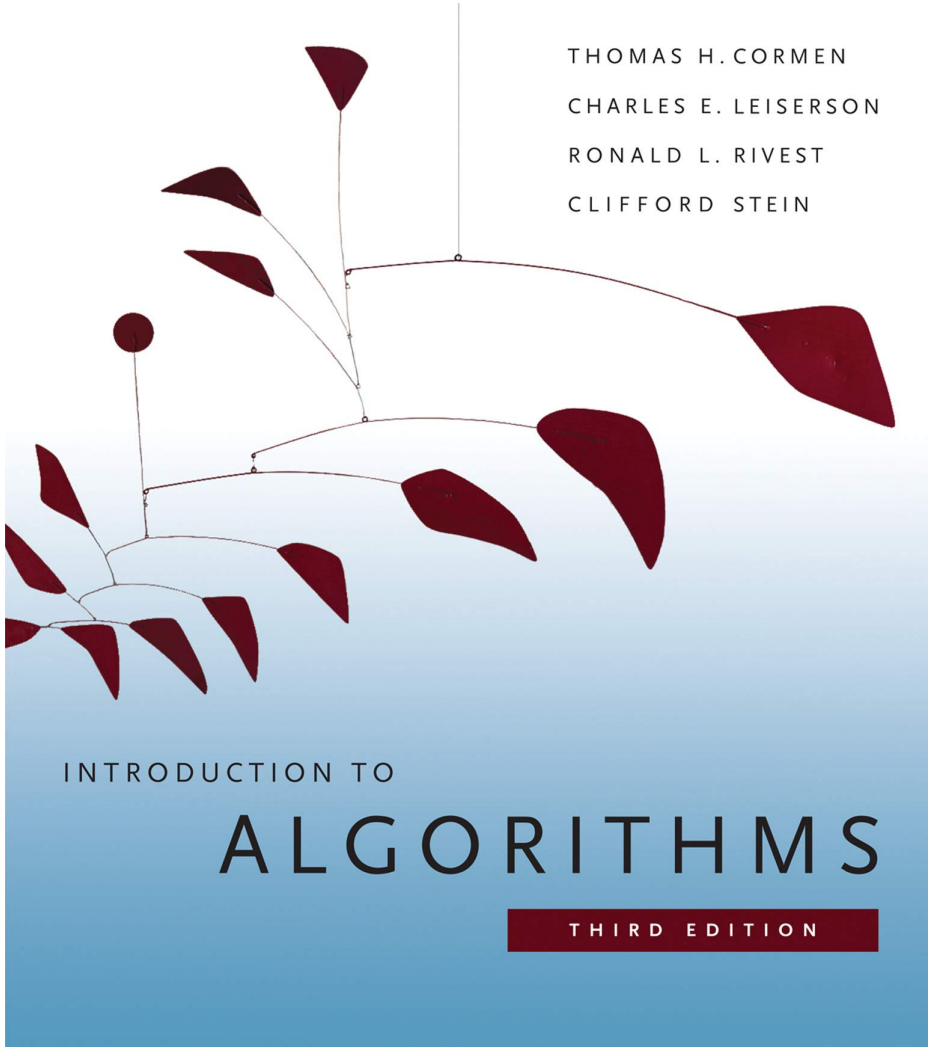
Algorithms and Complexity

1./2. crash course - recap - time complexity

Marc Hellmuth

University of Stockholm

Time complexity



Chapter 1-3 and 4.5

Time complexity

Runtime of an algorithm

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

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

Time complexity

Runtime of an algorithm

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

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

not really comparible since this differs by the used computers

Time complexity

Runtime of an algorithm

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

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

not really comparible since this differs by the used computers

$\Longrightarrow$ we need a notation that helps to classify “runtime” that does not depend on the architecture of a computer

Time complexity

Runtime of an algorithm

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

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

not really comparible since this differs by the used computers
$\Longrightarrow$ we need a notation that helps to classify “runtime” that does not depend on the architecture of a computer

Time complexity

NOT: measure runtime on a specific computer

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

Need abstract measure for complexity to estimate asymptotic costs that depends on the size of the input

Time complexity

Add two numbers

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

Takes 5 single additions.

Time complexity

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 a bit more if we consider ”carryover”) for two numbers having m, resp., n.

Time complexity

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 a bit more if we consider ”carryover”) for two numbers having m, resp., n.

However, in the RAM-model (as in real computer) instructions are executed one after another, with no concurrent operations and if we have an instruction add, then this execution is counted once.

Time complexity

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 a bit more if we consider ”carryover”) for two numbers having m, resp., n.

However, in the RAM-model (as in real computer) instructions are executed one after another, with no concurrent operations and if we have an instruction add, then this execution is counted once.

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).

Time complexity

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 a bit more if we consider ”carryover”) for two numbers having m, resp., n.

However, in the RAM-model (as in real computer) instructions are executed one after another, with no concurrent operations and if we have an instruction add, then this execution is counted once.

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)

Time complexity

The runtime of an algorithm with input I is denoted by $T(|I|)$, where $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A[n]$, input size $|I| = n$

COUNT_ZEROS(array $a[n]$)

```
1: int count = 0
2: for (int i=0; i<n; i++) do
3:     if a[i] == 0 then
4:         count++
```

Time complexity

The runtime of an algorithm with input I is denoted by $T(|I|)$, where $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A[n]$, input size $|I| = n$

COUNT_ZEROS(array $a[n]$)

```
1: int count = 0
2: for (int i=0; i<n; i++) do
3:     if a[i] == 0 then
4:         count++
```

variable declaration (e.g. **int** i):	2
assignment statement (e.g. i=0):	2
”$<$”-compare	n+1
”==”-compare	n
array access	n
increment (++)	n+n
Σsingle instructions $= T(n) =$	$5n+5$

Time complexity

The runtime of an algorithm with input I is denoted by $T(|I|)$, where $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A[n]$, input size $|I| = n$

COUNT_ZEROS(array $a[n]$)

```
1: int count = 0
2: for (int i=0; i<n; i++) do
3:     if a[i] == 0 then
4:         count++
```

variable declaration (e.g. **int** i):	2
assignment statement (e.g. i=0):	2
"$<$"-compare	n+1
"=="-compare	n
array access	n
increment (++)	n+n
Σsingle instructions $= T(n) =$	$5n+5$

Still, this is unsatisfying, e.g. if you have $T(n) = 5n+5$ vs $T'(n) = 6n$ (which is faster?)

Time complexity

The runtime of an algorithm with input I is denoted by $T(|I|)$, where $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A[n]$, input size $|I| = n$

COUNT_ZEROS(array $a[n]$)

```
1: int count = 0
2: for (int i=0; i<n; i++) do
3:     if a[i] == 0 then
4:         count++
```

variable declaration (e.g. **int** i):	2
assignment statement (e.g. i=0):	2
"$<$"-compare	n+1
"=="-compare	n
array access	n
increment (++)	n+n
Σsingle instructions $= T(n) =$	$5n+5$

Still, this is unsatisfying, e.g. if you have $T(n) = 5n+5$ vs $T'(n) = 6n$ (which is faster?)

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

Time complexity

The runtime of an algorithm with input I is denoted by $T(|I|)$, where $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A[n]$, input size $|I| = n$

COUNT_ZEROS(array $a[n]$)

```
1: int count = 0
2: for (int i=0; i<n; i++) do
3:     if a[i] == 0 then
4:         count++
```

variable declaration (e.g. **int** i):	2
assignment statement (e.g. i=0):	2
"$<$"-compare	n+1
"==”-compare	n
array access	n
increment (++)	n+n
Σsingle instructions $= T(n) =$	$5n+5$

Still, this is unsatisfying, e.g. if you have $T(n) = 5n+5$ vs $T'(n) = 6n$ (which is faster?)

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

For sure, there are far more complicated functions $T(n)$, e.g. $T(n) = log_2(n) + \sqrt{2}sin(n)$.

Time complexity

The runtime of an algorithm with input I is denoted by $T(|I|)$, where $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A[n]$, input size $|I| = n$

COUNT_ZEROS(array $a[n]$)

```
1: int count = 0
2: for (int i=0; i<n; i++) do
3:     if a[i] == 0 then
4:         count++
```

variable declaration (e.g. **int** i):	2
assignment statement (e.g. i=0):	2
"$<$"-compare	n+1
"==”-compare	n
array access	n
increment (++)	n+n
Σsingle instructions $= T(n) =$	$5n+5$

Still, this is unsatisfying, e.g. if you have $T(n) = 5n+5$ vs $T'(n) = 6n$ (which is faster?)

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

For sure, there are far more complicated functions $T(n)$, e.g. $T(n) = log_2(n) + \sqrt{2}sin(n)$.

We are, in general, not interested in specific values for n but the asymptotic behaviour of $T(n)$ (that is for large n)

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$

The notation $T(n) = O(f(n))$ is also very commonly used.

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$

Examples

$T(n) = 5n + 6$ is in $O(n)$:

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$

Examples

$T(n) = 5n+6$ is in $O(n)$:

$5n \leq 5n$ for all n (incl. $n \geq 1$)

$6 \leq 6n$ for all $n \geq 1$

Thus, $T(n) = 5n+6 \leq 5n+6n = 11n = cf(n)$ for all $n \geq 1$

Thus, $5n+6 \in O(n)$ (choose $c = 11$ and $n_0 = 1$.)

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$

Examples

$T(n) = 5n+6$ is in $O(n)$:

$$5n \leq 5n \text{ for all } n \text{ (incl. } n \geq 1 \text{)}$$

$$6 \leq 6n \text{ for all } n \geq 1$$

$$\text{Thus, } T(n) = 5n+6 \leq 5n+6n = 11n = cf(n) \text{ for all } n \geq 1$$

Thus, $5n+6 \in O(n)$ (choose $c = 11$ and $n_0 = 1$.)

Note, $T(n) = 5n+6 \leq 11 \cdot n \leq 11 \cdot n \log n \leq 11 \cdot n^{100}$

Hence, $T(n) \in O(n \log n)$, $T(n) \in O(n^{100})$. We usually want to find tight upper bounds.

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

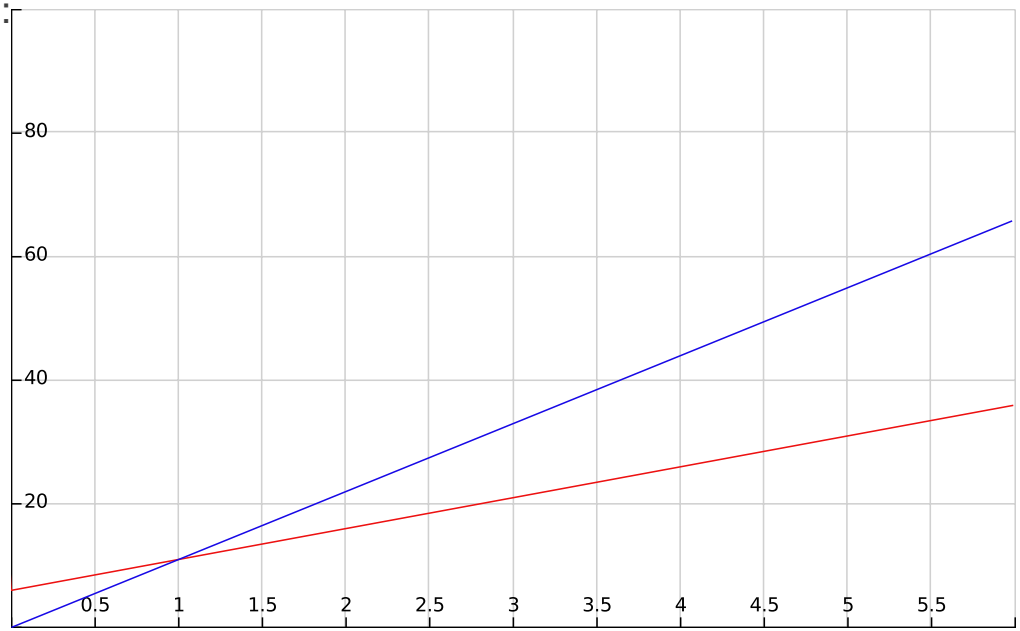
$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$

Examples

$T(n) = 5n + 6$ is in $O(n)$:



5n+6 and 11n

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$

Examples

$$T(n) = \sum_{i=0}^{p} x_i n^i = x_p n^p + \ldots + x_1 n^1 + x_0 n^0 \in O(n^p) \text{ for all } n \geq 1$$

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$

Examples

$$T(n) = \sum_{i=0}^{p} x_i n^i = x_p n^p + \ldots + x_1 n^1 + x_0 n^0 \in O(n^p) \text{ for all } n \geq 1$$

First, $x_i \leq \max\{|x_1|, \ldots, |x_p|\} =: M$ for all i implies

$$T(n) = \sum_{i=0}^{p} x_i n^i \leq \sum_{i=0}^{p} M n^i = M(\sum_{i=0}^{p} n^i)$$

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$

Examples

$$T(n) = \sum_{i=0}^{p} x_i n^i = x_p n^p + \ldots + x_1 n^1 + x_0 n^0 \in O(n^p) \text{ for all } n \geq 1$$

First, $x_i \leq \max\{|x_1|, \ldots, |x_p|\} =: M$ for all i implies

$$T(n) = \sum_{i=0}^{p} x_i n^i \leq \sum_{i=0}^{p} M n^i = M(\sum_{i=0}^{p} n^i)$$

Second $n^0 \leq n^1 \leq \cdots \leq n^p$ implies that

$$M(\sum_{i=0}^{p} n^i) \leq M(\sum_{i=0}^{p} n^p) = M \cdot ((p+1) \cdot n^p) = c \cdot n^p \text{ for } c = M \cdot (p+1) \text{ and all } n \geq n_0 = 1$$

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

The asymptotic complexity $T(n)$ is limited from above by a function $f(n)$ whenever there are postive constants n_0 and c such that for all $n > n_0$ it holds that

$$T(n) \leq cf(n).$$

In this case, we say that $T(n) \in O(f(n))$

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$

Examples

$$T(n) = \sum_{i=0}^{p} x_i n^i = x_p n^p + \ldots + x_1 n^1 + x_0 n^0 \in O(n^p) \text{ for all } n \geq 1$$

First, $x_i \leq \max\{|x_1|, \ldots, |x_p|\} =: M$ for all i implies

$$T(n) = \sum_{i=0}^{p} x_i n^i \leq \sum_{i=0}^{p} M n^i = M(\sum_{i=0}^{p} n^i)$$

Second $n^0 \leq n^1 \leq \cdots \leq n^p$ implies that

$$M(\sum_{i=0}^{p} n^i) \leq M(\sum_{i=0}^{p} n^p) = M \cdot ((p+1) \cdot n^p) = c \cdot n^p \text{ for } c = M \cdot (p+1) \text{ and all } n \geq n_0 = 1$$

Thus, choose $c = M \cdot p$ and $n_0 = 1$ to conclude that $T(n) \in O(n^p)$.

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

In a similar way, one can compute lower bounds $f(n)$ to show that $T(n)$ grows (asymptotically) at least as “fast” as $f(n)$

$$T(n) \in \Omega(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \geq cf(n)$$

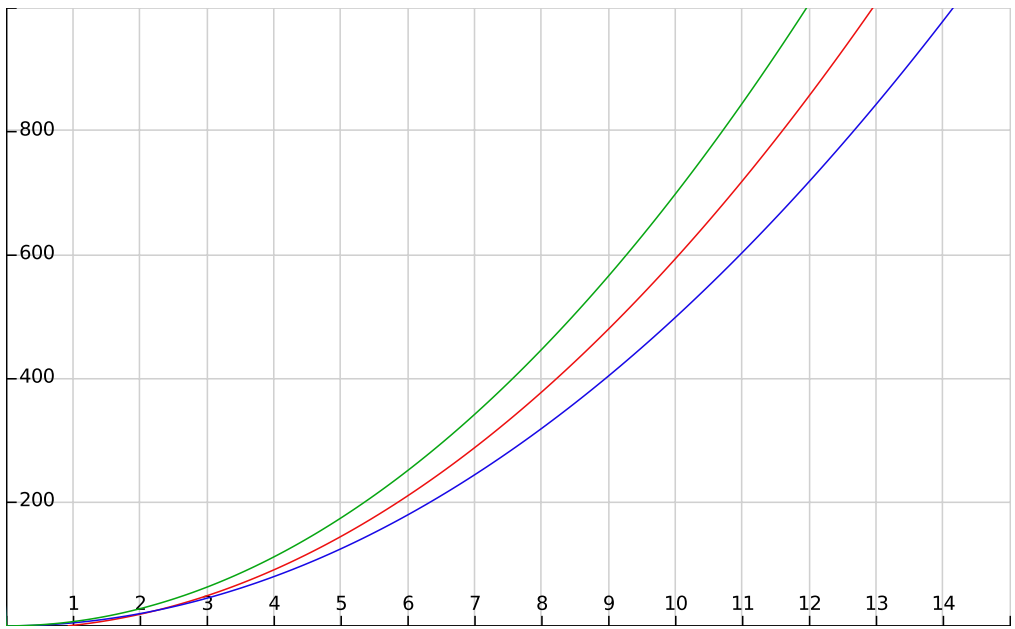
Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

In a similar way, one can compute lower bounds $f(n)$ to show that $T(n)$ grows (asymptotically) at least as "fast" as $f(n)$

$$T(n) \in \Omega(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \geq cf(n)$$

Example: $T(n) = 6n^2 - 5 \in \Omega(n^2)$ (choose $c = 5$ and $n_0 = 3$)



$$5n^2 \leq 6n^2 - 5 \leq 7n^2$$

Hence, $T(n) \in \Omega(n^2)$ and $T(n) \in O(n^2)$

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$
$$T(n) \in \Omega(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \geq cf(n)$$

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$
$$T(n) \in \Omega(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \geq cf(n)$$

If $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$ then $T(n) \in \Theta(f(n))$

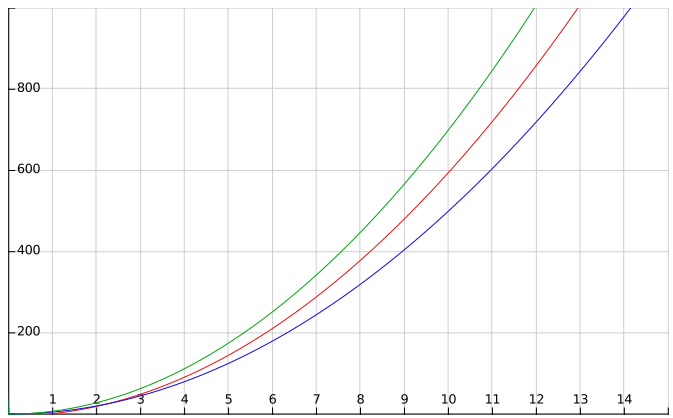
Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$
$$T(n) \in \Omega(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \geq cf(n)$$

If $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$ then $T(n) \in \Theta(f(n))$

Example: $T(n) = 6n^2 - 5 \in \Omega(n^2)$ (choose e.g. $c = 5$ and $n_0 = 3$)
$T(n) = 6n^2 - 5 \in O(n^2)$ (choose e.g. $c = 7$ and $n_0 = 1$)



$$5n^2 \leq 6n^2 - 5 \leq 7n^2$$

Hence, $T(n) \in \Omega(n^2)$ and $T(n) \in O(n^2)$ and thus, $T(n) \in \Theta(n^2)$

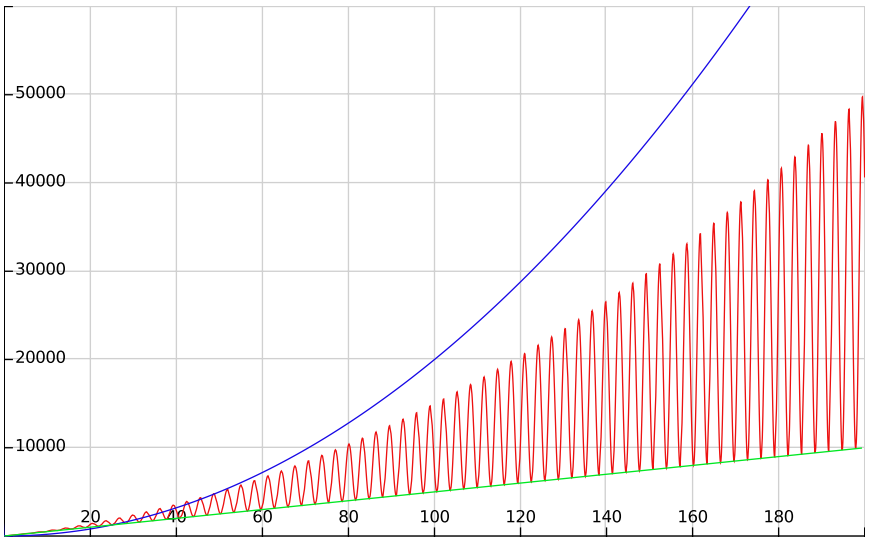
Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$
$$T(n) \in \Omega(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \geq cf(n)$$

If $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$ then $T(n) \in \Theta(f(n))$

Example: $T(n) = n^2(sin(n))^2 + 50n$



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

Time complexity

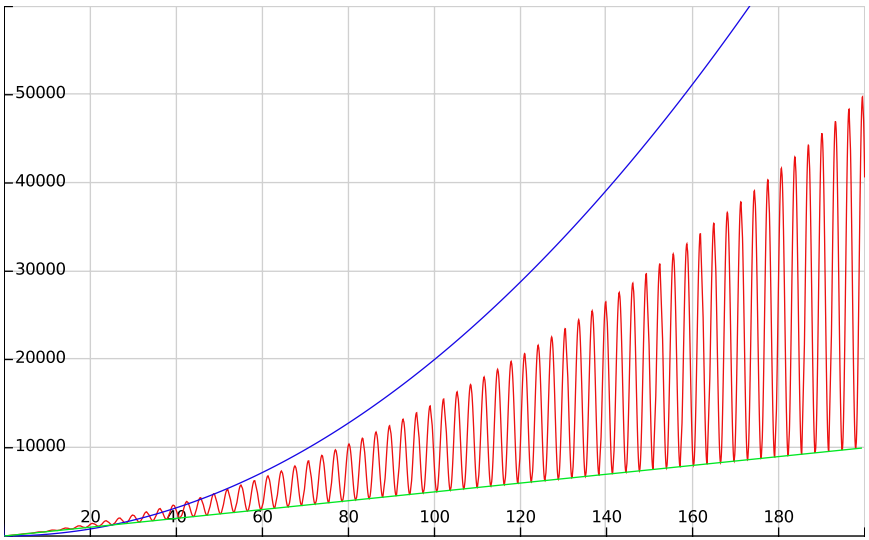
Big-O-, Big-Θ- and Big-Ω-Notation

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$
$$T(n) \in \Omega(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \geq cf(n)$$

If $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$ then $T(n) \in \Theta(f(n))$

Example: $T(n) = n^2(sin(n))^2 + 50n$

Since $sin(n) \leq 1$, we have $n^2(sin(n))^2 + 50n \leq n^2 + 50n \leq 2n^2$ for all $n \geq 50$ and thus, $T(n) \in O(n^2)$



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

Time complexity

Big-O-, Big-Θ- and Big-Ω-Notation

$$T(n) \in O(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \leq cf(n)$$
$$T(n) \in \Omega(f(n)) :\Leftrightarrow \exists c > 0, n_0 > 0 : \forall n > n_0 : T(n) \geq cf(n)$$

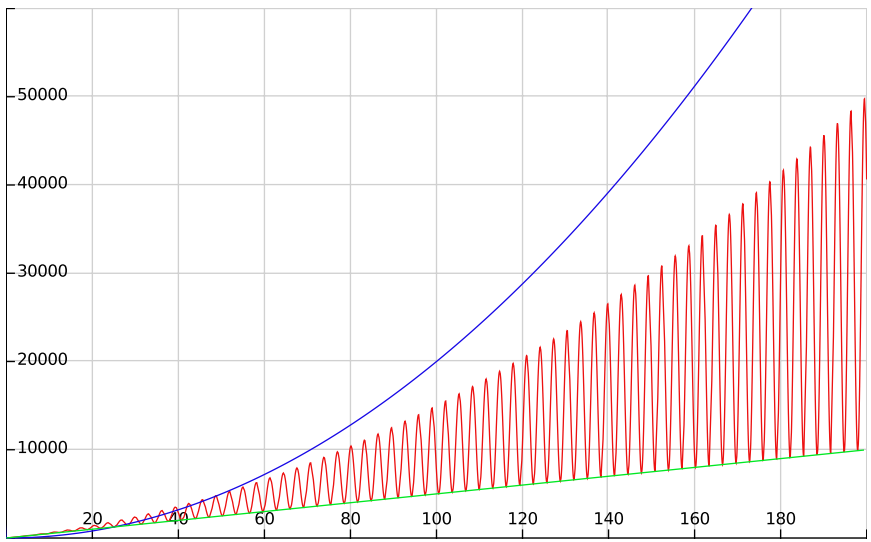
If $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$ then $T(n) \in \Theta(f(n))$

Example: $T(n) = n^2(sin(n))^2 + 50n$

Since $sin(n) \leq 1$, we have $n^2(sin(n))^2 + 50n \leq n^2 + 50n \leq 2n^2$ for all $n \geq 50$ and thus, $T(n) \in O(n^2)$

Since $n^2(sin(n))^2 \geq 0$, we have $n^2(sin(n))^2 + 50n \geq 50n$ for all $n \geq 1$ and thus, $T(n) \in \Omega(n)$

(and we can't do better - thus "no Θ for $T(n)$"!)



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

Time complexity

O-Notation

$O(\dots)$ (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

Time Complexity

General rules

Now, O-notation only which is the most interesting part for us.

Time Complexity

General rules

Now, $\mathcal{O}$-notation only which is the most interesting part for us.

Loops: Product of number of runs and costs of most expencive task within this loop

Time Complexity

General rules

Now, O-notation only which is the most interesting part for us.

Loops: Product of number of runs and costs of most expencive task within this loop

Summation of parts: If algorithm A consists of two independent parts A_1 and A_2 (with runtime $T_1(n) \in O(f(n))$, resp., $T_2(n) \in O(g(n))$), then complexity of A is

$$T(n) = T_1(n) + T_2(n) = O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

Time Complexity

General rules

Now, O-notation only which is the most interesting part for us.

Loops: Product of number of runs and costs of most expencive task within this loop

Summation of parts: If algorithm A consists of two independent parts A_1 and A_2 (with runtime $T_1(n) \in O(f(n))$, resp., $T_2(n) \in O(g(n))$), then complexity of A is

$$T(n) = T_1(n) + T_2(n) = O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

DO_SMTH(int n)

```
1: print "Hello World"
2: for (int i = 0; i < n; i++) do
3:     if n is even then
4:         return 0
5:     else
6:         for (int j = 0; j < n; j++) do
7:             n = n · n
```

Time Complexity

General rules

Now, O-notation only which is the most interesting part for us.

Loops: Product of number of runs and costs of most expencive task within this loop

Summation of parts: If algorithm A consists of two independent parts A_1 and A_2 (with runtime $T_1(n) \in O(f(n))$, resp., $T_2(n) \in O(g(n))$), then complexity of A is

$$T(n) = T_1(n) + T_2(n) = O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

DO_SMTH(int n)

```
1: print "Hello World"
2: for (int i = 0; i < n; i++) do
3:     if n is even then
4:         return 0
5:     else
6:         for (int j = 0; j < n; j++) do
7:             n = n · n
```

All assignments, cases, statements (eg. **print**, **int** $i = 0$, $n = n \cdot n$, **return** 0, $j < n$) in $O(1)$ time

Time Complexity

General rules

Now, O-notation only which is the most interesting part for us.

Loops: Product of number of runs and costs of most expencive task within this loop

Summation of parts: If algorithm A consists of two independent parts A_1 and A_2 (with runtime $T_1(n) \in O(f(n))$, resp., $T_2(n) \in O(g(n))$), then complexity of A is

$$T(n) = T_1(n) + T_2(n) = O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

DO_SMTH(int n)

```
1: print "Hello World"
2: for (int i = 0; i < n; i++) do
3:     if n is even then
4:         return 0
5:     else
6:         for (int j = 0; j < n; j++) do
7:             n = n · n
```

All assignments, cases, statements (eg. **print**, **int** $i = 0$, $n = n \cdot n$, **return** 0, $j < n$) in $O(1)$ time

DO_SMTH consists of two main-parts:

A_1 = **print** "Hello World" and A_2 = Line 2-7

Hence, runtime of DO_SMTH is $O(1)+$ runtime A_2 $\Longrightarrow$ examine A_2 !

Time Complexity

General rules

Now, O-notation only which is the most interesting part for us.

Loops: Product of number of runs and costs of most expencive task within this loop

Summation of parts: If algorithm A consists of two independent parts A_1 and A_2 (with runtime $T_1(n) \in O(f(n))$, resp., $T_2(n) \in O(g(n))$), then complexity of A is

$$T(n) = T_1(n) + T_2(n) = O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

DO_SMTH(int n)

```
1: print "Hello World"
2: for (int i = 0; i < n; i++) do
3:     if n is even then
4:         return 0
5:     else
6:         for (int j = 0; j < n; j++) do
7:             n = n · n
```

The most expensive task within the loop in Line 2 is the call of the 2nd for Loop in Line 6

Time Complexity

General rules

Now, O-notation only which is the most interesting part for us.

Loops: Product of number of runs and costs of most expencive task within this loop

Summation of parts: If algorithm A consists of two independent parts A_1 and A_2 (with runtime $T_1(n) \in O(f(n))$, resp., $T_2(n) \in O(g(n))$), then complexity of A is

$$T(n) = T_1(n) + T_2(n) = O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

DO_SMTH(int n)

```
1: print "Hello World"
2: for (int i = 0; i < n; i++) do
3:     if n is even then
4:         return 0
5:     else
6:         for (int j = 0; j < n; j++) do
7:             n = n · n
```

The most expensive task within the loop in Line 2 is the call of the 2nd for Loop in Line 6
$\Longrightarrow$ examine 2nd loop:

number of runs = n and most expensive task "$n = n \cdot n$" in $O(1)$ time

Thus, runtime 2nd loop is $O(n)$

Time Complexity

General rules

Now, O-notation only which is the most interesting part for us.

Loops: Product of number of runs and costs of most expencive task within this loop

Summation of parts: If algorithm A consists of two independent parts A_1 and A_2 (with runtime $T_1(n) \in O(f(n))$, resp., $T_2(n) \in O(g(n))$), then complexity of A is

$$T(n) = T_1(n) + T_2(n) = O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

DO_SMTH(int n)

```
1: print "Hello World"
2: for (int i = 0; i < n; i++) do
3:     if n is even then
4:         return 0
5:     else
6:         for (int j = 0; j < n; j++) do
7:             n = n · n
```

The most expensive task within the loop in Line 2 is the call of the 2nd for Loop in Line 6
$\Longrightarrow$ examine 2nd loop:

number of runs = n and most expensive task "$n = n \cdot n$" in $O(1)$ time

Thus, runtime 2nd loop is $O(n)$

Hence, runtime first loop = number of runs times most expensive task $\in O(n^2)$

Time Complexity

General rules

Now, O-notation only which is the most interesting part for us.

Loops: Product of number of runs and costs of most expencive task within this loop

Summation of parts: If algorithm A consists of two independent parts A_1 and A_2 (with runtime $T_1(n) \in O(f(n))$, resp., $T_2(n) \in O(g(n))$), then complexity of A is

$$T(n) = T_1(n) + T_2(n) = O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$

DO_SMTH(int n)

```
1: print "Hello World"
2: for (int i = 0; i < n; i++) do
3:     if n is even then
4:         return 0
5:     else
6:         for (int j = 0; j < n; j++) do
7:             n = n · n
```

The most expensive task within the loop in Line 2 is the call of the 2nd for Loop in Line 6
$\Longrightarrow$ examine 2nd loop:

number of runs = n and most expensive task "$n = n \cdot n$" in $O(1)$ time

Thus, runtime 2nd loop is $O(n)$

Hence, runtime first loop = number of runs times most expensive task $\in O(n^2)$

Thus, runtime DO_SMTH is $O(1)+$ runtime $A_2 \in O(1+n^2) = O(n^2)$

Time Complexity

Further example

DO_SMTH(graph $G = (V, E)$)

```
1: for (all vertices v ∈ V) do
2:     for (all vertices x ∈ N(v)) do
3:         print "neighbor of v is x"
```

Time Complexity

Further example

DO_SMTH(graph $G = (V, E)$)

```
1: for (all vertices v ∈ V) do
2:     for (all vertices x ∈ N(v)) do
3:         print "neighbor of v is x"
```

Naive way:

$T(n) = |V|^2$ since for all $|V|$ vertices, we print all neighbors in $N(v)$ and $|N(v)| \leq |V|$

Time Complexity

Further example

DO_SMTH(graph $G = (V, E)$)

```
1: for (all vertices v ∈ V) do
2:     for (all vertices x ∈ N(v)) do
3:         print "neighbor of v is x"
```

Naive way:

$T(n) = |V|^2$ since for all $|V|$ vertices, we print all neighbors in $N(v)$ and $|N(v)| \leq |V|$

Better way:

For each $v \in V$ we print all $\deg(v)$ vertices.

Time Complexity

Further example

DO_SMTH(graph $G = (V, E)$)

```
1: for (all vertices v ∈ V) do
2:     for (all vertices x ∈ N(v)) do
3:         print "neighbor of v is x"
```

Naive way:

$T(n) = |V|^2$ since for all $|V|$ vertices, we print all neighbors in $N(v)$ and $|N(v)| \leq |V|$

Better way:

For each $v \in V$ we print all $\deg(v)$ vertices.

Hence, $T(n) = \sum_{v \in V} \deg(v) = 2|E| \in O(|E|)$.

Thus, instead of a quadratic runtime $O(|V|^2)$ this algorithm has even linear runtime $O(|E|)$

Time Complexity

Further example

HALVE(int n)

1: **while** $(n > 1)$ **do**
2: $\quad n = \frac{n}{2}$ *(pre-decimal point position)*

Time Complexity

Further example

HALVE(int n)

1: **while** $(n > 1)$ **do**
2: $n = \frac{n}{2}$ *(pre-decimal point position)*

$$T(n) = T(1) + T(\frac{n}{2})$$

Time Complexity

Further example

HALVE(int n)

1: **while** $(n > 1)$ **do**
2: $\quad n = \frac{n}{2}$ *(pre-decimal point position)*

$$T(n) = T(1) + T(\frac{n}{2})$$
$$= T(1) + (T(1) + T(\frac{n}{4})) = 2T(1) + T(\frac{n}{4})$$

Time Complexity

Further example

HALVE(int n)

1: **while** $(n > 1)$ **do**
2: $\quad n = \frac{n}{2}$ *(pre-decimal point position)*

$$
\begin{aligned}
T(n) &= T(1) + T(\frac{n}{2}) \\
&= T(1) + (T(1) + T(\frac{n}{4})) = 2T(1) + T(\frac{n}{4}) \\
&= 2T(1) + (T(1) + T(\frac{n}{8})) = 3T(1) + T(\frac{n}{8})
\end{aligned}
$$

Time Complexity

Further example

HALVE(int n)

1: **while** $(n > 1)$ **do**
2: $\quad n = \frac{n}{2}$ *(pre-decimal point position)*

$$
\begin{aligned}
T(n) &= T(1) + T(\frac{n}{2}) \\
&= T(1) + (T(1) + T(\frac{n}{4})) = 2T(1) + T(\frac{n}{4}) \\
&= 2T(1) + (T(1) + T(\frac{n}{8})) = 3T(1) + T(\frac{n}{8}) \\
&= \ldots \\
&= NT(1) + T(\frac{n}{2^N}))
\end{aligned}
$$

Time Complexity

Further example

HALVE(int n)

1: **while** $(n > 1)$ **do**
2: $\quad n = \frac{n}{2}$ *(pre-decimal point position)*

$$
\begin{aligned}
T(n) &= T(1) + T(\frac{n}{2}) \\
&= T(1) + (T(1) + T(\frac{n}{4})) = 2T(1) + T(\frac{n}{4}) \\
&= 2T(1) + (T(1) + T(\frac{n}{8})) = 3T(1) + T(\frac{n}{8}) \\
&= \dots \\
&= NT(1) + T(\frac{n}{2^N}))
\end{aligned}
$$

How often can on repeat this, that is, what is N?

Time Complexity

Further example

HALVE(int n)

1: **while** $(n > 1)$ **do**
2: $\quad n = \frac{n}{2}$ *(pre-decimal point position)*

$$
\begin{aligned}
T(n) &= T(1) + T(\frac{n}{2}) \\
&= T(1) + (T(1) + T(\frac{n}{4})) = 2T(1) + T(\frac{n}{4}) \\
&= 2T(1) + (T(1) + T(\frac{n}{8})) = 3T(1) + T(\frac{n}{8}) \\
&= \dots \\
&= NT(1) + T(\frac{n}{2^N}))
\end{aligned}
$$

How often can on repeat this, that is, what is N?

Hence, we ask: When is "input for while" $\frac{n}{2^N} \leq 1$? Answer: $\frac{n}{2^N} \leq 1 \iff n \leq 2^N \iff log_2(n) \leq N$

Time Complexity

Further example

HALVE(int n)

1: **while** $(n > 1)$ **do**

2: $n = \frac{n}{2}$ *(pre-decimal point position)*

$$
\begin{aligned}
T(n) &= T(1) + T(\frac{n}{2}) \\
&= T(1) + (T(1) + T(\frac{n}{4})) = 2T(1) + T(\frac{n}{4}) \\
&= 2T(1) + (T(1) + T(\frac{n}{8})) = 3T(1) + T(\frac{n}{8}) \\
&= \dots \\
&= NT(1) + T(\frac{n}{2^N}))
\end{aligned}
$$

How often can on repeat this, that is, what is N?

Hence, we ask: When is "input for while" $\frac{n}{2^N} \leq 1$? Answer: $\frac{n}{2^N} \leq 1 \iff n \leq 2^N \iff log_2(n) \leq N$

Put $N = \log_2(n)$

$$
\begin{aligned}
T(n) &= NT(1) + T(\frac{n}{2^N}) \\
&= \log_2(n)T(1) + T(1) \\
&\in O(\log_2(n))
\end{aligned}
$$

Time Complexity

Further example

// SUM returns the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.
SUM(int n)

```
1: if (n = 1) then
2:     return 1
3: return n + SUM(n − 1)
```

$$
\begin{aligned}
T(n) &= T(1) + T(n-1) \\
&= T(1) + (T(1) + T(n-2) = 2T(1) + T(n-2) \\
&= \ldots \\
&= (n-1)T(1) + T(1) \in O(n) \text{ since } T(1) \in O(1)
\end{aligned}
$$

Time Complexity

Further example

In a similar way one may show that an algorithm as exponential runtime.

Time Complexity

Further example

In a similar way one may show that an algorithm as exponential runtime.

Example: Fibonacci Numbers $1,1,2,3,5,8,13,21,34,\ldots$

Fibonacci Numbers are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

Time Complexity

Further example

In a similar way one may show that an algorithm as exponential runtime.

Example: Fibonacci Numbers $1,1,2,3,5,8,13,21,34,\ldots$

Fibonacci Numbers are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way (there are more efficient algorithms (dynamic programming)):

FIB(int $n \geq 1$)

1: **if** $n \leq 2$ **then** $f = 1$
2: **else**
3: $\quad f = \text{FIB}(n-1) + \text{FIB}(n-2)$
4: **return** f

Time Complexity

Further example

In a similar way one may show that an algorithm as exponential runtime.

Example: Fibonacci Numbers $1,1,2,3,5,8,13,21,34,\ldots$

Fibonacci Numbers are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way (there are more efficient algorithms (dynamic programming)):

FIB(int $n \geq 1$)

1: **if** $n \leq 2$ **then** $f = 1$
2: **else**
3: $\quad f = \text{FIB}(n-1) + \text{FIB}(n-2)$
4: **return** f

$$T(n) = \Theta(1) + T(n-1) + T(n-2))$$

Time Complexity

Further example

In a similar way one may show that an algorithm as exponential runtime.

Example: Fibonacci Numbers $1,1,2,3,5,8,13,21,34,\ldots$

Fibonacci Numbers are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way (there are more efficient algorithms (dynamic programming)):

FIB(int $n \geq 1$)

1: **if** $n \leq 2$ **then** $f = 1$
2: **else**
3: $\quad f = \text{FIB}(n-1) + \text{FIB}(n-2)$
4: **return** f

$$\begin{aligned} T(n) &= \Theta(1) + T(n-1) + T(n-2)) \\ &\geq 2T(n-2) = 2(T(n-3) + T(n-4)) \end{aligned}$$

Time Complexity

Further example

In a similar way one may show that an algorithm as exponential runtime.

Example: Fibonacci Numbers $1,1,2,3,5,8,13,21,34,\ldots$

Fibonacci Numbers are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way (there are more efficient algorithms (dynamic programming)):

FIB(int $n \geq 1$)

1: **if** $n \leq 2$ **then** $f = 1$
2: **else**
3: $\quad f = \text{FIB}(n-1) + \text{FIB}(n-2)$
4: **return** f

$$\begin{aligned} T(n) &= \Theta(1) + T(n-1) + T(n-2)) \\ &\geq 2T(n-2) = 2(T(n-3) + T(n-4)) \\ &\geq 2(2T(n-4)) \end{aligned}$$

Time Complexity

Further example

In a similar way one may show that an algorithm as exponential runtime.

Example: Fibonacci Numbers $1,1,2,3,5,8,13,21,34,\ldots$

Fibonacci Numbers are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way (there are more efficient algorithms (dynamic programming)):

FIB(int $n \geq 1$)

1: **if** $n \leq 2$ **then** $f = 1$
2: **else**
3: $\quad f = \text{FIB}(n-1) + \text{FIB}(n-2)$
4: **return** f

$$\begin{aligned} T(n) &= \Theta(1) + T(n-1) + T(n-2)) \\ &\geq 2T(n-2) = 2(T(n-3) + T(n-4)) \\ &\geq 2(2T(n-4)) \\ &\geq \cdots \geq 2^{\frac{n}{2}} \in \Omega(2^{\frac{n}{2}}) \end{aligned}$$

Time Complexity

Further example

In a similar way one may show that an algorithm as exponential runtime.

Example: Fibonacci Numbers $1,1,2,3,5,8,13,21,34,\ldots$

Fibonacci Numbers are recursively defined:

- $f(1) = f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$, $n > 2$.

naive recursive way (there are more efficient algorithms (dynamic programming)):

FIB(int $n \geq 1$)

1: **if** $n \leq 2$ **then** $f = 1$
2: **else**
3: $\quad f = \text{FIB}(n-1) + \text{FIB}(n-2)$
4: **return** f

$$\begin{aligned} T(n) &= \Theta(1) + T(n-1) + T(n-2)) \\ &\geq 2T(n-2) = 2(T(n-3) + T(n-4)) \\ &\geq 2(2T(n-4)) \\ &\geq \cdots \geq 2^{\frac{n}{2}} \in \Omega(2^{\frac{n}{2}}) \end{aligned}$$

Hence, FIB has exponential runtime.

Time Complexity

Master Theorem

Helpful for recurrence relations in a very particular form, that often show up when analyzing recursive algorithms.

Time Complexity

Master Theorem

Helpful for recurrence relations in a very particular form, that often show up when analyzing recursive algorithms.

Let $a \geq 1$ and $b > 1$ be constants and let $T(n)$ be a function over the positive numbers defined by the recurrence

$$T(n) = aT(n/b) + \Theta(n^d).$$

Time Complexity

Master Theorem

Helpful for recurrence relations in a very particular form, that often show up when analyzing recursive algorithms.

Let $a \geq 1$ and $b > 1$ be constants and let $T(n)$ be a function over the positive numbers defined by the recurrence

$$T(n) = aT(n/b) + \Theta(n^d).$$

Then,

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Examples: Note $T(1) = \Theta(1)$.

Time Complexity

Master Theorem

Helpful for recurrence relations in a very particular form, that often show up when analyzing recursive algorithms.

Let $a \geq 1$ and $b > 1$ be constants and let $T(n)$ be a function over the positive numbers defined by the recurrence

$$T(n) = aT(n/b) + \Theta(n^d).$$

Then,

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Examples: Note $T(1) = \Theta(1)$.

- Exmpl HALVE: $T(n) = T(\frac{n}{2}) + \Theta(1)$. Here, $a = 1, b = 2, n^d = 1$ and thus, $d = 0$. We have $a = 1 = 2^0 = b^d$ and thus, $T(n) = \Theta(1 \log_2 n) = \Theta(\log_2 n)$

Time Complexity

Master Theorem

Helpful for recurrence relations in a very particular form, that often show up when analyzing recursive algorithms.

Let $a \geq 1$ and $b > 1$ be constants and let $T(n)$ be a function over the positive numbers defined by the recurrence

$$T(n) = aT(n/b) + \Theta(n^d).$$

Then,

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Examples: Note $T(1) = \Theta(1)$. Put d=2, b=3

a=8 $T(n) = 8T(\frac{n}{3}) + \Theta(n^2) \implies T(n) = \Theta(n^2)$

a=9 $T(n) = 9T(\frac{n}{3}) + \Theta(n^2) \implies T(n) = \Theta(n^2 \log_2 n)$

a=10 $T(n) = 10T(\frac{n}{3}) + \Theta(n^2) \implies T(n) = \Theta(n^{log_3(10)})$

Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs.

Similar to time complexity, space complexity is often expressed asymptotically in big-O notation

Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs.

Similar to time complexity, space complexity is often expressed asymptotically in big-O notation

SUM(int x, y, z)

1: **int** $r = x + y + z$
2: **return** r

Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs.

Similar to time complexity, space complexity is often expressed asymptotically in big-O notation

SUM(int x, y, z)

1: **int** $r = x + y + z$
2: **return** r

Requires 3 units of space for the parameters x, y, z and 1 for the local variable r.

Space complexity is in $O(1)$

Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs.

Similar to time complexity, space complexity is often expressed asymptotically in big-O notation

SUM(array a of length n)

```
1: int r = 0
2: for (int i = 0; i < n; ++i) do
3:     r = r + a[i];
4: return r
```

Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs.

Similar to time complexity, space complexity is often expressed asymptotically in big-O notation

SUM(array a of length n)

1: **int** $r = 0$
2: **for** (**int** $i = 0; i < n;$ ++i) **do**
3: $\quad r = r + a[i];$
4: **return** r

Requires n units of space for array a and 2 for the local variables r and i.

Space complexity is in $O(n)$

Space Complexity

Space complexity is a measure of the amount of working storage an algorithm needs.

Similar to time complexity, space complexity is often expressed asymptotically in big-O notation

But be careful here: If things are passed by pointer or reference, then space is shared.