Algorithms I

Markus Lohrey

Universität Siegen

Wintersemester 2023/2024

Overview, Literature

See `https://www.eti.uni-siegen.de/ti/lehre/ws2324/algo1/` for slides, exercise sheets, etc.

Overview:

1. Basics
2. Divide & Conquer
3. Sorting
4. Greedy algorithms
5. Dynamic programming
6. Graph algorithms

Literature:

- Cormen, Leiserson Rivest, Stein. Introduction to Algorithms (3. Auflage); MIT Press 2009
- Schöning, Algorithmik. Spektrum Akademischer Verlag 2001

Part 1: Basics

Overview

- Landau symbols
- logarithms, important formulas, Jensen's inequality
- complexity measures
- machine models

Landau symbols

Let $f, g : \mathbb{N} \to \mathbb{N}$ be functions.

- $g \in \mathcal{O}(f) \;\Leftrightarrow\; \exists c > 0 \,\exists n_0 \,\forall n \geq n_0 : \; g(n) \leq c \cdot f(n)$
  In other words: g is not growing faster than f.
- $g \in o(f) \;\Leftrightarrow\; \forall c > 0 \,\exists n_0 \,\forall n \geq n_0 : \; g(n) \leq c \cdot f(n)$
  In other words: g is growing strictly slower than f.
- $g \in \Omega(f) \;\Leftrightarrow\; f \in \mathcal{O}(g)$
  In other words: g is growing at least as fast than f.
- $g \in \omega(f) \;\Leftrightarrow\; f \in o(g)$
  In other words: g is growing strictly faster than f.
- $g \in \Theta(f) \;\Leftrightarrow\; (f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f))$
  In other words: g and f have the same asymptotic growth.

Landau Symbols

Reformulation of $g \in o(f)$ (assuming that $f(n) > 0$ for all $n \in \mathbb{N}$):

$$\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 : \; \frac{g(n)}{f(n)} \leq c.$$

This means that $\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$.

Examples:

- $2n \in \Theta(n)$
- $2n \notin o(n)$
- $2n \in o(n^2)$
- $\log_a(n) \in \mathcal{O}(\log_b(n))$ for all real numbers $a, b > 1$
- $(\log_a(n))^k \in o(n^\epsilon)$ for all $a, k > 1$ and $\epsilon > 0$

Logarithms

We assume some familiarity with logarithms.

Recall that following laws for all $b, c > 1$ and $x, y \geq 0$:

$$\begin{aligned} b^{\log_b x} &= x \\ \log_b(x \cdot y) &= \log_b(x) + \log_b(y) \\ \log_b(x^y) &= y \cdot \log_b(x) \\ \log_b(x) &= \frac{\log_c(x)}{\log_c(b)} \end{aligned}$$

Due to the last fact, we can write $\mathcal{O}(\log n)$ instead of $\mathcal{O}(\log_b n)$ (and similarly for Ω, Θ, o, and ω.

Some important formulas

Geometric sum:

$$\sum_{k=0}^{n} x^k = \frac{1 - x^{n+1}}{1 - x} \text{ for all } x \in \mathbb{R} \setminus \{1\}$$

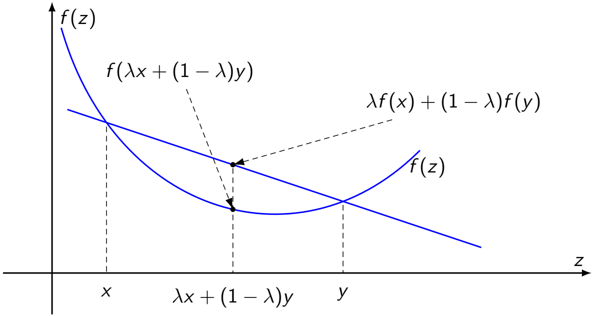
Geometric series:

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \text{ for all } x \in \mathbb{R} \text{ with } |x| < 1$$

Jensen's Inequality

Let $f : D \to \mathbb{R}$ be a function, where $D \subseteq \mathbb{R}$ is an interval.

- f is convex if for all $x, y \in D$ and all $0 \leq \lambda \leq 1$,
  $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$.



- f is concave if for all $x, y \in D$ and all $0 \leq \lambda \leq 1$,
  $f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda)f(y)$.

Jensen's Inequality

Jensen's inequality

If f is convex, then for all $x_1, \ldots, x_n \in D$ and all $\lambda_1, \ldots, \lambda_n \geq 0$ with $\lambda_1 + \cdots + \lambda_n = 1$:

$$f\left(\sum_{i=1}^{n} \lambda_i \cdot x_i\right) \leq \sum_{i=1}^{n} \lambda_i \cdot f(x_i).$$

If f is concave, then for all $x_1, \ldots, x_n \in D$ and all $\lambda_1, \ldots, \lambda_n \geq 0$ with $\lambda_1 + \cdots + \lambda_n = 1$:

$$f\left(\sum_{i=1}^{n} \lambda_i \cdot x_i\right) \geq \sum_{i=1}^{n} \lambda_i \cdot f(x_i).$$

Complexity measures

We describe the running time of an algorithm A as a function in the input length n.

Standard: Worst case complexity

Maximal running time on all inputs of length n:

$$t_{A,\text{worst}}(n) = \max\{t_A(x) \mid x \in X_n\},$$

where $X_n = \{x \mid |x| = n\}$.

Criticism: Unrealistic, since in practice worst-case inputs might not arise.

Complexity measures

Alternative: average case complexity.

Needs a probability distribution on X_n.

Standard: uniform distribution, i.e., $\text{Prob}(x) = \frac{1}{|X_n|}$.

Average running time:

$$
\begin{aligned}
t_{A,\varnothing}(n) &= \sum_{x \in X_n} \text{Prob}(x) \cdot t_A(x) \\
&= \frac{1}{|X_n|} \sum_{x \in X_n} t_A(x) \qquad \text{(for uniform distribution)}
\end{aligned}
$$

Problem: Difficult to analyse

Example: quicksort

Worst case number of comparisons of quicksort: $t_Q(n) \in \Theta(n^2)$.

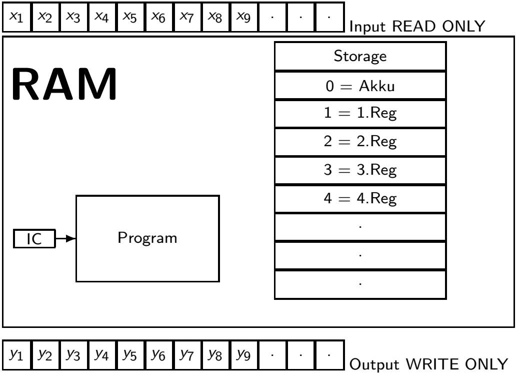
Average number of comparisons: $t_{Q,\varnothing}(n) = 1.38 n \log n$

Machine models: Turing machines

The Turing machine (TM) is a very simple and mathematically easy to define model of computation.

But: memory access (i.e., moving head to a certain symbol on the tape) is very time-consuming on a Turing machine and not realistic.

Machine models: Register machine (RAM)



Assumption: Elementary operations (e.g., the arithmetic operations $+, \times, -$, DIV, comparison, bitwise AND and OR) need a single computation step.

Part 2: Divide & Conquer

Overview

- Solving recursive equations
- Mergesort
- Fast multiplication of integers
- Matrix multiplication a la Strassen

Divide & Conquer: basic idea

As a first major design principle for algorithms, we will see Divide & Conquer:

Basic idea:

- Divide the input into several parts (usually of roughly equal size)
- Solve the problem on each part separately (recursion).
- Construct the overall solution from the sub-solutions.

Recursive equations

Divide & Conquer leads in a very natural way to recursive equations.

Assumptions:

- Input of length n will be split into a many parts of size n/b ($b \geq 2$).
- Dividing the input and merging the sub-solutions takes time $g(n)$.
- For an input of length 1 the computation time is $g(1)$.

This leads to the following recursive equation for the computation time:

$$\begin{aligned} t(1) &= g(1) \\ t(n) &= a \cdot t(n/b) + g(n) \end{aligned}$$

Recursive equations

Technical probem: What happens, if n is not divisible by b?

- Solution 1: Replace n/b by $\lceil n/b \rceil$.
- Solution 2: Assume that $n = b^k$ for some $k \geq 0$.

  If this does not hold: Stretch the input.

  For every $n \geq 1$ there exists a $k \geq 0$ with $n \leq b^k < b \cdot n$.

  If n is of the form b^k this is clear.

  Otherwise, there exists a unique k such that $b^{k-1} < n < b^k$.

  Hence, $n < b^k = b \cdot b^{k-1} < b \cdot n$.

  This means that padding the input length to a power of b only increases the input length by a constant.

Solving simple recursive equations

Theorem 1

Let $a, b \in \mathbb{N}$ *and* $b > 1$, $g : \mathbb{N} \longrightarrow \mathbb{N}$ *and assume the following equations:*

$$\begin{aligned} t(1) &= g(1) \\ t(n) &= a \cdot t(n/b) + g(n) \end{aligned}$$

Then for all $n = b^k$ *(i.e.,* $k = \log_b(n)$*):*

$$t(n) = \sum_{i=0}^{k} a^i \cdot g\left(\frac{n}{b^i}\right).$$

Proof: Induction over k.

$k = 0$: We have $n = b^0 = 1$ and $t(1) = g(1)$.

Solving simple recursive equations

$k > 0$: By induction we have

$$t\left(\frac{n}{b}\right) = \sum_{i=0}^{k-1} a^i \cdot g\left(\frac{n}{b^{i+1}}\right).$$

Hence:

$$\begin{aligned} t(n) &= a \cdot t\left(\frac{n}{b}\right) + g(n) \\ &= a\left(\sum_{i=0}^{k-1} a^i \cdot g\left(\frac{n}{b^{i+1}}\right)\right) + g(n) \\ &= \sum_{i=1}^{k} a^i \cdot g\left(\frac{n}{b^i}\right) + a^0 g\left(\frac{n}{b^0}\right) \\ &= \sum_{i=0}^{k} a^i \cdot g\left(\frac{n}{b^i}\right). \end{aligned}$$

□

Master theorem I

Theorem 2 (Master theorem I)

Let $a, b, c, d \in \mathbb{N}$ *with* $b > 1$ *and assume that*

$$\begin{aligned} t(1) &= d \\ t(n) &= a \cdot t(n/b) + d \cdot n^c \end{aligned}$$

Then, for all n *of the form* b^k *with* $k \geq 0$ *we have:*

$$t(n) \in \begin{cases} \Theta(n^c) & \text{if } a < b^c \\ \Theta(n^c \log n) & \text{if } a = b^c \\ \Theta(n^{\frac{\log a}{\log b}}) & \text{if } a > b^c \end{cases}$$

Remark: $\frac{\log a}{\log b} = \log_b a$. If $a > b^c$, then $\log_b a > c$.

Proof of the master theorem I

Let $g(n) = dn^c$. By Theorem 1 we have the following for $k = \log_b n$:

$$t(n) = \sum_{i=0}^{k} a^i \cdot d\left(\frac{n}{b^i}\right)^c = d \cdot n^c \cdot \sum_{i=0}^{k} \left(\frac{a}{b^c}\right)^i.$$

Case 1: $a < b^c$

$$t(n) \leq d \cdot n^c \cdot \sum_{i=0}^{\infty} \left(\frac{a}{b^c}\right)^i = d \cdot n^c \cdot \frac{1}{1 - \frac{a}{b^c}} \in \mathcal{O}(n^c).$$

Moreover, $t(n) \in \Omega(n^c)$, which implies $t(n) \in \Theta(n^c)$.

Case 2: $a = b^c$

$$t(n) = (k+1) \cdot d \cdot n^c \in \Theta(n^c \log n).$$

Proof of the master theorem I

Case 3: $a > b^c$

$$
\begin{aligned}
t(n) &= d \cdot n^c \cdot \sum_{i=0}^{k} \left(\frac{a}{b^c}\right)^i = d \cdot n^c \cdot \frac{\left(\frac{a}{b^c}\right)^{k+1} - 1}{\frac{a}{b^c} - 1} \\
&\in \Theta\left(n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b(n)}\right) \\
&= \Theta\left(\frac{n^c \cdot a^{\log_b(n)}}{b^{c \log_b(n)}}\right) \\
&= \Theta\left(a^{\log_b(n)}\right) \\
&= \Theta\left(b^{\log_b(a) \cdot \log_b(n)}\right) \\
&= \Theta\left(n^{\log_b(a)}\right)
\end{aligned}
$$

□

Stretching the input is ok

Stretching the input length to a b-power does not change the statement of the master theorem I.

Formally: Assume that the function t satisfies the following recursive equation for all $n \in \{b^m \mid m \geq 0\}$:

$$\begin{aligned} t(1) &= d \\ t(n) &= a \cdot t(n/b) + d \cdot n^c \end{aligned}$$

Define the function $t' : \mathbb{N} \to \mathbb{N}$ by $t'(n) = t(m)$, where m is the smallest number of the form b^k with $m \geq n$ (hence: $n \leq m \leq bn$).

With the master theorem I we get

$$t'(n) = t(m) \in \begin{cases} \Theta(m^c) = \Theta(n^c) & \text{if } a < b^c \\ \Theta(m^c \log m) = \Theta(n^c \log n) & \text{if } a = b^c \\ \Theta(m^{\frac{\log a}{\log b}}) = \Theta(n^{\frac{\log a}{\log b}}) & \text{if } a > b^c \end{cases}$$

Master theorem II

Theorem 3 (Master theorem II)

Let $r > 0$, $\sum_{i=0}^{r} \alpha_i < 1$ *and assume that for a constant* c,

$$t(n) \leq \left(\sum_{i=0}^{r} t(\lceil \alpha_i n \rceil) \right) + c \cdot n.$$

Then we have $t(n) \in \mathcal{O}(n)$.

Proof of the master theorem II

Choose $\varepsilon > 0$ and $n_0 > 0$ such that

$$\sum_{i=0}^{r} \lceil \alpha_i n \rceil \le (\sum_{i=0}^{r} \alpha_i) \cdot n + (r+1) \le (1-\varepsilon) n$$

for all $n \ge n_0$.

Choose γ such that $c \le \gamma\varepsilon$ and $t(n) \le \gamma n$ for all $n < n_0$.

By induction we get for all $n \ge n_0$:

$$\begin{aligned} t(n) &\le \left(\sum_{i=0}^{r} t(\lceil \alpha_i n \rceil)\right) + cn \\ &\le \left(\sum_{i=0}^{r} \gamma \lceil \alpha_i n \rceil\right) + cn \qquad \text{(induction)} \\ &\le (\gamma(1-\varepsilon) + c) n \\ &\le \gamma n \end{aligned}$$

□

Mergesort

We want to sort an array $A[1, n]$ of length n, where $n = 2^k$ for some $k \geq 0$.

The following recursive procedure sorts the subarray $A[\ell, r]$ going from position ℓ to position r ($\ell \leq r$):

Algorithm mergesort

```
procedure mergesort(ℓ, r)
var m : integer;
begin
  if (ℓ < r) then
    m := (r + ℓ) div 2;
    mergesort(ℓ, m);
    mergesort(m + 1, r);
    merge(ℓ, m, r);
  endif
endprocedure
```

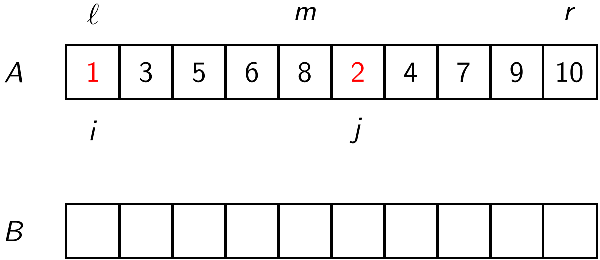
Mergesort

Algorithm merge

procedure merge(ℓ, m, r)
var i, j, k : **integer**;
begin
  $i = \ell$; $j := m + 1$;
  for $k := 1$ **to** $r - \ell + 1$ **do**
    if $i = m + 1$ **or** ($i \leq m$ **and** $j \leq r$ **and** $A[j] \leq A[i]$) **then**
      $B[k] := A[j]$; $j := j + 1$
    else
      $B[k] := A[i]$; $i := i + 1$
    endif
  endfor
  for $k := 0$ **to** $r - \ell$ **do**
    $A[\ell + k] := B[k + 1]$
  endfor
endprocedure

Mergesort

Example of merge(ℓ, m, r):



Mergesort

Example of merge(ℓ, m, r):

	ℓ				m					r
A	1	3	5	6	8	2	4	7	9	10
		i				j				
B	1									

Mergesort

Example of merge(ℓ, m, r):

	ℓ				m					r
A	1	3	5	6	8	2	4	7	9	10
		i					j			

B	1	2								

Mergesort

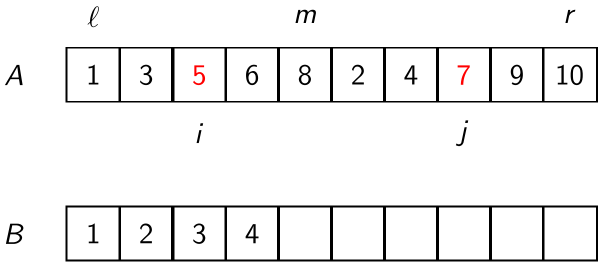
Example of merge(ℓ, m, r):

	ℓ				m					r
A	1	3	5	6	8	2	4	7	9	10
			i				j			

B	1	2	3							

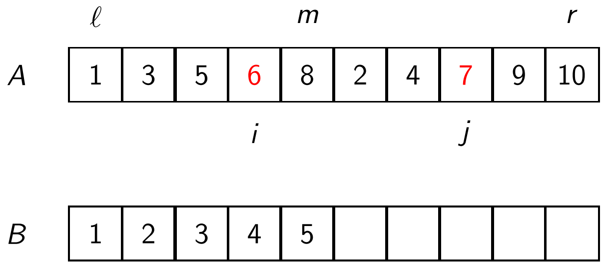
Mergesort

Example of merge(ℓ, m, r):



Mergesort

Example of merge(ℓ, m, r):



Mergesort

Example of merge(ℓ, m, r):

	ℓ				m					r
A	1	3	5	6	8	2	4	7	9	10
					i			j		

B	1	2	3	4	5	6				

Mergesort

Example of merge(ℓ, m, r):

	ℓ				m					r
A	1	3	5	6	8	2	4	7	9	10
					i				j	

B	1	2	3	4	5	6	7			

Mergesort

Example of merge(ℓ, m, r):

	ℓ				m					r
A	1	3	5	6	8	2	4	7	9	10
						i			j	

B	1	2	3	4	5	6	7	8		

Mergesort

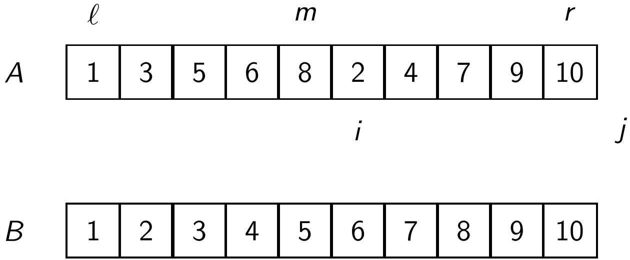
Example of merge(ℓ, m, r):

	ℓ				m					r
A	1	3	5	6	8	2	4	7	9	10
						i				j

B	1	2	3	4	5	6	7	8	9	

Mergesort

Example of merge(ℓ, m, r):



Mergesort

- Merge(ℓ, m, r) works in time $\mathcal{O}(r - \ell + 1)$.
- Total runng time of mergesort: $t_{\mathrm{ms}}(n) = 2 \cdot t_{\mathrm{ms}}(n/2) + d \cdot n$ for a constant d.

  Recall Master theorem 1: if

  $$t(1) = d \quad \text{and} \quad t(n) = a \cdot t(n/b) + d \cdot n^c.$$

  with $a, b, c, d \in \mathbb{N}$, $b > 1$, then we have:

  $$t(n) \in \begin{cases} \Theta(n^c) & \text{if } a < b^c \\ \Theta(n^c \log n) & \text{if } a = b^c \\ \Theta(n^{\frac{\log a}{\log b}}) & \text{if } a > b^c \end{cases}$$

  Setting $a = 2$, $b = 2$ and $c = 1$ yields $t_{\mathrm{ms}}(n) \in \Theta(n \log n)$.
- We will see later that $\mathcal{O}(n \log n)$ is asymptotically optimal for sorting algorithms that are only based on the comparison of elements.

Mergesort

- Drawback of Mergesort: no in-place sorting algorithm
- A sorting algorithm works in-place, if at every time instant only a constant number of elements from the input array A is stored outside of A.
- We will see in-place sorting algorithms with a running of $\mathcal{O}(n \log n)$.

Multiplication of natural numbers

We want to multiply two n-bit natural numbers, where $n = 2^k$ for some $k \geq 0$.

School method: $\Theta(n^2)$ bit operations.

Alternative approach:

$r =$	A	B
$s =$	C	D

Here, A (C) are the first $n/2$ bits and B (D) are the last $n/2$ bits of r (s).

We get

$$r = A\,2^{n/2} + B; \qquad s = C\,2^{n/2} + D$$

$$r\,s = A\,C\,2^n + (A\,D + B\,C)\,2^{n/2} + B\,D$$

Multiplication of natural numbers

Master theorem I: $t_{\text{mult}}(n) = 4 \cdot t_{\text{mult}}(n/2) + \Theta(n) \in \Theta(n^2)$

Here the overhead term $\Theta(n)$ comes from

- the addition of numbers of bit length at most $2n$ ($\Theta(n)$ bit operations)
- bit shifts like $AC \to AC \cdot 2^n$

No improvement over the school method!

Fast multiplication by A. Karatsuba, 1960

Compute recursively AC, $(A-B)(D-C)$ and BD.

Then, we get

$$\begin{aligned} rs &= A\,C\,2^n + A\,D\,2^{n/2} + B\,C\,2^{n/2} + B\,D \\ &= A\,C\,2^n + A\,D\,2^{n/2} - B\,D\,2^{n/2} - A\,C\,2^{n/2} + B\,C\,2^{n/2} \\ &\qquad + B\,D\,2^{n/2} + A\,C\,2^{n/2} + B\,D \\ &= A\,C\,2^n + (A-B)\,(D-C)\,2^{n/2} + (B\,D + A\,C)\,2^{n/2} + B\,D \end{aligned}$$

By the master theorem I, the total number of bit operations is:

$$t_{\text{mult}}(n) = 3 \cdot t_{\text{mult}}(n/2) + \Theta(n) \in \Theta(n^{\frac{\log 3}{\log 2}}) = \Theta(n^{1.58496\ldots}).$$

Using divide & conquer we reduced the exponent from 2 (school method) to 1.58496... !

How fast can we multiply?

- In 1971, Arnold Schönhage and Volker Strassen found an algorithm which multiplies two n-bit number in time $\mathcal{O}(n \cdot \log n \cdot \log \log n)$.
- The Schönhage-Strassen algorithm uses the so-called fast Fourier transformation (FFT); see Algorithms II.
- In practice, the Schönhage-Strassen algorithm beats Karatsuba's algorithm for numbers with approx. 10.000 digits.
- In 2019, Harvey and van der Hoeven found a multiplication algorithm running in time $\mathcal{O}(n \cdot \log n)$.

  `https://annals.math.princeton.edu/2021/193-2/p04`

  `https://web.maths.unsw.edu.au/~davidharvey/research/nlogn/index.html`

Matrix multiplication using naive divide & conquer

Let $A = (a_{i,j})_{1 \leq i,j \leq n}$ and $B = (b_{i,j})_{1 \leq i,j \leq n}$ be two $(n \times n)$-matrices.

For the product matrix $AB = (c_{i,j})_{1 \leq i,j \leq n} = C$ we have

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

$\Theta(n^3)$ scalar operations (additions and multiplications of numbers) are needed!

Divide & conquer: A, B are divided in 4 submatrices of roughly equal size. Then, the product $AB = C$ can be computed as follows:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Matrix multiplication using naive divide-and-conquer

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

where

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

We get (the $\Theta(n^2)$ term comes from the addition of $n \times n$ matrices)

$$t(n) = 8 \cdot t(n/2) + \Theta(n^2) \in \Theta(n^3).$$

No improvement!

Matrix multiplication by Volker Strassen (1969)

Compute the product of two 2×2 matrices with 7 multiplications:

$$
\begin{aligned}
M_1 &:= (A_{12} - A_{22})(B_{21} + B_{22}) \\
M_2 &:= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_3 &:= (A_{11} - A_{21})(B_{11} + B_{12}) \\
M_4 &:= (A_{11} + A_{12})B_{22} \\
M_5 &:= A_{11}(B_{12} - B_{22}) \\
M_6 &:= A_{22}(B_{21} - B_{11}) \\
M_7 &:= (A_{21} + A_{22})B_{11}
\end{aligned}
\qquad
\begin{aligned}
C_{11} &= M_1 + M_2 - M_4 + M_6 \\
C_{12} &= M_4 + M_5 \\
C_{21} &= M_6 + M_7 \\
C_{22} &= M_2 - M_3 + M_5 - M_7
\end{aligned}
$$

Running time: $t(n) = 7 \cdot t(n/2) + \Theta(n^2)$.

Master theorem I ($a = 7,\ b = 2,\ c = 2$):

$$t(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2{,}81\ldots})\ .$$

The story of fast matrix multiplication

- Strassen 1969: $n^{2.81\ldots}$
- Pan 1979: $n^{2.796\ldots}$
- Bini, Capovani, Romani, Lotti 1979: $n^{2.78\ldots}$
- Schönhage 1981: $n^{2.522\ldots}$
- Romani 1982: $n^{2.517\ldots}$
- Coppersmith, Winograd 1981: $n^{2.496\ldots}$
- Strassen 1986: $n^{2.479\ldots}$
- Coppersmith, Winograd 1987: $n^{2.376\ldots}$
- Stothers 2010: $n^{2.374\ldots}$
- Williams 2014: $n^{2.372873\ldots}$
- Le Gall 2014: $n^{2.3728639\ldots}$
- Alman, Williams 2020: $n^{2.3728596\ldots}$
- Duan, Wu, Zhou 2022: $n^{2.371866\ldots}$
- Williams, Xu, Xu, and Zhou 2023: $n^{2.371552\ldots}$

Part 3: Sorting

Overview

- Lower bounds for comparison-based sorting algorithms
- Quicksort
- Heapsort
- sorting in linearer time
- median computation

Comparison-based sorting algorithms

A sorting algorithm is comparison-based if the elements of the input array belong to a data type that only supports the comparison of two elements.

Such a sorting algorithm can be seen as a general purpose sorting algorithm.

Example: Mergesort, Heapsort, Quicksort are all comparison-based.

Theorem 4 (lower bound for comparison-based sorting)

For every comparison-based sorting algorithm and every n there exists an array of length n, on which the algorithm makes at least

$$n \log_2(n) - \log_2(e) n \geq n \log_2(n) - 1.443n$$

many comparisons.

Comparison-based sorting algorithms

Proof: We fix some comparison-based sorting algorithm $\mathcal{S}$ and an array length n.

We prove the lower bound for input arrays $A[1, \dots, n]$ with the following properties:

- $A[i] \in \{1, \dots, n\}$ for all $1 \leq i \leq n$.
- $A[i] \neq A[j]$ for $i \neq j$

In other words: The input is a permutation of the list $[1, 2, \dots, n]$.

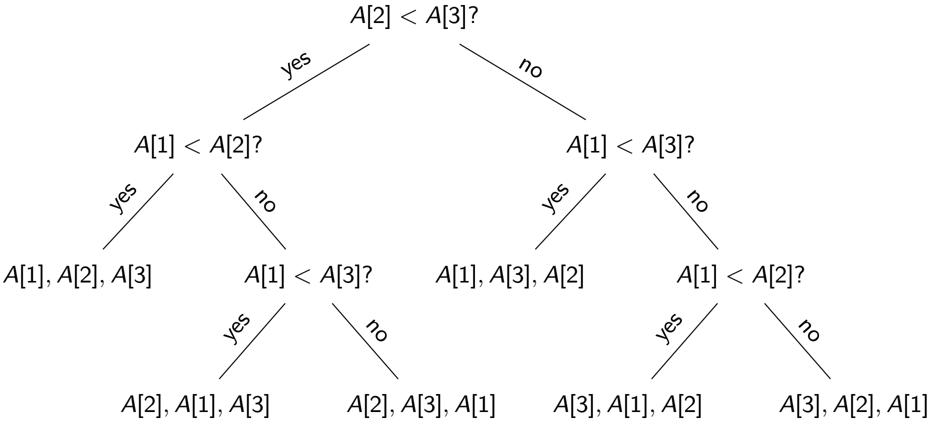
The sorting algorithm has $\mathcal{S}$ to sort this list.

Another point of view: $\mathcal{S}$ has to learn the order between the numbers $A[1], A[2], \dots, A[n]$ by comparing these numbers.

This process that can be described by a decision tree.

Comparison-based sorting algorithms

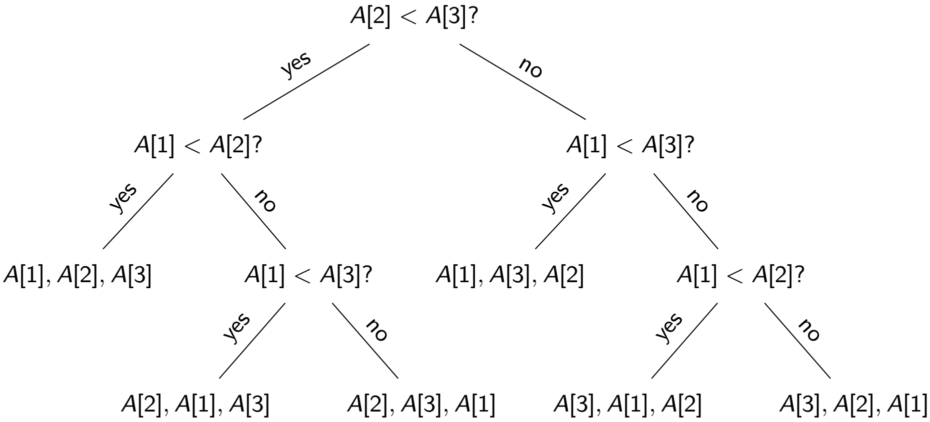
Example: $A[1]$ $A[2]$ $A[3]$



Comparison-based sorting algorithms

Example:

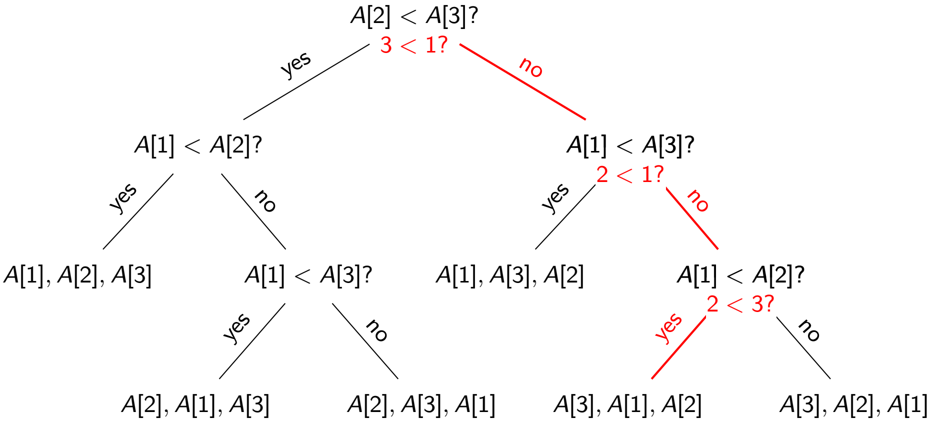
$A[1]$	$A[2]$	$A[3]$
2	3	1



Comparison-based sorting algorithms

Example:

$A[1]$	$A[2]$	$A[3]$
2	3	1



Lower bound for the worst case

Construction of the decision tree for our sorting algorithm $\mathcal{S}$ and array length n:

- We execute the algorithm on an array $A[1, \ldots, n]$ without knowing the concrete values $A[i]$.
- Assume that the algorithm compares $A[i]$ and $A[j]$ in the first step.
- The left (right) subtree is obtained by continuing the algorithm under the assumption that $A[i] < A[j]$ ($A[i] > A[j]$).

Observation 1:

- If $A[1, \ldots, n]$ and $B[1, \ldots, n]$ are different input arrays (i.e., different permutations of $[1, 2, \ldots, n]$) then they correspond to different leaves of the decision tree.
- Therefore, the decision tree has $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$ many leaves.

Lower bound for the worst case

Observation 2: The depth (= max. number of edges on a path from the root to a leaf) of the decision tree is the maximal number of comparisons of the algorithm on an input array of length n.

A combinatorial fact: A binary tree with N leaves has depth $\geq \log_2(N)$.

Stirling's formula (we only need $n! > \sqrt{2\pi n}\,(n/e)^n$) implies

$$\begin{aligned}
\log_2(n!) &\geq \log_2(\sqrt{2\pi n}\,(n/e)^n)) \\
&= \log_2((n/e)^n) + \log_2((2\pi n)^{1/2}) \\
&= n\log_2(n) - \log_2(e)n + \Theta(\log n) \\
&\geq n\log_2(n) - 1.443n.
\end{aligned}$$

Thus, there exists an input array for which the algorithm makes at least $n\log_2(n) - 1.443n$ many comparisons. □

Lower bound for the average case

A comparison-based sorting algorithm even makes $n \log_2(n) - 2.443n$ many comparisons on almost all input permutations of $[1, \ldots, n]$.

Theorem 5

For every comparison-based sorting algorithm and every n the following holds: for at least $(1 - 2^{-n+1}) \cdot n!$ *permutations of* $[1, 2, \ldots, n]$*, the algorithm makes at least*

$$\log_2(n!) - n \geq n \log_2(n) - 2.443n$$

many comparisons.

For the proof we need the following simple lemma.

Lower bound for the average case

Lemma 6

Let $A \subseteq \{0,1\}^*$ with $|A| = N$, and let $1 \leq n < \log_2(N)$. Then, at least $(1 - 2^{-n+1}) \cdot N$ many words in A have length $\geq \log_2(N) - n$.

Proof:

Case 1. $N = 2^m$ for some m.

Let $M = |\{w \in A : |w| < \log_2(N) - n = m - n\}|$.

$$M \leq \sum_{k=0}^{m-n} 2^k = 2^{m-n+1} - 1 < 2^{-n+1} \cdot 2^m = 2^{-n+1} \cdot N$$

Hence,

$$N - M > N - 2^{-n+1} \cdot N = (1 - 2^{-n+1}) \cdot N$$

words in A have length $\geq \log_2(N) - n$.

Lower bound for the average case

Case 2. N is not of the form 2^m.

Let us write $N = 2^m + r$ with $0 < r < 2^m$.

Let $M = |\{w \in A : |w| \leq \lfloor \log_2(N) \rfloor - n = m - n\}|$.

$$M \leq \sum_{k=0}^{m-n} 2^k = 2^{m-n+1} - 1 < 2^{-n+1} \cdot 2^m < 2^{-n+1} \cdot N$$

Hence,

$$N - M > N - 2^{-n+1} \cdot N = (1 - 2^{-n+1}) \cdot N$$

words in A have length $\geq \lfloor \log_2(N) \rfloor + 1 - n \geq \log_2(N) - n$. □

Lower bound for the average case

Consider again the decision tree. It has $n!$ leaves, and every leaf corresponds to a permutation of $[1, \dots, n]$.

Thus, each of the $n!$ many permutations can be encoded by a word over the alphabet $\{0, 1\}$:

- 0 means: go in the decision tree to the left child.
- 1 means: go in the decision tree to the right child.

By Lemma 6 (with $N = n!$), the decision tree has at least $(1 - 2^{-n+1}) \cdot n!$ many root-leaf paths of length $\geq \log_2(n!) - n \geq n \cdot \log_2(n) - 2.443n$. □

Lower bound for the average case

Corollary

Every comparison-based sorting algorithm makes on average at least $n \log_2(n) - 2.443n$ many comparisons when sorting a random permutation of $[1, \ldots, n]$ (for n large enough).

Proof: Due to Theorem 5 at least

$$\begin{aligned}
(1 - 2^{-n+1}) \cdot (\log_2(n!) - n) + 2^{-n+1} &= \\
\log_2(n!) - n - \frac{\log_2(n!) - n - 1}{2^{n-1}} &\geq \\
n \log_2(n) - \log_2(e)n + \Theta(\log n) - n - \frac{\log_2(n!) - n - 1}{2^{n-1}} &\geq \\
n \log_2(n) - 2.443n &
\end{aligned}$$

many comparisons are done in the average. □

Quicksort

The Quicksort-algorithm (Tony Hoare, 1962):

- Choose an array-element $P = A[p]$ (the pivot element).
- Partitioning: Permute the array entries such that on the left (resp., right) of the pivot element P all elements are $\leq P$ (resp., $> P$) (needs $n-1$ comparisons).
- Apply the algorithm recursively to the subarrays to the left and right of the pivot element.

Critical: choice of the pivot elements.

- Running time is optimal, if the pivot element is the middle element of the array entries $\{A[1], \ldots, A[n]\}$ (median).
- Good choice in practice: median-out-of-three

Partitioning

First, we present a procedure for partitioning a subarray $A[\ell, \dots, r]$ with respect to a pivot element $P = A[p]$, where $\ell < r$ and $\ell \leq p \leq r$.

The procedure returns an index $m \in \{\ell, \dots, r\}$ with the following properties:

- $A[m] = P$
- $A[k] \leq P$ for all $\ell \leq k \leq m - 1$
- $A[k] > P$ for all $m + 1 \leq k \leq r$

swap(i, j) swaps the array entries at positions i and j:
$x := A[i]$; $A[i] := A[j]$; $A[j] := x$

Partitioning

Algorithm Partition

```
function partition(A[ℓ...r] : array of integer, p : integer) : integer
begin
   swap(p, r);
   P := A[r];
   i := ℓ − 1; j := ℓ;
   while j < r do
      if A[j] ≤ P then
         i := i + 1;
         swap(i, j)
      endif
      j := j + 1;
   endwhile
   swap(i + 1, r)
   return i + 1
endfunction
```

Partitioning

Note: partition($A[\ell \ldots r]$) makes $r - \ell$ many comparisons.

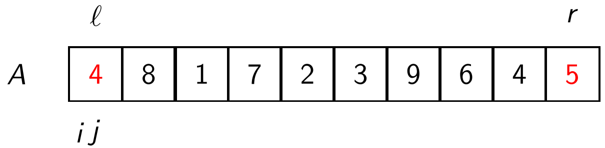
Example with $P = 5$.

		ℓ									r
A		4	8	1	7	2	3	9	6	4	5
	i	j									

Partitioning

Note: partition($A[\ell \ldots r]$) makes $r - \ell$ many comparisons.

Example with $P = 5$.



Partitioning

Note: partition($A[\ell \dots r]$) makes $r - \ell$ many comparisons.

Example with $P = 5$.

	ℓ									r
A	4	8	1	7	2	3	9	6	4	5
	i	j								

Partitioning

Note: partition($A[\ell \ldots r]$) makes $r - \ell$ many comparisons.

Example with $P = 5$.

	ℓ									r
A	4	8	1	7	2	3	9	6	4	5
	i		j							

Partitioning

Note: partition($A[\ell \dots r]$) makes $r - \ell$ many comparisons.

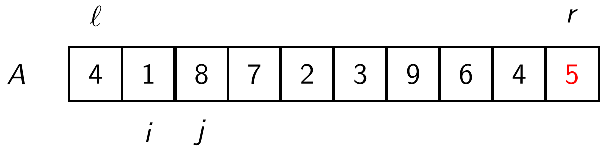
Example with $P = 5$.

	ℓ									r
A	4	8	1	7	2	3	9	6	4	5
		i	j							

Partitioning

Note: partition($A[\ell \dots r]$) makes $r - \ell$ many comparisons.

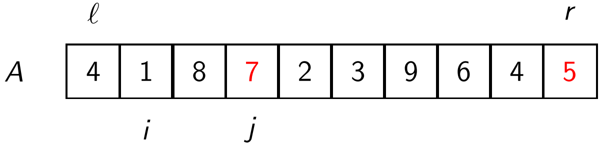
Example with $P = 5$.



Partitioning

Note: partition($A[\ell \dots r]$) makes $r - \ell$ many comparisons.

Example with $P = 5$.



Partitioning

Note: partition($A[\ell \dots r]$) makes $r - \ell$ many comparisons.

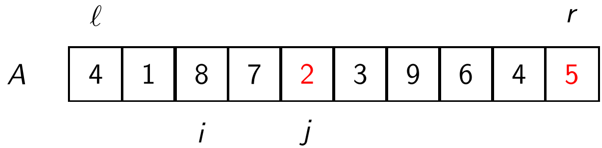
Example with $P = 5$.

	ℓ									r
A	4	1	8	7	2	3	9	6	4	5
		i			j					

Partitioning

Note: partition($A[\ell \ldots r]$) makes $r - \ell$ many comparisons.

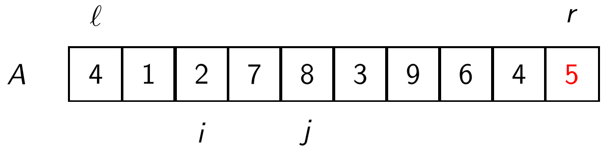
Example with $P = 5$.



Partitioning

Note: partition($A[\ell \dots r]$) makes $r - \ell$ many comparisons.

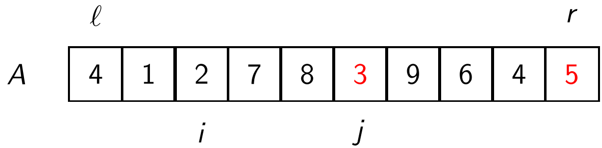
Example with $P = 5$.



Partitioning

Note: partition($A[\ell \ldots r]$) makes $r - \ell$ many comparisons.

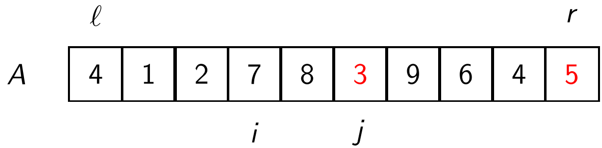
Example with $P = 5$.



Partitioning

Note: partition($A[\ell \dots r]$) makes $r - \ell$ many comparisons.

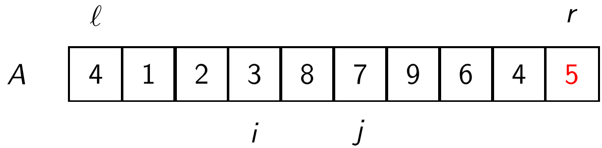
Example with $P = 5$.



Partitioning

Note: partition($A[\ell \dots r]$) makes $r - \ell$ many comparisons.

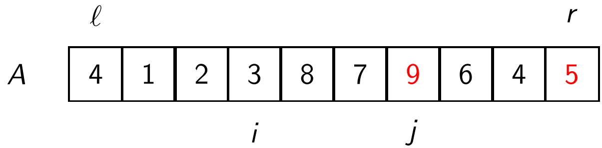
Example with $P = 5$.



Partitioning

Note: partition($A[\ell \ldots r]$) makes $r - \ell$ many comparisons.

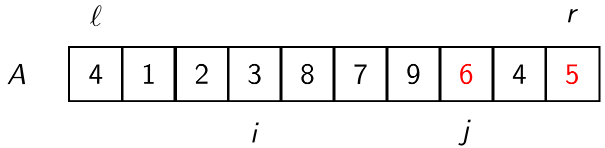
Example with $P = 5$.



Partitioning

Note: partition($A[\ell \ldots r]$) makes $r - \ell$ many comparisons.

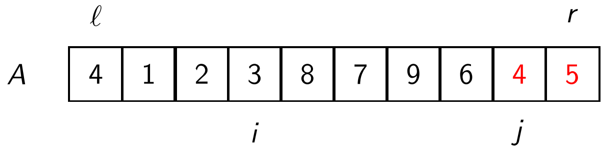
Example with $P = 5$.



Partitioning

Note: partition($A[\ell \dots r]$) makes $r - \ell$ many comparisons.

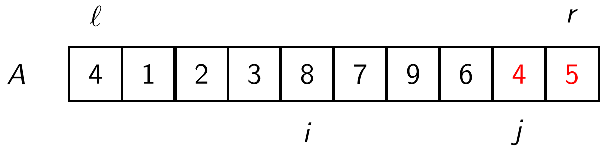
Example with $P = 5$.



Partitioning

Note: partition($A[\ell \ldots r]$) makes $r - \ell$ many comparisons.

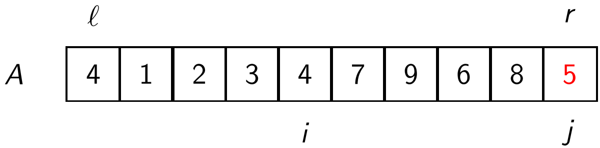
Example with $P = 5$.



Partitioning

Note: partition($A[\ell \ldots r]$) makes $r - \ell$ many comparisons.

Example with $P = 5$.



Partitioning

Note: partition($A[\ell \ldots r]$) makes $r - \ell$ many comparisons.

Example with $P = 5$.

	ℓ									r
A	4	1	2	3	4	5	9	6	8	7
						$i+1$				

Correctness of partitioning

The following invariants hold before every iteration of the **while**-loop:

- $A[r] = P$
- $A[k] \leq P$ for all $\ell \leq k \leq i$
- $A[k] > P$ for all $i + 1 \leq k \leq j - 1$

These invariants trivially hold before the first iteration of the **while**-loop, when $i = \ell - 1$ and $j = \ell$.

Assume now that the above invariant holds before a certain iteration of the **while**-loop and let A', i', $j' = j + 1$ be the values of A, i, j after the iteration.

Case 1. $A[j] > P$.

Then $A' = A$, $i' = i$ and $j' = j + 1$.

In particular $A'[j' - 1] = A[j] > P$.

Hence the invariants also hold for A', i', j'.

Correctness of partitioning

Case 2. $A[j] \leq P$.

Then $i' = i + 1$, $j' = j + 1$, $A'[i'] = A[j] \leq P$, $A'[j' - 1] = A[i + 1]$ and $A'[k] = A[k]$ for $i' \neq k \neq j' - 1$.

Note: if $i + 1 \leq j - 1$ (i.e., $i' + 1 \leq j' - 1$) then $A'[j' - 1] = A[i + 1] > P$.

Hence, the above invariants also hold for A', i' and j'.

Taking the invariants at the end of the **while**-loop (when $j = r$) yields:

- $A[r] = P$
- $A[k] \leq P$ for all $\ell \leq k \leq i$
- $A[k] > P$ for all $i + 1 \leq k \leq r - 1$

Hence, after swap$(i + 1, r)$ we have:

- $A[k] \leq P$ for all $\ell \leq k \leq i + 1$
- $A[k] > P$ for all $i + 2 \leq k \leq r$
- $A[i + 1] = P$

Quicksort

Algorithm Quicksort

procedure quicksort($A[\ell \dots r]$: **array of integer**)
begin
  if $\ell < r$ **then**
    $p :=$ index of the median of $A[\ell]$, $A[(\ell + r) \text{ div } 2]$, $A[r]$;
    $m :=$ partition($A[\ell \dots r]$, p);
    quicksort($A[\ell \dots m-1]$);
    quicksort($A[m+1 \dots r]$);
  endif
endprocedure

Worst-case running time: $\mathcal{O}(n^2)$.

The worst-case arises when after each call of partition($A[\ell \dots r]$, p), one of the subarrays ($A[\ell \dots m-1]$ or $A[m+1 \dots r]$) is empty.

Quicksort: average case analysis

Average case analysis under the assumption that the pivot element is chosen randomly.

Alternatively: Input array is chosen randomly.

Let $Q(n)$ be the avergage number of comparisons for an input array of length n.

Theorem 7

We have $Q(n) = 2(n+1)H(n) - 4n$*, where*

$$H(n) := \sum_{k=1}^{n} \frac{1}{k}$$

is the n-th harmonic number.

Quicksort: average case analysis

Proof:

For $n = 0$ we have $Q(0) = 0 = 2 \cdot 1 \cdot 0 - 4 \cdot 0$.

For $n = 1$ we have $Q(1) = 0 = 2 \cdot 2 \cdot 1 - 4 \cdot 1$.

For $n \geq 2$ we have:

$$
\begin{aligned}
Q(n) &= (n-1) + \frac{1}{n} \sum_{i=1}^{n} [Q(i-1) + Q(n-i)] \\
&= (n-1) + \frac{2}{n} \sum_{i=1}^{n} Q(i-1)
\end{aligned}
$$

Note:

- $(n-1)$ = number of comparisons for partitioning.
- $Q(i-1) + Q(n-i)$ = average number of comparisons for the recursive sorting of the two subarrays.
- The factor $1/n$ comes from the fact that every pivot element is chosen with probability $1/n$.

Quicksort: average case analysis

We get:

$$nQ(n) = n(n-1) + 2\sum_{i=1}^{n} Q(i-1)$$

Hence:

$$\begin{aligned} nQ(n) - (n-1)Q(n-1) &= n(n-1) + 2\sum_{i=1}^{n} Q(i-1) \\ &\quad -(n-1)(n-2) - 2\sum_{i=1}^{n-1} Q(i-1) \\ &= n(n-1) - (n-2)(n-1) + 2Q(n-1) \\ &= 2(n-1) + 2Q(n-1) \end{aligned}$$

We obtain:

$$\begin{aligned} nQ(n) &= 2(n-1) + 2Q(n-1) + (n-1)Q(n-1) \\ &= 2(n-1) + (n+1)Q(n-1) \end{aligned}$$

Quicksort: average case analysis

Dividing both sides by $n(n+1)$ gives:

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

Using induction on n we get:

$$\begin{aligned}
\frac{Q(n)}{n+1} &= \sum_{k=1}^{n} \frac{2(k-1)}{k(k+1)} \\
&= 2\sum_{k=1}^{n} \frac{(k-1)}{k(k+1)} \\
&= 2\left[\sum_{k=1}^{n} \frac{2}{k+1} - \sum_{k=1}^{n} \frac{1}{k}\right] \text{ since } \frac{2}{k+1} - \frac{1}{k} = \frac{(k-1)}{k(k+1)}
\end{aligned}$$

Quicksort: average case analysis

Recall that $H(n) = \sum_{k=1}^{n} \frac{1}{k}$.

$$\begin{aligned}
\frac{Q(n)}{n+1} &= 2\left[2\sum_{k=2}^{n+1}\frac{1}{k} - \sum_{k=1}^{n}\frac{1}{k}\right] \\
&= 2\left[2\left(\frac{1}{n+1} + H(n) - 1\right) - H(n)\right] \\
&= 2H(n) + \frac{4}{n+1} - 4.
\end{aligned}$$

Finally, we get for $Q(n)$:

$$\begin{aligned}
Q(n) &= 2(n+1)H(n) + 4 - 4(n+1) \\
&= 2(n+1)H(n) - 4n. \qquad \square
\end{aligned}$$

Quicksort: average case analysis

- One has $H(n) - \ln n \approx 0{,}57721\ldots =$ Euler's constant. Hence:

$$\begin{aligned} Q(n) &\approx 2(n+1)(0{,}58 + \ln n) - 4n \\ &\approx 2n \ln n - 2{,}8n \approx 1{,}38n \log_2 n - 2{,}8n. \end{aligned}$$

- Theoretical optimum: $\log_2(n!) \approx n \log_2 n - 1{,}44n$.
- In the average, quicksort is only 38% worse than the optimum.
- An average analysis of the media-out-of-three method yields $1{,}18n \log_2 n - 2{,}2n$.
- It is in the average only 18% worse than the optimum.

Heaps

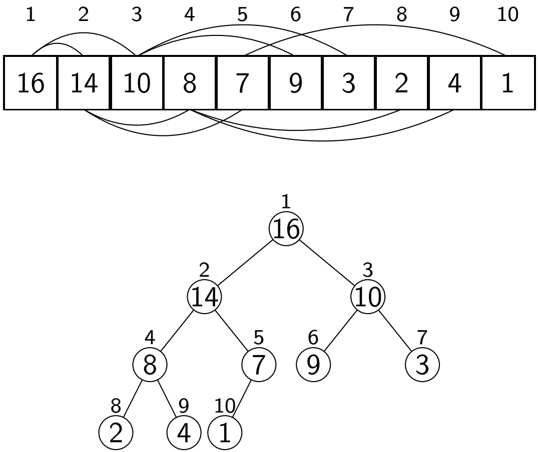
Definition 8

A (max-)heap is an array $A[1 \ldots n]$ with the following properties:

- $A[i] \geq A[2i]$ for all $i \geq 1$ with $2i \leq n$
- $A[i] \geq A[2i+1]$ for all $i \geq 1$ with $2i+1 \leq n$

Heaps

Example:

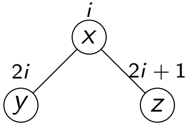


Sinking process

In a first step we will permute the entries of the array $A[1, \ldots, n]$ such that the heap condition is satisfied.

Assume that the subarray $A[i+1, \ldots, n]$ already satisfies the heap condition.

In order to enforce the heap condition also for i we let $A[i]$ sink:



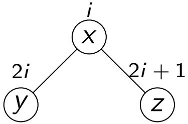
With 2 comparisons one can compute $\max\{x, y, z\}$.

Sinking process

In a first step we will permute the entries of the array $A[1, \ldots, n]$ such that the heap condition is satisfied.

Assume that the subarray $A[i+1, \ldots, n]$ already satisfies the heap condition.

In order to enforce the heap condition also for i we let $A[i]$ sink:



With 2 comparisons one can compute $\max\{x, y, z\}$.

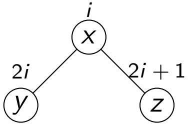
If x is the max., then the sinking process stops.

Sinking process

In a first step we will permute the entries of the array $A[1, \ldots, n]$ such that the heap condition is satisfied.

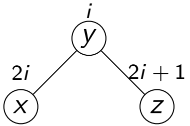
Assume that the subarray $A[i+1, \ldots, n]$ already satisfies the heap condition.

In order to enforce the heap condition also for i we let $A[i]$ sink:



With 2 comparisons one can compute $\max\{x, y, z\}$.

If y is the max., then x and y are swapped and we continue at $2i$.

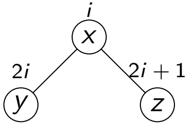


Sinking process

In a first step we will permute the entries of the array $A[1, \dots, n]$ such that the heap condition is satisfied.

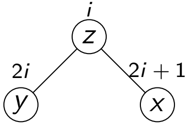
Assume that the subarray $A[i+1, \dots, n]$ already satisfies the heap condition.

In order to enforce the heap condition also for i we let $A[i]$ sink:



With 2 comparisons one can compute $\max\{x, y, z\}$.

If z is the max., then x and z are swapped and we continue at $2i+1$.



Reheap

Algorithm Reheap

```
procedure reheap(i, n: integer)                     (* i is the root *)
var m: integer;
begin
  if i ≤ n/2 then
    m := max{A[i], A[2i], A[2i + 1]};               (* 2 comparisons! *)
    if (m ≠ A[i]) ∧ (m = A[2i]) then
      swap(i, 2i);                                  (* swap x, y *)
      reheap(2i, n)
    elsif (m ≠ A[i]) ∧ (m = A[2i + 1]) then
      swap(i, 2i + 1);                              (* swap x, z *)
      reheap(2i + 1, n)
    endif
  endif
endprocedure
```

Building the heap

Algorithm Build Heap

procedure build-heap(n: **integer**)
begin
  for $i := \left\lfloor \frac{n}{2} \right\rfloor$ **downto** 1 **do**
    reheap(i, n)
  endfor
endprocedure

Invariant: Before the call of reheap(i, n) the subarray $A[i+1, \ldots, n]$ satisfies the heap condition.

Clearly, this hods for $i = \left\lfloor \frac{n}{2} \right\rfloor$.

Assume that the invariant holds for i.

Thus, the heap condition can only fail for i.

After the sinking process for $A[i]$, the heap condition also holds for i.

Time analysis for building the heap

Theorem 9

Built-heap runs in time $\mathcal{O}(n)$.

Proof: Sinking of $A[i]$ needs $2 \cdot \text{height(subtree under } A[i])$ comparisons.

We carry out the computation for $n = 2^k - 1$.

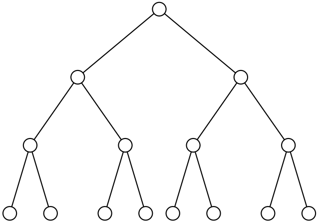
Then we have a complete binary tree of height $k - 1$.

There are

- 2^0 trees of height $k - 1$,
- 2^1 trees of height $k - 2$,

  ⋮
- 2^{k-1-i} trees of height i,

  ⋮
- 2^{k-1} trees of height 0.



$k = 4$

Time analysis for building the heap

Hence, building the heap needs at most

$$2 \cdot \sum_{i=0}^{k-1} 2^{k-1-i} i = 2^k \cdot \sum_{i=0}^{k-1} i \cdot 2^{-i} \leq (n+1) \cdot \sum_{i \geq 0} i \cdot 2^{-i}$$

many comparisons.

Claim: $\sum_{j \geq 0} j \cdot 2^{-j} = 2$

Proof of the claim: For every $|z| < 1$ we have

$$\sum_{j \geq 0} z^j = \frac{1}{1-z}.$$

Time analysis for building the heap

Taking derivatives yields

$$\sum_{j \geq 0} j \cdot z^{j-1} = \frac{1}{(1-z)^2},$$

and hence

$$\sum_{j \geq 0} j \cdot z^{j} = \frac{z}{(1-z)^2}.$$

Setting $z = 1/2$ yields

$$\sum_{j \geq 0} j \cdot 2^{-j} = 2.$$

□

Standard Heapsort (W. J. Williams, 1964)

Algorithm Heapsort

```
procedure heapsort(n: integer)
begin
   build-heap(n)
   for i := n downto 2 do
      swap(1, i);
      reheap(1, i − 1)
   endfor
endprocedure
```

Theorem 10

Standard Heapsort sorts an array with n elements and needs at most $2n\log_2 n + \mathcal{O}(n)$ *comparisons.*

Standard Heapsort

Proof:

Correctness: After build-heap(n), $A[1]$ is the maximal element of the array.

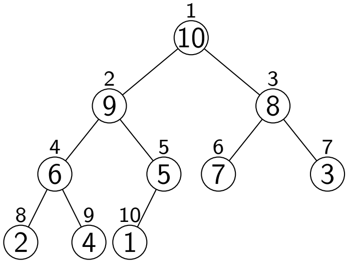
This element will be moved with swap$(1, n)$ to its correct position (n).

By induction, the subarray $A[1, \ldots, n-1]$ will be sorted in the remaining steps.

Running time: Building the heap needs $\mathcal{O}(n)$ comparison. Each of the remaining $n-1$ many reheap-calls needs at most $2 \log_2 n$ comparisons. $\square$

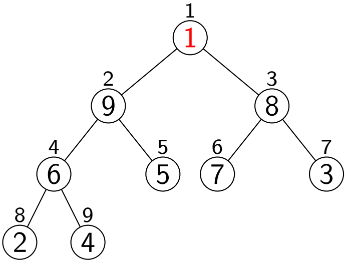
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10



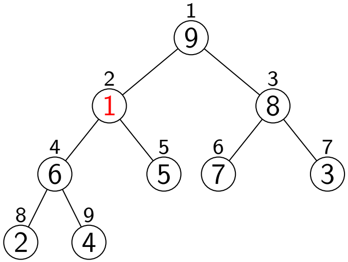
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
									10



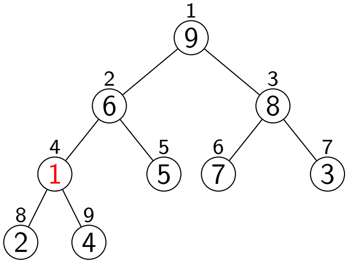
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
									10



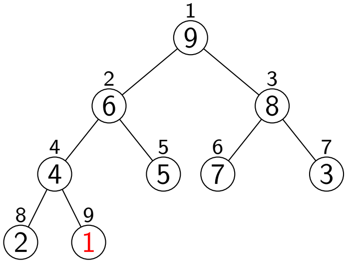
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
									10



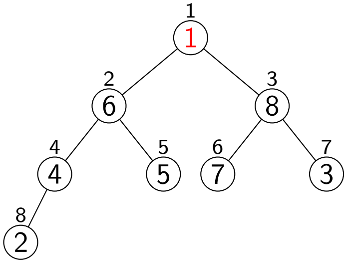
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
									10



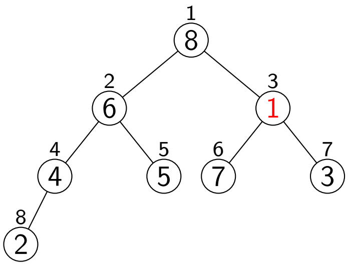
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
								9	10



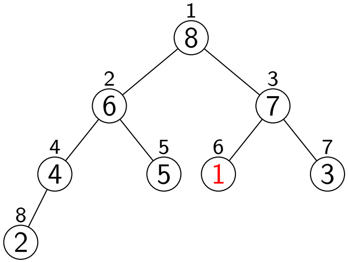
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
								9	10

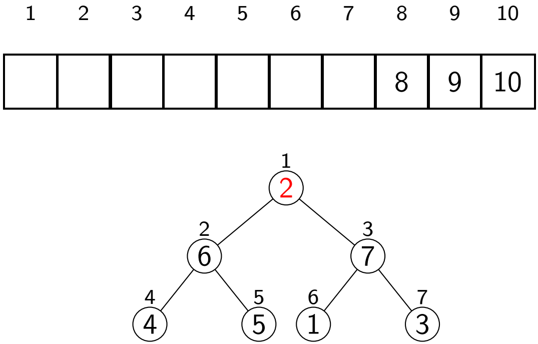


Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
								9	10

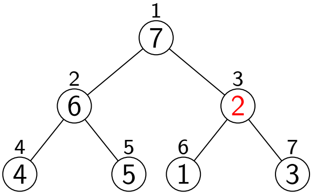


Example for Standard Heapsort



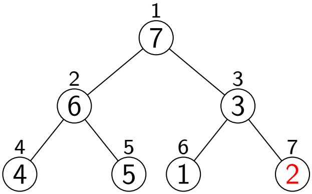
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
							8	9	10



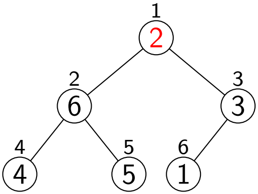
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
							8	9	10

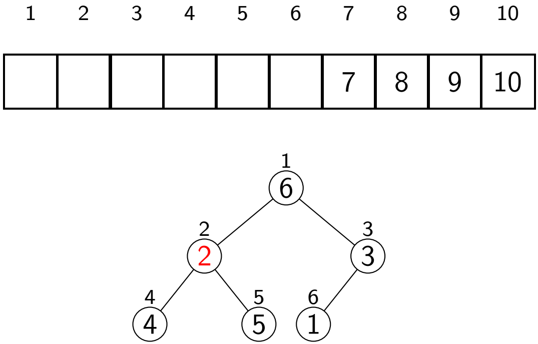


Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
						7	8	9	10

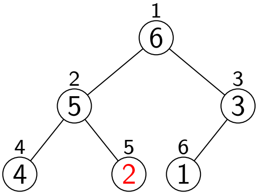


Example for Standard Heapsort



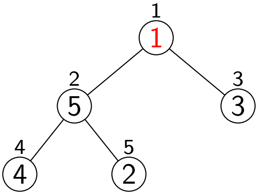
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
						7	8	9	10



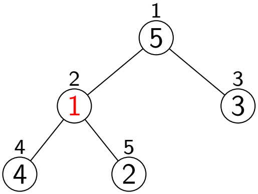
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
					6	7	8	9	10

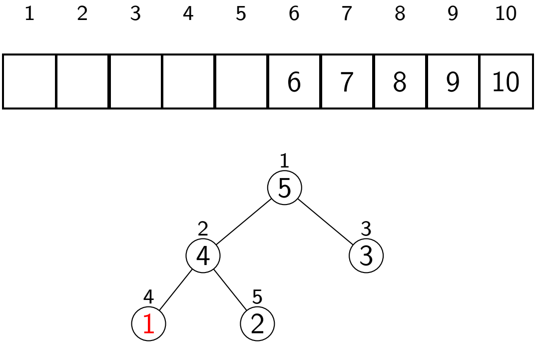


Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
					6	7	8	9	10

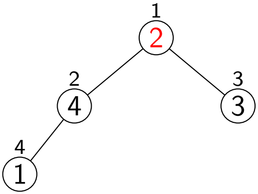


Example for Standard Heapsort



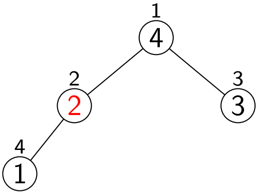
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
				5	6	7	8	9	10



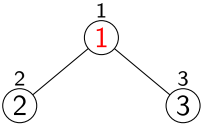
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
				5	6	7	8	9	10



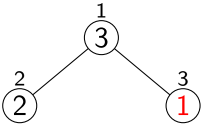
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
			4	5	6	7	8	9	10



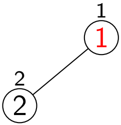
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
			4	5	6	7	8	9	10



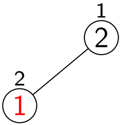
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
		3	4	5	6	7	8	9	10



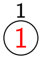
Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
		3	4	5	6	7	8	9	10



Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
	2	3	4	5	6	7	8	9	10



Example for Standard Heapsort

1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

Bottom-Up Heapsort

Remark: An analysis of the average case complexity of Heapsort yields $2n \log_2 n$ many comparisons in the average. Hence, standard Heapsort cannot compete with Quicksort.

Bottom-up Heapsort needs significantly fewer comparisons.

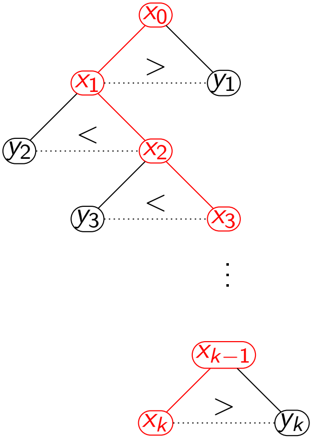
After $\mathrm{swap}(1, i)$ one first determines the potential path from the root to a leaf along which the elemente $A[i]$ will sink; the sink path.

For this, one follows the path that always goes to the larger child. This needs at most $\log n$ instead of $2 \log_2 n$ comparisons.

In most cases, $A[i]$ will sink deep into the heap. It is therefore more efficient to compute the actual position of $A[i]$ on the sink path bottom-up.

The hope is that the bottom-up computations need in total only $\mathcal{O}(n)$ comparisons.

The sink path



Elements will sink along the path $[x_0, x_1, x_2, \ldots, x_{k-1}, x_k]$ which can be computed with only $\log_2 n$ comparisons.

Finding the right position on the sink path

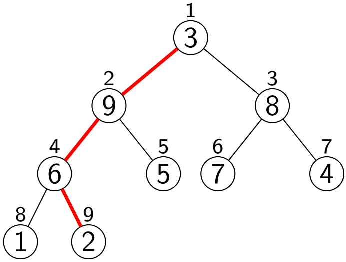
We now compute the correct position p on the sink path starting from the leaf and going up.

If this position p is found, then all elements $x_0, \ldots, x_p$ have to be rotated cyclically (x_0 goes to the position of x_p, and every $x_1, \ldots, x_p$ moves up one position).

Finding the right position on the sink path

We now compute the correct position p on the sink path starting from the leaf and going up.

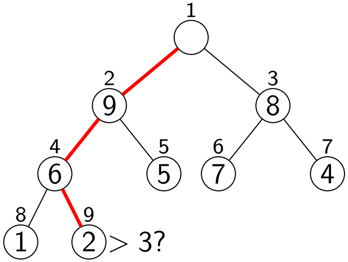
If this position p is found, then all elements $x_0, \ldots, x_p$ have to be rotated cyclically (x_0 goes to the position of x_p, and every $x_1, \ldots, x_p$ moves up one position).



Finding the right position on the sink path

We now compute the correct position p on the sink path starting from the leaf and going up.

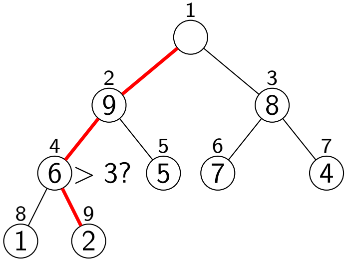
If this position p is found, then all elements $x_0, \ldots, x_p$ have to be rotated cyclically (x_0 goes to the position of x_p, and every $x_1, \ldots, x_p$ moves up one position).



Finding the right position on the sink path

We now compute the correct position p on the sink path starting from the leaf and going up.

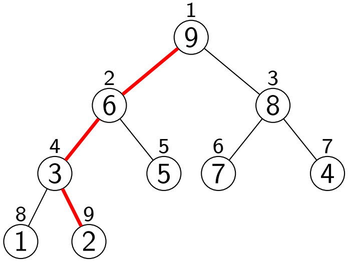
If this position p is found, then all elements $x_0, \ldots, x_p$ have to be rotated cyclically (x_0 goes to the position of x_p, and every $x_1, \ldots, x_p$ moves up one position).



Finding the right position on the sink path

We now compute the correct position p on the sink path starting from the leaf and going up.

If this position p is found, then all elements $x_0, \ldots, x_p$ have to be rotated cyclically (x_0 goes to the position of x_p, and every $x_1, \ldots, x_p$ moves up one position).



Average case analysis of Heapsort

Theorem 11

Standard heapsort makes on at least $(1-2^{-(n-1)})n!$ *many input permutations of* $[1,\dots,n]$ *at least* $2n\log_2(n)-\Theta(n)$ *comparisons.*

Bottom-up heapsort makes on at least $(1-2^{-(n-1)})n!$ *many input permutations of* $[1,\dots,n]$ *at most* $n\log_2(n)+\Theta(n)$ *comparisons.*

Proof: information-theoretic argument

A sorting algorithm computes from a permutation of $[1,\dots,n]$ the sorted list $[1,\dots,n]$.

One can specify (or encode) the input permutation by running the algorithm and in addition output information in form of a $\{0,1\}$-string that allows us to run the algorithm backwards starting with the output permutation $[1,\dots,n]$.

Average case analysis of Heapsort

In the case of standard heapsort: we output the sink paths, i.e., every time an element is swapped with the left (resp., right) child, we output a 0 (resp., 1). This makes heapsort reversible.

But: We have to know when one sink path (a $\{0,1\}$-string) stops and the next sink path starts.

Alternative 1: We encode a string $w = a_1 a_2 \cdots a_{t-1} a_t \in \{0,1\}^*$ by

$$c_1(w) = a_1 0 a_2 0 \cdots a_{t-1} 0 a_t 1.$$

Note: $|c_1(w)| = 2|w|$.

Alternative 2: We encode a string $w = a_1 a_2 \cdots a_{t-1} a_t \in \{0,1\}^*$ by

$$c_2(w) = c_1(\text{binary representation of } t) a_1 \cdots a_t$$

Thus, $|c_2(w)| = |w| + 2\log_2(|w|)$.

Average case analysis of Heapsort

Example:

- $c_1(0110) = 00101001$
- $c_2(0110) = c_1(100)0110 = 1000010110$

Note: For the empty word ε we have

$$c_2(\varepsilon) = c_1(0)\varepsilon = 01,$$

since $0 =$ binary representation of the number 0.

Average case analysis of Heapsort

Let $w = a_1 a_2 \cdots a_t \in \{0,1\}^*$ be a sink path of length t.

For phase 1 (built heap) we encode w by $c_1(w)$.

Our proof showing that building the heap only needs $\mathcal{O}(n)$ many comparisons also shows: In phase 1, we will output a $\{0,1\}$-string of length $\mathcal{O}(n)$.

For phase 2 we encode the sink path w by

$$c_2'(w) = c_1(\text{binary representation of } \log_2(n) - t) a_1 \cdots a_t.$$

Note: $t \leq \log_2(n)$, because every sink path has length $\leq \log_2 n$.

We now analyse the $\{0,1\}$-string produced in phase 2.

Average case analysis of Heapsort

Let $t_1, \ldots, t_n$ be the lengths of the sink paths during phase 2.

Hence, we produce in phase 2 a $\{0,1\}$-string of length

$$\sum_{i=1}^{n}(t_i + 2\log_2(\log_2(n) - t_i)) = \sum_{i=1}^{n} t_i \; + \; 2\sum_{i=1}^{n}\log_2(\log_2(n) - t_i)).$$

Define the average

$$\overline{t} = \frac{\sum_{i=1}^{n} t_i}{n}.$$

The function f with $f(x) = \log_2(\log_2(n) - x)$ is concave on $(-\infty, \log_2(n))$.

Jensen's inequality (slide 8) implies:

$$\log_2(\log_2(n) - \overline{t}) \geq \sum_{i=1}^{n} \frac{1}{n} \cdot \log_2(\log_2(n) - t_i)).$$

Average case analysis of Heapsort

Therefore:

$$\sum_{i=1}^{n} t_i \; + \; 2\sum_{i=1}^{n} \log_2(\log_2(n) - t_i)) \quad \leq \quad n\bar{t} + 2n\log_2(\log_2(n) - \bar{t}).$$

To sum up: The input permutation σ on $[1, \ldots, n]$ can be encoded by a $\{0,1\}$-string of length

$$I(\sigma) \leq cn + n\bar{t} + 2n\log_2(\log_2(n) - \bar{t}),$$

where c is a constant (for phase 1).

Lemma 6 implies

$$cn + n\bar{t} + 2n\log_2(\log_2(n) - \bar{t}) \geq I(\sigma) \geq \log_2(n!) - n \geq n\log_2(n) - 2.443n$$

for at least $(1 - 2^{-n+1})n!$ many input permutations.

With $d = 2.443 + c$ we get:

$$\bar{t} \geq \log_2(n) - 2\log_2(\log_2(n) - \bar{t}) - d. \tag{1}$$

Average case analysis of Heapsort

Since $\bar{t} \geq 0$ we obtain

$$\bar{t} \geq \log_2(n) - 2\log_2(\log_2(n)) - d. \tag{2}$$

From (1) and (2) we get the better estimate

$$\bar{t} \geq \log_2(n) - 2\log_2(2\log_2(\log_2(n)) + d) - d. \tag{3}$$

This estimate can be again applied to (1), and so on.

In general, we get for all $i \geq 1$:

$$\bar{t} \geq \log_2(n) - \alpha_i - d,$$

where $\alpha_1 = 2\log_2(\log_2(n))$ and $\alpha_{i+1} = 2\log_2(\alpha_i + d)$.

Average case analysis of Heapsort

We prove this statement by induction on $i \geq 1$.

$i = 1$: $\bar{t} \geq \log_2(n) - 2\log_2(\log_2(n)) - d = \log_2(n) - \alpha_1 - d$ holds by (2).

$i \geq 1$. Assume that $\bar{t} \geq \log_2(n) - \alpha_i - d$ holds.

We get

$$\begin{aligned}
\bar{t} &\overset{(1)}{\geq} \log_2(n) - 2\log_2(\log_2(n) - \bar{t}) - d \\
&\geq \log_2(n) - 2\log_2(\log_2(n) - (\log_2(n) - \alpha_i - d)) - d \\
&= \log_2(n) - 2\log_2(\alpha_i + d) - d \\
&= \log_2(n) - \alpha_{i+1} - d
\end{aligned}$$

Average case analysis of Heapsort

For all $x \geq \max\{10, d\}$ we have:

$$2\log_2(x+d) \leq 2\log_2(2x) = 2\log_2(x) + 2 \leq 0,9 \cdot x.$$

Hence, as long as $\alpha_i \geq \max\{10, d\}$ holds, we have $\alpha_{i+1} \leq 0,9 \cdot \alpha_i$.

Therefore, there exists a constant $\alpha > 0$ with

$$\bar{t} \geq \log_2(n) - \alpha - d. \tag{4}$$

Thus, for at least $(1 - 2^{-n+1})n!$ many input permutations we have

$$\sum_{i=1}^{n} t_i \geq n \log_2 n - \Theta(n). \tag{5}$$

We can now prove Theorem 11:

Average case analysis of Heapsort

The number of comparisons in phase 2 (after build heap) is

- $2\sum_{i=1}^{n} t_i$ for standard Heapsort,
- $\leq n\log_2(n) + \sum_{i=1}^{n}(\log_2(n) - t_i) = 2n\log_2(n) - \sum_{i=1}^{n} t_i$ for bottom-up Heapsort.

Hence, by (5) there are at least $(1 - 2^{-n+1})n!$ input permutations for which the number of comparisons in phase 1 and phase 2 is

- $\Theta(n) + 2\sum_{i=1}^{n} t_i \geq 2n\log_2 n - \Theta(n)$ for standard Heapsort,
- $\leq \Theta(n) + 2n\log_2(n) - \sum_{i=1}^{n} t_i \leq n\log_2(n) + \Theta(n)$ for bottom-up Heapsort.

Variant by Svante Carlsson, 1986

One can show that bottom-up Heapsort makes in the worst case at most $1.5 n \log n + \mathcal{O}(n)$ many comparisons.

Carlsson proposed to determine the correct position on the sink path using binary search.

This yields a worst-case bound of $n \log n + \mathcal{O}(n \log \log n)$ many comparison.

On the other hand, in practice binary search on the sink path does not seem to pay off.

Sorting in linear time: Counting Sort

Recall: The lower bound of $n \cdot \log_2(n) - 1.433n$ only holds for comparison-based sorting algorithms.

If we make further assumptions on the array elements, we can sort in time $\mathcal{O}(n)$.

Assumption: The array elements $A[1], \ldots, A[n]$ are natural numbers in the range $[0, k]$.

Counting sort (see next slide) sorts under this assumption in time $\mathcal{O}(k + n)$.

Hence, if $k \in \mathcal{O}(n)$, then counting sort works in linear time.

Counting Sort

Algorithm Counting Sort

procedure counting-sort(array $A[1,n]$ with $A[1],\dots A[n] \in [0,k]$)
begin
  var Arrays $C[0,k]$, $B[1,n]$
  for $i := 0$ **to** k **do**
    $C[i] := 0$
  for $i := 1$ **to** n **do**
    $C[A[i]] := C[A[i]] + 1$
  for $i := 1$ **to** k **do**
    $C[i] := C[i] + C[i-1]$
  for $i := n$ **downto** 1 **do**
    $B[C[A[i]]] := A[i]$;
    $C[A[i]] := C[A[i]] - 1$
endprocedure

Counting Sort

After the first three for-loops, $C[i] =$ number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

Array C after third **for**-loop:

	0	1	2	3	4	5
C	0	1	4	6	9	10

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$B[C[A[10]]] := A[10]$

	0	1	2	3	4	5
C	0	1	4	6	9	10

	1	2	3	4	5	6	7	8	9	10
B						3				

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$C[A[10]] := C[A[10]] - 1$

	0	1	2	3	4	5
C	0	1	4	5	9	10

	1	2	3	4	5	6	7	8	9	10
B						3				

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$B[C[A[9]]] := A[9]$

	0	1	2	3	4	5
C	0	1	4	5	9	10

	1	2	3	4	5	6	7	8	9	10
B				2		3				

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$C[A[9]] := C[A[9]] - 1$

	0	1	2	3	4	5
C	0	1	3	5	9	10

	1	2	3	4	5	6	7	8	9	10
B				2		3				

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$B[C[A[8]]] := A[8]$

	0	1	2	3	4	5
C	0	1	3	5	9	10

	1	2	3	4	5	6	7	8	9	10
B				2		3			4	

Counting Sort

After the first three for-loops, $C[i] =$ number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$C[A[8]] := C[A[8]] - 1$

	0	1	2	3	4	5
C	0	1	3	5	8	10

	1	2	3	4	5	6	7	8	9	10
B				2		3			4	

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$B[C[A[7]]] := A[7]$

	0	1	2	3	4	5
C	0	1	3	5	8	10

	1	2	3	4	5	6	7	8	9	10
B				2		3			4	5

Counting Sort

After the first three for-loops, $C[i] =$ number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$C[A[7]] := C[A[7]] - 1$

	0	1	2	3	4	5
C	0	1	3	5	8	9

	1	2	3	4	5	6	7	8	9	10
B				2		3			4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$B[C[A[6]]] := A[6]$

	0	1	2	3	4	5
C	0	1	3	5	8	9

	1	2	3	4	5	6	7	8	9	10
B			2	2		3			4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$C[A[6]] := C[A[6]] - 1$

	0	1	2	3	4	5
C	0	1	2	5	8	9

	1	2	3	4	5	6	7	8	9	10
B			2	2		3			4	5

Counting Sort

After the first three for-loops, $C[i] =$ number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$B[C[A[5]]] := A[5]$

	0	1	2	3	4	5
C	0	1	2	5	8	9

	1	2	3	4	5	6	7	8	9	10
B	1		2	2		3			4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$C[A[5]] := C[A[5]] - 1$

	0	1	2	3	4	5
C	0	0	2	5	8	9

	1	2	3	4	5	6	7	8	9	10
B	1		2	2		3			4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$B[C[A[4]]] := A[4]$

	0	1	2	3	4	5
C	0	0	2	5	8	9

	1	2	3	4	5	6	7	8	9	10
B	1		2	2		3		4	4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$C[A[4]] := C[A[4]] - 1$

	0	1	2	3	4	5
C	0	0	2	5	7	9

	1	2	3	4	5	6	7	8	9	10
B	1		2	2		3		4	4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$B[C[A[3]]] := A[3]$

	0	1	2	3	4	5
C	0	0	2	5	7	9

	1	2	3	4	5	6	7	8	9	10
B	1		2	2	3	3		4	4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$C[A[3]] := C[A[3]] - 1$

	0	1	2	3	4	5
C	0	0	2	4	7	9

	1	2	3	4	5	6	7	8	9	10
B	1		2	2	3	3		4	4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$B[C[A[2]]] := A[2]$

	0	1	2	3	4	5
C	0	0	2	4	7	9

	1	2	3	4	5	6	7	8	9	10
B	1	2	2	2	3	3		4	4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$C[A[2]] := C[A[2]] - 1$

	0	1	2	3	4	5
C	0	0	1	4	7	9

	1	2	3	4	5	6	7	8	9	10
B	1	2	2	2	3	3		4	4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$B[C[A[1]]] := A[1]$

	0	1	2	3	4	5
C	0	0	1	4	7	9

	1	2	3	4	5	6	7	8	9	10
B	1	2	2	2	3	3	4	4	4	5

Counting Sort

After the first three for-loops, $C[i]$ = number of array entries that are $\leq i$.

The statement $B[C[A[i]]] := A[i]$ puts the array element $A[i]$ at the right position $C[A[i]]$.

Example:

	1	2	3	4	5	6	7	8	9	10
A	4	2	3	4	1	2	5	4	2	3

$C[A[1]] := C[A[1]] - 1$

	0	1	2	3	4	5
C	0	0	1	4	6	9

	1	2	3	4	5	6	7	8	9	10
B	1	2	2	2	3	3	4	4	4	5

Counting Sort

Remark: Counting sort is a stable sorting algorithm.

This means: If $A[i] = A[j]$ for $i < j$, then in the sorted array B the array entry $A[i]$ is to the left of $A[j]$.

This is relevant if the array entries consist of (i) keys that are used for sorting and (ii) additional informations.

Example:

- Imagine a database with employees.
- With each employee several data are associated: first name, last name, ID, year of birth, etc.
- You want to sort the database by year of birth but for employees with the same year of birth you want to keep the initial order.

Stability of counting sort will be needed for radix sort on the next slide.

Radix Sort

We use counting sort to sort an array $A[1, n]$, where $A[1], \dots, A[n]$ are d-ary numbers in base k.

Radix sort sorts such an array in time $\mathcal{O}(d(n+k))$.

If in addition $d \in \mathcal{O}(1)$ and $k \in \mathcal{O}(n)$ (which means that we can represent number of size $\mathcal{O}(n^d)$), then radix sort works in linear time.

Algorithm Radix Sort

---

procedure radix sort(array $A[1, n]$ with $A[1], \dots, A[n]$)
begin
  for $i := 1$ **to** d **do**
    sort the array A with counting sort with respect to the i-th digit
    (where the first digit is the least significant digit)
  endfor
endprocedure

Radix Sort

Example: We sort the list

$$[5923, 8221, 6723, 3736, 1341, 7943, 3298, 6915, 2832]$$

with radix sort.

Radix Sort

Example: We sort the list

$$[5923, 8221, 6723, 3736, 1341, 7943, 3298, 6915, 2832]$$

with radix sort.

5	9	2	3
8	2	2	1
6	7	2	3
3	7	3	6
1	3	4	1
7	9	4	3
3	2	9	8
6	9	1	5
2	8	3	2

Radix Sort

Example: We sort the list

$$[5923, 8221, 6723, 3736, 1341, 7943, 3298, 6915, 2832]$$

with radix sort.

5	9	2	3		8	2	2	1
8	2	2	1		1	3	4	1
6	7	2	3		2	8	3	2
3	7	3	6		5	9	2	3
1	3	4	1		6	7	2	3
7	9	4	3		7	9	4	3
3	2	9	8		6	9	1	5
6	9	1	5		3	7	3	6
2	8	3	2		3	2	9	8

Radix Sort

Example: We sort the list

$$[5923, 8221, 6723, 3736, 1341, 7943, 3298, 6915, 2832]$$

with radix sort.

5	9	2	3		8	2	2	1
8	2	2	1		1	3	4	1
6	7	2	3		2	8	3	2
3	7	3	6		5	9	2	3
1	3	4	1		6	7	2	3
7	9	4	3		7	9	4	3
3	2	9	8		6	9	1	5
6	9	1	5		3	7	3	6
2	8	3	2		3	2	9	8

Radix Sort

Example: We sort the list

$$[5923, 8221, 6723, 3736, 1341, 7943, 3298, 6915, 2832]$$

with radix sort.

5	9	2	3	8	2	2	1	6	9	1	5
8	2	2	1	1	3	4	1	8	2	2	1
6	7	2	3	2	8	3	2	5	9	2	3
3	7	3	6	5	9	2	3	6	7	2	3
1	3	4	1	6	7	2	3	2	8	3	2
7	9	4	3	7	9	4	3	3	7	3	6
3	2	9	8	6	9	1	5	1	3	4	1
6	9	1	5	3	7	3	6	7	9	4	3
2	8	3	2	3	2	9	8	3	2	9	8

Radix Sort

Example: We sort the list

$$[5923, 8221, 6723, 3736, 1341, 7943, 3298, 6915, 2832]$$

with radix sort.

5	9	2	3	8	2	2	1	6	9	1	5
8	2	2	1	1	3	4	1	8	2	2	1
6	7	2	3	2	8	3	2	5	9	2	3
3	7	3	6	5	9	2	3	6	7	2	3
1	3	4	1	6	7	2	3	2	8	3	2
7	9	4	3	7	9	4	3	3	7	3	6
3	2	9	8	6	9	1	5	1	3	4	1
6	9	1	5	3	7	3	6	7	9	4	3
2	8	3	2	3	2	9	8	3	2	9	8

Radix Sort

Example: We sort the list

$$[5923, 8221, 6723, 3736, 1341, 7943, 3298, 6915, 2832]$$

with radix sort.

5	9	2	3		8	2	2	1		6	9	1	5		8	2	2	1
8	2	2	1		1	3	4	1		8	2	2	1		3	2	9	8
6	7	2	3		2	8	3	2		5	9	2	3		1	3	4	1
3	7	3	6		5	9	2	3		6	7	2	3		6	7	2	3
1	3	4	1		6	7	2	3		2	8	3	2		3	7	3	6
7	9	4	3		7	9	4	3		3	7	3	6		2	8	3	2
3	2	9	8		6	9	1	5		1	3	4	1		6	9	1	5
6	9	1	5		3	7	3	6		7	9	4	3		5	9	2	3
2	8	3	2		3	2	9	8		3	2	9	8		7	9	4	3

Radix Sort

Example: We sort the list

$$[5923, 8221, 6723, 3736, 1341, 7943, 3298, 6915, 2832]$$

with radix sort.

5	9	2	3		8	2	2	1		6	9	1	5		8	2	2	1
8	2	2	1		1	3	4	1		8	2	2	1		3	2	9	8
6	7	2	3		2	8	3	2		5	9	2	3		1	3	4	1
3	7	3	6		5	9	2	3		6	7	2	3		6	7	2	3
1	3	4	1		6	7	2	3		2	8	3	2		3	7	3	6
7	9	4	3		7	9	4	3		3	7	3	6		2	8	3	2
3	2	9	8		6	9	1	5		1	3	4	1		6	9	1	5
6	9	1	5		3	7	3	6		7	9	4	3		5	9	2	3
2	8	3	2		3	2	9	8		3	2	9	8		7	9	4	3

Radix Sort

Example: We sort the list

$$[5923, 8221, 6723, 3736, 1341, 7943, 3298, 6915, 2832]$$

with radix sort.

5	9	2	3		8	2	2	1		6	9	1	5		8	2	2	1		1	3	4	1
8	2	2	1		1	3	4	1		8	2	2	1		3	2	9	8		2	8	3	2
6	7	2	3		2	8	3	2		5	9	2	3		1	3	4	1		3	2	9	8
3	7	3	6		5	9	2	3		6	7	2	3		6	7	2	3		3	7	3	6
1	3	4	1		6	7	2	3		2	8	3	2		3	7	3	6		5	9	2	3
7	9	4	3		7	9	4	3		3	7	3	6		2	8	3	2		6	7	2	3
3	2	9	8		6	9	1	5		1	3	4	1		6	9	1	5		6	9	1	5
6	9	1	5		3	7	3	6		7	9	4	3		5	9	2	3		7	9	4	3
2	8	3	2		3	2	9	8		3	2	9	8		7	9	4	3		8	2	2	1

Bucket Sort

Assume that we want to sort a list $a_1, a_2, \ldots, a_n$ of n real numbers from the interval $(0, 1] = \{a \in \mathbb{R} \mid 0 < a \leq 1\}$.

The number $a_1, \ldots, a_n$ are randomly and independently chosen from $(0, 1]$.

This means that for all $b, c \in \mathbb{R}$ with $0 < b < c \leq 1$ and all $1 \leq i < j \leq n$ we have:

- $\text{Prob}\big[a_i \in [b, c]\big] = \frac{1}{c-b}$ and
- $\text{Prob}\big[a_i \in [b, c] \text{ and } a_j \in [b, c]\big] = \frac{1}{(c-b)^2}$.

We want to sort the list $a_1, \ldots, a_n$.

Idea of bucket sort:

- divide the interval $(0, 1]$ into n intervals of length $1/n$ (the buckets),
- store all a_i from the k-th bucket in a list $B[k]$,
- sort the lists $B[1], \ldots, B[n]$ and concatenate them.

Bucket Sort

Algorithm Bucket Sort

procedure bucket sort (list of numbers $a_1, \ldots, a_n \in (0, 1]$)
begin
  var Array $B[1, n]$
  for $i := 1$ **to** n **do**
    $B[i] :=$ empty list
  endfor
  for $i := 1$ **to** n **do**
    insert a_i into the list $B[\lceil a_i \cdot n \rceil]$
  endfor
  for $i := 1$ **to** n **do**
    sort the list $B[i]$ (for instance, using quicksort)
  endfor
  append the lists $B[1], B[2], \ldots, B[n]$ to a single list
endprocedure

Bucket Sort

Theorem 12

On average, bucket sort needs time $\mathcal{O}(n)$.

Proof:

Let n_k be the length of the list $B[k]$ after all elements a_i have been put into their buckets.

The running time of bucket sort is then $\mathcal{O}(n + \sum_{k=1}^{n} n_k^2)$.

The expected value for the running time is therefore (by linearity of expectation)

$$\mathrm{E}[\mathcal{O}(n + \sum_{k=1}^{n} n_k^2)] = \mathcal{O}(n + \sum_{k=1}^{n} \mathrm{E}[n_k^2])$$

We claim that $\mathrm{E}[n_k^2] = 2 - 1/n$, which implies $\sum_{k=1}^{n} \mathrm{E}[n_k^2] = 2n - 1$.

Bucket Sort

Let us now show $E[n_k^2] = 2 - 1/n$.

Define

$$X_{k,i} = \begin{cases} 1 & \text{if } \lceil a_i \cdot n \rceil = k \\ 0 & \text{otherwise} \end{cases}$$

We then have $n_k = \sum_{i=1}^{n} X_{k,i}$ and therefore

$$\begin{aligned} E[n_k^2] &= E\Bigg[\bigg(\sum_{i=1}^{n} X_{k,i}\bigg)^2\Bigg] \\ &= E\Bigg[\sum_{i=1}^{n}\sum_{j=1}^{n} X_{k,i}X_{k,j}\Bigg] \\ &= E\Bigg[\sum_{i=1}^{n} X_{k,i}^2 + \sum_{i=1}^{n} \sum_{\substack{1 \le j \le n \\ i \neq j}} X_{k,i}X_{k,j}\Bigg] \end{aligned}$$

Bucket Sort

$$
\begin{aligned}
&= \mathrm{E}\left[\sum_{i=1}^{n} X_{k,i}^2 + \sum_{i=1}^{n} \sum_{\substack{1\le j\le n\\ i\neq j}} X_{k,i}X_{k,j}\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_{k,i}^2] + \sum_{i=1}^{n} \sum_{\substack{1\le j\le n\\ i\neq j}} \mathrm{E}[X_{k,i}X_{k,j}] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_{k,i}] + \sum_{i=1}^{n} \sum_{\substack{1\le j\le n\\ i\neq j}} \mathrm{E}[X_{k,i}] \cdot \mathrm{E}[X_{k,j}] \\
&= \sum_{i=1}^{n} \frac{1}{n} + \sum_{i=1}^{n} \sum_{\substack{1\le j\le n\\ i\neq j}} \frac{1}{n^2} \\
&= 1 + \frac{n(n-1)}{n^2} = 1 + \frac{(n-1)}{n} = 2 - \frac{1}{n}
\end{aligned}
$$

Computation of the Median

Input: array $A[1, \ldots, n]$ of numbers and $1 \leq k \leq n$.

Output: k-th smallest element, i.e., the number $m \in \{A[i] \mid 1 \leq i \leq n\}$ such that

$$|\{i \mid A[i] < m\}| \leq k - 1 \quad \text{and} \quad |\{i \mid A[i] > m\}| \leq n - k$$

The median ist obtained for $k = \lceil n/2 \rceil$.

Naive approach:

- sort the array A in time $\mathcal{O}(n \log n)$,
- output the k-th element of the sorted array.

Median of the medians

Goal: Compute the k-th smallest element in linear time.

Idea: Compute a pivot element (as in quick sort) as the median of the medians of blocks of length 5.

- We split the array in blocks of length 5.
- For each block we compute the median (6 comparisons are sufficient).
- Compute recursively the median P of the array of medians and take P as the pivot element.

  Number of comparisons: $T(\frac{n}{5})$.

Quick sort step

Partition the array with the pivot element P such that for suitable positions $m_1 < m_2$ we have:

$$\begin{array}{lcll} A[i] & < & P & \text{for } 1 \leq i \leq m_1 \\ A[i] & = & P & \text{for } m_1 < i \leq m_2 \\ A[i] & > & P & \text{für } m_2 < i \leq n \end{array}$$

Number of comparisons: $\leq n$ (actually $2n/5$ comparisons suffice here, see Slide 105).

Case distinction:

1. $k \leq m_1$: search for the k-th element recursively in $A[1], \ldots, A[m_1]$.
2. $m_1 < k \leq m_2$: return P.
3. $k > m_2$: search for the $(k - m_2)$-th element in $A[m_2 + 1], \ldots, A[n]$.

Example for median search

15	14	5	4	16	10	1	12	6	23	25	8	19	18	20	21	24	7	3	13	17	9	2	22	11

Example for median search

15	14	5	4	16	10	1	12	6	23	25	8	19	18	20	21	24	7	3	13	17	9	2	22	11

Example for median search

15	**14**	5	4	16	**10**	1	12	6	23	25	8	**19**	18	20	21	24	7	3	**13**	17	9	2	22	**11**

Example for median search

15	**14**	5	4	16	**10**	1	12	6	23	25	8	**19**	18	20	21	24	7	3	**13**	17	9	2	22	**11**

14	10	19	13	11

Example for median search

15	**14**	5	4	16	**10**	1	12	6	23	25	8	**19**	18	20	21	24	7	3	**13**	17	9	2	22	**11**

14	10	19	**13**	11

Example for median search

15	14	5	4	16	10	1	12	6	23	25	8	19	18	20	21	24	7	3	**13**	17	9	2	22	11

Example for median search

5	4	10	1	12	6	8	7	3	9	2	11	**13**	15	14	16	23	25	19	18	20	21	24	17	22

30 – 70 splitting

The choice of the pivot element P as the median of the medians (of blocks of length 5) ensures the following inequalites for m_1 and m_2:

$$\frac{3}{10}n \leq m_2 \qquad \text{and} \qquad m_1 \leq \frac{7}{10}n$$

Proof:

- There are m_2 many elements $\leq P$ and $n - m_1$ many elements $\geq P$.
- Since there are $\frac{n}{5}$ many blocks of length 5, there are at least $\frac{n}{10}$ medians of 5-blocks that are $\leq P$ as well as at least $\frac{n}{10}$ medians of 5-blocks that are $\geq P$.
- In each each 5-block with median M, there are 3 elements $\leq M$ and 3 elements $\geq M$.
- Hence there are at least $\frac{3}{10}n$ many elements $\leq P$ as well at at least $\frac{3}{10}n$ many elements $\geq P$.
- Hence, $\frac{3}{10}n \leq m_2$ and $\frac{3}{10}n \leq n - m_1$. □

Total time for median search

By the previous slide, the recursive step needs at most $T(\frac{7n}{10})$ comparisons.

$T(n)$ is the totoal number of comparisons comparisons for an array of length n.

We get the following recurrence for $T(n)$:

$$T(n) \leq T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\left\lceil \frac{7n}{10} \right\rceil\right) + \mathcal{O}(n)$$

The master theorem II gives $T(n) \in \mathcal{O}(n)$.

Estimating the constant

Why are $\frac{2n}{5}$ comparisons enough for the partitioning step?

We have to compare every array element with the pivot element P (the median of the medians of the 5-blocks).

For every median M of a 5-block we know whether $M \leq P$ or $M \geq P$ (from the computation of the median of the medians of the 5-blocks).

Assume that $M \leq P$.

In the 5-block B, of which M is the median, there are 3 elements that $\leq M$ and we determined those elements (when we computed the median of B).

Hence, we have to compare in the partitioning step only 2 elements from B with P.

An analogous argument works for the case $M \geq P$.

Hence, we need only $\frac{2n}{5}$ comparisons in the partitioning step.

Estimating the constant

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \frac{6n}{5} + \frac{2n}{5},$$

where:

- $\frac{6n}{5}$ is the number of comparisons to compute the medians of the blocks of length 5.
- $\frac{2n}{5}$ is the number of comparisons for the partitioning step.

By induction we obtain $T(n) \leq 16n$:

$T(n) \leq 16n$ is certainly true for sufficiently small n.

For "large" n we have

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \frac{6n}{5} + \frac{2n}{5} \leq \frac{16n}{5} + \frac{112n}{10} + \frac{6n}{5} + \frac{2n}{5} = 16n$$

Quick select

Quick select is a randomized algorithm for computing the median:

Algorithm

function quickselect($A[\ell \dots r]$: **array of integer**, k : **integer**) : **integer**
begin
  if $\ell = r$ **then return** $A[\ell]$
  else
    $p := \mathrm{random}(\ell, r)$;
    $m :=$ partition($A[\ell \dots r]$, p);
    $k' := (m - \ell + 1)$;
    if $k = k'$ **then return** $A[m]$
    elsif $k < k'$ **then return** quickselect($A[\ell \dots m-1]$, k)
    else return quickselect($A[m+1 \dots r]$, $k - k'$)
    endif
  endif
endfunction

Analysis of quick select

Let $Q(n)$ be the average number of comparisons made by quick select on an array with n elements.

We have:

$$Q(n) \leq (n-1) + \frac{1}{n} \sum_{i=1}^{n} Q(\max\{i-1, n-i\}),$$

where:

- $(n-1)$ is the number of comparisons for partitioning the array, and
- $Q(\max\{i-1, n-i\})$ is the (maximal) average number of comparisons for a recursive call on *one* of the two subarrays.

Here, we make the pessimistic assumption that we continue searching in the larger subarray.

Analysis of quick select

We get:

$$\begin{aligned} Q(n) &\leq (n-1) + \frac{1}{n} \sum_{i=1}^{n} Q(\max\{i-1, n-i\}) \\ &= (n-1) + \frac{1}{n} \sum_{i=0}^{n-1} Q(\max\{i, n-i-1\}) \\ &= (n-1) + \frac{1}{n} \left(\sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} Q(i) + \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} Q(i) \right) \end{aligned}$$

For the last equality note that:

$$\left\lfloor \frac{n}{2} \right\rfloor \geq \left\lceil \frac{n}{2} \right\rceil - 1 = n - \left\lfloor \frac{n}{2} \right\rfloor - 1 \text{ and } \left\lfloor \frac{n}{2} \right\rfloor - 1 < \left\lceil \frac{n}{2} \right\rceil = n - \left(\left\lfloor \frac{n}{2} \right\rfloor - 1 \right) - 1$$

Claim: $Q(n) \leq 4n$:

Analysis of quick select

Proof by induction on n: OK for $n = 1$.

Let $n \geq 2$ and let $Q(i) \leq 4i$ for all $i < n$.

Case 1: n is even.

$$
\begin{aligned}
Q(n) &\leq (n-1) + \frac{2}{n} \sum_{i=\frac{n}{2}}^{n-1} Q(i) \\
&\leq (n-1) + \frac{8}{n} \sum_{i=\frac{n}{2}}^{n-1} i \\
&= (n-1) + \frac{8}{n} \left(\frac{(n-1)n}{2} - \frac{(\frac{n}{2}-1)\frac{n}{2}}{2} \right) \\
&= (n-1) + 4 \left((n-1) - \left(\frac{n}{2} - 1 \right) \frac{1}{2} \right) \\
&= (n-1) + 4(n-1) - (n-2) = 4n - 3 \leq 4n
\end{aligned}
$$

Analysis of quick select

Case 2: n is odd.

$$
\begin{aligned}
Q(n) &\leq (n-1) + \frac{2}{n} \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} Q(i) \; + \frac{1}{n} Q\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\
&\leq (n-1) + \frac{8}{n} \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} i \; + \; 2 \\
&= (n-1) + \frac{8}{n} \cdot \left(\frac{(n-1)n}{2} - \frac{\left(\left\lceil \frac{n}{2} \right\rceil - 1\right) \left\lceil \frac{n}{2} \right\rceil}{2} \right) \; + \; 2 \\
&\leq (n-1) + \frac{8}{n} \cdot \left(\frac{(n-1)n}{2} - \frac{(\frac{n}{2} - 1)\frac{n}{2}}{2} \right) \; + \; 2 \\
&= 4n - 3 + 2 \\
&\leq 4n.
\end{aligned}
$$

Best known bounds for median search

Dor and Zwick proved in 1995 that one can find the median with $2,95n + o(n)$ many comparisons; this is still the best algorithm.

The best known lower bound was shown by Brent and John 1985: Finding the median requires $2n + o(n)$ comparisons.

Part 4: Greedy algorithms

Overview

- Matroids and the generic greedy algorithm
- Kruskal's algorithm for spanning trees
- Dijkstra's algorithm for shortest paths

Greedy algorithms

Algorithms that take in each step the locally best optimal choice are called greedy.

For some problems this yields a globally optimal solution.

Problems where greedy algorithms always find an optimal solution can be characterized via the notion of a matroid.

Matroids and optimization problems

Let E be a finite set and $U \subseteq 2^E$ a set of subsets of E.

A pair (E, U) is a subset system, if the following holds:

- $\emptyset \in U$
- If $A \subseteq B \in U$ then $A \in U$ as well.

A set $A \in U$ is maximal (with respect to $\subseteq$) if for all $B \in U$ the following holds: if $A \subseteq B$, then $A = B$.

The optimization problems associated with (E, U) is:

- Input: A weight function $w : E \to \mathbb{R}$
- Output: A maximal set $A \in U$ with $w(A) \geq w(B)$ for all maximal sets $B \in U$, where
$$w(C) = \sum_{a \in C} w(a)$$
We call A an optimal solution.

Optimization problems

In order to solve such optimization problems, one can try to use the following generic greedy algorithm:

Algorithm generic greedy algorithm

---

procedure find-optimal (subset system (E, U), $w : E \to \mathbb{R}$)
begin
  order set E by descending weights as $e_1, e_2, \ldots, e_n$ with
  $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_n)$
  $T := \emptyset$
  for $k := 1$ **to** n **do**
    if $T \cup \{e_k\} \in U$ **then** $T := T \cup \{e_k\}$
  endfor
  return (T)
endprocedure

Matroids

Note: The solution computed by the generic greedy algorithm is always a maximal subset.

Unfortunately there exist subset systems for which the generic greedy algorithm does not find an optimal solution (will be shown later).

A subset system (E, U) is a matroid, if the following property (exchange property) holds:

$$\forall A, B \in U : |A| < |B| \implies \exists x \in B \setminus A : A \cup \{x\} \in U$$

Remark: If (E, U) is a matroid, then all maximal sets in U have the same cardinality.

Example: Let E be a finite set and $k \leq |E|$. Then

$$(E, \{A \subseteq E \mid |A| \leq k\})$$

is a matroid.

Matroids

Theorem 13

Let (E, U) be a subset system. The generic greedy algorithm computes for every weight function $w : E \to \mathbb{R}$ an optimal solution if and only if (E, U) is a matroid.

Proof: First assume that (E, U) is a matroid.

Let $w : E \to \mathbb{R}$ be a weight function and without loss of generality assume that $E = \{1, 2, \ldots, n\}$ with

$$w(1) \geq w(2) \geq \cdots \geq w(n).$$

Let $T = \{i_1, \ldots, i_k\} \in U$ with $i_1 < i_2 < \cdots < i_k$ be the solution computed by the generic greedy algorithm.

Assumption: There exists a maximal set $S = \{j_1, \ldots, j_l\} \in U$ with $w(S) > w(T)$, where $j_1 < j_2 < \cdots < j_l$.

Since (E, U) is a matroid, we have $k = l$.

Matroids

Since $w(S) > w(T)$, there exists $1 \leq p \leq k$ with $w(j_p) > w(i_p)$.

Since the weights where sorted in descending order, we must have $j_p < i_p$.

We now apply the exchange property to the sets

$$A = \{i_1, \ldots, i_{p-1}\} \in U \quad \text{and} \quad B = \{j_1, \ldots, j_p\} \in U.$$

Since $|A| < |B|$, there exists an element $j_q \in B \setminus A$ with $A \cup \{j_q\} \in U$.

We get $j_q \leq j_p < i_p$ and thus $j_q \in \{1, \ldots, i_p - 1\} \setminus \{i_1, \ldots, i_{p-1}\}$.

Choose $1 \leq r \leq p$ such that $i_{r-1} < j_q < i_r$ (where we set $i_0 = 0$).

Since $A \cup \{e_{j_q}\} \in U$ we get $\{i_1, \ldots, i_{r-1}, j_q\} \in U$.

But then, the generic greedy algorithm would have added j_q to the solution T in the j_q-th iteration of the for-loop — a contradiction.

Matroids

Now assume that (E, U) is not a matroid, i.e., the exchange property does not hold.

Let $A, B \in U$ with $|A| < |B|$ such that for all $b \in B \setminus A$: $A \cup \{b\} \notin U$.

Let $r = |B|$ and hence $|A| \leq r - 1$.

Define the weight function $w : E \to \mathbb{R}$ as follows:

$$w(x) = \begin{cases} r+1 & \text{for } x \in A \\ r & \text{for } x \in B \setminus A \\ 0 & \text{otherwise} \end{cases}$$

The generic greedy algorithm must compute a solution T with $A \subseteq T$ and $T \cap (B \setminus A) = \emptyset$.

We get $w(T) = (r+1) \cdot |A| \leq (r+1)(r-1) = r^2 - 1$.

Let $S \in U$ be a maximal subset with $B \subseteq S$.

Since $w(x) \geq 0$ for all x, we get $w(S) \geq w(B) \geq r^2$. □

Spanning trees and Kruskal's algorithm

Let $G = (V, E)$ be a finite undirected graph (the set of edges E is a subset of $\binom{V}{2} = \{\{x, y\} \mid x, y \in V, x \neq y\}$ of 2-element subsets of V).

A path from $u \in V$ to $v \in V$ is a sequence of nodes $(u_1, u_2, \ldots, u_n)$ with $u_1 = u$, $u_n = v$ and $\{u_i, u_{i+1}\} \in E$ for all $1 \leq i \leq n - 1$.

G is connected, if for all $u, v \in V$ with $u \neq v$ there is a path from u to v.

A circuit is a path $(u_1, u_2, \ldots, u_n)$ with $n \geq 3$, $u_i \neq u_j$ for all $1 \leq i < j \leq n$ and $\{u_n, u_1\} \in E$.

G is a tree, if it is connected and has no circuits.

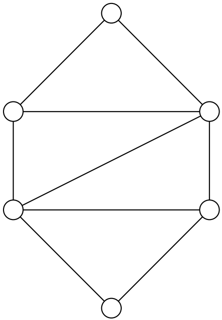
Excercise: For every tree $T = (V, E)$ we have $|E| = |V| - 1$. Every graph $G = (V, E)$ with at least $|V|$ edges has a circuit.

Spanning subtrees

Let $G = (V, E)$ be a connected graph. A spanning subtree of G is a subset $F \subseteq E$ of edges such that (V, F) is a tree.

Excercise: every connected graph has a spanning subtree.

Example:

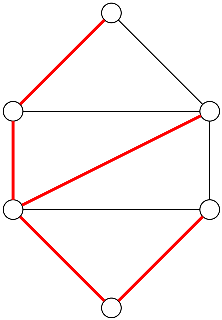


Spanning subtrees

Let $G = (V, E)$ be a connected graph. A spanning subtree of G is a subset $F \subseteq E$ of edges such that (V, F) is a tree.

Excercise: every connected graph has a spanning subtree.

Example:



Matroid of circuit-free edge sets

Let $G = (V, E)$ be again connected, and let $w : E \to \mathbb{R}$ be a weight function.

The weight of a spanning subtree $F \subseteq E$ is

$$w(F) = \sum_{e \in F} w(e).$$

Goal: Compute a spanning subtree of maximal weight.

The following lemma allows us to use the generic greedy algorithm:

Lemma 14

The subset system $(E, \{A \subseteq E \mid (V, A) \text{ has no circuit}\})$ is a matroid.

Note: Since $G = (V, E)$ is connected, the maximal subsets of the subset system $(E, \{A \subseteq E \mid (V, A) \text{ has no circuit}\})$ are the spanning subtrees.

Matroid of circuit-free edge sets

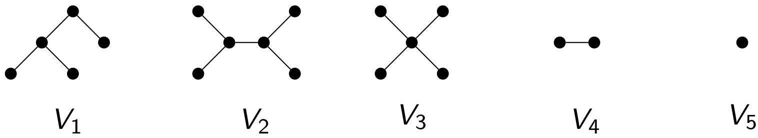
Proof: Let $A, B \subseteq E$ be edge sets without circuits such that $|A| < |B|$.

Let $V_1, V_2 \ldots, V_n$ be the connected components of the (V, A): Every graph $(V_i, A \cap \binom{V_i}{2})$ is connected and in (V, A) there is no path from a node $u \in V_i$ to a node $v \notin V_i$.

We have $|A| = \sum_{i=1}^{n}(|V_i| - 1)$, because the subgraph $(V_i, A \cap \binom{V_i}{2})$ of (V, A) induced by V_i is a tree and therefore has $|V_i| - 1$ many edges.

For every edge $e = \{u, v\} \in B$ one of the following two cases holds:

1. There is $1 \leq i \leq n$ with $u, v \in V_i$.
2. There are $i \neq j$ with $u \in V_i$ and $v \in V_j$.



Matroid of circuit-free edge sets

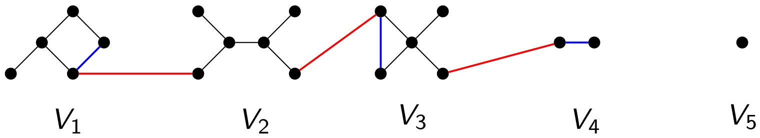
Proof: Let $A, B \subseteq E$ be edge sets without circuits such that $|A| < |B|$.

Let $V_1, V_2 \ldots, V_n$ be the connected components of the (V, A): Every graph $(V_i, A \cap \binom{V_i}{2})$ is connected and in (V, A) there is no path from a node $u \in V_i$ to a node $v \notin V_i$.

We have $|A| = \sum_{i=1}^{n}(|V_i| - 1)$, because the subgraph $(V_i, A \cap \binom{V_i}{2})$ of (V, A) induced by V_i is a tree and therefore has $|V_i| - 1$ many edges.

For every edge $e = \{u, v\} \in B$ one of the following two cases holds:

1. There is $1 \leq i \leq n$ with $u, v \in V_i$.
2. There are $i \neq j$ with $u \in V_i$ and $v \in V_j$.



Matroid of circuit-free edge sets

Assume that B contains more than $\sum_{i=1}^{n}(|V_i| - 1) = |A|$ many edges of type 1.

Then there would be an $i \in \{1, \dots, n\}$ such that B contains at least $|V_i|$ edges within V_i.

But then B would contain a circuit in V_i, which cannot be the case.

Hence: B contains $\leq \sum_{i=1}^{n}(|V_i| - 1) = |A|$ many edges of type 1.

Since $|B| > |A|$, there exists an edge $e \in B \setminus A$, which connects two connected components of (V, A).

Thus, $A \cup \{e\}$ contains no circuit. □

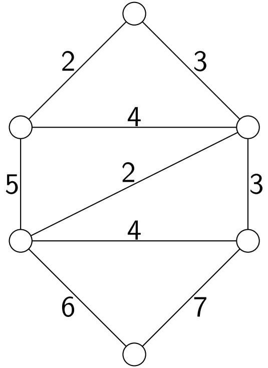
Kruskals algorithm

Algorithm Kruskals algorithm

procedure kruskal (edge-weighted connected graph (V, E, w))
begin
  sort E by decreasing weights $e_1, e_2, \ldots, e_n$ with
  $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_n)$
  $F := \emptyset$
  for $k := 1$ **to** n **do**
    if e_k connects two different connected components of (V, F) **then**
      $F := F \cup \{e_k\}$
  endfor
  return (F)
endprocedure

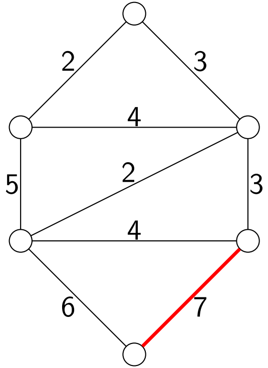
Kruskal's algorithm

Example for Kruskal's algorithm:



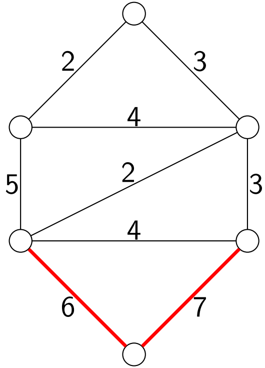
Kruskal's algorithm

Example for Kruskal's algorithm:



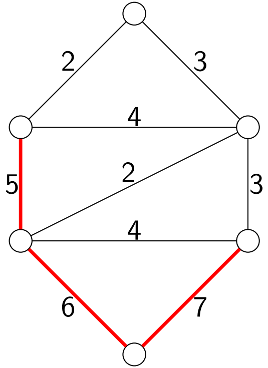
Kruskal's algorithm

Example for Kruskal's algorithm:



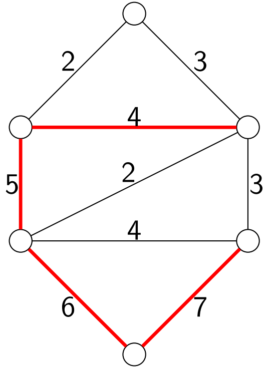
Kruskal's algorithm

Example for Kruskal's algorithm:



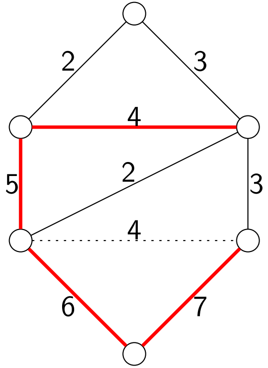
Kruskal's algorithm

Example for Kruskal's algorithm:



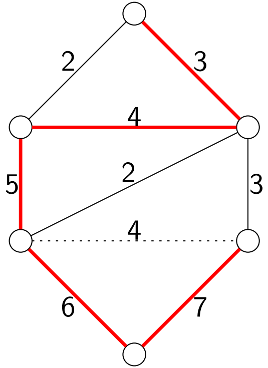
Kruskal's algorithm

Example for Kruskal's algorithm:



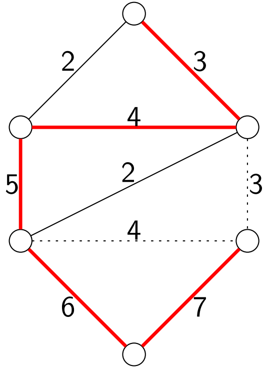
Kruskal's algorithm

Example for Kruskal's algorithm:



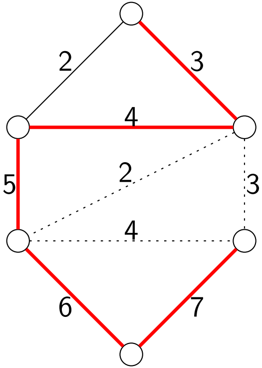
Kruskal's algorithm

Example for Kruskal's algorithm:



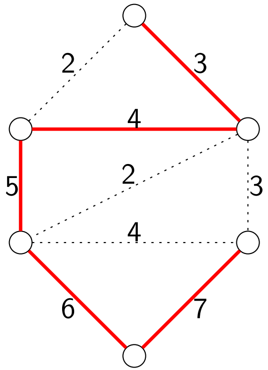
Kruskal's algorithm

Example for Kruskal's algorithm:



Kruskal's algorithm

Example for Kruskal's algorithm:



Running time of Kruskal's algorithm

Note: Since G is connected, we have $|V| - 1 \leq |E| \leq |V|^2$.

Sorting the edges by weight needs time $\mathcal{O}(|E| \log |E|) = \mathcal{O}(|E| \log |V|)$.

The connected components $V_1, V_2 \dots, V_n$ of the current graph (V, F) form a partition of V: $V = \bigcup_{i=1}^{n} V_i$, $V_i \cap V_j = \emptyset$ for $i \neq j$, $V_i \neq \emptyset$ for all i.

We start with the singleton connected components $\{v\}$ for all $v \in V$.

In every iteration of the **for**-loop ($|E|$ many) we test whether the end points of the edge e_k belong to different sets V_i, V_j $(i \neq j)$ of the partition.

If this holds, then we replace in the partition the sets V_i and V_j by the set $V_i \cup V_j$.

For this, so-called union-find data structures exist, which realizes the above operations in total time $\mathcal{O}(\alpha(|V|) \cdot |E|)$ for an extremely slow-growing function α.

This gives the running time $\mathcal{O}(|E| \log |V|)$ for Kruskal's algorithm.

Shortest paths and Dijkstra's algorithm

Another example for a greedy strategy: Computation of shortest paths in an edge-weighted directed graph $G = (V, E, \gamma)$.

- V is the set of nodes
- $E \subseteq V \times V$ is the set of edges, where $(x, x) \notin E$ for all $x \in V$.
- $\gamma : E \to \mathbb{N}$ is the weight function.

Weight of a path $(v_0, v_1, v_2, \ldots, v_n)$: $\sum_{i=0}^{n-1} \gamma(v_i, v_{i+1})$

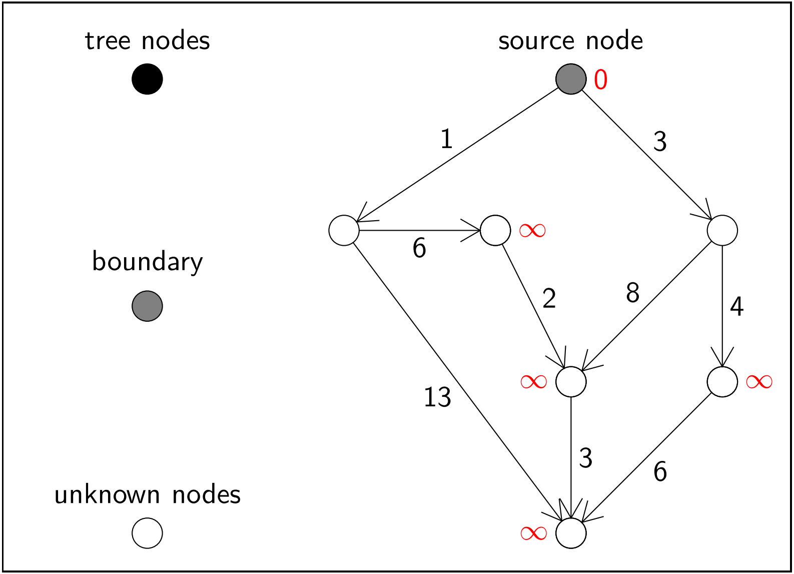
For $u, v \in V$, $d(u, v)$ denotes the minimum of the weight of all paths from u to v ($d(u, v) = \infty$ if such a path does not exist, and $d(u, u) = 0$).

Goal: Given $G = (V, E, \gamma)$ and a source node $u \in V$, compute for every $v \in V$ a path $u = v_0, v_1, v_2, \ldots, v_{n-1}, v_n = v$ with minimal weight $d(u, v)$.

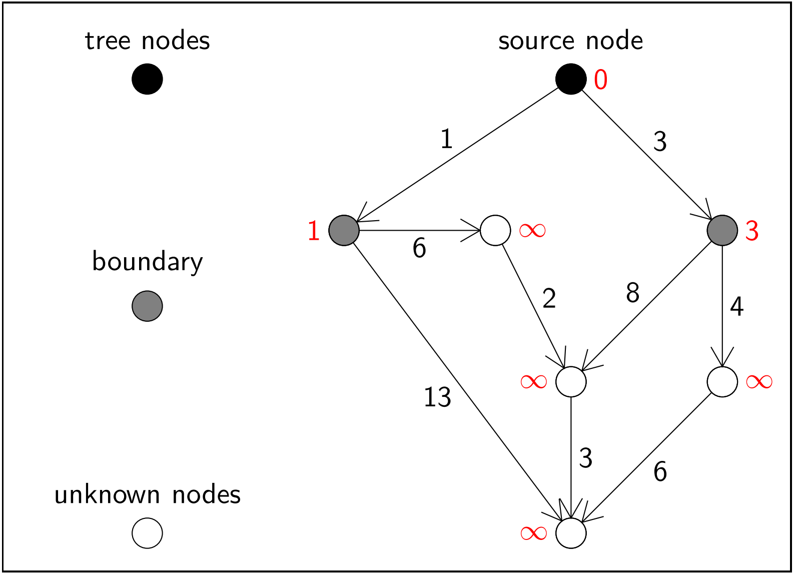
Dijkstra's algorithm

$B := \emptyset$ (tree nodes); $R := \{u\}$ (boundary); $U := V \setminus \{u\}$ (unknown nodes);
$p(u) :=$ **nil**; $D(u) := 0$;
while $R \neq \emptyset$ **do**
  $x :=$ **nil**; $\alpha := \infty$;
  forall $y \in R$ **do**
    if $D(y) < \alpha$ **then**
      $x := y$; $\alpha := D(y)$
    endif
  endfor
  $B := B \cup \{x\}$; $R := R \setminus \{x\}$
  forall $(x, y) \in E$ **do**
    if $y \in U$ **then**
      $D(y) := D(x) + \gamma(x, y)$; $p(y) := x$; $U := U \setminus \{y\}$; $R := R \cup \{y\}$
    elsif $y \in R$ **and** $D(x) + \gamma(x, y) < D(y)$ **then**
      $D(y) := D(x) + \gamma(x, y)$; $p(y) := x$
    endif
  endfor
endwhile

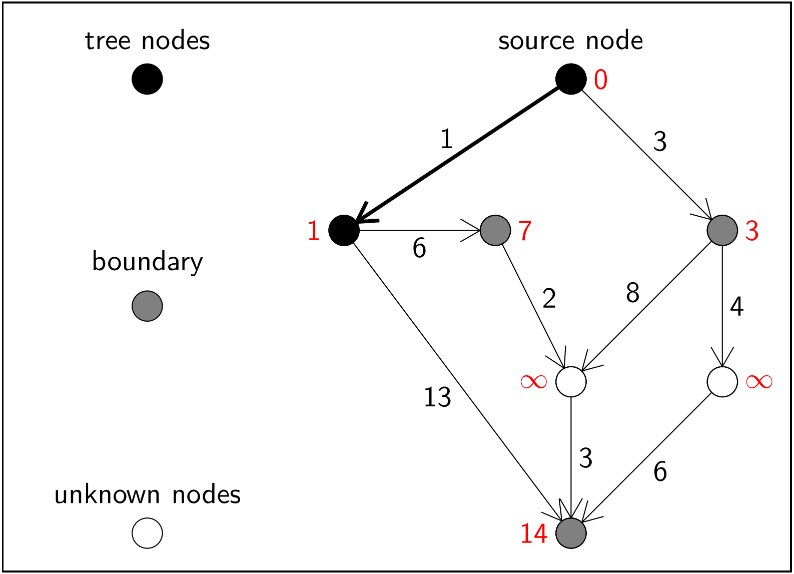
Example for Dijkstra's algorithm



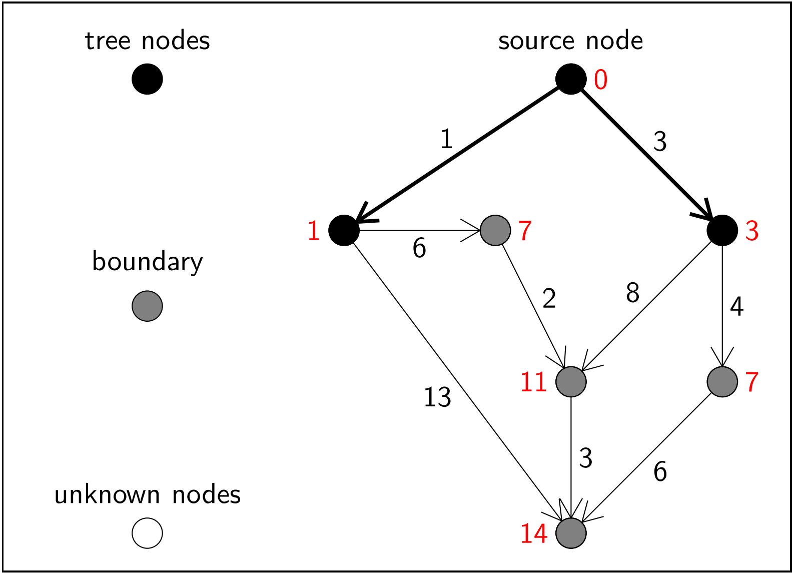
Example for Dijkstra's algorithm



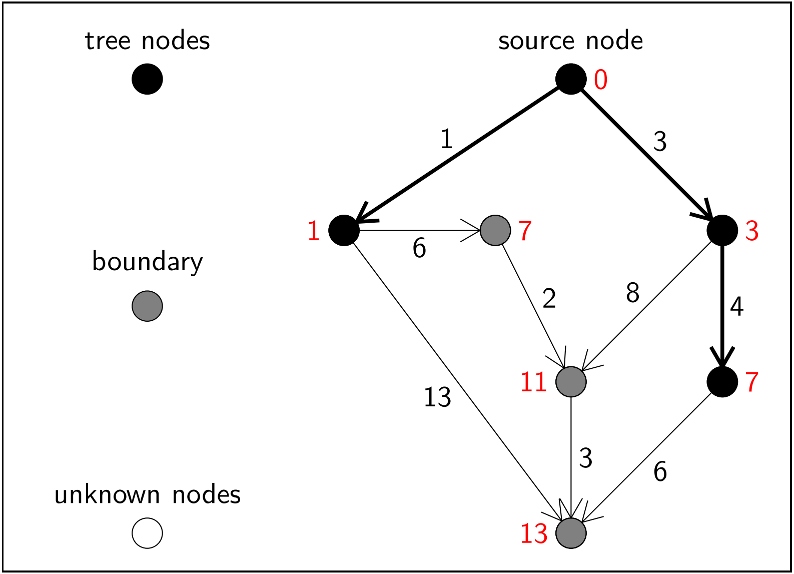
Example for Dijkstra's algorithm



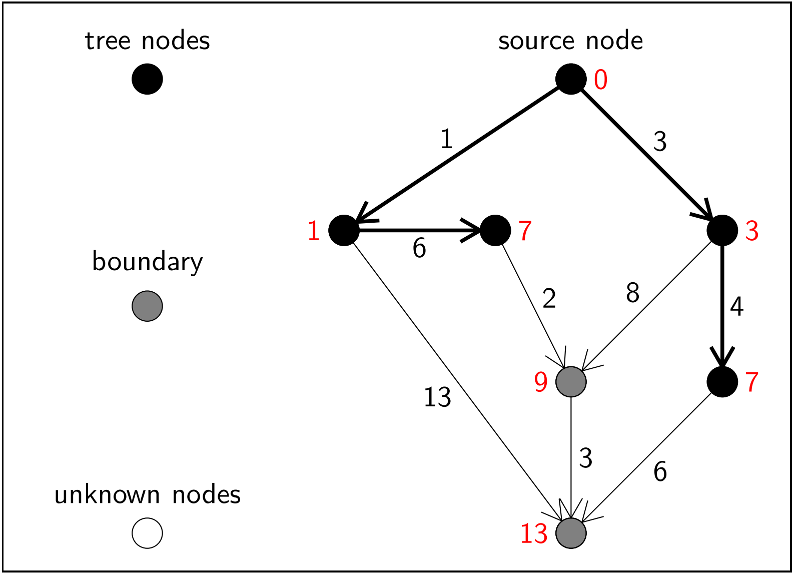
Example for Dijkstra's algorithm



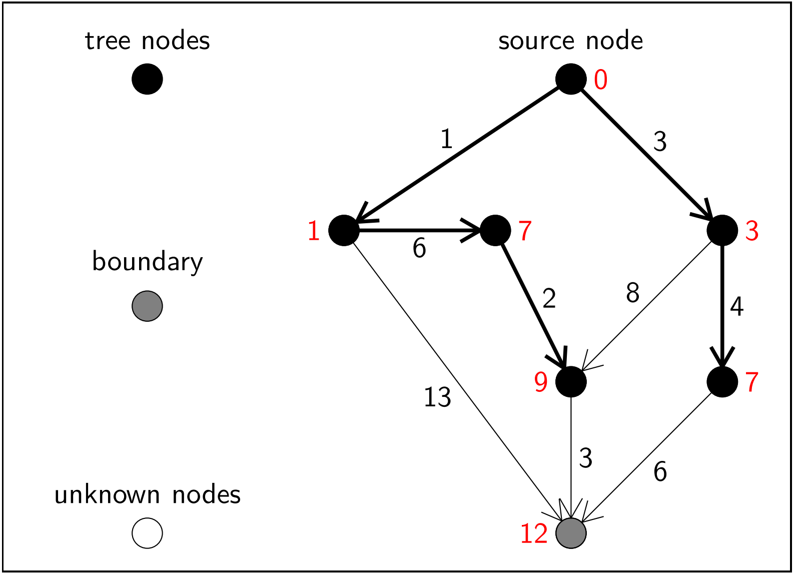
Example for Dijkstra's algorithm



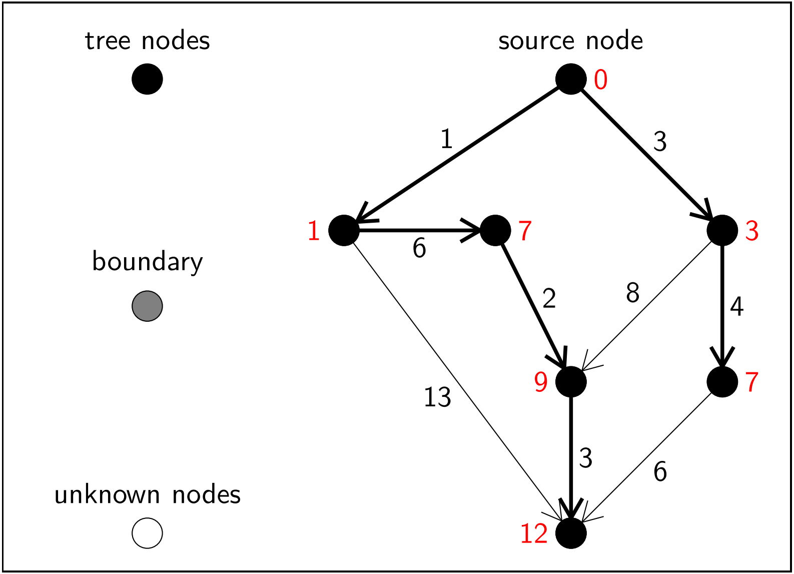
Example for Dijkstra's algorithm



Example for Dijkstra's algorithm



Example for Dijkstra's algorithm



Correctness of Dijkstra's algorithm

Theorem 15 (Correctness of Dijkstra's algorithm)

Dijkstra's algorithm computes shortest paths from the source node to all other nodes.

Proof: We show that the following invariants are preserved by the loop-body of the **while**-loop:

1. The sets B, R, and U form a partition of the node set V.
2. $R = \{y \mid \exists x \in B : (x, y) \in E\} \setminus B$
3. for all $x \in B$, $D(x) = d(u, x)$
4. for all $y \in R$, $D(y) = \min\{D(x) + \gamma(x, y) \mid x \in B, (x, y) \in E\}$

Consider an execution of the body of the **while**-Schleife, where the node x is moved from R to B.

(1)–(4) hold before the execution of the loop-body.

It is clear that (1) and (2) are preserved.

Correctness of Dijkstra's algorithm

(3): Because of (3) and (4) there exists a node $z \in B$ with

$$D(x) = D(z) + \gamma(z, x) = d(u, z) + \gamma(z, x).$$

Hence, there is path from u to x with weight $D(x)$.

Assume that there is a path from u to x with weight $< D(x)$.

Let $w \in R$ be the first node on this path, which does not belong to B (must exist since $x \notin B$) and let $v \in B$ be the predecessor of w on the path (exists, since $u \in B$).

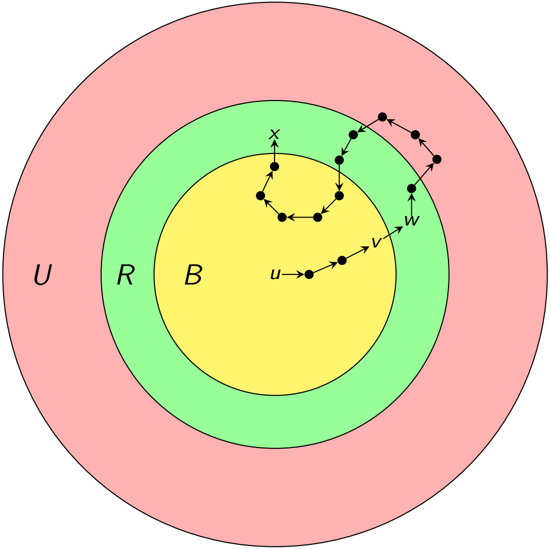
Since the whole path has weight $< D(x)$, we get

$$\begin{aligned} D(w) &= \min\{D(y) + \gamma(y, w) \mid y \in B, (y, w) \in E\} \\ &\leq D(v) + \gamma(v, w) < D(x), \end{aligned}$$

which contradicts the choice of $x \in R$.

Hence, we must have $d(u, x) = D(x)$.

Correctness of Dijkstra's algorithm



Correctness of Dijkstra's algorithm

(4): Let B', R', U', D' be the values of the variables B, R, U, D after the execution of the loop-body.

Note: $B' = B \cup \{x\}$, $D(z) = D'(z)$ for all $z \in B$ and $D(x) = D'(x)$.

Let $y \in R'$.

Case 1: $y \in R \setminus \{x\}$ and $(x, y) \in E$. We have

$$\begin{aligned} D'(y) &= \min\{D(y), D(x) + \gamma(x, y)\} \\ &= \min\{\min\{D(z) + \gamma(z, y) \mid z \in B, (z, y) \in E\}, D(x) + \gamma(x, y)\} \\ &= \min\{\min\{D'(z) + \gamma(z, y) \mid z \in B, (z, y) \in E\}, D'(x) + \gamma(x, y)\} \\ &= \min\{D'(z) + \gamma(z, y) \mid z \in B', (z, y) \in E\} \end{aligned}$$

Case 2: $y \in R \setminus \{x\}$ and $(x, y) \notin E$. We have

$$\begin{aligned} D'(y) &= D(y) \\ &= \min\{D(z) + \gamma(z, y) \mid z \in B, (z, y) \in E\} \\ &= \min\{D'(z) + \gamma(z, y) \mid z \in B', (z, y) \in E\}. \end{aligned}$$

Correctness of Dijkstra's algorithm

Case 3: $y \notin R$. We have $(x, y) \in E$, but there is no edge $(z, y) \in E$ with $z \in B$ (by invariant (2)).

Hence, we have

$$\begin{aligned} D'(y) &= D(x) + \gamma(x, y) \\ &= D'(x) + \gamma(y, x) \\ &= \min\{D'(z) + \gamma(z, y) \mid z \in B', (z, y) \in E\}, \end{aligned}$$

which concludes the proof. □

Remarks:

- One can extend our correctness proof for Dijkstra's algorithm in order to show: For every node $v \in B$, the sequence of nodes v_i with $v_0 = v$ and $v_i = p(v_{i-1})$ for $i \geq 1$ terminates in node u (say, $v_k = u$) and $(v_k, v_{k-1}, \dots, v_0)$ is a path of minimal weight from u to v.
- Dijkstra's algorithm in general does not produce a correct result if negative edge weights are allowed.

Dijkstra with abstract data types for the boundary

In order to analyze the running time of Dijkstra's algorithm, it is uselful to reformulate the algorithm with an abstract data type for the boundary R.

The following operations are needed for the boundary R:

insert	insert a new element into R.
decrease-key	decrease the key value of an element of R.
delete-min	find the element from R with the smallest key value and remove it from R.

Dijkstra with abstract data types for the boundary

$B := \emptyset$; $R := \{u\}$; $U := V \setminus \{u\}$; $p(u) :=$ **nil**; $D(u) := 0$;
while $(R \neq \emptyset)$ **do**
  $x :=$ delete-min(R);
  $B := B \cup \{x\}$;
  forall $(x, y) \in E$ **do**
    if $y \in U$ **then**
      $U := U \setminus \{y\}$; $p(y) := x$; $D(y) := D(x) + \gamma(x, y)$;
      insert$(R, y, D(y))$;
    elsif $y \in R$ **and** $D(x) + \gamma(x, y) < D(y)$ **then**
      $p(y) := x$; $D(y) := D(x) + \gamma(x, y)$;
      decrease-key$(R, y, D(y))$;
    endif
  endfor
endwhile

Running time of Dijkstra's algorithm

Number of operations ($n =$ number of nodes, $e =$ number of edges):

insert	n
decrease-key	e
delete-min	n

The total running time depends of the data structure that is used for the boundary:

1. Array of size n:
   single insert/decrease-key: $\mathcal{O}(1)$
   single delete-min: $\mathcal{O}(n)$
   total running time: $\mathcal{O}(n + e + n^2) = \mathcal{O}(n^2)$
2. Heap (balanced binary tree of depth $\mathcal{O}(\log(n))$):
   single insert/decrease-key/delete-min: $\mathcal{O}(\log(n))$
   total running time: $\mathcal{O}(n \log(n) + e \log(n)) = \mathcal{O}(e \log(n))$.

   If $\mathcal{O}(e) \subseteq o(n^2/\log n)$, then the heap beats the array.
   For instance, for planar graphs one has $e \leq 3n - 6$ for $n \geq 3$.

Fibonacci heaps (Fredman & Tarjan 1984)

Fibonacci heaps beat arrays as well as heaps: $\mathcal{O}(e + n \log n)$

A Fibonacci heap H is a list of rooted trees, i.e., a forest.

V is the set of nodes

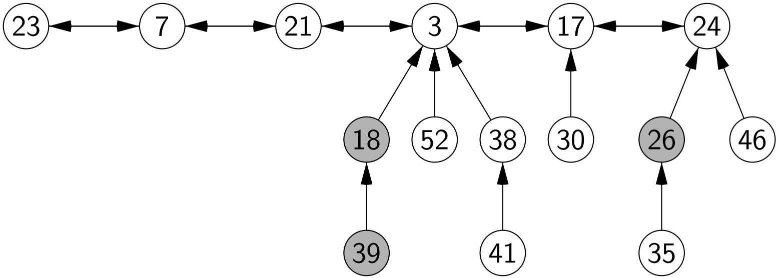
Every node $v \in V$ has a key $key(v) \in \mathbb{N}$.

Heap condition: $\forall x \in V : \; y \text{ is a child of } x \; \Rightarrow key(x) \leq key(y)$

Some of the nodes of V are marked. The root of a tree is never marked.

Example for a Fibonacci heap

(key values are in the circles, marked nodes are grey)



Fibonacci heaps

- The parent-child relation has to be realized by pointers, since the trees in a Fibonacci heap are not necessarily balanced.
- That means that pointer manipulations (expensive!) replace the index manipulations (cheap!) in standard heaps.
- Operations:
  1. **merge**
  2. **insert**
  3. **delete-min**
  4. **decrease-key**

Implementation of **merge** and **insert**

- **merge**: Concatenation of two lists — constant time
- **insert**: Special case of **merge** — constant time
- **merge** and **insert** produce long lists of one-element trees.
- Every such list is a Fibonacci heap.

Implementation of **delete-min**

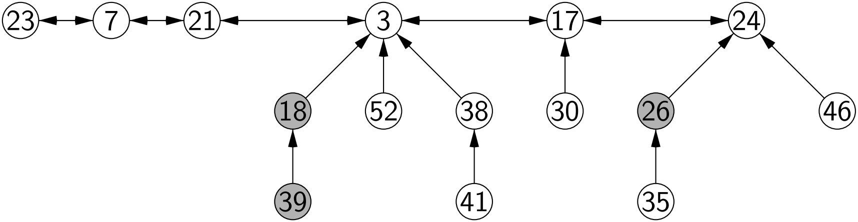
- Let H be a Fibonacci heap consisting of T trees and n nodes.
- for a nodes $x \in V$ let $\text{rank}(x)$ be the number of children of x.
- for a tree B in H let $\text{rank}(B)$ be the rank of the root of B.
- Let $r_{\max}(n)$ be the maximal rank that can appear in a Fibonacci heap with n nodes.
- Clearly, $r_{\max}(n) \leq n$. Later, we will show that $r_{\max}(n) \in \mathcal{O}(\log n)$.

Implementation of **delete-min**

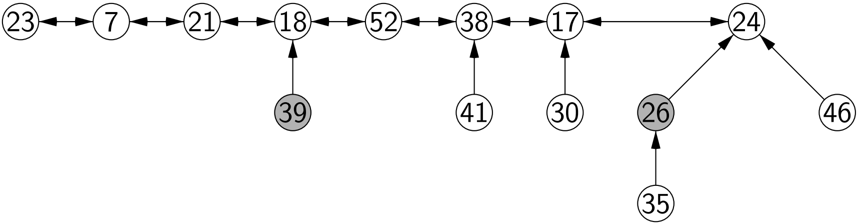
1. Search for the root x with minimal key. Time: $\mathcal{O}(T)$
2. Remove x and replace the subtree rooted in x by its rank(x) many subtrees. Remove possible markings from the new roots.
   Time: $\mathcal{O}(\text{rank}(x)) \subseteq \mathcal{O}(r_{\max}(n))$.
3. Define an array $L[0, \ldots, r_{\max}(n)]$, where $L[i]$ is a list of all trees of rank i.
   Time: $\mathcal{O}(T + r_{\max}(n))$.
4. **for** $i := 0$ **to** $r_{\max}(n) - 1$ **do**
   while $|L[i]| \geq 2$ **do**
   remove two trees from $L[i]$
   make the root with the larger key to a child of the other root
   add the resulting tree to $L[i + 1]$
   endwhile endfor

   Time: $\mathcal{O}(T + r_{\max}(n))$

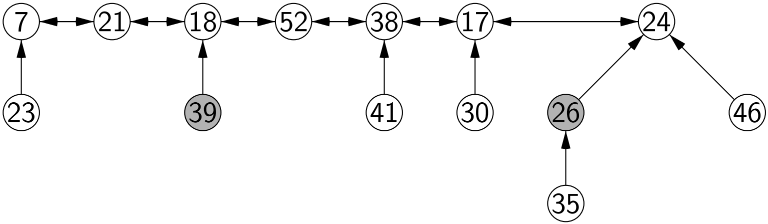
Example for **delete-min**



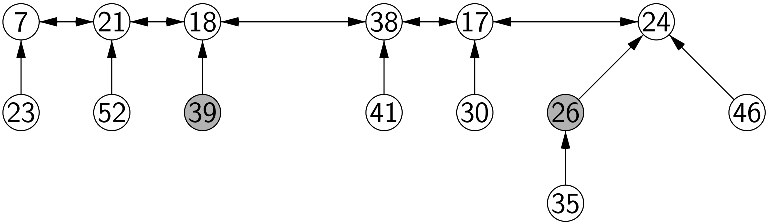
Example for **delete-min**



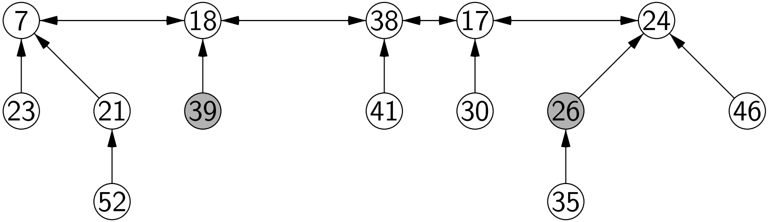
Example for **delete-min**



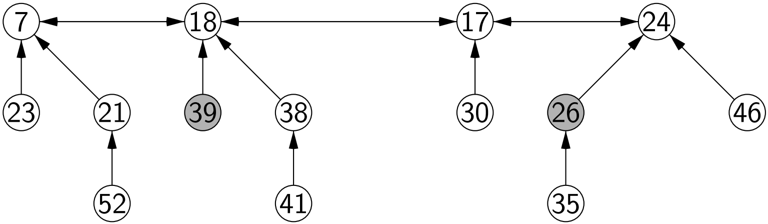
Example for **delete-min**



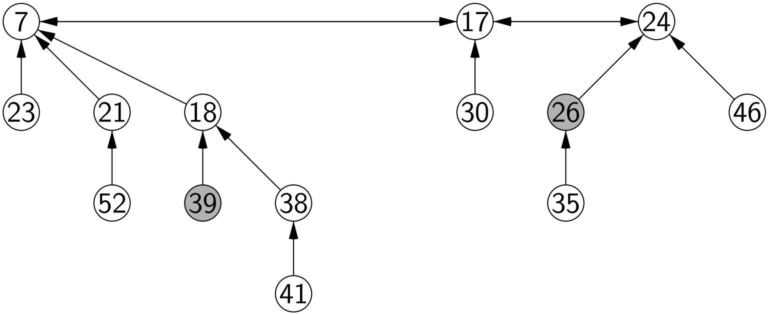
Example for **delete-min**



Example for **delete-min**



Example for **delete-min**



Remarks for **delete-min**

- **delete-min** needs time $\mathcal{O}(T + r_{\max}(n))$, where T is the number of trees before the operation.
- After the execution of **delete-min**, there exists for every $i \leq r_{\max}(n)$ at most one tree of rank i.
- Hence, the number of trees after **delete-min** is bounded by $r_{\max}(n)$.

Implementation of **decrease-key**

Let x be the node for which the key is reduced.

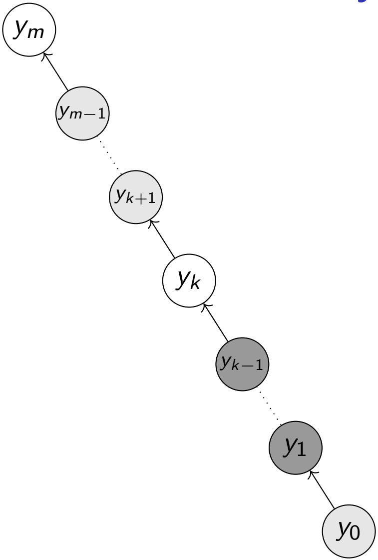
1. If x is a root, then we can reduce $key(x)$ without any other modifications.

   Now assume that x is not a root and let $x = y_0, y_1, \ldots, y_m$ be the path from x to the root y_m ($m \geq 1$).

   Let y_k ($1 \leq k \leq m$) be the first node on this path, which is not x and which is not marked (note: y_m is not marked).
2. For all $0 \leq i < k$, we cut off y_i from its parent node y_{i+1} and remove the marking from y_i ($y_0 = x$ can be marked).

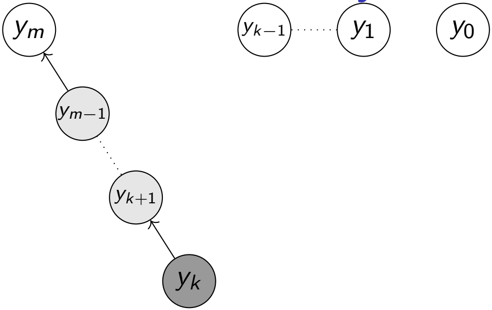
   y_i ($0 \leq i < k$) is now an unmarked root of a new tree.
3. If y_k is not a root, then we mark y_k (this tells us later that y_k lost a child).

Implementation of **decrease-key**



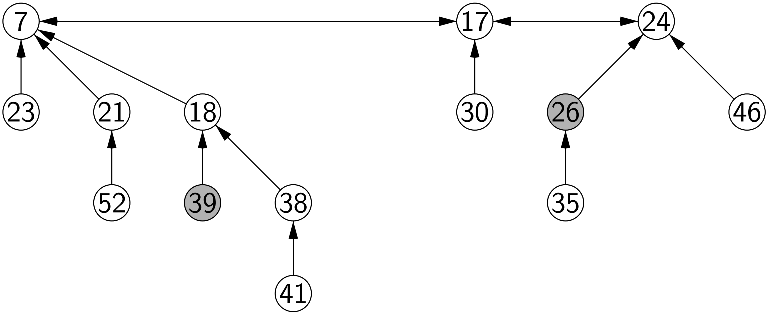
(dark gray nodes are marked, light gray nodes can be marked)

Implementation of **decrease-key**



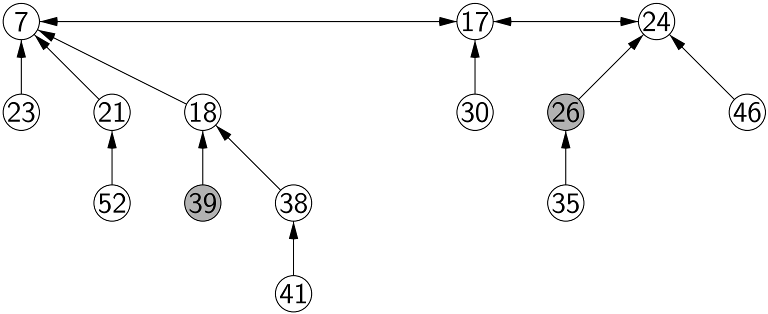
(dark gray nodes are marked, light gray nodes can be marked)

Example for **decrease-key**



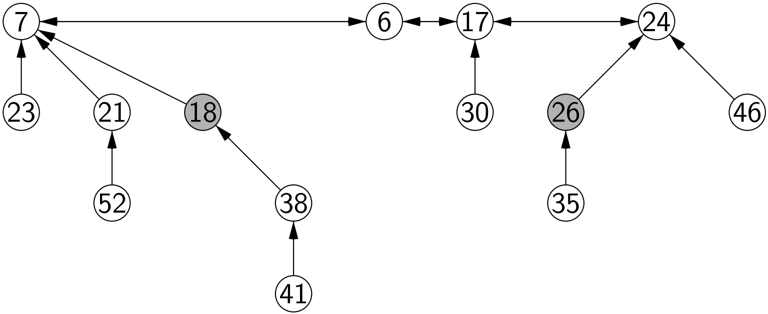
Example for **decrease-key**

decrease-key(node with key $39, 6$)



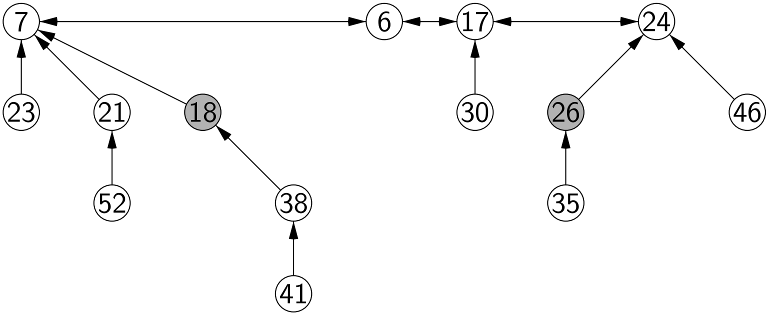
Example for **decrease-key**

decrease-key(node with key $39, 6$)



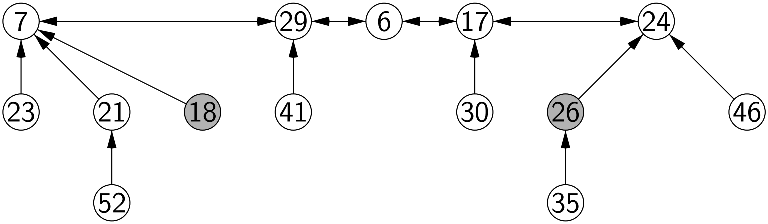
Example for **decrease-key**

decrease-key(node with key $38, 29$)



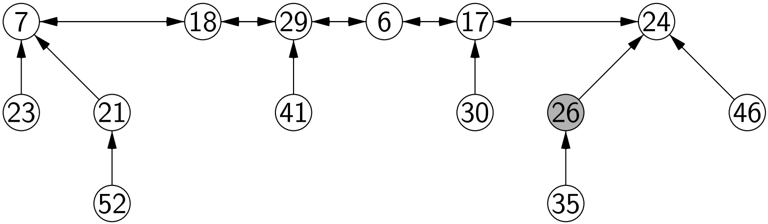
Example for **decrease-key**

decrease-key(node with key $38, 29$)



Example for **decrease-key**

decrease-key(node with key $38, 29$)



Remarks for **decrease-key**

- Time: $\mathcal{O}(k)$
- **decrease-key** reduces the number of marked nodes by at least $k - 2$ ($k \geq 1$).
- **decrease-key** increases the number of trees by k.

Definition of Fibonacci heaps

Definition (Fibonacci heap)

A Fibonacci heap is a list of rooted trees as described before, which can be obtained from the empty list by an arbitrary sequence of **merge**, **insert**, **delete-min**, and **decrease-key** operations

Lemma 16 (Fibonacci heap lemma)

Let x be a node of a Fibonacci heap with $\text{rank}(x) = k$.

1. *If $c_1, \ldots, c_k$ are the children of x, and c_i became a child of x before c_{i+1} became a child of x, then $\text{rank}(c_i) \geq i - 2$.*
2. *The subtree rooted in x contains at least F_{k+1} many nodes. Here, F_{k+1} is the $(k+1)$-th Fibonacci number ($F_0 = F_1 = 1, F_{k+1} = F_k + F_{k-1}$ for $k \geq 1$).*

Proof of the Fibonacci heap lemma

Part 1:

At the time instant t, where c_i became a child of x, the nodes $c_1, \dots, c_{i-1}$ were already children of x, i.e., the rank of x at time t was at least $i - 1$.

Since only trees with equal rank are merged to a single tree (in **delete-min**), that rank of c_i at time t was at least $i - 1$ as well.

In the meantime (i.e. after time t), c_i can loose at most one child: If c_i looses one child due to a **decrease-key**, then c_i will be marked, and after loosing second child, c_i will be cut off from the parent node x.

Hence, $\text{rank}(c_i) \geq i - 2$.

Proof of the Fibonacci heap lemma

Part 2:

Proof by induction on the height of the subtree rooted at x.

If x is a leaf, then $k = 0$ and the subtree rooted in x contains $1 = F_1$ node.

If x is not a leaf then we can count the number of nodes in the subtree rooted at x as follows:

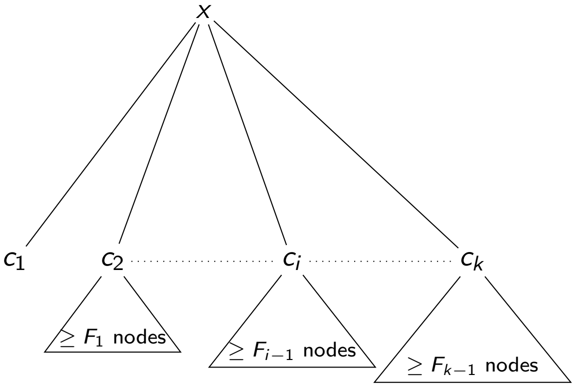
1. 2 (for x and c_1) plus
2. the number of nodes in the subtree rooted at c_i (for $2 \leq i \leq k$), which has rank $\geq i - 2$ (by part 1) and therefore contains by induction at least F_{i-1} many nodes.

Hence the subtree rooted in x contains at least

$$2 + \sum_{i=2}^{k} F_{i-1} = 2 + \sum_{i=1}^{k-1} F_i$$

many nodes.

Proof of the Fibonacci heap lemma



Proof of the Fibonacci heap lemma

The following claim concludes the proof of part 2.

Claim: $2 + \sum_{i=1}^{k-1} F_i = F_{k+1}$ for all $k \geq 1$.

Induction on $k \geq 1$:

$k = 1$: $2 + \sum_{i=1}^{k-1} F_i = 2 = F_2$

$k > 1$: By induction we get

$$2 + \sum_{i=1}^{k-1} F_i = 2 + \sum_{i=1}^{k-2} F_i + F_{k-1} = F_k + F_{k-1} = F_{k+1}$$

Growth of the Fibonacci numbers

Theorem 17

For all $k \geq 0$ we have:

$$F_k = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{k+1} - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^{k+1}$$

Asymptotically we get $F_k \approx 0,72 \cdot (1,62)^k$ (and $F_{k+1} \approx 1,17 \cdot (1,62)^k$).

If $\mathrm{rank}(x) = k$ and the Fibonacci heap has n nodes in total, then

$$n \geq \text{size of subtree rooted in } x \geq F_{k+1} \approx 1,17 \cdot (1,62)^k$$

Hence, $k \in \mathcal{O}(\log n)$.

Consequence: $r_{\max}(n) \in \mathcal{O}(\log n)$.

Summary of the running times

- **merge**, **insert**: constant time
- **delete-min**: $\mathcal{O}(T + r_{\max}(n)) \subseteq \mathcal{O}(T + \log n)$, where T is the current number of trees.
- **decrease-key**: $\mathcal{O}(k)$ ($k \geq 1$), where at least $k - 2$ markings are removed from the Fibonacci heap and k trees are added.

Amortized time

Definition (potential, amortized time)

For a Fibonacci heap H we define its potential $pot(H)$ as $pot(H) := T + 2M$, where T is its number of trees and M is the number of marked nodes.

For an operation op let $\Delta_{pot}(op)$ be the difference of the potential after and before the execution of the operation.

$$\Delta_{pot}(op) = pot(\text{heap after } op) - pot(\text{heap before } op).$$

The amortized time of the operation is op is

$$t_{amort}(op) = t(op) + \Delta_{pot}(op).$$

Amortized time

The potential has the following properties:

- $pot(H) \geq 0$
- $pot(H) \in \mathcal{O}(|H|)$
- $pot(nil) = 0$

Let $op_1, op_2, op_3, \ldots, op_m$ be sequence of m operations, and assume that the initial Fibonacci heap is empty.

For $1 \leq i \leq m$ let H_i be the Fibonacci heap after op_i.

Let H_0 be the initial Fibonacci heap (before op_1); hence $pot(H_0) = 0$.

Amortized time

We have

$$
\begin{aligned}
\sum_{i=1}^{m} t_{amort}(op_i) &= \sum_{i=1}^{m} (t(op_i) + \Delta(op_i)) \\
&= \sum_{i=1}^{m} (t(op_i) + pot(H_i) - pot(H_{i-1})) \\
&= pot(H_m) - pot(H_0) + \sum_{i=1}^{m} t(op_i) \\
&= pot(H_m) + \sum_{i=1}^{m} t(op_i) \\
&\geq \sum_{i=1}^{m} t(op_i).
\end{aligned}
$$

Hence, it suffices to bound $t_{amort}(op)$.

Amortized time

Convention: By multiplying all terms in the following computations with a suitable constant, we can assume that

- **merge** and **insert** need one time step,
- that **delete-min** needs at most $T + \log n$ time steps, and
- that **decrease-key** needs k time steps ($k \geq 1$).

This allows to omit the $\mathcal{O}$-notation.

Amortized time

- $t_{amort}(\textbf{merge}) = t(\textbf{merge}) = 1$, because the potential of the concatenation of two lists is the sum of the potentials of the two lists.
- $t_{amort}(\textbf{insert}) = t(\textbf{insert}) + \Delta_{pot}(op) = 1 + 1 = 2$.
- For **delete-min** we have $t(\textbf{delete-min}) \leq T + \log n$, where T is the number of trees before the execution of **delete-min**.

  After **delete-min**, the number of trees bounded by $r_{\max}(n)$.

  The number of marked nodes can only get smaller.

  Hence, we have $\Delta_{pot}(op) \leq r_{\max}(n) - T$ and $t_{amort}(\textbf{delete-min}) \leq T + \log n - T + r_{\max}(n) \in \mathcal{O}(\log n)$.

Amortized time

- For **decrease-key** we have $t(\textbf{decrease-key}) \leq k$ $(k \geq 1)$, where at least $k - 2$ markings will be removed.

  Moreover, k new trees are added to the Fibonacci heap.

  We get

$$\begin{aligned} \Delta_{pot}(op) &= \Delta(T) + 2\Delta(M) \\ &\leq k + 2 \cdot (2 - k) \\ &= 4 - k, \end{aligned}$$

  and hence $t_{amort}(\textbf{decrease-key}) \leq k + 4 - k = 4 \in \mathcal{O}(1)$.

Amortized time

Theorem 18

The following amortized time bounds hold for a Fibonacci heap with n nodes:

$$t_{amort}(\textbf{merge}) \in \mathcal{O}(1)$$
$$t_{amort}(\textbf{insert}) \in \mathcal{O}(1)$$
$$t_{amort}(\textbf{delete-min}) \in \mathcal{O}(\log n)$$
$$t_{amort}(\textbf{decrease-key}) \in \mathcal{O}(1)$$

Fibonacci heaps for Dijkstra

Back to Dijkstra's algorithm:

- For Dijkstra's algorithm let V be the boundary and let $key(v)$ be the current estimate for $d(u, v)$.
- Let n be the number of nodes and e be the number of edges of the input graph.
- Dijkstra's algorithm will execute at most n **insert**-, e **decrease-key**- and n **delete-min**-operations.

Fibonacci heaps for Dijkstra

$$\begin{aligned} t_{\text{Dijkstra}} \leq & \quad n \cdot t_{amort}(\textbf{insert}) \\ & + \; e \cdot t_{amort}(\textbf{decrease-key}) \\ & + \; n \cdot t_{amort}(\textbf{delete-min}) \\ \in & \; \mathcal{O}(n + e + n \log n) \\ = & \; \mathcal{O}(e + n \log n) \end{aligned}$$

Remember that:

- with arrays we got $t_{\text{Dijkstra}} \in \mathcal{O}(n^2)$, and
- with standard heaps we got $t_{\text{Dijkstra}} \in \mathcal{O}(e \log(n))$.

Part 5: Dynamic Programming

Overview

- Computing long products of (non-square) matrices
- Optimal binary search trees
- Warshall's and Floyd's algorithm

Idea of dynamic programming

Compute a table of all subsolutions of a problem, until the overall solution is computed.

Every subsolutions is computed using the already existing entries in the table.

Dynamic programming is tightly related to backtracking.

In contrast to backtracking, dynamic programming used iteration instead of recursion. By storing computed subsolutions in table we avoid to solve the same subproblem several times.

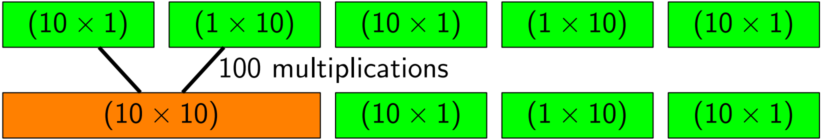
Example: Computing a long product of matrices

Multiplication from left to right:

(10×1)	(1×10)	(10×1)	(1×10)	(10×1)

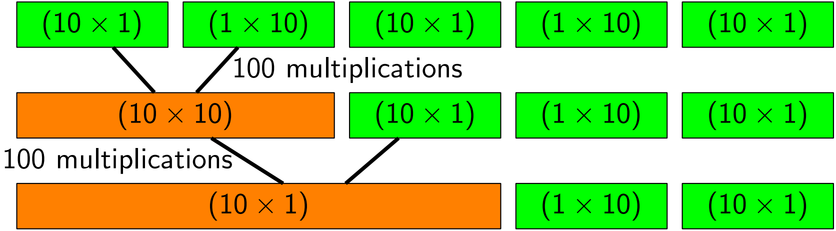
Example: Computing a long product of matrices

Multiplication from left to right:



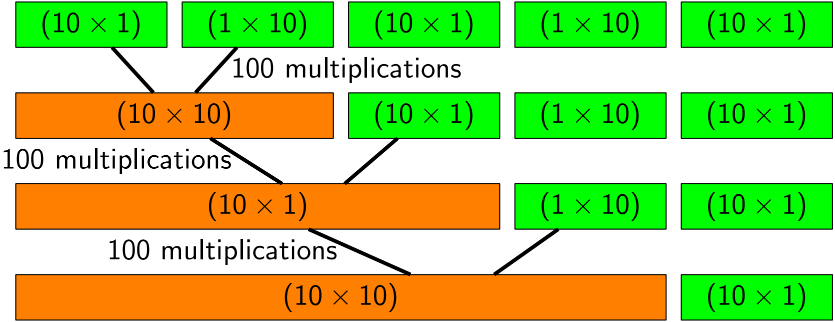
Example: Computing a long product of matrices

Multiplication from left to right:



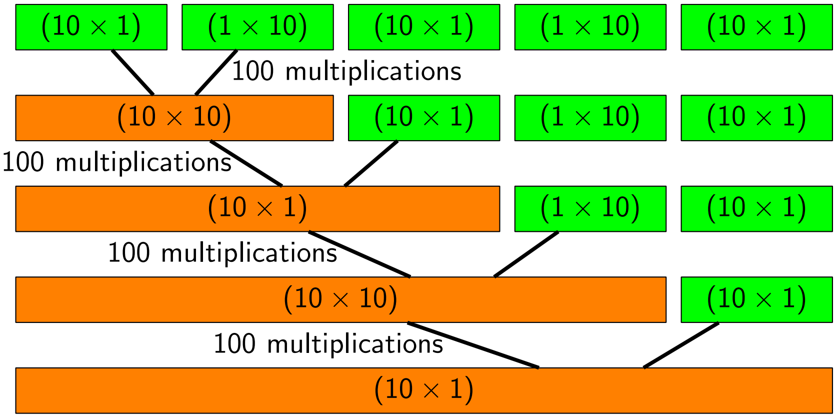
Example: Computing a long product of matrices

Multiplication from left to right:



Example: Computing a long product of matrices

Multiplication from left to right:



In total: **400** multiplications

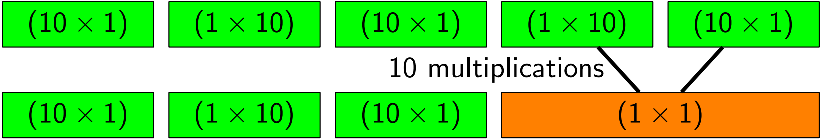
Example: Computing a long product of matrices

Multiplication from right to left:

(10×1)	(1×10)	(10×1)	(1×10)	(10×1)

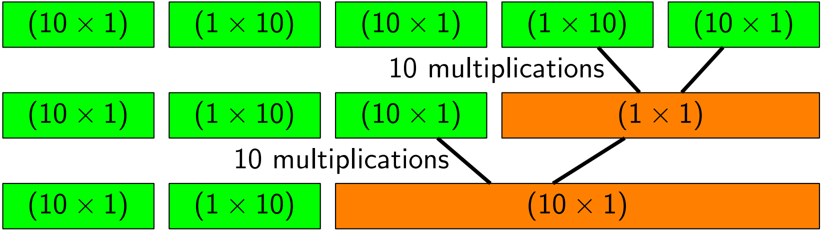
Example: Computing a long product of matrices

Multiplication from right to left:



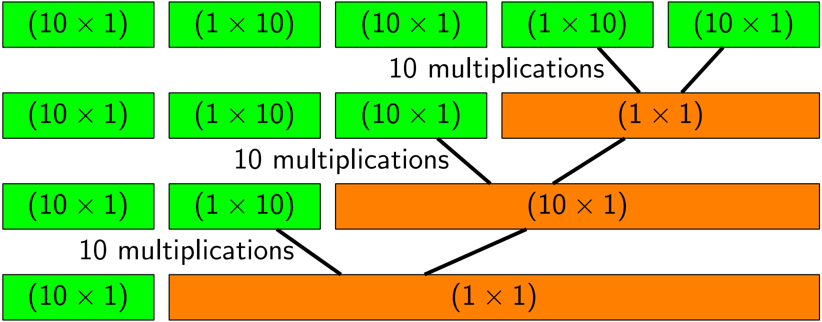
Example: Computing a long product of matrices

Multiplication from right to left:



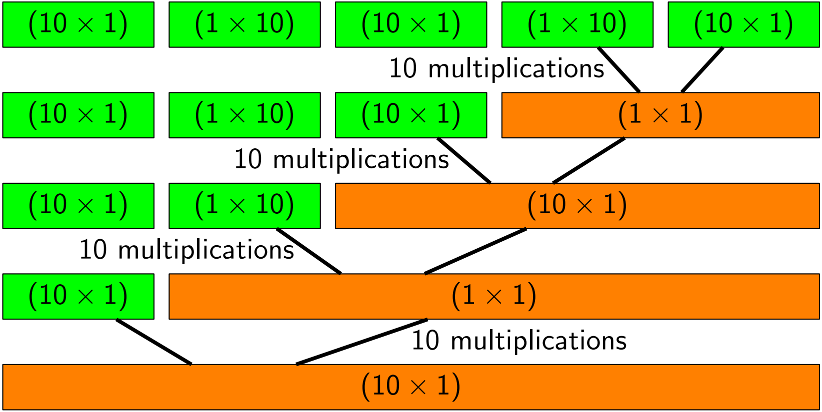
Example: Computing a long product of matrices

Multiplication from right to left:



Example: Computing a long product of matrices

Multiplication from right to left:



In total: **40** multiplications

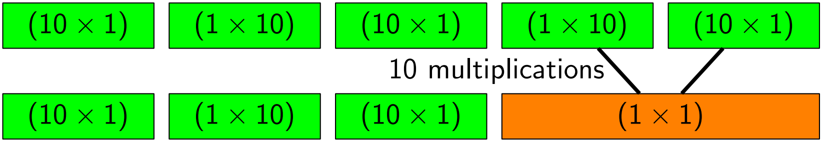
Example: Computing a long product of matrices

Multiplication in **optimal order**

(10×1)	(1×10)	(10×1)	(1×10)	(10×1)

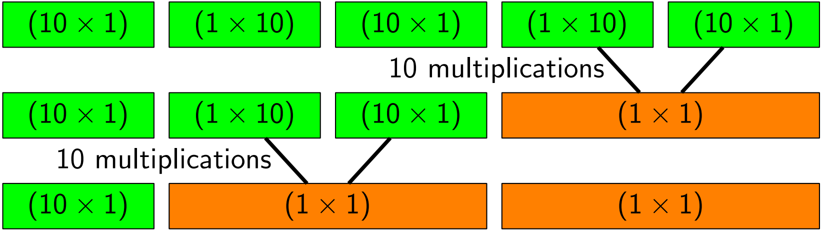
Example: Computing a long product of matrices

Multiplication in **optimal order**



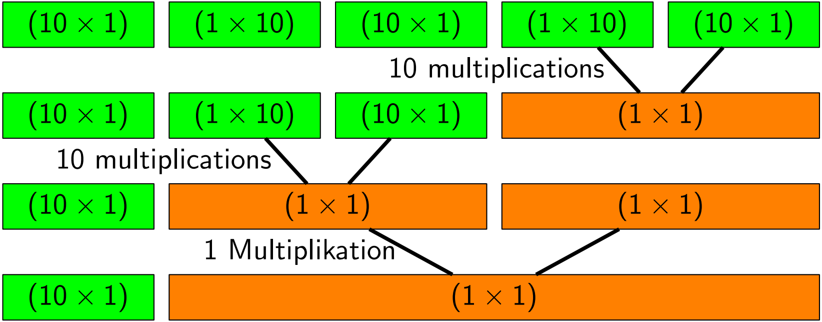
Example: Computing a long product of matrices

Multiplication in **optimal order**



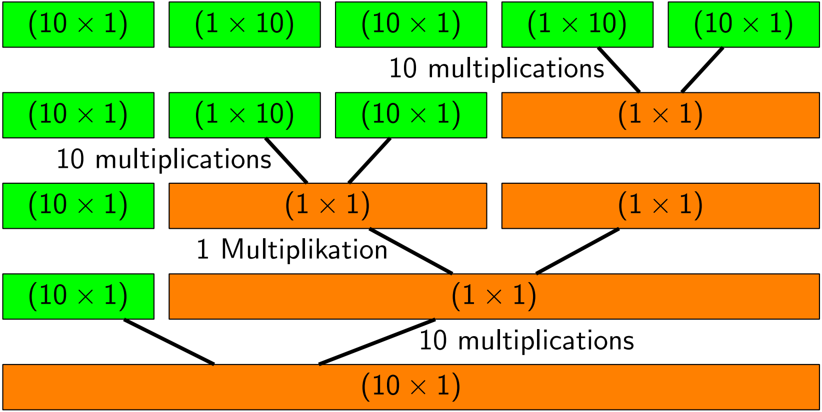
Example: Computing a long product of matrices

Multiplication in **optimal order**



Example: Computing a long product of matrices

Multiplication in **optimal order**



In total: **31** multiplications

Computing a long product of matrices

Let $\mathbb{Z}^{n \times m}$ be all matrices over $\mathbb{Z}$ with n columns and m rows.

Assumption: For $A \in \mathbb{Z}^{n \times m}$ and $B \in \mathbb{Z}^{m \times k}$, computing the product $A \cdot B$ needs $n \cdot m \cdot k$ scalar multiplications (multiplications in $\mathbb{Z}$).

Recall: matrix multiplication is associative, i.e., $A \cdot (B \cdot C) = (A \cdot B) \cdot C$.

Input: matrices $M_1, M_2, \dots, M_\ell$ with $M_i \in \mathbb{Z}^{n_{i-1} \times n_i}$.

$\text{cost}(M_1, \dots, M_\ell) :=$ minimal number of scalar multiplications needed to compute $M_1 \cdots M_\ell$ (minimum is taken over all possible bracketings).

Dynamic programming approach:

$$\begin{aligned}&\text{cost}(M_i, \dots, M_j) = \\ &\quad \min_k\{\text{cost}(M_i, \dots, M_k) + \text{cost}(M_{k+1}, \dots, M_j) + n_{i-1} \cdot n_k \cdot n_j\}\end{aligned}$$

Let $\text{cost}(M_i, \dots, M_j) = \text{cost}[i, j]$.

Computing a long product of matrices

```
for i := 1 to ℓ do
   cost[i, i] := 0;
   for j := i + 1 to ℓ do
      cost[i, j] := ∞;
   endfor
endfor
for d := 1 to ℓ − 1 do
   for i := 1 to ℓ − d do
      j := i + d;
      for k := i to j − 1 do
         t := cost[i, k] + cost[k + 1, j] + n_{i−1} · n_k · n_j;
         if t < cost[i, j] then
            cost[i, j] := t;
            best[i, j] := k;
         endif
      endfor
   endfor
endfor
return best
```

Optimal search trees

We will see a straightforward dynamic programming algorithm for computing optimal search trees with a running time of of $\Theta(n^3)$.

An algorithm of Donald E. Knuth reduces the time to $\Theta(n^2)$.

Optimal search trees

Let $V = \{v_1, \ldots, v_n\}$ be linearly ordered set of keys, $v_1 < v_2 < \cdots < v_n$.

For every key $v \in V$ we have given an access probability (also called the weight) $\gamma(v)$.

The idea is that with every key some additional information is associated (think about personnel numbers, and additional informations like name, birthday, salary, etc). Then $\gamma(v_i)$ is the probability that the information associated with key v_i is accessed.

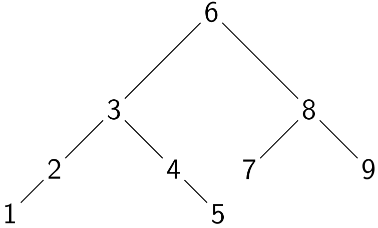
Definition (binary search tree)

A binary search tree for $v_1 < v_2 < \cdots < v_n$ is a binary tree with node set $\{v_1, v_2, \ldots, v_n\}$, such that:

For every node v with left (resp., right) subtree L (resp. R) and all $u \in L$ (resp. $w \in R$) we have: $u < v$ ($v < w$).

Optimal search trees

Example: A binary search tree for $1, 2, 3, 4, 5, 6, 7, 8, 9$



Optimal search trees

Every node v of a search tree B has a level $\ell_B(v)$:
$\ell_B(v) := 1+$ distance (in number of edges) from v to root.

Finding a node at level ℓ requires ℓ comparisons (start in root and then walk down the path to the node).

Problem: Find a binary search tree B with minimal weighted inner path length

$$P(B) := \sum_{v \in V} \ell_B(v) \cdot \gamma(v).$$

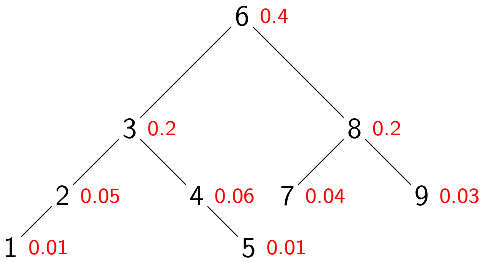
The weighted inner path length is the average cost for accessing a node.

Dynamic programming works because subtrees of optimal binary search trees have to be optimal again.

Optimal search trees

Example: A binary search tree B for $1, 2, 3, 4, 5, 6, 7, 8, 9$.

The weight $\gamma(v)$ of a node v is written next to v.



For the weighted inner path length we get

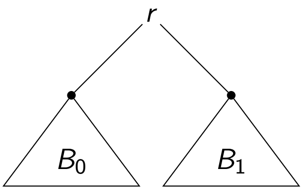
$$\begin{aligned} P(B) &= 1 \cdot 0.4 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.05 + 3 \cdot 0.06 + \\ &\quad 3 \cdot 0.04 + 3 \cdot 0.03 + 4 \cdot 0.01 + 4 \cdot 0.01 \\ &= 1.82. \end{aligned}$$

Optimal search trees

For a subtree B' of a binary search tree B let $\Gamma(B')$ denote the sum of all weights of keys in B'.

For a binary search tree B with left subtree B_0, right subtree B_1, and root r we have

$$\begin{aligned} P(B) &= P(B_0) + \Gamma(B_0) + 1 \cdot \gamma(r) + P(B_1) + \Gamma(B_1) \\ &= P(B_0) + P(B_1) + \Gamma(B). \end{aligned} \tag{6}$$



Optimal search trees

Notation:

- node set $= \{1, \ldots, n\}$, i.e., we identify node v_i with i.
- $P[i,j]$: weighted inner path length of an optimal search tree for the node set $\{i, \ldots, j\}$.
- $R[i,j]$: root of an optimal search tree for $\{i, \ldots, j\}$.

  Since there might be several optimal search trees we take for $R[i,j]$ for the largest root among all optimal search trees.
- $\Gamma[i,j] := \sum_{k=i}^{j} \gamma(k)$: total weight of the node set $\{i, \ldots, j\}$.

From (6) we get

- $P[i,j] = \Gamma[i,j] + \min\{P[i,k-1] + P[k+1,j] \mid k \in \{i, \ldots, j\}\}$
- $R[i,j] =$ largest key among all $k \in \{i, \ldots, j\}$ for which $P[i,k-1] + P[k+1,j]$ is minimal.

This yields the following dynamical programming algorithm.

Optimal search trees

```
for i := 1 to n do
    P[i, i − 1] := 0;
    P[i, i] := γ(i);
    Γ[i, i] := γ(i);
    R[i, i] := i;
endfor
```

```
for d := 1 to n − 1 do
    for i := 1 to n − d do
        j := i + d;
        root := i;
        t := ∞;
        for k := i to j do
            if P[i, k − 1] + P[k + 1, j] ≤ t then
                t := P[i, k − 1] + P[k + 1, j];
                root := k;
            endif
        endfor
        Γ[i, j] := Γ[i, j − 1] + γ(j);
        P[i, j] := t + Γ[i, j];
        R[i, j] := root;
    endfor
endfor
```

Computation of regular expressions

Recall from GTI: Computation of regular expressions by Kleene.

A nondeterministic finite automaton (NFA) is a tuple

$$A = (Q, \Sigma, \delta \subseteq Q \times \Sigma \times Q, I, F) \quad \text{(w.l.o.g. } Q = \{1, \dots, n\}\text{)}.$$

Let $L^k[i,j]$ be the set of all words that label a path in A, which

- leads from i to j and
- thereby only visits intermediate states from $\{1, \dots, k\}$ (i and j do not necessarily belong to $\{1, \dots, k\}$).

Goal: Regular expressions for all $L^n[i,j]$ with $i \in I$ and $j \in F$.

We have

$$L^0[i,j] = \begin{cases} \{a \in \Sigma \mid (i,a,j) \in \delta\} & \text{if } i \neq j \\ \{a \in \Sigma \mid (i,a,j) \in \delta\} \cup \{\varepsilon\} & \text{if } i = j \end{cases}$$

$$L^k[i,j] = L^{k-1}[i,j] + L^{k-1}[i,k] \cdot L^{k-1}[k,k]^* \cdot L^{k-1}[k,j]$$

Computation of regular expressions

Algorithm Regular from an NFA

procedure NFA2REGEXP
Input : NEA $A = (Q, \Sigma, \delta \subseteq Q \times \Sigma \times Q, I, F)$
(Initialize: $L[i,j] := \{a \mid (i,a,j) \in \delta \vee a = \varepsilon \wedge i = j\}$)
begin
  for $k := 1$ **to** n **do**
    for $i := 1$ **to** n **do**
      for $j := 1$ **to** n **do**
        $L[i,j] := L[i,j] + L[i,k] \cdot L[k,k]^* \cdot L[k,j]$
      endfor
    endfor
  endfor
end

Transitiv closure and Warshall's algorithm

Let $G = (V, E)$ be a finite directed graph, i.e., $E \subseteq V \times V$.

A non-empty path from $u \in V$ to $v \in V$ is a sequence of nodes $v_0, v_1, \dots, v_k \in V$ such that $k \geq 1$, $v_0 = u$, $v_k = v$ and $(v_i, v_{i+1}) \in E$ for all $i \in \{0, \dots, k-1\}$.

The transitive closure of G is the graph $G^+ = (V, E^+)$ where $(u, v) \in E^+$ if and only if there is a non-empty path in G from u to v.

The reflexive transitive closure of G is the graph $G^* = (V, E^*)$ where $(u, v) \in E^*$ if and only if $(u, v) \in E^+$ or $u = v$.

In other words: $E^* = E^+ \cup \{(v, v) \mid v \in V\}$.

Adjacency matrix

In the following we assume that the node set is $V = \{1, \ldots, n\}$.

Then, G (and similarly G^+ and G^*) can be represented by its adjacency matrix $A = (a_{i,j})_{1 \leq i,j \leq n} \in \text{Bool}^{n \times n}$ where

$$a_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Let us denote with A^+ (respectively A^*) the adjacency matrix of G^+ (respectively G^*).

Computing the transitiv closure

Warshall's algorithm is based on the following observation, where for a non-empty path $(v_0, v_1, \ldots, v_{m-1}, v_m)$ we denote with $v_1, \ldots, v_{m-1}$ the intermediate nodes of the path:

The following two statements are equivalent for all $i, j, k \in \{1, \ldots, n\}$.

- There is a non-empty path from i to j such that all intermediate nodes belong to $\{1, \ldots, k\}$.
- One the of the following is true:
  - There is a non-empty path from i to j such that all intermediate nodes belong to $\{1, \ldots, k-1\}$.
  - There are (i) a non-empty path from i to k such that all intermediate nodes belong to $\{1, \ldots, k-1\}$ and (ii) a non-empty path from k to j such that all intermediate nodes belong to $\{1, \ldots, k-1\}$.

This observation allows to apply dynamical programming.

Computing the transitiv closure

Algorithm Warshall-algorithm: computation of the transitive closure

---

```
procedure Warshall (var A : adjacency matrix)
Input : graph given by its adjacency matrix (A[i,j]) ∈ Bool^{n×n}
begin
  for k := 1 to n do
    for i := 1 to n do
      for j := 1 to n do
        if (A[i,k] = 1) and (A[k,j] = 1) then
          A[i,j] := 1
        endif
      endfor
    endfor
  endfor
end
```

Transitiv closure?

Algorithm Is this algorithm correct?

procedure Warshall (**var** A : adjacency matrix)
Input : graph given by its adjacency matrix $(A[i,j]) \in \text{Bool}^{n \times n}$

```
begin
  for i := 1 to n do
    for j := 1 to n do
      for k := 1 to n do
        if (A[i,k] = 1) and (A[k,j] = 1) then
          A[i,j] := 1
        endif
      endfor
    endfor
  endfor
end
```

Correctness of Warshall

Correctness of Warshall's algorithm follows from the following invariant:

1. After the k-th excecution of the body of the **for**-loop, we have: $A[i,j] = 1$, if there is a non-empty path from i to j with intermediate nodes from $1, \ldots, k$.

   Important: the outermost loop runs over k.

2. If $A[i,j]$ is set to 1, then there exists a non-empty path from i to j.

If the 0/1-entries in the adjacency matrix are replaced by edge weights from $\mathbb{N}$, one obtains Floyd's algorithm for computing distances in edge-weighted graphs.

In contrast to Dijkstra's algorithm, Floyd's algorithm computes for every pair (u,v) of nodes the distance from u to v):

Floyd's algorithm

Algorithm Floyd: all shortest paths in a graph

procedure Floyd (**var** A : adjacency matrix)
Input : edge-weighted graph given by its adjacency matrix $A[i,j] \in (\mathbb{N} \cup \infty)^{n \times n}$, where $A[i,j] = \infty$ means that there is no edge from i to j.

begin
  for $k := 1$ **to** n **do**
    for $i := 1$ **to** n **do**
      for $j := 1$ **to** n **do**
        $A[i,j] := \min\{A[i,j], A[i,k] + A[k,j]\}$;
      endfor
    endfor
  endfor
end

Floyd's algorithm

Correctness of Floyd's algorithm can be shown analogously to Warshall's algorithm: after the k-th excecution of the body of the **for**-loop, $A[i,j]$ is the minimal weight of path from i to j with intermediate nodes from $1, \ldots, k$.

Running time of Warshall and Floyd: $\Theta(n^3)$.

Simple „improvement": Before entering the j-loop, we test whether

- $A[i,k] = 1$ (for Warshall), respectively
- $A[i,k] < \infty$ (for Floyd)

holds.

This yields a running time of $\mathcal{O}(n^3)$:

Floyd's algorithm

Algorithm Floyd's algorithm in $\mathcal{O}(n^3)$

procedure Floyd (**var** A : adjacency matrix)
Input : adjacency matrix $A[i,j] \in (\mathbb{N} \cup \infty)^{n \times n}$

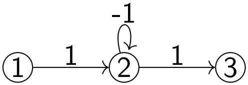
```
begin
  for k := 1 to n do
    for i := 1 to n do
      if A[i,k] < ∞ then
        for j := 1 to n do
          A[i,j] := min{A[i,j], A[i,k] + A[k,j]};
        endfor
      endif
    endfor
  endfor
end
```

Floyd's algorithm

Floyd's algorithm computes correct results also for graphs with negative weights provided that there do not exist cycles with negative total weight.

If negative cycles exist in the graph, then a problem arises!

What is the weight of the optimal path from 1 to 3 in the following graph?

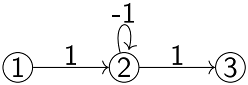


Floyd's algorithm

Floyd's algorithm computes correct results also for graphs with negative weights provided that there do not exist cycles with negative total weight.

If negative cycles exist in the graph, then a problem arises!

What is the weight of the optimal path from 1 to 3 in the following graph?



Answer: $-\infty$

Floyd's algorithm

Algorithm Floyd's algorithm for negative cycles

procedure Floyd (**var** A : adjacency matrix)
Input : adjacency matrix $A[i,j] \in (\mathbb{Z} \cup \{\infty, -\infty\})^{n \times n}$
begin
  for $k := 1$ **to** n **do**
    for $i := 1$ **to** n **do**
      if $A[i,k] < \infty$ **then**
        for $j := 1$ **to** n **do**
          if $A[k,j] < \infty$ **then**
            if $A[k,k] < 0$ **then** $A[i,j] := -\infty$
              else $A[i,j] := \min\{A[i,j], A[i,k] + A[k,j]\}$
            endif
          endif
  endfor endif endfor endfor
end

Transitiv closure and matrix multiplication

Warshall's algorithm computes the reflexive and transitive closure A^* of a boolean matrix A in time $\mathcal{O}(n^3)$.

We can also compute A^* by the formula $A^* = \sum_{k \geq 0} A^k$, where

- $A^0 = I_n$ is the identity matrix and
- $\vee$ (boolean or) is taken for the addition of boolean matrices.

We add matrix entries as follows: $0 + 0 = 0$, $0 + 1 = 1 + 0 = 1 + 1 = 1$.

Claim: $A^k[i,j] = 1 \iff$ there exists a path of length k from i to j.

Transitiv closure and matrix multiplication

Proof by induction on k:

$k = 0$: Since $A^0 = I_n$, we have

$$A^0[i,j] = 1 \iff i = j \iff \text{there is a path of length 0 from } i \text{ to } j.$$

$k > 0$: We have

$$A^k[i,j] = (A^{k-1} \cdot A)[i,j] = \sum_{p=1}^{n} A^{k-1}[i,p] \cdot A[p,j].$$

Hence: $A^k[i,j] = 1$ if and only if there exists a node p such that

- there is a path from i to p of length $k-1$ and
- there is an edge from p to j.

This is true if and only if there is a path from i to j of length k. □

Transitiv closure and matrix multiplication

Since there is a path from i to j if and only if there is a path of length at most $n-1$ ($n =$ number of nodes) from i to j, we have:

$$A^* = \sum_{k=0}^{n-1} A^k.$$

Let $B = I_n + A$. We get $A^* = B^m$ for all $m \geq n-1$.

It therefore suffices to square the matrix B $e := \lceil \log_2(n-1) \rceil$ times in order to compute $B^{2^e} = A^*$.

Let $M(n)$ be the time needed to multiply two boolean $(n \times n)$-matrices. Let $T(n)$ be the time needed to compute the reflexive and transitive closure of a boolean $(n \times n)$-matrix.

We get $T(n) \in \mathcal{O}(M(n) \cdot \log n)$.

Using Strassen's algorithm, we get for all $\varepsilon > 0$:

$$T(n) \in \mathcal{O}(n^{\log_2(7)} \cdot \log n) \subseteq \mathcal{O}(n^{\log_2(7)+\varepsilon}).$$

Transitiv closure and matrix multiplication

But wait: Can we use Strassen's algorithm for multiplying boolean matrices?

Strassen's algorithm works for matrices over $\mathbb{Z}$ (or any ring); it uses negation!

Solution: We take the boolean matrix B from the previous slide and compute the matrix $B^{2^e} \in \mathbb{N}^{n \times n}$ using Strassen's algorithm (with $1 + 1 = 2$).

Then, $B^{2^e}[i, j]$ is the number of paths of length 2^e from i to j in the graph defined by the adjacency matrix B.

By replacing every matrix entry ≥ 2 by 1, we obtain A^*.

Matrix multiplication $\leq$ transitiv closure

Under the plausible assumption that $T(3n) \in \mathcal{O}(T(n))$ we get $M(n) \in \mathcal{O}(T(n))$:

For all boolean matrices A and B we have:

$$\begin{aligned}
\begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}^* &= I_{3n} + \begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}^2 + \cdots \\
&= I_{3n} + \begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & AB \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\
&= \begin{pmatrix} I_n & A & AB \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.
\end{aligned}$$

Matrix multiplication $\leq$ transitiv closure

Under the also plausible assumption that $M(2n) \geq (2+\varepsilon)M(n)$ for an $\varepsilon > 0$, we can show that also $T(n) \in \mathcal{O}(M(n))$.

Hence: The computation of the reflexive and transitive closure is up to constant factors equally expensive as matrix multiplication.

Input: $E \in \mathrm{Bool}(n \times n)$

- Divide E into 4 submatrices A, B, C, D such that A and D are square matrices and each of the 4 matrices has size roughly $n/2 \times n/2$:

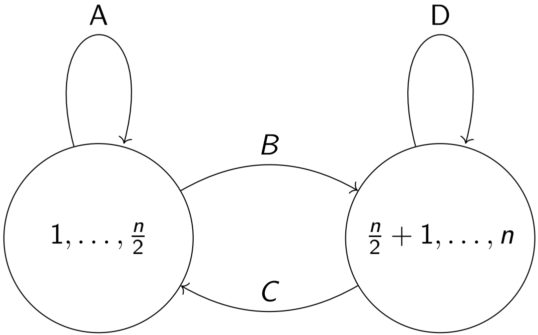
$$E = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

- Compute recursively D^*. Needs time $T(n/2)$.
- Compute $F = A + BD^*C$. Needs time $\mathcal{O}(M(n/2)) \leq \mathcal{O}(M(n))$.
- Compute recursively F^*. Needs time $T(n/2)$.

Computation of the transitiv closure

We finally obtain

$$E^* = \left(\begin{array}{c|c} F^* & F^* B D^* \\ \hline D^* C F^* & D^* + D^* C F^* B D^* \end{array} \right).$$



Computation of the transitiv closure

For the running time we obtain the recurrence

$$T(n) \le 2T(n/2) + c \cdot M(n) \qquad \text{for some } c > 0.$$

This yields

$$\begin{aligned} T(n) &\le c \cdot \left(\sum_{i \ge 0} 2^i \cdot M(n/2^i) \right) && \text{(Theorem 1, Slide 18)} \\ &\le c \cdot \sum_{i \ge 0} \left(\frac{2}{2+\varepsilon} \right)^i \cdot M(n) && \text{(since } M(n/2) \le \frac{1}{2+\varepsilon} M(n)\text{)} \\ &= \frac{c \cdot (2+\varepsilon)}{\varepsilon} M(n). \end{aligned}$$