Algorithms and Data Structures

Part 2: Sorting

Department of Mathematics
Stockholm University

Part 2: Sorting

Investigations by computer manufacturers and users have shown for many years that more than a quarter of commercially consumed computing time is dedicated to sorting operations.

It is therefore not surprising that significant efforts have been made to develop the most efficient procedures for sorting data using computers.

The accumulated knowledge about sorting algorithms now fills volumes. Still, new insights into sorting continue to appear in scientific journals, and numerous theoretically and practically important problems related to the task of sorting a set of data remain unresolved.

Part 2: Sorting

Investigations by computer manufacturers and users have shown for many years that more than a quarter of commercially consumed computing time is dedicated to sorting operations.

It is therefore not surprising that significant efforts have been made to develop the most efficient procedures for sorting data using computers.

The accumulated knowledge about sorting algorithms now fills volumes. Still, new insights into sorting continue to appear in scientific journals, and numerous theoretically and practically important problems related to the task of sorting a set of data remain unresolved.

Part 2: Sorting

Investigations by computer manufacturers and users have shown for many years that more than a quarter of commercially consumed computing time is dedicated to sorting operations.

It is therefore not surprising that significant efforts have been made to develop the most efficient procedures for sorting data using computers.

The accumulated knowledge about sorting algorithms now fills volumes. Still, new insights into sorting continue to appear in scientific journals, and numerous theoretically and practically important problems related to the task of sorting a set of data remain unresolved.

Part 2: The Sorting Problem

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

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.

Each record contains a key, which is the value to be sorted.

The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Part 2: The Sorting Problem

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.
Each record contains a key, which is the value to be sorted.
The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Part 2: The Sorting Problem

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.
Each record contains a key, which is the value to be sorted.
The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Part 2: The Sorting Problem

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.
Each record contains a key, which is the value to be sorted.
The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Part 2: The Sorting Problem

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.
Each record contains a key, which is the value to be sorted.
The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Part 2: The Sorting Problem

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.

Each record contains a key, which is the value to be sorted.

The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Part 2: The Sorting Problem

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

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.

Each record contains a key, which is the value to be sorted.

The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Example:

key (birth year)	sat-data (name)	sat-data (living town)
2000	Max	Linköping
1980	Anna	Uppsala
1988	Paula	Stockholm

Part 2: The Sorting Problem

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.

Each record contains a key, which is the value to be sorted.

The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Example:

key (birth year)	sat-data (name)	sat-data (living town)
1980	Anna	Uppsala
1988	Paula	Stockholm
2000	Max	Linköping

Part 2: The Sorting Problem

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.

Each record contains a key, which is the value to be sorted.

The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Example:

sat-data (birth year)	key (name)	sat-data (living town)
2000	Max	Linköping
1980	Anna	Uppsala
1988	Paula	Stockholm

Part 2: The Sorting Problem

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.

Each record contains a key, which is the value to be sorted.

The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Example:

sat-data (birth year)	key (name)	sat-data (living town)
1980	Anna	Uppsala
2000	Max	Linköping
1988	Paula	Stockholm

Part 2: The Sorting Problem

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

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

In practice, the numbers to be sorted are rarely isolated values.

We usually deal with a collection of data called a record.

Each record contains a key, which is the value to be sorted.

The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Example:

sat-data (birth year)	key (name)	sat-data (living town)
1980	Anna	Uppsala
2000	Max	Linköping
1988	Paula	Stockholm

To understand the principles of the basic sorting algorithms we focus here mainly on sequences of integers only.

Part 2: Insertion Sort

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

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

Recap (`Insertion_Sort`):

sorted list

5 2 4 6

2 5 4 6

2 4 5 6

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 2: Insertion Sort

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

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

Recap (`Insertion_Sort`):

sorted list

5 2 4 6

2 5 4 6

2 4 5 6

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`.

`Insertion_Sort` sorts the input numbers **in place**: it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time.

In the upcoming part, we introduce three more sorting algorithms (assuming n numbers need to be sorted):

- `Merge Sort`: not in place, but runtime $\Theta(n \log n)$
- `Heapsort`: in place, runtime $\Theta(n \log n)$
- `Quicksort`: in place, but worst-case runtime $\Theta(n^2)$. However, its expected runtime is $\Theta(n \log n)$ and in practice it outperforms heapsort

The latter algorithms (incl. insertion sort) are all *comparison sorts*: they determine the sorted order of an input array by comparing elements.

We then continue with proving a lower bound of $\Omega(n \log n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

This lower bound can be improved if one adds additional requirements on the input data and thus, if one can gather information about the sorted order of the input by means other than comparing elements. As an example, we consider

- `Counting Sort`: not in place, runtime $\Theta(n + k)$ in case the numbers to be sorted are in $\{1, \ldots, k\} \implies \Theta(n)$ if $k = O(n)$.

In the upcoming part, we introduce three more sorting algorithms (assuming n numbers need to be sorted):

`Merge Sort`: not in place, but runtime $\Theta(n \log n)$

`Heapsort`: in place, runtime $\Theta(n \log n)$

`Quicksort`: in place, but worst-case runtime $\Theta(n^2)$. However, its expected runtime is $\Theta(n \log n)$ and in practice it outperforms heapsort

The latter algorithms (incl. insertion sort) are all *comparison sorts*: they determine the sorted order of an input array by comparing elements.

We then continue with proving a lower bound of $\Omega(n \log n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

This lower bound can be improved if one adds additional requirements on the input data and thus, if one can gather information about the sorted order of the input by means other than comparing elements. As an example, we consider

`Counting Sort`: not in place, runtime $\Theta(n + k)$ in case the numbers to be sorted are in $\{1, \ldots, k\} \implies \Theta(n)$ if $k = O(n)$.

In the upcoming part, we introduce three more sorting algorithms (assuming n numbers need to be sorted):

- `Merge Sort`: not in place, but runtime $\Theta(n \log n)$
- `Heapsort`: in place, runtime $\Theta(n \log n)$
- `Quicksort`: in place, but worst-case runtime $\Theta(n^2)$. However, its expected runtime is $\Theta(n \log n)$ and in practice it outperforms heapsort

The latter algorithms (incl. insertion sort) are all *comparison sorts*: they determine the sorted order of an input array by comparing elements.

We then continue with proving a lower bound of $\Omega(n \log n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

This lower bound can be improved if one adds additional requirements on the input data and thus, if one can gather information about the sorted order of the input by means other than comparing elements. As an example, we consider

- `Counting Sort`: not in place, runtime $\Theta(n + k)$ in case the numbers to be sorted are in $\{1, \ldots, k\} \implies \Theta(n)$ if $k = O(n)$.

In the upcoming part, we introduce three more sorting algorithms (assuming n numbers need to be sorted):

- `Merge Sort`: not in place, but runtime $\Theta(n \log n)$
- `Heapsort`: in place, runtime $\Theta(n \log n)$
- `Quicksort`: in place, but worst-case runtime $\Theta(n^2)$. However, its expected runtime is $\Theta(n \log n)$ and in practice it outperforms heapsort

The latter algorithms (incl. insertion sort) are all *comparison sorts*: they determine the sorted order of an input array by comparing elements.

We then continue with proving a lower bound of $\Omega(n \log n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

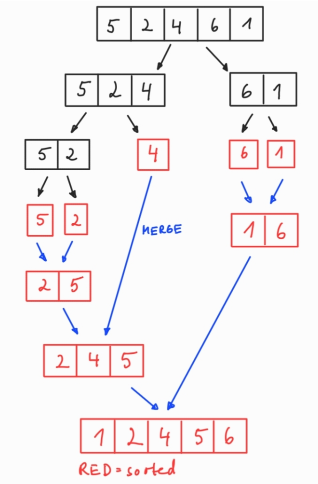
This lower bound can be improved if one adds additional requirements on the input data and thus, if one can gather information about the sorted order of the input by means other than comparing elements. As an example, we consider

- `Counting Sort`: not in place, runtime $\Theta(n + k)$ in case the numbers to be sorted are in $\{1, \ldots, k\} \implies \Theta(n)$ if $k = O(n)$.

Part 2: Merge Sort

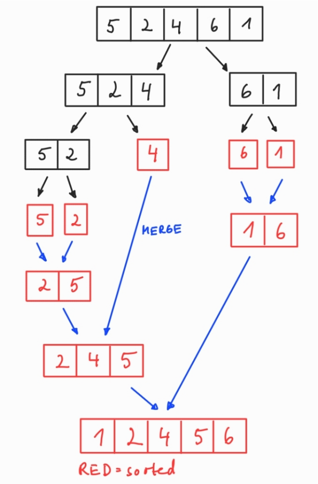
Part 2: Merge Sort

Main Idea:



Part 2: Merge Sort

Main Idea:



Merge sort is a classical example of divide-and-conquer approaches. The divide-and-conquer paradigm involves three steps at each level of the recursion:

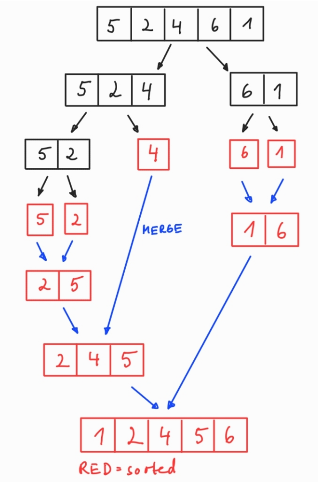
Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Merge Sort

Main Idea:



Merge sort is a classical example of divide-and-conquer approaches. The divide-and-conquer paradigm involves three steps at each level of the recursion:

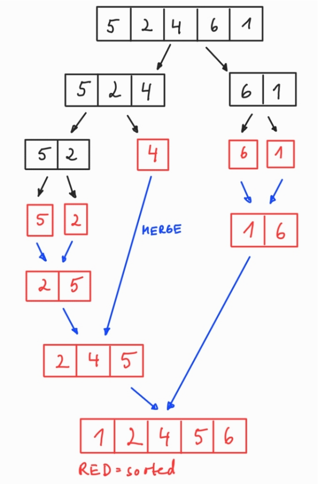
Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Merge Sort

Main Idea:



Merge sort is a classical example of divide-and-conquer approaches. The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

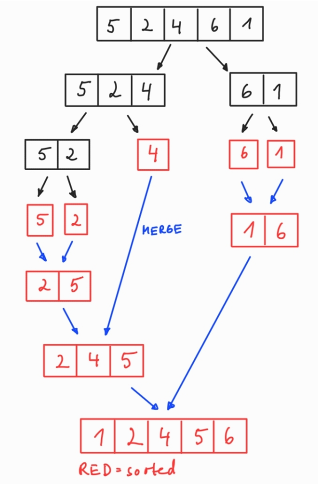
merge sort: Divide the n-element sequence to be sorted into two subsequences of approx. $n/2$ elements each

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Merge Sort

Main Idea:



Merge sort is a classical example of divide-and-conquer approaches. The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

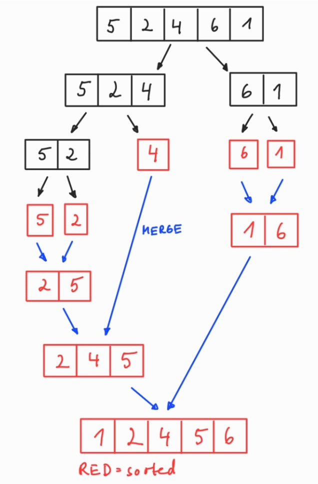
merge sort: Divide the n-element sequence to be sorted into two subsequences of approx. $n/2$ elements each

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Merge Sort

Main Idea:



Merge sort is a classical example of divide-and-conquer approaches. The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

merge sort: Divide the n-element sequence to be sorted into two subsequences of approx. $n/2$ elements each

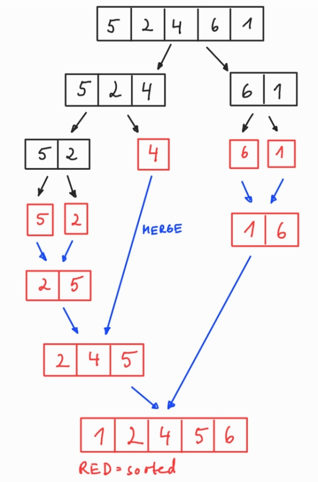
Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

merge sort: Sort the two subsequences recursively using merge sort

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Merge Sort

Main Idea:



Merge sort is a classical example of divide-and-conquer approaches. The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

merge sort: Divide the n-element sequence to be sorted into two subsequences of approx. $n/2$ elements each

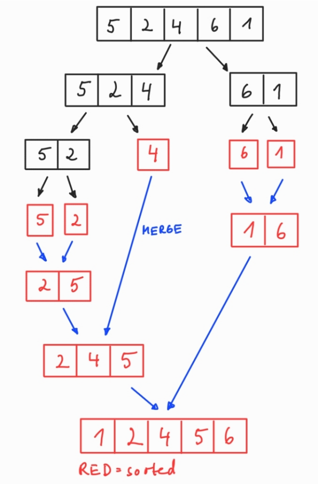
Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

merge sort: Sort the two subsequences recursively using merge sort

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Merge Sort

Main Idea:



Merge sort is a classical example of divide-and-conquer approaches. The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

merge sort: Divide the n-element sequence to be sorted into two subsequences of approx. $n/2$ elements each

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

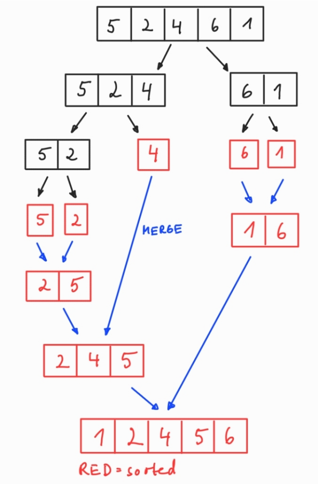
merge sort: Sort the two subsequences recursively using merge sort

Combine the solutions to the subproblems into the solution for the original problem.

merge sort: Merge the two sorted subsequences to produce the sorted answer

Part 2: Merge Sort

Main Idea:



Merge sort is a classical example of divide-and-conquer approaches. The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

merge sort: Divide the n-element sequence to be sorted into two subsequences of approx. $n/2$ elements each

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

merge sort: Sort the two subsequences recursively using merge sort

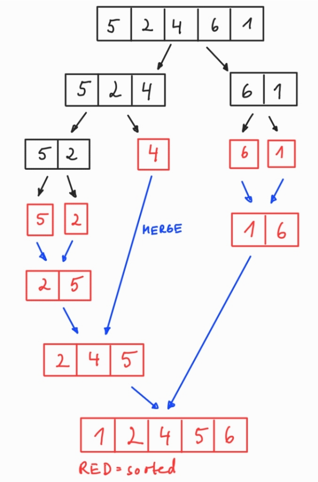
Combine the solutions to the subproblems into the solution for the original problem.

merge sort: Merge the two sorted subsequences to produce the sorted answer

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step.

Part 2: Merge Sort

Main Idea:



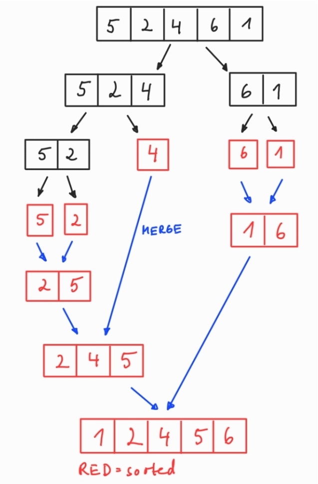
The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

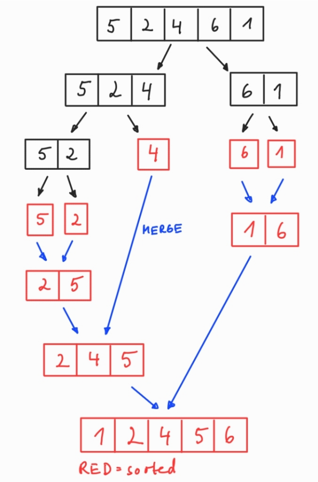
$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

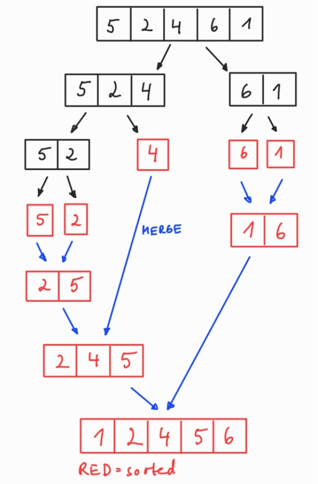
where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

Exmpl: $A[3..4] = (2, 4)$ and $A[5..7] = (1, 2, 7)$

take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

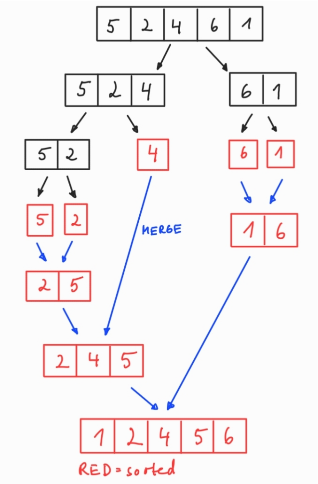
The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

Exmpl: $A[3..4] = (2, 4)$ and $A[5..7] = (1, 2, 7)$

take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

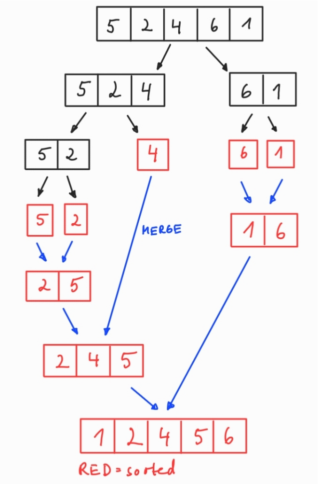
Exmpl: $A[3..4] = (2, 4)$ and $A[5..7] = (1, 2, 7)$

take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

1

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

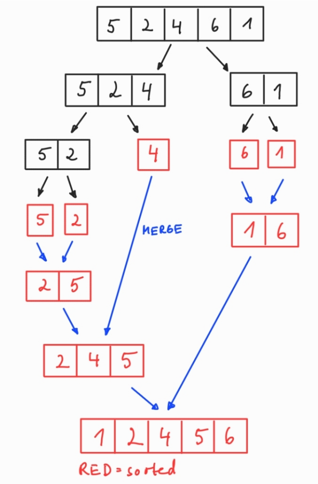
Exmpl: $A[3..4] = (2, 4)$ and $A[5..7] = (\cancel{1}, 2, 7)$

take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

1

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

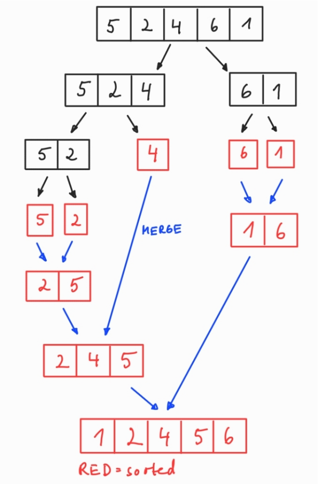
Exmpl: $A[3..4] = (2, 4)$ and $A[5..7] = (\cancel{1}, 2, 7)$

take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

$$1\ ,\ 2$$

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

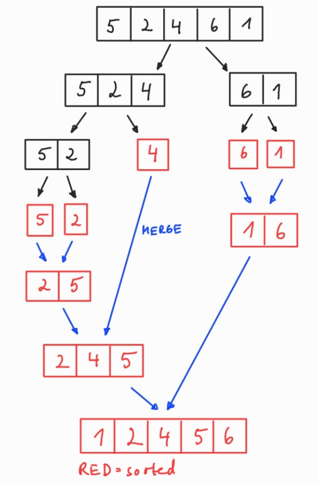
The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

Exmpl: $A[3..4] = (2, 4)$ and $A[5..7] = (\cancel{1}, \cancel{2}, 7)$
take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

$$1 \; , 2$$

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

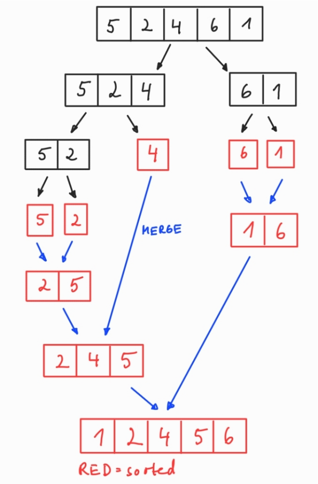
where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

Exmpl: $A[3..4] = (2, 4)$ and $A[5..7] = (\cancel{1}, \cancel{2}, 7)$
take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

1 , 2

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

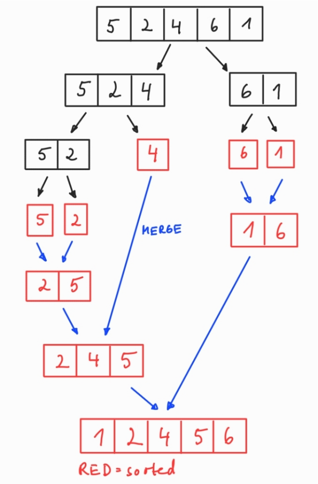
Exmpl: $A[3..4] = (2, 4)$ and $A[5..7] = (\cancel{1}, \cancel{2}, 7)$

take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

1 , 2 , 2

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

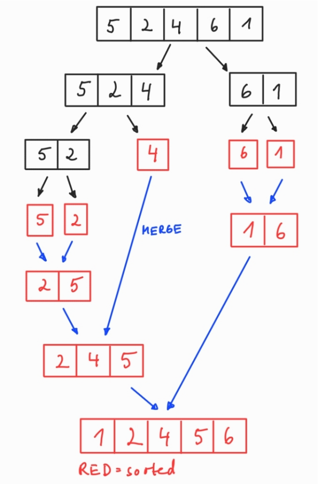
The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

Exmpl: $A[3..4] = (\cancel{2}, 4)$ and $A[5..7] = (\cancel{1}, \cancel{2}, 7)$
take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

1 , 2 , 2

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

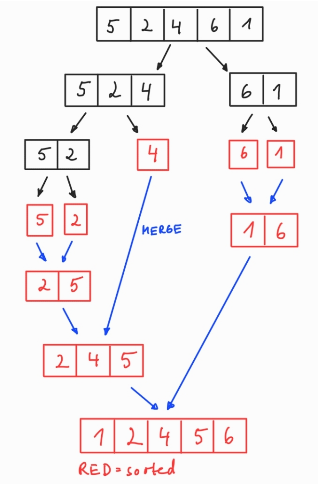
Exmpl: $A[3..4] = (\cancel{2}, 4)$ and $A[5..7] = (\cancel{1}, \cancel{2}, 7)$

take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

$$1 \, , 2 \, , 2 \, , 4$$

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

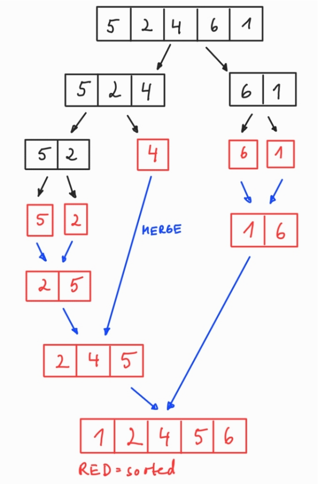
Exmpl: $A[3..4] = (\cancel{2}, \cancel{4})$ and $A[5..7] = (\cancel{1}, \cancel{2}, 7)$

take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

$$1\ ,\ 2\ ,\ 2\ ,\ 4$$

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

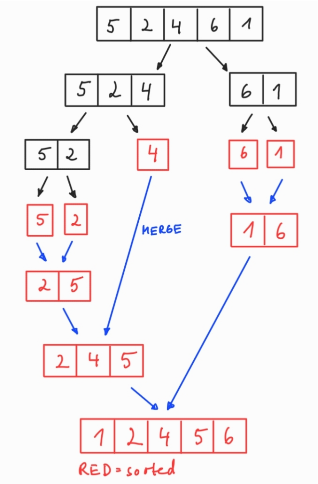
Exmpl: $A[3..4] = (\cancel{2}, \cancel{4})$ and $A[5..7] = (\cancel{1}, \cancel{2}, 7)$

take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

$$1\ ,\ 2\ ,\ 2\ ,\ 4\ ,\ 7$$

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

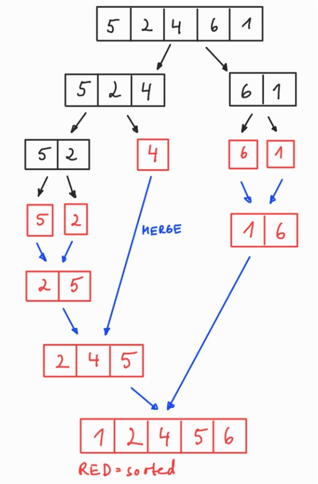
Exmpl: $A[3..4] = (\cancel{2}, \cancel{4})$ and $A[5..7] = (\cancel{1}, \cancel{2}, \cancel{7})$

take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

$$1\ ,\ 2\ ,\ 2\ ,\ 4\ ,\ 7$$

Part 2: Merge Sort

Main Idea:



The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

Exmpl: $A[3..4] = (\cancel{2}, \cancel{4})$ and $A[5..7] = (\cancel{1}, \cancel{2}, \cancel{7})$
take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

$$1 \, , 2 \, , 2 \, , 4 \, , 7$$

Each comparison: $\Theta(1)$ time
Thus, merging takes $\Theta(r - p + 1)$ time since we have in total $r - p + 1$ comparisons.

Part 2: Merge Sort

MERGE(A, p, q, r) //1st entry of array M is $M[1]$

1 $n_1 := q - p + 1$ //length of $A[p..q]$
2 $n_2 := r - q$ //length of $A[q+1..r]$
3 Init new arrays $L[1..n_1+1]$ and $R[1..n_2+1]$
4 FOR ($i = 1$ to n_1) DO $L[i] := A[p+i-1]$ //"copy" $A[p..q]$ to $L[1..n_1]$
5 FOR ($j = 1$ to n_2) DO $R[j] := A[q+j]$ //"copy" $A[q+1..r]$ to $R[1..n_2]$
6 $L[n_1+1] := \infty$ and $R[n_2+1] := \infty$ //to avoid: "array index out of bounds" in L9, 12 and to further increment i, resp, j when L, resp., R has been copied
7 $i := 1$ and $j := 1$, $k := p$
8 WHILE ($k \leq r$) DO //"merge" elements, while runs from $k = p..r$
9   IF($L[i] \leq R[j]$) THEN
10         $A[k] := L[i]$
11         $i := i + 1$
12   ELSE
13         $A[k] := R[j]$
14         $j := j + 1$
15     $k := k + 1$

Lemma

MERGE(A, p, q, r) *correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$ in $\Theta(r - p + 1)$ time.* *[proof next slide]*

Part 2: Merge Sort

```
MERGE(A, p, q, r)                                     //1st entry of array M is M[1]
1  n1 := q − p + 1                                    //length of A[p..q]
2  n2 := r − q                                        //length of A[q + 1..r]
3  Init new arrays L[1..n1 + 1] and R[1..n2 + 1]
4  FOR (i = 1 to n1) DO L[i] := A[p + i − 1]          //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]              //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞ //to avoid: "array index out of bounds" in
   L9, 12 and to further increment i, resp, j when L, resp., R has been copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO                     //"merge" elements, while runs from k = p..r
9       IF(L[i] ≤ R[j]) THEN
10          A[k] := L[i]
11          i := i + 1
12      ELSE
13          A[k] := R[j]
14          j := j + 1
15      k := k + 1
```

Lemma

MERGE(A, p, q, r) *correctly merges the sorted arrays* $A[p..q]$ *and* $A[q+1..r]$ *into the sorted array* $A[p..r]$ *in* $\Theta(r - p + 1)$ *time.*

[proof next slide]

Part 2: Merge Sort

MERGE(A, p, q, r)

1 $n_1 := q - p + 1$ //length of $A[p..q]$
2 $n_2 := r - q$ //length of $A[q+1..r]$
3 Init new arrays $L[1..n_1]$ and $R[1..n_2]$
4 FOR ($i = 1$ to n_1) DO $L[i] := A[p+i-1]$ //"copy" $A[p..q]$ to $L[1..n_1]$
5 FOR ($j = 1$ to n_2) DO $R[j] := A[q+j]$ //"copy" $A[q+1..r]$ to $R[1..n_2]$
6 $L[n_1+1] := \infty$ and $R[n_2+1] := \infty$ //to avoid: "array index out of bounds" in L9, 12 and to further increment i, resp, j when L, resp., R has been copied
7 $i := 1$ and $j := 1$, $k := p$
8 WHILE ($k \leq r$) DO //"merge" elements, while runs from $k = p, .., r$
9   IF($L[i] \leq R[j]$) THEN
10         $A[k] := L[i]$
11         $i := i + 1$
12   ELSE
13         $A[k] := R[j]$
14         $j := j + 1$
15     $k := k + 1$

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Part 2: Merge Sort

MERGE(A, p, q, r)

1 $n_1 := q - p + 1$ //length of $A[p..q]$
2 $n_2 := r - q$ //length of $A[q+1..r]$
3 Init new arrays $L[1..n_1]$ and $R[1..n_2]$
4 FOR ($i = 1$ to n_1) DO $L[i] := A[p+i-1]$ //"copy" $A[p..q]$ to $L[1..n_1]$
5 FOR ($j = 1$ to n_2) DO $R[j] := A[q+j]$ //"copy" $A[q+1..r]$ to $R[1..n_2]$
6 $L[n_1+1] := \infty$ and $R[n_2+1] := \infty$ //to avoid: "array index out of bounds" in L9, 12 and to further increment i, resp, j when L, resp., R has been copied
7 $i := 1$ and $j := 1$, $k := p$
8 WHILE ($k \leq r$) DO //"merge" elements, while runs from $k = p, .., r$
9   IF($L[i] \leq R[j]$) THEN
10         $A[k] := L[i]$
11         $i := i + 1$
12   ELSE
13         $A[k] := R[j]$
14         $j := j + 1$
15     $k := k + 1$

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9      IF(L[i] ≤ R[j]) THEN
10         A[k] := L[i]
11         i := i + 1
12     ELSE
13         A[k] := R[j]
14         j := j + 1
15     k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Part 2: Merge Sort

MERGE(A, p, q, r)

1 $n_1 := q - p + 1$ //length of $A[p..q]$
2 $n_2 := r - q$ //length of $A[q+1..r]$
3 Init new arrays $L[1..n_1]$ and $R[1..n_2]$
4 FOR ($i = 1$ to n_1) DO $L[i] := A[p+i-1]$
//"copy" $A[p..q]$ to $L[1..n_1]$
5 FOR ($j = 1$ to n_2) DO $R[j] := A[q+j]$
//"copy" $A[q+1..r]$ to $R[1..n_2]$
6 $L[n_1+1] := \infty$ and $R[n_2+1] := \infty$
//to avoid: "array index out of bounds" in L9, 12 and to further increment i, resp, j when L, resp., R has been copied
7 $i := 1$ and $j := 1$, $k := p$
8 WHILE ($k \leq r$) DO //"merge" elements, while runs from $k = p, .., r$
9   IF($L[i] \leq R[j]$) THEN
10     $A[k] := L[i]$
11     $i := i + 1$
12   ELSE
13     $A[k] := R[j]$
14     $j := j + 1$
15   $k := k + 1$

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

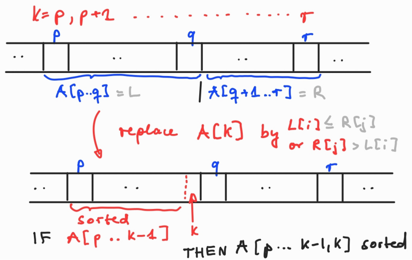
Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.



Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9      IF(L[i] ≤ R[j]) THEN
10         A[k] := L[i]
11         i := i + 1
12     ELSE
13         A[k] := R[j]
14         j := j + 1
15     k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

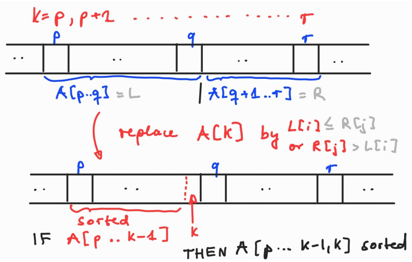
Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.



```
MERGE(A, p, q, r)
1   n1 := q − p + 1 //length of A[p..q]
2   n2 := r − q //length of A[q + 1..r]
3   Init new arrays L[1..n1] and R[1..n2]
4   FOR (i = 1 to n1) DO L[i] := A[p+i−1]
    //"copy" A[p..q] to L[1..n1]
5   FOR (j = 1 to n2) DO R[j] := A[q + j]
    //"copy" A[q + 1..r] to R[1..n2]
6   L[n1 + 1] := ∞ and R[n2 + 1] := ∞
    //to avoid: "array index out of bounds"
    in L9, 12 and to further increment i,
    resp, j when L, resp., R has been
    copied
7   i := 1 and j := 1, k := p
8   WHILE (k ≤ r) DO //"merge" elements,
    while runs from k = p, .., r
9         IF(L[i] ≤ R[j]) THEN
10             A[k] := L[i]
11             i := i + 1
12        ELSE
13             A[k] := R[j]
14             j := j + 1
15        k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

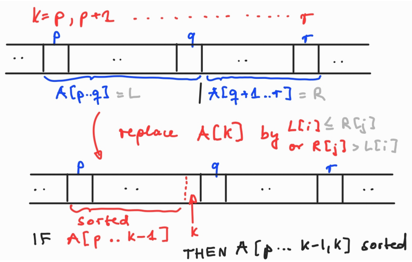
Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.



```
MERGE(A, p, q, r)
1   n1 := q − p + 1 //length of A[p..q]
2   n2 := r − q //length of A[q + 1..r]
3   Init new arrays L[1..n1] and R[1..n2]
4   FOR (i = 1 to n1) DO L[i] := A[p+i−1]
    //"copy" A[p..q] to L[1..n1]
5   FOR (j = 1 to n2) DO R[j] := A[q + j]
    //"copy" A[q + 1..r] to R[1..n2]
6   L[n1 + 1] := ∞ and R[n2 + 1] := ∞
    //to avoid: "array index out of bounds"
    in L9, 12 and to further increment i,
    resp, j when L, resp., R has been
    copied
7   i := 1 and j := 1, k := p
8   WHILE (k ≤ r) DO //"merge" elements,
    while runs from k = p, .., r
9       IF(L[i] ≤ R[j]) THEN
10          A[k] := L[i]
11          i := i + 1
12      ELSE
13          A[k] := R[j]
14          j := j + 1
15      k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

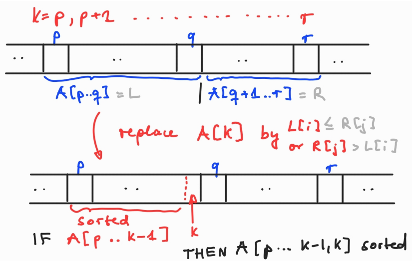
Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.



```
MERGE(A, p, q, r)
 1  n1 := q − p + 1 //length of A[p..q]
 2  n2 := r − q //length of A[q + 1..r]
 3  Init new arrays L[1..n1] and R[1..n2]
 4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
    //"copy" A[p..q] to L[1..n1]
 5  FOR (j = 1 to n2) DO R[j] := A[q + j]
    //"copy" A[q + 1..r] to R[1..n2]
 6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
    //to avoid: "array index out of bounds"
    in L9, 12 and to further increment i,
    resp, j when L, resp., R has been
    copied
 7  i := 1 and j := 1, k := p
 8  WHILE (k ≤ r) DO //"merge" elements,
    while runs from k = p, .., r
 9      IF(L[i] ≤ R[j]) THEN
10          A[k] := L[i]
11          i := i + 1
12      ELSE
13          A[k] := R[j]
14          j := j + 1
15      k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

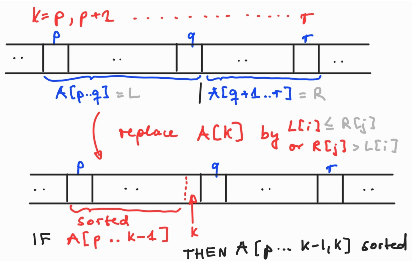
Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.



Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9       IF(L[i] ≤ R[j]) THEN
10          A[k] := L[i]
11          i := i + 1
12      ELSE
13          A[k] := R[j]
14          j := j + 1
15      k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

[I] & [II] hold when initializing first run of while-loop:

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n_1 := q − p + 1 //length of A[p..q]
2  n_2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n_1] and R[1..n_2]
4  FOR (i = 1 to n_1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n_1]
5  FOR (j = 1 to n_2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n_2]
6  L[n_1 + 1] := ∞ and R[n_2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9        IF(L[i] ≤ R[j]) THEN
10             A[k] := L[i]
11             i := i + 1
12       ELSE
13             A[k] := R[j]
14             j := j + 1
15       k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

[I] & [II] hold when initializing first run of while-loop:

$k = p$ (just k initialized in L7, no run of while-loop so-far) $\Rightarrow$ $A[p..p-1]$ "empty" and thus has $0 = k - p$ elements $\Rightarrow$ [I] holds

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9        IF(L[i] ≤ R[j]) THEN
10            A[k] := L[i]
11            i := i + 1
12       ELSE
13            A[k] := R[j]
14            j := j + 1
15       k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

[I] & [II] hold when initializing first run of while-loop:

$k = p$ (just k initialized in L7, no run of while-loop so-far) $\Rightarrow$ $A[p..p-1]$ "empty" and thus has $0 = k - p$ elements $\Rightarrow$ [I] holds

Since $i = 1$, $j = 1$ and $A[p..k-1]$ empty so-far and since L, R are sorted $\Rightarrow$ [II] holds

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9      IF(L[i] ≤ R[j]) THEN
10         A[k] := L[i]
11         i := i + 1
12     ELSE
13         A[k] := R[j]
14         j := j + 1
15     k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

[I] & [II] are maintained when running the while-loop:

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9       IF(L[i] ≤ R[j]) THEN
10           A[k] := L[i]
11           i := i + 1
12      ELSE
13           A[k] := R[j]
14           j := j + 1
15      k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i], R[j]$ are the smallest elements of their arrays that have not been copied back into A.

[I] & [II] are maintained when running the while-loop:

Two Cases: $L[i] \leq R[j]$ or $L[i] > R[j]$. Assume that $L[i] \leq R[j]$ in L9

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9       IF(L[i] ≤ R[j]) THEN
10          A[k] := L[i]
11          i := i + 1
12      ELSE
13          A[k] := R[j]
14          j := j + 1
15      k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

[I] & [II] are maintained when running the while-loop:

Two Cases: $L[i] \leq R[j]$ or $L[i] > R[j]$. Assume that $L[i] \leq R[j]$ in L9

"By induction": $L[i]$ is the smallest element not yet copied back into A ([I] holds) and in L10 $L[i]$ is copied back to A ($A[k] := L[i]$)

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9        IF(L[i] ≤ R[j]) THEN
10            A[k] := L[i]
11            i := i + 1
12       ELSE
13            A[k] := R[j]
14            j := j + 1
15       k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

[I] & [II] are maintained when running the while-loop:

Two Cases: $L[i] \leq R[j]$ or $L[i] > R[j]$. Assume that $L[i] \leq R[j]$ in L9

"By induction": $L[i]$ is the smallest element not yet copied back into A ([I] holds) and in L10 $L[i]$ is copied back to A ($A[k] := L[i]$)

This with $L[i] \leq R[j]$ and the fact that, by [I], $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ implies that $A[p..k]$ contains now the $k-p+1$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, . . . , r
9      IF(L[i] ≤ R[j]) THEN
10         A[k] := L[i]
11         i := i + 1
12     ELSE
13         A[k] := R[j]
14         j := j + 1
15     k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

[I] & [II] are maintained when running the while-loop:

Two Cases: $L[i] \leq R[j]$ or $L[i] > R[j]$. Assume that $L[i] \leq R[j]$ in L9

"By induction": $L[i]$ is the smallest element not yet copied back into A ([I] holds) and in L10 $L[i]$ is copied back to A ($A[k] := L[i]$)

This with $L[i] \leq R[j]$ and the fact that, by [I], $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ implies that $A[p..k]$ contains now the $k-p+1$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$

Incrementing k (in L15) and i (in L11) restablishes the loop invariant for the next iteration.

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9        IF(L[i] ≤ R[j]) THEN
10           A[k] := L[i]
11           i := i + 1
12       ELSE
13           A[k] := R[j]
14           j := j + 1
15       k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

[I] & [II] are maintained when running the while-loop:

Two Cases: $L[i] \leq R[j]$ or $L[i] > R[j]$. Assume that $L[i] \leq R[j]$ in L9

"By induction": $L[i]$ is the smallest element not yet copied back into A ([I] holds) and in L10 $L[i]$ is copied back to A ($A[k] := L[i]$)

This with $L[i] \leq R[j]$ and the fact that, by [I], $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ implies that $A[p..k]$ contains now the $k-p+1$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$

Incrementing k (in L15) and i (in L11) restablishes the loop invariant for the next iteration.

If instead $L[i] > R[j]$ then L13-14 perform the appropriate action to maintain the loop invariant.

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1   n1 := q − p + 1 //length of A[p..q]
2   n2 := r − q //length of A[q + 1..r]
3   Init new arrays L[1..n1] and R[1..n2]
4   FOR (i = 1 to n1) DO L[i] := A[p+i−1]
    //"copy" A[p..q] to L[1..n1]
5   FOR (j = 1 to n2) DO R[j] := A[q + j]
    //"copy" A[q + 1..r] to R[1..n2]
6   L[n1 + 1] := ∞ and R[n2 + 1] := ∞
    //to avoid: "array index out of bounds"
    in L9, 12 and to further increment i,
    resp, j when L, resp., R has been
    copied
7   i := 1 and j := 1, k := p
8   WHILE (k ≤ r) DO //"merge" elements,
    while runs from k = p, .., r
9       IF(L[i] ≤ R[j]) THEN
10          A[k] := L[i]
11          i := i + 1
12      ELSE
13          A[k] := R[j]
14          j := j + 1
15      k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Since [I] & [II] hold in each step, we have at termination of while-loop:

Part 2: Merge Sort

```
MERGE(A, p, q, r)
 1  n1 := q − p + 1 //length of A[p..q]
 2  n2 := r − q //length of A[q + 1..r]
 3  Init new arrays L[1..n1] and R[1..n2]
 4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
    //"copy" A[p..q] to L[1..n1]
 5  FOR (j = 1 to n2) DO R[j] := A[q + j]
    //"copy" A[q + 1..r] to R[1..n2]
 6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
    //to avoid: "array index out of bounds"
    in L9, 12 and to further increment i,
    resp, j when L, resp., R has been
    copied
 7  i := 1 and j := 1, k := p
 8  WHILE (k ≤ r) DO //"merge" elements,
    while runs from k = p, .., r
 9      IF(L[i] ≤ R[j]) THEN
10          A[k] := L[i]
11          i := i + 1
12      ELSE
13          A[k] := R[j]
14          j := j + 1
15      k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Since [I] & [II] hold in each step, we have at termination of while-loop:

After termination, we have $k = r + 1$

By the loop-invariant, $A[p..k-1] = A[p..r]$ contains the $k - p = r - p + 1$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n_1 := q − p + 1 //length of A[p..q]
2  n_2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n_1] and R[1..n_2]
4  FOR (i = 1 to n_1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n_1]
5  FOR (j = 1 to n_2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n_2]
6  L[n_1 + 1] := ∞ and R[n_2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9      IF(L[i] ≤ R[j]) THEN
10         A[k] := L[i]
11         i := i + 1
12     ELSE
13         A[k] := R[j]
14         j := j + 1
15     k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Since [I] & [II] hold in each step, we have at termination of while-loop:

After termination, we have $k = r + 1$

By the loop-invariant, $A[p..k-1] = A[p..r]$ contains the $k - p = r - p + 1$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order

Since $L[1..n_1]$ and $R[1..n_2]$ have together $n_1 + n_2 = r - p + 1$ elements
$\implies$ $A[p..k-1] = A[p..r]$ contains the $r - p + 1$ smallest elements and thus ALL elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order.

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9        IF(L[i] ≤ R[j]) THEN
10           A[k] := L[i]
11           i := i + 1
12       ELSE
13           A[k] := R[j]
14           j := j + 1
15       k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Since [I] & [II] hold in each step, we have at termination of while-loop:

After termination, we have $k = r + 1$

By the loop-invariant, $A[p..k-1] = A[p..r]$ contains the $k - p = r - p + 1$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order

Since $L[1..n_1]$ and $R[1..n_2]$ have together $n_1 + n_2 = r - p + 1$ elements
$\implies$ $A[p..k-1] = A[p..r]$ contains the $r - p + 1$ smallest elements and thus ALL elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order.

Since $L[1..n_1]$ and $R[1..n_2]$ contain all elements of $A[p..r]$ $\implies$ $A[p..r]$ is now sorted.

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1   n1 := q − p + 1 //length of A[p..q]
2   n2 := r − q //length of A[q + 1..r]
3   Init new arrays L[1..n1] and R[1..n2]
4   FOR (i = 1 to n1) DO L[i] := A[p+i−1]
    //"copy" A[p..q] to L[1..n1]
5   FOR (j = 1 to n2) DO R[j] := A[q + j]
    //"copy" A[q + 1..r] to R[1..n2]
6   L[n1 + 1] := ∞ and R[n2 + 1] := ∞
    //to avoid: "array index out of bounds"
    in L9, 12 and to further increment i,
    resp, j when L, resp., R has been
    copied
7   i := 1 and j := 1, k := p
8   WHILE (k ≤ r) DO //"merge" elements,
    while runs from k = p, .., r
9       IF(L[i] ≤ R[j]) THEN
10          A[k] := L[i]
11          i := i + 1
12      ELSE
13          A[k] := R[j]
14          j := j + 1
15      k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

$\Longrightarrow$ MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Part 2: Merge Sort

MERGE(A, p, q, r)

1 $n_1 := q - p + 1$ //length of $A[p..q]$
2 $n_2 := r - q$ //length of $A[q+1..r]$
3 Init new arrays $L[1..n_1]$ and $R[1..n_2]$
4 FOR ($i = 1$ to n_1) DO $L[i] := A[p+i-1]$ //"copy" $A[p..q]$ to $L[1..n_1]$
5 FOR ($j = 1$ to n_2) DO $R[j] := A[q+j]$ //"copy" $A[q+1..r]$ to $R[1..n_2]$
6 $L[n_1+1] := \infty$ and $R[n_2+1] := \infty$ //to avoid: "array index out of bounds" in L9, 12 and to further increment i, resp, j when L, resp., R has been copied
7 $i := 1$ and $j := 1$, $k := p$
8 WHILE ($k \leq r$) DO //"merge" elements, while runs from $k = p, .., r$
9   IF($L[i] \leq R[j]$) THEN
10         $A[k] := L[i]$
11         $i := i + 1$
12   ELSE
13         $A[k] := R[j]$
14         $j := j + 1$
15     $k := k + 1$

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Show Runtime: Let $N = n_1 + n_2$. It suffices to use N and this is, in particular, helpful when analyzing merge-sort

Part 2: Merge Sort

```
MERGE(A, p, q, r)
 1  n_1 := q − p + 1 //length of A[p..q]
 2  n_2 := r − q //length of A[q + 1..r]
 3  Init new arrays L[1..n_1] and R[1..n_2]
 4  FOR (i = 1 to n_1) DO L[i] := A[p+i−1]
    //"copy" A[p..q] to L[1..n_1]
 5  FOR (j = 1 to n_2) DO R[j] := A[q + j]
    //"copy" A[q + 1..r] to R[1..n_2]
 6  L[n_1 + 1] := ∞ and R[n_2 + 1] := ∞
    //to avoid: "array index out of bounds"
    in L9, 12 and to further increment i,
    resp, j when L, resp., R has been
    copied
 7  i := 1 and j := 1, k := p
 8  WHILE (k ≤ r) DO //"merge" elements,
    while runs from k = p, .., r
 9      IF(L[i] ≤ R[j]) THEN
10          A[k] := L[i]
11          i := i + 1
12      ELSE
13          A[k] := R[j]
14          j := j + 1
15      k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Show Runtime: Let $N = n_1 + n_2$. It suffices to use N and this is, in particular, helpful when analyzing merge-sort

L1,2,6,7,9-15: each $\Theta(1)$

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1   n1 := q − p + 1 //length of A[p..q]
2   n2 := r − q //length of A[q + 1..r]
3   Init new arrays L[1..n1] and R[1..n2]
4   FOR (i = 1 to n1) DO L[i] := A[p+i−1]
    //"copy" A[p..q] to L[1..n1]
5   FOR (j = 1 to n2) DO R[j] := A[q + j]
    //"copy" A[q + 1..r] to R[1..n2]
6   L[n1 + 1] := ∞ and R[n2 + 1] := ∞
    //to avoid: "array index out of bounds"
    in L9, 12 and to further increment i,
    resp, j when L, resp., R has been
    copied
7   i := 1 and j := 1, k := p
8   WHILE (k ≤ r) DO //"merge" elements,
    while runs from k = p, .., r
9        IF(L[i] ≤ R[j]) THEN
10            A[k] := L[i]
11            i := i + 1
12       ELSE
13            A[k] := R[j]
14            j := j + 1
15       k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i], R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Show Runtime: Let $N = n_1 + n_2$. It suffices to use N and this is, in particular, helpful when analyzing merge-sort

L1,2,6,7,9-15: each $\Theta(1)$

L3-5: $\Theta(n_1) + \Theta(n_2) = \Theta(n_1 + n_2) = \Theta(N)$.

Part 2: Merge Sort

MERGE(A, p, q, r)

1 $n_1 := q - p + 1$ //length of $A[p..q]$
2 $n_2 := r - q$ //length of $A[q+1..r]$
3 Init new arrays $L[1..n_1]$ and $R[1..n_2]$
4 FOR ($i = 1$ to n_1) DO $L[i] := A[p+i-1]$
//"copy" $A[p..q]$ to $L[1..n_1]$
5 FOR ($j = 1$ to n_2) DO $R[j] := A[q+j]$
//"copy" $A[q+1..r]$ to $R[1..n_2]$
6 $L[n_1+1] := \infty$ and $R[n_2+1] := \infty$
//to avoid: "array index out of bounds" in L9, 12 and to further increment i, resp, j when L, resp., R has been copied
7 $i := 1$ and $j := 1$, $k := p$
8 WHILE ($k \leq r$) DO //"merge" elements, while runs from $k = p, \ldots, r$
9   IF($L[i] \leq R[j]$) THEN
10         $A[k] := L[i]$
11         $i := i + 1$
12   ELSE
13         $A[k] := R[j]$
14         $j := j + 1$
15     $k := k + 1$

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Show Runtime: Let $N = n_1 + n_2$. It suffices to use N and this is, in particular, helpful when analyzing merge-sort

L1,2,6,7,9-15: each $\Theta(1)$

L3-5: $\Theta(n_1) + \Theta(n_2) = \Theta(n_1 + n_2) = \Theta(N)$.

While-loop in L8 runs $r - p = (n_2 + q) - (-n_1 + q - 1) = n_1 + n_2 + 1 = \Theta(N)$ times (for latter equation see L1,2)

Part 2: Merge Sort

MERGE(A, p, q, r)
1 $n_1 := q - p + 1$ //length of $A[p..q]$
2 $n_2 := r - q$ //length of $A[q+1..r]$
3 Init new arrays $L[1..n_1]$ and $R[1..n_2]$
4 FOR ($i = 1$ to n_1) DO $L[i] := A[p+i-1]$ //"copy" $A[p..q]$ to $L[1..n_1]$
5 FOR ($j = 1$ to n_2) DO $R[j] := A[q+j]$ //"copy" $A[q+1..r]$ to $R[1..n_2]$
6 $L[n_1+1] := \infty$ and $R[n_2+1] := \infty$ //to avoid: "array index out of bounds" in L9, 12 and to further increment i, resp, j when L, resp., R has been copied
7 $i := 1$ and $j := 1$, $k := p$
8 WHILE ($k \leq r$) DO //"merge" elements, while runs from $k = p, \dots, r$
9   IF($L[i] \leq R[j]$) THEN
10         $A[k] := L[i]$
11         $i := i + 1$
12   ELSE
13         $A[k] := R[j]$
14         $j := j + 1$
15     $k := k + 1$

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i], R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Show Runtime: Let $N = n_1 + n_2$. It suffices to use N and this is, in particular, helpful when analyzing merge-sort

L1,2,6,7,9-15: each $\Theta(1)$

L3-5: $\Theta(n_1) + \Theta(n_2) = \Theta(n_1 + n_2) = \Theta(N)$.

While-loop in L8 runs $r - p = (n_2 + q) - (-n_1 + q - 1) = n_1 + n_2 + 1 = \Theta(N)$ times (for latter equation see L1,2)
Since all tasks within this while-loop take constant time we have runtime $\Theta(N)$ for L8-14.

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1  n1 := q − p + 1 //length of A[p..q]
2  n2 := r − q //length of A[q + 1..r]
3  Init new arrays L[1..n1] and R[1..n2]
4  FOR (i = 1 to n1) DO L[i] := A[p+i−1]
   //"copy" A[p..q] to L[1..n1]
5  FOR (j = 1 to n2) DO R[j] := A[q + j]
   //"copy" A[q + 1..r] to R[1..n2]
6  L[n1 + 1] := ∞ and R[n2 + 1] := ∞
   //to avoid: "array index out of bounds"
   in L9, 12 and to further increment i,
   resp, j when L, resp., R has been
   copied
7  i := 1 and j := 1, k := p
8  WHILE (k ≤ r) DO //"merge" elements,
   while runs from k = p, .., r
9      IF(L[i] ≤ R[j]) THEN
10         A[k] := L[i]
11         i := i + 1
12     ELSE
13         A[k] := R[j]
14         j := j + 1
15     k := k + 1
```

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Show Runtime: Let $N = n_1 + n_2$. It suffices to use N and this is, in particular, helpful when analyzing merge-sort

L1,2,6,7,9-15: each $\Theta(1)$

L3-5: $\Theta(n_1) + \Theta(n_2) = \Theta(n_1 + n_2) = \Theta(N)$.

While-loop in L8 runs $r - p = (n_2 + q) - (-n_1 + q - 1) = n_1 + n_2 + 1 = \Theta(N)$ times (for latter equation see L1,2)
Since all tasks within this while-loop take constant time we have runtime $\Theta(N)$ for L8-14.

$\Longrightarrow$ overall runtime $\Theta(N)$.

Part 2: Merge Sort

```
MERGE(A, p, q, r)
1   n1 := q − p + 1 //length of A[p..q]
2   n2 := r − q //length of A[q + 1..r]
3   Init new arrays L[1..n1] and R[1..n2]
4   FOR (i = 1 to n1) DO L[i] := A[p+i−1]
    //"copy" A[p..q] to L[1..n1]
5   FOR (j = 1 to n2) DO R[j] := A[q + j]
    //"copy" A[q + 1..r] to R[1..n2]
6   L[n1 + 1] := ∞ and R[n2 + 1] := ∞
    //to avoid: "array index out of bounds"
    in L9, 12 and to further increment i,
    resp, j when L, resp., R has been
    copied
7   i := 1 and j := 1, k := p
8   WHILE (k ≤ r) DO //"merge" elements,
    while runs from k = p, . . , r
9       IF(L[i] ≤ R[j]) THEN
10          A[k] := L[i]
11          i := i + 1
12      ELSE
13          A[k] := R[j]
14          j := j + 1
15      k := k + 1
```

In summary, we have shown:

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

- [I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and
- [II] $L[i]$, $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

Lemma

MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$ in $\Theta(N)$ time with N being the sum of the length of $A[q+1..r]$ and $A[p..r]$.

Part 2: Merge Sort

We can now use the `MERGE` procedure as a subroutine in the merge sort algorithm `MERGE_SORT`(A, p, r).

Part 2: Merge Sort

We can now use the `MERGE` procedure as a subroutine in the merge sort algorithm `MERGE_SORT`(A, p, r).

2 Cases:

$p \geq r$**:** Then $A[p..r]$ has 0 or 1 elements and is thus sorted

$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.

In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:

$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

Part 2: Merge Sort

We can now use the `MERGE` procedure as a subroutine in the merge sort algorithm `MERGE_SORT`(A, p, r).

2 Cases:

$p \geq r$**:** Then $A[p..r]$ has 0 or 1 elements and is thus sorted

$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.

In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:

$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q + 1..r]$ of size $\lfloor N/2 \rfloor$

Part 2: Merge Sort

We can now use the `MERGE` procedure as a subroutine in the merge sort algorithm `MERGE_SORT`(A, p, r).

2 Cases:

$p \geq r$**:** Then $A[p..r]$ has 0 or 1 elements and is thus sorted

$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.

In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:

$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

$\lceil x \rceil$ denotes the least integer greater than or equal to x

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Part 2: Merge Sort

We can now use the MERGE procedure as a subroutine in the merge sort algorithm MERGE_SORT(A, p, r).

2 Cases:

$p \geq r$: Then $A[p..r]$ has 0 or 1 elements and is thus sorted

$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.

In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:

$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

```
MERGE_SORT(A, p, r)
    IF(p < r) THEN
        q = ⌊(p + r)/2⌋
        MERGE_SORT(A, p, q)
        MERGE_SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

$\lceil x \rceil$ denotes the least integer greater than or equal to x

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Part 2: Merge Sort

We can now use the MERGE procedure as a subroutine in the merge sort algorithm MERGE_SORT(A, p, r).

2 Cases:

$p \geq r$**:** Then $A[p..r]$ has 0 or 1 elements and is thus sorted

$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.

In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:

$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

```
MERGE_SORT(A, p, r)
IF(p < r) THEN
    q = ⌊(p + r)/2⌋
    MERGE_SORT(A, p, q)
    MERGE_SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Start algorithm by calling MERGE_SORT(A, 1, $A.length$)

$\lceil x \rceil$ denotes the least integer greater than or equal to x

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Part 2: Merge Sort

We can now use the MERGE procedure as a subroutine in the merge sort algorithm MERGE_SORT(A, p, r).

2 Cases:

$p \geq r$: Then $A[p..r]$ has 0 or 1 elements and is thus sorted

$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.

In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:

$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

```
MERGE_SORT(A, p, r)
    IF(p < r) THEN
        q = ⌊(p + r)/2⌋
        MERGE_SORT(A, p, q)
        MERGE_SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Start algorithm by calling MERGE_SORT(A, 1, $A.length$)

$\lceil x \rceil$ denotes the least integer greater than or equal to x

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

5	2	4

1 MERGE_SORT$(A, \overset{p}{1}, \overset{r}{3})$

$p < r \Rightarrow q = \lfloor \frac{1+3}{2} \rfloor = 2$

Part 2: Merge Sort

We can now use the `MERGE` procedure as a subroutine in the merge sort algorithm `MERGE_SORT`(A, p, r).

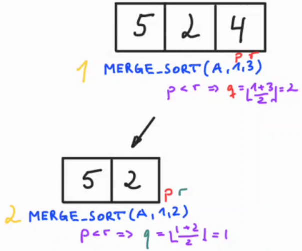
2 Cases:

$p \geq r$: Then $A[p..r]$ has 0 or 1 elements and is thus sorted

$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.
In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:
$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

```
MERGE_SORT(A, p, r)
    IF(p < r) THEN
        q = ⌊(p + r)/2⌋
        MERGE_SORT(A, p, q)
        MERGE_SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Start algorithm by calling `MERGE_SORT`$(A, 1, A.length)$



$\lceil x \rceil$ denotes the least integer greater than or equal to x
$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Part 2: Merge Sort

We can now use the MERGE procedure as a subroutine in the merge sort algorithm MERGE_SORT(A, p, r).

2 Cases:

$p \geq r$: Then $A[p..r]$ has 0 or 1 elements and is thus sorted

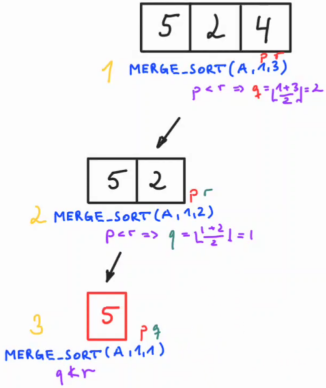
$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.

In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:

$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

```
MERGE_SORT(A, p, r)
    IF(p < r) THEN
        q = ⌊(p + r)/2⌋
        MERGE_SORT(A, p, q)
        MERGE_SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Start algorithm by calling MERGE_SORT$(A, 1, A.length)$



$\lceil x \rceil$ denotes the least integer greater than or equal to x

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Part 2: Merge Sort

We can now use the `MERGE` procedure as a subroutine in the merge sort algorithm `MERGE_SORT`(A, p, r).

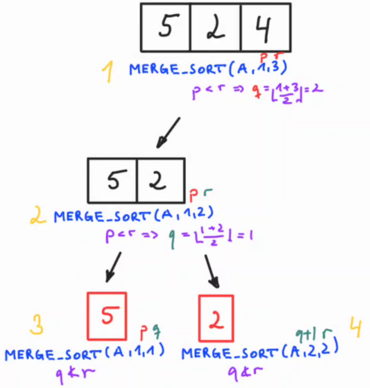
2 Cases:

$p \geq r$**:** Then $A[p..r]$ has 0 or 1 elements and is thus sorted

$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.
In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:
$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

```
MERGE_SORT(A, p, r)
    IF(p < r) THEN
        q = ⌊(p + r)/2⌋
        MERGE_SORT(A, p, q)
        MERGE_SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Start algorithm by calling `MERGE_SORT`$(A, 1, A.length)$



$\lceil x \rceil$ denotes the least integer greater than or equal to x

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

We can now use the `MERGE` procedure as a subroutine in the merge sort algorithm `MERGE_SORT`(A, p, r).

2 Cases:

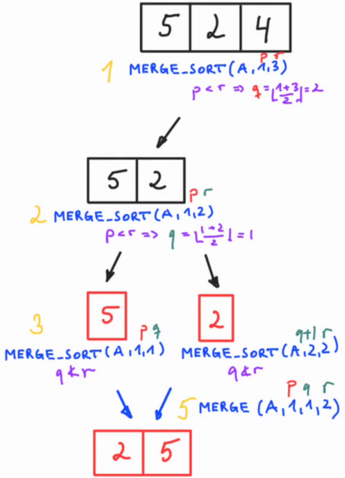
$p \geq r$**:** Then $A[p..r]$ has 0 or 1 elements and is thus sorted

$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.
In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:
$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

```
MERGE_SORT(A, p, r)
    IF(p < r) THEN
        q = ⌊(p + r)/2⌋
        MERGE_SORT(A, p, q)
        MERGE_SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Start algorithm by calling `MERGE_SORT`$(A, 1, A.length)$

$\lceil x \rceil$ denotes the least integer greater than or equal to x
$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.



Part 2: Merge Sort

We can now use the MERGE procedure as a subroutine in the merge sort algorithm MERGE_SORT(A, p, r).

2 Cases:

$p \geq r$**:** Then $A[p..r]$ has 0 or 1 elements and is thus sorted

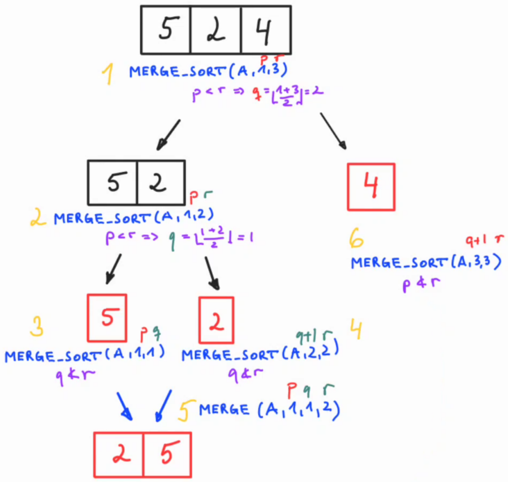
$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.

In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:

$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

```
MERGE_SORT(A, p, r)
    IF(p < r) THEN
        q = ⌊(p + r)/2⌋
        MERGE_SORT(A, p, q)
        MERGE_SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Start algorithm by calling MERGE_SORT(A, 1, $A.length$)



$\lceil x \rceil$ denotes the least integer greater than or equal to x

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Part 2: Merge Sort

We can now use the MERGE procedure as a subroutine in the merge sort algorithm MERGE_SORT(A, p, r).

2 Cases:

$p \geq r$: Then $A[p..r]$ has 0 or 1 elements and is thus sorted

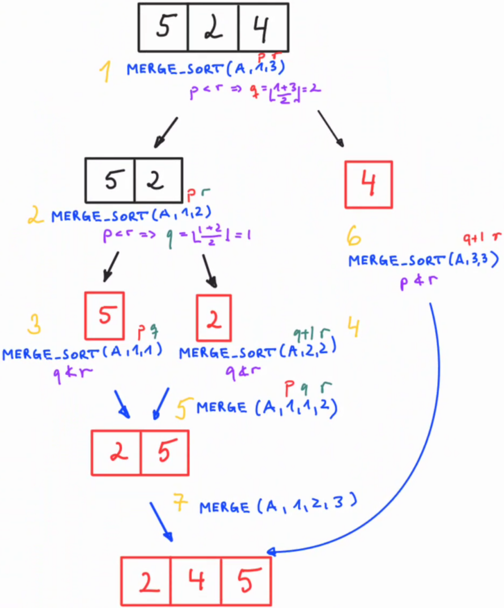
$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.

In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:

$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

```
MERGE_SORT(A, p, r)
    IF(p < r) THEN
        q = ⌊(p + r)/2⌋
        MERGE_SORT(A, p, q)
        MERGE_SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

Start algorithm by calling MERGE_SORT(A, 1, $A.length$)



$\lceil x \rceil$ denotes the least integer greater than or equal to x

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

Part 2: Merge Sort

By the previous arguments, we obtain

Theorem

MERGE_SORT$(A, 1, A.length)$ *correctly sorts the array* A

```
MERGE_SORT(A, p, r)
  IF(p < r) THEN
      q = ⌊(p + r)/2⌋
      MERGE_SORT(A, p, q)
      MERGE_SORT(A, q + 1, r)
      MERGE(A, p, q, r)
```

We start this algorithm by calling MERGE_SORT$(A, 1, A.length)$

Part 2: Merge Sort

By the previous arguments, we obtain

Theorem

`MERGE_SORT`$(A, 1, A.length)$ correctly sorts the array A in $\Theta(n \log_2(n))$ time.

Any idea for proof of runtime ?

```
MERGE_SORT(A, p, r)
  IF(p < r) THEN
      q = ⌊(p + r)/2⌋
      MERGE_SORT(A, p, q)
      MERGE_SORT(A, q + 1, r)
      MERGE(A, p, q, r)
```

We start this algorithm by calling `MERGE_SORT`$(A, 1, A.length)$

Part 2: Merge Sort

By the previous arguments, we obtain

Theorem

`MERGE_SORT`$(A, 1, A.length)$ *correctly sorts the array A in $\Theta(n \log_2(n))$ time.*

Any idea for proof of runtime ?

```
MERGE_SORT(A, p, r)
IF(p < r) THEN
    q = ⌊(p + r)/2⌋
    MERGE_SORT(A, p, q)
    MERGE_SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

We can achieve runtime by using the Master theorem:
If $T(n) = aT(n/b) + \Theta(n^d)$ with constants $a \geq 1$ and $b > 1$, then

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

We start this algorithm by calling `MERGE_SORT`$(A, 1, A.length)$

Part 2: Merge Sort

By the previous arguments, we obtain

Theorem

MERGE_SORT$(A, 1, A.length)$ correctly sorts the array A in $\Theta(n \log_2(n))$ time.

Any idea for proof of runtime ?

$$T(n) = 2T(n/2) + \Theta(n^1), \text{ i.e., } a = b = 2,\ d = 1 \xrightarrow{2=2^1} \Theta(n \log_2 n)$$

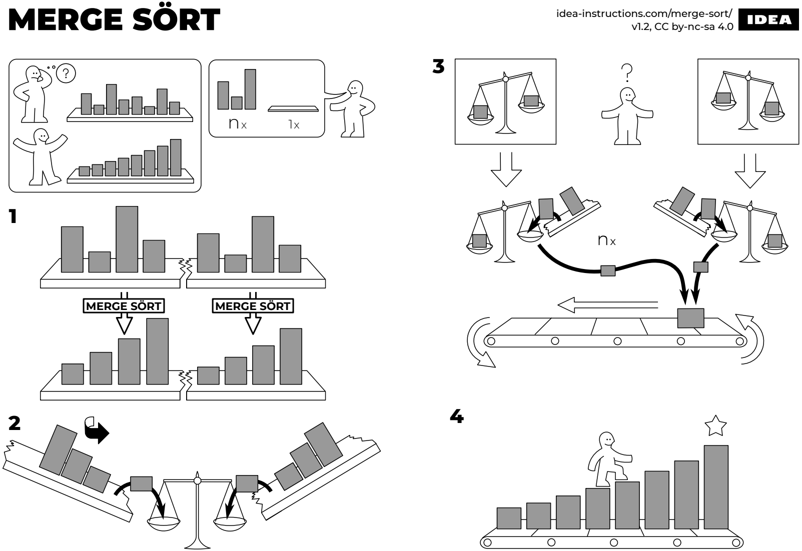
```
MERGE_SORT(A, p, r)
    IF(p < r) THEN
        q = ⌊(p + r)/2⌋
        MERGE_SORT(A, p, q)
        MERGE_SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

We can achieve runtime by using the Master theorem:
If $T(n) = aT(n/b) + \Theta(n^d)$ with constants $a \geq 1$ and $b > 1$, then

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

We start this algorithm by calling MERGE_SORT$(A, 1, A.length)$

Part 2: Merge Sort - Comic



Part 2: Heapsort

We introduce another sorting algorithm: `Heapsort`.

Like merge sort, but unlike insertion sort, heapsort's running time is in $O(n \log n)$.

Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time.

Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Before starting we need some further basic introduction into some new terms: Graphs, Tree & Co [next slides]

Part 2: Heapsort

We introduce another sorting algorithm: `Heapsort`.

Like merge sort, but unlike insertion sort, heapsort's running time is in $O(n \log n)$.

Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time.

Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Before starting we need some further basic introduction into some new terms: Graphs, Tree & Co [next slides]

Part 2: Heapsort

We introduce another sorting algorithm: `Heapsort`.

Like merge sort, but unlike insertion sort, heapsort's running time is in $O(n \log n)$.

Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time.

Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Before starting we need some further basic introduction into some new terms: Graphs, Tree & Co [next slides]

Part 2: Heapsort

We introduce another sorting algorithm: `Heapsort`.

Like merge sort, but unlike insertion sort, heapsort's running time is in $O(n \log n)$.

Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time.

Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Before starting we need some further basic introduction into some new terms: Graphs, Tree & Co [next slides]

Part 2: Heapsort

We introduce another sorting algorithm: `Heapsort`.

Like merge sort, but unlike insertion sort, heapsort's running time is in $O(n \log n)$.

Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time.

Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Before starting we need some further basic introduction into some new terms: Graphs, Tree & Co [next slides]

Part 2: Heapsort - some graph theory first

To recall:

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.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree (per def: G is connected and acyclic).*
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 $|E| = |V| - 1$.*

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of $T = h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has at most two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$.* (next slides)

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.* (next slides)

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.* (next slides)

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of $T = h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has at most two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$. (next slides)*

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$. (next slides)*

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$. (next slides)*

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of $T = h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has <u>at most</u> two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$.* (*next slides*)

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.* (*next slides*)

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.* (*next slides*)

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of $T = h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has at most two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$.* (next slides)

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.* (next slides)

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.* (next slides)

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of $T = h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has <u>at most</u> two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$.* (next slides)

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.* (next slides)

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.* (next slides)

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of T = $h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has <u>at most</u> two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$.* (next slides)

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.* (next slides)

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.* (next slides)

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of $T = h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has <u>at most</u> two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$.* (next slides)

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.* (next slides)

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.* (next slides)

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of $T = h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has at most two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$.* (next slides)

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.* (next slides)

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.* (next slides)

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of T = $h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has at most two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$.* (next slides)

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.* (next slides)

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.* (next slides)

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of T = $h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has <u>at most</u> two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$.* (next slides)

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.* (next slides)

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.* (next slides)

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of T = $h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has at most two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$.* (next slides)

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.* (next slides)

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.* (next slides)

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of T = $h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has at most two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$. (next slides)*

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$. (next slides)*

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$. (next slides)*

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of T = $h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has at most two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$. (next slides)*

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$. (next slides)*

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$. (next slides)*

Part 2: Heapsort - some graph theory first

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T [for all we give examples - board!!!]

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.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

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 child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of T = $h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has at most two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L - 1$. (next slides)*

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$. (next slides)*

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$. (next slides)*

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n=1 \implies T=(\{v\},\emptyset)$ = “single_vertex_graph” and thus $L=1$ and $B=0 \implies B=L-1=0$ is correct.

Assume the statement is true for all trees on $n\geq 1$ vertices.

Let T be a tree with $|V(T)|=n+1$ vertices and L leaves. [It holds that $L\geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T-x$ from which x and the unique edge $\{w,x\}$ containing x has been removed.

Denote with L' the number of leaves in $T-x$ and with B' the number of vertices in $T-x$ having 2 children.

Since $T-x$ has n vertices we can apply the Ind-Hyp. on $T-x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T-x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L'=L-1$.
By Ind-Hyp: $B'=L'-1=L-2$.
In T we have now exactly one vertex more than in $T-x$ that has two children, namely w.
Thus, $B=B'+1=L-1$

Case (b): In $T-x$, vertex w is now a leaf. Hence, $L'=L$. Moreover, observe that $B=B'$
By Ind-Hyp: $B'=L'-1=L-1$. Since $B=B'$, we have thus $B=L-1$.

$\square$

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.
Base case: $n=1 \implies T=(\{v\},\emptyset)$ = "single_vertex_graph" and thus $L=1$ and $B=0 \implies B=L-1=0$ is correct.
Assume the statement is true for all trees on $n \geq 1$ vertices.
Let T be a tree with $|V(T)|=n+1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]
Let x be a leaf and consider the tree $T-x$ from which x and the unique edge $\{w,x\}$ containing x has been removed.
Denote with L' the number of leaves in $T-x$ and with B' the number of vertices in $T-x$ having 2 children.
Since $T-x$ has n vertices we can apply the Ind-Hyp. on $T-x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.
Case (a): In $T-x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L'=L-1$.
By Ind-Hyp: $B'=L'-1=L-2$.
In T we have now exactly one vertex more than in $T-x$ that has two children, namely w.
Thus, $B=B'+1=L-1$

Case (b): In $T-x$, vertex w is now a leaf. Hence, $L'=L$. Moreover, observe that $B=B'$
By Ind-Hyp: $B'=L'-1=L-1$. Since $B=B'$, we have thus $B=L-1$.

□

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n = 1 \implies T = (\{v\}, \emptyset)$ = "single_vertex_graph" and thus $L = 1$ and $B = 0 \implies B = L - 1 = 0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)| = n + 1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T - x$ from which x and the unique edge $\{w, x\}$ containing x has been removed.

Denote with L' the number of leaves in $T - x$ and with B' the number of vertices in $T - x$ having 2 children.

Since $T - x$ has n vertices we can apply the Ind-Hyp. on $T - x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T - x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L' = L - 1$.
By Ind-Hyp: $B' = L' - 1 = L - 2$.
In T we have now exactly one vertex more than in $T - x$ that has two children, namely w.
Thus, $B = B' + 1 = L - 1$

Case (b): In $T - x$, vertex w is now a leaf. Hence, $L' = L$. Moreover, observe that $B = B'$
By Ind-Hyp: $B' = L' - 1 = L - 1$. Since $B = B'$, we have thus $B = L - 1$.

□

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n = 1 \implies T = (\{v\}, \emptyset)$ = "single_vertex_graph" and thus $L = 1$ and $B = 0 \implies B = L - 1 = 0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)| = n + 1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T - x$ from which x and the unique edge $\{w, x\}$ containing x has been removed.

Denote with L' the number of leaves in $T - x$ and with B' the number of vertices in $T - x$ having 2 children.

Since $T - x$ has n vertices we can apply the Ind-Hyp. on $T - x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T - x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L' = L - 1$.
By Ind-Hyp: $B' = L' - 1 = L - 2$.
In T we have now exactly one vertex more than in $T - x$ that has two children, namely w.
Thus, $B = B' + 1 = L - 1$

Case (b): In $T - x$, vertex w is now a leaf. Hence, $L' = L$. Moreover, observe that $B = B'$
By Ind-Hyp: $B' = L' - 1 = L - 1$. Since $B = B'$, we have thus $B = L - 1$.

$\square$

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n = 1 \implies T = (\{v\}, \emptyset)$ = "single_vertex_graph" and thus $L = 1$ and $B = 0 \implies B = L - 1 = 0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)| = n+1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T - x$ from which x and the unique edge $\{w, x\}$ containing x has been removed.

Denote with L' the number of leaves in $T - x$ and with B' the number of vertices in $T - x$ having 2 children.

Since $T - x$ has n vertices we can apply the Ind-Hyp. on $T - x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T - x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L' = L - 1$.
By Ind-Hyp: $B' = L' - 1 = L - 2$.
In T we have now exactly one vertex more than in $T - x$ that has two children, namely w.
Thus, $B = B' + 1 = L - 1$

Case (b): In $T - x$, vertex w is now a leaf. Hence, $L' = L$. Moreover, observe that $B = B'$
By Ind-Hyp: $B' = L' - 1 = L - 1$. Since $B = B'$, we have thus $B = L - 1$.

$\square$

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n=1 \implies T=(\{v\},\emptyset)$ = "single_vertex_graph" and thus $L=1$ and $B=0 \implies B=L-1=0$ is correct.

Assume the statement is true for all trees on $n\geq 1$ vertices.

Let T be a tree with $|V(T)|=n+1$ vertices and L leaves. [It holds that $L\geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T-x$ from which x and the unique edge $\{w,x\}$ containing x has been removed.

Denote with L' the number of leaves in $T-x$ and with B' the number of vertices in $T-x$ having 2 children.

Since $T-x$ has n vertices we can apply the Ind-Hyp. on $T-x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T-x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L'=L-1$.

By Ind-Hyp: $B'=L'-1=L-2$.

In T we have now exactly one vertex more than in $T-x$ that has two children, namely w.

Thus, $B=B'+1=L-1$

Case (b): In $T-x$, vertex w is now a leaf. Hence, $L'=L$. Moreover, observe that $B=B'$

By Ind-Hyp: $B'=L'-1=L-1$. Since $B=B'$, we have thus $B=L-1$.

□

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.

By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n = 1 \implies T = (\{v\}, \emptyset)$ = "single_vertex_graph" and thus $L = 1$ and $B = 0 \implies B = L - 1 = 0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)| = n + 1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T - x$ from which x and the unique edge $\{w, x\}$ containing x has been removed.

Denote with L' the number of leaves in $T - x$ and with B' the number of vertices in $T - x$ having 2 children.

Since $T - x$ has n vertices we can apply the Ind-Hyp. on $T - x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T - x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L' = L - 1$.
By Ind-Hyp: $B' = L' - 1 = L - 2$.
In T we have now exactly one vertex more than in $T - x$ that has two children, namely w.
Thus, $B = B' + 1 = L - 1$

Case (b): In $T - x$, vertex w is now a leaf. Hence, $L' = L$. Moreover, observe that $B = B'$
By Ind-Hyp: $B' = L' - 1 = L - 1$. Since $B = B'$, we have thus $B = L - 1$.

$\square$

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n=1 \implies T=(\{v\},\emptyset)$ = “single_vertex_graph” and thus $L=1$ and $B=0 \implies B=L-1=0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)|=n+1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T-x$ from which x and the unique edge $\{w,x\}$ containing x has been removed.

Denote with L' the number of leaves in $T-x$ and with B' the number of vertices in $T-x$ having 2 children.

Since $T-x$ has n vertices we can apply the Ind-Hyp. on $T-x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T-x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L'=L-1$.
By Ind-Hyp: $B'=L'-1=L-2$.
In T we have now exactly one vertex more than in $T-x$ that has two children, namely w.
Thus, $B=B'+1=L-1$

Case (b): In $T-x$, vertex w is now a leaf. Hence, $L'=L$. Moreover, observe that $B=B'$
By Ind-Hyp: $B'=L'-1=L-1$. Since $B=B'$, we have thus $B=L-1$.

$\square$

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n = 1 \implies T = (\{v\}, \emptyset)$ = "single_vertex_graph" and thus $L = 1$ and $B = 0 \implies B = L - 1 = 0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)| = n + 1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T - x$ from which x and the unique edge $\{w, x\}$ containing x has been removed.

Denote with L' the number of leaves in $T - x$ and with B' the number of vertices in $T - x$ having 2 children.

Since $T - x$ has n vertices we can apply the Ind-Hyp. on $T - x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T - x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L' = L - 1$.
By Ind-Hyp: $B' = L' - 1 = L - 2$.
In T we have now exactly one vertex more than in $T - x$ that has two children, namely w.
Thus, $B = B' + 1 = L - 1$

Case (b): In $T - x$, vertex w is now a leaf. Hence, $L' = L$. Moreover, observe that $B = B'$
By Ind-Hyp: $B' = L' - 1 = L - 1$. Since $B = B'$, we have thus $B = L - 1$.

$\square$

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n=1 \implies T=(\{v\},\emptyset)$ = "single_vertex_graph" and thus $L=1$ and $B=0 \implies B=L-1=0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)|=n+1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T-x$ from which x and the unique edge $\{w,x\}$ containing x has been removed.

Denote with L' the number of leaves in $T-x$ and with B' the number of vertices in $T-x$ having 2 children.

Since $T-x$ has n vertices we can apply the Ind-Hyp. on $T-x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T-x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L'=L-1$.
By Ind-Hyp: $B'=L'-1=L-2$.
In T we have now exactly one vertex more than in $T-x$ that has two children, namely w.
Thus, $B=B'+1=L-1$

Case (b): In $T-x$, vertex w is now a leaf. Hence, $L'=L$. Moreover, observe that $B=B'$
By Ind-Hyp: $B'=L'-1=L-1$. Since $B=B'$, we have thus $B=L-1$.

□

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n=1 \implies T=(\{v\},\emptyset)$ = "single_vertex_graph" and thus $L=1$ and $B=0 \implies B=L-1=0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)|=n+1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T-x$ from which x and the unique edge $\{w,x\}$ containing x has been removed.

Denote with L' the number of leaves in $T-x$ and with B' the number of vertices in $T-x$ having 2 children.

Since $T-x$ has n vertices we can apply the Ind-Hyp. on $T-x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T-x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L'=L-1$.

By Ind-Hyp: $B'=L'-1=L-2$.

In T we have now exactly one vertex more than in $T-x$ that has two children, namely w.

Thus, $B=B'+1=L-1$

Case (b): In $T-x$, vertex w is now a leaf. Hence, $L'=L$. Moreover, observe that $B=B'$

By Ind-Hyp: $B'=L'-1=L-1$. Since $B=B'$, we have thus $B=L-1$.

□

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n=1 \implies T=(\{v\},\emptyset)$ = "single_vertex_graph" and thus $L=1$ and $B=0 \implies B=L-1=0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)|=n+1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T-x$ from which x and the unique edge $\{w,x\}$ containing x has been removed.

Denote with L' the number of leaves in $T-x$ and with B' the number of vertices in $T-x$ having 2 children.

Since $T-x$ has n vertices we can apply the Ind-Hyp. on $T-x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T-x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L'=L-1$.

By Ind-Hyp: $B'=L'-1=L-2$.

In T we have now exactly one vertex more than in $T-x$ that has two children, namely w.

Thus, $B=B'+1=L-1$

Case (b): In $T-x$, vertex w is now a leaf. Hence, $L'=L$. Moreover, observe that $B=B'$

By Ind-Hyp: $B'=L'-1=L-1$. Since $B=B'$, we have thus $B=L-1$.

□

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n = 1 \implies T = (\{v\}, \emptyset)$ = "single_vertex_graph" and thus $L = 1$ and $B = 0 \implies B = L - 1 = 0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)| = n + 1$ vertices and L leaves. [It holds that $L \geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T - x$ from which x and the unique edge $\{w, x\}$ containing x has been removed.

Denote with L' the number of leaves in $T - x$ and with B' the number of vertices in $T - x$ having 2 children.

Since $T - x$ has n vertices we can apply the Ind-Hyp. on $T - x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T - x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L' = L - 1$.
By Ind-Hyp: $B' = L' - 1 = L - 2$.
In T we have now exactly one vertex more than in $T - x$ that has two children, namely w.
Thus, $B = B' + 1 = L - 1$

Case (b): In $T - x$, vertex w is now a leaf. Hence, $L' = L$. Moreover, observe that $B = B'$
By Ind-Hyp: $B' = L' - 1 = L - 1$. Since $B = B'$, we have thus $B = L - 1$.

□

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n=1 \implies T=(\{v\},\emptyset)$ = "single_vertex_graph" and thus $L=1$ and $B=0 \implies B=L-1=0$ is correct.

Assume the statement is true for all trees on $n\geq 1$ vertices.

Let T be a tree with $|V(T)|=n+1$ vertices and L leaves. [It holds that $L\geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T-x$ from which x and the unique edge $\{w,x\}$ containing x has been removed.

Denote with L' the number of leaves in $T-x$ and with B' the number of vertices in $T-x$ having 2 children.

Since $T-x$ has n vertices we can apply the Ind-Hyp. on $T-x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T-x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L'=L-1$.

By Ind-Hyp: $B'=L'-1=L-2$.

In T we have now exactly one vertex more than in $T-x$ that has two children, namely w.

Thus, $B=B'+1=L-1$

Case (b): In $T-x$, vertex w is now a leaf. Hence, $L'=L$. Moreover, observe that $B=B'$

By Ind-Hyp: $B'=L'-1=L-1$. Since $B=B'$, we have thus $B=L-1$.

$\square$

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.

By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n=1 \implies T=(\{v\},\emptyset)$ = "single_vertex_graph" and thus $L=1$ and $B=0 \implies B=L-1=0$ is correct.

Assume the statement is true for all trees on $n\geq 1$ vertices.

Let T be a tree with $|V(T)|=n+1$ vertices and L leaves. [It holds that $L\geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T-x$ from which x and the unique edge $\{w,x\}$ containing x has been removed.

Denote with L' the number of leaves in $T-x$ and with B' the number of vertices in $T-x$ having 2 children.

Since $T-x$ has n vertices we can apply the Ind-Hyp. on $T-x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T-x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L'=L-1$.

By Ind-Hyp: $B'=L'-1=L-2$.

In T we have now exactly one vertex more than in $T-x$ that has two children, namely w.

Thus, $B=B'+1=L-1$

Case (b): In $T-x$, vertex w is now a leaf. Hence, $L'=L$. Moreover, observe that $B=B'$

By Ind-Hyp: $B'=L'-1=L-1$. Since $B=B'$, we have thus $B=L-1$.

□

Part 2: Heapsort - some graph theory first

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n=1 \implies T=(\{v\},\emptyset)$ = "single_vertex_graph" and thus $L=1$ and $B=0 \implies B=L-1=0$ is correct.

Assume the statement is true for all trees on $n\geq 1$ vertices.

Let T be a tree with $|V(T)|=n+1$ vertices and L leaves. [It holds that $L\geq 1$ (exercise!)]

Let x be a leaf and consider the tree $T-x$ from which x and the unique edge $\{w,x\}$ containing x has been removed.

Denote with L' the number of leaves in $T-x$ and with B' the number of vertices in $T-x$ having 2 children.

Since $T-x$ has n vertices we can apply the Ind-Hyp. on $T-x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T-x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L'=L-1$.

By Ind-Hyp: $B'=L'-1=L-2$.

In T we have now exactly one vertex more than in $T-x$ that has two children, namely w.

Thus, $B=B'+1=L-1$

Case (b): In $T-x$, vertex w is now a leaf. Hence, $L'=L$. Moreover, observe that $B=B'$

By Ind-Hyp: $B'=L'-1=L-1$. Since $B=B'$, we have thus $B=L-1$.

□

Part 2: Heapsort - some graph theory first

Lemma. *A complete binary tree* $T = (V, E)$ *has height* $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.

Proof. Let L be the number of leaves in T, $h := h(T)$ its height and $n = |V|$.

"Easy to verify by induction:"

$$L = 2^h.$$

$L = 2^h \iff h = \log_2(L)$.

By the previous lemma, we have $B = L - 1$ with B being the number of vertices with 2 children.

Since T is a complete binary tree its vertex set can be partitioned into leaves and vertices with two children, i.e.,

$$n = B + L = L - 1 + L = 2L - 1 \iff L = \frac{n+1}{2}$$

$\implies h = \log_2(\frac{n+1}{2}) = \log_2(n+1) - \log_2(2) = \log_2(n+1) - 1$ □

Part 2: Heapsort - some graph theory first

Lemma. *A complete binary tree* $T = (V, E)$ *has height* $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.

Proof. Let L be the number of leaves in T, $h := h(T)$ its height and $n = |V|$.

"Easy to verify by induction:"

$$L = 2^h.$$

$L = 2^h \iff h = \log_2(L)$.

By the previous lemma, we have $B = L - 1$ with B being the number of vertices with 2 children.

Since T is a complete binary tree its vertex set can be partitioned into leaves and vertices with two children, i.e.,

$$n = B + L = L - 1 + L = 2L - 1 \iff L = \frac{n+1}{2}$$

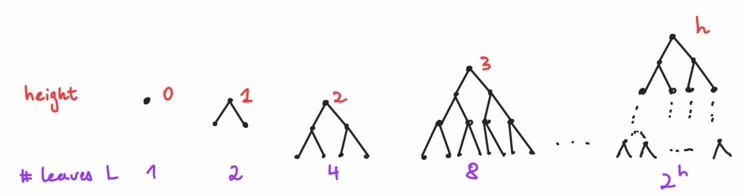
$\implies h = \log_2(\frac{n+1}{2}) = \log_2(n+1) - \log_2(2) = \log_2(n+1) - 1$ $\square$

Part 2: Heapsort - some graph theory first

Lemma. *A complete binary tree* $T = (V, E)$ *has height* $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.

Proof. Let L be the number of leaves in T, $h := h(T)$ its height and $n = |V|$.

"Easy to verify by induction:"



$$L = 2^h.$$

$L = 2^h \iff h = \log_2(L)$.

By the previous lemma, we have $B = L - 1$ with B being the number of vertices with 2 children.

Since T is a complete binary tree its vertex set can be partitioned into leaves and vertices with two children, i.e.,

$$n = B + L = L - 1 + L = 2L - 1 \iff L = \frac{n+1}{2}$$

$\implies h = \log_2(\frac{n+1}{2}) = \log_2(n+1) - \log_2(2) = \log_2(n+1) - 1$

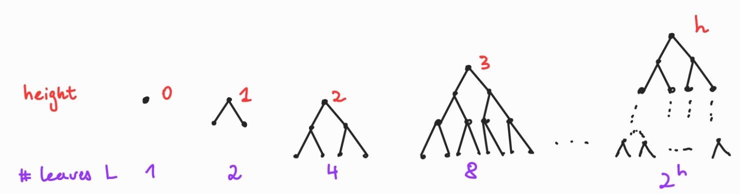
$\square$

Part 2: Heapsort - some graph theory first

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.*

Proof. Let L be the number of leaves in T, $h := h(T)$ its height and $n = |V|$.

"Easy to verify by induction:"



$$L = 2^h.$$

$L = 2^h \iff h = \log_2(L)$.

By the previous lemma, we have $B = L - 1$ with B being the number of vertices with 2 children.

Since T is a complete binary tree its vertex set can be partitioned into leaves and vertices with two children, i.e.,

$$n = B + L = L - 1 + L = 2L - 1 \iff L = \frac{n+1}{2}$$

$$\implies h = \log_2(\tfrac{n+1}{2}) = \log_2(n+1) - \log_2(2) = \log_2(n+1) - 1$$

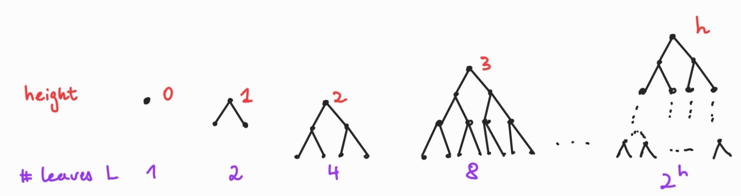
$\square$

Part 2: Heapsort - some graph theory first

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.*

Proof. Let L be the number of leaves in T, $h := h(T)$ its height and $n = |V|$.

"Easy to verify by induction:"



$$L = 2^h.$$

$L = 2^h \iff h = \log_2(L)$.

By the previous lemma, we have $B = L - 1$ with B being the number of vertices with 2 children.

Since T is a complete binary tree its vertex set can be partitioned into leaves and vertices with two children, i.e.,

$$n = B + L = L - 1 + L = 2L - 1 \iff L = \frac{n+1}{2}$$

$$\implies h = \log_2(\tfrac{n+1}{2}) = \log_2(n+1) - \log_2(2) = \log_2(n+1) - 1$$

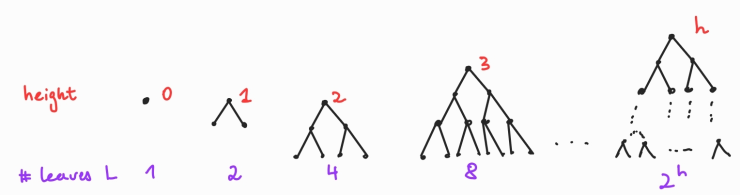
$\square$

Part 2: Heapsort - some graph theory first

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.*

Proof. Let L be the number of leaves in T, $h := h(T)$ its height and $n = |V|$.

"Easy to verify by induction:"



$$L = 2^h.$$

$L = 2^h \iff h = \log_2(L)$.

By the previous lemma, we have $B = L - 1$ with B being the number of vertices with 2 children.

Since T is a complete binary tree its vertex set can be partitioned into leaves and vertices with two children, i.e.,

$$n = B + L = L - 1 + L = 2L - 1 \iff L = \frac{n+1}{2}$$

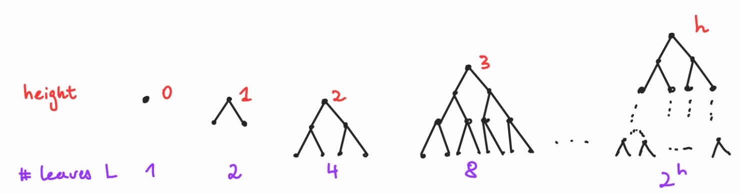
$\implies h = \log_2(\frac{n+1}{2}) = \log_2(n+1) - \log_2(2) = \log_2(n+1) - 1$

Part 2: Heapsort - some graph theory first

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.*

Proof. Let L be the number of leaves in T, $h := h(T)$ its height and $n = |V|$.

"Easy to verify by induction:"



$$L = 2^h.$$

$L = 2^h \iff h = \log_2(L)$.

By the previous lemma, we have $B = L - 1$ with B being the number of vertices with 2 children.

Since T is a complete binary tree its vertex set can be partitioned into leaves and vertices with two children, i.e.,

$$n = B + L = L - 1 + L = 2L - 1 \iff L = \frac{n+1}{2}$$

$$\implies h = \log_2(\tfrac{n+1}{2}) = \log_2(n+1) - \log_2(2) = \log_2(n+1) - 1$$

□

Part 2: Heapsort - some graph theory first

Lemma. *A nearly-complete binary tree* $T = (V, E)$ *has height* $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.

Proof.

$\implies |V| \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

If only one leaf at depth h exists, then $|V| = 1 + \sum_{i=0}^{h-1} 2^i = 1 + (2^h - 1) = 2^h$.

Since at least one leaf at depths h must exist, we obtain

$$2^h \leq |V| \leq 2^{h+1} - 1 < 2^{h+1}$$

The latter is, if and only if,

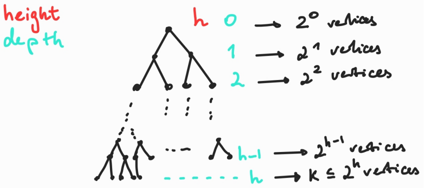
$$h \leq \log_2(|V|) < h + 1$$

Since h is an integer, $h = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$. □

Part 2: Heapsort - some graph theory first

Lemma. *A nearly-complete binary tree* $T = (V, E)$ *has height* $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.

Proof.



$\Longrightarrow \ |V| \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

If only one leaf at depth h exists, then $|V| = 1 + \sum_{i=0}^{h-1} 2^i = 1 + (2^h - 1) = 2^h$.

Since at least one leaf at depths h must exist, we obtain

$$2^h \leq |V| \leq 2^{h+1} - 1 < 2^{h+1}$$

The latter is, if and only if,

$$h \leq \log_2(|V|) < h + 1$$

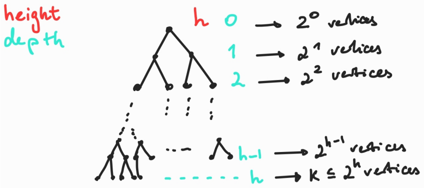
Since h is an integer, $h = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.

□

Part 2: Heapsort - some graph theory first

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.*

Proof.



$\implies |V| \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

If only one leaf at depth h exists, then $|V| = 1 + \sum_{i=0}^{h-1} 2^i = 1 + (2^h - 1) = 2^h$.

Since at least one leaf at depths h must exist, we obtain

$$2^h \leq |V| \leq 2^{h+1} - 1 < 2^{h+1}$$

The latter is, if and only if,

$$h \leq \log_2(|V|) < h + 1$$

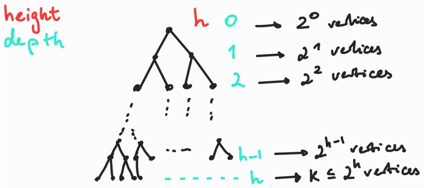
Since h is an integer, $h = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.

□

Part 2: Heapsort - some graph theory first

Lemma. *A nearly-complete binary tree* $T = (V, E)$ *has height* $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.

Proof.



$\Longrightarrow \ |V| \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

If only one leaf at depth h exists, then $|V| = 1 + \sum_{i=0}^{h-1} 2^i = 1 + (2^h - 1) = 2^h$.

Since at least one leaf at depths h must exist, we obtain

$$2^h \leq |V| \leq 2^{h+1} - 1 < 2^{h+1}$$

The latter is, if and only if,

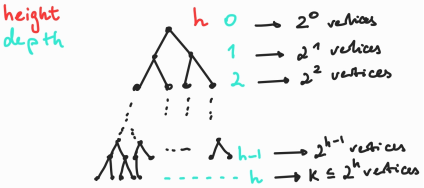
$$h \leq \log_2(|V|) < h + 1$$

Since h is an integer, $h = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.

Part 2: Heapsort - some graph theory first

Lemma. *A nearly-complete binary tree* $T = (V, E)$ *has height* $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.

Proof.



$\implies |V| \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

If only one leaf at depth h exists, then $|V| = 1 + \sum_{i=0}^{h-1} 2^i = 1 + (2^h - 1) = 2^h$.

Since at least one leaf at depths h must exist, we obtain

$$2^h \leq |V| \leq 2^{h+1} - 1 < 2^{h+1}$$

The latter is, if and only if,

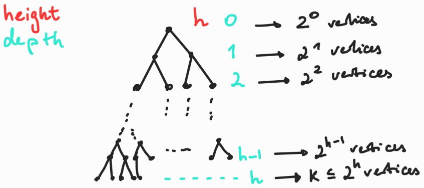
$$h \leq \log_2(|V|) < h + 1$$

Since h is an integer, $h = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$. □

Part 2: Heapsort - some graph theory first

Lemma. *A nearly-complete binary tree* $T = (V, E)$ *has height* $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.

Proof.



$\implies |V| \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

If only one leaf at depth h exists, then $|V| = 1 + \sum_{i=0}^{h-1} 2^i = 1 + (2^h - 1) = 2^h$.

Since at least one leaf at depths h must exist, we obtain

$$2^h \leq |V| \leq 2^{h+1} - 1 < 2^{h+1}$$

The latter is, if and only if,

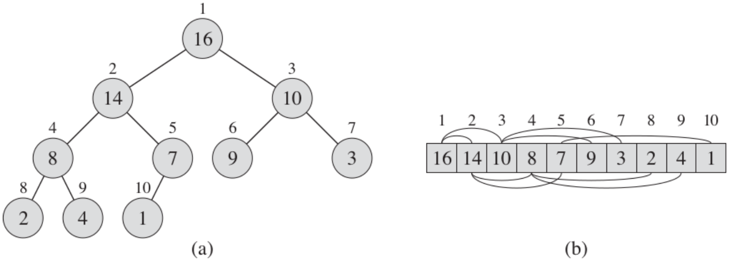
$$h \leq \log_2(|V|) < h + 1$$

Since h is an integer, $h = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.

□

Part 2: Heapsort

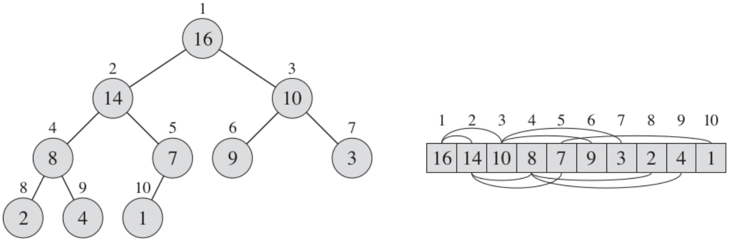
Heapsort uses a data structure called **(binary) heap** which is an array A with a particular structure that can be viewed as a nearly complete binary tree:



Number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

Part 2: Heapsort

Heapsort uses a data structure called **(binary) heap** which is an array A with a particular structure that can be viewed as a nearly complete binary tree:



(a) (b)

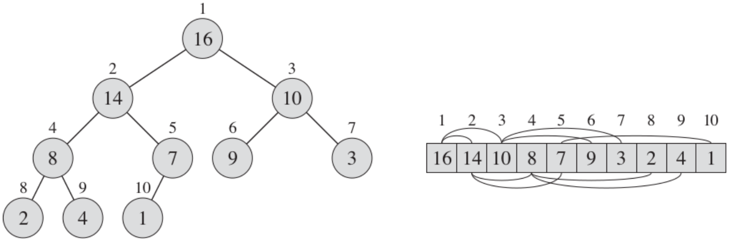
Number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

1st entry of A (i.e., $A[1]$) corresponds to value of root 1,

Parent of i is $\lfloor i/2 \rfloor$ and children of i are $2i$ (left) and $2i + 1$ (right)

Part 2: Heapsort

Heapsort uses a data structure called **(binary) heap** which is an array A with a particular structure that can be viewed as a nearly complete binary tree:



(a) (b)

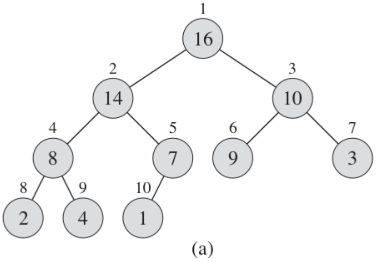
Number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

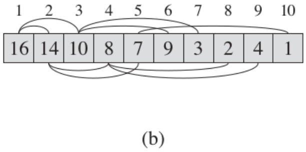
1st entry of A (i.e., $A[1]$) corresponds to value of root 1,

Parent of i is $\lfloor i/2 \rfloor$ and children of i are $2i$ (left) and $2i+1$ (right)

Part 2: Heapsort

Heapsort uses a data structure called **(binary) heap** which is an array A with a particular structure that can be viewed as a nearly complete binary tree:





Number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

1st entry of A (i.e., $A[1]$) corresponds to value of root 1,

Parent of i is $\lfloor i/2 \rfloor$ and children of i are $2i$ (left) and $2i + 1$ (right)

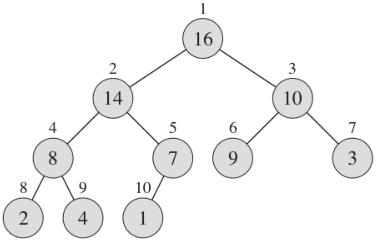
A.length = length of array

A.heap_size = number of elements in heap that are stored in A

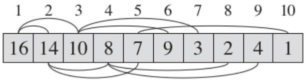
That is, although $A[1..A.length]$ may contain numbers, only the elements in $A[1..A.heap\_size]$, where $0 \leq A.heap\_size \leq A.length$, are valid elements of the heap

Part 2: Heapsort

Heapsort uses a data structure called **(binary) heap** which is an array A with a particular structure that can be viewed as a nearly complete binary tree:



(a)



(b)

Number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

1st entry of A (i.e., $A[1]$) corresponds to value of root 1,

Parent of i is $\lfloor i/2 \rfloor$ and children of i are $2i$ (left) and $2i+1$ (right)

$A.length$ = length of array

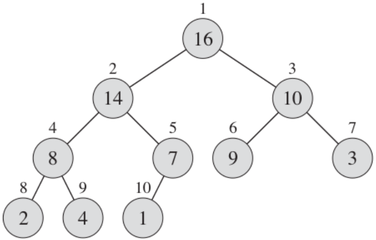
$A.heap\_size$ = number of elements in heap that are stored in A

That is, although $A[1..A.length]$ may contain numbers, only the elements in $A[1..A.heap\_size]$, where $0 \leq A.heap\_size \leq A.length$, are valid elements of the heap

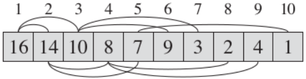
Lemma. *If A is a heap with $n = A.heap\_size$, then the leaves in the tree representation have indices $i \in I = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.* [proof board]

Part 2: Heapsort

Heapsort uses a data structure called **(binary) heap** which is an array A with a particular structure that can be viewed as a nearly complete binary tree:



(a)



(b)

Number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

There are two kinds of binary heaps: max-heaps and min-heaps; specified by a "heap property" that must be satisfied by the values stored at each vertex.

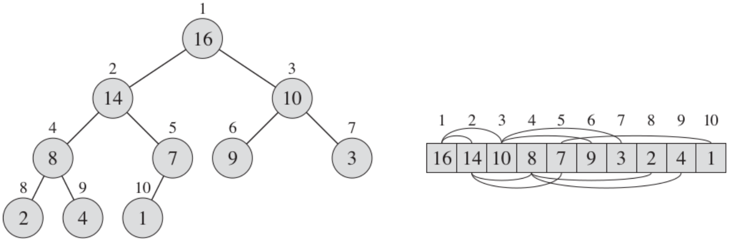
Heap property

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

min-heap: for every node i other than the root it holds that $A[parent(i)] \leq A[i]$

Part 2: Heapsort

Heapsort uses a data structure called **(binary) heap** which is an array A with a particular structure that can be viewed as a nearly complete binary tree:



(a) (b)

Number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

There are two kinds of binary heaps: max-heaps and min-heaps; specified by a "heap property" that must be satisfied by the values stored at each vertex.

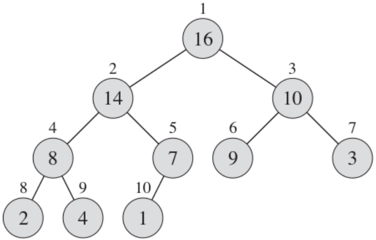
Heap property

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

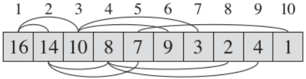
min-heap: for every node i other than the root it holds that $A[parent(i)] \leq A[i]$

Part 2: Heapsort

Heapsort uses a data structure called **(binary) heap** which is an array A with a particular structure that can be viewed as a nearly complete binary tree:



(a)



(b)

Number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

There are two kinds of binary heaps: max-heaps and min-heaps; specified by a "heap property" that must be satisfied by the values stored at each vertex.

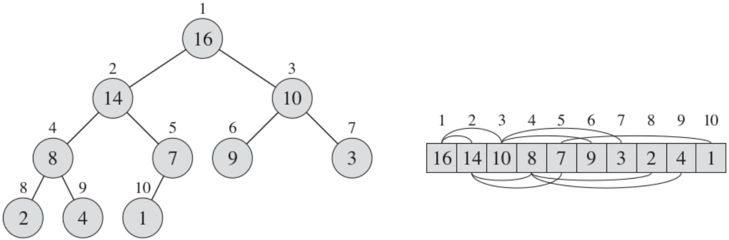
Heap property

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

min-heap: for every node i other than the root it holds that $A[parent(i)] \leq A[i]$

Part 2: Heapsort

Heapsort uses a data structure called **(binary) heap** which is an array A with a particular structure that can be viewed as a nearly complete binary tree:



(a) (b)

Number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

There are two kinds of binary heaps: max-heaps and min-heaps; specified by a "heap property" that must be satisfied by the values stored at each vertex.

Heap property

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

min-heap: for every node i other than the root it holds that $A[parent(i)] \leq A[i]$

For heapsort we use max-heaps (see Example in figure).

Part 2: Heapsort - procedure MAX-HEAPIFY

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i + 1]$$

$\implies$ i violates the max-heap property.

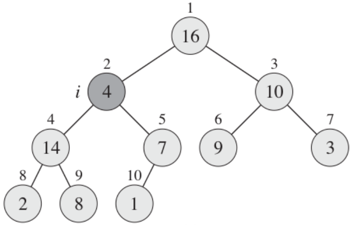
Part 2: Heapsort - procedure MAX-HEAPIFY

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i+1]$$

$\implies$ i violates the max-heap property.



`Max-Heapify` lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

This is achieved by exchanging $A[i]$ recursively with the largest element of the children

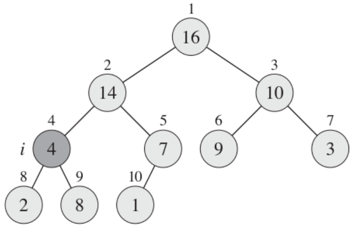
Part 2: Heapsort - procedure MAX-HEAPIFY

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i + 1]$$

$\Longrightarrow$ i violates the max-heap property.



`Max-Heapify` lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

This is achieved by exchanging $A[i]$ recursively with the largest element of the children

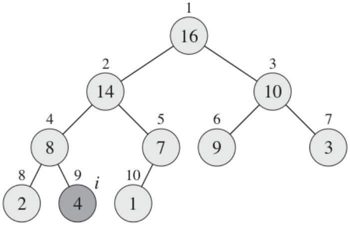
Part 2: Heapsort - procedure MAX-HEAPIFY

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i+1]$$

$\implies$ i violates the max-heap property.



`Max-Heapify` lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

This is achieved by exchanging $A[i]$ recursively with the largest element of the children

Part 2: Heapsort - procedure Max-Heapify

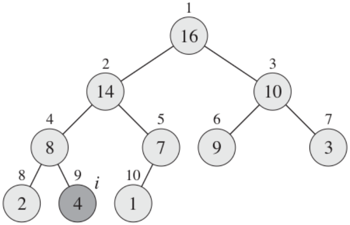
max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i + 1]$$

$\implies$ i violates the max-heap property.

```
Max-Heapify(A, i)
l := 2i and r := 2i + 1
largest := i
IF (l ≤ A.heap_size and A[l] > A[largest])
THEN largest := l
IF (r ≤ A.heap_size and A[r] > A[largest])
THEN largest := r
IF (largest ≠ i) THEN
    exchange A[i] with A[largest]
    Max-Heapify(A, largest)
```



Max-Heapify lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

This is achieved by exchanging $A[i]$ recursively with the largest element of the children

Part 2: Heapsort - procedure MAX-HEAPIFY

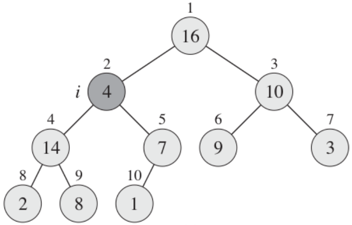
max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i+1]$$

$\implies$ i violates the max-heap property.

```
Max-Heapify(A, i)
  l := 2i and r := 2i + 1
  largest := i
  IF (l ≤ A.heap_size and A[l] > A[largest])
  THEN largest := l
  IF (r ≤ A.heap_size and A[r] > A[largest])
  THEN largest := r
  IF (largest ≠ i) THEN
      exchange A[i] with A[largest]
      Max-Heapify(A, largest)
```



Max-Heapify lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

This is achieved by exchanging $A[i]$ recursively with the largest element of the children

Part 2: Heapsort - procedure MAX-HEAPIFY

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

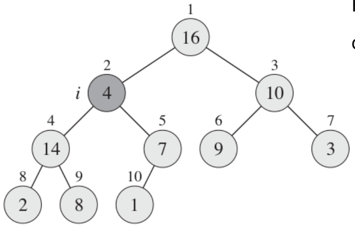
Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i+1]$$

$\Longrightarrow$ i violates the max-heap property.

`Max-Heapify`(A, i)

1 $l := 2i$ and $r := 2i + 1$
2 $largest := i$
3 `IF` $(l \leq A.heap\_size$ and $A[l] > A[largest])$
`THEN` $largest := l$
4 `IF` $(r \leq A.heap\_size$ and $A[r] > A[largest])$
`THEN` $largest := r$
5 `IF` $(largest \neq i)$ `THEN`
6 exchange $A[i]$ with $A[largest]$
7 `Max-Heapify`(A, largest)



Here, $A.heap\_size = 10$

call `Max-Heapify`$(A, 2)$

1 $l := 4$, $r := 5$,
2 $largest := 2$
3 $A[4] = 14 > A[2] = 4$
$\Longrightarrow$ $largest := 4$
4 $A[5] = 7 \not> A[4]$.
5-7 $largest \neq 2$
exchange $A[2]$ with $A[4]$
call `Max-Heapify`$(A, 4)$

`Max-Heapify` lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

This is achieved by exchanging $A[i]$ recursively with the largest element of the children

Part 2: Heapsort - procedure MAX-HEAPIFY

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

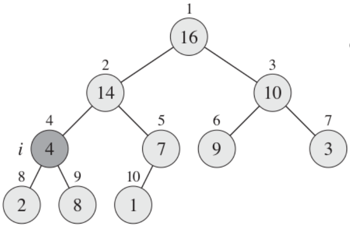
Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i+1]$$

$\implies$ i violates the max-heap property.

`Max-Heapify`(A, i)

1 $l := 2i$ and $r := 2i + 1$
2 $largest := i$
3 IF $(l \leq A.heap\_size$ and $A[l] > A[largest])$
  THEN $largest := l$
4 IF $(r \leq A.heap\_size$ and $A[r] > A[largest])$
  THEN $largest := r$
5 IF $(largest \neq i)$ THEN
6   exchange $A[i]$ with $A[largest]$
7   `Max-Heapify`(A, largest)



Here, $A.heap\_size = 10$

call `Max-Heapify`$(A, 4)$

1 $l := 8$, $r := 9$,
2 $largest := 4$
3 $A[8] = 2 \not> A[4] = 4$
4 $A[9] = 8 > A[4] = 4$.
  $\implies$ $largest := 9$
5-7 $largest \neq 4$
  exchange $A[4]$ with $A[9]$
  call `Max-Heapify`$(A, 9)$

`Max-Heapify` lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

This is achieved by exchanging $A[i]$ recursively with the largest element of the children

Part 2: Heapsort - procedure MAX-HEAPIFY

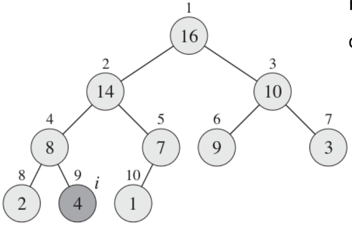
max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i + 1]$$

$\Longrightarrow$ i violates the max-heap property.

```
Max-Heapify(A, i)
1  l := 2i and r := 2i + 1
2  largest := i
3  IF (l ≤ A.heap_size and A[l] > A[largest])
   THEN largest := l
4  IF (r ≤ A.heap_size and A[r] > A[largest])
   THEN largest := r
5  IF (largest ≠ i) THEN
6      exchange A[i] with A[largest]
7      Max-Heapify(A, largest)
```



Here, $A.heap\_size = 10$

call Max-Heapify(A, 9)

1 $l := 18$, $r := 19$,
2 $largest := 9$
3 $l \not\leq A.heap\_size = 10$
4 $r \not\leq A.heap\_size = 10$
5-7 $largest = i = 9$
Stop.

Max-Heapify lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

This is achieved by exchanging $A[i]$ recursively with the largest element of the children

Part 2: Heapsort - procedure Max-Heapify

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

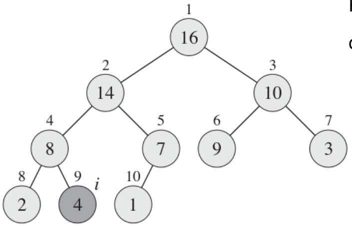
Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i+1]$$

$\Longrightarrow$ i violates the max-heap property.

`Max-Heapify`(A, i)

1 $l := 2i$ and $r := 2i + 1$
2 $largest := i$
3 IF ($l \leq A.heap\_size$ and $A[l] > A[largest]$)
  THEN $largest := l$
4 IF ($r \leq A.heap\_size$ and $A[r] > A[largest]$)
  THEN $largest := r$
5 IF ($largest \neq i$) THEN
6   exchange $A[i]$ with $A[largest]$
7   `Max-Heapify`(A, largest)



Here, $A.heap\_size = 10$

call `Max-Heapify`(A, 9)

1 $l := 18$, $r := 19$,
2 $largest := 9$
3 $l \not\leq A.heap\_size = 10$
4 $r \not\leq A.heap\_size = 10$
5-7 $largest = i = 9$
Stop.

Since we always exchanged with largest child-value, the final tree $T(i)$ is a max-heap.

Part 2: Heapsort - procedure MAX-HEAPIFY

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

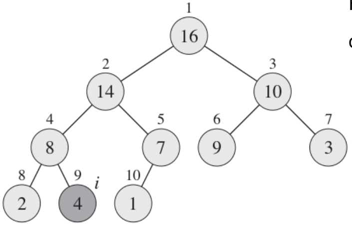
Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i+1]$$

$\implies$ i violates the max-heap property.

`Max-Heapify`(A, i)

1 $l := 2i$ and $r := 2i + 1$
2 $largest := i$
3 IF ($l \leq A.heap\_size$ and $A[l] > A[largest]$)
  THEN $largest := l$
4 IF ($r \leq A.heap\_size$ and $A[r] > A[largest]$)
  THEN $largest := r$
5 IF ($largest \neq i$) THEN
6   exchange $A[i]$ with $A[largest]$
7   `Max-Heapify`(A, largest)



Here, $A.heap\_size = 10$

call `Max-Heapify`(A, 9)

1 $l := 18$, $r := 19$,
2 $largest := 9$
3 $l \not\leq A.heap\_size = 10$
4 $r \not\leq A.heap\_size = 10$
5-7 $largest = i = 9$
Stop.

Since we always exchanged with largest child-value, the final tree $T(i)$ is a max-heap.

Runtime?

Part 2: Heapsort - procedure MAX-HEAPIFY

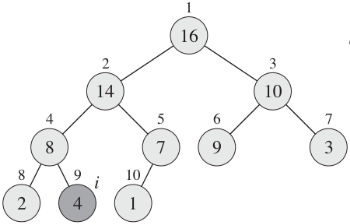
max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i+1]$$

$\implies$ i violates the max-heap property.

```
Max-Heapify(A, i)
1  l := 2i and r := 2i + 1
2  largest := i
3  IF (l ≤ A.heap_size and A[l] > A[largest])
   THEN largest := l
4  IF (r ≤ A.heap_size and A[r] > A[largest])
   THEN largest := r
5  IF (largest ≠ i) THEN
6      exchange A[i] with A[largest]
7      Max-Heapify(A, largest)
```



Here, $A.heap\_size = 10$

call Max-Heapify$(A, 9)$

1 $l := 18$, $r := 19$,
2 $largest := 9$
3 $l \not\leq A.heap\_size = 10$
4 $r \not\leq A.heap\_size = 10$
5-7 $largest = i = 9$
Stop.

Since we always exchanged with largest child-value, the final tree $T(i)$ is a max-heap.

Runtime?

Comparing values and exchange per call: $\Theta(1)$

Part 2: Heapsort - procedure MAX-HEAPIFY

max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

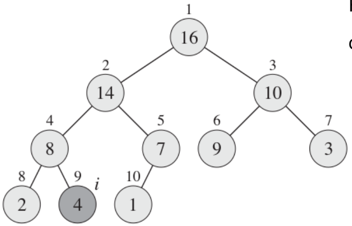
Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i+1]$$

$\Longrightarrow$ i violates the max-heap property.

`Max-Heapify`(A, i)

1 $l := 2i$ and $r := 2i + 1$
2 $largest := i$
3 IF ($l \leq A.heap\_size$ and $A[l] > A[largest]$)
  THEN $largest := l$
4 IF ($r \leq A.heap\_size$ and $A[r] > A[largest]$)
  THEN $largest := r$
5 IF ($largest \neq i$) THEN
6   exchange $A[i]$ with $A[largest]$
7   `Max-Heapify`(A, largest)



Here, $A.heap\_size = 10$

call `Max-Heapify`$(A, 9)$

1 $l := 18$, $r := 19$,
2 $largest := 9$
3 $l \not\leq A.heap\_size = 10$
4 $r \not\leq A.heap\_size = 10$
5-7 $largest = i = 9$
Stop.

Since we always exchanged with largest child-value, the final tree $T(i)$ is a max-heap.

Runtime?
Comparing values and exchange per call: $\Theta(1)$
\# of calls: $O(\log(n))$ as $h(T) \in O(\log(n))$ or via master theorem [exercise - see Cormen Sec 6.2]

Part 2: Heapsort - procedure MAX-HEAPIFY

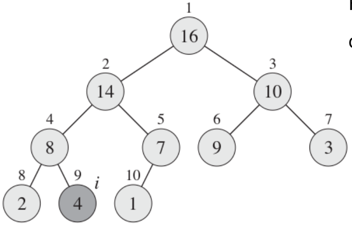
max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i + 1]$$

$\Longrightarrow$ i violates the max-heap property.

```
Max-Heapify(A, i)
  1  l := 2i and r := 2i + 1
  2  largest := i
  3  IF (l ≤ A.heap_size and A[l] > A[largest])
     THEN largest := l
  4  IF (r ≤ A.heap_size and A[r] > A[largest])
     THEN largest := r
  5  IF (largest ≠ i) THEN
  6      exchange A[i] with A[largest]
  7      Max-Heapify(A, largest)
```



Here, $A.heap\_size = 10$

call Max-Heapify(A, 9)

1 $l := 18$, $r := 19$,
2 $largest := 9$
3 $l \not\leq A.heap\_size = 10$
4 $r \not\leq A.heap\_size = 10$
5-7 $largest = i = 9$
Stop.

Since we always exchanged with largest child-value, the final tree $T(i)$ is a max-heap.

Runtime?

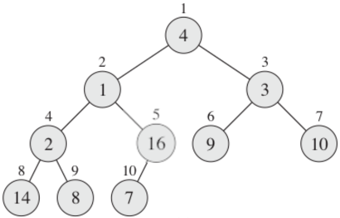
Comparing values and exchange per call: $\Theta(1)$

\# of calls: $O(\log(n))$ as $h(T) \in O(\log(n))$ or via master theorem [exercise - see Cormen Sec 6.2]

$\Longrightarrow$ Total runtime is $O(\log(n))$

Part 2: Heapsort - Building a heap

Every array *A* can naturally be viewed as a heap (however, *A* might not be a max-heap)

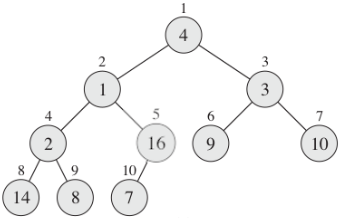


$A = [\quad 4_{1} \mid 1_{2} \mid 3_{3} \mid 2_{4} \mid 16_{5} \mid 9_{6} \mid 10_{7} \mid 14_{8} \mid 8_{9} \mid 7_{10} \quad]$

To transform *A* into a max-heap, we may simply apply `Max-Heapify` bottom-up to each *i* violating the max-heap property.

Part 2: Heapsort - Building a heap

Every array A can naturally be viewed as a heap (however, A might not be a max-heap)

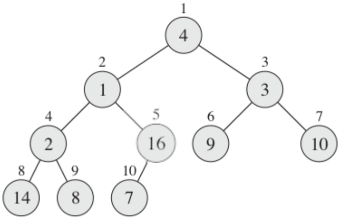



```
Build-Max-Heap(A)
    A.heap_size = A.length
    FOR (i = ⌊A.length/2⌋ TO 1 DO
        Max-Heapify(A, i)
```

$A = [\ 4_{1} \mid 1_{2} \mid 3_{3} \mid 2_{4} \mid 16_{5} \mid 9_{6} \mid 10_{7} \mid 14_{8} \mid 8_{9} \mid 7_{10}\ ]$

To transform A into a max-heap, we may simply apply `Max-Heapify` bottom-up to each i violating the max-heap property.

Every array A can naturally be viewed as a heap (however, A might not be a max-heap)



`Build-Max-Heap`(A)

$A.heap\_size = A.length$
`FOR` ($i = \lfloor A.length/2 \rfloor$ `TO` 1 `DO`
    `Max-Heapify`(A, i)

$A = [\ 4_1 \mid 1_2 \mid 3_3 \mid 2_4 \mid 16_5 \mid 9_6 \mid 10_7 \mid 14_8 \mid 8_9 \mid 7_{10}\ ]$

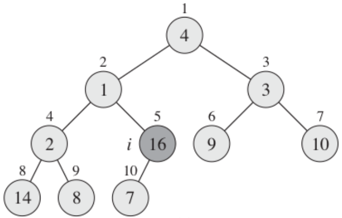
To transform A into a max-heap, we may simply apply `Max-Heapify` bottom-up to each i violating the max-heap property.

Lemma. *If A is a heap with $n = A.heap\_size$, then the leaves in the tree representation have indices $i \in I = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.*

For each leaf i, $T(i)$ is a single vertex and thus, already a max-heap.

Part 2: Heapsort - Building a heap

Every array A can naturally be viewed as a heap (however, A might not be a max-heap)



`Build-Max-Heap`(A)

$A.heap\_size = A.length$

`FOR` $(i = \lfloor A.length/2 \rfloor$ `TO` 1 `DO`

    `Max-Heapify`(A, i)

$A = [\ 4_1 \mid 1_2 \mid 3_3 \mid 2_4 \mid 16_5 \mid 9_6 \mid 10_7 \mid 14_8 \mid 8_9 \mid 7_{10}\ ]$

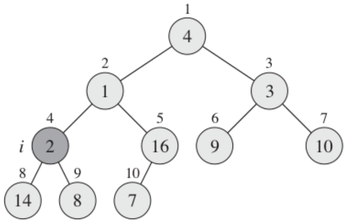
To transform A into a max-heap, we may simply apply `Max-Heapify` bottom-up to each i violating the max-heap property.

Lemma. *If A is a heap with $n = A.heap\_size$, then the leaves in the tree representation have indices $i \in I = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.*

For each leaf i, $T(i)$ is a single vertex and thus, already a max-heap.

Part 2: Heapsort - Building a heap

Every array A can naturally be viewed as a heap (however, A might not be a max-heap)



`Build-Max-Heap`(A)

```
A.heap_size = A.length
FOR (i = ⌊A.length/2⌋ TO 1 DO
    Max-Heapify(A, i)
```

$A = [\ 4_1 \mid 1_2 \mid 3_3 \mid 2_4 \mid 16_5 \mid 9_6 \mid 10_7 \mid 14_8 \mid 8_9 \mid 7_{10}\ ]$

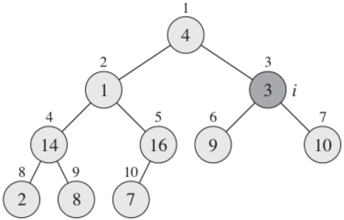
To transform A into a max-heap, we may simply apply `Max-Heapify` bottom-up to each i violating the max-heap property.

Lemma. *If A is a heap with $n = A.heap\_size$, then the leaves in the tree representation have indices $i \in I = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.*

For each leaf i, $T(i)$ is a single vertex and thus, already a max-heap.

Part 2: Heapsort - Building a heap

Every array A can naturally be viewed as a heap (however, A might not be a max-heap)



`Build-Max-Heap`(A)

$A.heap\_size = A.length$

`FOR` $(i = \lfloor A.length/2 \rfloor$ `TO` 1 `DO`

  `Max-Heapify`(A, i)

$A = [\ 4_{1} \mid 1_{2} \mid 3_{3} \mid 14_{4} \mid 16_{5} \mid 9_{6} \mid 10_{7} \mid 2_{8} \mid 8_{9} \mid 7_{10}\ ]$

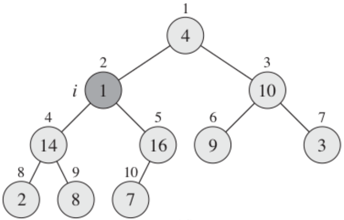
To transform A into a max-heap, we may simply apply `Max-Heapify` bottom-up to each i violating the max-heap property.

Lemma. *If A is a heap with $n = A.heap\_size$, then the leaves in the tree representation have indices $i \in I = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.*

For each leaf i, $T(i)$ is a single vertex and thus, already a max-heap.

Part 2: Heapsort - Building a heap

Every array A can naturally be viewed as a heap (however, A might not be a max-heap)



A : 4 $_1$ | 1 $_2$ | 10 $_3$ | 14 $_4$ | 16 $_5$ | 9 $_6$ | 3 $_7$ | 2 $_8$ | 8 $_9$ | 7 $_{10}$

```
Build-Max-Heap(A)
  A.heap_size = A.length
  FOR (i = ⌊A.length/2⌋ TO 1 DO
      Max-Heapify(A, i)
```

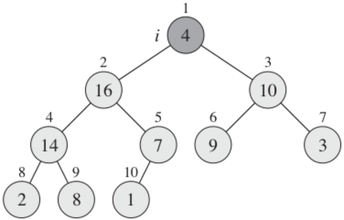
To transform A into a max-heap, we may simply apply `Max-Heapify` bottom-up to each i violating the max-heap property.

Lemma. *If A is a heap with $n = A.heap\_size$, then the leaves in the tree representation have indices $i \in I = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.*

For each leaf i, $T(i)$ is a single vertex and thus, already a max-heap.

Part 2: Heapsort - Building a heap

Every array A can naturally be viewed as a heap (however, A might not be a max-heap)



Build-Max-Heap(A)

```
A.heap_size = A.length
FOR (i = ⌊A.length/2⌋ TO 1 DO
    Max-Heapify(A, i)
```

$A = [\quad 4_{\,1} \mid 16_{\,2} \mid 10_{\,3} \mid 14_{\,4} \mid 7_{\,5} \mid 9_{\,6} \mid 3_{\,7} \mid 2_{\,8} \mid 8_{\,9} \mid 1_{\,10} \quad]$

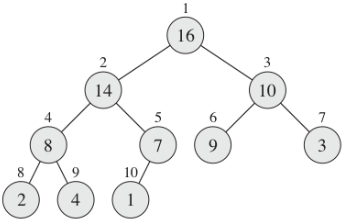
To transform A into a max-heap, we may simply apply Max-Heapify bottom-up to each i violating the max-heap property.

Lemma. *If A is a heap with $n = A.heap\_size$, then the leaves in the tree representation have indices $i \in I = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.*

For each leaf i, $T(i)$ is a single vertex and thus, already a max-heap.

Part 2: Heapsort - Building a heap

Every array A can naturally be viewed as a heap (however, A might not be a max-heap)



Build-Max-Heap(A)

```
A.heap_size = A.length
FOR (i = ⌊A.length/2⌋ TO 1 DO
    Max-Heapify(A, i)
```

$A = [\quad 16_{\,1} \mid 14_{\,2} \mid 10_{\,3} \mid 8_{\,4} \mid 7_{\,5} \mid 9_{\,6} \mid 3_{\,7} \mid 2_{\,8} \mid 4_{\,9} \mid 1_{\,10} \quad]$

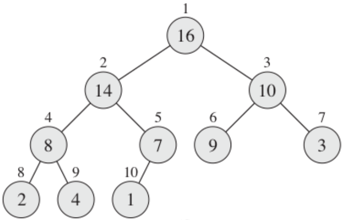
To transform A into a max-heap, we may simply apply Max-Heapify bottom-up to each i violating the max-heap property.

Lemma. *If A is a heap with $n = A.heap\_size$, then the leaves in the tree representation have indices $i \in I = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.*

For each leaf i, $T(i)$ is a single vertex and thus, already a max-heap.

Part 2: Heapsort - Building a heap

Every array A can naturally be viewed as a heap (however, A might not be a max-heap)



Build-Max-Heap(A)

$A.heap\_size = A.length$

FOR ($i = \lfloor A.length/2 \rfloor$ TO 1 DO

  Max-Heapify(A, i)

$A = [\quad 16_1 \,|\, 14_2 \,|\, 10_3 \,|\, 8_4 \,|\, 7_5 \,|\, 9_6 \,|\, 3_7 \,|\, 2_8 \,|\, 4_9 \,|\, 1_{10} \quad]$

To transform A into a max-heap, we may simply apply Max-Heapify bottom-up to each i violating the max-heap property.

Lemma. *If A is a heap with $n = A.heap\_size$, then the leaves in the tree representation have indices $i \in I = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.*

For each leaf i, $T(i)$ is a single vertex and thus, already a max-heap.

Theorem. Build-Max-Heap(A) *correctly transforms A into a max-heap in $O(A.heap\_size)$ time*

[proof correctness on board, proof runtime omitted - rather lengthy (ideas in Cormen Sec 6.3) - full proof 5 pages in my notes]

Part 2: Heapsort - the algorithm

Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

Part 2: Heapsort - the algorithm

Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1 Transform A to a max-heap

2 By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)

3 Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n

4 Goto Step 1 with $A[1..n-1]$ “playing the role” of A and repeat until all elements in A have been processed

Part 2: Heapsort - the algorithm

Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1 Transform A to a max-heap

2 By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)

3 Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n

4 Goto Step 1 with $A[1..n-1]$ “playing the role” of A and repeat until all elements in A have been processed

Part 2: Heapsort - the algorithm

Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

Part 2: Heapsort - the algorithm

Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ “playing the role” of A and repeat until all elements in A have been processed

Part 2: Heapsort - the algorithm

Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
Build-Max-Heap(A)
FOR i = A.length TO 2 DO
    Exchange A[1] with A[i]
    A.heap_size := A.heap_size − 1
    Max-Heapify(A, 1)
```

Part 2: Heapsort - the algorithm

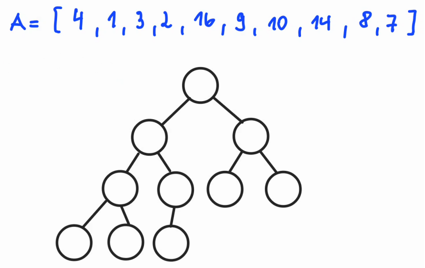
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1 Transform A to a max-heap
2 By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3 Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4 Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size − 1
      Max-Heapify(A, 1)
```



Part 2: Heapsort - the algorithm

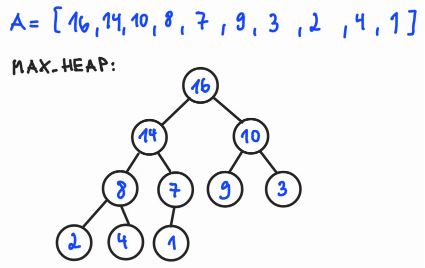
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size − 1
      Max-Heapify(A, 1)
```



Part 2: Heapsort - the algorithm

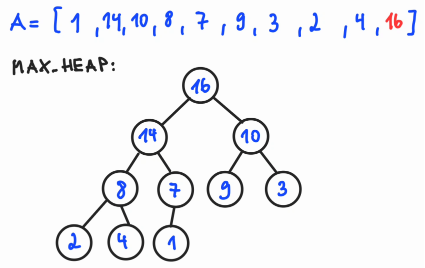
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ “playing the role” of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size − 1
      Max-Heapify(A, 1)
```



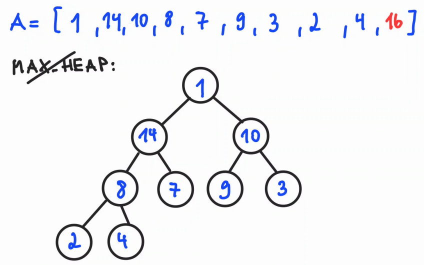
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size − 1
      Max-Heapify(A, 1)
```



Part 2: Heapsort - the algorithm

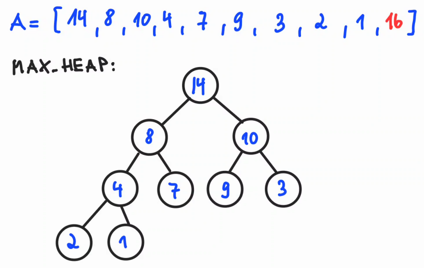
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ “playing the role” of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size - 1
      Max-Heapify(A, 1)
```



Part 2: Heapsort - the algorithm

Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

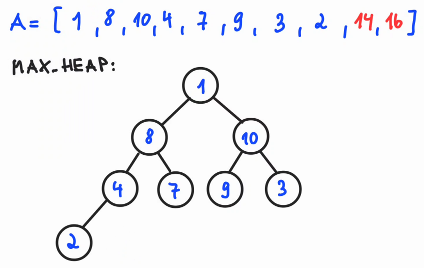
Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

Heapsort(A)

```
Build-Max-Heap(A)
FOR i = A.length TO 2 DO
    Exchange A[1] with A[i]
    A.heap_size := A.heap_size − 1
    Max-Heapify(A, 1)
```



Part 2: Heapsort - the algorithm

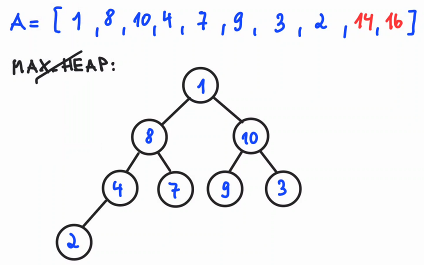
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size − 1
      Max-Heapify(A, 1)
```



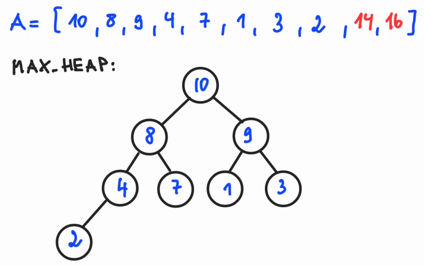
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size − 1
      Max-Heapify(A, 1)
```



Part 2: Heapsort - the algorithm

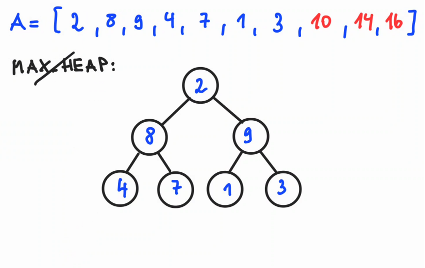
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size - 1
      Max-Heapify(A, 1)
```



Part 2: Heapsort - the algorithm

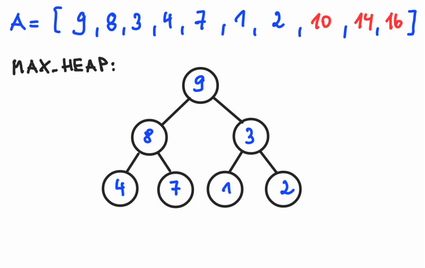
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ “playing the role” of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size − 1
      Max-Heapify(A, 1)
```



Part 2: Heapsort - the algorithm

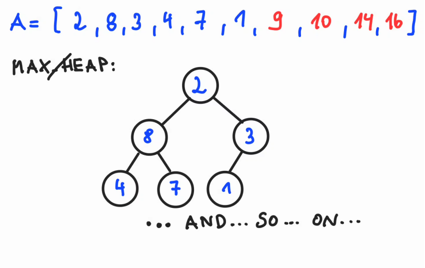
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ "playing the role" of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size − 1
      Max-Heapify(A, 1)
```



Part 2: Heapsort - the algorithm

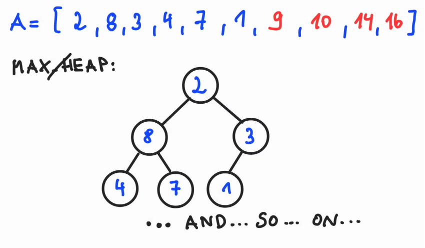
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ “playing the role” of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
  Build-Max-Heap(A)
  FOR i = A.length TO 2 DO
      Exchange A[1] with A[i]
      A.heap_size := A.heap_size - 1
      Max-Heapify(A, 1)
```



Theorem. Heapsort(A) *correctly orders A in $O(n \log(n))$ time ($n = A.length$)*

Part 2: Heapsort - the algorithm

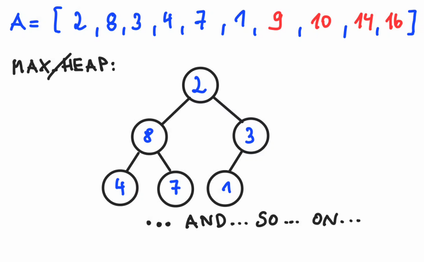
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

1. Transform A to a max-heap
2. By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3. Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4. Goto Step 1 with $A[1..n-1]$ “playing the role” of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

```
Heapsort(A)
Build-Max-Heap(A)
FOR i = A.length TO 2 DO
    Exchange A[1] with A[i]
    A.heap_size := A.heap_size − 1
    Max-Heapify(A, 1)
```



Theorem. Heapsort(A) *correctly orders A in $O(n \log(n))$ time ($n = A.length$)*

Proof: By the latter arguments, Heapsort is correct. Runtime: $(n-1)$ calls of Max-Heapify each takes $O(\log(n))$ time. $\square$

Part 2: Quicksort

Part 2: Quicksort

`Quicksort`: in place sorting, but worst-case runtime $\Theta(n^2)$. However, its expected runtime is $\Theta(n \log n)$ and in practice it outperforms heapsort.

Part 2: Quicksort

Part 2: Quicksort

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Quicksort

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Quicksort

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Quicksort

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

quicksort: partition/rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that all elements in $A[p..q-1]$ are less than or equal to the **pivot** $A[q]$, which is, in turn, less than or equal to each element $A[q+1..r]$. Compute the index q of the pivot as part of this partitioning procedure.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Quicksort

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

quicksort: partition/rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that all elements in $A[p..q-1]$ are less than or equal to the **pivot** $A[q]$, which is, in turn, less than or equal to each element $A[q+1..r]$. Compute the index q of the pivot as part of this partitioning procedure.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Quicksort

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

quicksort: partition/rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that all elements in $A[p..q-1]$ are less than or equal to the **pivot** $A[q]$, which is, in turn, less than or equal to each element $A[q+1..r]$. Compute the index q of the pivot as part of this partitioning procedure.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

quicksort: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Quicksort

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

quicksort: partition/rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that all elements in $A[p..q-1]$ are less than or equal to the **pivot** $A[q]$, which is, in turn, less than or equal to each element $A[q+1..r]$. Compute the index q of the pivot as part of this partitioning procedure.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

quicksort: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine the solutions to the subproblems into the solution for the original problem.

Part 2: Quicksort

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

quicksort: partition/rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that all elements in $A[p..q-1]$ are less than or equal to the **pivot** $A[q]$, which is, in turn, less than or equal to each element $A[q+1..r]$. Compute the index q of the pivot as part of this partitioning procedure.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

quicksort: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine the solutions to the subproblems into the solution for the original problem.

quicksort: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
IF (p < r) THEN
    q = Partition(A, p, r)
    Quicksort(A, p, q − 1)
    Quicksort(A, q + 1, r)
```

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

quicksort: partition/rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that all elements in $A[p..q-1]$ are less than or equal to the **pivot** $A[q]$, which is, in turn, less than or equal to each element $A[q+1..r]$. Compute the index q of the pivot as part of this partitioning procedure.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

quicksort: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine the solutions to the subproblems into the solution for the original problem.

quicksort: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
IF (p < r) THEN
    q = Partition(A, p, r)
    Quicksort(A, p, q − 1)
    Quicksort(A, q + 1, r)
```

The key to `Quicksort` is the `Partition` procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is
`Quicksort`$(A, 1, A.length)$

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

quicksort: partition/rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that all elements in $A[p..q-1]$ are less than or equal to the **pivot** $A[q]$, which is, in turn, less than or equal to each element $A[q+1..r]$. Compute the index q of the pivot as part of this partitioning procedure.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

quicksort: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine the solutions to the subproblems into the solution for the original problem.

quicksort: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
IF (p < r) THEN
    q = Partition(A, p, r)
    Quicksort(A, p, q − 1)
    Quicksort(A, q + 1, r)
```

The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
x = A[r] //pivot (other ways to pick x are possible!)
i = p − 1 //i is highest index of low side
FOR j = p TO r − 1 DO
//process each element other than pivot
  IF (A[j] ≤ x) THEN
  //does this element belong on the low side?
    i := i + 1 //index of a new slot in the low side
    exchange A[i] with A[j]
exchange A[i + 1] with A[r]
//pivot goes just to the right of the low side
RETURN i + 1
```

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

quicksort: partition/rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that all elements in $A[p..q-1]$ are less than or equal to the **pivot** $A[q]$, which is, in turn, less than or equal to each element $A[q+1..r]$. Compute the index q of the pivot as part of this partitioning procedure.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

quicksort: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine the solutions to the subproblems into the solution for the original problem.

quicksort: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
  IF (p < r) THEN
      q = Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)
```

The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is
Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
  x = A[r] //pivot (other ways to pick x are possible!)
  i = p − 1 //i is highest index of low side
  FOR j = p TO r − 1 DO
  //process each element other than pivot
    IF (A[j] ≤ x) THEN
    //does this element belong on the low side?
      i := i + 1 //index of a new slot in the low side
      exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  //pivot goes just to the right of the low side
  RETURN i + 1
```

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

quicksort: partition/rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that all elements in $A[p..q-1]$ are less than or equal to the **pivot** $A[q]$, which is, in turn, less than or equal to each element $A[q+1..r]$. Compute the index q of the pivot as part of this partitioning procedure.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

quicksort: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine the solutions to the subproblems into the solution for the original problem.

quicksort: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

i	p,j							r
	2	8	7	1	3	5	6	4

$x = 4$
$i = p-1$
$j = p$

$A[j] = 2 \le 4$
$i \leftarrow i+1$

Exchange $A[i]$ with $A[j]$ has no effect now.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
  IF (p < r) THEN
      q = Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)
```

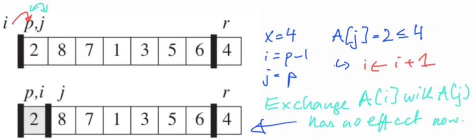
The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
  x = A[r] //pivot (other ways to pick x are possible!)
  i = p − 1 //i is highest index of low side
  FOR j = p TO r − 1 DO
  //process each element other than pivot
    IF (A[j] ≤ x) THEN
    //does this element belong on the low side?
      i := i + 1 //index of a new slot in the low side
      exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  //pivot goes just to the right of the low side
  RETURN i + 1
```



Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
   IF (p < r) THEN
      q = Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)
```

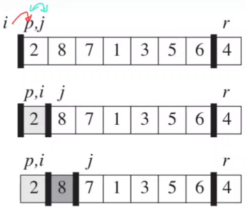
The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
   x = A[r] //pivot (other ways to pick x are possible!)
   i = p − 1 //i is highest index of low side
   FOR j = p TO r − 1 DO
   //process each element other than pivot
      IF (A[j] ≤ x) THEN
      //does this element belong on the low side?
         i := i + 1 //index of a new slot in the low side
         exchange A[i] with A[j]
   exchange A[i + 1] with A[r]
   //pivot goes just to the right of the low side
   RETURN i + 1
```



$x=4$

$i=p-1$

$j=p$

$A[j]=2\le 4$

↳ $i \leftarrow i+1$

Exchange $A[i]$ with $A[j]$ has no effect now.

$A[j] \not\le x$.

Part 2: Quicksort

`Quicksort`(A, p, r) //$A[1]$=first entry of A

  IF $(p < r)$ THEN
    $q =$ `Partition`(A, p, r)
    `Quicksort`$(A, p, q-1)$
    `Quicksort`$(A, q+1, r)$

The key to `Quicksort` is the `Partition` procedure, which rearranges the subarray $A[p..r]$ in place.

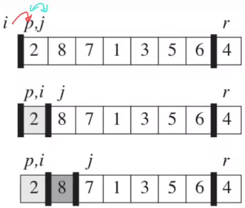
To sort an entire array A, the initial call is

`Quicksort`$(A, 1, A.length)$

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

`Partition`(A, p, r)

  $x = A[r]$ //pivot (other ways to pick x are possible!)
  $i = p - 1$ //i is highest index of low side
  FOR $j = p$ TO $r - 1$ DO
  //process each element other than pivot
    IF $(A[j] \leq x)$ THEN
    //does this element belong on the low side?
      $i := i + 1$ //index of a new slot in the low side
      exchange $A[i]$ with $A[j]$
  exchange $A[i+1]$ with $A[r]$
  //pivot goes just to the right of the low side
  RETURN $i + 1$



$x=4$ $A[j]=2\leq 4$
$i=p-1$ ↳ $i \leftarrow i+1$
$j=p$
Exchange $A[i]$ with $A[j]$ has no effect now.

$A[j] \nleq x$.

again $A[j] \nleq x$

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

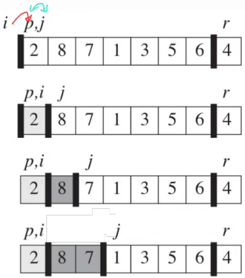
The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```



$x = 4$
$i = p-1$
$j = p$

$A[j] = 2 \leq 4$
↳ $i \leftarrow i+1$

Exchange $A[i]$ with $A[j]$ has no effect now.

$A[j] \nleq x$.

again $A[j] \nleq x$

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

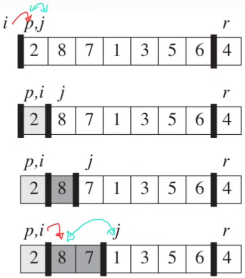
The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
       IF (A[j] ≤ x) THEN
       //does this element belong on the low side?
         i := i + 1 //index of a new slot in the low side
         exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```



$x=4$
$i=p-1$
$j=p$

$A[j]=2\le 4$
$\hookrightarrow i \leftarrow i+1$

Exchange $A[i]$ with $A[j]$ has no effect now.

$A[j] \nleq x$.

again $A[j] \nleq x$

$A[j] \le x$
$\Rightarrow i \leftarrow i+1$
Exchange $A[i]$ & $A[j]$

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

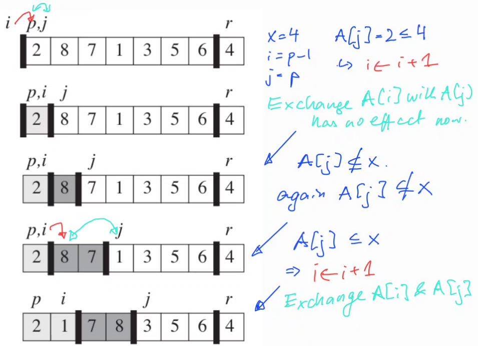
The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```



Part 2: Quicksort

`Quicksort`(A, p, r) //$A[1]$=first entry of A

- IF $(p < r)$ THEN
  - $q =$ `Partition`(A, p, r)
  - `Quicksort`$(A, p, q-1)$
  - `Quicksort`$(A, q+1, r)$

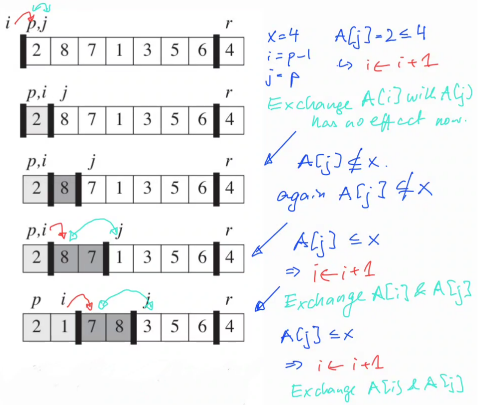
The key to `Quicksort` is the `Partition` procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is
`Quicksort`$(A, 1, A.length)$

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

`Partition`(A, p, r)

- $x = A[r]$ //pivot (other ways to pick x are possible!)
- $i = p - 1$ //i is highest index of low side
- FOR $j = p$ TO $r - 1$ DO
  //process each element other than pivot
  - IF $(A[j] \leq x)$ THEN
    //does this element belong on the low side?
    - $i := i + 1$ //index of a new slot in the low side
    - exchange $A[i]$ with $A[j]$
- exchange $A[i+1]$ with $A[r]$
  //pivot goes just to the right of the low side
- RETURN $i + 1$



Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

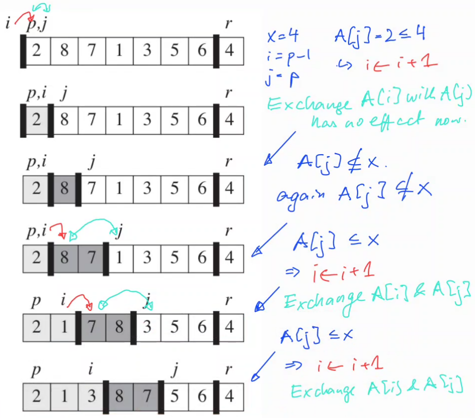
The key to `Quicksort` is the `Partition` procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

`Quicksort`$(A, 1, A.length)$

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```



Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

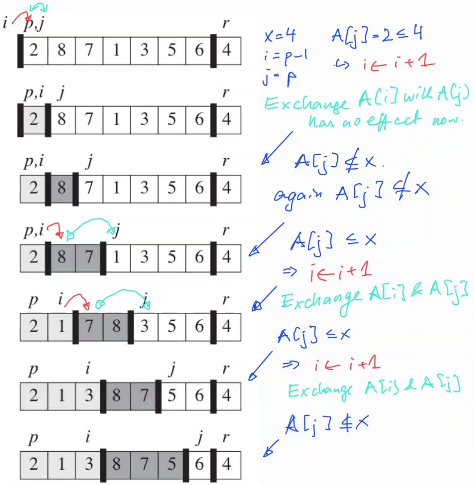
The key to `Quicksort` is the `Partition` procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

`Quicksort`$(A, 1, A.length)$

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
       IF (A[j] ≤ x) THEN
       //does this element belong on the low side?
         i := i + 1 //index of a new slot in the low side
         exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```



Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
  IF (p < r) THEN
      q = Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)
```

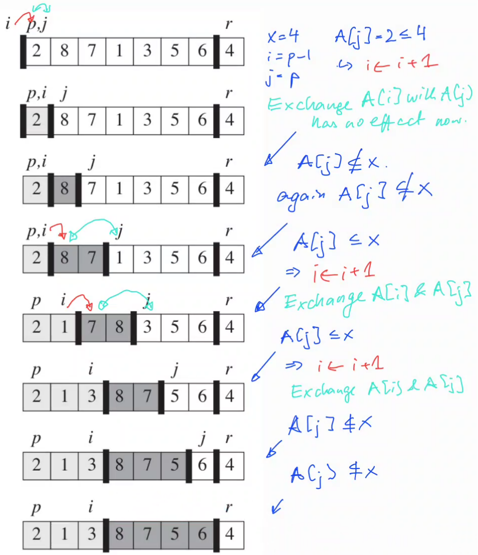
The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
  x = A[r] //pivot (other ways to pick x are possible!)
  i = p − 1 //i is highest index of low side
  FOR j = p TO r − 1 DO
  //process each element other than pivot
    IF (A[j] ≤ x) THEN
    //does this element belong on the low side?
      i := i + 1 //index of a new slot in the low side
      exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  //pivot goes just to the right of the low side
  RETURN i + 1
```



Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

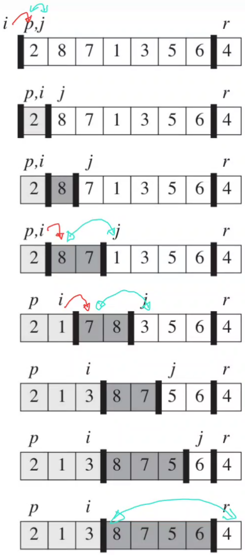
The key to `Quicksort` is the `Partition` procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

`Quicksort`$(A, 1, A.length)$

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```



$x=4$ $A[j]=2\leq 4$
$i=p-1$ ↳ $i \leftarrow i+1$
$j=p$
Exchange $A[i]$ with $A[j]$ has no effect now.

$A[j] \nleq x$.

again $A[j] \nleq x$

$A[j] \leq x$
$\Rightarrow i \leftarrow i+1$
Exchange $A[i]$ & $A[j]$

$A[j] \leq x$
$\Rightarrow i \leftarrow i+1$
Exchange $A[i]$ & $A[j]$

$A[j] \nleq x$

$A[j] \nleq x$

Exchange $A[i+1]$ with $A[r]$

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

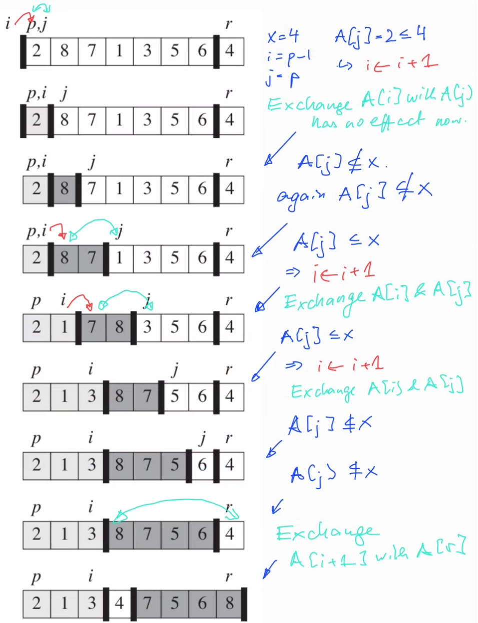
The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
       IF (A[j] ≤ x) THEN
       //does this element belong on the low side?
          i := i + 1 //index of a new slot in the low side
          exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```



Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

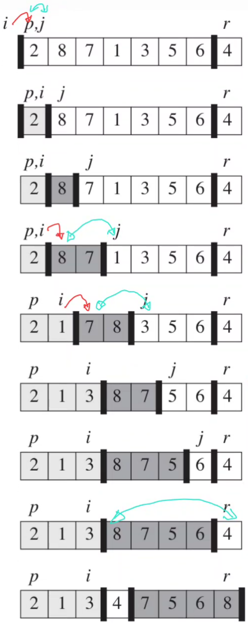
The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```



$x=4$ $A[j]=2\le 4$
$i=p-1$ ↳ $i \leftarrow i+1$
$j=p$
Exchange $A[i]$ with $A[j]$ has no effect now.

$A[j] \not\le x$.

again $A[j] \not\le x$

$A[j] \le x$
$\Rightarrow i \leftarrow i+1$
Exchange $A[i]$ & $A[j]$

$A[j] \le x$
$\Rightarrow i \leftarrow i+1$
Exchange $A[i]$ & $A[j]$

$A[j] \not\le x$

$A[j] \not\le x$

Exchange $A[i+1]$ with $A[r]$

q=4 and call
Quicksort(A,1,3) and
Quicksort(A,4,8) and

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
  IF (p < r) THEN
      q = Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)
```

The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition(A, p, r)
  x = A[r] //pivot (other ways to pick x are possible!)
  i = p − 1 //i is highest index of low side
  FOR j = p TO r − 1 DO
  //process each element other than pivot
     IF (A[j] ≤ x) THEN
     //does this element belong on the low side?
       i := i + 1 //index of a new slot in the low side
       exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  //pivot goes just to the right of the low side
  RETURN i + 1
```

In what follows we show:

Thm. Quicksort(A, 1, n) *correctly sorts the array (in place) in* $O(n^2)$ *time,* $n = A.length$.

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the* $n!$ *possible permutations of the entries in A are equiprobable), then the average time of* Quicksort(A, 1, n) *is in* $O(n \log n)$.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p \,.\,.\, q-1]$ = **low side**
$A[q+1 \,.\,.\, r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. `Quicksort`$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.* ($n = A.length$)

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p\,..\,q-1]$ = **low side**
$A[q+1\,..\,r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. `Quicksort`$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.*
($n = A.length$)

Proof. correctness: easy exercise - see Cormen Sec 7.1.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p\,..\,q-1]$ = **low side**
$A[q+1\,..\,r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. Quicksort$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.*
($n = A.length$)

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. Quicksort$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.*
($n = A.length$)

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \text{PARTITION}(A, 1, n)$ runs in $\Theta(n)$ time.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
IF (p < r) THEN
    q = Partition(A, p, r)
    Quicksort(A, p, q − 1)
    Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
x = A[r] //pivot (other ways to pick x are possible!)
i = p − 1 //i is highest index of low side
FOR j = p TO r − 1 DO
//process each element other than pivot
   IF (A[j] ≤ x) THEN
   //does this element belong on the low side?
      i := i + 1 //index of a new slot in the low side
      exchange A[i] with A[j]
exchange A[i + 1] with A[r]
//pivot goes just to the right of the low side
RETURN i + 1
```

Thm. `Quicksort`$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.*
($n = A.length$)

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p..q-1]$ and $A[q+1..r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
  IF (p < r) THEN
      q = Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
  x = A[r] //pivot (other ways to pick x are possible!)
  i = p − 1 //i is highest index of low side
  FOR j = p TO r − 1 DO
  //process each element other than pivot
     IF (A[j] ≤ x) THEN
     //does this element belong on the low side?
        i := i + 1 //index of a new slot in the low side
        exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  //pivot goes just to the right of the low side
  RETURN i + 1
```

Thm. Quicksort$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.*
($n = A.length$)

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p..q-1]$ and $A[q+1..r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

$$\implies T(n) = \max_{0 \le \ell \le n-1} \{T(\ell) + T(n-1-\ell)\} + \Theta(n)$$

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
  IF (p < r) THEN
      q = Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
  x = A[r] //pivot (other ways to pick x are possible!)
  i = p − 1 //i is highest index of low side
  FOR j = p TO r − 1 DO
  //process each element other than pivot
     IF (A[j] ≤ x) THEN
     //does this element belong on the low side?
        i := i + 1 //index of a new slot in the low side
        exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  //pivot goes just to the right of the low side
  RETURN i + 1
```

Thm. Quicksort$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.* $(n = A.length)$

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \text{PARTITION}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p..q-1]$ and $A[q+1..r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

$$\Longrightarrow \quad T(n) = \max_{0 \le \ell \le n-1} \{T(\ell) + T(n-1-\ell)\} + \Theta(n)$$

Show by induction $T(n) \in O(n^2)$

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p\,..\,q-1]$ = **low side**
$A[q+1\,..\,r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. Quicksort$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.*
($n = A.length$)

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p\,..\,q-1]$ and $A[q+1\,..\,r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

$$\implies T(n) = \max_{0 \le \ell \le n-1} \{T(\ell) + T(n-1-\ell)\} + \Theta(n)$$

Show by induction $T(n) \in O(n^2)$

Base case $n = 1$: $T(n) = T(0) + T(0) + \Theta(1) = \Theta(1) = \Theta(1^2)$, since the recursive call on array of size 0 just returns.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. Quicksort$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.*
($n = A.length$)

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p..q-1]$ and $A[q+1..r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

$$\implies T(n) = \max_{0 \le \ell \le n-1} \{T(\ell) + T(n-1-\ell)\} + \Theta(n)$$

Show by induction $T(n) \in O(n^2)$

Base case $n = 1$: $T(n) = T(0) + T(0) + \Theta(1) = \Theta(1) = \Theta(1^2)$, since the recursive call on array of size 0 just returns.

Assume, the statement is true for all instance of size $< n$. Consider an instance of size n

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p\,..\,q-1]$ = **low side**
$A[q+1\,..\,r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. Quicksort$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.* *($n = A.length$)*

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p\,..\,q-1]$ and $A[q+1\,..\,r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

$$\implies T(n) = \max_{0\le\ell\le n-1} \{T(\ell) + T(n-1-\ell)\} + \Theta(n)$$

Show by induction $T(n) \in O(n^2)$

Base case $n = 1$: $T(n) = T(0) + T(0) + \Theta(1) = \Theta(1) = \Theta(1^2)$, since the recursive call on array of size 0 just returns.

Assume, the statement is true for all instance of size $< n$. Consider an instance of size n

By Ind-hyp., $T(i) \in O(i^2)$ and thus, $T(i) \le c \cdot i^2$ for all $i < n$ and large enough constants c

$$\implies T(n) \le \max_{0\le\ell\le n-1} \left\{c \cdot \ell^2 + c \cdot (n-1-\ell)^2\right\} + \Theta(n)$$
$$= c \cdot \max_{0\le\ell\le n-1} \left\{\ell^2 + (n-1-\ell)^2\right\} + \Theta(n)$$

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. Quicksort$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.* ($n = A.length$)

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \text{PARTITION}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p..q-1]$ and $A[q+1..r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

$$\implies T(n) = \max_{0 \le \ell \le n-1} \{T(\ell) + T(n-1-\ell)\} + \Theta(n)$$

Show by induction $T(n) \in O(n^2)$

Base case $n = 1$: $T(n) = T(0) + T(0) + \Theta(1) = \Theta(1) = \Theta(1^2)$, since the recursive call on array of size 0 just returns.

Assume, the statement is true for all instance of size $< n$. Consider an instance of size n

By Ind-hyp., $T(i) \in O(i^2)$ and thus, $T(i) \le c \cdot i^2$ for all $i < n$ and large enough constants c

$$\implies T(n) \le \max_{0 \le \ell \le n-1} \left\{c \cdot \ell^2 + c \cdot (n-1-\ell)^2\right\} + \Theta(n)$$
$$= c \cdot \max_{0 \le \ell \le n-1} \left\{\ell^2 + (n-1-\ell)^2\right\} + \Theta(n)$$

For which ℓ is maximum in $f(\ell) = \ell^2 + (n-1-\ell)^2$ achieved?

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
  IF (p < r) THEN
      q = Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
  x = A[r] //pivot (other ways to pick x are possible!)
  i = p − 1 //i is highest index of low side
  FOR j = p TO r − 1 DO
  //process each element other than pivot
     IF (A[j] ≤ x) THEN
     //does this element belong on the low side?
        i := i + 1 //index of a new slot in the low side
        exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  //pivot goes just to the right of the low side
  RETURN i + 1
```

Thm. `Quicksort`$(A, 1, n)$ *correctly sorts the array (in place) in $O(n^2)$ time.* *($n = A.length$)*

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p..q-1]$ and $A[q+1..r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

$$\implies T(n) = \max_{0 \le \ell \le n-1} \{T(\ell) + T(n-1-\ell)\} + \Theta(n)$$

Show by induction $T(n) \in O(n^2)$

Base case $n = 1$: $T(n) = T(0) + T(0) + \Theta(1) = \Theta(1) = \Theta(1^2)$, since the recursive call on array of size 0 just returns.

Assume, the statement is true for all instance of size $< n$. Consider an instance of size n

By Ind-hyp., $T(i) \in O(i^2)$ and thus, $T(i) \le c \cdot i^2$ for all $i < n$ and large enough constants c

$$\implies T(n) \le \max_{0 \le \ell \le n-1} \left\{c \cdot \ell^2 + c \cdot (n-1-\ell)^2\right\} + \Theta(n)$$
$$= c \cdot \max_{0 \le \ell \le n-1} \left\{\ell^2 + (n-1-\ell)^2\right\} + \Theta(n)$$

For which ℓ is maximum in $f(\ell) = \ell^2 + (n-1-\ell)^2$ achieved? Answer: For $\ell = 0$ and $\ell = n-1$ (take first derivative $\frac{df(\ell)}{d\ell} = 0$)

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. Quicksort$(A, 1, n)$ *correctly sorts the array (in place) in* $O(n^2)$ *time.* ($n = A.length$)

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p..q-1]$ and $A[q+1..r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

$$\implies T(n) = \max_{0\le\ell\le n-1} \{T(\ell) + T(n-1-\ell)\} + \Theta(n)$$

Show by induction $T(n) \in O(n^2)$

Base case $n = 1$: $T(n) = T(0) + T(0) + \Theta(1) = \Theta(1) = \Theta(1^2)$, since the recursive call on array of size 0 just returns.

Assume, the statement is true for all instance of size $< n$. Consider an instance of size n

By Ind-hyp., $T(i) \in O(i^2)$ and thus, $T(i) \le c \cdot i^2$ for all $i < n$ and large enough constants c

$$\implies T(n) \le \max_{0\le\ell\le n-1} \left\{c\cdot\ell^2 + c\cdot(n-1-\ell)^2\right\} + \Theta(n)$$
$$= c \cdot \max_{0\le\ell\le n-1} \left\{\ell^2 + (n-1-\ell)^2\right\} + \Theta(n)$$

For which ℓ is maximum in $f(\ell) = \ell^2 + (n-1-\ell)^2$ achieved? Answer: For $\ell = 0$ and $\ell = n-1$ (take first derivative $\frac{df(\ell)}{d\ell} = 0$)

Either choice of ℓ implies that $T(n) \le c\cdot(n-1)^2 + c\cdot 0 + \Theta(n) = c(n^2 - 2n + 1) + \Theta(n) \le \tilde{c}n^2 \in O(n^2)$ □

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. Quicksort$(A, 1, n)$ *correctly sorts the array (in place) in $O(n^2)$ time.* *($n = A.length$)*

Proof. correctness: easy exercise - see Cormen Sec 7.1.

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p..q-1]$ and $A[q+1..r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

$$\implies T(n) = \max_{0\leq \ell \leq n-1} \{T(\ell) + T(n-1-\ell)\} + \Theta(n)$$

Show by induction $T(n) \in O(n^2)$

Base case $n = 1$: $T(n) = T(0) + T(0) + \Theta(1) = \Theta(1) = \Theta(1^2)$, since the recursive call on array of size 0 just returns.

Assume, the statement is true for all instance of size $< n$. Consider an instance of size n

By Ind-hyp., $T(i) \in O(i^2)$ and thus, $T(i) \leq c \cdot i^2$ for all $i < n$ and large enough constants c

$$\implies T(n) \leq \max_{0\leq \ell \leq n-1} \left\{c \cdot \ell^2 + c \cdot (n-1-\ell)^2\right\} + \Theta(n)$$
$$= c \cdot \max_{0\leq \ell \leq n-1} \left\{\ell^2 + (n-1-\ell)^2\right\} + \Theta(n)$$

For which ℓ is maximum in $f(\ell) = \ell^2 + (n-1-\ell)^2$ achieved? Answer: For $\ell = 0$ and $\ell = n-1$ (take first derivative $\frac{df(\ell)}{d\ell} = 0$)

Either choice of ℓ implies that $T(n) \leq c \cdot (n-1)^2 + c \cdot 0 + \Theta(n) = c(n^2 - 2n + 1) + \Theta(n) \leq \tilde{c}n^2 \in O(n^2)$ $\square$

In fact, there are example where worst-case is achieved, i.e., there are instances such that $T(n) \in \Omega(n^2)$ (e.g. if A is in reversed order (exercise))

These worst-case examples are rare and we can obtained better expected runtime when entries in A are are pairwise distinct and randomly distributed.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the $n!$ possible permutations of the entries in A are equiprobable), then the average time of* `Quicksort`$(A, 1, n)$ *is in* $O(n \log n)$.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p\,..\,q-1]$ = **low side**
$A[q+1\,..\,r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 ///i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the $n!$ possible permutations of the entries in A are equiprobable), then the average time of* `Quicksort`$(A, 1, n)$ *is in* $O(n \log n)$.

Proof. W.l.o.g. $n = A.length$ and $A[i] \in \{1, \ldots, n\}$. Recall that $q = \text{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Part 2: Quicksort

Quicksort(A, p, r) //$A[1]$=first entry of A

IF $(p < r)$ THEN
    $q = $ Partition(A, p, r)
    Quicksort$(A, p, q-1)$
    Quicksort$(A, q+1, r)$

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

Partition(A, p, r)

$x = A[r]$ //pivot (other ways to pick x are possible!)
$i = p - 1$ //i is highest index of low side
FOR $j = p$ TO $r - 1$ DO
//process each element other than pivot
    IF $(A[j] \leq x)$ THEN
    //does this element belong on the low side?
        $i := i + 1$ //index of a new slot in the low side
        exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
//pivot goes just to the right of the low side
RETURN $i + 1$

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the $n!$ possible permutations of the entries in A are equiprobable), then the average time of* Quicksort$(A, 1, n)$ *is in* $O(n \log n)$.

Proof. W.l.o.g. $n = A.length$ and $A[i] \in \{1, \ldots, n\}$. Recall that $q = \text{PARTITION}(A, 1, n)$ runs in $\Theta(n)$ time.

Step (1)
By induction, let us assume that $T(i) \in O(i \log i)$ for every $i \in \{1, \ldots, N\}$ for some $N \geq 2$
(base case $N = 1, 2$ check by yourself).

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p\,..\,q-1]$ = **low side**
$A[q+1\,..\,r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 ///i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the $n!$ possible permutations of the entries in A are equiprobable), then the average time of* `Quicksort`$(A, 1, n)$ *is in* $O(n \log n)$.

Proof. W.l.o.g. $n = A.length$ and $A[i] \in \{1, \ldots, n\}$. Recall that $q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Step (1)
By induction, let us assume that $T(i) \in O(i \log i)$ for every $i \in \{1, \ldots, N\}$ for some $N \geq 2$ (base case $N = 1, 2$ check by yourself).

Choose $n = N + 1$. We want to show that $T(n) \in O(n \log n)$.

Part 2: Quicksort

`Quicksort`(A, p, r) //$A[1]$=first entry of A

  `IF` $(p < r)$ `THEN`
    $q =$ `Partition`(A, p, r)
    `Quicksort`$(A, p, q - 1)$
    `Quicksort`$(A, q + 1, r)$

$A[p..q - 1]$ = **low side**
$A[q + 1..r]$ = **high side**

`Partition`(A, p, r)

  $x = A[r]$ //pivot (other ways to pick x are possible!)
  $i = p - 1$ //i is highest index of low side
  `FOR` $j = p$ `TO` $r - 1$ `DO`
  //process each element other than pivot
    `IF` $(A[j] \leq x)$ `THEN`
    //does this element belong on the low side?
      $i := i + 1$ //index of a new slot in the low side
      exchange $A[i]$ with $A[j]$
  exchange $A[i + 1]$ with $A[r]$
  //pivot goes just to the right of the low side
  `RETURN` $i + 1$

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the $n!$ possible permutations of the entries in A are equiprobable), then the average time of* `Quicksort`$(A, 1, n)$ *is in* $O(n \log n)$.

Proof. W.l.o.g. $n = A.\mathit{length}$ and $A[i] \in \{1, \ldots, n\}$. Recall that $q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Step (1)
By induction, let us assume that $T(i) \in O(i \log i)$ for every $i \in \{1, \ldots, N\}$ for some $N \geq 2$ (base case $N = 1, 2$ check by yourself).

Choose $n = N + 1$. We want to show that $T(n) \in O(n \log n)$.

By induction hypothesis: $T(i) \leq c' i \log i \leq c' n \log n$ for all $1 \leq i < n$ and large enough c'

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
  IF (p < r) THEN
      q = Partition(A, p, r)
      Quicksort(A, p, q − 1)
      Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
  x = A[r] //pivot (other ways to pick x are possible!)
  i = p − 1 //i is highest index of low side
  FOR j = p TO r − 1 DO
  //process each element other than pivot
     IF (A[j] ≤ x) THEN
     //does this element belong on the low side?
        i := i + 1 //index of a new slot in the low side
        exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  //pivot goes just to the right of the low side
  RETURN i + 1
```

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the $n!$ possible permutations of the entries in A are equiprobable), then the average time of* `Quicksort`$(A, 1, n)$ *is in* $O(n \log n)$.

Proof. W.l.o.g. $n = A.length$ and $A[i] \in \{1, \ldots, n\}$. Recall that $q =$ Partition$(A, 1, n)$ runs in $\Theta(n)$ time.

Step (1)
By induction, let us assume that $T(i) \in O(i \log i)$ for every $i \in \{1, \ldots, N\}$ for some $N \geq 2$ (base case $N = 1, 2$ check by yourself).

Choose $n = N + 1$. We want to show that $T(n) \in O(n \log n)$.

By induction hypothesis: $T(i) \leq c' i \log i \leq c' n \log n$ for all $1 \leq i < n$ and large enough c'

Step (2)
By Thm.-assumption, each $k \in \{1, \ldots, n\}$ is equally likely on some position of A.

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p\,..\,q-1]$ = **low side**
$A[q+1\,..\,r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the $n!$ possible permutations of the entries in A are equiprobable), then the average time of* `Quicksort`$(A, 1, n)$ *is in* $O(n \log n)$.

Proof. W.l.o.g. $n = A.\mathit{length}$ and $A[i] \in \{1, \ldots, n\}$. Recall that $q = \text{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Step (1)
By induction, let us assume that $T(i) \in O(i \log i)$ for every $i \in \{1, \ldots, N\}$ for some $N \geq 2$ (base case $N = 1, 2$ check by yourself).

Choose $n = N + 1$. We want to show that $T(n) \in O(n \log n)$.

By induction hypothesis: $T(i) \leq c' i \log i \leq c' n \log n$ for all $1 \leq i < n$ and large enough c'

Step (2)
By Thm.-assumption, each $k \in \{1, \ldots, n\}$ is equally likely on some position of A.

$\Longrightarrow$ with probability $\frac{1}{n}$ we have $A[n] = k$ and thus, k is a pivot in which case the problem is subdivided into two smaller problems of size $k - 1$ and $n - k$

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the n! possible permutations of the entries in A are equiprobable), then the average time of* `Quicksort`$(A, 1, n)$ *is in* $O(n \log n)$.

Proof. W.l.o.g. $n = A.length$ and $A[i] \in \{1, \dots, n\}$. Recall that $q =$ PARTITION$(A, 1, n)$ runs in $\Theta(n)$ time.

Step (1)
By induction, let us assume that $T(i) \in O(i \log i)$ for every $i \in \{1, \dots, N\}$ for some $N \geq 2$ (base case $N = 1, 2$ check by yourself).

Choose $n = N + 1$. We want to show that $T(n) \in O(n \log n)$.

By induction hypothesis: $T(i) \leq c' i \log i \leq c' n \log n$ for all $1 \leq i < n$ and large enough c'

Step (2)
By Thm.-assumption, each $k \in \{1, \dots, n\}$ is equally likely on some position of A.

$\implies$ with probability $\frac{1}{n}$ we have $A[n] = k$ and thus, k is a pivot in which case the problem is subdivided into two smaller problems of size $k-1$ and $n-k$

$$\implies T(n) = \frac{1}{n} \sum_{k=1}^{n} (T(k-1) + T(n-k)) + \Theta(n)$$

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the $n!$ possible permutations of the entries in A are equiprobable), then the average time of* `Quicksort`$(A, 1, n)$ *is in* $O(n \log n)$.

Proof. W.l.o.g. $n = A.length$ and $A[i] \in \{1, \ldots, n\}$. Recall that $q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Step (1)
By induction, let us assume that $T(i) \in O(i \log i)$ for every $i \in \{1, \ldots, N\}$ for some $N \geq 2$ (base case $N = 1, 2$ check by yourself).

Choose $n = N + 1$. We want to show that $T(n) \in O(n \log n)$.

By induction hypothesis: $T(i) \leq c' i \log i \leq c' n \log n$ for all $1 \leq i < n$ and large enough c'

Step (2)
By Thm.-assumption, each $k \in \{1, \ldots, n\}$ is equally likely on some position of A.

$\Longrightarrow$ with probability $\frac{1}{n}$ we have $A[n] = k$ and thus, k is a pivot in which case the problem is subdivided into two smaller problems of size $k-1$ and $n-k$

$$\Longrightarrow \quad T(n) = \frac{1}{n} \sum_{k=1}^{n} (T(k-1) + T(n-k)) + \Theta(n)$$

Step (1) + (2)
Since for each $k \in \{1, \ldots, n\}$ we have $k - 1 < n$ and $n - k < n$ we can apply the induction hypothesis:

Part 2: Quicksort

```
Quicksort(A, p, r) //A[1]=first entry of A
    IF (p < r) THEN
        q = Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

$A[p..q-1]$ = **low side**
$A[q+1..r]$ = **high side**

```
Partition(A, p, r)
    x = A[r] //pivot (other ways to pick x are possible!)
    i = p − 1 //i is highest index of low side
    FOR j = p TO r − 1 DO
    //process each element other than pivot
        IF (A[j] ≤ x) THEN
        //does this element belong on the low side?
            i := i + 1 //index of a new slot in the low side
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    //pivot goes just to the right of the low side
    RETURN i + 1
```

Thm. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the $n!$ possible permutations of the entries in A are equiprobable), then the average time of* `Quicksort`$(A, 1, n)$ *is in $O(n \log n)$.*

Proof. W.l.o.g. $n = A.length$ and $A[i] \in \{1, \ldots, n\}$. Recall that $q = \text{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Step (1)
By induction, let us assume that $T(i) \in O(i \log i)$ for every $i \in \{1, \ldots, N\}$ for some $N \geq 2$ (base case $N = 1, 2$ check by yourself).

Choose $n = N + 1$. We want to show that $T(n) \in O(n \log n)$.

By induction hypothesis: $T(i) \leq c' i \log i \leq c' n \log n$ for all $1 \leq i < n$ and large enough c'

Step (2)
By Thm.-assumption, each $k \in \{1, \ldots, n\}$ is equally likely on some position of A.

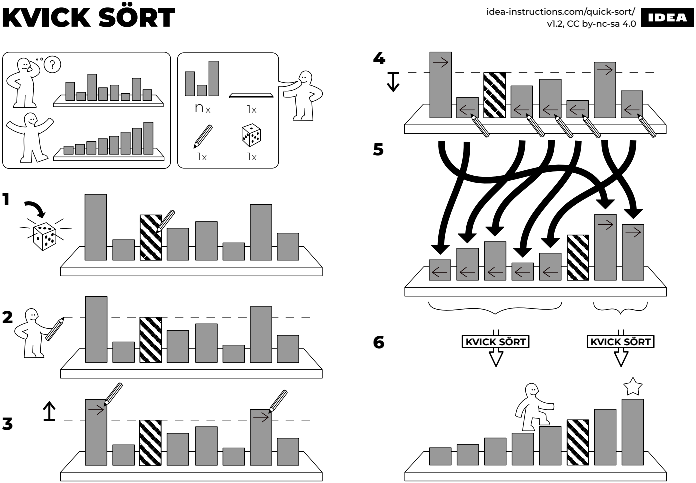
$\implies$ with probability $\frac{1}{n}$ we have $A[n] = k$ and thus, k is a pivot in which case the problem is subdivided into two smaller problems of size $k - 1$ and $n - k$

$$\implies T(n) = \frac{1}{n} \sum_{k=1}^{n} (T(k-1) + T(n-k)) + \Theta(n)$$

Step (1) + (2)
Since for each $k \in \{1, \ldots, n\}$ we have $k - 1 < n$ and $n - k < n$ we can apply the induction hypothesis:

$$T(n) \leq \frac{1}{n} \sum_{k=1}^{n} (c' n \log n + c' n \log n) + \Theta(n) \leq \frac{1}{n} n((cn \log n + cn \log n)) + cn \text{ for large enough } c$$

$\implies T(n) \in O(n \log n)$ (easy exercise). $\square$



Here, the pivot is randomly chosen!

Part 2: lower bound for "comparison sort"

Part 2: Lower bound for runtime of "comparison sort" algorithms

We have now seen a handful of algorithms that can sort n numbers in $O(n \log n)$ time.

Whereas merge sort and heapsort achieve this upper bound in the worst case, quicksort achieves it on average.

Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \log n)$ time [exercise]

These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts**. Thus, all the sorting algorithms introduced so far are comparison sorts.

In what follows, we show that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Part 2: Lower bound for runtime of "comparison sort" algorithms

We have now seen a handful of algorithms that can sort n numbers in $O(n \log n)$ time.

Whereas merge sort and heapsort achieve this upper bound in the worst case, quicksort achieves it on average.

Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \log n)$ time [exercise]

These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts**. Thus, all the sorting algorithms introduced so far are comparison sorts.

In what follows, we show that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Part 2: Lower bound for runtime of "comparison sort" algorithms

We have now seen a handful of algorithms that can sort n numbers in $O(n \log n)$ time.

Whereas merge sort and heapsort achieve this upper bound in the worst case, quicksort achieves it on average.

Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \log n)$ time [exercise]

These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts**. Thus, all the sorting algorithms introduced so far are comparison sorts.

In what follows, we show that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Part 2: Lower bound for runtime of "comparison sort" algorithms

We have now seen a handful of algorithms that can sort n numbers in $O(n \log n)$ time.

Whereas merge sort and heapsort achieve this upper bound in the worst case, quicksort achieves it on average.

Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \log n)$ time [exercise]

These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts**. Thus, all the sorting algorithms introduced so far are comparison sorts.

In what follows, we show that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Part 2: Lower bound for runtime of "comparison sort" algorithms

We have now seen a handful of algorithms that can sort n numbers in $O(n \log n)$ time.

Whereas merge sort and heapsort achieve this upper bound in the worst case, quicksort achieves it on average.

Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \log n)$ time [exercise]

These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts**. Thus, all the sorting algorithms introduced so far are comparison sorts.

In what follows, we show that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Part 2: Lower bound for runtime of "comparison sort" algorithms

For the worst case, we can assume that all elements in $A[1..n]$ are distinct.

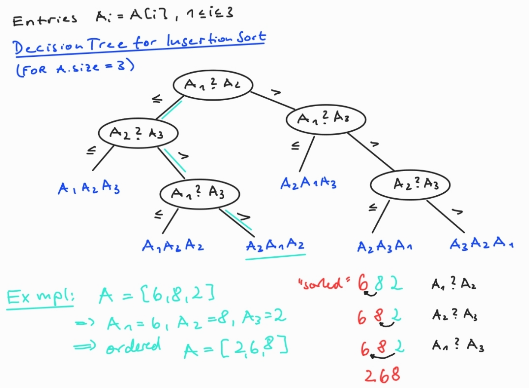
We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct.

Since a binary tree of height h has no more than 2^h leaves, we have $2^h \geq n! \iff h \geq \log(n!) = \Omega(n \log n)$

So, in worst-case at least $\Omega(n \log n)$ comparisons must be performed in any comparison sort to sort n elements.

Part 2: Lower bound for runtime of "comparison sort" algorithms



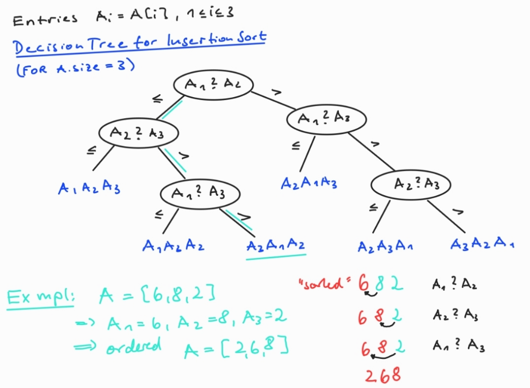
For the worst case, we can assume that all elements in $A[1..n]$ are distinct.

We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct.

Since a binary tree of height h has no more than 2^h leaves, we have $2^h \ge n! \iff h \ge \log(n!) = \Omega(n \log n)$

So, in worst-case at least $\Omega(n \log n)$ comparisons must be performed in any comparison sort to sort n elements.



For the worst case, we can assume that all elements in $A[1..n]$ are distinct.

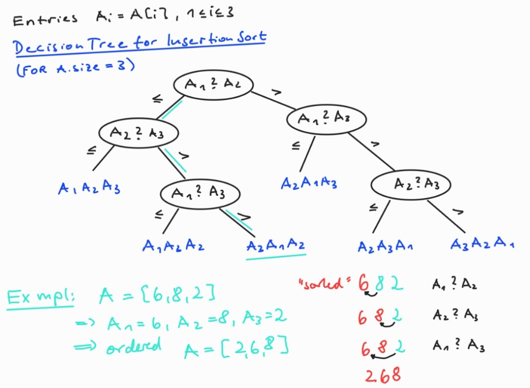
We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct.

Since a binary tree of height h has no more than 2^h leaves, we have $2^h \geq n! \iff h \geq \log(n!) = \Omega(n \log n)$

So, in worst-case at least $\Omega(n \log n)$ comparisons must be performed in any comparison sort to sort n elements.

Part 2: Lower bound for runtime of "comparison sort" algorithms



For the worst case, we can assume that all elements in $A[1..n]$ are distinct.

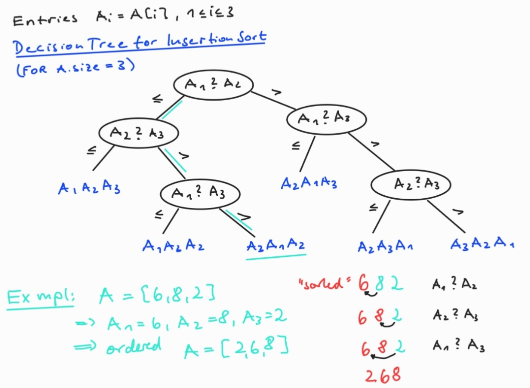
We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct.

Since a binary tree of height h has no more than 2^h leaves, we have $2^h \ge n! \iff h \ge \log(n!) = \Omega(n \log n)$

So, in worst-case at least $\Omega(n \log n)$ comparisons must be performed in any comparison sort to sort n elements.

Part 2: Lower bound for runtime of "comparison sort" algorithms



$\log(n!) = \log(1 \cdot 2 \cdot \ldots \cdot n)$ implies together with log-rules:

For the worst case, we can assume that all elements in $A[1..n]$ are distinct.

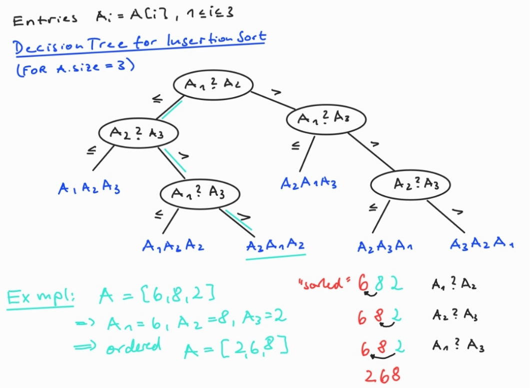
We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct.

Since a binary tree of height h has no more than 2^h leaves, we have $2^h \geq n! \iff h \geq \log(n!) = \Omega(n \log n)$

So, in worst-case at least $\Omega(n \log n)$ comparisons must be performed in any comparison sort to sort n elements.

Part 2: Lower bound for runtime of "comparison sort" algorithms



$\log(n!) = \log(1 \cdot 2 \cdot \ldots \cdot n)$ implies together with log-rules:

$\log(n!) = \sum_{i=1}^{n} \log(i) \le \sum_{i=1}^{n} \log(n) = n \log(n) \in O(n \log(n))$

For the worst case, we can assume that all elements in $A[1..n]$ are distinct.

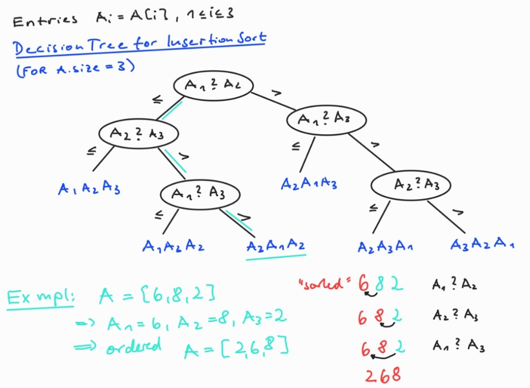
We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct.

Since a binary tree of height h has no more than 2^h leaves, we have $2^h \ge n! \iff h \ge \log(n!) = \Omega(n \log n)$

So, in worst-case at least $\Omega(n \log n)$ comparisons must be performed in any comparison sort to sort n elements.

Part 2: Lower bound for runtime of "comparison sort" algorithms



$\log(n!) = \log(1 \cdot 2 \cdot \ldots \cdot n)$ implies together with log-rules:

$$\log(n!) = \sum_{i=1}^{n} \log(i) \le \sum_{i=1}^{n} \log(n) = n \log(n) \in O(n \log(n))$$

$$\log(n!) = \sum_{i=1}^{n} \log(i) \ge \sum_{i=n/2}^{n} \log(i) =$$
$$= \log(n/2) + \log(n/2 + 1) + \cdots + \log(n-1) + \log(n)$$
$$\ge \log(n/2) + \ldots + \log(n/2) = n/2 \cdot \log(n/2) \in \Omega(n \log(n))$$

For the worst case, we can assume that all elements in $A[1..n]$ are distinct.

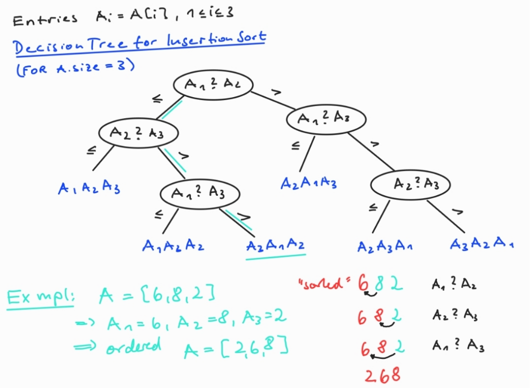
We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct.

Since a binary tree of height h has no more than 2^h leaves, we have $2^h \ge n! \iff h \ge \log(n!) = \Omega(n \log n)$

So, in worst-case at least $\Omega(n \log n)$ comparisons must be performed in any comparison sort to sort n elements.

Part 2: Lower bound for runtime of "comparison sort" algorithms



$\log(n!) = \log(1 \cdot 2 \cdot \ldots \cdot n)$ implies together with log-rules:

$$\log(n!) = \sum_{i=1}^{n} \log(i) \leq \sum_{i=1}^{n} \log(n) = n\log(n) \in O(n\log(n))$$

$$\log(n!) = \sum_{i=1}^{n} \log(i) \geq \sum_{i=n/2}^{n} \log(i) =$$
$$= \log(n/2) + \log(n/2+1) + \cdots + \log(n-1) + \log(n)$$
$$\geq \log(n/2) + \ldots + \log(n/2) = n/2 \cdot \log(n/2) \in \Omega(n\log(n))$$

For the worst case, we can assume that all elements in $A[1..n]$ are distinct.

We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct.

Since a binary tree of height h has no more than 2^h leaves, we have $2^h \geq n! \iff h \geq \log(n!) = \Omega(n\log n)$

So, in worst-case at least $\Omega(n\log n)$ comparisons must be performed in any comparison sort to sort n elements.

Part 2: Counting sort

Part 2: Counting sort

How to sort an array, if not not by comparing the elements ??

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k and runs in $\Theta(n+k)$ time.

Hence, if $k = O(n)$, the counting sort runs in $\Theta(n)$ time.

Counting sort first determines, for each input element x, the number of elements less than or equal to x.

It then uses this information to place element x directly into its position in the sorted output array.

Example: $A = [2, 6, 5, 0, 1] \implies$ 4 elements are less than or equal to $x = 5$
$\implies$ in sorted array, $x = 5$ must be placed in position 4, i.e., $A[4] = 5$ must hold.

We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want them all to end up in the same position.

Part 2: Counting sort

How to sort an array, if not not by comparing the elements ??

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k and runs in $\Theta(n+k)$ time.

Hence, if $k = O(n)$, the counting sort runs in $\Theta(n)$ time.

Counting sort first determines, for each input element x, the number of elements less than or equal to x.

It then uses this information to place element x directly into its position in the sorted output array.

Example: $A = [2, 6, 5, 0, 1] \implies$ 4 elements are less than or equal to $x = 5$
$\implies$ in sorted array, $x = 5$ must be placed in position 4, i.e., $A[4] = 5$ must hold.

We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want them all to end up in the same position.

Part 2: Counting sort

How to sort an array, if not not by comparing the elements ??

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k and runs in $\Theta(n + k)$ time.

Hence, if $k = O(n)$, the counting sort runs in $\Theta(n)$ time.

Counting sort first determines, for each input element x, the number of elements less than or equal to x.

It then uses this information to place element x directly into its position in the sorted output array.

Example: $A = [2, 6, 5, 0, 1]$ $\Longrightarrow$ 4 elements are less than or equal to $x = 5$
$\Longrightarrow$ in sorted array, $x = 5$ must be placed in position 4, i.e., $A[4] = 5$ must hold.

We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want them all to end up in the same position.

Part 2: Counting sort

How to sort an array, if not not by comparing the elements ??

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k and runs in $\Theta(n+k)$ time.

Hence, if $k = O(n)$, the counting sort runs in $\Theta(n)$ time.

Counting sort first determines, for each input element x, the number of elements less than or equal to x.

It then uses this information to place element x directly into its position in the sorted output array.

Example: $A = [2, 6, 5, 0, 1] \implies$ 4 elements are less than or equal to $x = 5$
$\implies$ in sorted array, $x = 5$ must be placed in position 4, i.e., $A[4] = 5$ must hold.

We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want them all to end up in the same position.

Part 2: Counting sort

How to sort an array, if not not by comparing the elements ??

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k and runs in $\Theta(n + k)$ time.

Hence, if $k = O(n)$, the counting sort runs in $\Theta(n)$ time.

Counting sort first determines, for each input element x, the number of elements less than or equal to x.

It then uses this information to place element x directly into its position in the sorted output array.

Example: $A = [2, 6, 5, 0, 1] \implies$ 4 elements are less than or equal to $x = 5$
$\implies$ in sorted array, $x = 5$ must be placed in position 4, i.e., $A[4] = 5$ must hold.

We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want them all to end up in the same position.

Part 2: Counting sort

```
COUNTING-SORT(A, n, k)
let B[1..n] and C[0..k] be new arrays
FOR (i = 0 to k) DO C[i] = 0
FOR (j = 1 to n) DO C[A[j]] = C[A[j]] + 1 //C[i] now contains the nr of elements equal to i
FOR (i = 1 to k) DO C[i] = C[i] + C[i − 1] //C[i] now contains the nr of elements ≤ i
FOR (j = n to 1) DO
    B[C[A[j]]] = A[j] //Copy A to B, starting from the end of A.
    C[A[j]] = C[A[j]] − 1 //to handle duplicate values
RETURN B
```

Part 2: Counting sort

```
COUNTING-SORT(A, n, k)
1 let B[1..n] and C[0..k] be new arrays
2 FOR (i = 0 to k) DO C[i] = 0
3 FOR (j = 1 to n) DO C[A[j]] = C[A[j]] + 1 //C[i] now contains the nr of elements equal to i
4 FOR (i = 1 to k) DO C[i] = C[i] + C[i − 1] //C[i] now contains the nr of elements ≤ i
5 FOR (j = n to 1) DO
6       B[C[A[j]]] = A[j] //Copy A to B, starting from the end of A.
7       C[A[j]] = C[A[j]] − 1 //to handle duplicate values
8 RETURN B
```

Example Board

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

after Line 3

	0	1	2	3	4	5
C	2	2	4	7	7	8

after Line 4

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

after Line 5-7 with j=8

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

after Line 5-7 with j=7

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

after Line 5-7 with j=6

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

final sorted array B

Part 2: Counting sort

```
COUNTING-SORT(A, n, k)
1 let B[1..n] and C[0..k] be new arrays
2 FOR (i = 0 to k) DO C[i] = 0
3 FOR (j = 1 to n) DO C[A[j]] = C[A[j]] + 1 //C[i] now contains the nr of elements equal to i
4 FOR (i = 1 to k) DO C[i] = C[i] + C[i − 1] //C[i] now contains the nr of elements ≤ i
5 FOR (j = n to 1) DO
6     B[C[A[j]]] = A[j] //Copy A to B, starting from the end of A.
7     C[A[j]] = C[A[j]] − 1 //to handle duplicate values
8 RETURN B
```

Theorem. COUNTING-SORT(A, n, k) *correctly sorts the elements of A into B in* $\Theta(n + k)$ *time.*

Proof: correctness [Exercise].

runtime:

Line 1 takes $\Theta(n + k)$ time

FOR-loops of line 2 and 4 both take $\Theta(k)$ time

FOR-loops of line 3 and 5-7 both takes $\Theta(n)$ time

$\implies$ overall time is $\Theta(k + n)$ $\square$

Part 2: Counting sort

COUNTING-SORT(A, n, k)

1 let $B[1..n]$ and $C[0..k]$ be new arrays
2 FOR ($i = 0$ to k) DO $C[i] = 0$
3 FOR ($j = 1$ to n) DO $C[A[j]] = C[A[j]] + 1$ //$C[i]$ now contains the nr of elements equal to i
4 FOR ($i = 1$ to k) DO $C[i] = C[i] + C[i-1]$ //$C[i]$ now contains the nr of elements $\leq i$
5 FOR ($j = n$ to 1) DO
6     $B[C[A[j]]] = A[j]$ //Copy A to B, starting from the end of A.
7     $C[A[j]] = C[A[j]] - 1$ //to handle duplicate values
8 RETURN B

Theorem. COUNTING-SORT(A, n, k) *correctly sorts the elements of A into B in* $\Theta(n + k)$ *time.*

Proof: correctness [Exercise].
runtime:
Line 1 takes $\Theta(n + k)$ time
FOR-loops of line 2 and 4 both take $\Theta(k)$ time
FOR-loops of line 3 and 5-7 both takes $\Theta(n)$ time
$\implies$ overall time is $\Theta(k + n)$ □

In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.
Thus, counting sort can beat the lower bound of $\Omega(n \log n)$ because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code.

Part 2: Counting sort

COUNTING-SORT(A, n, k)

1 let $B[1..n]$ and $C[0..k]$ be new arrays
2 FOR ($i = 0$ to k) DO $C[i] = 0$
3 FOR ($j = 1$ to n) DO $C[A[j]] = C[A[j]] + 1$ //$C[i]$ now contains the nr of elements equal to i
4 FOR ($i = 1$ to k) DO $C[i] = C[i] + C[i-1]$ //$C[i]$ now contains the nr of elements $\leq i$
5 FOR ($j = n$ to 1) DO
6     $B[C[A[j]]] = A[j]$ //Copy A to B, starting from the end of A.
7     $C[A[j]] = C[A[j]] - 1$ //to handle duplicate values
8 RETURN B

Theorem. COUNTING-SORT(A, n, k) *correctly sorts the elements of A into B in* $\Theta(n + k)$ *time.*

Proof: correctness [Exercise].
runtime:
Line 1 takes $\Theta(n + k)$ time
FOR-loops of line 2 and 4 both take $\Theta(k)$ time
FOR-loops of line 3 and 5-7 both takes $\Theta(n)$ time
$\Longrightarrow$ overall time is $\Theta(k + n)$ $\square$

In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.
Thus, counting sort can beat the lower bound of $\Omega(n \log n)$ because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code.

There are many further algorithms that can beat this lower when additional information about the data to be sorted is available. E.g. bucket sorts assumes that the input is drawn from a uniform distribution and has an average-case running time of $O(n)$.

Part 2: Summary

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

We have seen now several sorting algorithms (assuming n numbers need to be sorted):

- `Insertion Sort`: in place, runtime $O(n^2)$
- `Merge Sort`: not in place, runtime $\Theta(n \log n)$
- `Heapsort`: in place, runtime $\Theta(n \log n)$
- `Quicksort`: in place, worst-case runtime $\Theta(n^2)$. However, expected runtime is $\Theta(n \log n)$ and in practice it outperforms heapsort

The latter algorithms are all *comparison sorts*: they determine the sorted order of an input array by comparing elements.

We provided a lower bound of $\Omega(n \log n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

This lower bound can be improved if one adds additional requirements on the input data and thus, if one can gather information about the sorted order of the input by means other than comparing elements. As an example, we considered

- `Counting Sort`: not in place, runtime $\Theta(n + k)$ in case the n numbers to be sorted are in $\{1, \ldots, k\}$
  $\implies$ runtime $\Theta(n)$ if $k = O(n)$.

Part 2: Summary

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

We have seen now several sorting algorithms (assuming n numbers need to be sorted):

- `Insertion Sort`: in place, runtime $O(n^2)$
- `Merge Sort`: not in place, runtime $\Theta(n \log n)$
- `Heapsort`: in place, runtime $\Theta(n \log n)$
- `Quicksort`: in place, worst-case runtime $\Theta(n^2)$. However, expected runtime is $\Theta(n \log n)$ and in practice it outperforms heapsort

The latter algorithms are all *comparison sorts*: they determine the sorted order of an input array by comparing elements.

We provided a lower bound of $\Omega(n \log n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

This lower bound can be improved if one adds additional requirements on the input data and thus, if one can gather information about the sorted order of the input by means other than comparing elements. As an example, we considered

- `Counting Sort`: not in place, runtime $\Theta(n + k)$ in case the n numbers to be sorted are in $\{1, \ldots, k\}$
  $\implies$ runtime $\Theta(n)$ if $k = O(n)$.

Part 2: Summary

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

We have seen now several sorting algorithms (assuming n numbers need to be sorted):

- `Insertion Sort`: in place, runtime $O(n^2)$
- `Merge Sort`: not in place, runtime $\Theta(n \log n)$
- `Heapsort`: in place, runtime $\Theta(n \log n)$
- `Quicksort`: in place, worst-case runtime $\Theta(n^2)$. However, expected runtime is $\Theta(n \log n)$ and in practice it outperforms heapsort

The latter algorithms are all *comparison sorts*: they determine the sorted order of an input array by comparing elements.

We provided a lower bound of $\Omega(n \log n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

This lower bound can be improved if one adds additional requirements on the input data and thus, if one can gather information about the sorted order of the input by means other than comparing elements. As an example, we considered

`Counting Sort`: not in place, runtime $\Theta(n + k)$ in case the n numbers to be sorted are in $\{1, \ldots, k\}$
$\Longrightarrow$ runtime $\Theta(n)$ if $k = O(n)$.