Data Structures and Algorithms

Course at D-MATH (CSE) of ETH Zurich

Felix Friedrich

FS 2018

1. Introduction

Algorithms and Data Structures, Three Examples

Goals of the course

- Understand the design and analysis of fundamental algorithms and data structures.
- An advanced insight into a modern programming model (with C++).
- Knowledge about chances, problems and limits of the parallel and concurrent computing.

Goals of the course

On the one hand

- Essential basic knowlegde from computer science.

Andererseits

- Preparation for your further course of studies and practical considerations.

Contents

data structures / algorithms

The notion invariant, cost model, Landau notation

algorithms design, induction

searching, selection and sorting

dynamic programming

dictionaries: hashing and search trees

graphs, shortest paths, backtracking, flow

geometric algorithms, high peformance LA

Randomized algorithms (Gibbs/SA), multiscale approach

sorting networks, parallel algorithms

prorgamming with C++

RAII, Move Konstruktion, Smart Pointers,

Templates and generic programming

Exceptions

Constexpr, user defined literals

functors and lambdas

threads, mutex and monitors

promises and futures

parallel programming

parallelism vs. concurrency, speedup (Amdahl/-Gustavson), races, memory reordering, atomir registers, RMW (CAS,TAS), deadlock/starvation

1.2 Algorithms

[Cormen et al, Kap. 1;Ottman/Widmayer, Kap. 1.1]

Algorithm

Algorithm: well defined computing procedure to compute *output* data from *input* data

example problem

Input : A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

example problem

Input : A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output : Permutation $(a'_1, a'_2, \ldots, a'_n)$ of the sequence $(a_i)_{1 \leq i \leq n}$, such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

example problem

Input : A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output : Permutation $(a'_1, a'_2, \ldots, a'_n)$ of the sequence $(a_i)_{1 \leq i \leq n}$, such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Possible input

$(1, 7, 3)$, $(15, 13, 12, -0.5)$, (1) ...

example problem

Input : A sequence of n numbers $(a_1, a_2, \ldots, a_n)$
Output : Permutation $(a'_1, a'_2, \ldots, a'_n)$ of the sequence $(a_i)_{1 \leq i \leq n}$, such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Possible input

$(1, 7, 3)$, $(15, 13, 12, -0.5)$, (1) ...

Every example represents a *problem instance*

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- fabrication pipeline: Topological Sorting

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- fabrication pipeline: Topological Sorting
- autocomletion and spell-checking: Dictionaries / Trees

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- fabrication pipeline: Topological Sorting
- autocomletion and spell-checking: Dictionaries / Trees
- Symboltables (compiler) : Hash-Tables

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- fabrication pipeline: Topological Sorting
- autocomletion and spell-checking: Dictionaries / Trees
- Symboltables (compiler) : Hash-Tables
- The travelling Salesman: Dynamic Programming, Minimum Spanning Tree, Simulated Annealing

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- fabrication pipeline: Topological Sorting
- autocomletion and spell-checking: Dictionaries / Trees
- Symboltables (compiler) : Hash-Tables
- The travelling Salesman: Dynamic Programming, Minimum Spanning Tree, Simulated Annealing
- Drawing at the computer: Digitizing lines and circles, filling polygons

Examples for algorithmic problems

- Tables and statistis: sorting, selection and searching
- routing: shortest path algorithm, heap data structure
- DNA matching: Dynamic Programming
- fabrication pipeline: Topological Sorting
- autocomletion and spell-checking: Dictionaries / Trees
- Symboltables (compiler) : Hash-Tables
- The travelling Salesman: Dynamic Programming, Minimum Spanning Tree, Simulated Annealing
- Drawing at the computer: Digitizing lines and circles, filling polygons
- Page-Rank: (Markov-Chain) Monte Carlo ...

Characteristics

- Extremely large number of potential solutions
- Practical applicability

Darta Structures

- Organisation of the data tailored towards the algorithms that operate on the data.
- Programs = algorithms + data structures.

Very hard problems.

- NP-compleete problems: no known efficient solution (but the non-existence of such a solution is not proven yet!)
- Example: travelling salesman problem

A dream

- If computers were infinitely fast and had an infinite amount of memory ...
- ... then we would still need the theory of algorithms (only) for statements about correctness (and termination).

The reality

Resources are bounded and not free:

- Computing time $\rightarrow$ Efficiency
- Storage space $\rightarrow$ Efficiency

1.3 Ancient Egyptian Multiplication

Ancient Egyptian Multiplication

Ancient Egyptian Multiplication[1]

Compute $11 \cdot 9$

$$11 \mid 9 \qquad\qquad 9 \mid 11$$

[1] Also known as russian multiplication

Ancient Egyptian Multiplication[1]

Compute $11 \cdot 9$

$$11 \mid 9 \qquad\qquad 9 \mid 11$$

1. Double left, integer division by 2 on the right

[1] Also known as russian multiplication

Ancient Egyptian Multiplication[1]

Compute $11 \cdot 9$

$$\begin{array}{r|r} 11 & 9 \\ 22 & 4 \end{array} \qquad \begin{array}{r|r} 9 & 11 \\ 18 & 5 \end{array}$$

1. Double left, integer division by 2 on the right

[1] Also known as russian multiplication

Ancient Egyptian Multiplication[1]

Compute $11 \cdot 9$

$$\begin{array}{r|r} 11 & 9 \\ 22 & 4 \\ 44 & 2 \end{array} \qquad \begin{array}{r|r} 9 & 11 \\ 18 & 5 \\ 36 & 2 \end{array}$$

1. Double left, integer division by 2 on the right

[1] Also known as russian multiplication

Ancient Egyptian Multiplication[1]

Compute $11 \cdot 9$

$$\begin{array}{r|l} 11 & 9 \\ 22 & 4 \\ 44 & 2 \\ 88 & 1 \end{array} \qquad \begin{array}{r|r} 9 & 11 \\ 18 & 5 \\ 36 & 2 \\ 72 & 1 \end{array}$$

1. Double left, integer division by 2 on the right
2. Even number on the right $\Rightarrow$ eliminate row.

[1] Also known as russian multiplication

Ancient Egyptian Multiplication[1]

Compute $11 \cdot 9$

$$\begin{array}{r|r} 11 & 9 \\ \cancel{22} & \cancel{4} \\ \cancel{44} & \cancel{2} \\ 88 & 1 \end{array} \qquad \begin{array}{r|r} 9 & 11 \\ 18 & 5 \\ \cancel{36} & \cancel{2} \\ 72 & 1 \end{array}$$

1. Double left, integer division by 2 on the right
2. Even number on the right $\Rightarrow$ eliminate row.

[1] Also known as russian multiplication

Ancient Egyptian Multiplication[1]

Compute $11 \cdot 9$

$$\begin{array}{r|r} 11 & 9 \\ \cancel{22} & \cancel{4} \\ \cancel{44} & \cancel{2} \\ 88 & 1 \end{array} \qquad \begin{array}{r|r} 9 & 11 \\ 18 & 5 \\ \cancel{36} & \cancel{2} \\ 72 & 1 \end{array}$$

1. Double left, integer division by 2 on the right
2. Even number on the right $\Rightarrow$ eliminate row.
3. Add remaining rows on the left.

[1] Also known as russian multiplication

Ancient Egyptian Multiplication[1]

Compute $11 \cdot 9$

$$\begin{array}{r|r} 11 & 9 \\ \cancel{22} & \cancel{4} \\ \cancel{44} & \cancel{2} \\ 88 & 1 \\ \hline 99 & - \end{array} \qquad \begin{array}{r|r} 9 & 11 \\ 18 & 5 \\ \cancel{36} & \cancel{2} \\ 72 & 1 \\ \hline 99 & \end{array}$$

1. Double left, integer division by 2 on the right
2. Even number on the right $\Rightarrow$ eliminate row.
3. Add remaining rows on the left.

[1] Also known as russian multiplication

Advantages

- Short description, easy to grasp
- Efficient to implement on a computer: double = left shift, divide by 2 = right shift

Beispiel

left shift $9 = 01001_2 \to 10010_2 = 18$

right shift $9 = 01001_2 \to 00100_2 = 4$

Questions

- Does this always work (negative numbers?)?
- If not, when does it work?
- How do you prove correctness?
- Is it better than the school method?
- What does “good” mean at all?
- How to write this down precisely?

Observation

If $b > 1$, $a \in \mathbb{Z}$, then:

$$a \cdot b = \begin{cases} 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade,} \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade.} \end{cases}$$

Termination

$$a \cdot b = \begin{cases} a & \text{falls } b = 1, \\ 2a \cdot \frac{b}{2} & \text{falls } b \text{ gerade}, \\ a + 2a \cdot \frac{b-1}{2} & \text{falls } b \text{ ungerade}. \end{cases}$$

Recursively, Functional

$$f(a,b) = \begin{cases} a & \text{falls } b = 1, \\ f(2a, \frac{b}{2}) & \text{falls } b \text{ gerade}, \\ a + f(2a, \frac{b-1}{2}) & \text{falls } b \text{ ungerade}. \end{cases}$$

Implemented

```
// pre: b>0
// post: return a*b
int f(int a, int b){
    if(b==1)
        return a;
    else if (b%2 == 0)
        return f(2*a, b/2);
    else
        return a + f(2*a, (b-1)/2);
}
```

Correctnes

$$f(a,b) = \begin{cases} a & \text{if } b = 1, \\ f(2a, \frac{b}{2}) & \text{if } b \text{ even}, \\ a + f(2a \cdot \frac{b-1}{2}) & \text{if } b \text{ odd}. \end{cases}$$

Remaining to show: $f(a,b) = a \cdot b$ for $a \in \mathbb{Z}$, $b \in \mathbb{N}^+$.

Proof by induction

Base clause: $b = 1 \Rightarrow f(a, b) = a = a \cdot 1$.
Hypothesis: $f(a, b') = a \cdot b'$ für $0 < b' \leq b$
Step: $f(a, b+1) \stackrel{!}{=} a \cdot (b+1)$

$$f(a, b+1) = \begin{cases} f(2a, \overbrace{\frac{b+1}{2}}^{\leq b}) = a \cdot (b+1) & \text{if } b \text{ odd,} \\ a + f(2a, \underbrace{\frac{b}{2}}_{\leq b}) = a + a \cdot b & \text{if } b \text{ even.} \end{cases}$$

$\blacksquare$

End Recursion

The recursion can be writen as *end recursion*

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  else if (b%2 == 0)
    return f(2*a, b/2);
  else
    return a + f(2*a, (b-1)/2);
}
```

→

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  int z=0;
  if (b%2 != 0){
    --b;
    z=a;
  }
  return z + f(2*a, b/2);
}
```

End-Recursion ⇒ Iteration

```
// pre: b>0
// post: return a*b
int f(int a, int b){
  if(b==1)
    return a;
  int z=0;
  if (b%2 != 0){
    --b;
    z=a;
  }
  return z + f(2*a, b/2);
}
```

→

```
int f(int a, int b) {
  int res = 0;
  while (b != 1) {
    int z = 0;
    if (b % 2 != 0){
      --b;
      z = a;
    }
    res += z;
    a *= 2; // neues a
    b /= 2; // neues b
  }
  res += a; // Basisfall b=1
  return res;
}
```

Simplify

```
int f(int a, int b) {
  int res = 0;
  while (b != 1) {
    int z = 0;
    if (b % 2 != 0){
      --b;
      z = a;
    }
    res += z;
    a *= 2;
    b /= 2;
  }
  res += a;
  return res;
}
```

--b; → Teil der Division

z = a; → Direkt in res

res += a; → in den Loop

→

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0)
      res += a;
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Invariants!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

Invariants!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

here: $x = \boxed{a \cdot b + res}$

Invariants!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

here: $x = \boxed{a \cdot b + res}$

if here $x = a \cdot b + res$ ...

Invariants!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

here: $x = \boxed{a \cdot b + res}$

if here $x = a \cdot b + res$ ...

... then also here $x = a \cdot b + res$

Invariants!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

here: $x = \boxed{a \cdot b + res}$

if here $x = a \cdot b + res$ ...

... then also here $x = a \cdot b + res$

b even

Invariants!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

here: $x = \boxed{a \cdot b + res}$

if here $x = a \cdot b + res$ ...

... then also here $x = a \cdot b + res$

b even

here: $x = a \cdot b + res$

Invariants!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

here: $x = \boxed{a \cdot b + res}$

if here $x = a \cdot b + res$ ...

... then also here $x = a \cdot b + res$

b even

here: $x = a \cdot b + res$

here: $x = a \cdot b + res$ und $b = 0$

Invariants!

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Sei $x := a \cdot b$.

here: $x = \boxed{a \cdot b + res}$

if here $x = a \cdot b + res$ ...

... then also here $x = a \cdot b + res$

b even

here: $x = a \cdot b + res$

here: $x = a \cdot b + res$ und $b = 0$

Also $res = x$.

Conclusion

The expression $a \cdot b + res$ is an *invariant*

- Values of a, b, res change but the invariant remains basically unchanged
- The invariant is only temporarily discarded by some statement but then re-established
- If such short statement sequences are considered atomiv, the value remains indeed invariant
- In particular the loop contains an invariant, called *loop invariant* and operates there like the induction step in induction proofs.
- Invariants are obviously powerful tools for proofs!

Further simplification

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    if (b % 2 != 0){
      res += a;
      --b;
    }
    a *= 2;
    b /= 2;
  }
  return res;
}
```

→

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a * (b%2);
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Analysis

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a * (b%2);
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Ancient Egyptian Multiplication corresponds to the school method with radix 2.

$$1 \quad 0 \quad 0 \quad 1 \quad \times \quad 1 \quad 0 \quad 1 \quad 1$$

Analysis

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a * (b%2);
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Ancient Egyptian Multiplication corresponds to the school method with radix 2.

$$
\begin{array}{ccccccccccc}
1 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 & & \\
\hline
 & & & & & 1 & 0 & 0 & 1 & & (9)
\end{array}
$$

Analysis

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a * (b%2);
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Ancient Egyptian Multiplication corresponds to the school method with radix 2.

$$
\begin{array}{ccccccccccc}
1 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 & & \\
\hline
 & & & & & 1 & 0 & 0 & 1 & & (9) \\
 & & & & 1 & 0 & 0 & 1 & & & (18)
\end{array}
$$

Analysis

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a * (b%2);
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Ancient Egyptian Multiplication corresponds to the school method with radix 2.

$$
\begin{array}{cccccccccccc}
1 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 & & \\
\hline
 & & & & & 1 & 0 & 0 & 1 & & (9) \\
 & & & & 1 & 0 & 0 & 1 & & & (18) \\
\hline
 & & & & 1 & 1 & 0 & 1 & 1 & & \\
\end{array}
$$

Analysis

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a * (b%2);
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Ancient Egyptian Multiplication corresponds to the school method with radix 2.

$$
\begin{array}{ccccccccccc}
 & & 1 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 & \\
\hline
 & & & & & & & 1 & 0 & 0 & 1 & (9) \\
 & & & & & & 1 & 0 & 0 & 1 & & (18) \\
\hline
 & & & & & & 1 & 1 & 0 & 1 & 1 & \\
 & & & 1 & 0 & 0 & 1 & & & & & (72)
\end{array}
$$

Analysis

```
// pre: b>0
// post: return a*b
int f(int a, int b) {
  int res = 0;
  while (b > 0) {
    res += a * (b%2);
    a *= 2;
    b /= 2;
  }
  return res;
}
```

Ancient Egyptian Multiplication corresponds to the school method with radix 2.

$$
\begin{array}{ccccccccccc}
1 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 & & \\
\hline
 & & & & & 1 & 0 & 0 & 1 & & (9) \\
 & & & & 1 & 0 & 0 & 1 & & & (18) \\
\hline
 & & & & 1 & 1 & 0 & 1 & 1 & & \\
 & & 1 & 0 & 0 & 1 & & & & & (72) \\
\hline
 & & 1 & 1 & 0 & 0 & 0 & 1 & 1 & & (99)
\end{array}
$$

Efficiency

Question: how long does a multiplication of a and b take?

- Measure for efficiency
  - Total number of fundamental operations: double, divide by 2, shift, test for “even”, addition
  - In the recursive and recursive code: maximally 6 operations per call or iteration, respectively
- Essential criterion:
  - Number of recursion calls or
  - Number iterations (in the iterative case)
- $\frac{b}{2^n} \leq 1$ holds for $n \geq \log_2 b$. Consequently not more than $6\lceil \log_2 b \rceil$ fundamental operations.

1.4 Fast Integer Multiplication

[Ottman/Widmayer, Kap. 1.2.3]

Example 2: Multiplication of large Numbers

Primary school:

$$
\begin{array}{cccccc|c}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & d \cdot b
\end{array}
$$

Example 2: Multiplication of large Numbers

Primary school:

$$
\begin{array}{ccccc|c}
a & b & & c & d & \\
6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & 1 & 4 & d \cdot b \\
 & & 4 & 2 & & d \cdot a
\end{array}
$$

Example 2: Multiplication of large Numbers

Primary school:

$$\begin{array}{ccccc|c}
a & b & & c & d & \\
6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & 1 & 4 & d \cdot b \\
 & & 4 & 2 & & d \cdot a \\
 & & & 6 & & c \cdot b
\end{array}$$

Example 2: Multiplication of large Numbers

Primary school:

$$
\begin{array}{cccccc|c}
a & b & & c & d & \\
6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & 1 & 4 & d \cdot b \\
 & & 4 & 2 & & d \cdot a \\
 & & & 6 & & c \cdot b \\
 & 1 & 8 & & & c \cdot a \\
\hline
\end{array}
$$

Example 2: Multiplication of large Numbers

Primary school:

$$
\begin{array}{cccccc|c}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & d \cdot b \\
 & & & 4 & 2 & & d \cdot a \\
 & & & & 6 & & c \cdot b \\
 & & 1 & 8 & & & c \cdot a \\
\hline
= & & 2 & 2 & 9 & 4 &
\end{array}
$$

Example 2: Multiplication of large Numbers

Primary school:

$$
\begin{array}{ccccc|c}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & d \cdot b \\
 & & & 4 & 2 & & d \cdot a \\
 & & & & 6 & & c \cdot b \\
 & & 1 & 8 & & & c \cdot a \\
\hline
= & & 2 & 2 & 9 & 4 &
\end{array}
$$

$2 \cdot 2 = 4$ single-digit multiplications.

Example 2: Multiplication of large Numbers

Primary school:

$$
\begin{array}{cccccc|c}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & d \cdot b \\
 & & & 4 & 2 & & d \cdot a \\
 & & & & 6 & & c \cdot b \\
 & & 1 & 8 & & & c \cdot a \\
\hline
= & & 2 & 2 & 9 & 4 &
\end{array}
$$

$2 \cdot 2 = 4$ single-digit multiplications. $\Rightarrow$ Multiplication of two n-digit numbers: n^2 single-digit multiplications

Observation

$$ab \cdot cd = (10 \cdot a + b) \cdot (10 \cdot c + d)$$

Observation

$$
\begin{aligned}
ab \cdot cd &= (10 \cdot a + b) \cdot (10 \cdot c + d) \\
&= 100 \cdot a \cdot c + 10 \cdot a \cdot c \\
&\quad + 10 \cdot b \cdot d + b \cdot d \\
&\quad + 10 \cdot (a - b) \cdot (d - c)
\end{aligned}
$$

Improvement?

$$
\begin{array}{cccccc|l}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & {\color{red} d \cdot b}
\end{array}
$$

Improvement?

$$
\begin{array}{ccccc|c}
a & b & & c & d & \\
6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & 1 & 4 & d \cdot b \\
 & & 1 & 4 & & d \cdot b
\end{array}
$$

Improvement?

$$
\begin{array}{cccccc|l}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & d \cdot b \\
 & & & 1 & 4 & & d \cdot b \\
 & & & 1 & 6 & & (a-b)\cdot(d-c)
\end{array}
$$

Improvement?

$$
\begin{array}{cccccc|l}
a & b & & c & d & & \\
6 & 2 & \cdot & 3 & 7 & & \\
\hline
 & & & 1 & 4 & & d \cdot b \\
 & & 1 & 4 & & & d \cdot b \\
 & & 1 & 6 & & & (a-b)\cdot(d-c) \\
 & & 1 & 8 & & & c \cdot a
\end{array}
$$

Improvement?

$$
\begin{array}{ccccc|l}
a & b & & c & d & \\
6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & 1 & 4 & d \cdot b \\
 & & 1 & 4 & & d \cdot b \\
 & & 1 & 6 & & (a-b)\cdot(d-c) \\
 & & 1 & 8 & & c \cdot a \\
 & 1 & 8 & & & c \cdot a \\
\hline
\end{array}
$$

Improvement?

$$
\begin{array}{rcccc|l}
 & a & b & & c\ \ d & \\
 & 6 & 2 & \cdot & 3\ \ 7 & \\
\hline
 & & & & 1\ \ 4 & d \cdot b \\
 & & & 1 & 4 & d \cdot b \\
 & & & 1 & 6 & (a-b)\cdot(d-c) \\
 & & & 1 & 8 & c \cdot a \\
 & & 1 & 8 & & c \cdot a \\
\hline
= & & 2 & 2 & 9\ \ 4 &
\end{array}
$$

Improvement?

$$\begin{array}{cccccc|l}
 & a & b & & c & d & \\
 & 6 & 2 & \cdot & 3 & 7 & \\
\hline
 & & & & 1 & 4 & d \cdot b \\
 & & & 1 & 4 & & d \cdot b \\
 & & & 1 & 6 & & (a-b)\cdot(d-c) \\
 & & & 1 & 8 & & c \cdot a \\
 & & 1 & 8 & & & c \cdot a \\
\hline
= & & 2 & 2 & 9 & 4 &
\end{array}$$

$\rightarrow 3$ single-digit multiplications.

Large Numbers

$$6237 \cdot 5898 = \underbrace{62}_{a'}\underbrace{37}_{b'} \cdot \underbrace{58}_{c'}\underbrace{98}_{d'}$$

Large Numbers

$$6237 \cdot 5898 = \underbrace{62}_{a'}\underbrace{37}_{b'} \cdot \underbrace{58}_{c'}\underbrace{98}_{d'}$$

Recursive / inductive application: compute $a' \cdot c'$, $a' \cdot d'$, $b' \cdot c'$ and $c' \cdot d'$ as shown above.

Large Numbers

$$6237 \cdot 5898 = \underbrace{62}_{a'}\underbrace{37}_{b'} \cdot \underbrace{58}_{c'}\underbrace{98}_{d'}$$

Recursive / inductive application: compute $a' \cdot c'$, $a' \cdot d'$, $b' \cdot c'$ and $c' \cdot d'$ as shown above.

$\rightarrow 3 \cdot 3 = 9$ instead of 16 single-digit multiplications.

Generalization

Assumption: two numbers with n digits each, $n = 2^k$ for some k.

$$
\begin{aligned}
(10^{n/2}a + b) \cdot (10^{n/2}c + d) = {} & 10^n \cdot a \cdot c + 10^{n/2} \cdot a \cdot c \\
& + 10^{n/2} \cdot b \cdot d + b \cdot d \\
& + 10^{n/2} \cdot (a - b) \cdot (d - c)
\end{aligned}
$$

Recursive application of this formula: algorithm by Karatsuba and Ofman (1962).

Analysis

$M(n)$: Number of single-digit multiplications.

Recursive application of the algorithm from above $\Rightarrow$ recursion equality:

$$M(2^k) = \begin{cases} 1 & \text{if } k = 0, \\ 3 \cdot M(2^{k-1}) & \text{if } k > 0. \end{cases}$$

Iterative Substition

Iterative substition of the recursion formula in order to guess a solution of the recursion formula:

$$M(2^k) = 3 \cdot M(2^{k-1})$$

Iterative Substition

Iterative substition of the recursion formula in order to guess a solution of the recursion formula:

$$M(2^k) = 3 \cdot M(2^{k-1}) = 3 \cdot 3 \cdot M(2^{k-2}) = 3^2 \cdot M(2^{k-2})$$

Iterative Substition

Iterative substition of the recursion formula in order to guess a solution of the recursion formula:

$$\begin{aligned} M(2^k) &= 3 \cdot M(2^{k-1}) = 3 \cdot 3 \cdot M(2^{k-2}) = 3^2 \cdot M(2^{k-2}) \\ &= \dots \\ &\stackrel{!}{=} 3^k \cdot M(2^0) = 3^k. \end{aligned}$$

Proof: induction

Hypothesis H:

$$M(2^k) = 3^k.$$

Proof: induction

Hypothesis H:

$$M(2^k) = 3^k.$$

Base clause ($k = 0$):

$$M(2^0) = 3^0 = 1. \quad \checkmark$$

Proof: induction

Hypothesis H:

$$M(2^k) = 3^k.$$

Base clause ($k = 0$):

$$M(2^0) = 3^0 = 1. \quad \checkmark$$

Induction step ($k \to k + 1$):

$$M(2^{k+1}) \stackrel{\text{def}}{=} 3 \cdot M(2^k) \stackrel{\text{H}}{=} 3 \cdot 3^k = 3^{k+1}.$$

$\blacksquare$

Comparison

Traditionally n^2 single-digit multiplications.

Comparison

Traditionally n^2 single-digit multiplications.

Karatsuba/Ofman:

$$M(n) = 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n} = 2^{\log_2 3 \log_2 n} = n^{\log_2 3} \approx n^{1.58}.$$

Comparison

Traditionally n^2 single-digit multiplications.

Karatsuba/Ofman:

$$M(n) = 3^{\log_2 n} = (2^{\log_2 3})^{\log_2 n} = 2^{\log_2 3 \log_2 n} = n^{\log_2 3} \approx n^{1.58}.$$

Example: number with 1000 digits: $1000^2/1000^{1.58} \approx 18$.

Best possible algorithm?

We only know the upper bound $n^{\log_2 3}$.

There are (for large n) practically relevant algorithms that are faster. The best upper bound is not known.

Lower bound: $n/2$ (each digit has to be considered at at least once)

1.5 Finde den Star

Is this constructive?

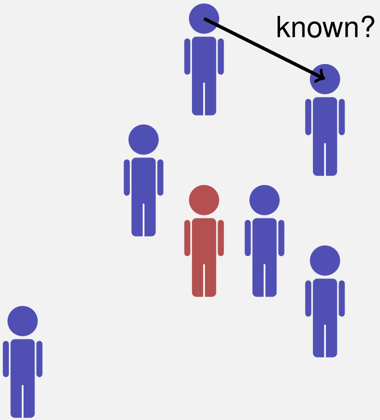
Exercise: find a faster multiplication algorithm.
Unsystematic search for a solution $\Rightarrow$ ☹.

Let us consider a more constructive example.

Example 3: find the star!

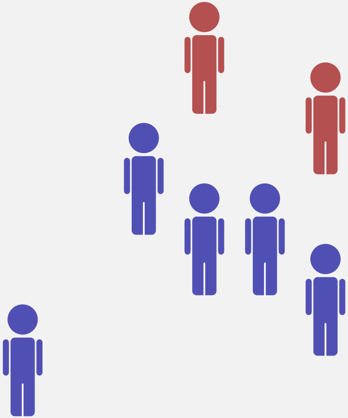
Room with $n > 1$ people.

- *Star:* Person that does not know anyone but is known by everyone.
- *Fundamental operation:* Only allowed question to a person A: "Do you know B?" ($B \neq A$)



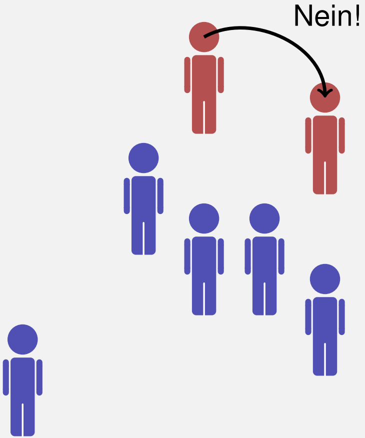
Problemeigenschaften

- Possible: no star present
- Possible: one star present
- More than one star possible?



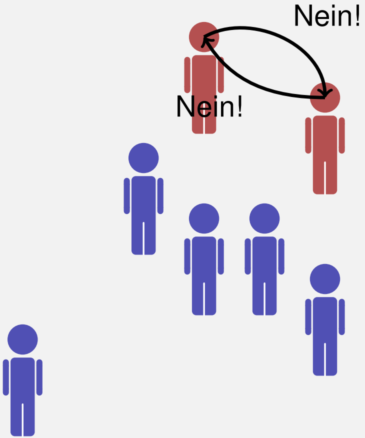
Problemeigenschaften

- Possible: no star present
- Possible: one star present
- More than one star possible?



Problemeigenschaften

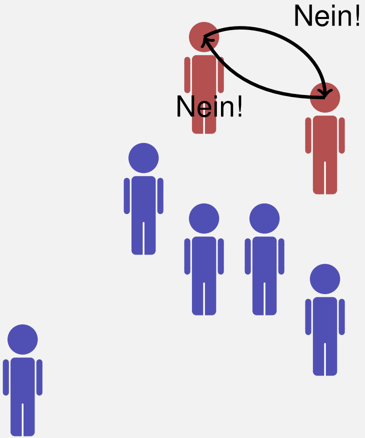
- Possible: no star present
- Possible: one star present
- More than one star possible?



Problemeigenschaften

- Possible: no star present
- Possible: one star present
- More than one star possible?

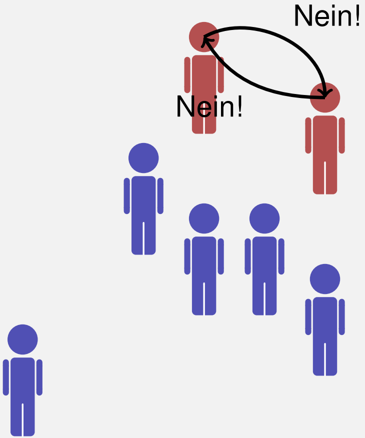
Assumption: two stars S_1, S_2.



Problemeigenschaften

- Possible: no star present
- Possible: one star present
- More than one star possible?

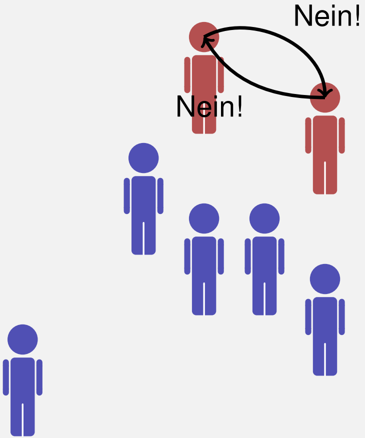
Assumption: two stars S_1, S_2.
S_1 knows $S_2 \Rightarrow S_1$ no star.



Problemeigenschaften

- Possible: no star present
- Possible: one star present
- More than one star possible?

Assumption: two stars S_1, S_2.
S_1 knows $S_2 \Rightarrow S_1$ no star.
S_1 does not know $S_2 \Rightarrow S_2$ no star. $\perp$



Naive solution

Ask everyone about everyone

Result:

	1	2	3	4
1	-	yes	no	no
2	no	-	no	no
3	yes	yes	-	no
4	yes	yes	yes	-

Naive solution

Ask everyone about everyone

Result:

	1	2	3	4
1	-	yes	no	no
2	no	-	no	no
3	yes	yes	-	no
4	yes	yes	yes	-

Star is 2.

Naive solution

Ask everyone about everyone

Result:

	1	2	3	4
1	-	yes	no	no
2	no	-	no	no
3	yes	yes	-	no
4	yes	yes	yes	-

Star is 2.

Numer operations (questions): $n \cdot (n-1)$.

Better approach?

Induction: partition the problem into smaller pieces.

Better approach?

Induction: partition the problem into smaller pieces.

- $n = 2$: Two questions suffice

Better approach?

Induction: partition the problem into smaller pieces.

- $n = 2$: Two questions suffice
- $n > 2$: Send one person out. Find the star within $n - 1$ people. Then check A with $2 \cdot (n - 1)$ questions.

Better approach?

Induction: partition the problem into smaller pieces.

- $n = 2$: Two questions suffice
- $n > 2$: Send one person out. Find the star within $n - 1$ people. Then check A with $2 \cdot (n - 1)$ questions.

Overal
$F(n) = 2(n-1) + F(n-1) = 2(n-1) + 2(n-2) + \cdots + 2 = n(n-1)$.

Better approach?

Induction: partition the problem into smaller pieces.

- $n = 2$: Two questions suffice
- $n > 2$: Send one person out. Find the star within $n - 1$ people. Then check A with $2 \cdot (n - 1)$ questions.

Overal
$F(n) = 2(n-1) + F(n-1) = 2(n-1) + 2(n-2) + \cdots + 2 = n(n-1).$

No benefit. ☹

Improvement

Idea: avoid to send the star out.

Improvement

Idea: avoid to send the star out.

- Ask an arbitrary person A if she knows B.

Improvement

Idea: avoid to send the star out.

- Ask an arbitrary person A if she knows B.
- If yes: A is no star.

Improvement

Idea: avoid to send the star out.

- Ask an arbitrary person A if she knows B.
- If yes: A is no star.
- If no: B is no star.

Improvement

Idea: avoid to send the star out.

- Ask an arbitrary person A if she knows B.
- If yes: A is no star.
- If no: B is no star.
- At the end 2 people remain that might contain a star. We check the potential star X with any person that is out.

Analyse

$$F(n) = \begin{cases} 2 & \text{for } n = 2, \\ 1 + F(n-1) + 2 & \text{for } n > 2. \end{cases}$$

Analyse

$$F(n) = \begin{cases} 2 & \text{for } n = 2, \\ 1 + F(n-1) + 2 & \text{for } n > 2. \end{cases}$$

Iterative substitution:

$$F(n) = 3 + F(n-1) = 2 \cdot 3 + F(n-2) = \cdots = 3 \cdot (n-2) + 2 = 3n - 4.$$

Proof: exercise!

Moral

With many problems an inductive or recursive pattern can be developed that is based on the piecewise simplification of the problem. Next example in the next lecture.