Introduction

… algorithms, data structures, design principles …

Algorithm courses have the goal to learn about designing 'high quality' data structures and algorithms.
Main criteria for quality are **correctness**, **runtime**- and **space** requirement.
Even simples problems can lead to enormous runtime, if implemented naively.
Complex problems can often only be solved with well designed algorithms, even with fast advance in hardware.
This becomes obvious with the exponential growth in runtime for many naive implementations.

Goals

- Learning methods to design and analyse efficient data structures and algorithms using concrete examples.

Goals

- Learning methods to design and analyse efficient data structures and algorithms using concrete examples.
- Applying these methods independently to different problems.

Goals

- Learning methods to design and analyse efficient data structures and algorithms using concrete examples.
- Applying these methods independently to different problems.
- Getting insight (or 'gut feeling') into the existence (or non-existence) of efficient solutions.

Important for designing good algorithms

- design principles:
  - divide & conquer, scanline principle, recursion, dynamic programming, greedy algorithm, . . .

Important for designing good algorithms

- design principles:
  - divide & conquer, scanline principle, recursion, dynamic programming, greedy algorithm, ...
- data structures:
  - heap, stack, queue, tree, geometric data structures,...

Important for designing good algorithms

- design principles:
  - divide & conquer, scanline principle, recursion, dynamic programming, greedy algorithm, ...
- data structures:
  - heap, stack, queue, tree, geometric data structures,...
- runtime and memory analysis:
  - asymptotic behaviour, $\mathcal{O}-, \Omega-, \Theta-$ notation, worst case, average case, randomised behaviour,...

Important for designing good algorithms

- design principles:
  - divide & conquer, scanline principle, recursion, dynamic programming, greedy algorithm, ...
- data structures:
  - heap, stack, queue, tree, geometric data structures,...
- runtime and memory analysis:
  - asymptotic behaviour, $\mathcal{O}-, \Omega-, \Theta-$ notation, worst case, average case, randomised behaviour,...
- realizability:
  - Is an efficient algorithm possible? NP-completeness, lower bound for runtime,...

Important for designing good algorithms

- design principles:
  - divide & conquer, scanline principle, recursion, dynamic programming, greedy algorithm, ...
- data structures:
  - heap, stack, queue, tree, geometric data structures,...
- runtime and memory analysis:
  - asymptotic behaviour, $\mathcal{O}-, \Omega-, \Theta-$ notation, worst case, average case, randomised behaviour,...
- realizability:
  - Is an efficient algorithm possible? NP-completeness, lower bound for runtime,...
- correctness:
  - runtime/memory bounds, special cases, ...

Example 1: Maximum Subarray Sum

Given: Array A[1,. . . ,n] of integers (also negative values)

Goal: continuous subarray A[i,. . . ,j] with maximum sum

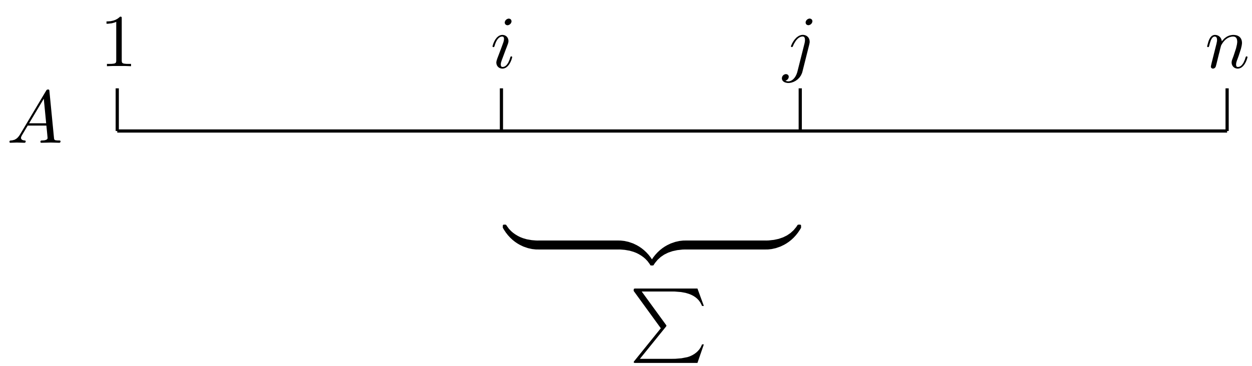
Example 1: Maximum Subarray Sum

Given: Array A[1,. . . ,n] of integers (also negative values)

Goal: continuous subarray A[i,. . . ,j] with maximum sum

4	-5	4	2	-3

$\Rightarrow \sum = 6$



Example 1: Maximum Subarray Sum

Method 1:
Check all subsets of A and check for connectedness.

Example 1: Maximum Subarray Sum

Method 1:

Check all subsets of A and check for connectedness.

$n \ldots$ sum of a subset
$2^n \ldots$ number of subsets
$\mathcal{O}(n2^n)$ runtime

A computer with 10^6 Operations per second needs for $n = 1000$ numbers $\approx 10^{304}$ operations, i.e., $\approx 10^{290}$ years.

Same computer: for $n = 10^6$ numbers $\approx 10^{300000}$ years.

Example 1: Maximum Subarray Sum

Method 2:

Only checking connected sequences:

$$\max_{1 \le i \le j \le n} \left\{ \sum_{k=i}^{j} A[k], 0 \right\}$$

Example 1: Maximum Subarray Sum

Method 2:
Only checking connected sequences:

$$\max_{1\leq i\leq j\leq n}\left\{\sum_{k=i}^{j} A[k], 0\right\}$$

i, j, k run for at most n steps.
Three nested loops $\Rightarrow \mathcal{O}(n^3)$

For $n = 1000 \Rightarrow 10^9$ steps ∼16 min.

For $n = 10^6 \Rightarrow\approx 32000$ years.

Example 1: Maximum Subarray Sum

Method 3 (pseudo code):

Computing the sum 'online' with j.

$max := 0;\ from := 0;\ to := 0;$
for $i := 1$ to n do
    $sum := 0;$
    for $j := i$ to n do
        $sum := sum + A[j];$
        if $sum > max$ then $max := sum;\ from = i;\ to = j;$
        fi
    od
od
$output("A[", from, "-", to, "]$maximum sum $=", max)$

Example 1: Maximum Subarray Sum

Method 3 (pseudo code):

Computing the sum 'online' with j.

i, j go through at most n values $\Rightarrow \mathcal{O}(n^2)$ steps

For $n = 1000 \Rightarrow 10^6$ steps $\sim$1 sec

For $n = 10^6 \Rightarrow \approx 11,5$ days.

Example 1: Maximum Subarray Sum

Method 4:

Run through the input once with a 'scanline' and only consider the part of the input currently covered by the scanline.

Example 1: Maximum Subarray Sum

Method 4:

Run through the input once with a 'scanline' and only consider the part of the input currently covered by the scanline.
Idea: Calculate for every index k the maximum sequence T_k ending at k.
From this get a k with a global maximum sequence.

Observe:

$$T_k \geq 0,$$
$$T_k = \max\{T_{k-1} + A[k], 0\}$$

Example 1: Maximum Subarray Sum

Method 4 (pseudo code):

$max := 0;\ from := 0;\ to := 0;\ f := 1;\ T := 0$
for $k := 1$ to n do
    $T := T + A[k]$
    if $T < 0$ then $T := 0;\ f = k + 1$
    fi
    if $T > max$ then $max := T;\ from := f;\ to := k$
    fi
od
$output("A[", from, " - ", to, "]\text{maximum sum} = ", max)$

f is the start of the current sequence stored in $T \equiv T_k$.
The runtime requirement is $\Theta(n)$.

Example 1: Maximum Subarray Sum

Method 4 (pseudo code):

$max := 0;\ from := 0;\ to := 0;\ f := 1;\ T := 0$

for $k := 1$ to n do

  $T := T + A[k]$

  if $T < 0$ then $T := 0;\ f = k + 1$

  fi

  if $T > max$ then $max := T;\ from := f;\ to := k$

  fi

od

$output("A[", from, "-", to, "]\text{maximum sum} = ", max)$

$n = 1000 \Rightarrow \approx 1/1000$ second; for $n = 10^6 \Rightarrow \approx 1$ second.
Compare: 10^{300000} years (Method 1) or 32000 years (M. 2).

Example 2: Multiplication of large integers

Given: p, q two n-digit decimal numbers.
Goal: compute the product $p \cdot q$ efficiently

Example 2: Multiplication of large integers

Given: p, q two n-digit decimal numbers.
Goal: compute the product $p \cdot q$ efficiently

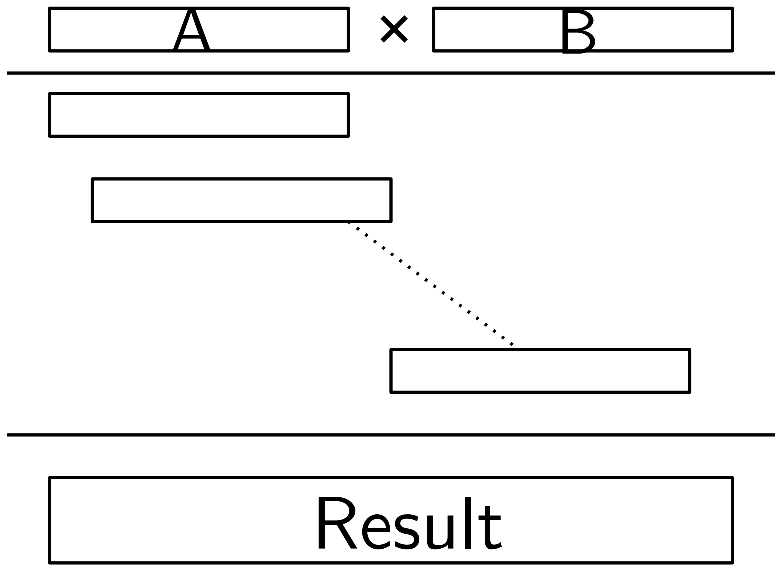
p, q are stored as arrays $P = [1, \ldots, n]$ and $Q = [1, \ldots, n]$, with p_1, q_1 as 'most significant digit'.

$$p = \sum_{i=1}^{n} P[i] \cdot 10^{n-i}$$

$$q = \sum_{i=1}^{n} Q[i] \cdot 10^{n-i}$$

Example 2: Multiplication of large integers

Method 1: School method



n^2 multiplications (plus additions) $\Rightarrow \mathcal{O}(n^2)$ time.

Example 2: Multiplication of large integers

Method 2: Divide & Conquer

Divide p into a and b: $p = a \,||\, b = a \cdot 10^{n/2} + b$

Divide q into c and d: $q = c \,||\, d = c \cdot 10^{n/2} + d$

Example 2: Multiplication of large integers

Method 2: Divide & Conquer

Divide p into a and b: $p = a \,||\, b = a \cdot 10^{n/2} + b$

Divide q into c and d: $q = c \,||\, d = c \cdot 10^{n/2} + d$

$$p \cdot q = a \cdot c \cdot 10^n + (a \cdot d + c \cdot b) \cdot 10^{n/2} + b \cdot d$$

4 multiplications with $n/2$ digits (10^x only needs a shift-operation)

Example 2: Multiplication of large integers

Method 2: Divide & Conquer

$$\begin{aligned} T(n) &= 4 \cdot T(n/2) + \mathcal{O}(n) \\ &= 4(4T(n/4) + \mathcal{O}(n/2)) + \mathcal{O}(n) \\ &= 16T(n/4) + 2\mathcal{O}(n) + \mathcal{O}(n) \\ &= \cdots = 4^k T(n/2^k) + \mathcal{O}(n) \sum_{i=0}^{k-1} 2^i \\ &= 4^{ld(n)} \mathcal{O}(1) + \mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^2). \end{aligned}$$

Divide & Conquer alone is not enough.

Example 2: Multiplication of large integers

Method 3: Improved Divide & Conquer

Calculate $u = ac, v = bd, w = (a + b)(c + d)$. This are three multiplications with $n/2$ digits. We also need $ad + bc$.

Example 2: Multiplication of large integers

Method 3: Improved Divide & Conquer

Calculate $u = ac, v = bd, w = (a+b)(c+d)$. This are three multiplications with $n/2$ digits. We also need $ad + bc$.

$$\begin{aligned}(ad+bc) &= ad + ac + bd + bc - ac - bd \\ &= (a+b)(c+d) - ac - bd \\ &= w - u - v\end{aligned}$$

So no more multiplications are required. Additions can be done in $\mathcal{O}(n)$ time.

Example 2: Multiplication of large integers

Method 3: Improved Divide & Conquer

$$
\begin{aligned}
T(n) &= 3T(n/2) + \mathcal{O}(n) \\
&= 3[3T(n/4) + \mathcal{O}(n/2)] + \mathcal{O}(n) \\
&= 3^2 T(n/2^2) + 3^1 \mathcal{O}(n/2^1) + 3^0 \mathcal{O}(n/2^0) = \cdots = \\
&= 3^k T(n/2^k) + \sum_{i=0}^{k-1} 3^i \mathcal{O}(n/2^i) \\
&= \mathcal{O}(3^{ld(n)}) + \mathcal{O}(n) \sum_{i=0}^{ldn-1} (3/2)^i \\
&= \mathcal{O}(n^{ld(3)}) \sim \mathcal{O}(n^{1.59})
\end{aligned}
$$

Example 3: Exponentiation x^n

Calculate $x^n, x \in \mathbb{R}, n \in \mathbb{N}$ efficiently.

Example 3: Exponentiation x^n

Calculate $x^n, x \in \mathbb{R}, n \in \mathbb{N}$ efficiently.
Example x^{23}:
trivial: 22 multiplications

Example 3: Exponentiation x^n

Calculate $x^n, x \in \mathbb{R}, n \in \mathbb{N}$ efficiently.

Example x^{23}:

trivial: 22 multiplications

better:

$$
\begin{aligned}
x \cdot x &\to x^2 \\
x^2 \cdot x^2 &\to x^4 \\
x^4 \cdot x^4 &\to x^8 \\
x^8 \cdot x^8 &\to x^{16} \\
x^{23} &= x^{16} \cdot x^4 \cdot x^2 \cdot x^1
\end{aligned}
$$

7 multiplications

Example 3: Exponentiation x^n

Idea: Calculate $x^2, x^4, x^8, \ldots, x^{2^k}, k = \lfloor ld(n) \rfloor$ by repeated squaring.

Example 3: Exponentiation x^n

Idea: Calculate $x^2, x^4, x^8, \ldots, x^{2^k}, k = \lfloor ld(n) \rfloor$ by repeated squaring.

Multiply the terms x^{2^i} for which the binary representation $(b_k, b_{k-1}, \ldots, b_1, b_0)$ of n has a 1 in the i-th position, i.e., $b_i = 1$.

In total at most $2 \cdot \lfloor ld(n) \rfloor = \mathcal{O}(log(n))$ multiplications.

Example 3: Exponentiation x^n

Another example: x^{62}

$x^2, x^4, x^8, x^{16}, x^{32}$
$x^{62} = x^2 \cdot x^4 \cdot x^8 \cdot x^{16} \cdot x^{32}$
$\Rightarrow$ 9 multiplications

Example 3: Exponentiation x^n

Another example: x^{62}

$x^2, x^4, x^8, x^{16}, x^{32}$
$x^{62} = x^2 \cdot x^4 \cdot x^8 \cdot x^{16} \cdot x^{32}$
$\Rightarrow$ 9 multiplications

But there exists an even better option:

$x^{62} = x^{20} \cdot x^{20} \cdot x^{20} \cdot x^2$
$x^{20} = x^{16} \cdot x^4$
x^2, x^4, x^8, x^{16}
$\Rightarrow$ 8 multiplications

Example 3: Exponentiation x^n

Another example: x^{62}

$x^2, x^4, x^8, x^{16}, x^{32}$
$x^{62} = x^2 \cdot x^4 \cdot x^8 \cdot x^{16} \cdot x^{32}$
$\Rightarrow$ 9 multiplications

But there exists an even better option:

$x^{62} = x^{20} \cdot x^{20} \cdot x^{20} \cdot x^2$
$x^{20} = x^{16} \cdot x^4$
x^2, x^4, x^8, x^{16}
$\Rightarrow$ 8 multiplications

Finding the optimal solution for general n is an open research problem.

Example 4: Matrix multiplication

With 2×2 matrices:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

Example 4: Matrix multiplication

With 2×2 matrices:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

Traditional method:

$$\begin{aligned} r &= ae + bf \\ s &= ag + bh \\ t &= ce + df \\ u &= cg + dh \end{aligned}$$

In total 8 multiplications and 4 additions are needed.

Example 4: Matrix multiplication

Strassen-method:

$$
\begin{aligned}
p_1 &= a(g-h)\\
p_2 &= (a+b)h\\
p_3 &= (c+d)e\\
p_4 &= d(f-e)\\
p_5 &= (a+d)(e+h)\\
p_6 &= (b-d)(f+h)\\
p_7 &= (a-c)(e+g)
\end{aligned}
$$

$$
\begin{aligned}
s &= p_1 + p_2\\
t &= p_3 + p_4\\
u &= p_1 + p_5 - p_3 - p_7\\
r &= p_4 + p_5 + p_6 - p_2
\end{aligned}
$$

In total 7 multiplications and 18 additions are needed.

Example 4: Matrix multiplication

This method can be generalised for larger matrices:
Let A and B be $n \times n$ matrices with $n = 2^k, k \in \mathbb{N}$ and $C = A \cdot B$. With the traditional method $\Theta(n^3)$ multiplications are needed to calculate C.

Example 4: Matrix multiplication

This method can be generalised for larger matrices:
Let A and B be $n \times n$ matrices with $n = 2^k, k \in \mathbb{N}$ and $C = A \cdot B$. With the traditional method $\Theta(n^3)$ multiplications are needed to calculate C.

Idea:

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

with $C_{ij} = A_{i1} \cdot B_{1j} + A_{i2} \cdot B_{2j}$

If $n \neq 2^k$, it can be filled to the next higher dimension k.

Example 4: Matrix multiplication

With traditional method:
8 $n/2 \times n/2$ matrix multiplications

$$\Rightarrow T(n) = 8T(n/2) + \underbrace{\mathcal{O}(n^2)}_{\text{for matrix additions}}$$

$\Rightarrow T(n) = \mathcal{O}(n^3)$ multiplications $\Rightarrow$ no improvement

Example 4: Matrix multiplication

With traditional method:
8 $n/2 \times n/2$ matrix multiplications
$\Rightarrow T(n) = 8T(n/2) + \underbrace{\mathcal{O}(n^2)}_{\text{for matrix additions}}$

$\Rightarrow T(n) = \mathcal{O}(n^3)$ multiplications $\Rightarrow$ no improvement

With Strassen method:
7 $n/2 \times n/2$ matrix multiplications

$$\Rightarrow T(n) = 7T(n/2) + \mathcal{O}(n^2) = \cdots =$$
$$= \mathcal{O}(n^{ld(7)}) = \mathcal{O}(n^{2.81})$$

Example 4: Matrix multiplication

Remarks:
For multiplication of large matrices there exist even more efficient methods, especially if the matrices have useful properties (such as many 0-entries).

The best known general method has a runtime of $\mathcal{O}(n^{2.373})$.
Obviously $\Omega(n^2)$ is a lower bound.

Summary

Summary:

- Correctness, runtime- and space requirement are important
- Scanline for partial sum problem: from 10^{300000} years to 1 second.
- Simple math helps: subquatratic time for long integer multiplication
- Binary coding might help: exponentiation with logarithmically many multiplications
- Small instances sometimes matter: matrix multiplication efficient with Strassen-method

Summary

Summary:

- Correctness, runtime- and space requirement are important
- Scanline for partial sum problem: from 10^{300000} years to 1 second.
- Simple math helps: subquatratic time for long integer multiplication
- Binary coding might help: exponentiation with logarithmically many multiplications
- Small instances sometimes matter: matrix multiplication efficient with Strassen-method

Thank you for your attention.