	n
Σsingle instructions $= T(n) =$	$3n + 4$

Still, this is unsatisfying, e.g. if you have $T(n) = 3n + 4$ vs $T'(n) = 4n$ (which is faster?)

Part 1-3: Runtime of algorithms

We denote with $T(|I|)$ the runtime of an algorithm with input I. Here, $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A$ with n entries: $|I| = n$

```
Count_Zeros(array A)
    int i, count
    count := 0
    FOR(i := 0 to n − 1)
        IF(A[i] == 0) DO count++
```

variable declaration (e.g. `int` i, *count*):	2
assignment statement (e.g. $i := 0$):	2
increment (i and *count*)	$n + n$
compare "$A[i] == 0$"	n
Σsingle instructions $= T(n) =$	$3n + 4$

Still, this is unsatisfying, e.g. if you have $T(n) = 3n + 4$ vs $T'(n) = 4n$ (which is faster?)

Part 1-3: Runtime of algorithms

We denote with $T(|I|)$ the runtime of an algorithm with input I. Here, $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A$ with n entries: $|I| = n$

```
Count_Zeros(array A)
    int i, count
    count := 0
    FOR(i := 0 to n − 1)
        IF(A[i] == 0) DO count++
```

variable declaration (e.g. `int` i, $count$):	2
assignment statement (e.g. $i := 0$):	2
increment (i and $count$)	$n + n$
compare "$A[i] == 0$"	n
Σsingle instructions $= T(n) =$	$3n + 4$

Still, this is unsatisfying, e.g. if you have $T(n) = 3n + 4$ vs $T'(n) = 4n$ (which is faster?)

Part 1-3: Runtime of algorithms

We denote with $T(|I|)$ the runtime of an algorithm with input I. Here, $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A$ with n entries: $|I| = n$

```
Count_Zeros(array A)
    int i, count
    count := 0
    FOR(i := 0 to n − 1)
        IF(A[i] == 0) DO count + +
```

variable declaration (e.g. int i, $count$):	2
assignment statement (e.g. $i := 0$):	2
increment (i and $count$)	$n + n$
compare "$A[i] == 0$"	n
Σsingle instructions $= T(n) =$	$3n + 4$

Still, this is unsatisfying, e.g. if you have $T(n) = 3n + 4$ vs $T'(n) = 4n$ (which is faster?)

Part 1-3: Runtime of algorithms

We denote with $T(|I|)$ the runtime of an algorithm with input I. Here, $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A$ with n entries: $|I| = n$

```
Count_Zeros(array A)
    int i, count
    count := 0
    FOR(i := 0 to n − 1)
        IF(A[i] == 0) DO count++
```

variable declaration (e.g. `int` i, *count*):	2
assignment statement (e.g. $i := 0$):	2
increment (i and *count*)	$n + n$
compare "$A[i] == 0$"	n
Σsingle instructions $= T(n) =$	$3n + 4$

Still, this is unsatisfying, e.g. if you have $T(n) = 3n + 4$ vs $T'(n) = 4n$ (which is faster?)

For $n = 1, 2, 3, 4$ we have $T(n) \geq T'(n)$ and $T(n) < T'(n)$ for $n > 5$

Part 1-3: Runtime of algorithms

$T(|I|)$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

```
Insertion_Sort(A)
1 FOR (j = 1 to n − 1) DO
2     key = A[j]
3     i := j − 1
4     WHILE (i ≥ 0 and A[i] > key)
5         A[i + 1] := A[i]
6         i := i − 1
7     A[i + 1] := key
8 RETURN Sorted array A
```

T_{line_nr}

1 $n-1$

2 $n-1$

3 $n-1$

4 $2\sum_{j=1}^{n-1} t_j$ with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j

5 $\sum_{j=1}^{n-1}(t_j - 1)$ here: $(t_j - 1)$ since "last" test of while-loop-test

6 $\sum_{j=1}^{n-1}(t_j - 1)$ stops this loop and we have no extra run

7 $n-1$

8 1

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j - 1) + 1$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j - 1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5$ (*linear runtime*)

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \ldots j-1]$.
Then, body of while-loop executed $j - 1$ times + one extra test to terminate while-loop $\implies t_j = j$ for all j.

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$$

$$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$$

$$= 4(n-1) + 4((n^2 - n)/2) - 2(n-1) + 1 = 2n^2 - 1 \text{ (quadratic runtime)}$$

Part 1-3: Runtime of algorithms

$T(|I|)$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

```
Insertion_Sort(A)
1 FOR (j = 1 to n − 1) DO
2     key = A[j]
3     i := j − 1
4     WHILE (i ≥ 0 and A[i] > key)
5         A[i + 1] := A[i]
6         i := i − 1
7     A[i + 1] := key
8 RETURN Sorted array A
```

	T_{line_nr}
1	$n-1$
2	$n-1$
3	$n-1$
4	$2\sum_{j=1}^{n-1} t_j$ with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j
5	$\sum_{j=1}^{n-1}(t_j-1)$ here: (t_j-1) since "last" test of while-loop-test
6	$\sum_{j=1}^{n-1}(t_j-1)$ stops this loop and we have no extra run
7	$n-1$
8	1

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j-1) + 1$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j-1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5$ (*linear runtime*)

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \dots j-1]$. Then, body of while-loop executed $j-1$ times + one extra test to terminate while-loop $\implies t_j = j$ for all j.

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$$

$$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$$

$$= 4(n-1) + 4((n^2-n)/2) - 2(n-1) + 1 = 2n^2 - 1 \text{ (quadratic runtime)}$$

Part 1-3: Runtime of algorithms

$T(|I|))$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

```
Insertion_Sort(A)
1 FOR (j = 1 to n − 1) DO
2     key = A[j]
3     i := j − 1
4     WHILE (i ≥ 0 and A[i] > key)
5         A[i + 1] := A[i]
6         i := i − 1
7     A[i + 1] := key
8 RETURN Sorted array A
```

T_{line_nr}

1 $n-1$

2 $n-1$

3 $n-1$

4 $2\sum_{j=1}^{n-1} t_j$ with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j

5 $\sum_{j=1}^{n-1}(t_j - 1)$ here: $(t_j - 1)$ since "last" test of while-loop-test

6 $\sum_{j=1}^{n-1}(t_j - 1)$ stops this loop and we have no extra run

7 $n-1$

8 1

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j - 1) + 1$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j - 1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5$ (*linear runtime*)

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \ldots j-1]$. Then, body of while-loop executed $j - 1$ times + one extra test to terminate while-loop $\Longrightarrow$ $t_j = j$ for all j.

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$$

$$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$$

$$= 4(n-1) + 4((n^2 - n)/2) - 2(n-1) + 1 = 2n^2 - 1$$ (*quadratic runtime*)

Part 1-3: Runtime of algorithms

$T(|I|)$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

```
Insertion_Sort(A)
1 FOR (j = 1 to n − 1) DO
2     key = A[j]
3     i := j − 1
4     WHILE (i ≥ 0 and A[i] > key)
5         A[i + 1] := A[i]
6         i := i − 1
7     A[i + 1] := key
8 RETURN Sorted array A
```

	T_{line_nr}	
1	$n-1$	
2	$n-1$	
3	$n-1$	
4	$2\sum_{j=1}^{n-1} t_j$	with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j
5	$\sum_{j=1}^{n-1}(t_j - 1)$	here: $(t_j - 1)$ since "last" test of while-loop-test
6	$\sum_{j=1}^{n-1}(t_j - 1)$	stops this loop and we have no extra run
7	$n-1$	
8	1	

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j - 1) + 1$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j - 1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5$ (*linear runtime*)

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \ldots j-1]$.
Then, body of while-loop executed $j - 1$ times + one extra test to terminate while-loop $\implies t_j = j$ for all j.

$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$

$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$

$= 4(n-1) + 4((n^2 - n)/2) - 2(n-1) + 1 = 2n^2 - 1$ (*quadratic runtime*)

Part 1-3: Runtime of algorithms

$T(|I|)$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

```
Insertion_Sort(A)
1 FOR (j = 1 to n − 1) DO
2     key = A[j]
3     i := j − 1
4     WHILE (i ≥ 0 and A[i] > key)
5         A[i + 1] := A[i]
6         i := i − 1
7     A[i + 1] := key
8 RETURN Sorted array A
```

	T_{line_nr}	
1	$n-1$	
2	$n-1$	
3	$n-1$	
4	$2\sum_{j=1}^{n-1} t_j$	with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j
5	$\sum_{j=1}^{n-1}(t_j-1)$	here: (t_j-1) since "last" test of while-loop-test
6	$\sum_{j=1}^{n-1}(t_j-1)$	stops this loop and we have no extra run
7	$n-1$	
8	1	

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j-1) + 1$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j-1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5 \text{ (\textit{linear runtime})}$$

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \ldots j-1]$. Then, body of while-loop executed $j-1$ times + one extra test to terminate while-loop $\Longrightarrow t_j = j$ for all j.

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$$

$$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$$

$$= 4(n-1) + 4((n^2-n)/2) - 2(n-1) + 1 = 2n^2 - 1 \text{ (\textit{quadratic runtime})}$$

Part 1-3: Runtime of algorithms

$T(|I|)$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

```
Insertion_Sort(A)
1 FOR (j = 1 to n − 1) DO
2     key = A[j]
3     i := j − 1
4     WHILE (i ≥ 0 and A[i] > key)
5         A[i + 1] := A[i]
6         i := i − 1
7     A[i + 1] := key
8 RETURN Sorted array A
```

	T_{line_nr}	
1	$n-1$	
2	$n-1$	
3	$n-1$	
4	$2\sum_{j=1}^{n-1} t_j$	with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j
5	$\sum_{j=1}^{n-1}(t_j - 1)$	here: $(t_j - 1)$ since "last" test of while-loop-test
6	$\sum_{j=1}^{n-1}(t_j - 1)$	stops this loop and we have no extra run
7	$n-1$	
8	1	

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j - 1) + 1$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j - 1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5 \text{ (\textit{linear runtime})}$$

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \ldots j-1]$.
Then, body of while-loop executed $j - 1$ times + one extra test to terminate while-loop $\implies t_j = j$ for all j.

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$$

$$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$$

$$= 4(n-1) + 4((n^2 - n)/2) - 2(n-1) + 1 = 2n^2 - 1 \text{ (\textit{quadratic runtime})}$$

Part 1-3: Runtime of algorithms

$T(|I|)$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

```
Insertion_Sort(A)
1 FOR (j = 1 to n − 1) DO
2     key = A[j]
3     i := j − 1
4     WHILE (i ≥ 0 and A[i] > key)
5         A[i + 1] := A[i]
6         i := i − 1
7     A[i + 1] := key
8 RETURN Sorted array A
```

	T_{line_nr}	
1	$n-1$	
2	$n-1$	
3	$n-1$	
4	$2\sum_{j=1}^{n-1} t_j$	with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j
5	$\sum_{j=1}^{n-1}(t_j - 1)$	here: $(t_j - 1)$ since "last" test of while-loop-test
6	$\sum_{j=1}^{n-1}(t_j - 1)$	stops this loop and we have no extra run
7	$n-1$	
8	1	

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j - 1) + 1$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j - 1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5$ (*linear runtime*)

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \ldots j-1]$. Then, body of while-loop executed $j - 1$ times + one extra test to terminate while-loop $\Longrightarrow t_j = j$ for all j.

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$$

$$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$$

$$= 4(n-1) + 4((n^2 - n)/2) - 2(n-1) + 1 = 2n^2 - 1 \text{ (quadratic runtime)}$$

Part 1-3: Runtime of algorithms

$T(|I|)$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

```
Insertion_Sort(A)
1 FOR (j = 1 to n − 1) DO
2     key = A[j]
3     i := j − 1
4     WHILE (i ≥ 0 and A[i] > key)
5         A[i + 1] := A[i]
6         i := i − 1
7     A[i + 1] := key
8 RETURN Sorted array A
```

	T_{line_nr}	
1	$n-1$	
2	$n-1$	
3	$n-1$	
4	$2\sum_{j=1}^{n-1} t_j$	with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j
5	$\sum_{j=1}^{n-1}(t_j - 1)$	here: $(t_j - 1)$ since "last" test of while-loop-test
6	$\sum_{j=1}^{n-1}(t_j - 1)$	stops this loop and we have no extra run
7	$n-1$	
8	1	

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j - 1) + 1$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j - 1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5 \text{ (\textit{linear runtime})}$$

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \ldots j-1]$.
Then, body of while-loop executed $j - 1$ times + one extra test to terminate while-loop $\Longrightarrow$ $t_j = j$ for all j.

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$$

$$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$$

$$= 4(n-1) + 4((n^2 - n)/2) - 2(n-1) + 1 = 2n^2 - 1 \text{ (\textit{quadratic runtime})}$$

Part 1-3: Runtime of algorithms

$T(|I|)$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

```
Insertion_Sort(A)
1 FOR (j = 1 to n − 1) DO
2     key = A[j]
3     i := j − 1
4     WHILE (i ≥ 0 and A[i] > key)
5         A[i + 1] := A[i]
6         i := i − 1
7     A[i + 1] := key
8 RETURN Sorted array A
```

	T_{line_nr}
1	$n-1$
2	$n-1$
3	$n-1$
4	$2\sum_{j=1}^{n-1} t_j$ with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j
5	$\sum_{j=1}^{n-1}(t_j - 1)$ here: $(t_j - 1)$ since "last" test of while-loop-test
6	$\sum_{j=1}^{n-1}(t_j - 1)$ stops this loop and we have no extra run
7	$n-1$
8	1

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j - 1) + 1$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j-1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5$ (*linear runtime*)

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \dots j-1]$. Then, body of while-loop executed $j-1$ times + one extra test to terminate while-loop $\Longrightarrow$ $t_j = j$ for all j.

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$$

$$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$$

$$= 4(n-1) + 4((n^2 - n)/2) - 2(n-1) + 1 = 2n^2 - 1 \text{ (quadratic runtime)}$$

Part 1-3: Runtime of algorithms

$T(|I|)$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

`Insertion_Sort(A)`

```
1 FOR (j = 1 to n − 1) DO
2     key = A[j]
3     i := j − 1
4     WHILE (i ≥ 0 and A[i] > key)
5         A[i + 1] := A[i]
6         i := i − 1
7     A[i + 1] := key
8 RETURN Sorted array A
```

T_{line_nr}

1 $n-1$

2 $n-1$

3 $n-1$

4 $2\sum_{j=1}^{n-1} t_j$ with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j

5 $\sum_{j=1}^{n-1}(t_j - 1)$ here: $(t_j - 1)$ since "last" test of while-loop-test

6 $\sum_{j=1}^{n-1}(t_j - 1)$ stops this loop and we have no extra run

7 $n-1$

8 1

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j - 1) + 1$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j - 1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5 \text{ (\textit{linear runtime})}$$

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \ldots j-1]$. Then, body of while-loop executed $j - 1$ times + one extra test to terminate while-loop $\implies t_j = j$ for all j.

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$$

$$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$$

$$= 4(n-1) + 4((n^2 - n)/2) - 2(n-1) + 1 = 2n^2 - 1 \text{ (\textit{quadratic runtime})}$$

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

- The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

  Example insertion sort: $2n^2 - 1$

- The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

  Example insertion sort: $6n - 5$

- The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

  Example insertion sort: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $2n^2 - 1$

The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $6n - 5$

The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

Example insertion sort: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.

Part 1-3: Runtime of algorithms

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $2n^2 - 1$

The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $6n - 5$

The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

Example insertion sort: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $2n^2 - 1$

The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $6n - 5$

The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

Example insertion sort: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.

Part 1-3: Runtime of algorithms

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $2n^2 - 1$

The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $6n - 5$

The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

Example insertion sort: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.

Part 1-3: Runtime of algorithms

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $2n^2 - 1$

The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $6n - 5$

The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

Example insertion sort: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.

Part 1-3: Runtime of algorithms

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $2n^2 - 1$

The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $6n - 5$

The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

Example insertion sort: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.

Part 1-3: Runtime of algorithms

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $2n^2 - 1$

The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $6n - 5$

The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

Example insertion sort: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $2n^2 - 1$

The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $6n - 5$

The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

Example insertion sort: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.

Part 1-3: Runtime of algorithms

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $2n^2 - 1$

The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $6n - 5$

The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

Example insertion sort: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.

Part 1-3: Runtime of algorithms

As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

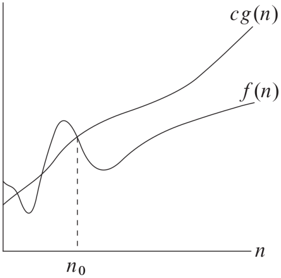
Notation Big-O, Big-Θ- and Big-Ω:

Part 1-3: Runtime of algorithms

As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

Notation Big-O, Big-Θ- and Big-Ω:

$$O(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

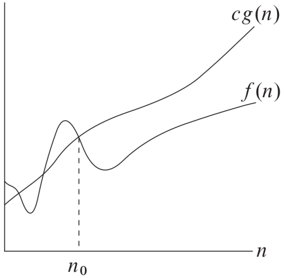


Part 1-3: Runtime of algorithms

As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

Notation Big-O, Big-Θ- and Big-Ω:

$$O(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$



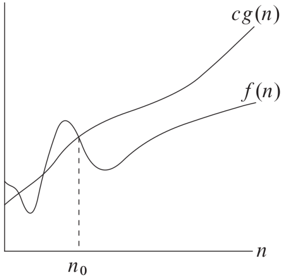
We use O-notation to give an upper bound on a function, to within a constant factor (asymptotic upper bound)

Part 1-3: Runtime of algorithms

As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

Notation Big-O, Big-Θ- and Big-Ω:

$$O(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}$$



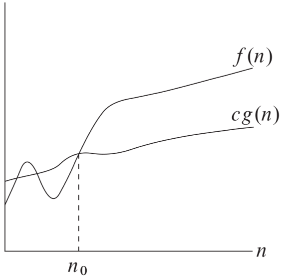
Part 1-3: Runtime of algorithms

As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

Notation Big-O, Big-Θ- and Big-Ω:

$$O(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$$\Omega(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$



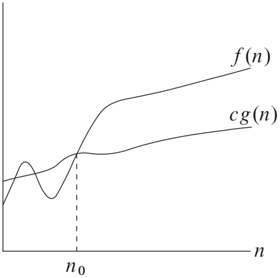
Part 1-3: Runtime of algorithms

As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

Notation Big-O, Big-Θ- and Big-Ω:

$$O(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$$\Omega(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$



We use Ω-notation to give a lower bound on a function, to within a constant factor (asymptotic lower bound)

Part 1-3: Runtime of algorithms

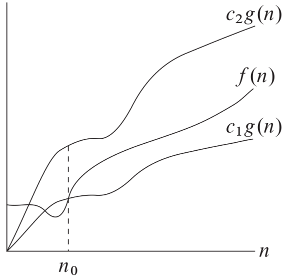
As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

Notation Big-O, Big-Θ- and Big-Ω:

$$O(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$$\Omega(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

$$\Theta(g(n)) := \{f(n) : \text{there are positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$



Part 1-3: Runtime of algorithms

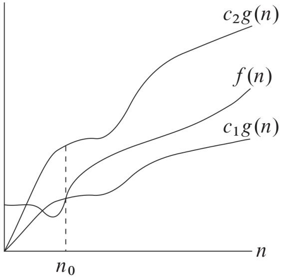
As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

Notation Big-O, Big-Θ- and Big-Ω:

$$O(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$$\Omega(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

$$\Theta(g(n)) := \{f(n) : \text{there are positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$



For all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor (asymptotically tight bound for $f(n)$)

Part 1-3: Runtime of algorithms

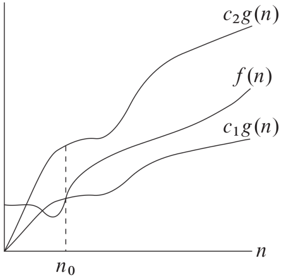
As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

Notation Big-O, Big-Θ- and Big-Ω:

$$O(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$$\Omega(g(n)) := \{f(n) : \text{there are positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

$$\Theta(g(n)) := \{f(n) : \text{there are positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$



Theorem: $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ if and only if $f(n) \in \Theta(g(n))$.

[proof exercise]

Part 1-3: Runtime of algorithms (Proofs on board)

For $f(n) = 0.5n^2 + 3n$ show:

$f(n) \in O(n^2)$; $f(n) \in \Omega(n^2)$ (and thus, $f(n) \in \Theta(n^2)$).

$f(n) \notin O(n)$; $f(n) \in \Omega(n)$ (and thus, $f(n) \notin \Theta(n)$).

$f(n) \in O(n^3)$; $f(n) \notin \Omega(n^3)$ (and thus, $f(n) \notin \Theta(n^3)$).

Show $f(n) = 2^{n+1} \in \Theta(2^n)$

For $f(n) = n^2(sin(n))^2 + 50n$ show:

$f(n) \in O(n^2)$ and $f(n) \in \Omega(n)$.

$50n \leq n^2 sin(n) \leq x^2$

Part 1-3: Runtime of algorithms (Proofs on board)

For $f(n) = 0.5n^2 + 3n$ show:

$f(n) \in O(n^2)$; $f(n) \in \Omega(n^2)$ (and thus, $f(n) \in \Theta(n^2)$).

$f(n) \notin O(n)$; $f(n) \in \Omega(n)$ (and thus, $f(n) \notin \Theta(n)$).

$f(n) \in O(n^3)$; $f(n) \notin \Omega(n^3)$ (and thus, $f(n) \notin \Theta(n^3)$).

Show $f(n) = 2^{n+1} \in \Theta(2^n)$

For $f(n) = n^2(sin(n))^2 + 50n$ show:

$f(n) \in O(n^2)$ and $f(n) \in \Omega(n)$.

$50n \leq n^2 sin(n) \leq x^2$

Part 1-3: Runtime of algorithms (Proofs on board)

For $f(n) = 0.5n^2 + 3n$ show:

$f(n) \in O(n^2)$; $f(n) \in \Omega(n^2)$ (and thus, $f(n) \in \Theta(n^2)$).

$f(n) \notin O(n)$; $f(n) \in \Omega(n)$ (and thus, $f(n) \notin \Theta(n)$).

$f(n) \in O(n^3)$; $f(n) \notin \Omega(n^3)$ (and thus, $f(n) \notin \Theta(n^3)$).

Show $f(n) = 2^{n+1} \in \Theta(2^n)$

For $f(n) = n^2(sin(n))^2 + 50n$ show:

$f(n) \in O(n^2)$ and $f(n) \in \Omega(n)$.

$50n \leq n^2 sin(n) \leq x^2$

Part 1-3: Runtime of algorithms (Proofs on board)

For $f(n) = 0.5n^2 + 3n$ show:

$f(n) \in O(n^2)$; $f(n) \in \Omega(n^2)$ (and thus, $f(n) \in \Theta(n^2)$).

$f(n) \notin O(n)$; $f(n) \in \Omega(n)$ (and thus, $f(n) \notin \Theta(n)$).

$f(n) \in O(n^3)$; $f(n) \notin \Omega(n^3)$ (and thus, $f(n) \notin \Theta(n^3)$).

Show $f(n) = 2^{n+1} \in \Theta(2^n)$

For $f(n) = n^2(sin(n))^2 + 50n$ show:

$f(n) \in O(n^2)$ and $f(n) \in \Omega(n)$.

$50n \leq n^2 sin(n) \leq x^2$

Part 1-3: Runtime of algorithms (Proofs on board)

For $f(n) = 0.5n^2 + 3n$ show:

$f(n) \in O(n^2)$; $f(n) \in \Omega(n^2)$ (and thus, $f(n) \in \Theta(n^2)$).

$f(n) \notin O(n)$; $f(n) \in \Omega(n)$ (and thus, $f(n) \notin \Theta(n)$).

$f(n) \in O(n^3)$; $f(n) \notin \Omega(n^3)$ (and thus, $f(n) \notin \Theta(n^3)$).

Show $f(n) = 2^{n+1} \in \Theta(2^n)$

For $f(n) = n^2(sin(n))^2 + 50n$ show:

$f(n) \in O(n^2)$ and $f(n) \in \Omega(n)$.

$50n \leq n^2 sin(n) \leq x^2$

For $f(n) = 0.5n^2 + 3n$ show:

$f(n) \in O(n^2)$; $f(n) \in \Omega(n^2)$ (and thus, $f(n) \in \Theta(n^2)$).

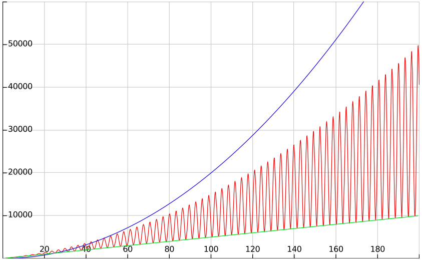
$f(n) \notin O(n)$; $f(n) \in \Omega(n)$ (and thus, $f(n) \notin \Theta(n)$).

$f(n) \in O(n^3)$; $f(n) \notin \Omega(n^3)$ (and thus, $f(n) \notin \Theta(n^3)$).

Show $f(n) = 2^{n+1} \in \Theta(2^n)$

For $f(n) = n^2(sin(n))^2 + 50n$ show:

$f(n) \in O(n^2)$ and $f(n) \in \Omega(n)$.



$50n \leq n^2 sin(n) \leq x^2$

Part 1-3: Runtime of algorithms

Insertion-sort revisited:

best-case: $T(n) = 6n - 5$

worst-case: $T(n) = 2n^2 - 1$

Thus, the running time of insertion-sort is in $O(n^2)$, that is, no matter what particular input of size n is chosen, the running time on that input is always bounded from above by some function cn^2 for some constants $c, n_0 > 0$ and all $n \geq n_0$.

At the same time, the running time of insertion-sort is in $\Omega(n)$, that is, no matter what particular input of size n is chosen, the running time on that input is at least cn, for some constants $c, n_0 > 0$ and all $n \geq n_0$.

Moreover, these bounds are asymptotically as tight as possible:
The running time of insertion-sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted).

It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$ since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

Part 1-3: Runtime of algorithms

Insertion-sort revisited:

best-case: $T(n) = 6n - 5 \xRightarrow{\text{Exerc.}} T(n) \in \Omega(n)$

worst-case: $T(n) = 2n^2 - 1 \xRightarrow{\text{Exerc.}} T(n) \in O(n^2)$

Thus, the running time of insertion-sort is in $O(n^2)$, that is, no matter what particular input of size n is chosen, the running time on that input is always bounded from above by some function cn^2 for some constants $c, n_0 > 0$ and all $n \geq n_0$.

At the same time, the running time of insertion-sort is in $\Omega(n)$, that is, no matter what particular input of size n is chosen, the running time on that input is at least cn, for some constants $c, n_0 > 0$ and all $n \geq n_0$.

Moreover, these bounds are asymptotically as tight as possible:
The running time of insertion-sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted).

It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$ since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

Part 1-3: Runtime of algorithms

Insertion-sort revisited:

best-case: $T(n) = 6n - 5 \overset{\text{Exerc.}}{\Longrightarrow} T(n) \in \Omega(n)$

worst-case: $T(n) = 2n^2 - 1 \overset{\text{Exerc.}}{\Longrightarrow} T(n) \in O(n^2)$

Thus, the running time of insertion-sort is in $O(n^2)$, that is, no matter what particular input of size n is chosen, the running time on that input is always bounded from above by some function cn^2 for some constants $c, n_0 > 0$ and all $n \geq n_0$.

At the same time, the running time of insertion-sort is in $\Omega(n)$, that is, no matter what particular input of size n is chosen, the running time on that input is at least cn, for some constants $c, n_0 > 0$ and all $n \geq n_0$.

Moreover, these bounds are asymptotically as tight as possible:
The running time of insertion-sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted).

It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$ since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

Part 1-3: Runtime of algorithms

Insertion-sort revisited:

best-case: $T(n) = 6n - 5 \stackrel{\text{Exerc.}}{\Longrightarrow} T(n) \in \Omega(n)$

worst-case: $T(n) = 2n^2 - 1 \stackrel{\text{Exerc.}}{\Longrightarrow} T(n) \in O(n^2)$

Thus, the running time of insertion-sort is in $O(n^2)$, that is, no matter what particular input of size n is chosen, the running time on that input is always bounded from above by some function cn^2 for some constants $c, n_0 > 0$ and all $n \geq n_0$.

At the same time, the running time of insertion-sort is in $\Omega(n)$, that is, no matter what particular input of size n is chosen, the running time on that input is at least cn, for some constants $c, n_0 > 0$ and all $n \geq n_0$.

Moreover, these bounds are asymptotically as tight as possible:
The running time of insertion-sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted).

It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$ since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

Insertion-sort revisited:

best-case: $T(n) = 6n - 5 \xRightarrow{\text{Exerc.}} T(n) \in \Omega(n)$

worst-case: $T(n) = 2n^2 - 1 \xRightarrow{\text{Exerc.}} T(n) \in O(n^2)$

Thus, the running time of insertion-sort is in $O(n^2)$, that is, no matter what particular input of size n is chosen, the running time on that input is always bounded from above by some function cn^2 for some constants $c, n_0 > 0$ and all $n \geq n_0$.

At the same time, the running time of insertion-sort is in $\Omega(n)$, that is, no matter what particular input of size n is chosen, the running time on that input is at least cn, for some constants $c, n_0 > 0$ and all $n \geq n_0$.

Moreover, these bounds are asymptotically as tight as possible:
The running time of insertion-sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted).

It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$ since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

Part 1-3: Runtime of algorithms

Insertion-sort revisited:

best-case: $T(n) = 6n - 5 \overset{\text{Exerc.}}{\Longrightarrow} T(n) \in \Omega(n)$

worst-case: $T(n) = 2n^2 - 1 \overset{\text{Exerc.}}{\Longrightarrow} T(n) \in O(n^2)$

Thus, the running time of insertion-sort is in $O(n^2)$, that is, no matter what particular input of size n is chosen, the running time on that input is always bounded from above by some function cn^2 for some constants $c, n_0 > 0$ and all $n \geq n_0$.

At the same time, the running time of insertion-sort is in $\Omega(n)$, that is, no matter what particular input of size n is chosen, the running time on that input is at least cn, for some constants $c, n_0 > 0$ and all $n \geq n_0$.

Moreover, these bounds are asymptotically as tight as possible:
The running time of insertion-sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted).

It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$ since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

Part 1-3: Runtime of algorithms

Insertion-sort revisited:

best-case: $T(n) = 6n - 5 \overset{\text{Exerc.}}{\Longrightarrow} T(n) \in \Omega(n)$

worst-case: $T(n) = 2n^2 - 1 \overset{\text{Exerc.}}{\Longrightarrow} T(n) \in O(n^2)$

Thus, the running time of insertion-sort is in $O(n^2)$, that is, no matter what particular input of size n is chosen, the running time on that input is always bounded from above by some function cn^2 for some constants $c, n_0 > 0$ and all $n \geq n_0$.

At the same time, the running time of insertion-sort is in $\Omega(n)$, that is, no matter what particular input of size n is chosen, the running time on that input is at least cn, for some constants $c, n_0 > 0$ and all $n \geq n_0$.

Moreover, these bounds are asymptotically as tight as possible:
The running time of insertion-sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted).

It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$ since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

Let $f(n)$ and $g(n)$ be asymptotically positive functions.

Then, for $\Upsilon \in \{O, \Omega, \Theta\}$, it holds that

- $f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity] — proof board
- $f(n) \in \Upsilon(f(n))$ [reflexivity] — proof exercise

Moreover, it holds that

- $f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry] — proof exercise
- $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry] — proof exercise

Let $f(n)$ and $g(n)$ be asymptotically positive functions.

Then, for $\Upsilon \in \{O, \Omega, \Theta\}$, it holds that

- $f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity] — proof board
- $f(n) \in \Upsilon(f(n))$ [reflexivity] — proof exercise

Moreover, it holds that

- $f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry] — proof exercise
- $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry] — proof exercise

Let $f(n)$ and $g(n)$ be asymptotically positive functions.

Then, for $\Upsilon \in \{O, \Omega, \Theta\}$, it holds that

- $f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity] — proof board
- $f(n) \in \Upsilon(f(n))$ [reflexivity] — proof exercise

Moreover, it holds that

- $f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry] — proof exercise
- $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry] — proof exercise

Let $f(n)$ and $g(n)$ be asymptotically positive functions.

Then, for $\Upsilon \in \{O, \Omega, \Theta\}$, it holds that

- $f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity] — proof board
- $f(n) \in \Upsilon(f(n))$ [reflexivity] — proof exercise

Moreover, it holds that

- $f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry] — proof exercise
- $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry] — proof exercise

Let $f(n)$ and $g(n)$ be asymptotically positive functions.

Then, for $\Upsilon \in \{O, \Omega, \Theta\}$, it holds that

- $f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity] — proof board
- $f(n) \in \Upsilon(f(n))$ [reflexivity] — proof exercise

Moreover, it holds that

- $f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry] — proof exercise
- $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry] — proof exercise

Part 1-3: Runtime of algorithms

$O(\ldots)$ (rt=runtime)	typical framework	typical examples
$O(1)$ constant rt	`a=b+c` // `if(a<b)`	assignments, in/output, 32/64bit-arithmetic, cases
$O(\log n)$ logarithmic rt	`while(N>1) N = N/2`	binary search
$O(n)$ linear rt	`for(i=0; i<n; i++){...}`	loop find the maximum
$O(n^2)$ quadratic rt	`for(i=0; i<n; i++)` `for(j=0; j<n; j++) {...}`	double loop, check all pairs

$O(n^3)$ cubic rt	`for(i=0; i<n; i++)`
	`for(j=0; j<n; j++)`
	`for(k=0; k<n; k++) {...}`
		triple loop,
		check all triples

$O(2^n)$ exponential rt	`see combinatorial lecture;)`	exhaustive search check all subsets

Instead of $f(n) \in O(g(n))$ one often writes $f(n) = O(g(n))$ (similar for Ω, Θ)

This is sometimes convenient when establishing certain estimations or calculations.

IMPORTANT NOT !!! $O(g(n)) = f(n)$

Example

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that there is some anonymous function $f(n) \in \Theta(n)$ that we do not care to name, such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.

This can help eliminate inessential detail and clutter in an equation and allows us also to write for $\Upsilon \in \{O, \Omega, \Theta\}$:

$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$ proof next slides

$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$ proof exercise

$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$ proof exercise

Instead of $f(n) \in O(g(n))$ one often writes $f(n) = O(g(n))$ (similar for Ω, Θ)

This is sometimes convenient when establishing certain estimations or calculations.

IMPORTANT NOT !!! $O(g(n)) = f(n)$

Example

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that there is some anonymous function $f(n) \in \Theta(n)$ that we do not care to name, such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.

This can help eliminate inessential detail and clutter in an equation and allows us also to write for $\Upsilon \in \{O, \Omega, \Theta\}$:

$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$ proof next slides

$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$ proof exercise

$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$ proof exercise

Instead of $f(n) \in O(g(n))$ one often writes $f(n) = O(g(n))$ (similar for Ω, Θ)

This is sometimes convenient when establishing certain estimations or calculations.

IMPORTANT NOT !!! $O(g(n)) = f(n)$

Example

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that there is some anonymous function $f(n) \in \Theta(n)$ that we do not care to name, such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.

This can help eliminate inessential detail and clutter in an equation and allows us also to write for $\Upsilon \in \{O, \Omega, \Theta\}$:

$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$ proof next slides

$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$ proof exercise

$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$ proof exercise

Instead of $f(n) \in O(g(n))$ one often writes $f(n) = O(g(n))$ (similar for Ω, Θ)

This is sometimes convenient when establishing certain estimations or calculations.

IMPORTANT NOT !!! $O(g(n)) = f(n)$

Example

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that there is some anonymous function $f(n) \in \Theta(n)$ that we do not care to name, such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.

This can help eliminate inessential detail and clutter in an equation and allows us also to write for $\Upsilon \in \{O, \Omega, \Theta\}$:

$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$ proof next slides

$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$ proof exercise

$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$ proof exercise

Instead of $f(n) \in O(g(n))$ one often writes $f(n) = O(g(n))$ (similar for Ω, Θ)

This is sometimes convenient when establishing certain estimations or calculations.

IMPORTANT NOT !!! $O(g(n)) = f(n)$

Example

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that there is some anonymous function $f(n) \in \Theta(n)$ that we do not care to name, such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.

This can help eliminate inessential detail and clutter in an equation and allows us also to write for $\Upsilon \in \{O, \Omega, \Theta\}$:

$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$ proof next slides

$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$ proof exercise

$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$ proof exercise

Instead of $f(n) \in O(g(n))$ one often writes $f(n) = O(g(n))$ (similar for Ω, Θ)

This is sometimes convenient when establishing certain estimations or calculations.

IMPORTANT NOT !!! $O(g(n)) = f(n)$

Example

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that there is some anonymous function $f(n) \in \Theta(n)$ that we do not care to name, such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.

This can help eliminate inessential detail and clutter in an equation and allows us also to write for $\Upsilon \in \{O, \Omega, \Theta\}$:

$$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$$ proof next slides

$$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$$ proof exercise

$$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$$ proof exercise

Instead of $f(n) \in O(g(n))$ one often writes $f(n) = O(g(n))$ (similar for Ω, Θ)

This is sometimes convenient when establishing certain estimations or calculations.

IMPORTANT NOT !!! $O(g(n)) = f(n)$

Example

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that there is some anonymous function $f(n) \in \Theta(n)$ that we do not care to name, such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.

This can help eliminate inessential detail and clutter in an equation and allows us also to write for $\Upsilon \in \{O, \Omega, \Theta\}$:

$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$ proof next slides

$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$ proof exercise

$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$ proof exercise

Instead of $f(n) \in O(g(n))$ one often writes $f(n) = O(g(n))$ (similar for Ω, Θ)

This is sometimes convenient when establishing certain estimations or calculations.

IMPORTANT NOT !!! $O(g(n)) = f(n)$

Example

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that there is some anonymous function $f(n) \in \Theta(n)$ that we do not care to name, such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.

This can help eliminate inessential detail and clutter in an equation and allows us also to write for $\Upsilon \in \{O, \Omega, \Theta\}$:

$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$ proof next slides

$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$ proof exercise

$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$ proof exercise

Part 1-3: Runtime of algorithms

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

Part 1-3: Runtime of algorithms

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

Part 1-3: Runtime of algorithms

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

We need to show that for any $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ it holds that

$$h(n) := \tilde{f}(n) + \tilde{g}(n) \in O(f(n) + g(n))$$

$\tilde{f}(n) \in O(f(n)) \implies \tilde{f}(n) \leq c' f(n)$ for some constants $c', n_0' > 0$ and all $n \geq n_0'$

$\tilde{g}(n) \in O(g(n)) \implies \tilde{g}(n) \leq c'' g(n)$ for some constants $c'', n_0'' > 0$ and all $n \geq n_0''$

Thus, $h(n) := \tilde{f}(n) + \tilde{g}(n) \leq c' f(n) + c'' g(n)$

$\leq c(f(n) + g(n))$ for all $n \geq n_0$ with $c = \max\{c', c''\}$ and $n_0 = \max\{n_0', n_0''\}$

Hence, $h(n) \in O(f(n) + g(n))$.

Since $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ have been arbitrarily chosen, we have

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)).$$

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

We need to show that for any $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ it holds that

$$h(n) := \tilde{f}(n) + \tilde{g}(n) \in O(f(n) + g(n))$$

$\tilde{f}(n) \in O(f(n)) \implies \tilde{f}(n) \leq c' f(n)$ for some constants $c', n_0' > 0$ and all $n \geq n_0'$

$\tilde{g}(n) \in O(g(n)) \implies \tilde{g}(n) \leq c'' g(n)$ for some constants $c'', n_0'' > 0$ and all $n \geq n_0''$

Thus, $h(n) := \tilde{f}(n) + \tilde{g}(n) \leq c' f(n) + c'' g(n)$

$\leq c(f(n) + g(n))$ for all $n \geq n_0$ with $c = \max\{c', c''\}$ and $n_0 = \max\{n_0', n_0''\}$

Hence, $h(n) \in O(f(n) + g(n))$.

Since $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ have been arbitrarily chosen, we have

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)).$$

Part 1-3: Runtime of algorithms

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

We need to show that for any $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ it holds that

$$h(n) := \tilde{f}(n) + \tilde{g}(n) \in O(f(n) + g(n))$$

$\tilde{f}(n) \in O(f(n)) \implies \tilde{f}(n) \leq c' f(n)$ for some constants $c', n_0' > 0$ and all $n \geq n_0'$

$\tilde{g}(n) \in O(g(n)) \implies \tilde{g}(n) \leq c'' g(n)$ for some constants $c'', n_0'' > 0$ and all $n \geq n_0''$

Thus, $h(n) := \tilde{f}(n) + \tilde{g}(n) \leq c' f(n) + c'' g(n)$

$\leq c(f(n) + g(n))$ for all $n \geq n_0$ with $c = \max\{c', c''\}$ and $n_0 = \max\{n_0', n_0''\}$

Hence, $h(n) \in O(f(n) + g(n))$.

Since $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ have been arbitrarily chosen, we have

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)).$$

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

We need to show that for any $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ it holds that

$$h(n) := \tilde{f}(n) + \tilde{g}(n) \in O(f(n) + g(n))$$

$\tilde{f}(n) \in O(f(n)) \implies \tilde{f}(n) \leq c' f(n)$ for some constants $c', n_0' > 0$ and all $n \geq n_0'$

$\tilde{g}(n) \in O(g(n)) \implies \tilde{g}(n) \leq c'' g(n)$ for some constants $c'', n_0'' > 0$ and all $n \geq n_0''$

Thus, $h(n) := \tilde{f}(n) + \tilde{g}(n) \leq c' f(n) + c'' g(n)$
$\leq c(f(n) + g(n))$ for all $n \geq n_0$ with $c = \max\{c', c''\}$ and $n_0 = \max\{n_0', n_0''\}$

Hence, $h(n) \in O(f(n) + g(n))$.

Since $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ have been arbitrarily chosen, we have

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)).$$

Part 1-3: Runtime of algorithms

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

We need to show that for any $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ it holds that

$$h(n) := \tilde{f}(n) + \tilde{g}(n) \in O(f(n) + g(n))$$

$\tilde{f}(n) \in O(f(n)) \implies \tilde{f}(n) \leq c' f(n)$ for some constants $c', n_0' > 0$ and all $n \geq n_0'$

$\tilde{g}(n) \in O(g(n)) \implies \tilde{g}(n) \leq c'' g(n)$ for some constants $c'', n_0'' > 0$ and all $n \geq n_0''$

Thus, $h(n) := \tilde{f}(n) + \tilde{g}(n) \leq c' f(n) + c'' g(n)$

$\leq c(f(n) + g(n))$ for all $n \geq n_0$ with $c = \max\{c', c''\}$ and $n_0 = \max\{n_0', n_0''\}$

Hence, $h(n) \in O(f(n) + g(n))$.

Since $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ have been arbitrarily chosen, we have

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)).$$

Part 1-3: Runtime of algorithms

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

We need to show that for any $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ it holds that

$$h(n) := \tilde{f}(n) + \tilde{g}(n) \in O(f(n) + g(n))$$

$\tilde{f}(n) \in O(f(n)) \implies \tilde{f}(n) \leq c' f(n)$ for some constants $c', n_0' > 0$ and all $n \geq n_0'$

$\tilde{g}(n) \in O(g(n)) \implies \tilde{g}(n) \leq c'' g(n)$ for some constants $c'', n_0'' > 0$ and all $n \geq n_0''$

Thus, $h(n) := \tilde{f}(n) + \tilde{g}(n) \leq c' f(n) + c'' g(n)$

$\leq c(f(n) + g(n))$ for all $n \geq n_0$ with $c = \max\{c', c''\}$ and $n_0 = \max\{n_0', n_0''\}$

Hence, $h(n) \in O(f(n) + g(n))$.

Since $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ have been arbitrarily chosen, we have

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)).$$

Part 1-3: Runtime of algorithms

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

Let $h(n) \in O(f(n) + g(n))$.

$\implies h(n) \leq c(f(n) + g(n))$ for some constants $c, n_0 > 0$ and all $n \geq n_0$

$\implies h(n) \leq c \cdot 2 \cdot \max\{f(n), g(n)\}$

$\implies h(n) \leq \tilde{c} \cdot \max\{f(n), g(n)\}$ with $\tilde{c} = 2c$

$\implies h(n) \in O(\max(f(n), g(n)))$

Since $h(n) \in O(f(n) + g(n))$ has been arbitrarily chosen we have

$$O(f(n) + g(n)) = O(\max(f(n), g(n)))$$

Part 1-3: Runtime of algorithms

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

Let $h(n) \in O(f(n) + g(n))$.

$\implies h(n) \le c(f(n) + g(n))$ for some constants $c, n_0 > 0$ and all $n \ge n_0$

$\implies h(n) \le c \cdot 2 \cdot \max\{f(n), g(n)\}$

$\implies h(n) \le \tilde{c} \cdot \max\{f(n), g(n)\}$ with $\tilde{c} = 2c$

$\implies h(n) \in O(\max(f(n), g(n)))$

Since $h(n) \in O(f(n) + g(n))$ has been arbitrarily chosen we have

$$O(f(n) + g(n)) = O(\max(f(n), g(n)))$$

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

Let $h(n) \in O(f(n) + g(n))$.

$\implies h(n) \leq c(f(n) + g(n))$ for some constants $c, n_0 > 0$ and all $n \geq n_0$

$\implies h(n) \leq c \cdot 2 \cdot \max\{f(n), g(n)\}$

$\implies h(n) \leq \tilde{c} \cdot \max\{f(n), g(n)\}$ with $\tilde{c} = 2c$

$\implies h(n) \in O(\max(f(n), g(n)))$

Since $h(n) \in O(f(n) + g(n))$ has been arbitrarily chosen we have

$$O(f(n) + g(n)) = O(\max(f(n), g(n)))$$

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

Let $h(n) \in O(f(n) + g(n))$.

$\implies h(n) \leq c(f(n) + g(n))$ for some constants $c, n_0 > 0$ and all $n \geq n_0$

$\implies h(n) \leq c \cdot 2 \cdot \max\{f(n), g(n)\}$

$\implies h(n) \leq \tilde{c} \cdot \max\{f(n), g(n)\}$ with $\tilde{c} = 2c$

$\implies h(n) \in O(\max(f(n), g(n)))$

Since $h(n) \in O(f(n) + g(n))$ has been arbitrarily chosen we have

$$O(f(n) + g(n)) = O(\max(f(n), g(n)))$$

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

Let $h(n) \in O(f(n) + g(n))$.

$\implies h(n) \leq c(f(n) + g(n))$ for some constants $c, n_0 > 0$ and all $n \geq n_0$

$\implies h(n) \leq c \cdot 2 \cdot \max\{f(n), g(n)\}$

$\implies h(n) \leq \tilde{c} \cdot \max\{f(n), g(n)\}$ with $\tilde{c} = 2c$

$\implies h(n) \in O(\max(f(n), g(n)))$

Since $h(n) \in O(f(n) + g(n))$ has been arbitrarily chosen we have

$$O(f(n) + g(n)) = O(\max(f(n), g(n)))$$

Part 1-3: Runtime of algorithms

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

Let $h(n) \in O(f(n) + g(n))$.

$\implies h(n) \leq c(f(n) + g(n))$ for some constants $c, n_0 > 0$ and all $n \geq n_0$

$\implies h(n) \leq c \cdot 2 \cdot \max\{f(n), g(n)\}$

$\implies h(n) \leq \tilde{c} \cdot \max\{f(n), g(n)\}$ with $\tilde{c} = 2c$

$\implies h(n) \in O(\max(f(n), g(n)))$

Since $h(n) \in O(f(n) + g(n))$ has been arbitrarily chosen we have

$$O(f(n) + g(n)) = O(\max(f(n), g(n)))$$

Part 1-3: Runtime of algorithms

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

Let $h(n) \in O(f(n) + g(n))$.

$\implies h(n) \leq c(f(n) + g(n))$ for some constants $c, n_0 > 0$ and all $n \geq n_0$

$\implies h(n) \leq c \cdot 2 \cdot \max\{f(n), g(n)\}$

$\implies h(n) \leq \tilde{c} \cdot \max\{f(n), g(n)\}$ with $\tilde{c} = 2c$

$\implies h(n) \in O(\max(f(n), g(n)))$

Since $h(n) \in O(f(n) + g(n))$ has been arbitrarily chosen we have

$$O(f(n) + g(n)) = O(\max(f(n), g(n)))$$

Part 1-3: Runtime of algorithms

Exmpl: Applying $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$ and $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

```
Do_Smth(int n)
PRINT "Hello World"
FOR (i = 0 to n − 1) DO
    i := i + 1
    IF (n is even) THEN RETURN 0
    ELSE
        FOR (j = 0 to n − 1) DO
            j := j + 1
```

Exmpl: Applying $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$ and $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

```
Do_Smth(int n)
PRINT "Hello World"
FOR (i = 0 to n − 1) DO
    i := i + 1
    IF (n is even) THEN RETURN 0
    ELSE
        FOR (j = 0 to n − 1) DO
            j := j + 1
```

Exmpl: Applying $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$ and $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

```
Do_Smth(int n)
PRINT "Hello World"
FOR (i = 0 to n − 1) DO
    i := i + 1
    IF (n is even) THEN RETURN 0
    ELSE
        FOR (j = 0 to n − 1) DO
            j := j + 1
```

All basic-instructions (eg. PRINT, $i = 0$, $j := j + 1$, RETURN 0, ...) in $O(1)$ time

Part 1-3: Runtime of algorithms

Exmpl: Applying $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$ and $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

```
Do_Smth(int n)
1 PRINT "Hello World"
2 FOR (i = 0 to n − 1) DO
3     i := i + 1
4     IF (n is even) THEN RETURN 0
5     ELSE
6        FOR (j = 0 to n − 1) DO
7           j := j + 1
```

All basic-instructions (eg. PRINT, $i = 0$, $j := j + 1$, RETURN 0, $\ldots$) in $O(1)$ time

Do_Smth consists of two main-parts:

$$A_1 = \texttt{PRINT "Hello World"} \text{ and } A_2 = \text{Line 2-7}$$

Hence, runtime of DO_SMTH is in $O(1)+$ runtime $A_2 \implies$ examine A_2 !

Part 1-3: Runtime of algorithms

Exmpl: Applying $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$ and $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

```
Do_Smth(int n)
1 PRINT "Hello World"
2 FOR (i = 0 to n − 1) DO
3     i := i + 1
4     IF (n is even) THEN RETURN 0
5     ELSE
6         FOR (j = 0 to n − 1) DO
7             j := j + 1
```

runtime A_2 = Line 2-7

The most expensive task within the loop in Line 2 is in $O(n)$:

Part 1-3: Runtime of algorithms

Exmpl: Applying $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$ and $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

```
Do_Smth(int n)
1 PRINT "Hello World"
2 FOR (i = 0 to n − 1) DO
3     i := i + 1
4     IF (n is even) THEN RETURN 0
5     ELSE
6        FOR (j = 0 to n − 1) DO
7           j := j + 1
```

runtime A_2 = Line 2-7

The most expensive task within the loop in Line 2 is in $O(n)$:

Line 3: $O(1)$
Line 4: $O(1) + O(1) = O(\max(1, 1)) = O(1)$
Line 5: $O(1)$
Line 6-7: $O(n) \cdot O(1) = O(n \cdot 1) = O(n)$

Part 1-3: Runtime of algorithms

Exmpl: Applying $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$ and $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

```
Do_Smth(int n)
1 PRINT "Hello World"
2 FOR (i = 0 to n − 1) DO
3     i := i + 1
4     IF (n is even) THEN RETURN 0
5     ELSE
6         FOR (j = 0 to n − 1) DO
7             j := j + 1
```

runtime A_2 = Line 2-7

The most expensive task within the loop in Line 2 is in $O(n)$:

Line 3: $O(1)$
Line 4: $O(1) + O(1) = O(\max(1, 1)) = O(1)$
Line 5: $O(1)$
Line 6-7: $O(n) \cdot O(1) = O(n \cdot 1) = O(n)$

Line 3-7: $O(1) + O(1) + O(1) + O(n) = O(\max(1, 1, 1, n)) = O(n)$

Part 1-3: Runtime of algorithms

Exmpl: Applying $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$ and $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

```
Do_Smth(int n)
1 PRINT "Hello World"
2 FOR (i = 0 to n − 1) DO
3     i := i + 1
4     IF (n is even) THEN RETURN 0
5     ELSE
6         FOR (j = 0 to n − 1) DO
7             j := j + 1
```

runtime A_2 = Line 2-7

The most expensive task within the loop in Line 2 is in $O(n)$:

Line 3: $O(1)$
Line 4: $O(1) + O(1) = O(\max(1, 1)) = O(1)$
Line 5: $O(1)$
Line 6-7: $O(n) \cdot O(1) = O(n \cdot 1) = O(n)$

Line 3-7: $O(1) + O(1) + O(1) + O(n) = O(\max(1, 1, 1, n)) = O(n)$

FOR-loop in Line 2 runs n times. runtime $A_2 = O(n) \cdot O(n) = O(n \cdot n) = O(n^2)$

Part 1-3: Runtime of algorithms

Exmpl: Applying $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$ and $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

```
Do_Smth(int n)
PRINT "Hello World"
FOR (i = 0 to n − 1) DO
    i := i + 1
    IF (n is even) THEN RETURN 0
    ELSE
        FOR (j = 0 to n − 1) DO
            j := j + 1
```

Runtime DO_SMTH is in $O(1)+$ runtime A_2 = $O(1) + O(n^2) = O(\max(1, n^2)) = O(n^2)$

Summary up to here

$O(g(n)) := \{f(n)\colon \exists$ constants $c, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$

$\Omega(g(n)) := \{f(n)\colon \exists$ constants $c, n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$

$\Theta(g(n)) := \{f(n)\colon \exists$ constants $c_1, c_2, n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$

Theorem: $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ if and only if $f(n) \in \Theta(g(n))$.

Let $f(n)$ and $g(n)$ be asymptotically positive functions and $\Upsilon \in \{O, \Omega, \Theta\}$. Then,

$f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity]

$f(n) \in \Upsilon(f(n))$ [reflexivity]

Moreover, it holds that

$f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry]

$f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry]

The following rules can be applied:

$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$

$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$

$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$

It could be that some of these equations must be proven in exercises or the exam!

Summary up to here

$O(g(n)) := \{f(n) \colon \exists \text{ constants } c, n_0 > 0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}$

$\Omega(g(n)) := \{f(n) \colon \exists \text{ constants } c, n_0 > 0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}$

$\Theta(g(n)) := \{f(n) \colon \exists \text{ constants } c_1, c_2, n_0 > 0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}$

Theorem: $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ if and only if $f(n) \in \Theta(g(n))$.

Let $f(n)$ and $g(n)$ be asymptotically positive functions and $\Upsilon \in \{O, \Omega, \Theta\}$. Then,

$f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity]

$f(n) \in \Upsilon(f(n))$ [reflexivity]

Moreover, it holds that

$f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry]

$f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry]

The following rules can be applied:

$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$

$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$

$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$

It could be that some of these equations must be proven in exercises or the exam!

Summary up to here

$O(g(n)) := \{f(n) \colon \exists \text{ constants } c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$

$\Omega(g(n)) := \{f(n) \colon \exists \text{ constants } c, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$

$\Theta(g(n)) := \{f(n) \colon \exists \text{ constants } c_1, c_2, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$

Theorem: $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ if and only if $f(n) \in \Theta(g(n))$.

Let $f(n)$ and $g(n)$ be asymptotically positive functions and $\Upsilon \in \{O, \Omega, \Theta\}$. Then,

$f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity]

$f(n) \in \Upsilon(f(n))$ [reflexivity]

Moreover, it holds that

$f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry]

$f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry]

The following rules can be applied:

$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$

$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$

$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$

It could be that some of these equations must be proven in exercises or the exam!

Summary up to here

$$O(g(n)) := \{f(n) : \exists \text{ constants } c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$
$$\Omega(g(n)) := \{f(n) : \exists \text{ constants } c, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$
$$\Theta(g(n)) := \{f(n) : \exists \text{ constants } c_1, c_2, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

Theorem: $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ if and only if $f(n) \in \Theta(g(n))$.

Let $f(n)$ and $g(n)$ be asymptotically positive functions and $\Upsilon \in \{O, \Omega, \Theta\}$. Then,

$f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity]

$f(n) \in \Upsilon(f(n))$ [reflexivity]

Moreover, it holds that

$f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry]

$f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry]

The following rules can be applied:

$$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$$
$$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$$
$$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$$

It could be that some of these equations must be proven in exercises or the exam!

Summary up to here

$$O(g(n)) := \{f(n) : \exists \text{ constants } c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$
$$\Omega(g(n)) := \{f(n) : \exists \text{ constants } c, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$
$$\Theta(g(n)) := \{f(n) : \exists \text{ constants } c_1, c_2, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

Theorem: $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ if and only if $f(n) \in \Theta(g(n))$.

Let $f(n)$ and $g(n)$ be asymptotically positive functions and $\Upsilon \in \{O, \Omega, \Theta\}$. Then,

$f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity]

$f(n) \in \Upsilon(f(n))$ [reflexivity]

Moreover, it holds that

$f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry]

$f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry]

The following rules can be applied:

$$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$$
$$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$$
$$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$$

It could be that some of these equations must be proven in exercises or the exam!

Summary up to here

$$O(g(n)) := \{f(n) : \exists \text{ constants } c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$
$$\Omega(g(n)) := \{f(n) : \exists \text{ constants } c, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$
$$\Theta(g(n)) := \{f(n) : \exists \text{ constants } c_1, c_2, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

Theorem: $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ if and only if $f(n) \in \Theta(g(n))$.

Let $f(n)$ and $g(n)$ be asymptotically positive functions and $\Upsilon \in \{O, \Omega, \Theta\}$. Then,

$f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity]

$f(n) \in \Upsilon(f(n))$ [reflexivity]

Moreover, it holds that

$f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry]

$f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry]

The following rules can be applied:

$$\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$$
$$c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$$
$$\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$$

It could be that some of these equations must be proven in exercises or the exam!

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$T(n) =$

Part 1-3: Runtime of algorithms

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$T(n) = \Theta(1) + T(\frac{n}{2})$$

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$T(n) = \Theta(1) + T(\frac{n}{2})$$

$$= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2})$$

Part 1-3: Runtime of algorithms

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$T(n) = \Theta(1) + T(\frac{n}{2})$$
$$= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2})$$
$$= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3})$$

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$
\begin{aligned}
T(n) &= \Theta(1) + T(\frac{n}{2}) \\
&= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2}) \\
&= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3}) \\
&= \ldots \\
&= N \cdot \Theta(1) + T(\frac{n}{2^N}))
\end{aligned}
$$

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$
\begin{aligned}
T(n) &= \Theta(1) + T(\frac{n}{2}) \\
&= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2}) \\
&= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3}) \\
&= \dots \\
&= N \cdot \Theta(1) + T(\frac{n}{2^N}))
\end{aligned}
$$

How often can one repeat this, that is, what is N?

Part 1-3: Runtime of algorithms

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$T(n) = \Theta(1) + T(\frac{n}{2})$$
$$= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2})$$
$$= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3})$$
$$= \ldots$$
$$= N \cdot \Theta(1) + T(\frac{n}{2^N}))$$

How often can one repeat this, that is, what is N?

In other words: For which $k = \frac{n}{2^N}$ does `Halve`(k) terminate?

Answer: For any $k \leq 1$

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$
\begin{aligned}
T(n) &= \Theta(1) + T(\frac{n}{2}) \\
&= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2}) \\
&= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3}) \\
&= \dots \\
&= N \cdot \Theta(1) + T(\frac{n}{2^N}))
\end{aligned}
$$

How often can one repeat this, that is, what is N?

In other words: For which $k = \frac{n}{2^N}$ does Halve(k) terminate?

Answer: For any $k \leq 1 \iff \frac{n}{2^N} \leq 1 \iff n \leq 2^N \iff \log_2(n) \leq N$

Further example

`Halve`(number n)

    WHILE ($n > 1$) DO
        $n := \frac{n}{2}$

$$
\begin{aligned}
T(n) &= \Theta(1) + T(\frac{n}{2}) \\
&= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2}) \\
&= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3}) \\
&= \dots \\
&= N \cdot \Theta(1) + T(\frac{n}{2^N}))
\end{aligned}
$$

How often can one repeat this, that is, what is N?

In other words: For which $k = \frac{n}{2^N}$ does `Halve`(k) terminate?

Answer: For any $k \leq 1 \iff \frac{n}{2^N} \leq 1 \iff n \leq 2^N \iff \log_2(n) \leq N$

Put $N = \log_2(n)$ and note $T(1) = \Theta(1)$: $T(n) = N \cdot \Theta(1) + T(\frac{n}{2^N})$

Part 1-3: Runtime of algorithms

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$
\begin{aligned}
T(n) &= \Theta(1) + T(\frac{n}{2}) \\
&= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2}) \\
&= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3}) \\
&= \dots \\
&= N \cdot \Theta(1) + T(\frac{n}{2^N}))
\end{aligned}
$$

How often can one repeat this, that is, what is N?

In other words: For which $k = \frac{n}{2^N}$ does Halve(k) terminate?

Answer: For any $k \leq 1 \iff \frac{n}{2^N} \leq 1 \iff n \leq 2^N \iff \log_2(n) \leq N$

Put $N = \log_2(n)$ and note $T(1) = \Theta(1)$:

$$
\begin{aligned}
T(n) &= N \cdot \Theta(1) + T(\frac{n}{2^N}) \\
&= \log_2(n) \cdot \Theta(1) + T(1)
\end{aligned}
$$

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$T(n) = \Theta(1) + T(\frac{n}{2})$$
$$= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2})$$
$$= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3})$$
$$= \ldots$$
$$= N \cdot \Theta(1) + T(\frac{n}{2^N}))$$

How often can one repeat this, that is, what is N?

In other words: For which $k = \frac{n}{2^N}$ does Halve(k) terminate?

Answer: For any $k \leq 1 \iff \frac{n}{2^N} \leq 1 \iff n \leq 2^N \iff \log_2(n) \leq N$

Put $N = \log_2(n)$ and note $T(1) = \Theta(1)$: $T(n) = N \cdot \Theta(1) + T(\frac{n}{2^N})$

$$= \log_2(n) \cdot \Theta(1) + T(1)$$
$$= \Theta(\log_2(n)) \cdot \Theta(1) + \Theta(1)$$

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$
\begin{aligned}
T(n) &= \Theta(1) + T(\frac{n}{2}) \\
&= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2}) \\
&= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3}) \\
&= \dots \\
&= N \cdot \Theta(1) + T(\frac{n}{2^N}))
\end{aligned}
$$

How often can one repeat this, that is, what is N?

In other words: For which $k = \frac{n}{2^N}$ does Halve(k) terminate?

Answer: For any $k \leq 1 \iff \frac{n}{2^N} \leq 1 \iff n \leq 2^N \iff \log_2(n) \leq N$

Put $N = \log_2(n)$ and note $T(1) = \Theta(1)$:

$$
\begin{aligned}
T(n) &= N \cdot \Theta(1) + T(\frac{n}{2^N}) \\
&= \log_2(n) \cdot \Theta(1) + T(1) \\
&= \Theta(\log_2(n)) \cdot \Theta(1) + \Theta(1) \\
&= \Theta(\log_2(n) \cdot 1) + \Theta(1)
\end{aligned}
$$

Further example

```
Halve(number n)
    WHILE (n > 1) DO
        n := n/2
```

$$T(n) = \Theta(1) + T(\frac{n}{2})$$
$$= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2})$$
$$= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3})$$
$$= \dots$$
$$= N \cdot \Theta(1) + T(\frac{n}{2^N}))$$

How often can one repeat this, that is, what is N?

In other words: For which $k = \frac{n}{2^N}$ does Halve(k) terminate?

Answer: For any $k \leq 1 \iff \frac{n}{2^N} \leq 1 \iff n \leq 2^N \iff \log_2(n) \leq N$

Put $N = \log_2(n)$ and note $T(1) = \Theta(1)$:

$$T(n) = N \cdot \Theta(1) + T(\frac{n}{2^N})$$
$$= \log_2(n) \cdot \Theta(1) + T(1)$$
$$= \Theta(\log_2(n)) \cdot \Theta(1) + \Theta(1)$$
$$= \Theta(\log_2(n) \cdot 1) + \Theta(1)$$
$$= \Theta(\max\{\log_2(n), 1\}) = \Theta(\log_2(n))$$

Iterative vs. recursive algorithms

iterative	recursive
Sum(n) $total\_sum := 0$ FOR ($i = 1$ to n) DO $total\_sum := total\_sum + i$ PRINT $total\_sum$	Sum(int n) IF($n = 1$) THEN RETURN 1 RETURN $n +$ SUM$(n - 1)$

Iterative vs. recursive algorithms

iterative	recursive

Sum(n)
  $total\_sum := 0$
  FOR ($i = 1$ to n) DO
    $total\_sum := total\_sum + i$
  PRINT $total\_sum$
	Sum(int n)
	  IF($n = 1$) THEN RETURN 1
	  RETURN $n + \text{SUM}(n-1)$

What are these algorithms doing?

Iterative vs. recursive algorithms

iterative	recursive

`Sum`(n)
  $total\_sum := 0$
  `FOR` ($i = 1$ to n) `DO`
    $total\_sum := total\_sum + i$
  `PRINT` $total\_sum$
	`Sum`(int n)
	  `IF`($n = 1$) `THEN RETURN` 1
	  `RETURN` $n + \text{SUM}(n-1)$

What are these algorithms doing? **Answer:** `Sum` computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

Iterative vs. recursive algorithms

iterative	recursive
Sum(n) $total\_sum := 0$ FOR ($i = 1$ to n) DO $total\_sum := total\_sum + i$ PRINT $total\_sum$	Sum(int n) IF($n = 1$) THEN RETURN 1 RETURN $n +$ SUM$(n - 1)$

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

iterative ($n = 4$)	recursive ($n = 4$)
$total\_sum := 0$ $total\_sum := 0 + 1 = 1$ $total\_sum := 1 + 2 = 3$ $total\_sum := 3 + 3 = 6$ $total\_sum := 6 + 4 = 10$	

Iterative vs. recursive algorithms

iterative	recursive
Sum(n) $total\_sum := 0$ FOR ($i = 1$ to n) DO $total\_sum := total\_sum + i$ PRINT $total\_sum$	Sum(int n) IF($n = 1$) THEN RETURN 1 RETURN $n + \text{SUM}(n-1)$

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

iterative ($n = 4$)	recursive ($n = 4$)
$total\_sum := 0$	RETURN $4 + \text{SUM}(3)$ (the return value of SUM(4))
$total\_sum := 0 + 1 = 1$	
$total\_sum := 1 + 2 = 3$	
$total\_sum := 3 + 3 = 6$	
$total\_sum := 6 + 4 = 10$	

Iterative vs. recursive algorithms

iterative	recursive
Sum(n) $total\_sum := 0$ FOR ($i = 1$ to n) DO $total\_sum := total\_sum + i$ PRINT $total\_sum$	Sum(int n) IF($n = 1$) THEN RETURN 1 RETURN $n + \text{SUM}(n-1)$

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

iterative ($n = 4$)	**recursive** ($n = 4$)
$total\_sum := 0$	RETURN $4 + \text{SUM}(3)$ (the return value of SUM(4))
$total\_sum := 0 + 1 = 1$	RETURN $3 + \text{SUM}(2)$ (the return value of SUM(3))
$total\_sum := 1 + 2 = 3$	
$total\_sum := 3 + 3 = 6$	
$total\_sum := 6 + 4 = 10$	

Iterative vs. recursive algorithms

iterative	recursive

Sum(n)
  $total\_sum := 0$
  FOR ($i = 1$ to n) DO
    $total\_sum := total\_sum + i$
  PRINT $total\_sum$
	Sum(int n)
	  IF($n = 1$) THEN RETURN 1
	  RETURN $n + \text{SUM}(n-1)$

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

iterative ($n = 4$)	recursive ($n = 4$)
$total\_sum := 0$	RETURN $4 + \text{SUM}(3)$ (the return value of SUM(4))
$total\_sum := 0 + 1 = 1$	RETURN $3 + \text{SUM}(2)$ (the return value of SUM(3))
$total\_sum := 1 + 2 = 3$	RETURN $2 + \text{SUM}(1)$ (the return value of SUM(2))
$total\_sum := 3 + 3 = 6$	
$total\_sum := 6 + 4 = 10$	

Iterative vs. recursive algorithms

iterative	recursive
Sum(n) $total\_sum := 0$ FOR ($i = 1$ to n) DO $total\_sum := total\_sum + i$ PRINT $total\_sum$	Sum(int n) IF($n = 1$) THEN RETURN 1 RETURN $n + \text{SUM}(n-1)$

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

iterative ($n = 4$)	recursive ($n = 4$)
$total\_sum := 0$	RETURN $4 + \text{SUM}(3)$ (the return value of SUM(4))
$total\_sum := 0 + 1 = 1$	RETURN $3 + \text{SUM}(2)$ (the return value of SUM(3))
$total\_sum := 1 + 2 = 3$	RETURN $2 + \text{SUM}(1)$ (the return value of SUM(2))
$total\_sum := 3 + 3 = 6$	RETURN 1 (the return value of SUM(1)) [SUM(1) = 1]
$total\_sum := 6 + 4 = 10$	

Iterative vs. recursive algorithms

iterative	recursive
Sum(n) $total\_sum := 0$ FOR ($i = 1$ to n) DO $total\_sum := total\_sum + i$ PRINT $total\_sum$	Sum(int n) IF($n = 1$) THEN RETURN 1 RETURN $n +$ SUM$(n - 1)$

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

iterative ($n = 4$)	recursive ($n = 4$)
$total\_sum := 0$	RETURN $4 +$ SUM(3) (the return value of SUM(4))
$total\_sum := 0 + 1 = 1$	RETURN $3 +$ SUM(2) (the return value of SUM(3))
$total\_sum := 1 + 2 = 3$	RETURN $2 +$ SUM(1) (the return value of SUM(2))
$total\_sum := 3 + 3 = 6$	RETURN 1 (the return value of SUM(1)) [SUM(1) = 1]
$total\_sum := 6 + 4 = 10$	$\Longrightarrow$ $2 +$ SUM$(1) = 2 + 1 = 3$ [SUM(2) = 3]

Iterative vs. recursive algorithms

iterative	recursive
`Sum`(n) $total\_sum := 0$ `FOR` ($i = 1$ to n) `DO` $total\_sum := total\_sum + i$ `PRINT` $total\_sum$	`Sum`(int n) `IF`($n = 1$) `THEN RETURN` 1 `RETURN` $n + \text{SUM}(n-1)$

What are these algorithms doing? **Answer:** `Sum` computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

iterative ($n = 4$)	recursive ($n = 4$)
$total\_sum := 0$	`RETURN` $4 + \text{SUM}(3)$ (the return value of $\text{SUM}(4)$)
$total\_sum := 0 + 1 = 1$	`RETURN` $3 + \text{SUM}(2)$ (the return value of $\text{SUM}(3)$)
$total\_sum := 1 + 2 = 3$	`RETURN` $2 + \text{SUM}(1)$ (the return value of $\text{SUM}(2)$)
$total\_sum := 3 + 3 = 6$	`RETURN` 1 (the return value of $\text{SUM}(1)$) [$\text{SUM}(1) = 1$]
$total\_sum := 6 + 4 = 10$	$\Longrightarrow 2 + \text{SUM}(1) = 2 + 1 = 3$ [$\text{SUM}(2) = 3$]
	$\Longrightarrow 3 + \text{SUM}(2) = 3 + 3 = 6$ [$\text{SUM}(3) = 6$]

Iterative vs. recursive algorithms

iterative	recursive
`Sum`(n) $total\_sum := 0$ `FOR` ($i = 1$ to n) `DO` $total\_sum := total\_sum + i$ `PRINT` $total\_sum$	`Sum`(int n) `IF`($n = 1$) `THEN RETURN` 1 `RETURN` $n + \textsc{Sum}(n-1)$

What are these algorithms doing? **Answer:** `Sum` computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

iterative ($n = 4$)	recursive ($n = 4$)
$total\_sum := 0$	`RETURN` $4 + \textsc{Sum}(3)$ (the return value of $\textsc{Sum}(4)$)
$total\_sum := 0 + 1 = 1$	`RETURN` $3 + \textsc{Sum}(2)$ (the return value of $\textsc{Sum}(3)$)
$total\_sum := 1 + 2 = 3$	`RETURN` $2 + \textsc{Sum}(1)$ (the return value of $\textsc{Sum}(2)$)
$total\_sum := 3 + 3 = 6$	`RETURN` 1 (the return value of $\textsc{Sum}(1)$) [$\textsc{Sum}(1) = 1$]
$total\_sum := 6 + 4 = 10$	$\Longrightarrow 2 + \textsc{Sum}(1) = 2 + 1 = 3$ [$\textsc{Sum}(2) = 3$]
	$\Longrightarrow 3 + \textsc{Sum}(2) = 3 + 3 = 6$ [$\textsc{Sum}(3) = 6$]
	$\Longrightarrow 4 + \textsc{Sum}(3) = 4 + 6 = 10$ [$\textsc{Sum}(4) = 10$]

Iterative vs. recursive algorithms

iterative	recursive

Sum(n)
  $total\_sum := 0$
  FOR ($i = 1$ to n) DO
    $total\_sum := total\_sum + i$
  PRINT $total\_sum$
	Sum(int n)
	  IF($n = 1$) THEN RETURN 1
	  RETURN $n +$ SUM($n - 1$)

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

Runtime iterative SUM: $\Theta(n)$ [Exercise]

Part 1-3: Runtime of algorithms

Iterative vs. recursive algorithms

iterative	recursive

`Sum(n)`
  $total\_sum := 0$
  `FOR` $(i = 1$ to $n)$ `DO`
    $total\_sum := total\_sum + i$
  `PRINT` $total\_sum$
	`Sum`(int n)
	  `IF`$(n = 1)$ `THEN RETURN` 1
	  `RETURN` $n + \text{SUM}(n-1)$

What are these algorithms doing? **Answer:** `Sum` computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

Runtime iterative SUM: $\Theta(n)$ [Exercise]

Runtime recursive SUM:

$$T(n) = \Theta(1) + T(n-1)$$

Iterative vs. recursive algorithms

iterative	recursive
Sum(n) $total\_sum := 0$ FOR ($i = 1$ to n) DO $total\_sum := total\_sum + i$ PRINT $total\_sum$	Sum(int n) IF($n = 1$) THEN RETURN 1 RETURN $n +$ SUM($n - 1$)

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

Runtime iterative SUM: $\Theta(n)$ [Exercise]

Runtime recursive SUM:

$$
\begin{aligned}
T(n) &= \Theta(1) + T(n-1) \\
&= \Theta(1) + (\Theta(1) + T(n-2)
\end{aligned}
$$

Iterative vs. recursive algorithms

iterative	recursive

`Sum`(n)
  $total\_sum := 0$
  `FOR` ($i = 1$ to n) `DO`
    $total\_sum := total\_sum + i$
  `PRINT` $total\_sum$
	`Sum`(int n)
	  `IF`($n = 1$) `THEN RETURN` 1
	  `RETURN` $n +$ SUM$(n - 1)$

What are these algorithms doing? **Answer:** `Sum` computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

Runtime iterative SUM: $\Theta(n)$ [Exercise]

Runtime recursive SUM:

$$
\begin{aligned}
T(n) &= \Theta(1) + T(n-1) \\
&= \Theta(1) + (\Theta(1) + T(n-2) = 2 \cdot \Theta(1) + T(n-2)
\end{aligned}
$$

Iterative vs. recursive algorithms

iterative	recursive

Sum(n)
  $total\_sum := 0$
  FOR ($i = 1$ to n) DO
    $total\_sum := total\_sum + i$
  PRINT $total\_sum$
	Sum(int n)
	  IF($n = 1$) THEN RETURN 1
	  RETURN $n +$ SUM($n - 1$)

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

Runtime iterative SUM: $\Theta(n)$ [Exercise]

Runtime recursive SUM:

$$
\begin{aligned}
T(n) &= \Theta(1) + T(n-1) \\
&= \Theta(1) + (\Theta(1) + T(n-2) = 2 \cdot \Theta(1) + T(n-2) \\
&= \ldots
\end{aligned}
$$

Iterative vs. recursive algorithms

iterative	recursive

Sum(n)
  $total\_sum := 0$
  FOR ($i = 1$ to n) DO
    $total\_sum := total\_sum + i$
  PRINT $total\_sum$
	Sum(int n)
	  IF($n = 1$) THEN RETURN 1
	  RETURN $n +$ SUM$(n - 1)$

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

Runtime iterative SUM: $\Theta(n)$ [Exercise]

Runtime recursive SUM:

$$
\begin{aligned}
T(n) &= \Theta(1) + T(n-1) \\
&= \Theta(1) + (\Theta(1) + T(n-2) = 2 \cdot \Theta(1) + T(n-2) \\
&= \dots \\
&= (n-1)\Theta(1) + T(1)
\end{aligned}
$$

Iterative vs. recursive algorithms

iterative	recursive

Sum(n)
  $total\_sum := 0$
  FOR ($i = 1$ to n) DO
    $total\_sum := total\_sum + i$
  PRINT $total\_sum$
	Sum(int n)
	  IF($n = 1$) THEN RETURN 1
	  RETURN $n + \text{Sum}(n - 1)$

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

Runtime iterative Sum: $\Theta(n)$ [Exercise]

Runtime recursive Sum:

$$
\begin{aligned}
T(n) &= \Theta(1) + T(n-1) \\
&= \Theta(1) + (\Theta(1) + T(n-2) = 2 \cdot \Theta(1) + T(n-2) \\
&= \ldots \\
&= (n-1)\Theta(1) + T(1) \in \Theta(n) \text{ since } T(1) \in \Theta(1)
\end{aligned}
$$

Part 1-3: Runtime of algorithms

Often recurrences come in the form

$$T(n) = aT(n/b) + f(n)$$

with constants $a \geq 1$ and $b > 1$.

$n \in \mathbb{N}_{\geq 1}$ is the input size

a is the number of subproblems in the recursion

n/b is the size of a single subproblem and means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

$f(n)$ denotes the cost incurred by dividing the problem and combining the partial solutions

Part 1-3: Runtime of algorithms

Often recurrences come in the form

$$T(n) = aT(n/b) + f(n)$$

with constants $a \geq 1$ and $b > 1$.

$n \in \mathbb{N}_{\geq 1}$ is the input size

a is the number of subproblems in the recursion

n/b is the size of a single subproblem and means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

$f(n)$ denotes the cost incurred by dividing the problem and combining the partial solutions

Part 1-3: Runtime of algorithms

Often recurrences come in the form

$$T(n) = aT(n/b) + f(n)$$

with constants $a \geq 1$ and $b > 1$.

$n \in \mathbb{N}_{\geq 1}$ is the input size

a is the number of subproblems in the recursion

n/b is the size of a single subproblem and means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

$f(n)$ denotes the cost incurred by dividing the problem and combining the partial solutions

Part 1-3: Runtime of algorithms

Often recurrences come in the form

$$T(n) = aT(n/b) + f(n)$$

with constants $a \geq 1$ and $b > 1$.

$n \in \mathbb{N}_{\geq 1}$ is the input size

a is the number of subproblems in the recursion

n/b is the size of a single subproblem and means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

$f(n)$ denotes the cost incurred by dividing the problem and combining the partial solutions

$\lceil x \rceil$ denotes the least integer greater than or equal to x
$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Part 1-3: Runtime of algorithms

Often recurrences come in the form

$$T(n) = aT(n/b) + f(n)$$

with constants $a \geq 1$ and $b > 1$.

$n \in \mathbb{N}_{\geq 1}$ is the input size

a is the number of subproblems in the recursion

n/b is the size of a single subproblem and means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

$f(n)$ denotes the cost incurred by dividing the problem and combining the partial solutions

$\lceil x \rceil$ denotes the least integer greater than or equal to x
$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Part 1-3: Runtime of algorithms

Often recurrences come in the form

$$T(n) = aT(n/b) + f(n)$$

with constants $a \geq 1$ and $b > 1$.

$n \in \mathbb{N}_{\geq 1}$ is the input size

a is the number of subproblems in the recursion

n/b is the size of a single subproblem and means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

$f(n)$ denotes the cost incurred by dividing the problem and combining the partial solutions

Here, we assume that $f(n) = \Theta(n^d)$ for some $d \geq 0$.

$\lceil x \rceil$ denotes the least integer greater than or equal to x
$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Part 1-3: Runtime of algorithms

Often recurrences come in the form

$$T(n) = aT(n/b) + f(n)$$

with constants $a \geq 1$ and $b > 1$.

$n \in \mathbb{N}_{\geq 1}$ is the input size

a is the number of subproblems in the recursion

n/b is the size of a single subproblem and means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

$f(n)$ denotes the cost incurred by dividing the problem and combining the partial solutions

Here, we assume that $f(n) = \Theta(n^d)$ for some $d \geq 0$.

Example:

```
Some_Rec(n)
  IF (n > 1) THEN
     someTask
     Some_Rec(n/3) + Some_Rec(n/3)
```

Suppose someTask has runtime in $\Theta(n^5)$

$\implies T(n) = 2T(n/3) + \Theta(n^5)$, $a = 2, b = 3, d = 5$

$\lceil x \rceil$ denotes the least integer greater than or equal to x
$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Example:

```
Some_Rec(n)
  IF (n > 1) THEN
     someTask
     Some_Rec(n/3) + Some_Rec(n/3)
```

Suppose `someTask` has runtime in $\Theta(n^5)$

$\Longrightarrow \; T(n) = 2T(n/3) + \Theta(n^5)$, $a = 2, b = 3, d = 5$

$\Longrightarrow \; a < b^d \; \Longrightarrow \; T(n) = \Theta(n^5)$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Example:

```
Halve(n)
    IF (n > 1) THEN Halve(n/2)
```

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Example:

```
Halve(n)
    IF (n > 1) THEN Halve(n/2)
```

Runtime without Master Theorem: $O(\log_2(n))$ (similar arguments as for `Halve` above with `WHILE`-loop)

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Example:

```
Halve(n)
    IF (n > 1) THEN Halve(n/2)
```

Runtime without Master Theorem: $O(\log_2(n))$ (similar arguments as for `Halve` above with `WHILE`-loop)

With Master Theorem: $T(n) = T(n/2) + \Theta(1)$

$\implies a = 1, b = 2, d = 0$

In formula above: $1 = a = b^d = 2^0$ and thus, runtime is in $\Theta(n^d \log_2 n) = \Theta(n^0 \log_2 n) = \Theta(\log_2 n)$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Further examples: Assume that $d = 2$ and $b = 3$:

$a = 8$: $T(n) = 8T(\frac{n}{3}) + \Theta(n^2) \xrightarrow{8<3^2} T(n) = \Theta(n^2)$

$a = 9$: $T(n) = 9T(\frac{n}{3}) + \Theta(n^2) \xrightarrow{9=3^2} T(n) = \Theta(n^2 \log_2 n)$

$a = 10$: $T(n) = 10T(\frac{n}{3}) + \Theta(n^2) \xrightarrow{10>3^2} T(n) = \Theta(n^{log_3(10)})$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b \\ &= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2 T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2 \\ &= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3 T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d \\ &= \ldots \\ &= a^k T(\tfrac{n}{b^k}) + \textstyle\sum_{j=0}^{k-1} a^j \left(\tfrac{n}{b^j}\right)^d \quad \text{//by induction (exercise)} \\ &= a^k T(\tfrac{n}{b^k}) + n^d \textstyle\sum_{j=0}^{k-1} \left(\tfrac{a}{b^d}\right)^j \end{aligned}$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$

$$T(n) = aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b$$

$$= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2 T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2$$

$$= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3 T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d$$

$$= \ldots$$

$$= a^k T(\tfrac{n}{b^k}) + \sum_{j=0}^{k-1} a^j \left(\tfrac{n}{b^j}\right)^d \quad \text{//by induction (exercise)}$$

$$= a^k T(\tfrac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\tfrac{a}{b^d}\right)^j$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$

$$T(n) = aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b$$

$$= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2$$

$$= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d$$

$$= \dots$$

$$= a^kT(\tfrac{n}{b^k}) + \sum_{j=0}^{k-1} a^j \left(\tfrac{n}{b^j}\right)^d \quad \text{//by induction (exercise)}$$

$$= a^kT(\tfrac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\tfrac{a}{b^d}\right)^j$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$

$$T(n) = aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b$$

$$= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2$$

$$= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d$$

$$= \dots$$

$$= a^kT(\tfrac{n}{b^k}) + \textstyle\sum_{j=0}^{k-1} a^j \left(\frac{n}{b^j}\right)^d \quad \text{//by induction (exercise)}$$

$$= a^kT(\tfrac{n}{b^k}) + n^d \textstyle\sum_{j=0}^{k-1} \left(\frac{a}{b^d}\right)^j$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$

$$T(n) = aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b$$

$$= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2 T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2$$

$$= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3 T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d$$

$$= \dots$$

$$= a^k T(\tfrac{n}{b^k}) + \sum_{j=0}^{k-1} a^j \left(\tfrac{n}{b^j}\right)^d \quad \text{//by induction (exercise)}$$

$$= a^k T(\tfrac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\tfrac{a}{b^d}\right)^j$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$

$$T(n) = aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b$$

$$= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2 T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2$$

$$= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3 T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d$$

$$= \ldots$$

$$= a^k T(\tfrac{n}{b^k}) + \sum_{j=0}^{k-1} a^j \left(\tfrac{n}{b^j}\right)^d \quad \text{//by induction (exercise)}$$

$$= a^k T(\tfrac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\tfrac{a}{b^d}\right)^j$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$

$$T(n) = aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b$$

$$= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2 T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2$$

$$= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3 T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d$$

$$= \ldots$$

$$= a^k T(\tfrac{n}{b^k}) + \sum_{j=0}^{k-1} a^j \left(\tfrac{n}{b^j}\right)^d \quad \text{//by induction (exercise)}$$

$$= a^k T(\tfrac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\tfrac{a}{b^d}\right)^j$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$

$$T(n) = aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b$$

$$= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2$$

$$= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d$$

$$= \ldots$$

$$= a^kT(\tfrac{n}{b^k}) + \sum_{j=0}^{k-1} a^j \left(\tfrac{n}{b^j}\right)^d \quad \text{//by induction (exercise)}$$

$$= a^kT(\tfrac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\tfrac{a}{b^d}\right)^j$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b \\ &= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2 T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2 \\ &= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3 T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d \\ &= \ldots \\ &= a^k T(\tfrac{n}{b^k}) + \textstyle\sum_{j=0}^{k-1} a^j \left(\tfrac{n}{b^j}\right)^d \quad \text{//by induction (exercise)} \\ &= a^k T(\tfrac{n}{b^k}) + n^d \textstyle\sum_{j=0}^{k-1} \left(\tfrac{a}{b^d}\right)^j \end{aligned}$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = a^k T(\frac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\frac{a}{b^d}\right)^j$ for all $k \geq 1$

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b \\ &= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2 T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2 \\ &= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3 T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d \\ &= \ldots \\ &= a^k T(\tfrac{n}{b^k}) + \textstyle\sum_{j=0}^{k-1} a^j \left(\frac{n}{b^j}\right)^d \quad \text{//by induction (exercise)} \\ &= a^k T(\tfrac{n}{b^k}) + n^d \textstyle\sum_{j=0}^{k-1} \left(\frac{a}{b^d}\right)^j \end{aligned}$$

Part 1-3: Runtime of algorithms

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = a^k T(\frac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\frac{a}{b^d}\right)^j$ for all $k \geq 1$

$T(1)$ terminates.

Hence, it terminates for k for which $T(1) = T(\frac{n}{b^k})$ holds

$\iff 1 = \frac{n}{b^k} \iff b^k = n \iff k = \log_b(n)$

Hence, we can write: $T(n) = a^{\log_b(n)} T(\frac{n}{b^{\log_b(n)}}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $T(\frac{n}{b^{\log_b(n)}}) = T(1)$ we have: $T(n) = \Theta(a^{\log_b(n)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $a^{\log_b(n)} = n^{\log_b(a)}$ (exercise!), we have: $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Part 1-3: Runtime of algorithms

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = a^k T(\frac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\frac{a}{b^d}\right)^j$ for all $k \geq 1$

$T(1)$ terminates.

Hence, it terminates for k for which $T(1) = T(\frac{n}{b^k})$ holds

$\iff 1 = \frac{n}{b^k} \iff b^k = n \iff k = \log_b(n)$

Hence, we can write: $T(n) = a^{\log_b(n)} T(\frac{n}{b^{\log_b(n)}}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $T(\frac{n}{b^{\log_b(n)}}) = T(1)$ we have: $T(n) = \Theta(a^{\log_b(n)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $a^{\log_b(n)} = n^{\log_b(a)}$ (exercise!), we have: $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = a^k T(\frac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\frac{a}{b^d}\right)^j$ for all $k \geq 1$

$T(1)$ terminates.

Hence, it terminates for k for which $T(1) = T(\frac{n}{b^k})$ holds

$$\iff 1 = \frac{n}{b^k} \iff b^k = n \iff k = \log_b(n)$$

Hence, we can write: $T(n) = a^{\log_b(n)} T(\frac{n}{b^{\log_b(n)}}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $T(\frac{n}{b^{\log_b(n)}}) = T(1)$ we have: $T(n) = \Theta(a^{\log_b(n)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $a^{\log_b(n)} = n^{\log_b(a)}$ (exercise!), we have: $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = a^k T(\frac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\frac{a}{b^d}\right)^j$ for all $k \geq 1$

$T(1)$ terminates.

Hence, it terminates for k for which $T(1) = T(\frac{n}{b^k})$ holds

$\iff 1 = \frac{n}{b^k} \iff b^k = n \iff k = \log_b(n)$

Hence, we can write: $T(n) = a^{\log_b(n)} T(\frac{n}{b^{\log_b(n)}}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $T(\frac{n}{b^{\log_b(n)}}) = T(1)$ we have: $T(n) = \Theta(a^{\log_b(n)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $a^{\log_b(n)} = n^{\log_b(a)}$ (exercise!), we have: $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = a^k T(\frac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\frac{a}{b^d}\right)^j$ for all $k \geq 1$

$T(1)$ terminates.

Hence, it terminates for k for which $T(1) = T(\frac{n}{b^k})$ holds

$\iff 1 = \frac{n}{b^k} \iff b^k = n \iff k = \log_b(n)$

Hence, we can write: $T(n) = a^{\log_b(n)} T(\frac{n}{b^{\log_b(n)}}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $T(\frac{n}{b^{\log_b(n)}}) = T(1)$ we have: $T(n) = \Theta(a^{\log_b(n)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $a^{\log_b(n)} = n^{\log_b(a)}$ (exercise!), we have: $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

$T(1)$ terminates.

Hence, it terminates for k for which $T(1) = T(\frac{n}{b^k})$ holds

$\iff 1 = \frac{n}{b^k} \iff b^k = n \iff k = \log_b(n)$

Hence, we can write: $T(n) = a^{\log_b(n)} T(\frac{n}{b^{\log_b(n)}}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $T(\frac{n}{b^{\log_b(n)}}) = T(1)$ we have: $T(n) = \Theta(a^{\log_b(n)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $a^{\log_b(n)} = n^{\log_b(a)}$ (exercise!), we have: $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

We consider now the three cases: $a < b^d$, $a = b^d$ and $a > b^d$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$:

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \geq \left(\frac{a}{b^d}\right)^0 = 1 \implies \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Omega(1)$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. series}}{=} \frac{1}{1-\frac{a}{b^d}} = O(1)$ //since $\frac{a}{b^d} \in (0,1)$ and constant

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \geq \left(\frac{a}{b^d}\right)^0 = 1 \implies \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Omega(1)$$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. series}}{=} \frac{1}{1-\frac{a}{b^d}} = O(1) \text{ //since } \frac{a}{b^d} \in (0,1) \text{ and constant}$$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \geq \left(\frac{a}{b^d}\right)^0 = 1 \implies \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Omega(1)$$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. series}}{=} \frac{1}{1-\frac{a}{b^d}} = O(1) \text{ //since } \frac{a}{b^d} \in (0,1) \text{ and constant}$$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \geq \left(\frac{a}{b^d}\right)^0 = 1 \implies \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Omega(1)$$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. series}}{=} \frac{1}{1-\frac{a}{b^d}} = O(1) \text{ //since } \frac{a}{b^d} \in (0,1) \text{ and constant}$$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \geq \left(\frac{a}{b^d}\right)^0 = 1 \implies \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Omega(1)$$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. series}}{=} \frac{1}{1-\frac{a}{b^d}} = O(1) \text{ //since } \tfrac{a}{b^d} \in (0,1) \text{ and constant}$$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \geq \left(\frac{a}{b^d}\right)^0 = 1 \implies \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Omega(1)$$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. series}}{=} \frac{1}{1-\frac{a}{b^d}} = O(1)$ //since $\frac{a}{b^d} \in (0,1)$ and constant

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \geq \left(\frac{a}{b^d}\right)^0 = 1 \implies \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Omega(1)$$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. series}}{=} \frac{1}{1-\frac{a}{b^d}} = O(1)$$ //since $\frac{a}{b^d} \in (0,1)$ and constant

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$: $\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \geq \left(\frac{a}{b^d}\right)^0 = 1 \implies \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Omega(1)$$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. series}}{=} \frac{1}{1-\frac{a}{b^d}} = O(1)$$ //since $\frac{a}{b^d} \in (0,1)$ and constant

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$: $\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$

$$T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$$

$$= \Theta(n^{\log_b(a)}) + n^d \Theta(1)$$

$$= \Theta(n^{\log_b(a)} + n^d)$$

Since $a < b^d$ we have $\log_b(a) \leq \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d + n^d) = \Theta(n^d)$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$: $\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$

$$T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$$

$$= \Theta(n^{\log_b(a)}) + n^d \Theta(1)$$

$$= \Theta(n^{\log_b(a)} + n^d)$$

Since $a < b^d$ we have $\log_b(a) \leq \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d + n^d) = \Theta(n^d)$ as desired

Part 1-3: Runtime of algorithms

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$: $\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$

$$T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$$

$$= \Theta(n^{\log_b(a)}) + n^d \Theta(1)$$

$$= \Theta(n^{\log_b(a)} + n^d)$$

Since $a < b^d$ we have $\log_b(a) \leq \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d + n^d) = \Theta(n^d)$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$: $\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$

$$T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$$

$$= \Theta(n^{\log_b(a)}) + n^d \Theta(1)$$

$$= \Theta(n^{\log_b(a)} + n^d)$$

Since $a < b^d$ we have $\log_b(a) \leq \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d + n^d) = \Theta(n^d)$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$: $\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$

$$T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$$

$$= \Theta(n^{\log_b(a)}) + n^d \Theta(1)$$

$$= \Theta(n^{\log_b(a)} + n^d)$$

Since $a < b^d$ we have $\log_b(a) \leq \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d + n^d) = \Theta(n^d)$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a < b^d$: $\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$

$$\begin{aligned} T(n) &= \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \\ &= \Theta(n^{\log_b(a)}) + n^d \Theta(1) \\ &= \Theta(n^{\log_b(a)} + n^d) \end{aligned}$$

Since $a < b^d$ we have $\log_b(a) \leq \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d + n^d) = \Theta(n^d)$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a = b^d$: $\iff \frac{a}{b^d} = 1$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b(n)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \log_b(n)$

Since $a = b^d$ we have $\log_b(a) = \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d) + n^d \log_b(n) = \Theta(n^d \log_b(n))$

Since $\log_2(n) = \frac{\log_b(n)}{\log_b(2)}$ and $\log_b(2)$ is constant, we have $\Theta(\log_b(n)) = \Theta(\log_2(n))$ and thus,

$T(n) = \Theta(n^d \log_2(n))$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a = b^d$: $\iff \frac{a}{b^d} = 1$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b(n)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \log_b(n)$

Since $a = b^d$ we have $\log_b(a) = \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d) + n^d \log_b(n) = \Theta(n^d \log_b(n))$

Since $\log_2(n) = \frac{\log_b(n)}{\log_b(2)}$ and $\log_b(2)$ is constant, we have $\Theta(\log_b(n)) = \Theta(\log_2(n))$ and thus,

$T(n) = \Theta(n^d \log_2(n))$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a = b^d$: $\iff \frac{a}{b^d} = 1$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b(n)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \log_b(n)$

Since $a = b^d$ we have $\log_b(a) = \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d) + n^d \log_b(n) = \Theta(n^d \log_b(n))$

Since $\log_2(n) = \frac{\log_b(n)}{\log_b(2)}$ and $\log_b(2)$ is constant, we have $\Theta(\log_b(n)) = \Theta(\log_2(n))$ and thus,

$T(n) = \Theta(n^d \log_2(n))$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a = b^d$: $\iff \frac{a}{b^d} = 1$

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b(n)$$

$$T(n) = \Theta(n^{\log_b(a)}) + n^d \log_b(n)$$

Since $a = b^d$ we have $\log_b(a) = \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d) + n^d \log_b(n) = \Theta(n^d \log_b(n))$

Since $\log_2(n) = \frac{\log_b(n)}{\log_b(2)}$ and $\log_b(2)$ is constant, we have $\Theta(\log_b(n)) = \Theta(\log_2(n))$ and thus,

$T(n) = \Theta(n^d \log_2(n))$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a = b^d$: $\iff \frac{a}{b^d} = 1$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b(n)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \log_b(n)$

Since $a = b^d$ we have $\log_b(a) = \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d) + n^d \log_b(n) = \Theta(n^d \log_b(n))$

Since $\log_2(n) = \frac{\log_b(n)}{\log_b(2)}$ and $\log_b(2)$ is constant, we have $\Theta(\log_b(n)) = \Theta(\log_2(n))$ and thus,

$T(n) = \Theta(n^d \log_2(n))$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a = b^d$: $\iff \frac{a}{b^d} = 1$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b(n)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \log_b(n)$

Since $a = b^d$ we have $\log_b(a) = \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d) + n^d \log_b(n) = \Theta(n^d \log_b(n))$

Since $\log_2(n) = \frac{\log_b(n)}{\log_b(2)}$ and $\log_b(2)$ is constant, we have $\Theta(\log_b(n)) = \Theta(\log_2(n))$ and thus,

$T(n) = \Theta(n^d \log_2(n))$ as desired

Part 1-3: Runtime of algorithms

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a = b^d$: $\iff \frac{a}{b^d} = 1$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b(n)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \log_b(n)$

Since $a = b^d$ we have $\log_b(a) = \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d) + n^d \log_b(n) = \Theta(n^d \log_b(n))$

Since $\log_2(n) = \frac{\log_b(n)}{\log_b(2)}$ and $\log_b(2)$ is constant, we have $\Theta(\log_b(n)) = \Theta(\log_2(n))$ and thus,

$T(n) = \Theta(n^d \log_2(n))$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a = b^d$: $\iff \frac{a}{b^d} = 1$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b(n)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \log_b(n)$

Since $a = b^d$ we have $\log_b(a) = \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d) + n^d \log_b(n) = \Theta(n^d \log_b(n))$

Since $\log_2(n) = \frac{\log_b(n)}{\log_b(2)}$ and $\log_b(2)$ is constant, we have $\Theta(\log_b(n)) = \Theta(\log_2(n))$ and thus,

$T(n) = \Theta(n^d \log_2(n))$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a = b^d$: $\iff \frac{a}{b^d} = 1$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b(n)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \log_b(n)$

Since $a = b^d$ we have $\log_b(a) = \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d) + n^d \log_b(n) = \Theta(n^d \log_b(n))$

Since $\log_2(n) = \frac{\log_b(n)}{\log_b(2)}$ and $\log_b(2)$ is constant, we have $\Theta(\log_b(n)) = \Theta(\log_2(n))$ and thus,

$T(n) = \Theta(n^d \log_2(n))$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a > b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. sum}}{=} \frac{\left(\frac{a}{b^d}\right)^{\log_b(n)-1} - 1}{\frac{a}{b^d} - 1} \overset{\text{exerc.}}{=} \Theta\left(\left(\frac{a}{b^d}\right)^{\log_b(n)}\right) \overset{\text{exerc.(log-rules)}}{=} \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right)$$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right) = \Theta(n^{\log_b(a)})$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a > b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. sum}}{=} \frac{\left(\frac{a}{b^d}\right)^{\log_b(n)-1} - 1}{\frac{a}{b^d} - 1} \overset{\text{exerc.}}{=} \Theta\left(\left(\frac{a}{b^d}\right)^{\log_b(n)}\right) \overset{\text{exerc.(log-rules)}}{=} \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right)$$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right) = \Theta(n^{\log_b(a)})$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a > b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. sum}}{=} \frac{\left(\frac{a}{b^d}\right)^{\log_b(n)-1} - 1}{\frac{a}{b^d} - 1} \overset{\text{exerc.}}{=} \Theta\left(\left(\frac{a}{b^d}\right)^{\log_b(n)}\right) \overset{\text{exerc.(log-rules)}}{=} \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right)$$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right) = \Theta(n^{\log_b(a)})$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a > b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. sum}}{=} \frac{\left(\frac{a}{b^d}\right)^{\log_b(n)-1} - 1}{\frac{a}{b^d} - 1} \overset{\text{exerc.}}{=} \Theta\left(\left(\frac{a}{b^d}\right)^{\log_b(n)}\right) \overset{\text{exerc.(log-rules)}}{=} \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right)$$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right) = \Theta(n^{\log_b(a)})$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a > b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. sum}}{=} \frac{\left(\frac{a}{b^d}\right)^{\log_b(n)-1} - 1}{\frac{a}{b^d} - 1} \overset{\text{exerc.}}{=} \Theta\left(\left(\frac{a}{b^d}\right)^{\log_b(n)}\right) \overset{\text{exerc.(log-rules)}}{=} \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right)$$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right) = \Theta(n^{\log_b(a)})$ as desired

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a > b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. sum}}{=} \frac{\left(\frac{a}{b^d}\right)^{\log_b(n)-1} - 1}{\frac{a}{b^d} - 1} \overset{\text{exerc.}}{=} \Theta\left(\left(\frac{a}{b^d}\right)^{\log_b(n)}\right) \overset{\text{exerc.(log-rules)}}{=} \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right)$$

$$T(n) = \Theta(n^{\log_b(a)}) + n^d \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right) = \Theta(n^{\log_b(a)}) \text{ as desired}$$

Master Theorem [simplified version]

Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$ and we have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Case $a > b^d$:

$$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. sum}}{=} \frac{\left(\frac{a}{b^d}\right)^{\log_b(n)-1} - 1}{\frac{a}{b^d} - 1} \overset{\text{exerc.}}{=} \Theta\left(\left(\frac{a}{b^d}\right)^{\log_b(n)}\right) \overset{\text{exerc.(log-rules)}}{=} \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right)$$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \Theta\left(\frac{n^{\log_b(a)}}{n^d}\right) = \Theta(n^{\log_b(a)})$ as desired

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Some_Sum(int x, y, z)
  int r = x + y + z
  RETURN r
```

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Some_Sum(int x, y, z)
   int r = x + y + z
   RETURN r
```

Requires 3 units of space for the parameters x, y, z and 1 for the local variable r.

Space complexity is in $O(1)$

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Sum(array a of length n)
  int r = 0
  FOR (i = 1 to n) DO r := r + a[i]
  RETURN r
```

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Sum(array a of length n)
  int r = 0
  FOR (i = 1 to n) DO r := r + a[i]
  RETURN r
```

Requires n units of space for array a and 2 for the local variables r and i.

Space complexity is in $O(n)$

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Fact_iter(int n)
  int fac = 1
  FOR (i = 1 to n) DO
      fac := fac · i
  RETURN fac
```

```
Fact_rec(int n)
  IF(n == 0 or n == 1) THEN
      RETURN 1
  ELSE
      RETURN n · Fact_rec(n − 1)
```

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Fact_iter(int n)
  int fac = 1
  FOR (i = 1 to n) DO
      fac := fac · i
  RETURN fac
```

requires 3 space units for the variables n, fac and i

$\Longrightarrow O(1)$ space.

```
Fact_rec(int n)
  IF(n == 0 or n == 1) THEN
      RETURN 1
  ELSE
      RETURN n · Fact_rec(n − 1)
```

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Fact_iter(int n)
  int fac = 1
  FOR (i = 1 to n) DO
      fac := fac · i
  RETURN fac
```

requires 3 space units for the variables n, `fac` and i

$\Longrightarrow$ $O(1)$ space.

```
Fact_rec(int n)
  IF(n == 0 or n == 1) THEN
      RETURN 1
  ELSE
      RETURN n · Fact_rec(n − 1)
```

requires 1 space units for the variable n

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Fact_iter(int n)
  int fac = 1
  FOR (i = 1 to n) DO
      fac := fac · i
  RETURN fac
```

requires 3 space units for the variables n, fac and i

$\Longrightarrow$ $O(1)$ space.

```
Fact_rec(int n)
  IF(n == 0 or n == 1) THEN
      RETURN 1
  ELSE
      RETURN n · Fact_rec(n − 1)
```

requires 1 space units for the variable n

Now, examine the extra space that is taken by the algorithm temporarily to finish its work [auxiliary space]:

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Fact_iter(int n)
  int fac = 1
  FOR (i = 1 to n) DO
      fac := fac · i
  RETURN fac
```

requires 3 space units for the variables n, fac and i

$\Longrightarrow$ $O(1)$ space.

```
Fact_rec(int n)
  IF(n == 0 or n == 1) THEN
      RETURN 1
  ELSE
      RETURN n · Fact_rec(n − 1)
```

requires 1 space units for the variable n

Now, examine the extra space that is taken by the algorithm temporarily to finish its work [auxiliary space]:

for n the return-value Fact_rec$(n-1)$ must temporarily be stored
for $n-1$ the return-value Fact_rec$(n-2)$ must temporarily be stored

⋮

for 2 the return-value Fact_rec(1) must temporarily be stored
for 1 the return-value is 1

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Fact_iter(int n)
  int fac = 1
  FOR (i = 1 to n) DO
      fac := fac · i
  RETURN fac
```

requires 3 space units for the variables n, fac and i

$\implies O(1)$ space.

```
Fact_rec(int n)
  IF(n == 0 or n == 1) THEN
      RETURN 1
  ELSE
      RETURN n · Fact_rec(n − 1)
```

requires 1 space units for the variable n

Now, examine the extra space that is taken by the algorithm temporarily to finish its work [auxiliary space]:

for n the return-value Fact_rec$(n-1)$ must temporarily be stored
for $n-1$ the return-value Fact_rec$(n-2)$ must temporarily be stored

$\vdots$

for 2 the return-value Fact_rec(1) must temporarily be stored
for 1 the return-value is 1

At this point the values can be used to compute Fact_rec(n) and we temporarily stored $n-1 = O(n)$ variables.

$\implies O(n)$ space

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

But be careful here: If things are passed by pointer or reference, then space is shared [later].

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

But be careful here: If things are passed by pointer or reference, then space is shared [later].

We mainly focus here on time complexity

Part 1-3: Time Complexity & Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

But be careful here: If things are passed by pointer or reference, then space is shared [later].

We mainly focus here on time complexity

Side Note:
During each time step, you can only access one memory location. Therefore you can never access more memory locations than you have time
$\implies$ space complexity is bounded by time complexity

Does runtime matter?

	insertion-sort	merge-sort [later]
runtime	$O(n^2)$	$O(n\log_2(n))$

Should we care about factor n vs $\log_2(n)$?

For large enough n and constant c, we have

	insertion-sort	merge-sort
runtime	$c \cdot n^2$	$c \cdot n\log_2(n)$

Say $c = 100$ and $n = 10^7$ (e.g. list of population in Sweden). Suppose we have a computer that can perform $10^9 op/s$ where op/s = operations per seconds.

insertion-sort: $\frac{100\cdot(10^7)^2 op}{10^9 op/s} = \frac{10^{16} op}{10^9 op/s} = 10^7 s$

merge-sort: $\frac{100\cdot 10^7 \cdot \log_2(10^7) op}{10^9 op/s} = \frac{10^9 \cdot \log_2(10^7) op}{10^9 op/s} = \log_2(10^7) s$

Runtime matters !

Does runtime matter?

	insertion-sort	merge-sort [later]
runtime	$O(n^2)$	$O(n\log_2(n))$

Should we care about factor n vs $\log_2(n)$?

For large enough n and constant c, we have

	insertion-sort	merge-sort
runtime	$c \cdot n^2$	$c \cdot n\log_2(n)$

Say $c = 100$ and $n = 10^7$ (e.g. list of population in Sweden). Suppose we have a computer that can perform $10^9 op/s$ where op/s = operations per seconds.

insertion-sort: $\frac{100\cdot(10^7)^2 op}{10^9 op/s} = \frac{10^{16} op}{10^9 op/s} = 10^7 s$

merge-sort: $\frac{100\cdot 10^7\cdot\log_2(10^7) op}{10^9 op/s} = \frac{10^9\cdot\log_2(10^7) op}{10^9 op/s} = \log_2(10^7) s$

Runtime matters !

Does runtime matter?

	insertion-sort	merge-sort [later]
runtime	$O(n^2)$	$O(n\log_2(n))$

Should we care about factor n vs $\log_2(n)$?

For large enough n and constant c, we have

	insertion-sort	merge-sort
runtime	$c \cdot n^2$	$c \cdot n\log_2(n)$

Say $c = 100$ and $n = 10^7$ (e.g. list of population in Sweden). Suppose we have a computer that can perform $10^9\, op/s$ where op/s = operations per seconds.

insertion-sort: $\frac{100 \cdot (10^7)^2 op}{10^9\, op/s} = \frac{10^{16} op}{10^9\, op/s} = 10^7 s$

merge-sort: $\frac{100 \cdot 10^7 \cdot \log_2(10^7) op}{10^9\, op/s} = \frac{10^9 \cdot \log_2(10^7) op}{10^9\, op/s} = \log_2(10^7) s$

Runtime matters !

Does runtime matter?

	insertion-sort	merge-sort [later]
runtime	$O(n^2)$	$O(n\log_2(n))$

Should we care about factor n vs $\log_2(n)$?

For large enough n and constant c, we have

	insertion-sort	merge-sort
runtime	$c \cdot n^2$	$c \cdot n\log_2(n)$

Say $c = 100$ and $n = 10^7$ (e.g. list of population in Sweden). Suppose we have a computer that can perform $10^9 op/s$ where op/s = operations per seconds.

insertion-sort: $\frac{100 \cdot (10^7)^2 op}{10^9 op/s} = \frac{10^{16} op}{10^9 op/s} = 10^7 s$

merge-sort: $\frac{100 \cdot 10^7 \cdot \log_2(10^7) op}{10^9 op/s} = \frac{10^9 \cdot \log_2(10^7) op}{10^9 op/s} = \log_2(10^7) s$

Runtime matters !

Does runtime matter?

	insertion-sort	merge-sort [later]
runtime	$O(n^2)$	$O(n\log_2(n))$

Should we care about factor n vs $\log_2(n)$?

For large enough n and constant c, we have

	insertion-sort	merge-sort
runtime	$c \cdot n^2$	$c \cdot n\log_2(n)$

Say $c = 100$ and $n = 10^7$ (e.g. list of population in Sweden). Suppose we have a computer that can perform $10^9 op/s$ where op/s = operations per seconds.

insertion-sort: $\frac{100\cdot(10^7)^2 op}{10^9 op/s} = \frac{10^{16} op}{10^9 op/s} = 10^7 s$

merge-sort: $\frac{100\cdot 10^7\cdot\log_2(10^7) op}{10^9 op/s} = \frac{10^9\cdot\log_2(10^7) op}{10^9 op/s} = \log_2(10^7)s$

Runtime matters !

Part 1-3: Runtime

Does runtime matter?

	insertion-sort	merge-sort [later]
runtime	$O(n^2)$	$O(n \log_2(n))$

Should we care about factor n vs $\log_2(n)$?

For large enough n and constant c, we have

	insertion-sort	merge-sort
runtime	$c \cdot n^2$	$c \cdot n \log_2(n)$

Say $c = 100$ and $n = 10^7$ (e.g. list of population in Sweden). Suppose we have a computer that can perform $10^9 \, op/s$ where op/s = operations per seconds.

insertion-sort: $\frac{100 \cdot (10^7)^2 op}{10^9 op/s} = \frac{10^{16} op}{10^9 op/s} = 10^7 s$

merge-sort: $\frac{100 \cdot 10^7 \cdot \log_2(10^7) op}{10^9 op/s} = \frac{10^9 \cdot \log_2(10^7) op}{10^9 op/s} = \log_2(10^7) s$

Runtime matters !

Part 1-3: Runtime

Does runtime matter?

	insertion-sort	merge-sort [later]
runtime	$O(n^2)$	$O(n \log_2(n))$

Should we care about factor n vs $\log_2(n)$?

For large enough n and constant c, we have

	insertion-sort	merge-sort
runtime	$c \cdot n^2$	$c \cdot n \log_2(n)$

Say $c = 100$ and $n = 10^7$ (e.g. list of population in Sweden). Suppose we have a computer that can perform $10^9\, op/s$ where op/s = operations per seconds.

insertion-sort: $\frac{100 \cdot (10^7)^2 op}{10^9\, op/s} = \frac{10^{16} op}{10^9\, op/s} = 10^7 s \approx 115$ days

merge-sort: $\frac{100 \cdot 10^7 \cdot \log_2(10^7) op}{10^9\, op/s} = \frac{10^9 \cdot \log_2(10^7) op}{10^9\, op/s} = \log_2(10^7) s \approx 24s$

Runtime matters !

Part 1-3: Runtime

Does runtime matter?

	insertion-sort	merge-sort [later]
runtime	$O(n^2)$	$O(n\log_2(n))$

Should we care about factor n vs $\log_2(n)$?

For large enough n and constant c, we have

	insertion-sort	merge-sort
runtime	$c \cdot n^2$	$c \cdot n\log_2(n)$

Say $c = 100$ and $n = 10^7$ (e.g. list of population in Sweden). Suppose we have a computer that can perform $10^9 op/s$ where op/s = operations per seconds.

insertion-sort: $\frac{100 \cdot (10^7)^2 op}{10^9 op/s} = \frac{10^{16} op}{10^9 op/s} = 10^7 s \approx 115$ days

merge-sort: $\frac{100 \cdot 10^7 \cdot \log_2(10^7) op}{10^9 op/s} = \frac{10^9 \cdot \log_2(10^7) op}{10^9 op/s} = \log_2(10^7) s \approx 24s$

Runtime matters !

Part 1-4: Elementary Data Structures

Part 1-4: Elementary Data Structures

The right organizational form and choice of data structure significantly impact the efficiency of data operations.

Example:
Consider a phone book. There it is easy to find a phone number for a given name based on the alphabetical order.
What if we are interested in the reverse task (finding for a given number the name)?
Ideas?

The optimal choice of a data structure is not always obvious and one data structure might be very suitable for one task but not for some other

Determining an efficient data structure is usually influenced by the operations needed later on the data (searching, replacing, re-sorting, . . .)

Part 1-4: Elementary Data Structures

The right organizational form and choice of data structure significantly impact the efficiency of data operations.

Example:
Consider a phone book. There it is easy to find a phone number for a given name based on the alphabetical order.
What if we are interested in the reverse task (finding for a given number the name)?
Ideas?

The optimal choice of a data structure is not always obvious and one data structure might be very suitable for one task but not for some other

Determining an efficient data structure is usually influenced by the operations needed later on the data (searching, replacing, re-sorting, . . .)

The right organizational form and choice of data structure significantly impact the efficiency of data operations.

Example:
Consider a phone book. There it is easy to find a phone number for a given name based on the alphabetical order.
What if we are interested in the reverse task (finding for a given number the name)?
Ideas?

The optimal choice of a data structure is not always obvious and one data structure might be very suitable for one task but not for some other

Determining an efficient data structure is usually influenced by the operations needed later on the data (searching, replacing, re-sorting, ...)

The right organizational form and choice of data structure significantly impact the efficiency of data operations.

Example:
Consider a phone book. There it is easy to find a phone number for a given name based on the alphabetical order.
What if we are interested in the reverse task (finding for a given number the name)?
Ideas?

The optimal choice of a data structure is not always obvious and one data structure might be very suitable for one task but not for some other

Determining an efficient data structure is usually influenced by the operations needed later on the data (searching, replacing, re-sorting, . . .)

Part 1-4: Elementary Data Structures

The right organizational form and choice of data structure significantly impact the efficiency of data operations.

Example:
Consider a phone book. There it is easy to find a phone number for a given name based on the alphabetical order.
What if we are interested in the reverse task (finding for a given number the name)?
Ideas?

The optimal choice of a data structure is not always obvious and one data structure might be very suitable for one task but not for some other

Determining an efficient data structure is usually influenced by the operations needed later on the data (searching, replacing, re-sorting, . . .)

Part 1-4: Elementary Data Structures

The right organizational form and choice of data structure significantly impact the efficiency of data operations.

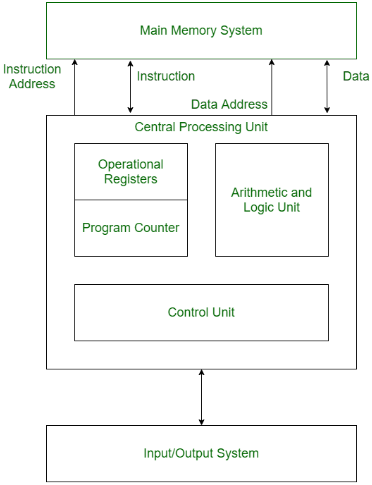
Example:
Consider a phone book. There it is easy to find a phone number for a given name based on the alphabetical order.
What if we are interested in the reverse task (finding for a given number the name)?
Ideas?

The optimal choice of a data structure is not always obvious and one data structure might be very suitable for one task but not for some other

Determining an efficient data structure is usually influenced by the operations needed later on the data (searching, replacing, re-sorting, . . .)

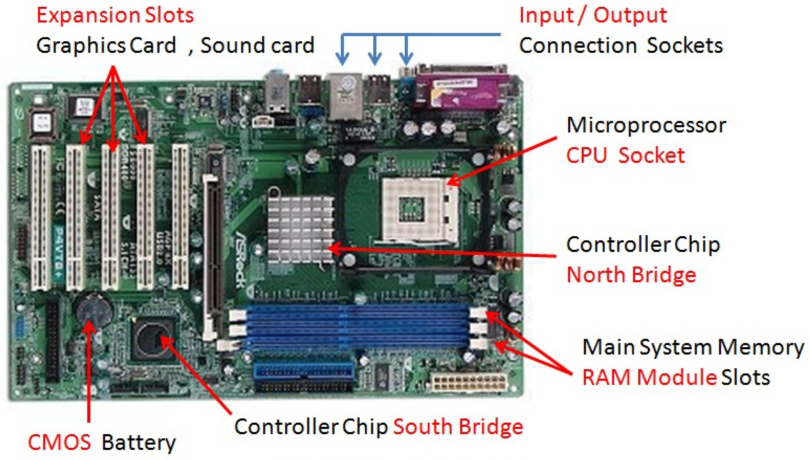
Part 1-4: Elementary Data Structures

How does memory work and how is this related to Data Structures?



Harvard Architecture

How does memory work and how is this related to Data Structures?



mainboard of a computer

Part 1-4: Elementary Data Structures

How does memory work and how is this related to Data Structures?

main memory or also RAM = Random Access Memory

How does memory work and how is this related to Data Structures?

Main memory consists of a number of regularly arranged memory cells, comparable to the compartments of a cabinet.

How does memory work and how is this related to Data Structures?

Since memory cells are regularly arranged, they can be numbered consecutively.
Each cell therefore has a unique number (=address).

How does memory work and how is this related to Data Structures?

All memory cells are the same size and can store a value (number, character, . . .).
This value is a fixed-length sequence of 0s and 1s (e.g. 1byte = 8 bits)

[8 bit per cell is pure convention (a few exceptions exist)].

How does memory work and how is this related to Data Structures?

The value stored in a cell represents some information (e.g. a number or a character)

How does memory work and how is this related to Data Structures?

But also "longer" information can be stored using "chunks of cells"

(e.g., the first 8bits of a 32bit integer *n* [to store *n* we need then 4 cells each of size 1byte]
or the first 3 characters of the alphabet)

How does memory work and how is this related to Data Structures?

But also "longer" information can be stored using "chunks of cells"

Difference between a 32-bit and a 64-bit architecture? *n*-bit architecure means that CPU can handle data in chunks of *n*-bit at a time. Thus, *n*-bit computer can can process data and perform calculations on numbers that are *n*-bits long.

32-bit system that can access 2^{32} (or 4,294,967,296) bytes of RAM. Meanwhile, a 64-bit processor can handle 2^{64} (or 18,446,744,073,709,551,616) bytes of RAM. In other words, a 64-bit processor can process more data than 4 billion 32-bit processors combined.

Part 1-4: Elementary Data Structures

How does memory work and how is this related to Data Structures?

The value can also be the *address of another memory cell*. In this case, we refer to it as a **pointer**.

How does memory work and how is this related to Data Structures?

The value can also be the *address of another memory cell*. In this case, we refer to it as a **pointer**.

A variable in (compiled) source-code refers to one or more consecutive cells in memory that store the "value/information" we assigned to this variable.

How does memory work and how is this related to Data Structures?

The value can also be the *address of another memory cell*. In this case, we refer to it as a **pointer**.

A variable in (compiled) source-code refers to one or more consecutive cells in memory that store the "value/information" we assigned to this variable.

Variables can thus contain values or be pointers to another variable.

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`

Adr.	.
	.
	.
6	–
7	–
8	–
	.
	.
80	–
	.

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"

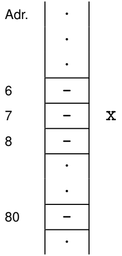
```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`	Memory "somewhat similar to what `Python` does"




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

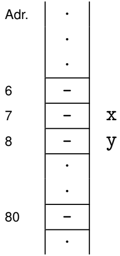
```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`	"somewhat similar to what `Python` does"




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory			C
Adr.	.		
	.		
	.		
6	–		
7	5	x	
8	–	y	
	.		
	.		
80	–		
	.		

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"

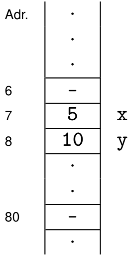
```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"

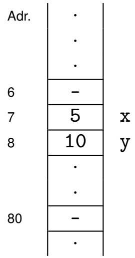
```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory — `C`




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory — "somewhat similar to what `Python` does"

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`

Adr.	Memory	
	.	
	.	
	.	
6	–	
7	5	x
8	10	y
	.	
	.	
80	–	
	.	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`

Adr.		
	·	
	·	
	·	
6	–	
7	5	x
8	10	y
	·	
	·	
80	–	
	·	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`

Adr.		
	·	
	·	
	·	
6	–	
7	10	x
8	10	y
	·	
	·	
80	–	
	·	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
As "5" is no longer used, memory cell 21 is freed up [garbage collector].
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`

Adr.		
	.	
	.	
	.	
6	–	
7	10	x
8	10	y
	.	
	.	
80	–	
	.	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

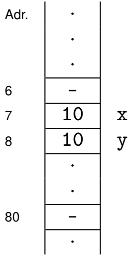
Memory "somewhat similar to what `Python` does"

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
As "5" is no longer used, memory cell 21 is freed up [garbage collector].
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Part 1-4: Elementary Data Structures

Storage of information in different languages (here as example `C` / `Python`)

Memory	C



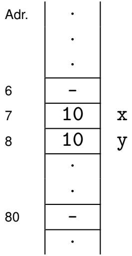

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory	"somewhat similar to what `Python` does"

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
As "5" is no longer used, memory cell 21 is freed up [garbage collector].
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

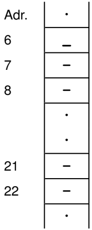
Storage of information in different languages (here as example `C` / `Python`)

Memory `C`




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"



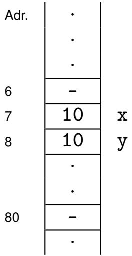

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

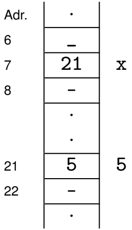
Storage of information in different languages (here as example `C` / `Python`)

Memory `C`




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"



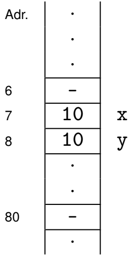

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

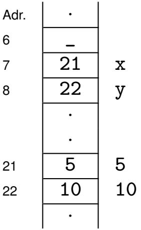
Storage of information in different languages (here as example `C` / `Python`)

Memory `C`




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"



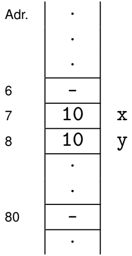

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

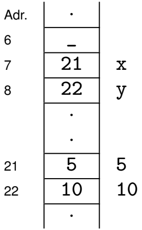
Storage of information in different languages (here as example `C` / `Python`)

Memory `C`




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"



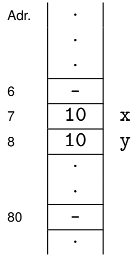

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

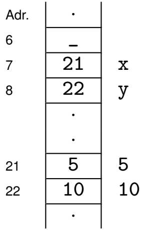
Storage of information in different languages (here as example `C` / `Python`)

Memory `C`




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"




```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example C / Python)

Memory C

Adr.		
	.	
	.	
	.	
6	–	
7	10	x
8	10	y
	.	
	.	
80	–	
	.	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what Python does"

Adr.	.	
6	–	
7	22	x
8	22	y
	.	
	.	
21	5	
22	10	10
	.	

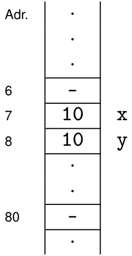
```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

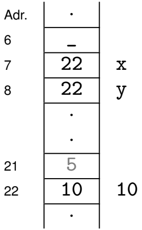
Storage of information in different languages (here as example `C` / `Python`)

Memory `C`




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"



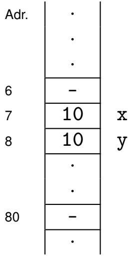

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

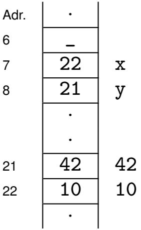
Storage of information in different languages (here as example `C` / `Python`)

Memory `C`




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"




```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory — `C`

Adr.		
	.	
	.	
	.	
6	–	
7	10	x
8	10	y
	.	
	.	
80	–	
	.	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory — "somewhat similar to what `Python` does"

Adr.		
	.	
6	–	
7	22	x
8	21	y
	.	
	.	
21	42	42
22	10	10
	.	

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

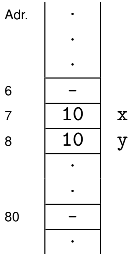
As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

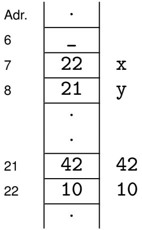
Part 1-4: Elementary Data Structures

Storage of information in different languages (here as example C / Python)

Memory C	Memory "somewhat similar to what Python does"




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```




```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`

Adr.	.	
	.	
	.	
6	–	
7	10	x
8	10	y
	.	
	.	
80		
	.	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"

Adr.	.	
6	–	
7	22	x
8	21	y
	.	
	.	
21	42	42
22	10	10
	.	

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

In `python` there are lot of secrets in the memory allocation that cannot directly be handled by user and a lot of vodoo (incl. garbage collection) takes control about the latter

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`

Adr.		
	.	
	.	
	.	
6	–	
7	10	x
8	10	y
	.	
	.	
80	7	px
	.	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"

Adr.		
	.	
6	–	
7	22	x
8	21	y
	.	
	.	
21	42	42
22	10	10
	.	

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

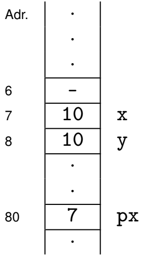
As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

In `python` there are lot of secrets in the memory allocation that cannot directly be handled by user and a lot of vodoo (incl. garbage collection) takes control about the latter

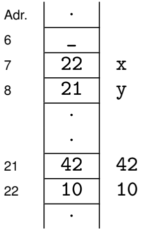
Storage of information in different languages (here as example `C` / `Python`)

Memory `C`




```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"




```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

In `python` there are lot of secrets in the memory allocation that cannot directly be handled by user and a lot of vodoo (incl. garbage collection) takes control about the latter

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`

Adr.	Memory	Variable
	.	
	.	
	.	
6	–	
7	10	x
8	10	y
	.	
	.	
80	7	px
	.	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

Memory "somewhat similar to what `Python` does"

Adr.	Memory	Variable
	.	
6	–	
7	22	x
8	21	y
	.	
	.	
21	42	42
22	10	10
	.	

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
```

As "5" is no longer used, memory cell 21 is freed up [garbage collector].

```
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

In `python` there are lot of secrets in the memory allocation that cannot directly be handled by user and a lot of vodoo (incl. garbage collection) takes control about the latter

Part 1-4: Elementary Data Structures

Storage of information in different languages (here as example `C` / `Python`)

Memory — C

Adr.		
	.	
	.	
	.	
6	–	
7	10	x
8	10	y
	.	
	.	
80	7	px
	.	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

many famous games are based on game engines written in C/C++ (Fortnite, GTA, DOOM, Civilization,. . .)

Memory — "somewhat similar to what `Python` does"

Adr.		
	.	
6	–	
7	22	x
8	21	y
	.	
	.	
21	42	42
22	10	10
	.	

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
As "5" is no longer used, memory cell 21 is freed up [garbage collector].
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

In `python` there are lot of secrets in the memory allocation that cannot directly be handled by user and a lot of vodoo (incl. garbage collection) takes control about the latter

Part 1-4: Elementary Data Structures

Pointer = variable `p` that stores address of another memory cell containing information about "some object *x*".

in symbols "$\mathtt{p} \rightarrow x$"

Data structures can be classified as either contiguous or linked, depending upon whether they are based on arrays or pointers:

- **Contiguously-allocated structures** are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- **Linked data structures** are composed of distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists.

Pointer = variable `p` that stores address of another memory cell containing information about "some object *x*".

in symbols "$\mathtt{p} \to x$"

Data structures can be classified as either contiguous or linked, depending upon whether they are based on arrays or pointers:

- **Contiguously-allocated structures** are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- **Linked data structures** are composed of distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists.

Part 1-4: Elementary Data Structures

Pointer = variable $\texttt{p}$ that stores address of another memory cell containing information about "some object x".

in symbols "$\texttt{p} \rightarrow x$"

Data structures can be classified as either contiguous or linked, depending upon whether they are based on arrays or pointers:

- **Contiguously-allocated structures** are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- **Linked data structures** are composed of distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists.

Part 1-4: Elementary Data Structures

Pointer = variable $\mathtt{p}$ that stores address of another memory cell containing information about "some object x".

in symbols "$\mathtt{p} \to x$"

Data structures can be classified as either contiguous or linked, depending upon whether they are based on arrays or pointers:

- **Contiguously-allocated structures** are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- **Linked data structures** are composed of distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists.

Part 1-4: Elementary Data Structures (Arrays)

The **array** is the fundamental contiguously-allocated data structure. Arrays are structures of fixed-size data records such that each element can be efficiently located by its index (or address).

Part 1-4: Elementary Data Structures (Arrays)

The **array** is the fundamental contiguously-allocated data structure. Arrays are structures of fixed-size data records such that each element can be efficiently located by its index (or address).

Analogy/Example:
Init new array L of length 3 [=allocate 3 consecutive cells (here the ones with address 13,14,15)] and put $L[1] = a$, $L[2] = b$, $L[3] = c$

Part 1-4: Elementary Data Structures (Arrays)

The **array** is the fundamental contiguously-allocated data structure. Arrays are structures of fixed-size data records such that each element can be efficiently located by its index (or address).

Advantages:

Constant-time access given the index

Because the index of each element maps directly to a particular memory address, we can access arbitrary data items instantly provided we know the index.

Space efficiency

Arrays consist purely of data, so no space is wasted with links or other formatting information.

Further, end-of-record information is not needed because arrays are built from fixed-size records.

Part 1-4: Elementary Data Structures (Arrays)

The **array** is the fundamental contiguously-allocated data structure. Arrays are structures of fixed-size data records such that each element can be efficiently located by its index (or address).

Advantages:

Constant-time access given the index

Because the index of each element maps directly to a particular memory address, we can access arbitrary data items instantly provided we know the index.

Space efficiency

Arrays consist purely of data, so no space is wasted with links or other formatting information.

Further, end-of-record information is not needed because arrays are built from fixed-size records.

Disadvantages:

Fixed size and content

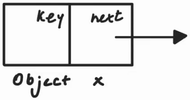
An array can only save one type of data (e.g. only integer, or only bool, ...)

One cannot adjust the size of an array in the middle of a program's execution

Our program will fail soon as we try to add an $(n+1)$-entry if only space for n records was allocated (= overflow). This can be compensated by allocating extremely large arrays, but this can waste space.

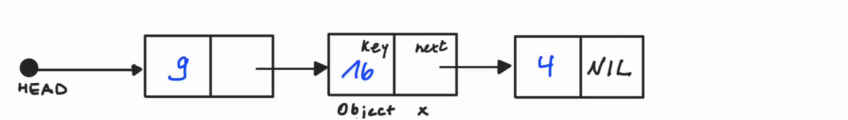
Part 1-4: Elementary Data Structures (Linked Lists)

A **(single) linked list** is a data structure in which the elements are arranged in a linear order.
Each list element is an object with an attribute key (data) and one pointer: next.
Last element points to NIL. Head points to first element.



Part 1-4: Elementary Data Structures (Linked Lists)

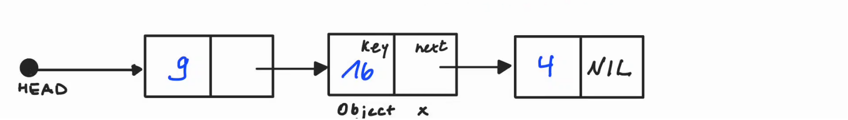
A **(single) linked list** is a data structure in which the elements are arranged in a linear order.
Each list element is an object with an attribute key (data) and one pointer: next.
Last element points to NIL. Head points to first element.



x.next points to its successor in the linked list
x.key is the data stored in object x (here x.key = 16)

Part 1-4: Elementary Data Structures (Linked Lists)

A **(single) linked list** is a data structure in which the elements are arranged in a linear order.
Each list element is an object with an attribute key (data) and one pointer: next.
Last element points to NIL. Head points to first element.



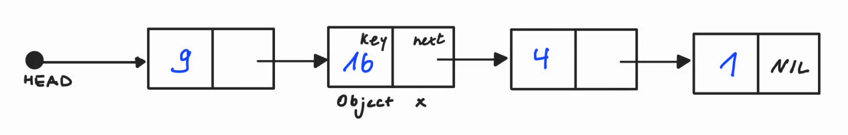
x.next points to its successor in the linked list
x.key is the data stored in object x (here x.key = 16)

Unlike an array in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.

The list elements can be scattered arbitrarily throughout the memory; in particular, it is no longer necessary to preallocate a region of sufficient size to accommodate all list elements.

Part 1-4: Elementary Data Structures (Linked Lists)

A **(single) linked list** is a data structure in which the elements are arranged in a linear order.
Each list element is an object with an attribute key (data) and one pointer: next.
Last element points to NIL. Head points to first element.



x.next points to its successor in the linked list
x.key is the data stored in object x (here x.key = 16)

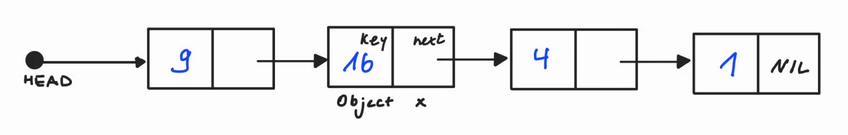
Unlike an array in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.

The list elements can be scattered arbitrarily throughout the memory; in particular, it is no longer necessary to preallocate a region of sufficient size to accommodate all list elements.

Instead, the occupied memory space dynamically adjusts to the current size of the list. However, one needs memory space not only for the list elements themselves but also for the pointers.

Part 1-4: Elementary Data Structures (Linked Lists)

A **(single) linked list** is a data structure in which the elements are arranged in a linear order.
Each list element is an object with an attribute key (data) and one pointer: next.
Last element points to NIL. Head points to first element.



x.next points to its successor in the linked list
x.key is the data stored in object x (here x.key = 16)

Unlike an array in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.

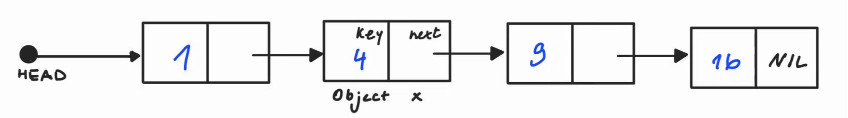
The list elements can be scattered arbitrarily throughout the memory; in particular, it is no longer necessary to preallocate a region of sufficient size to accommodate all list elements.

Instead, the occupied memory space dynamically adjusts to the current size of the list. However, one needs memory space not only for the list elements themselves but also for the pointers.

Such linked lists can be used to realize "dynamic sets" (here the set $\{1, 4, 9, 16\}$)

Part 1-4: Elementary Data Structures (Linked Lists)

A **(single) linked list** is a data structure in which the elements are arranged in a linear order.
Each list element is an object with an attribute key (data) and one pointer: next.
Last element points to NIL. Head points to first element.

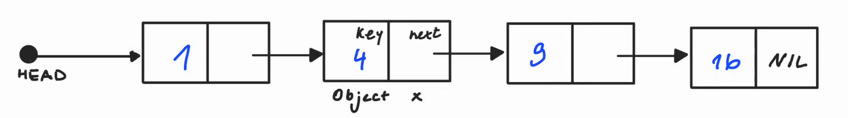


Suppose a linked list L and an array A is sorted:

How easy is it in L, resp., A to remove an object such that L, resp., A stays sorted?

Part 1-4: Elementary Data Structures (Linked Lists)

A **(single) linked list** is a data structure in which the elements are arranged in a linear order.
Each list element is an object with an attribute key (data) and one pointer: next.
Last element points to NIL. Head points to first element.



Suppose a linked list L and an array A is sorted:

How easy is it in L, resp., A to remove an object such that L, resp., A stays sorted?

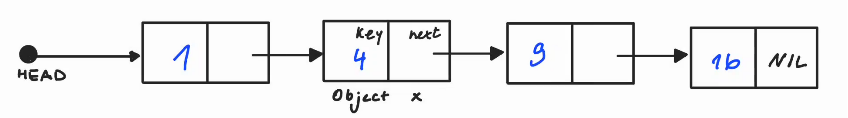
$A = [1, 4, 9, 16]$ and remove $A[1] = 4$:

$A[1] = A[2]$, $A[2] = A[3]$, $A[3] =$fantasy number "42" stating "$A[3]$ is not in use"

$\Longrightarrow$ $\Theta(|A|)$ time

Part 1-4: Elementary Data Structures (Linked Lists)

A **(single) linked list** is a data structure in which the elements are arranged in a linear order.
Each list element is an object with an attribute key (data) and one pointer: next.
Last element points to NIL. Head points to first element.



Suppose a linked list L and an array A is sorted:

How easy is it in L, resp., A to remove an object such that L, resp., A stays sorted?

$A = [1, 4, 9, 16]$ and remove $A[1] = 4$:

$A[1] = A[2]$, $A[2] = A[3]$, $A[3] =$fantasy number "42" stating "$A[3]$ is not in use"

$\implies \Theta(|A|)$ time

$L = [1, 4, 9, 16]$ remove x (say the one with $x.key = 4$ and assume we know the predecessor x' of x)

$x'.next = x.next$

$\implies \Theta(|1|)$ time

Part 1-4: Elementary Data Structures (Linked Lists)

A **(single) linked list** is a data structure in which the elements are arranged in a linear order.
Each list element is an object with an attribute key (data) and one pointer: next.
Last element points to NIL. Head points to first element.

Suppose a linked list L and an array A is sorted:

How easy is it in L, resp., A to remove an object such that L, resp., A stays sorted?

$A = [1, 4, 9, 16]$ and remove $A[1] = 4$:

$A[1] = A[2]$, $A[2] = A[3]$, $A[3] =$fantasy number "42" stating "$A[3]$ is not in use"

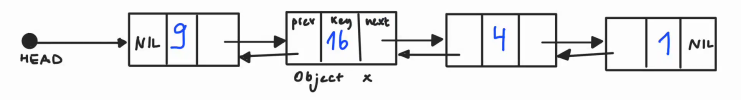
$\implies \Theta(|A|)$ time

$L = [1, 4, 9, 16]$ remove x (say the one with $x.key = 4$ and assume we know the predecessor x' of x)

$x'.next = x.next$

$\implies \Theta(|1|)$ time

Keeping track of predecessor can be done more efficiently with **doubly linked list**:
Each list element is an object with an attribute key (data) and two pointers: next, prev.



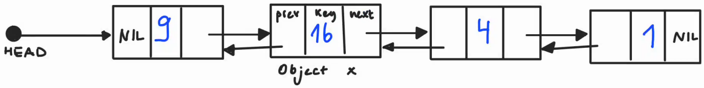
Advantages:

New elements can be placed anywhere in memory and added in constant time before or after a given element by changing the pointers.

Disadvantages:

When searching for an element, you have to go through the list from the first (or last) element to the respective position.
Searching in (sorted) lists L only takes $O(|L|)$ time.

Keeping track of predecessor can be done more efficiently with **doubly linked list**:
Each list element is an object with an attribute key (data) and two pointers: next, prev.



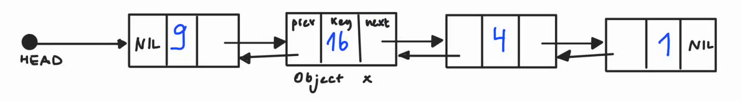
Advantages:

New elements can be placed anywhere in memory and added in constant time before or after a given element by changing the pointers.

Disadvantages:

When searching for an element, you have to go through the list from the first (or last) element to the respective position.
Searching in (sorted) lists L only takes $O(|L|)$ time.

Keeping track of predecessor can be done more efficiently with **doubly linked list**:
Each list element is an object with an attribute key (data) and two pointers: next, prev.



Advantages:

New elements can be placed anywhere in memory and added in constant time before or after a given element by changing the pointers.

Disadvantages:

When searching for an element, you have to go through the list from the first (or last) element to the respective position.
Searching in (sorted) lists L only takes $O(|L|)$ time.

Part 1-4: Elementary Data Structures (Queues and Stacks)

Stacks and **queues** are dynamic sets in which the position of elements inserted / removed from the set is prespecified by a particular order [can be realized by using e.g. single-linked lists].

- **Stacks** follow LIFO = last-in, first-out
  - S.push(x): Inserts item x at the top (last item) of stack S.
  - S.pop(): Removes the top item of stack S.
  - S.top(): Returns the top item of stack S.

S:	$\|2\| \to \|6\| \to \|7\|$
S.push(3):	$\|2\| \to \|6\| \to \|7\| \to \|3\|$
S.top():	returns 3
S.pop():	modifies S to $\|2\| \to \|6\| \to \|7\|$

- **Queues** follow FIFO = first in, first out
  - Q.enqueue(x): Inserts item x at the back (last item) of queue Q.
  - Q.dequeue(): Removes the first item from queue Q.
  - Q.front(): Returns the first item from queue Q.

Q:	$\|2\| \to \|6\| \to \|7\|$
Q.enqueue(3):	$\|2\| \to \|6\| \to \|7\| \to \|3\|$
Q.front():	returns 2
Q.dequeue():	modifies Q to $\|6\| \to \|7\| \to \|3\|$

Part 1-4: Elementary Data Structures (Queues and Stacks)

Stacks and **queues** are dynamic sets in which the position of elements inserted / removed from the set is prespecified by a particular order [can be realized by using e.g. single-linked lists].

Stacks follow LIFO = last-in, first-out

S.push(x): Inserts item x at the top (last item) of stack S.

S.pop(): Removes the top item of stack S.

S.top(): Returns the top item of stack S.

S:	$\|2\| \rightarrow \|6\| \rightarrow \|7\|$
S.push(3):	$\|2\| \rightarrow \|6\| \rightarrow \|7\| \rightarrow \|3\|$
S.top():	returns 3
S.pop():	modifies S to $\|2\| \rightarrow \|6\| \rightarrow \|7\|$

Queues follow FIFO = first in, first out

Q.enqueue(x): Inserts item x at the back (last item) of queue Q.

Q.dequeue(): Removes the first item from queue Q.

Q.front(): Returns the first item from queue Q.

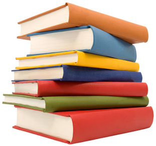
Q:	$\|2\| \rightarrow \|6\| \rightarrow \|7\|$
Q.enqueue(3):	$\|2\| \rightarrow \|6\| \rightarrow \|7\| \rightarrow \|3\|$
Q.front():	returns 2
Q.dequeue():	modifies Q to $\|6\| \rightarrow \|7\| \rightarrow \|3\|$

Part 1-4: Elementary Data Structures (Queues and Stacks)

Stacks and **queues** are dynamic sets in which the position of elements inserted / removed from the set is prespecified by a particular order [can be realized by using e.g. single-linked lists].

Stacks follow LIFO = last-in, first-out

- S.push(x): Inserts item x at the top (last item) of stack S.
- S.pop(): Removes the top item of stack S.
- S.top(): Returns the top item of stack S.



S:	$\|2\| \rightarrow \|6\| \rightarrow \|7\|$
S.push(3):	$\|2\| \rightarrow \|6\| \rightarrow \|7\| \rightarrow \|3\|$
S.top():	returns 3
S.pop():	modifies S to $\|2\| \rightarrow \|6\| \rightarrow \|7\|$

Queues follow FIFO = first in, first out

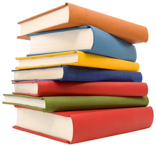
- Q.enqueue(x): Inserts item x at the back (last item) of queue Q.
- Q.dequeue(): Removes the first item from queue Q.
- Q.front(): Returns the first item from queue Q.

Q:	$\|2\| \rightarrow \|6\| \rightarrow \|7\|$
Q.enqueue(3):	$\|2\| \rightarrow \|6\| \rightarrow \|7\| \rightarrow \|3\|$
Q.front():	returns 2
Q.dequeue():	modifies Q to $\|6\| \rightarrow \|7\| \rightarrow \|3\|$

Stacks and **queues** are dynamic sets in which the position of elements inserted / removed from the set is prespecified by a particular order [can be realized by using e.g. single-linked lists].

Stacks follow LIFO = last-in, first-out

- S.push(x): Inserts item x at the top (last item) of stack S.
- S.pop(): Removes the top item of stack S.
- S.top(): Returns the top item of stack S.



S:	$\|2\| \rightarrow \|6\| \rightarrow \|7\|$
S.push(3):	$\|2\| \rightarrow \|6\| \rightarrow \|7\| \rightarrow \|3\|$
S.top():	returns 3
S.pop():	modifies S to $\|2\| \rightarrow \|6\| \rightarrow \|7\|$

Queues follow FIFO = first in, first out

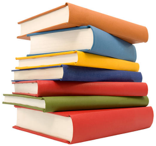
- Q.enqueue(x): Inserts item x at the back (last item) of queue Q.
- Q.dequeue(): Removes the first item from queue Q.
- Q.front(): Returns the first item from queue Q.

Q:	$\|2\| \rightarrow \|6\| \rightarrow \|7\|$
Q.enqueue(3):	$\|2\| \rightarrow \|6\| \rightarrow \|7\| \rightarrow \|3\|$
Q.front():	returns 2
Q.dequeue():	modifies Q to $\|6\| \rightarrow \|7\| \rightarrow \|3\|$

Part 1-4: Elementary Data Structures (Queues and Stacks)

Stacks and **queues** are dynamic sets in which the position of elements inserted / removed from the set is prespecified by a particular order [can be realized by using e.g. single-linked lists].

Stacks follow LIFO = last-in, first-out




S.push(x): Inserts item x at the top (last item) of stack S.

S.pop(): Removes the top item of stack S.

S.top(): Returns the top item of stack S.

S:	$\|2\| \rightarrow \|6\| \rightarrow \|7\|$
S.push(3):	$\|2\| \rightarrow \|6\| \rightarrow \|7\| \rightarrow \|3\|$
S.top():	returns 3
S.pop():	modifies S to $\|2\| \rightarrow \|6\| \rightarrow \|7\|$

Queues follow FIFO = first in, first out

Q.enqueue(x): Inserts item x at the back (last item) of queue Q.

Q.dequeue(): Removes the first item from queue Q.

Q.front(): Returns the first item from queue Q.

Q:	$\|2\| \rightarrow \|6\| \rightarrow \|7\|$
Q.enqueue(3):	$\|2\| \rightarrow \|6\| \rightarrow \|7\| \rightarrow \|3\|$
Q.front():	returns 2
Q.dequeue():	modifies Q to $\|6\| \rightarrow \|7\| \rightarrow \|3\|$

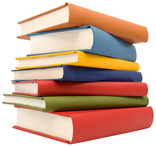
Part 1-4: Elementary Data Structures (Queues and Stacks)

Stacks and **queues** are dynamic sets in which the position of elements inserted / removed from the set is prespecified by a particular order [can be realized by using e.g. single-linked lists].

Stacks follow LIFO = last-in, first-out

- S.push(x): Inserts item x at the top (last item) of stack S.
- S.pop(): Removes the top item of stack S.
- S.top(): Returns the top item of stack S.

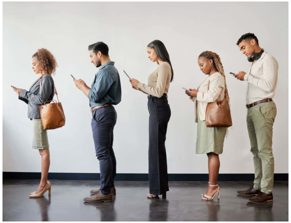
S:	$\|2\| \to \|6\| \to \|7\|$
S.push(3):	$\|2\| \to \|6\| \to \|7\| \to \|3\|$
S.top():	returns 3
S.pop():	modifies S to $\|2\| \to \|6\| \to \|7\|$



Queues follow FIFO = first in, first out

- Q.enqueue(x): Inserts item x at the back (last item) of queue Q.
- Q.dequeue(): Removes the first item from queue Q.
- Q.front(): Returns the first item from queue Q.

Q:	$\|2\| \to \|6\| \to \|7\|$
Q.enqueue(3):	$\|2\| \to \|6\| \to \|7\| \to \|3\|$
Q.front():	returns 2
Q.dequeue():	modifies Q to $\|6\| \to \|7\| \to \|3\|$

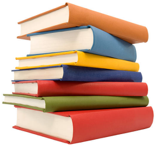


Part 1-4: Elementary Data Structures (Queues and Stacks)

Stacks and **queues** are dynamic sets in which the position of elements inserted / removed from the set is prespecified by a particular order [can be realized by using e.g. single-linked lists].

Stacks follow LIFO = last-in, first-out

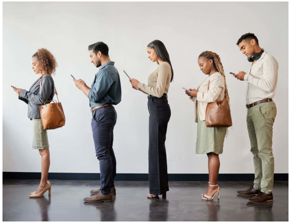
- S.push(x): Inserts item x at the top (last item) of stack S.
- S.pop(): Removes the top item of stack S.
- S.top(): Returns the top item of stack S.



S:	$\|2\| \to \|6\| \to \|7\|$
S.push(3):	$\|2\| \to \|6\| \to \|7\| \to \|3\|$
S.top():	returns 3
S.pop():	modifies S to $\|2\| \to \|6\| \to \|7\|$

Queues follow FIFO = first in, first out

- Q.enqueue(x): Inserts item x at the back (last item) of queue Q.
- Q.dequeue(): Removes the first item from queue Q.
- Q.front(): Returns the first item from queue Q.



Q:	$\|2\| \to \|6\| \to \|7\|$
Q.enqueue(3):	$\|2\| \to \|6\| \to \|7\| \to \|3\|$
Q.front():	returns 2
Q.dequeue():	modifies Q to $\|6\| \to \|7\| \to \|3\|$

Part 1-4: Elementary Data Structures (Trees)

Trees form a more general framework than linked list and are defined as "special graphs".

Let us start with the formal definition first.

The next slide contains a lot of definitions that we also need later on (e.g. for heaps, binary search trees, AVL trees, . . .). Most of these defs refer Sec B4 and B5 in the Cormen et al. course-book.

BUT don't be afraid, they are easy to grasp: STEP-BY-STEP and stay with me!

Trees form a more general framework than linked list and are defined as "special graphs".

Let us start with the formal definition first.

The next slide contains a lot of definitions that we also need later on (e.g. for heaps, binary search trees, AVL trees, . . .). Most of these defs refer Sec B4 and B5 in the Cormen et al. course-book.

BUT don't be afraid, they are easy to grasp: STEP-BY-STEP and stay with me!

Part 1-4: Elementary Data Structures (Trees)

Trees form a more general framework than linked list and are defined as "special graphs".

Let us start with the formal definition first.

The next slide contains a lot of definitions that we also need later on (e.g. for heaps, binary search trees, AVL trees, . . .). Most of these defs refer Sec B4 and B5 in the Cormen et al. course-book.

BUT don’t be afraid, they are easy to grasp: STEP-BY-STEP and stay with me!

Part 1-4: Elementary Data Structures (Trees)

Trees form a more general framework than linked list and are defined as "special graphs".

Let us start with the formal definition first.

The next slide contains a lot of definitions that we also need later on (e.g. for heaps, binary search trees, AVL trees, . . .). Most of these defs refer Sec B4 and B5 in the Cormen et al. course-book.

BUT don’t be afraid, they are easy to grasp: STEP-BY-STEP and stay with me!

Part 1-4: Elementary Data Structures (Trees) - some graph theory

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k + 1$ is defined by:

$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

Part 1-4: Elementary Data Structures (Trees) - some graph theory

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k + 1$ is defined by:

$$P = (v_0, v_1, \ldots, v_k) \text{ is a simple path of length } k \geq 2 \text{ and } \{v_k, v_0\} \in E.$$

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k+1$ is defined by:

$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

Part 1-4: Elementary Data Structures (Trees) - some graph theory

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k + 1$ is defined by:
$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k + 1$ is defined by:
$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

Part 1-4: Elementary Data Structures (Trees) - some graph theory

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k + 1$ is defined by:

$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \dots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \dots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \dots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \dots, v_k, v_0)$ of length $k + 1$ is defined by:

$P = (v_0, v_1, \dots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k+1$ is defined by:

$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k + 1$ is defined by:

$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k + 1$ is defined by:
$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k + 1$ is defined by:

$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k + 1$ is defined by:

$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

Part 1-4: Elementary Data Structures (Trees) - some graph theory

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \ldots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \ldots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \ldots, v_k, v_0)$ of length $k+1$ is defined by:

$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

Part 1-4: Elementary Data Structures (Trees) - some graph theory

[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \dots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \le i < k$. $P = (v_0, v_1, \dots, v_k)$ is also called $v_0 v_k$-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \dots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \dots, v_k, v_0)$ of length $k + 1$ is defined by:

$P = (v_0, v_1, \dots, v_k)$ is a simple path of length $k \ge 2$ and $\{v_k, v_0\} \in E$.

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

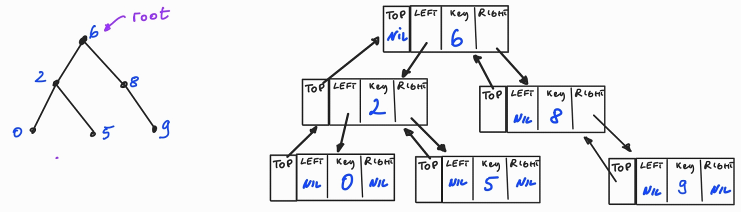
If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

A rooted tree T is **binary** if each vertex as *at most* two children. If T is ordered and binary, then there is a clear distinction between right and left child (even if a vertex has only child).



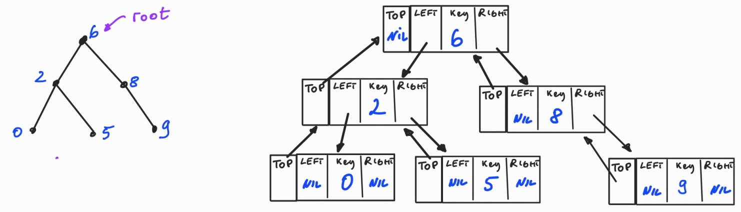
Advantages:

- New elements can be placed anywhere in memory and added in constant time before or after a given element by changing the pointers.
- Searching in a sorted tree takes $O(h)$ time, with $h =$ height of tree (=longest simple path from root to some leaf).
- In so-called "balanced trees" $h \in O(logn)$ where $n =$ number of vertex (key/data) stored in T [details in upcoming lectures]

Disadvantages:

- Searching in "non-balanced" tree $O(|n|)$ time (as in linked-lists)
- Making a non-balanced tree to a balanced one gets tricky (in particular, insertion of elements is more complicated)

A rooted tree T is **binary** if each vertex as *at most* two children. If T is ordered and binary, then there is a clear distinction between right and left child (even if a vertex has only child).



Advantages:

New elements can be placed anywhere in memory and added in constant time before or after a given element by changing the pointers.

Searching in a sorted tree takes $O(h)$ time, with $h =$ height of tree (=longest simple path from root to some leaf).

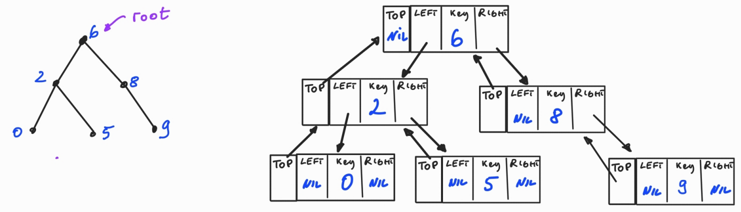
In so-called "balanced trees" $h \in O(logn)$ where $n =$ number of vertex (key/data) stored in T [details in upcoming lectures]

Disadvantages:

Searching in "non-balanced" tree $O(|n|)$ time (as in linked-lists)

Making a non-balanced tree to a balanced one gets tricky (in particular, insertion of elements is more complicated)

A rooted tree T is **binary** if each vertex as *at most* two children. If T is ordered and binary, then there is a clear distinction between right and left child (even if a vertex has only child).



Advantages:

New elements can be placed anywhere in memory and added in constant time before or after a given element by changing the pointers.

Searching in a sorted tree takes $O(h)$ time, with $h =$ height of tree (=longest simple path from root to some leaf).

In so-called "balanced trees" $h \in O(logn)$ where $n =$ number of vertex (key/data) stored in T [details in upcoming lectures]

Disadvantages:

Searching in "non-balanced" tree $O(|n|)$ time (as in linked-lists)

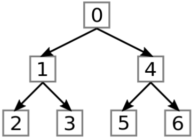
Making a non-balanced tree to a balanced one gets tricky (in particular, insertion of elements is more complicated)

Part 1-4: Elementary Data Structures (Rooted Binary Trees)

Traversal of trees (more details in upcoming lectures).

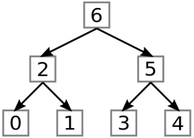
Preorder:

1. visit current vertex
2. recursively traverse left subtree
3. recursively traverse right subtree



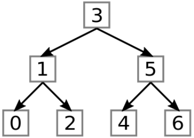
Postorder:

1. recursively traverse left subtree
2. recursively traverse right subtree
3. visit current vertex



Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree



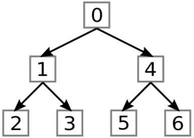
numbers in squares =
order in which nodes are visited

Part 1-4: Elementary Data Structures (Rooted Binary Trees)

Traversal of trees (more details in upcoming lectures).

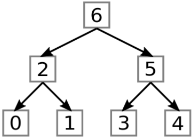
Preorder:

1. visit current vertex
2. recursively traverse left subtree
3. recursively traverse right subtree



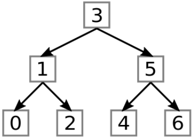
Postorder:

1. recursively traverse left subtree
2. recursively traverse right subtree
3. visit current vertex



Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree



numbers in squares =
order in which nodes are visited

Part 1-4: Elementary Data Structures

Plenty of other data structures exist and we will examine some of them later in the course