Sorting

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

Using MERGE-like algorithms

Consider the following variant of MERGE.

Algorithm MERGE(L_1, L_2):

Input: L_1 and L_2 are ordered lists of distinct values.

output := $\emptyset$.
$i_1, i_2 := 0, 0$.
while $i_1 < |L_1|$ **or** $i_2 < |L_2|$ **do**
  if $(i_1 < |L_1|$ **and** $i_2 < |L_2|)$ **and also** $L_1[i_1] = L_2[i_2]$ **then**
    Add $L_1[i_1]$ to *output*.
    $i_1, i_2 := i_1 + 1, i_2 + 1$.
  else if $i_2 = |L_2|$ **or else** $(i_1 < |L_1|$ **and also** $L_1[i_1] < L_2[i_2])$ **then**
    Add $L_1[i_1]$ to *output*.
    $i_1 := i_1 + 1$.
  else $L_1[i_1] > L_2[i_2]$
    Add $L_2[i_2]$ to *output*.
    $i_2 := i_2 + 1$.
return *output*. /* return $L_1 \cup L_2$. */

Using Merge-like algorithms

Consider the following variant of Merge.

Algorithm Merge(L_1, L_2):

Input: L_1 and L_2 are ordered lists of distinct values.

output := $\emptyset$.
$i_1, i_2 := 0, 0$.
while $i_1 < |L_1|$ **or** $i_2 < |L_2|$ **do**
  if ($i_1 < |L_1|$ **and** $i_2 < |L_2|$) **and also** $L_1[i_1] = L_2[i_2]$ **then**
    Add $L_1[i_1]$ to *output*.
    $i_1, i_2 := i_1 + 1, i_2 + 1$.
  else if $i_2 = |L_2|$ **or else** ($i_1 < |L_1|$ **and also** $L_1[i_1] < L_2[i_2]$) **then**
    ~~Add $L_1[i_1]$ to *output*~~.
    $i_1 := i_1 + 1$.
  else $L_1[i_1] > L_2[i_2]$
    ~~Add $L_2[i_2]$ to *output*~~.
    $i_2 := i_2 + 1$.
return *output*. /* return $L_1 \cap L_2$. */

Using MERGE-like algorithms

Consider the following variant of MERGE.

Algorithm MERGE(L_1, L_2):

Input: L_1 and L_2 are ordered lists of distinct values.

output := $\emptyset$.
$i_1, i_2 := 0, 0$.
while $i_1 < |L_1|$ **or** $i_2 < |L_2|$ **do**
  if $(i_1 < |L_1|$ **and** $i_2 < |L_2|)$ **and also** $L_1[i_1] = L_2[i_2]$ **then**
    ~~Add $L_1[i_1]$ to *output*~~.
    $i_1, i_2 := i_1 + 1, i_2 + 1$.
  else if $i_2 = |L_2|$ **or else** $(i_1 < |L_1|$ **and also** $L_1[i_1] < L_2[i_2])$ **then**
    Add $L_1[i_1]$ to *output*.
    $i_1 := i_1 + 1$.
  else $L_1[i_1] > L_2[i_2]$
    ~~Add $L_2[i_2]$ to *output*~~.
    $i_2 := i_2 + 1$.
return *output*. /* return $L_1 \setminus L_2$. */

Using MERGE-like algorithms

Consider the following variant of MERGE.

Algorithm MERGE(L_1, L_2):

Input: L_1 and L_2 are ordered lists of distinct values.

output := $\emptyset$.
$i_1, i_2 := 0, 0$.
while $i_1 < |L_1|$ **or** $i_2 < |L_2|$ **do**
  if $(i_1 < |L_1|$ **and** $i_2 < |L_2|)$ **and also** $L_1[i_1] = L_2[i_2]$ **then**
    ~~Add $L_1[i_1]$ to *output*~~.
    $i_1, i_2 := i_1 + 1, i_2 + 1$.
  else if $i_2 = |L_2|$ **or else** $(i_1 < |L_1|$ **and also** $L_1[i_1] < L_2[i_2])$ **then**
    ~~Add $L_1[i_1]$ to *output*~~.
    $i_1 := i_1 + 1$.
  else $L_1[i_1] > L_2[i_2]$
    Add $L_2[i_2]$ to *output*.
    $i_2 := i_2 + 1$.
return *output*. /* return $L_2 \setminus L_1$. */

Using MERGE-like algorithms

Consider the following variant of MERGE.

Algorithm MERGE(L_1, L_2):

Input: L_1 and L_2 are ordered lists of distinct values.

output := $\emptyset$.
$i_1, i_2 := 0, 0$.
while $i_1 < |L_1|$ **or** $i_2 < |L_2|$ **do**
  if $(i_1 < |L_1|$ **and** $i_2 < |L_2|)$ **and also** $L_1[i_1] = L_2[i_2]$ **then**
    ~~Add $L_1[i_1]$ to *output*~~.
    $i_1, i_2 := i_1 + 1, i_2 + 1$.
  else if $i_2 = |L_2|$ **or else** $(i_1 < |L_1|$ **and also** $L_1[i_1] < L_2[i_2])$ **then**
    Add $L_1[i_1]$ to *output*.
    $i_1 := i_1 + 1$.
  else $L_1[i_1] > L_2[i_2]$
    Add $L_2[i_2]$ to *output*.
    $i_2 := i_2 + 1$.
return *output*. /* return $(L_1 \cup L_2) \setminus (L_1 \cap L_2)$. */

Using Merge-like algorithms

Consider relations enrolled(c, *student*) and teaches(c, *faculty*), *ordered* on course *course*.

Problem

Compute all pairs (*student*, *faculty*) such that *faculty* is a teacher of *student*.

Solutions

- A nested-loop join: $\Theta(|\text{enrolled}| \cdot |\text{teaches}|)$.
- Using binary search: $\Theta(|\text{enrolled}| \cdot \log_2(|\text{teaches}|) + |\mathit{result}|)$.

Can we do better?

Using MERGE-like algorithms

Consider relations enrolled(c, *student*) and teaches(c, *faculty*), *ordered* on course *course*.

Algorithm ETMERGEJOIN(*enrolled*, *teaches*):

output $:= \emptyset$.
$i_1, i_2 := 0, 0$.
while $i_1 < |enrolled|$ **and** $i_2 < |teaches|$ **do**
    if enrolled$[i_1].c$ = teaches$[i_2].c$ **then**
        A potential join output!
        Need to find all enrolled students for course enrolled$[i_1].c$.
        Need to find all teaching faculty for course teaches$[i_2].c$.

    else if enrolled$[i_1].c$ < teaches$[i_2].c$ **then**
        $i_1 := i_1 + 1$.
    else enrolled$[i_1].c$ < teaches$[i_2].c$
        $i_2 := i_2 + 1$.
return *output*. /* return pairs (s, f) such that f is a teacher of s. */

Using MERGE-like algorithms

Consider relations enrolled(c, *student*) and teaches(c, *faculty*), *ordered* on course *course*.

Algorithm ETMERGEJOIN(*enrolled*, *teaches*):

output := $\emptyset$.
$i_1, i_2 := 0, 0$.
while $i_1 < |enrolled|$ **and** $i_2 < |teaches|$ **do**
    if enrolled[i_1].c = teaches[i_2].c **then**
        j_1 := first j with either j = |enrolled| or else enrolled[j].$c \neq$ enrolled[i_1].c.
        j_2 := first j with either j = |teaches| or else teaches[j].$c \neq$ teaches[i_2].c.
        Add all (s, f) with $(c_1, s) \in$ enrolled[i_1, j_1) and $(c_2, f) \in$ teaches[i_2, j_2) to *output*.
        $i_1, i_2 := j_1, j_2$.
    else if enrolled[i_1].c < teaches[i_2].c **then**
        $i_1 := i_1 + 1$.
    else enrolled[i_1].c < teaches[i_2].c
        $i_2 := i_2 + 1$.
return *output*. /* return pairs (s, f) such that f is a teacher of s. */

Using Merge-like algorithms

Consider relations enrolled(c, *student*) and teaches(c, *faculty*), *ordered* on course *course*.

Algorithm ETMergeJoin(*enrolled*, *teaches*):

```
output := ∅.
i1, i2 := 0, 0.
while i1 < |enrolled| and i2 < |teaches| do
    if enrolled[i1].c = teaches[i2].c then
        j1 := first j with either j = |enrolled| or else enrolled[j].c ≠ enrolled[i1].c.
        j2 := first j with either j = |teaches| or else teaches[j].c ≠ teaches[i2].c.
        Add all (s, f) with (c1, s) ∈ enrolled[i1, j1) and (c2, f) ∈ teaches[i2, j2) to output.
        i1, i2 := j1, j2.
```

Complexity

- The *merge*-part visits every value in enrolled and teaches once.
- The *join*-part only visits those pairs of values necessary for the result.

Hence, the complexity is $\Theta(|enrolled| + |teaches| + |result|)$.

Using Merge-like algorithms

Consider relations enrolled(c, *student*) and teaches(c, *faculty*), *ordered* on course *course*.

Problem

Compute all pairs (*student*, *faculty*) such that *faculty* is a teacher of *student*.

Solutions

- A nested-loop join: $\Theta(|\text{enrolled}| \cdot |\text{teaches}|)$.
- Using binary search: $\Theta(|\text{enrolled}| \cdot \log_2(|\text{teaches}|) + |\mathit{result}|)$.
- Using merge join: $\Theta(|\mathit{enrolled}| + |\mathit{teaches}| + |\mathit{result}|)$.

Stable sorting

Consider a list *enrolled* of enrollment data with schema

$$\text{enrolled}(\mathit{dept}, \mathit{code}, \mathit{sid}, \mathit{date}).$$

If we add enrollment data to *the end of the list*, then enrolled is always sorted on *date*.

Problem

Group enrolled on (dept, code) and within each group sort enrollments on *date*.

Stable sorting

Consider a list *enrolled* of enrollment data with schema

$$\text{enrolled}(\mathit{dept}, \mathit{code}, \mathit{sid}, \mathit{date}).$$

If we add enrollment data to *the end of the list*, then enrolled is always sorted on *date*.

Problem

Group enrolled on ($\mathit{dept}, \mathit{code}$) and within each group sort enrollments on *date*.

Brute-force solution: Lexicographical sorting on ($\mathit{dept}, \mathit{code}, \mathit{date}$)

Let $(d_1, c_1, s_1, t_1), (d_2, c_2, s_2, t_2) \in \text{enrolled}$. We use the comparison

$$(d_1, c_1, s_1, t_1) \mathrel{\textit{before}} (d_2, c_2, s_2, t_2) \text{ if } (d_1 < d_2) \lor ((d_1 = d_2) \land (c_1 < c_2)) \lor \\ ((d_1 = d_2) \land (c_1 = c_2) \land (t_1 < t_2)).$$

Stable sorting

Consider a list *enrolled* of enrollment data with schema

$$\text{enrolled}(\mathit{dept}, \mathit{code}, \mathit{sid}, \mathit{date}).$$

If we add enrollment data to *the end of the list*, then enrolled is always sorted on *date*.

Problem

Group enrolled on $(\mathit{dept}, \mathit{code})$ and within each group sort enrollments on *date*.

Brute-force solution: Lexicographical sorting on $(\mathit{dept}, \mathit{code}, \mathit{date})$

Let $(d_1, c_1, s_1, t_1), (d_2, c_2, s_2, t_2) \in \text{enrolled}$. We use the comparison

$$(d_1, c_1, s_1, t_1)\ \mathit{before}\ (d_2, c_2, s_2, t_2) \text{ if } (d_1 < d_2) \lor ((d_1 = d_2) \land (c_1 < c_2)) \lor$$
$$((d_1 = d_2) \land (c_1 = c_2) \land (t_1 < t_2)).$$

Downside: During sorting, we end up throwing away the existing ordering on *date*, and then we rebuild that order from scratch!

Stable sorting

Consider a list *enrolled* of enrollment data with schema

$$\text{enrolled}(\mathit{dept}, \mathit{code}, \mathit{sid}, \mathit{date}).$$

If we add enrollment data to *the end of the list*, then enrolled is always sorted on *date*.

Problem

Group enrolled on (dept, code) and within each group sort enrollments on *date*.

Better solution: Use a *stable sort algorithm*

A *stable sort algorithm* maintains the relative order of "equal values".

Let $(d_1, c_1, s_1, t_1), (d_2, c_2, s_2, t_2) \in$ enrolled. If we sort enrolled using a *stable sort algorithm* using the comparison

$$(d_1, c_1, s_1, t_1) \textit{ before } (d_2, c_2, s_2, t_2) \text{ if } (d_1 < d_2) \lor ((d_1 = d_2) \land (c_1 < c_2))$$

then within each (dept, code)-group, enrollments remain ordered on *date* for free!

Stable sorting

Definition

Let L be a list that is already ordered with respect to some attributes $a_1, \ldots, a_n$.
Consider a sort step S that re-orders L based on other attributes $b_1, \ldots, b_m$.

We say that the sort step S is *stable* if, for every value $r_1 \in L$ and $r_2 \in L$ such that r_1 originally came before r_2 and r_1 and r_2 agreee on attributes $b_1, \ldots, b_m$, the resulting re-ordered list will still have r_1 come before r_2.

Stable sorting

Definition

Let L be a list that is already ordered with respect to some attributes $a_1, \ldots, a_n$.
Consider a sort step S that re-orders L based on other attributes $b_1, \ldots, b_m$.

We say that the sort step S is *stable* if, for every value $r_1 \in L$ and $r_2 \in L$ such that r_1 originally came before r_2 and r_1 and r_2 agreee on attributes $b_1, \ldots, b_m$, the resulting re-ordered list will still have r_1 come before r_2.

Question: Have we already seen stable sort algorithms?

Yes: SelectionSort, InsertionSort, and MergeSort.

Note: even minor changes to these algorithms will make them non-stable!
(e.g., changing $<$ into $\leq$).

Intermezzo: Recurrence trees

In a recurrence tree

- nodes labeled N represent a *function call* with “input size N”;
- the children of a node represent *recursive calls*;
- per node, we can determine *the work* within that call (besides recursion);
- per depth, we can determine the *total work for that depth*;
- by *summing over all depths*: the total complexity.

Intermezzo: Recurrence trees

In a recurrence tree

- nodes labeled N represent a *function call* with "input size N";
- the children of a node represent *recursive calls*;
- per node, we can determine *the work* within that call (besides recursion);
- per depth, we can determine the *total work for that depth*;
- by *summing over all depths*: the total complexity.

We already saw two examples: LowerBoundRec and MergeSortR.

Intermezzo: Recurrence trees

Example: the *Fibonacci numbers*

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.

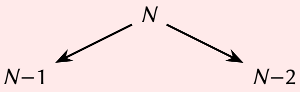
	Number	Cost	Total
N	$1 = 2^0$	1	$1 \cdot 1 = 1$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



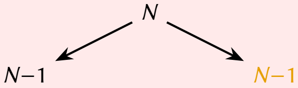
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



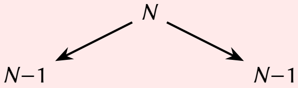
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



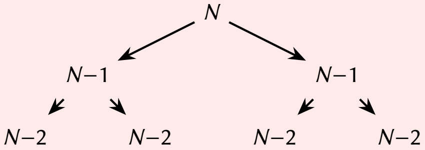
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



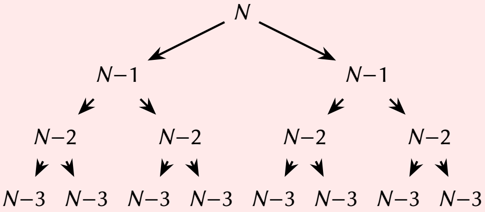
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



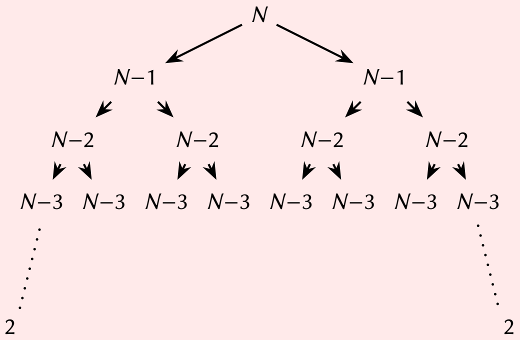
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



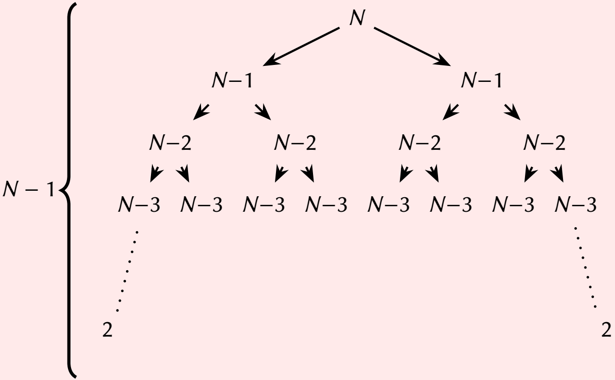
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$
$\vdots$	$\vdots$	$\vdots$
2^i	1	$2^i \cdot 1 = 2^i$
$\vdots$	$\vdots$	$\vdots$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



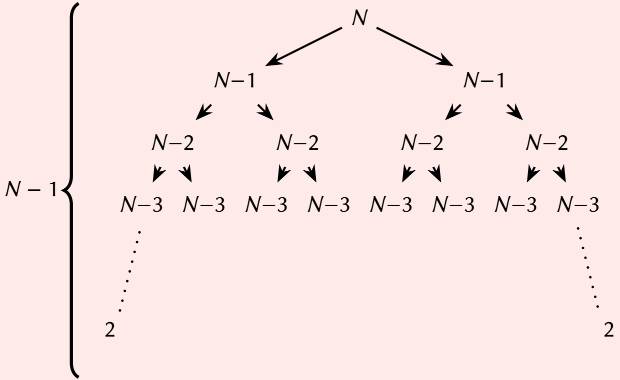
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$
$\vdots$	$\vdots$	$\vdots$
2^i	1	$2^i \cdot 1 = 2^i$
$\vdots$	$\vdots$	$\vdots$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



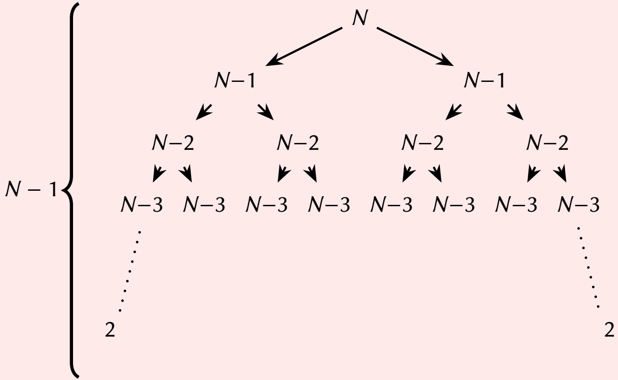
Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$
$\vdots$	$\vdots$	$\vdots$
2^i	1	$2^i \cdot 1 = 2^i$
$\vdots$	$\vdots$	$\vdots$
		+
		$\sum_{i=0}^{N-2} 2^i$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $fib(N) \leq 2^N$

Simplication: $fib(i-2) \leq fib(i-1)$.



Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$
$\vdots$	$\vdots$	$\vdots$
2^i	1	$2^i \cdot 1 = 2^i$
$\vdots$	$\vdots$	$\vdots$
		+

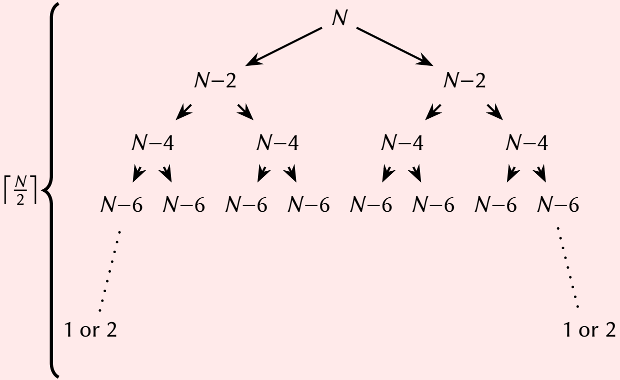
$$\sum_{i=0}^{N-2} 2^i = 2^{N-1} - 1$$

Intermezzo: Recurrence trees

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Prove that $2^{\lceil \frac{N}{2} \rceil} \leq fib(N)$

Simplication: $fib(i-1) \geq fib(i-2)$.



Number	Cost	Total
$1 = 2^0$	1	$1 \cdot 1 = 1$
$2 = 2^1$	1	$2 \cdot 1 = 2$
$4 = 2^2$	1	$4 \cdot 1 = 4$
$8 = 2^3$	1	$8 \cdot 1 = 8$
$\vdots$	$\vdots$	$\vdots$
2^i	1	$2^i \cdot 1 = 2^i$
$\vdots$	$\vdots$	$\vdots$
		+
		$\sum_{i=0}^{\lceil \frac{N}{2} \rceil} 2^i = 2^{\lceil \frac{N}{2} \rceil + 1} - 1$

Intermezzo: Recurrence trees

Example: the *Fibonacci numbers*

$$fib(N) = \begin{cases} 1 & \text{if } N = 1 \text{ or } N = 2; \\ fib(N-1) + fib(N-2) & \text{if } N > 2. \end{cases}$$

Via recurrence trees, we have proven that:

$$2^{\lceil \frac{N}{2} \rceil} \leq fib(N) \leq 2^N.$$

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Someone else has already proved this—so we can reuse the result!

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of LowerBoundRec

$$T(N) = \begin{cases} 4 & \text{if } N = 1; \\ T\left(\frac{N}{2}\right) + 8 & \text{if } N > 1. \end{cases}$$

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of LowerBoundRec

$$T(N) = \begin{cases} 4 & \text{if } N = 1; \\ T\left(\frac{N}{2}\right) + 8 & \text{if } N > 1. \end{cases}$$

We have $a = 1$, $b = 2$, $f(N) = 8 = \Theta(1) = N^{\log_2(1)}$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of LowerBoundRec

$$T(N) = \begin{cases} 4 & \text{if } N = 1; \\ T\left(\frac{N}{2}\right) + 8 & \text{if } N > 1. \end{cases}$$

We have $a = 1$, $b = 2$, $f(N) = 8 = \Theta(1) = N^{\log_2(1)}$.

Case 2 yields: $T(N) = \Theta(N^{\log_2(1)} \log^1(N)) = \log(N)$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + N & \text{if } N > 1. \end{cases}$$

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ T\left(\lfloor \frac{N}{2} \rfloor\right) + T\left(\lceil \frac{N}{2} \rceil\right) + N & \text{if } N > 1. \end{cases}$$

We have $a = 2$, $b = 2$, $f(N) = N = \Theta(N) = N^{\log_2(2)}$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Example: Runtime complexity of MergeSortR

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ T\left(\lfloor \frac{N}{2} \rfloor\right) + T\left(\lceil \frac{N}{2} \rceil\right) + N & \text{if } N > 1. \end{cases}$$

We have $a = 2$, $b = 2$, $f(N) = N = \Theta(N) = N^{\log_2(2)}$.

Case 2 yields: $T(N) = \Theta(N^{\log_2(2)} \log^1(N)) = \Theta(N \log(N))$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A third example

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ 7T\left(\left\lfloor \frac{N}{4} \right\rfloor\right) + N & \text{if } N > 1. \end{cases}$$

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A third example

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ 7T\left(\left\lfloor \frac{N}{4} \right\rfloor\right) + N & \text{if } N > 1. \end{cases}$$

We have $a = 7$, $b = 4$, $f(N) = N = \mathcal{O}N^{\log_4(7)-\epsilon}$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A third example

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ 7T\left(\lfloor \frac{N}{4} \rfloor\right) + N & \text{if } N > 1. \end{cases}$$

We have $a = 7$, $b = 4$, $f(N) = N = \mathcal{O}N^{\log_4(7)-\epsilon}$.

Case 1 yields: $T(N) = \Theta(N^{\log_4(7)}) \approx \Theta(N^{1.40367\ldots})$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A fourth example

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ 2T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + N^3 & \text{if } N > 1. \end{cases}$$

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A fourth example

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ 2T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + N^3 & \text{if } N > 1. \end{cases}$$

We have $a = 2$, $b = 2$, $f(N) = N^3 = \Omega N^{\log_2(2)+\epsilon}$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\lceil \frac{N}{b} \rceil$ or $\lfloor \frac{N}{b} \rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

A fourth example

$$T(N) = \begin{cases} 1 & \text{if } N = 1; \\ 2T\left(\left\lfloor \frac{N}{2} \right\rfloor\right) + N^3 & \text{if } N > 1. \end{cases}$$

We have $a = 2$, $b = 2$, $f(N) = N^3 = \Omega N^{\log_2(2)+\epsilon}$.

Case 3 yields: $T(N) = \Theta(N^3)$.

Intermezzo: The Master Theorem

Let $T(N)$ be a *recurrence* of the form

$$T(N) = \begin{cases} \textit{constant} & \text{if } \textit{base case}; \\ aT\left(\frac{N}{b}\right) + f(N) & \text{if } \textit{recursive case}, \end{cases}$$

with $a \geq 1$, $b > 1$, and we can read $\frac{N}{b}$ also as $\left\lceil \frac{N}{b} \right\rceil$ or $\left\lfloor \frac{N}{b} \right\rfloor$. We have the following

1. if $f(N) = \mathcal{O}(N^{\log_b(a-\epsilon)})$ with $\epsilon > 0$, then $T(N) = \Theta(N^{\log_b(a)})$.
2. if $f(N) = \Theta(N^{\log_b(a)} \log^k(N))$ with $k \geq 0$, then $T(N) = \Theta(N^{\log_b(a)} \log^{k+1}(N))$.
3. if $f(N) = \Omega(N^{\log_b(a+\epsilon)})$ with $\epsilon > 0$ and $af\left(\frac{N}{b}\right) \leq cf(N)$ for a $c < 1$ (for large N), then $T(N) = \Theta(f(N))$.

Feel free to use the Master Theorem, we will provide a copy during the final exam.

Can we do better than MergeSort?

Can we do better than MergeSort?

Algorithm CountSort($L[0 \ldots N]$):

Input: Each value in L is either 0 or 1.

$count_0 := 0$

for all $v \in L$ **do** Count number of 0's

    if $v = 0$ **then**

        $count_0 := count_0 + 1$.

for $i := 0$ to $count_0 - 1$ **do** Write the counted number of 0's

    $L[i] := 0$.

for $i := count_0$ to $N - 1$ **do** Write the remaining 1's

    $L[i] := 1$.

Can we do better than MergeSort?

Algorithm CountSort($L[0 \ldots N]$):

Input: Each value in L is either 0 or 1.

$count_0 := 0$
for all $v \in L$ **do** Count number of 0's
  if $v = 0$ **then**
    $count_0 := count_0 + 1$.
for $i := 0$ to $count_0 - 1$ **do** Write the counted number of 0's
  $L[i] := 0$.
for $i := count_0$ to $N - 1$ **do** Write the remaining 1's
  $L[i] := 1$.

Complexity: Linear ($\Theta(N)$ comparisons, $\Theta(N)$ changes)

Can we do better than MergeSort?

Algorithm CountSort($L[0 \ldots N]$):

Input: Each value in L is either 0 or 1.

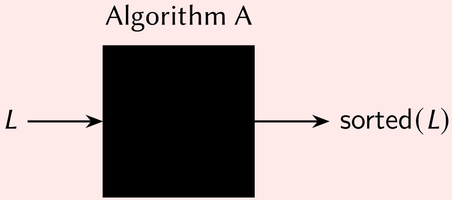
$count_0 := 0$
for all $v \in L$ **do** Count number of 0's
  if $v = 0$ **then**
    $count_0 := count_0 + 1$.
for $i := 0$ to $count_0 - 1$ **do** Write the counted number of 0's
  $L[i] := 0$.
for $i := count_0$ to $N - 1$ **do** Write the remaining 1's
  $L[i] := 1$.

Complexity: Linear ($\Theta(N)$ comparisons, $\Theta(N)$ changes)

CountSort does *not* solve general-purpose sorting!

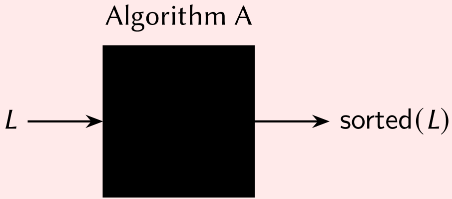
A lower bound for general-purpose sorting

Assume: We have a list $L[0 \ldots N)$ of N distinct values



A lower bound for general-purpose sorting

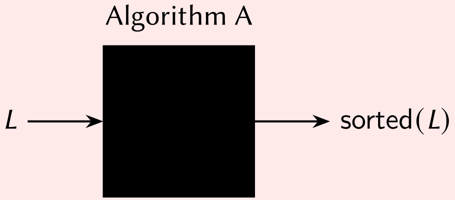
Assume: We have a list $L[0 \ldots N)$ of N distinct values



When is Algorithm A *general-purpose*?

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \ldots N)$ of N distinct values

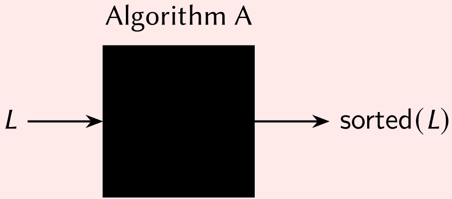


When is Algorithm A *general-purpose*?

- A uses *comparisons* to determine sorted order;
- A does *not require assumptions* on the value distribution in L.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \ldots N)$ of N distinct values

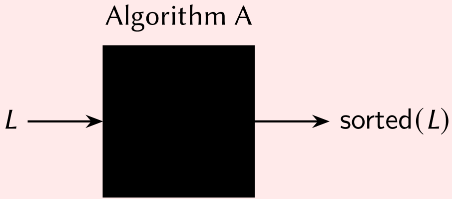


What do we know about *general-purpose* Algorithm A?

Consider lists $L_1 = [1, 3, 2, 4]$ and $L_2 = [1, 2, 3, 4]$.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \dots N)$ of N distinct values



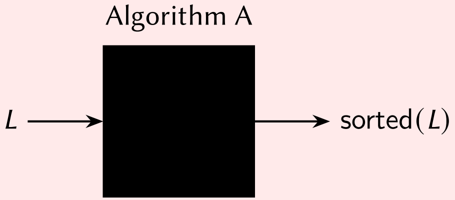
What do we know about *general-purpose* Algorithm A?

Consider lists $L_1 = [1, 3, 2, 4]$ and $L_2 = [1, 2, 3, 4]$.

- Algorithm A must perform *different* operations to order L_1 and L_2.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \ldots N)$ of N distinct values



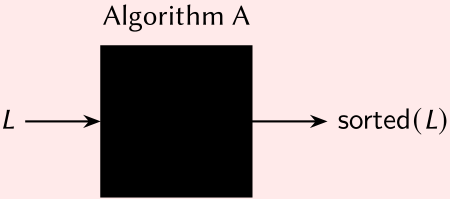
What do we know about *general-purpose* Algorithm A?

Consider lists $L_1 = [1, 3, 2, 4]$ and $L_2 = [1, 2, 3, 4]$.

- Algorithm A must perform *different* operations to order L_1 and L_2.
- Algorithm A uses *comparisons* to decide which operations to perform.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \ldots N)$ of N distinct values



What do we know about *general-purpose* Algorithm A?

Consider lists $L_1 = [1, 3, 2, 4]$ and $L_2 = [1, 2, 3, 4]$.

- Algorithm A must perform *different* operations to order L_1 and L_2.
- Algorithm A uses *comparisons* to decide which operations to perform.

There must be a *distinguishing comparison* after which A behaves *differently*.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

$\mathcal{S}$: All possible lists L that are treated the same by Algorithm A up till this point

C: $L[i] < L[j]$

All lists in $\mathcal{S}$ for which C *did not* hold

All lists in $\mathcal{S}$ for which C *did* hold

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

$\mathcal{S}$: All possible lists L that are
treated the same by Algorithm A up till this point

C: $L[i] < L[j]$

All lists in $\mathcal{S}$ for which C *did not* hold

All lists in $\mathcal{S}$ for which C *did* hold

We can build a comparison tree $\mathcal{T}$ for Algorithm A that starts with all possible L

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

$\mathcal{S}$: All possible lists L that are treated the same by Algorithm A up till this point

C: $L[i] < L[j]$

All lists in $\mathcal{S}$ for which C *did not* hold

All lists in $\mathcal{S}$ for which C *did* hold

We can build a comparison tree $\mathcal{T}$ for Algorithm A that starts with all possible L.

- in $\mathcal{T}$, each leaf of $\mathcal{T}$ must represent *one* list;
- in $\mathcal{T}$, there must be a leaf for *every possible* list L.

A lower bound for general-purpose sorting

We can represent a distinguishing comparison via a comparison tree node

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

$\mathcal{S}$: All possible lists L that are treated the same by Algorithm A up till this point

C: $L[i] < L[j]$

All lists in $\mathcal{S}$ for which C *did not* hold

All lists in $\mathcal{S}$ for which C *did* hold

We can build a comparison tree $\mathcal{T}$ for Algorithm A that starts with all possible L:

- in $\mathcal{T}$, each leaf of $\mathcal{T}$ must represent *one* list;
- in $\mathcal{T}$, there must be a leaf for *every possible* list L.

Otherwise not all distinct lists L are processed in a different way.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

Consider a path π in $\mathcal{T}$ from *root* to a leaf for a specific list L'

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

Consider a path π in $\mathcal{T}$ from *root* to a leaf for a specific list L'

- This path π specifies *all distinguishing comparisons* made by Algorithm A to sort L'.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

Consider a path π in $\mathcal{T}$ from *root* to a leaf for a specific list L'

- This path π specifies *all distinguishing comparisons* made by Algorithm A to sort L'.
- The length of path π is a *lower bound* for the *complexity* to sort L'!

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

Consider a path π in $\mathcal{T}$ from *root* to a leaf for a specific list L'

- This path π specifies *all distinguishing comparisons* made by Algorithm A to sort L'.
- The length of path π is a *lower bound* for the *complexity* to sort L'!

What is the worst-case length of path π?

The lengths of paths in $\mathcal{T}$ depend on the *height of* $\mathcal{T}$,

$\longrightarrow$ which depends on the *number of leaves* in $\mathcal{T}$.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The number of leaves in $\mathcal{T}$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The number of leaves in $\mathcal{T}$

How many distinct lists of length N exist with values $1,\ldots,N$ in an unknown order?

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The number of leaves in $\mathcal{T}$

How many distinct lists of length N exist with values $1,\ldots,N$ in an unknown order?

- N possible values for the first value,
- $N-1$ possible values for the second value,
- $\ldots$
- 1 possible value for the last value.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The number of leaves in $\mathcal{T}$

How many distinct lists of length N exist with values $1,\ldots,N$ in an unknown order?

- N possible values for the first value,
- $N-1$ possible values for the second value,
- …
- 1 possible value for the last value.

$$\prod_{i=1}^{N} i = N! \text{ leaves} \qquad \text{(all possible permutations).}$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

Consider a path π in $\mathcal{T}$ from *root* to a leaf for a specific list L'

- This path π specifies *all distinguishing comparisons* made by Algorithm A to sort L'.
- The length of path π is a *lower bound* for the *complexity* to sort L'!

What is the worst-case length of path π?

The lengths of paths in $\mathcal{T}$ depend on the *height of* $\mathcal{T}$,

$\longrightarrow$ which depends on the *number of leaves* $N!$ in $\mathcal{T}$.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

Consider a node n from which we can reach M leaves.

How do we make the distance from n to all its leaves minimal?

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

Consider a node n from which we can reach M leaves.

How do we make the distance from n to all its leaves minimal?

The left and right child of n each can reach $\frac{M}{2}$ leaves:

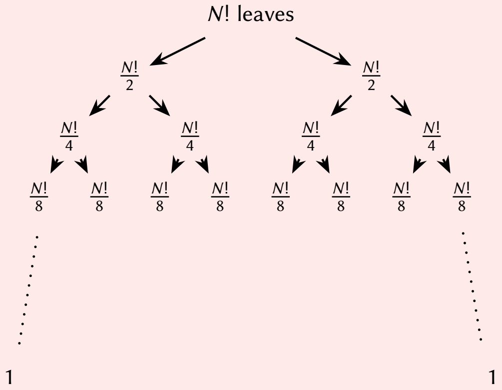
$\rightarrow$ minimize the size of the tree rooted at *both children*.

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

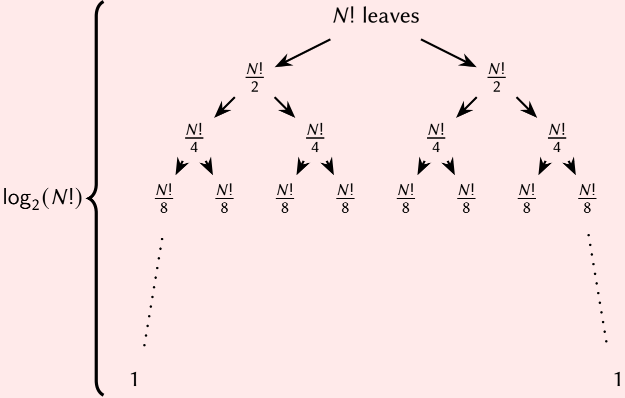


A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves



A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\log_2(N!) = \log_2(N \cdot (N-1) \cdot \cdots \cdot 1)$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\begin{aligned}\log_2(N!) &= \log_2(N\cdot(N-1)\cdot\cdots\cdot 1)\\ &= \log_2(N)+\log_2(N-1)+\cdots+\log_2(1)\end{aligned}$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\begin{aligned}
\log_2(N!) &= \log_2(N \cdot (N-1) \cdot \cdots \cdot 1) \\
&= \log_2(N) + \log_2(N-1) + \cdots + \log_2(1) \\
&\geq \log_2(N) + \log_2(N-1) + \cdots + \log_2(\lceil \tfrac{N}{2} \rceil)
\end{aligned}$$

A lower bound for general-purpose sorting

We can represent a distinguishing comparison via a comparison tree node

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\begin{aligned}
\log_2(N!) &= \log_2(N\cdot(N-1)\cdot\cdots\cdot 1)\\
&= \log_2(N)+\log_2(N-1)+\cdots+\log_2(1)\\
&\geq \log_2(N)+\log_2(N-1)+\cdots+\log_2(\lceil \tfrac{N}{2}\rceil)\\
&\geq \tfrac{N}{2}\log_2(\tfrac{N}{2})
\end{aligned}$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots,N)$ with values $1,\ldots,N$ in an *unknown order*.

The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\begin{aligned}
\log_2(N!) &= \log_2(N\cdot(N-1)\cdot\dots\cdot 1)\\
&= \log_2(N) + \log_2(N-1) + \dots + \log_2(1)\\
&\geq \log_2(N) + \log_2(N-1) + \dots + \log_2(\lceil \tfrac{N}{2} \rceil)\\
&\geq \tfrac{N}{2}\log_2(\tfrac{N}{2})\\
&= \tfrac{N}{2}(\log_2(N) - 1)
\end{aligned}$$

A lower bound for general-purpose sorting

We can represent a *distinguishing comparison* via a *comparison tree node*

Consider sorting lists $L[0\ldots, N)$ with values $1, \ldots, N$ in an *unknown order*.

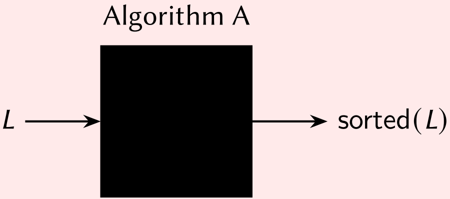
The *minimal* height of a tree $\mathcal{T}$ with $N!$ leaves

We have to find a lower bound on $\log_2(N!)$.

$$\begin{aligned}
\log_2(N!) &= \log_2(N \cdot (N-1) \cdot \cdots \cdot 1) \\
&= \log_2(N) + \log_2(N-1) + \cdots + \log_2(1) \\
&\geq \log_2(N) + \log_2(N-1) + \cdots + \log_2(\lceil \tfrac{N}{2} \rceil) \\
&\geq \tfrac{N}{2} \log_2(\tfrac{N}{2}) \\
&= \tfrac{N}{2}(\log_2(N) - 1) \\
&= \tfrac{N}{2} \log_2(N) - \tfrac{N}{2} = \Theta(N \log_2(N)).
\end{aligned}$$

A lower bound for general-purpose sorting

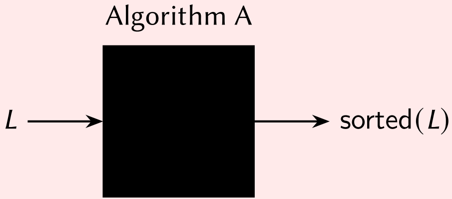
Assume: We have a list $L[0 \ldots N)$ of N distinct values



If Algorithm A is general-purpose, then A will perform *at-least* $\Theta(N \log_2(N))$ *comparisons* for some inputs of N values.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \dots N)$ of N distinct values

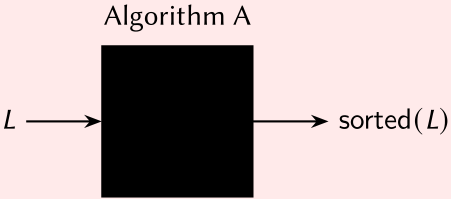


If Algorithm A is general-purpose, then A will perform *at-least* $\Theta(N \log_2(N))$ *comparisons* for some inputs of N values.

If Algorithm A performs less comparisons for *some* inputs, then A will perform more comparisons for *other* inputs.

A lower bound for general-purpose sorting

Assume: We have a list $L[0 \ldots N)$ of N distinct values



General-purpose sorting algorithms such as MergeSort are *optimal*:
their worst-case complexity matches the lower bound of $\Theta(N \log_2(N))$.

A potentially-faster sort: QuickSort

Can we improve upon the *optimal* MergeSort algorithm?

A potentially-faster sort: QuickSort

Can we improve upon the *optimal* MergeSort algorithm?

- Reduce massive $\Theta(N)$ memory consumption?
- Reduce constants: Merge performs many operations on several lists.

A potentially-faster sort: QuickSort

Divide-and-conquer

Divide Turn problem into smaller subproblems.

Conquer Solve the smaller subproblems using *recursion*.

Combine Combine the subproblem solutions into a final solution.

A potentially-faster sort: QuickSort

Divide-and-conquer

Divide Turn problem into smaller subproblems.
Divide the list into small and large values.

Conquer Solve the smaller subproblems using *recursion*.

Combine Combine the subproblem solutions into a final solution.

A potentially-faster sort: QuickSort

Divide-and-conquer

Divide Turn problem into smaller subproblems.
Divide the list into small and large values.

Conquer Solve the smaller subproblems using *recursion*.
Sort the small values and large values separately.

Combine Combine the subproblem solutions into a final solution.

A potentially-faster sort: QuickSort

Divide-and-conquer

Divide Turn problem into smaller subproblems.
Divide the list into small and large values.

Conquer Solve the smaller subproblems using *recursion*.
Sort the small values and large values separately.

Combine Combine the subproblem solutions into a final solution.
The list is sorted if the small values and large values are sorted.

A potentially-faster sort: QuickSort

Divide-and-conquer

Divide Turn problem into smaller subproblems.
Divide the list into small and large values.

Conquer Solve the smaller subproblems using *recursion*.
Sort the small values and large values separately.

Combine Combine the subproblem solutions into a final solution.
The list is sorted if the small values and large values are sorted.

Dividing a list into *small* and *large* values sounds easier than Merge!

QuickSort: High-level overview

Algorithm QuickSort($L[start \ldots end]$):

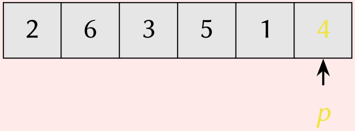
if $end - start > 1$ **then**

2	6	3	5	1	4

QuickSort: High-level overview

Algorithm QuickSort($L[start \ldots end]$):

if $end - start > 1$ **then**
    Choose the position $p \in [start, end)$ of the *pivot value* $v := L[pos]$.



QuickSort: High-level overview

Algorithm QuickSort($L[start \dots end)$):

if $end - start > 1$ **then**

  Choose the position $p \in [start, end)$ of the *pivot value* $v := L[pos]$.

  Partition $L[start \dots end)$ such that

- all values smaller-or-equal to v come first;
- followed by the have the value v;
- followed by all other values (that are larger than v).

3	1	2	4	6	5

QuickSort: High-level overview

Algorithm QuickSort($L[start \ldots end)$):

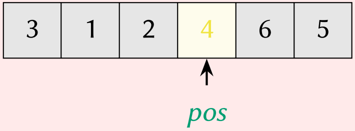
if $end - start > 1$ **then**

  Choose the position $p \in [start, end)$ of the *pivot value* $v := L[pos]$.

  Partition $L[start \ldots end)$ such that

- all values smaller-or-equal to v come first;
- followed by the have the value v;
- followed by all other values (that are larger than v).

  Let pos be the position of v after Partition.



QuickSort: High-level overview

Algorithm QuickSort($L[start \dots end]$):

if $end - start > 1$ **then**

  Choose the position $p \in [start, end)$ of the *pivot value* $v := L[pos]$.

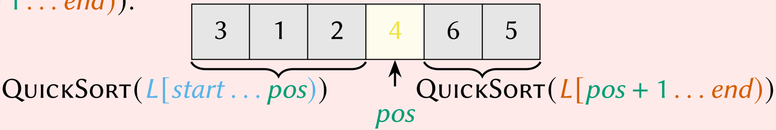
  Partition $L[start \dots end)$ such that

- all values smaller-or-equal to v come first;
- followed by the have the value v;
- followed by all other values (that are larger than v).

  Let pos be the position of v after Partition.

  QuickSort($L[start \dots pos)$).

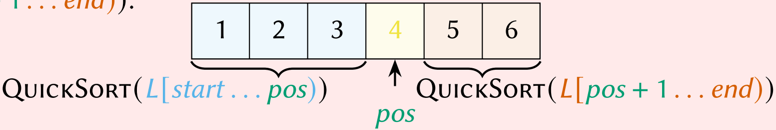
  QuickSort($L[pos + 1 \dots end)$).



QuickSort: High-level overview

Algorithm QuickSort($L[start \dots end)$):

if $end - start > 1$ **then**
- Choose the position $p \in [start, end)$ of the *pivot value* $v := L[pos]$.
- Partition $L[start \dots end)$ such that
  - all values smaller-or-equal to v come first;
  - followed by the have the value v;
  - followed by all other values (that are larger than v).
- Let pos be the position of v after Partition.
- QuickSort($L[start \dots pos)$).
- QuickSort($L[pos + 1 \dots end)$).



Proof of correctness: QuickSort($L[start \ldots end]$) sorts

Proof of correctness: QuickSort($L[start \dots end]$) sorts

Base case QuickSort sorts $0 \leq end - start \leq 1$ values.

Proof of correctness: QuickSort($L[start \dots end]$) sorts

Base case: QuickSort sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis: QuickSort sorts $0 \leq end - start < n$ values correctly.

Proof of correctness: QuickSort($L[start \dots end]$) sorts

Base case QuickSort sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis QuickSort sorts $0 \leq end - start < n$ values correctly.

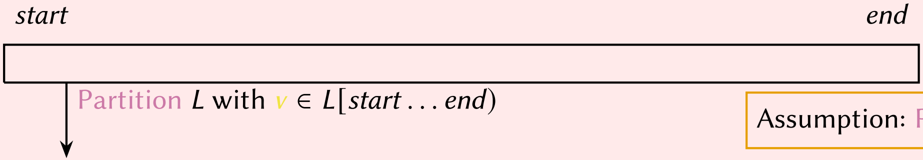
Induction step Consider QuickSort with $2 \leq end - start = n$ values.

Proof of correctness: QuickSort($L[start \ldots end)$) sorts

Base case QuickSort sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis QuickSort sorts $0 \leq end - start < n$ values correctly.

Induction step Consider QuickSort with $2 \leq end - start = n$ values.



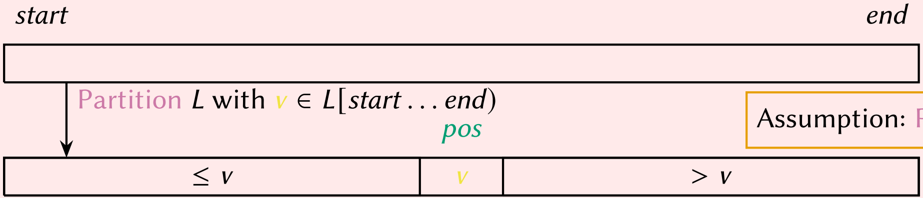
Assumption: Partition is correct

Proof of correctness: QuickSort($L[start \dots end)$) sorts

Base case QuickSort sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis QuickSort sorts $0 \leq end - start < n$ values correctly.

Induction step Consider QuickSort with $2 \leq end - start = n$ values.



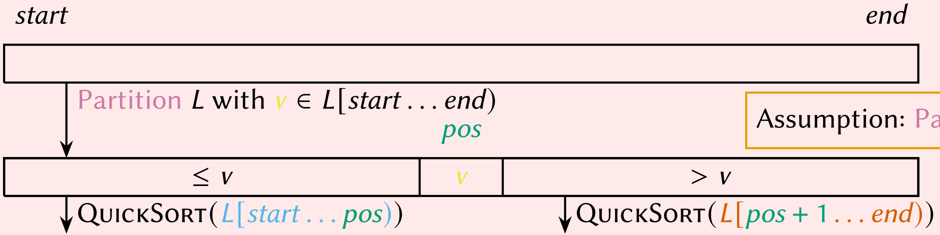
Assumption: Partition is correct

Proof of correctness: $\textsc{QuickSort}(L[start \ldots end))$ sorts

Base case $\textsc{QuickSort}$ sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis $\textsc{QuickSort}$ sorts $0 \leq end - start < n$ values correctly.

Induction step Consider $\textsc{QuickSort}$ with $2 \leq end - start = n$ values.



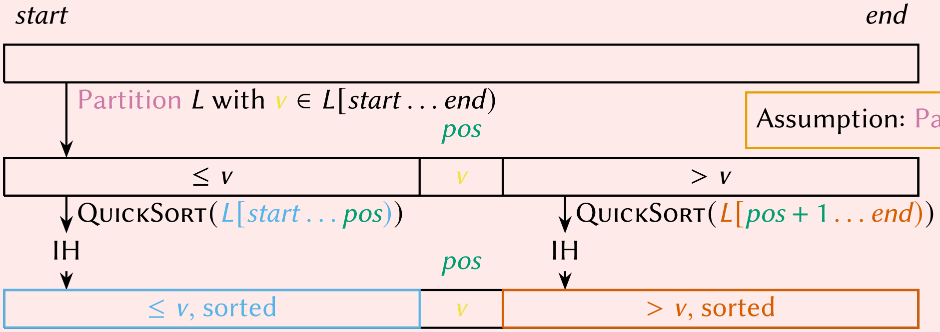
Assumption: Partition is correct

Proof of correctness: QuickSort($L[start \ldots end)$) sorts

Base case QuickSort sorts $0 \leq end - start \leq 1$ values.

Induction hypothesis QuickSort sorts $0 \leq end - start < n$ values correctly.

Induction step Consider QuickSort with $2 \leq end - start = n$ values.



Assumption: Partition is correct

$pos < end$

Assumption: Partition is correct

Algorithm PARTITION(L, $start$, end, p):

Assumption: Partition is correct

Algorithm PARTITION(L, *start*, *end*, p):

1: Exchange $L[start]$ and $L[p]$.

Assumption: Partition is correct

Algorithm PARTITION(L, $start$, end, p):

1: Exchange $L[start]$ and $L[p]$.
2: $v, i, j := L[start], start, start + 1$.

Values in $L[start + 1 \dots i + 1)$ are smaller-or-equal to v.

Values in $L[i + 1 \dots j)$ are larger than v.

Assumption: Partition is correct

Algorithm PARTITION(L, *start*, *end*, p):

Exchange $L[start]$ and $L[p]$.

$v, i, j := L[start], start, start + 1$.

Values in $L[start + 1 \dots i + 1)$ are smaller-or-equal to v.

Values in $L[i + 1 \dots j)$ are larger than v.

Exchange $L[i]$ and $L[start]$.

return i.

Assumption: Partition is correct

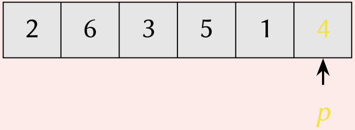
Algorithm PARTITION(L, $start$, end, p):

```
Exchange L[start] and L[p].
v, i, j := L[start], start, start + 1.
Values in L[start + 1 ... i + 1) are smaller-or-equal to v.
Values in L[i + 1 ... j) are larger than v.
while j ≠ end do
    if L[j] ≤ v then
        i := i + 1.
        Exchange L[i] and L[j].
    j := j + 1.
Exchange L[i] and L[start].
return i.
```

Assumption: Partition is correct

Algorithm PARTITION(L, $start$, end, p):

```
Exchange L[start] and L[p].
v, i, j := L[start], start, start + 1.
Values in L[start + 1 ... i + 1) are smaller-or-equal to v.
Values in L[i + 1 ... j) are larger than v.
while j ≠ end do
    if L[j] ≤ v then
        i := i + 1.
        Exchange L[i] and L[j].
    j := j + 1.
Exchange L[i] and L[start].
return i.
```



Assumption: Partition is correct

Algorithm PARTITION(L, $start$, end, p):

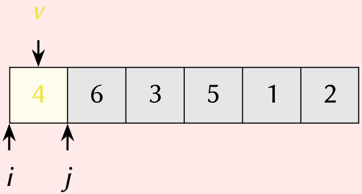
Exchange $L[start]$ and $L[p]$.
$v, i, j := L[start], start, start + 1$.
Values in $L[start + 1 \dots i + 1)$ are smaller-or-equal to v.
Values in $L[i + 1 \dots j)$ are larger than v.
while $j \neq end$ **do**
  if $L[j] \leq v$ **then**
    $i := i + 1$.
    Exchange $L[i]$ and $L[j]$.
  $j := j + 1$.
Exchange $L[i]$ and $L[start]$.
return i.

4	6	3	5	1	2

Assumption: Partition is correct

Algorithm PARTITION(L, *start*, *end*, p):

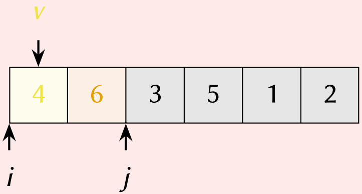
```
Exchange L[start] and L[p].
v, i, j := L[start], start, start + 1.
Values in L[start + 1 ... i + 1) are smaller-or-equal to v.
Values in L[i + 1 ... j) are larger than v.
while j ≠ end do
    if L[j] ≤ v then
        i := i + 1.
        Exchange L[i] and L[j].
    j := j + 1.
Exchange L[i] and L[start].
return i.
```



Assumption: Partition is correct

Algorithm PARTITION(L, $start$, end, p):

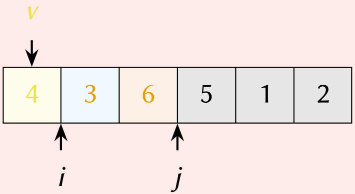
```
Exchange L[start] and L[p].
v, i, j := L[start], start, start + 1.
Values in L[start + 1 ... i + 1) are smaller-or-equal to v.
Values in L[i + 1 ... j) are larger than v.
while j ≠ end do
    if L[j] ≤ v then
        i := i + 1.
        Exchange L[i] and L[j].
    j := j + 1.
Exchange L[i] and L[start].
return i.
```



Assumption: Partition is correct

Algorithm PARTITION(L, *start*, *end*, p):

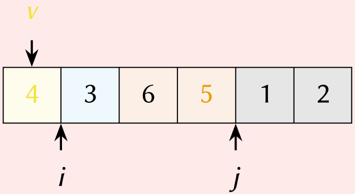
```
Exchange L[start] and L[p].
v, i, j := L[start], start, start + 1.
Values in L[start + 1 ... i + 1) are smaller-or-equal to v.
Values in L[i + 1 ... j) are larger than v.
while j ≠ end do
    if L[j] ≤ v then
        i := i + 1.
        Exchange L[i] and L[j].
    j := j + 1.
Exchange L[i] and L[start].
return i.
```



Assumption: Partition is correct

Algorithm PARTITION(L, *start*, *end*, p):

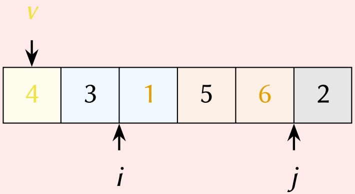
```
Exchange L[start] and L[p].
v, i, j := L[start], start, start + 1.
Values in L[start + 1 ... i + 1) are smaller-or-equal to v.
Values in L[i + 1 ... j) are larger than v.
while j ≠ end do
    if L[j] ≤ v then
        i := i + 1.
        Exchange L[i] and L[j].
    j := j + 1.
Exchange L[i] and L[start].
return i.
```



Assumption: Partition is correct

Algorithm Partition(L, $start$, end, p):

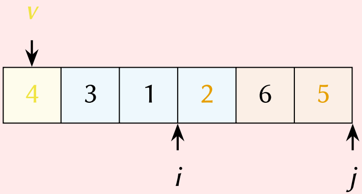
```
Exchange L[start] and L[p].
v, i, j := L[start], start, start + 1.
Values in L[start + 1 ... i + 1) are smaller-or-equal to v.
Values in L[i + 1 ... j) are larger than v.
while j ≠ end do
    if L[j] ≤ v then
        i := i + 1.
        Exchange L[i] and L[j].
    j := j + 1.
Exchange L[i] and L[start].
return i.
```



Assumption: Partition is correct

Algorithm PARTITION(L, *start*, *end*, p):

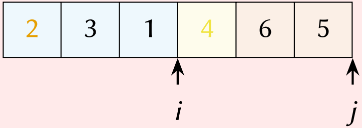
Exchange $L[start]$ and $L[p]$.
$v, i, j := L[start], start, start + 1$.
Values in $L[start + 1 \ldots i + 1)$ are smaller-or-equal to v.
Values in $L[i + 1 \ldots j)$ are larger than v.
while $j \neq end$ **do**
    if $L[j] \leq v$ **then**
        $i := i + 1$.
        Exchange $L[i]$ and $L[j]$.
    $j := j + 1$.
Exchange $L[i]$ and $L[start]$.
return i.



Assumption: Partition is correct

Algorithm PARTITION(L, *start*, *end*, p):

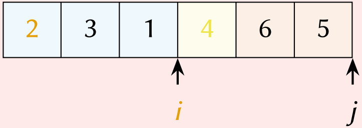
Exchange $L[start]$ and $L[p]$.
$v, i, j := L[start], start, start + 1$.
Values in $L[start + 1 \dots i + 1)$ are smaller-or-equal to v.
Values in $L[i + 1 \dots j)$ are larger than v.
while $j \neq end$ **do**
    if $L[j] \leq v$ **then**
        $i := i + 1$.
        Exchange $L[i]$ and $L[j]$.
    $j := j + 1$.
Exchange $L[i]$ and $L[start]$.
return i.



Assumption: Partition is correct

Algorithm PARTITION(L, $start$, end, p):

Exchange $L[start]$ and $L[p]$.
$v, i, j := L[start], start, start + 1$.
Values in $L[start + 1 \dots i + 1)$ are smaller-or-equal to v.
Values in $L[i + 1 \dots j)$ are larger than v.
while $j \neq end$ **do**
    if $L[j] \leq v$ **then**
        $i := i + 1$.
        Exchange $L[i]$ and $L[j]$.
    $j := j + 1$.
Exchange $L[i]$ and $L[start]$.
return i.



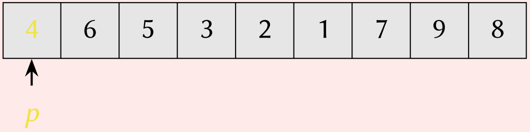
QuickSort: A complete example

We did not specify yet how to choose a pivot value!

4	6	5	3	2	1	7	9	8

QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



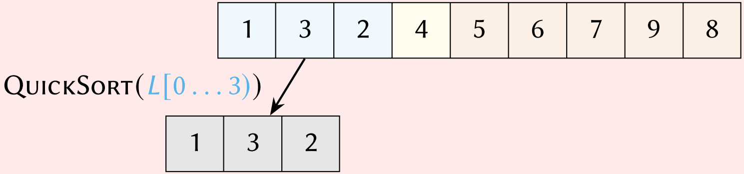
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.

1	3	2	4	5	6	7	9	8

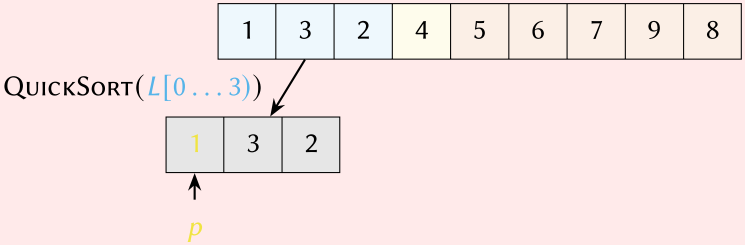
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



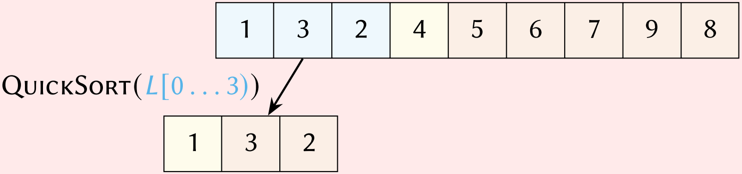
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



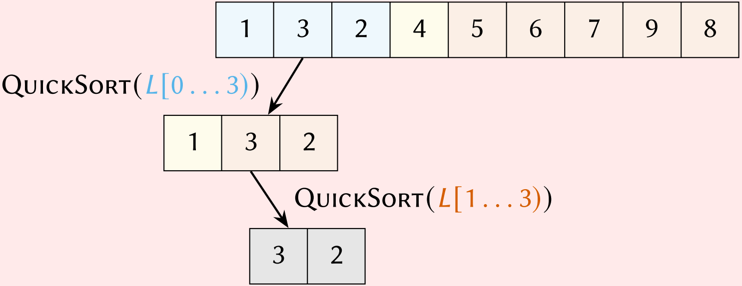
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



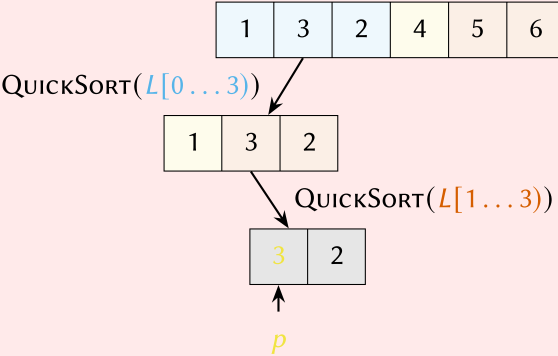
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



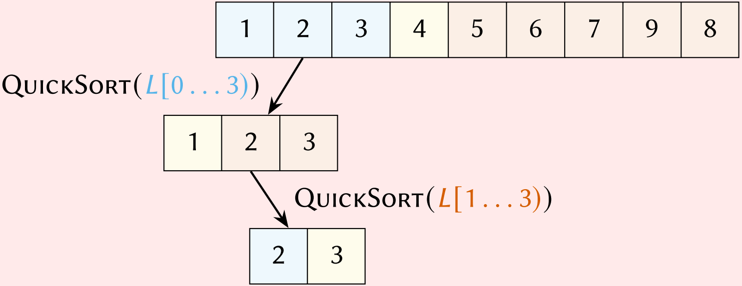
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



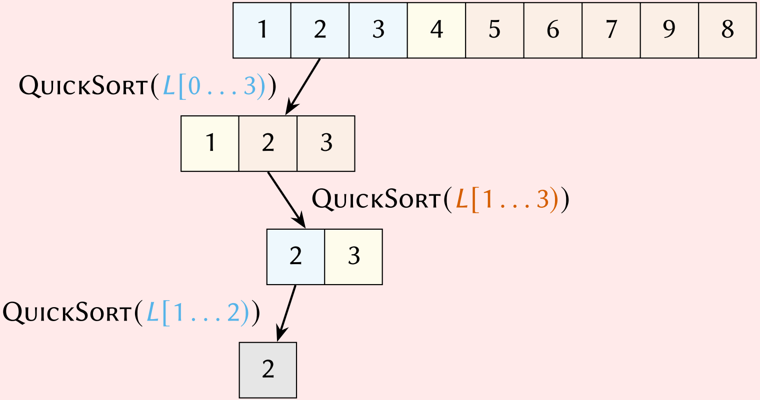
QuickSort: A complete example

We did not specify yet how to choose a pivot value $\longrightarrow$ random choices for now.



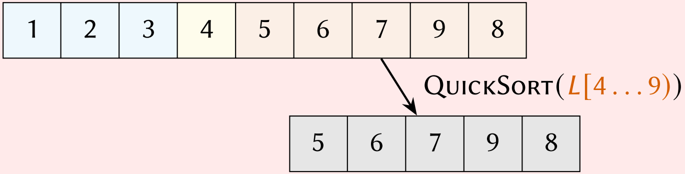
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



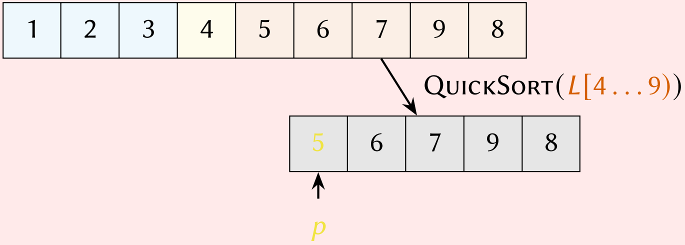
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



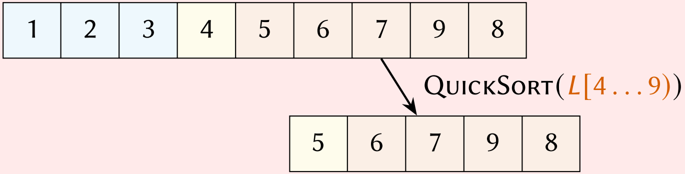
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



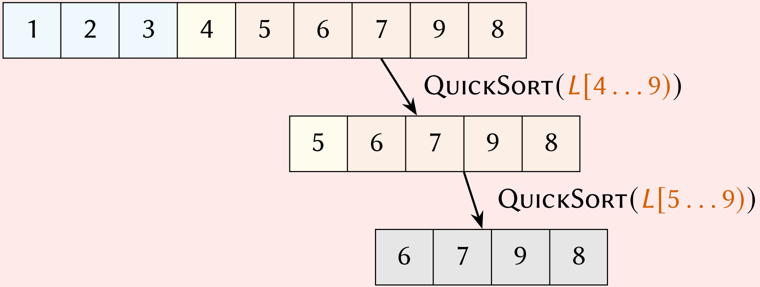
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



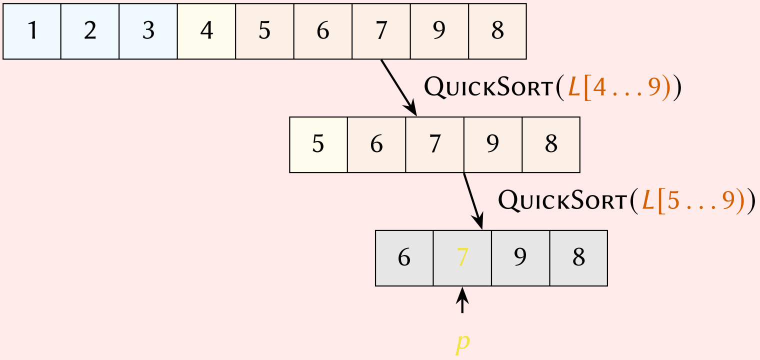
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



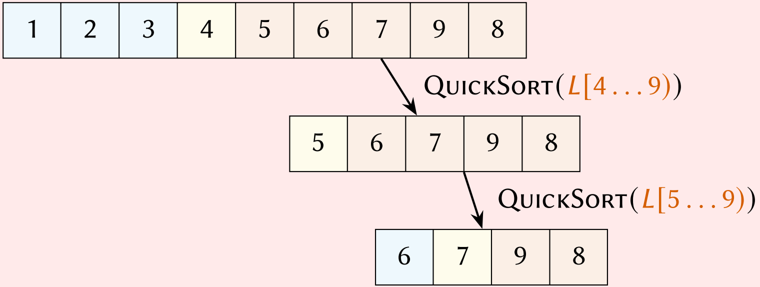
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



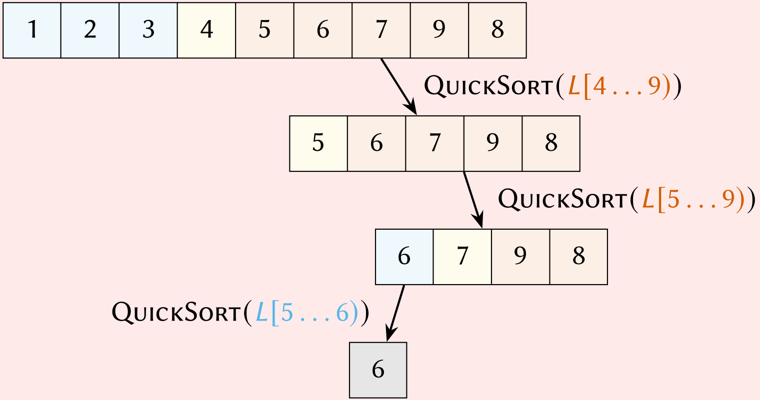
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



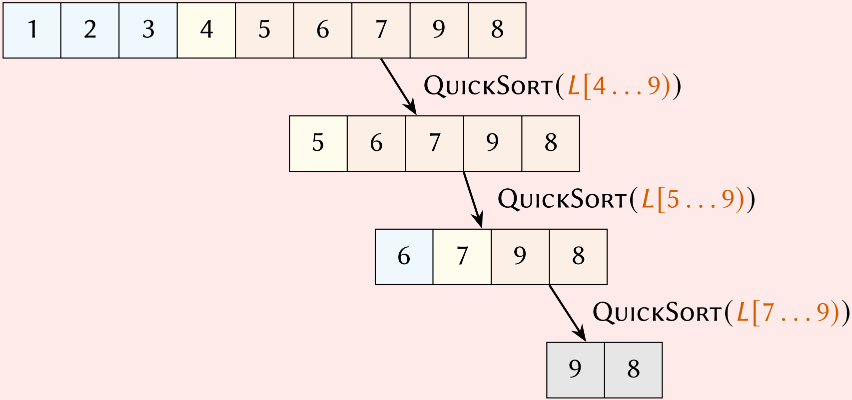
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



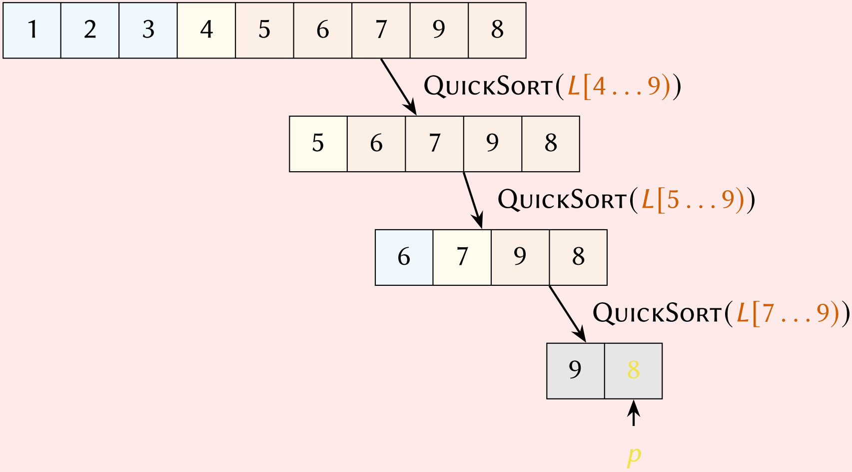
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



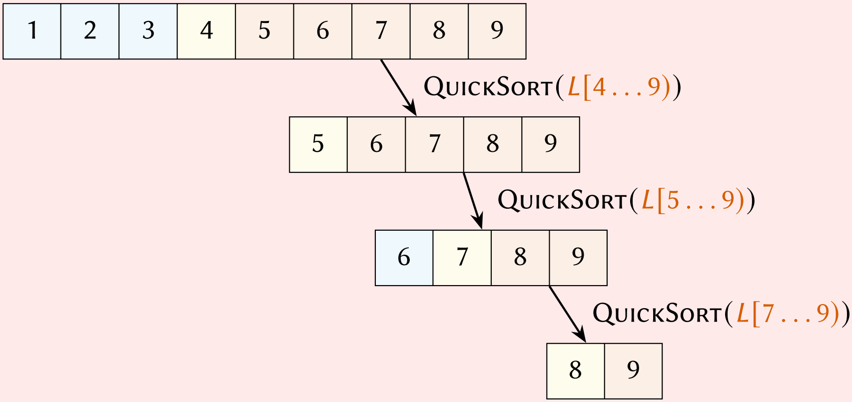
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



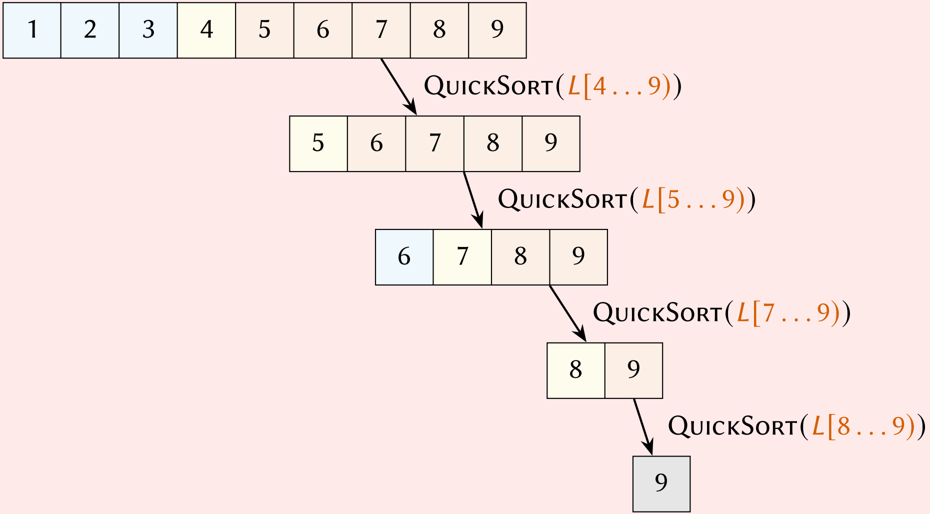
QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.



QuickSort: A complete example

We did not specify yet how to choose a pivot value → random choices for now.

1	2	3	4	5	6	7	8	9

The complexity of QuickSort

The complexity of QuickSort

The complexity of QuickSort depends on the chosen pivot values.

The complexity of QuickSort

Example: Pivots are always smaller than all other values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ & \text{if } N > 1. \end{cases}$$

The complexity of QuickSort

Example: Pivots are always smaller than all other values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T(N-1) + N & \text{if } N > 1. \end{cases}$$

The complexity of QuickSort

Example: Pivots are always smaller than all other values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T(N-1) + N & \text{if } N > 1. \end{cases}$$

	Number	Cost	Total
N	1	N	N

The complexity of QuickSort

Example: Pivots are always smaller than all other values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T(N-1) + N & \text{if } N > 1. \end{cases}$$

	Number	Cost	Total
N	1	N	N
↓			
$N-1$	1	$N-1$	$N-1$

The complexity of QuickSort

Example: Pivots are always smaller than all other values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T(N-1) + N & \text{if } N > 1. \end{cases}$$

	Number	Cost	Total
N	1	N	N
↓			
$N-1$	1	$N-1$	$N-1$
↓			
$N-2$	1	$N-2$	$N-2$

The complexity of QuickSort

Example: Pivots are always smaller than all other values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T(N-1) + N & \text{if } N > 1. \end{cases}$$

	Number	Cost	Total
N	1	N	N
↓			
$N-1$	1	$N-1$	$N-1$
↓			
$N-2$	1	$N-2$	$N-2$
↓			
$N-3$	1	$N-3$	$N-3$

The complexity of QuickSort

Example: Pivots are always smaller than all other values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T(N-1) + N & \text{if } N > 1. \end{cases}$$

	Number	Cost	Total
N	1	N	N
↓			
$N-1$	1	$N-1$	$N-1$
↓			
$N-2$	1	$N-2$	$N-2$
↓			
$N-3$	1	$N-3$	$N-3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	1	1

The complexity of QuickSort

Example: Pivots are always smaller than all other values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ T(N-1) + N & \text{if } N > 1. \end{cases}$$

	Number	Cost	Total
N	1	N	N
$\downarrow$			
$N-1$	1	$N-1$	$N-1$
$\downarrow$			
$N-2$	1	$N-2$	$N-2$
$\downarrow$			
$N-3$	1	$N-3$	$N-3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	1	1	1

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} = \Theta(N^2).$$

The complexity of QuickSort

Example: Pivots are "in the middle" of all values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ & \text{if } N > 1. \end{cases}$$

The complexity of QuickSort

Example: Pivots are “in the middle” of all values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T(\lfloor \frac{N}{2} \rfloor) + N & \text{if } N > 1. \end{cases}$$

The complexity of QuickSort

Example: Pivots are “in the middle” of all values

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ 2T(\lfloor \frac{N}{2} \rfloor) + N & \text{if } N > 1. \end{cases}$$

We have seen this one before: $T(N) = \Theta(N \log_2(N))$.

The complexity of QuickSort

The complexity of QuickSort depends *a lot* on the chosen pivot values.

The complexity of QuickSort

The complexity of QuickSort depends *a lot* on the chosen pivot values.

Randomized QuickSort: Choose pivot values fully at random

We *cannot* provide an exact complexity for Randomized QuickSort:
Executions on *the same list* can have vastly different random choices (and complexities).

The complexity of QuickSort

The complexity of QuickSort depends *a lot* on the chosen pivot values.

Randomized QuickSort: Choose pivot values fully at random

We *cannot* provide an exact complexity for Randomized QuickSort:
Executions on *the same list* can have vastly different random choices (and complexities).

Expected-case analysis: an analysis in terms of the distribution of random choices.

The complexity of QuickSort

The complexity of QuickSort depends *a lot* on the chosen pivot values.

Randomized QuickSort: Choose pivot values fully at random

We *cannot* provide an exact complexity for Randomized QuickSort:
Executions on *the same list* can have vastly different random choices (and complexities).

Expected-case analysis: an analysis in terms of the distribution of random choices.

Expected-case analysis is *not* average-case analysis!
Average-case analysis: an analysis in terms of the distribution of inputs.

The complexity of QuickSort

The complexity of QuickSort depends *a lot* on the chosen pivot values.

Randomized QuickSort: Choose pivot values fully at random

We *cannot* provide an exact complexity for Randomized QuickSort:
Executions on *the same list* can have vastly different random choices (and complexities).

Expected-case analysis: an analysis in terms of the distribution of random choices.

Any random choice in Randomized QuickSort is equally likely:

$$T(N) = \begin{cases} 1 & \text{if } N \leq 1; \\ \frac{1}{N}\left(\sum_{i=0}^{N-1}\Big(T(i) + T(N-(i+1))\Big)\right) + N & \text{if } N > 1. \end{cases}$$

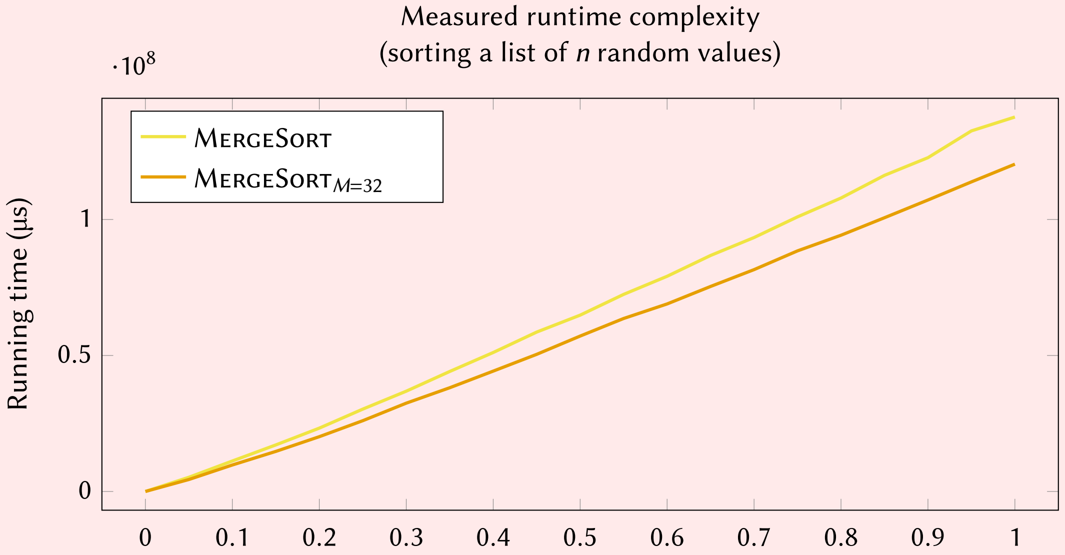
With some *mathematical tricks*, we can show that $T(N) = \Theta(N \log_2(N))$.

The complexity of QuickSort

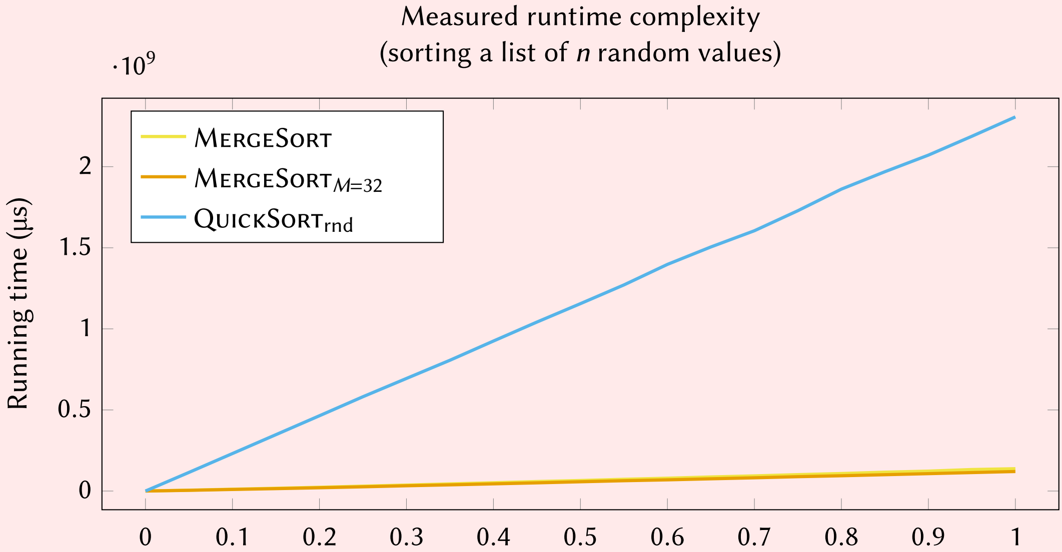
The complexity of QuickSort depends *a lot* on the chosen pivot values.

We will later develop a QuickSort variant that always has a $\Theta(N \log_2(N))$ complexity, this independent of how pivot values are chosen.

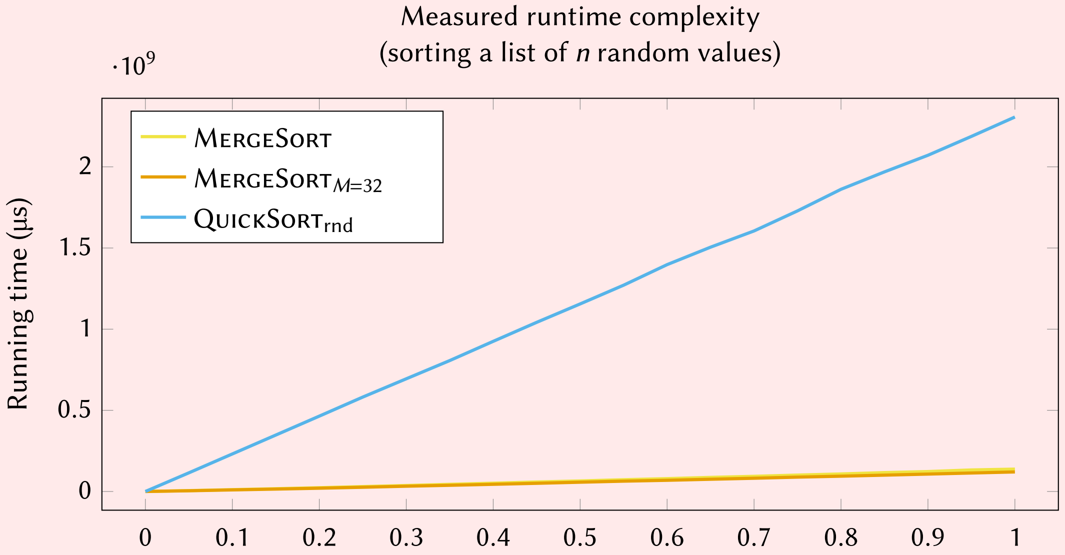
The performance of QuickSort



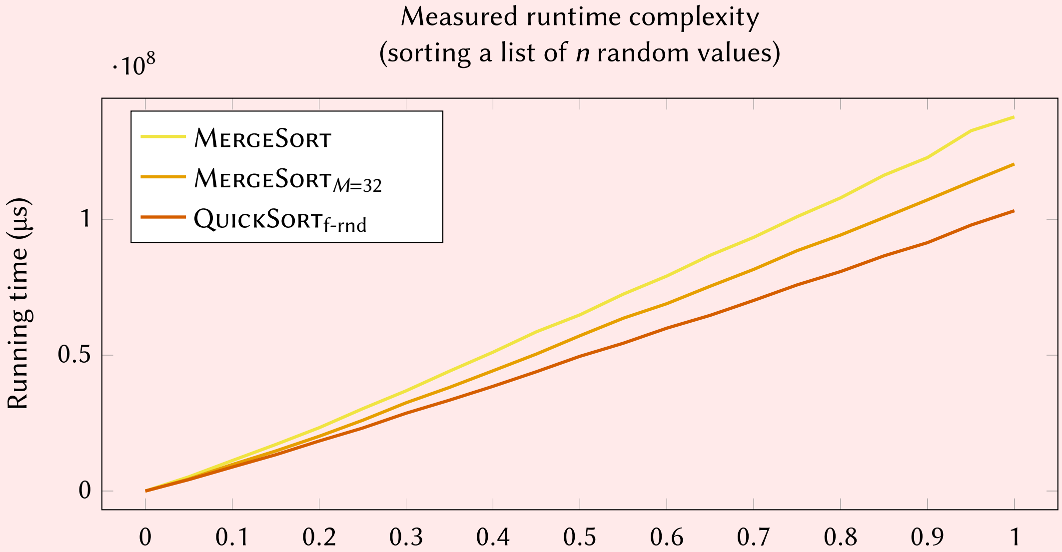
The performance of QuickSort



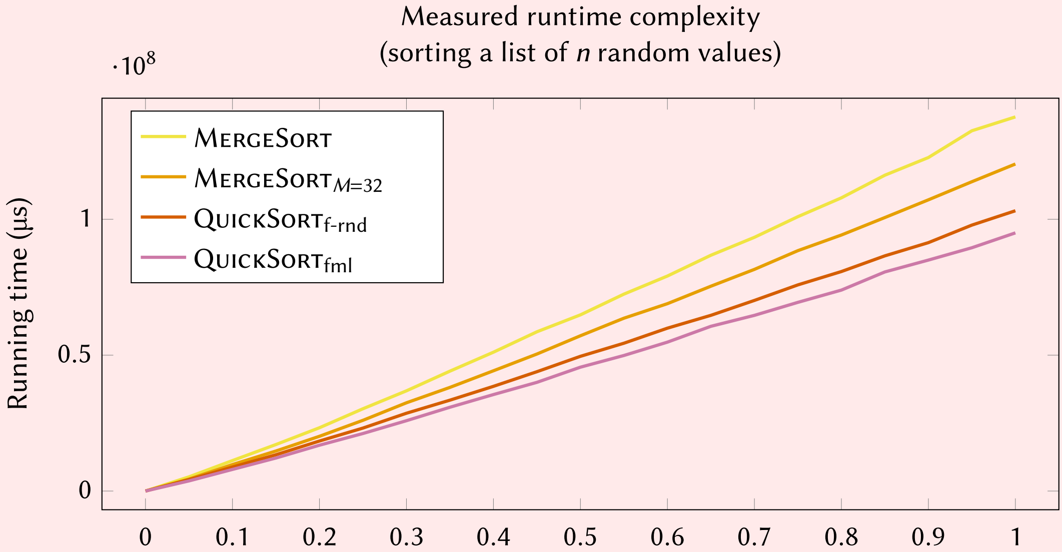
The performance of QuickSort



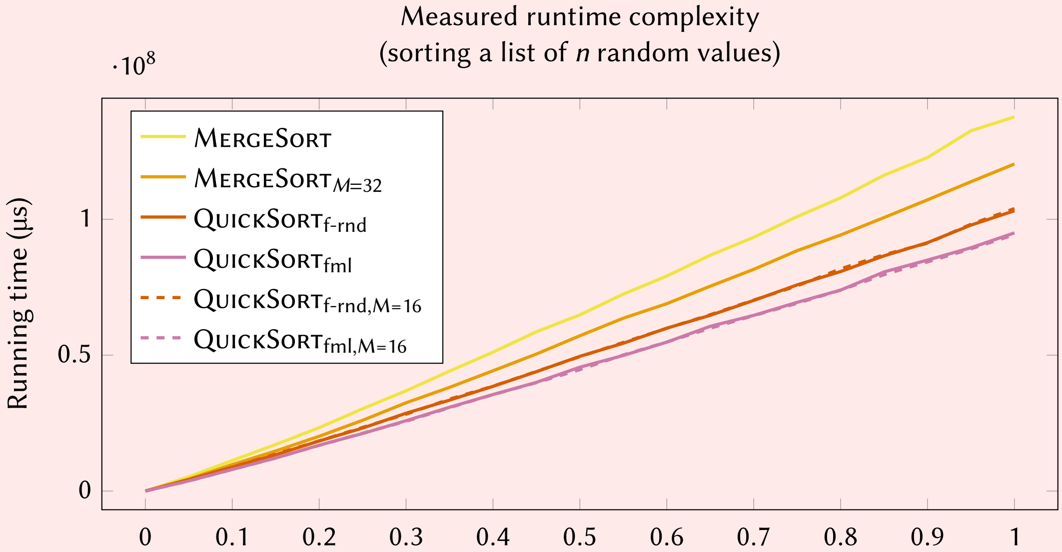
The performance of QuickSort



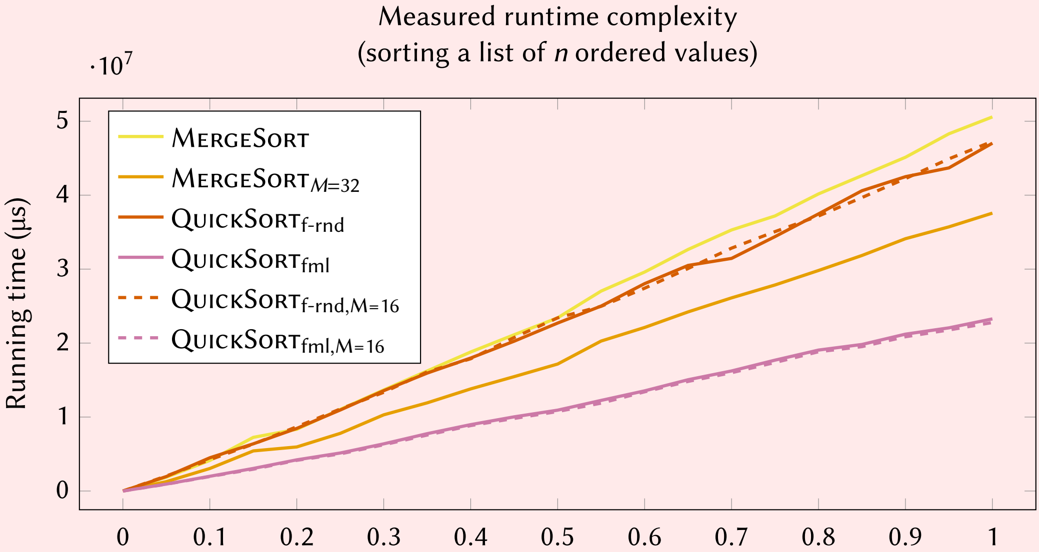
The performance of QuickSort



The performance of QuickSort



The performance of QuickSort



Further comparing MergeSort and QuickSort

	Comparisons	Changes	Memory
MergeSort	$\Theta(N \log_2(N))$	$N \log_2(N)$	$\Theta(N)$
QuickSort	$\Theta(N \log_2(N))$ (expected)	$\Theta(N \log_2(N))$ (expected)	$\Theta(\log_2(N))$ (expected)

Further comparing MergeSort and QuickSort

QuickSort is *not* stable

Consider a L list of pairs $(name, age)$ that is already sorted on age:

$$L = [(\text{Alicia}, 12), (\text{Dafni}, 20), (\text{Celeste}, 27), (\text{Dafni}, 35), (\text{Alicia}, 56), (\text{Celeste}, 80)].$$

Further comparing MergeSort and QuickSort

QuickSort is *not* stable

Consider a L list of pairs $(name, age)$ that is already sorted on age:

$$L = [(\text{Alicia}, 12), (\text{Dafni}, 20), (\text{Celeste}, 27), (\text{Dafni}, 35), (\text{Alicia}, 56), (\text{Celeste}, 80)].$$

- QuickSort$(L[0, 6))$ on names only will *not maintain* ordering on age:

$$[(\text{Alicia}, 56), (\text{Alicia}, 12), (\text{Celeste}, 27), (\text{Celeste}, 80), (\text{Dafni}, 35), (\text{Dafni}, 20)].$$

Further comparing MergeSort and QuickSort

QuickSort is *not* stable

Consider a L list of pairs $(name, age)$ that is already sorted on age:

$$L = [(\text{Alicia}, 12), (\text{Dafni}, 20), (\text{Celeste}, 27), (\text{Dafni}, 35), (\text{Alicia}, 56), (\text{Celeste}, 80)].$$

- QuickSort($L[0, 6)$) on names only will *not maintain* ordering on age:

  $$[(\text{Alicia}, 56), (\text{Alicia}, 12), (\text{Celeste}, 27), (\text{Celeste}, 80), (\text{Dafni}, 35), (\text{Dafni}, 20)].$$

- MergeSort($L[0, 6)$) on names only will *always maintain* pre-existing ordering (for values that are “identical”):

  $$[(\text{Alicia}, 12), (\text{Alicia}, 56), (\text{Celeste}, 27), (\text{Celeste}, 80), (\text{Dafni}, 20), (\text{Dafni}, 35)].$$

Further comparing MergeSort and QuickSort

QuickSort is *not* stable

Consider a L list of pairs $(name, age)$ that is already sorted on age:

$$L = [(\text{Alicia}, 12), (\text{Dafni}, 20), (\text{Celeste}, 27), (\text{Dafni}, 35), (\text{Alicia}, 56), (\text{Celeste}, 80)].$$

- QuickSort($L[0, 6)$) on names only will *not maintain* ordering on age:

$$[(\text{Alicia}, 56), (\text{Alicia}, 12), (\text{Celeste}, 27), (\text{Celeste}, 80), (\text{Dafni}, 35), (\text{Dafni}, 20)].$$

- MergeSort($L[0, 6)$) on names only will *always maintain* pre-existing ordering (for values that are "identical"):

$$[(\text{Alicia}, 12), (\text{Alicia}, 56), (\text{Celeste}, 27), (\text{Celeste}, 80), (\text{Dafni}, 20), (\text{Dafni}, 35)].$$

We say that MergeSort is *stable*.

Using Partition: Order statistics

Problem

Given a list $L[start \ldots end)$ and k, $start \leq k < end$, return the k-th smallest value in $L[start \ldots end)$.

Using PARTITION: Order statistics

Problem

Given a list $L[start \ldots end)$ and k, $start \leq k < end$,
return the k-th smallest value in $L[start \ldots end)$.

Algorithm SELECT(L, $start$, end, k):

```
Choose the position p ∈ [start, end) of the pivot value v := L[pos].
pos := PARTITION(L, start, end, p).
if pos = k then
    return L[pos].
else if pos > k then
    return SELECT(L, start, pos − 1, k).
else
    return SELECT(L, pos, end, k).
```

Using PARTITION: Order statistics

Problem

Given a list $L[start \dots end)$ and k, $start \leq k < end$, return the k-th smallest value in $L[start \dots end)$.

Algorithm SELECT(L, $start$, end, k):

Choose the position $p \in [start, end)$ of the *pivot value* $v := L[pos]$.
$pos :=$ PARTITION($L, start, end, p$).
if $pos = k$ **then**
    return $L[pos]$.
else if $pos > k$ **then**
    return SELECT($L, start, pos - 1, k$).
else
    return SELECT(L, pos, end, k).

Essentially a "half" QUICKSORT that only sorts those values that could be the k-th.

Using PARTITION: Order statistics

Problem

Given a list $L[start \ldots end)$ and k, $start \leq k < end$, return the k-th smallest value in $L[start \ldots end)$.

Algorithm SELECT(L, $start$, end, k):

Choose the position $p \in [start, end)$ of the *pivot value* $v := L[pos]$.
$pos :=$ PARTITION(L, $start$, end, p).
if $pos = k$ **then**
    return $L[pos]$.
else if $pos > k$ **then**
    return SELECT(L, $start$, $pos - 1$, k).
else
    return SELECT(L, pos, end, k).

Randomized SELECT: $\Theta(N)$ (expected).

Using Partition: Order statistics

Select(L, 0, 9, 6): We want the $k = 6$-th smallest value.

4	6	5	3	2	1	7	9	8

Using PARTITION: Order statistics

SELECT(L, 0, 9, 6): We want the k = 6-th smallest value.

4	6	5	3	2	1	7	9	8

↑

p

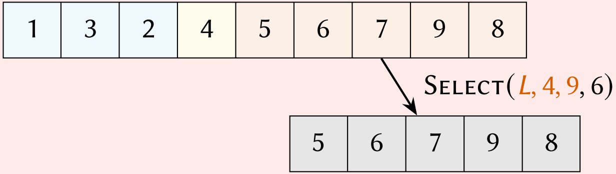
Using Partition: Order statistics

Select(L, 0, 9, 6): We want the $k = 6$-th smallest value.

1	3	2	4	5	6	7	9	8

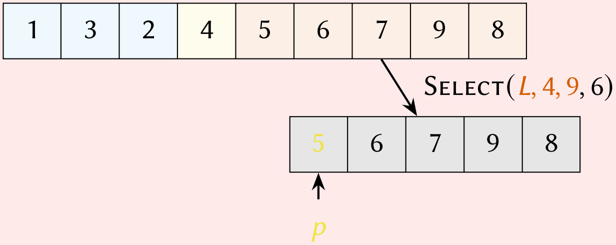
Using PARTITION: Order statistics

SELECT(L, 0, 9, 6): We want the $k = 6$-th smallest value.



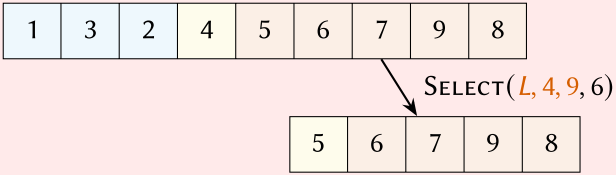
Using PARTITION: Order statistics

SELECT(L, 0, 9, 6): We want the k = 6-th smallest value.



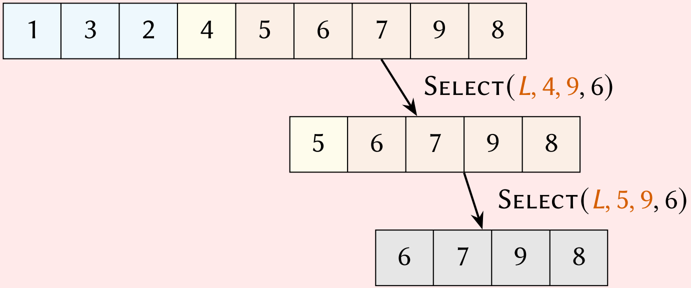
Using PARTITION: Order statistics

SELECT(L, 0, 9, 6): We want the k = 6-th smallest value.



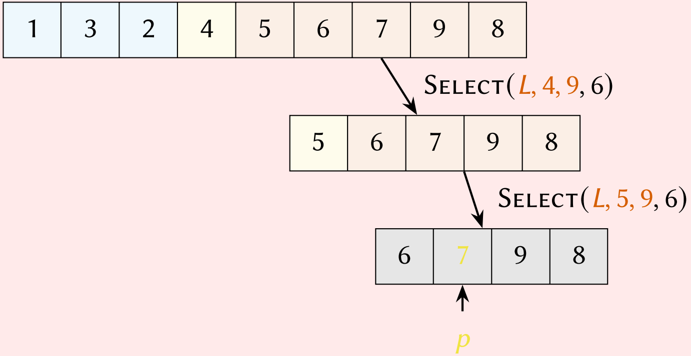
Using Partition: Order statistics

Select(L, 0, 9, 6): We want the $k = 6$-th smallest value.



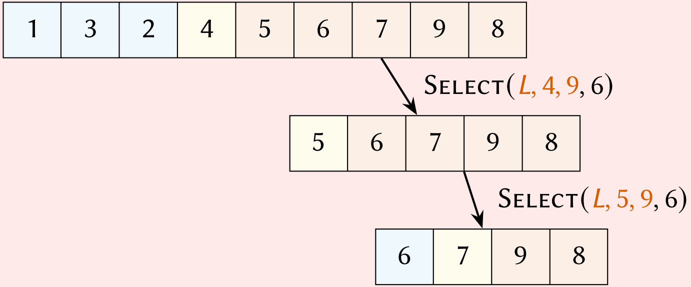
Using PARTITION: Order statistics

SELECT(L, 0, 9, 6): We want the k = 6-th smallest value.



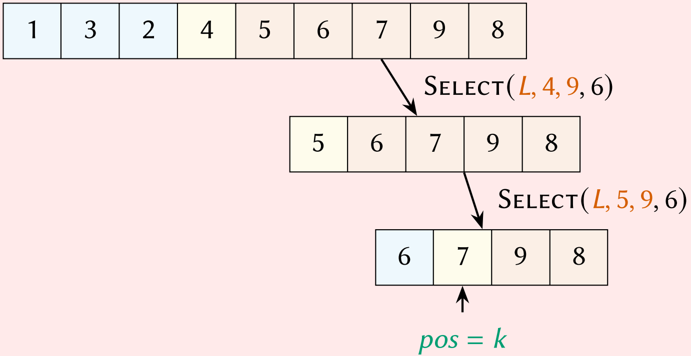
Using PARTITION: Order statistics

SELECT(L, 0, 9, 6): We want the k = 6-th smallest value.



Using PARTITION: Order statistics

SELECT(L, 0, 9, 6): We want the $k = 6$-th smallest value.



Final notes on QuickSort

	C++	Java
QuickSort	std::sort	java.util.Arrays.sort (non-Objects)
Partition	std::partition	
(related)	std::stable_partition	