Algorithms and Data Structures

Asymptotic Complexity

Ulf Leser

Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples

Efficiency of Algorithms

- Algorithms have an input and solve a defined problem
  - Sort this list of names
  - Compute the running 3-month average over this table of 10 years of daily revenues
  - Find the shortest path between node X and node Y in this graph with n nodes and m edges
- Research in algorithms focuses on efficiency
  - Efficiency: Use as few resources as possible for solving the task
  - Resources: CPU cycles, memory cells, (network traffic, disk IO, ...)
- How can we measure efficiency for different inputs?
- How can we compare the efficiency of two algorithms solving the same problem?

Option 1: Use a Reference Machine

- Empirical evaluation
  - Chose a concrete machine (CPU, RAM, BUS, ...)
    - Or many different machines
  - Chose a set of different input data sets (workloads)
    - The more, the better
    - Real, synthetic, realistic, ...
  - Run algorithm on all inputs and measure time (or space or ...)
- Pro: Gives real runtimes and practical guidance
- Contra
  - Will all potential users have this machine?
  - Performance dependent on prog language and skill of engineer
  - Are the datasets used typical for what we expect in an application?
  - Can we extrapolate results beyond the given data sets?

Option 2: Computational Complexity

- Derive an estimate of the maximal (worst-case) number of operations as a function of the input
  - "For an input of size n, the alg. will perform "~n^3" operations"
  - Abstraction: Define a (realistic) model of a machine
- Advantages
  - Analyses the abstract algorithm, not its concrete implementation
  - Independent of concrete hardware; future-proof
- Disadvantages
  - No real runtimes, no practical guidance
  - What is an operation? What do we count?
  - Requires assumptions on the cost of primitive operations
  - Assumes that all machines offer the same set of operations

Next steps

- In this lecture, we focus on complexity
  - Note again: When it comes to practical problems, complexity is not everything
  - There can be extremely large runtime differences between algorithms having the same complexity
  - Difference between theoretical and practical computer science
- We need to define what we count: Machine model
- We need to define how we estimate: O-notation

Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples

Our Machine Model: RAM

- Very simple model: Random Access Machines (RAM)
- Work: What a traditional CPU can execute in 1 cycle
  - Addition, comparison, jumps, ...
  - Forget multi-core, disks, ALUs, GPUs, FPGA, cache levels, pipelining, hyper-threading, ...
  - Note: There are machine models for many of these variations
- Space: Infinite amount of storage cells
  - Each cell holds one (possibly infinitely large) value (number)
    - Separate program storage – no interference with data
    - Cells are addressed by consecutive integers
    - Access to each cell in one CPU cycle
  - Special treatment of input and output
  - One special register (switch) storing results of a comparison

Operations

- Load value into cell, move value from cell to cell
  - LOADv 3, 5: Load value “5” in cell 3
  - LOAD 3, 5: Copy value of cell 5 into cell 3
- Add/subtract/multiply/divide value/cell to/from/by cell and store in cell
  - ADDv 3, 5, 6; Add “6” to value of cell 5 and store result in cell 3
  - ADD 3, 5, 6; Add value of cell 6 to value of cell 5 and store in cell 3
- Compare values of two cells
  - If equal, set switch to TRUE, otherwise to FALSE
- Jump to position if switch is TRUE
- Jump to position
- Stop
  - RET 6; Returns value of cell 6 as result and stop

Example: x^y (for y>0)

```
input
  x,y: integer;
t: integer;
i: integer;
t := x;
for i := 1 … y-1 do
  t := t * x;
end for;
return t;
```

```
1. LOADv 1, x;    # provide input
2. LOADv 2, y;
3. LOAD 3, 1;     # t := x
4. LOADv 4, 1;    # i := 1
5. CMP 4, 2;      # check i = y
6. IFTRUE 10;
7. MULT 3, 1, 3; # t := t*x
8. ADDv 4, 4, 1; # i := i+1
9. GOTO 5;
10.RET 3;         # return t
```

Cost Models

- We count the number of operations (time) performed and the number of cells (space) required
- This is called uniform cost model (UCM)
  - Every operation costs time 1, every value needs space 1
    - Not realistic
    - Data access has non-uniform cost (cache lines)
    - Comparing two real numbers requires more work than to integers
    - ...
- Alternative model: Machine cost (logarithmic cost)
  - Consider concrete machine representation of every data element
  - Cells hold 1 byte – how many bytes do I need?
  - More realistic, yet more complex
  - Derives identical complexity results for most sensible cases

Counting Operations in the RAM Model

```
1. LOADv 1, x;    # input
2. LOADv 2, y;
3. LOAD 3, 1;     # t := x
4. LOADv 4, 1;    # i := 1
5. CMP 4, 2;      # check i=y
6. IFTRUE 10;
7. MULT 3, 1, 3; # t := t*x
8. ADDv 4, 4, 1; # i := i+1
9. GOTO 5;
10.RET 3;         # return t
```

- If y>1
  - Startup (lines 1-4) costs 4
  - Loop (lines 5-9) costs 5
  - Loop is passed y times
  - Last loop costs 2, return costs 1
  - Total costs: 4+(y-1)*5+3
- If y=1
  - Total costs: 7=4+(y-1)*5+3

Selection Sort: Uniform versus Machine Cost

```
1.  S: array_of_names;
2.  n := |S|
3.  for i = 1..n-1 do
4.    for j = i+1..n do
5.      if S[i]>S[j] then
6.        tmp := S[i];
7.        S[i] := S[j];
8.        S[j] := tmp;
9.      end if;
10.   end for;
11. end for;
```

- With UCM, we showed $f(n) \sim 4n^2-3n$
  - But: Every cell needs to hold a name = string of arbitrary length
  - We used a UCM including strings
- Towards machine cost
  - Assume max length m for any S[i]
  - Then, line 5 costs m comps in WC
  - Lines 6-8; additional cost for loops for copying char-by-char
- We did not consider super-long strings ($i>2^{64}$) or super-large alphabets (char comp in 1 cycle?)

Conclusions

- We usually assume RAM with uniform cost, but will not give the RAM program itself
  - Translation from pseudo code is simple and adds only constant costs per operation – which we will ignore anyway
- We assume UCM for primitive data types: numbers, strings
  - We sometimes look at strings in more detail
  - More complex data type (lists, sets etc.) will be analyzed in detail
- When analyzing real programs, many more issues arise
  - Performance killer in Java: Garbage collection
  - Performance trick in Java: Object reuse
  - Performance killer in Java: new vector (1,1)
  - …

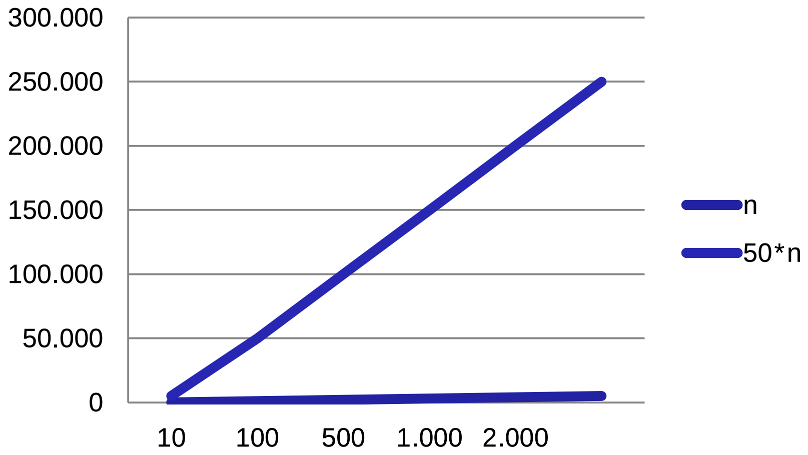
Content of this Lecture

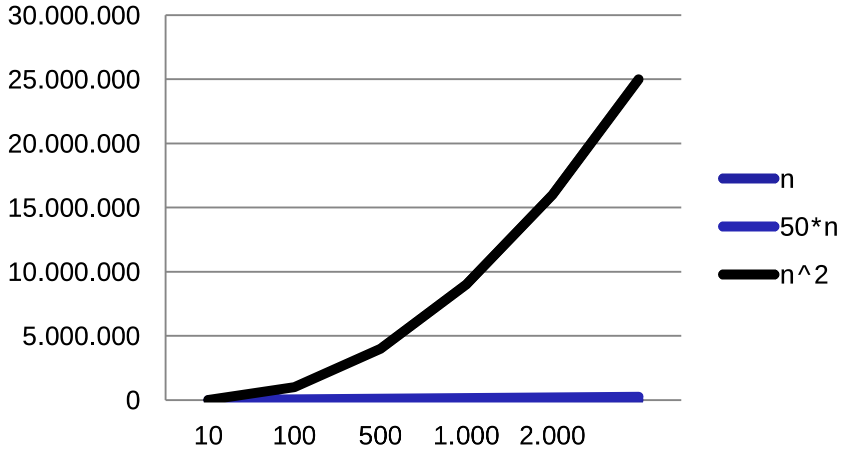
- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples

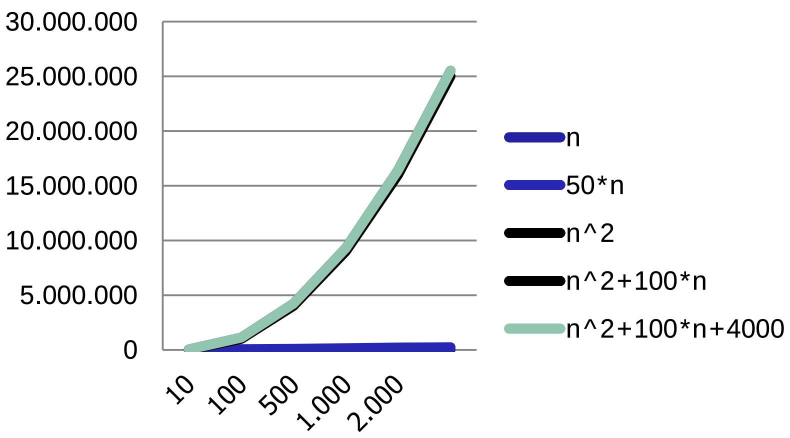
Complexity

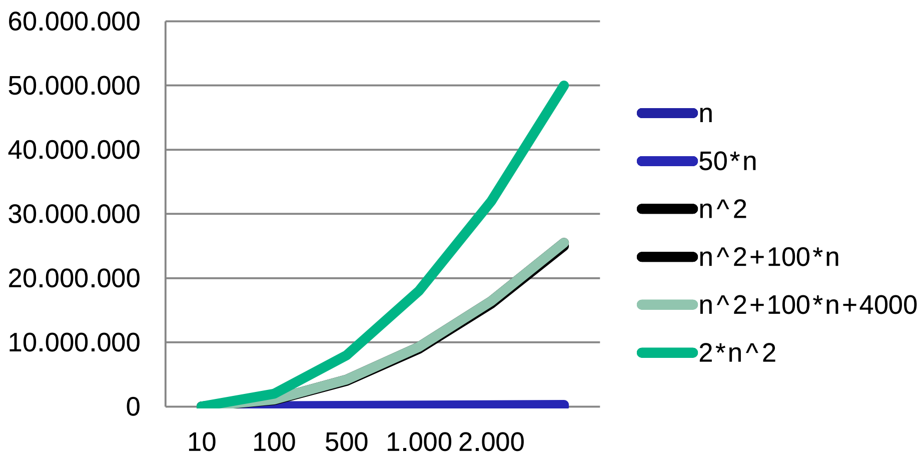
- Counting the exact number of operations for an algorithm (wrt. input size) seems overly complicated
  - Linear scale-ups are often possible by using newer/more machines
  - Estimations need not be good for all cases - for small inputs, many algorithms are lightning-fast anyway
  - We don’t want long formulas – focus on the dominant factors
- Intuitive goal: Analyzes the major cost drivers when the input gets “large”
- Asymptotic complexity – behavior if input size goes to infinity

Examples

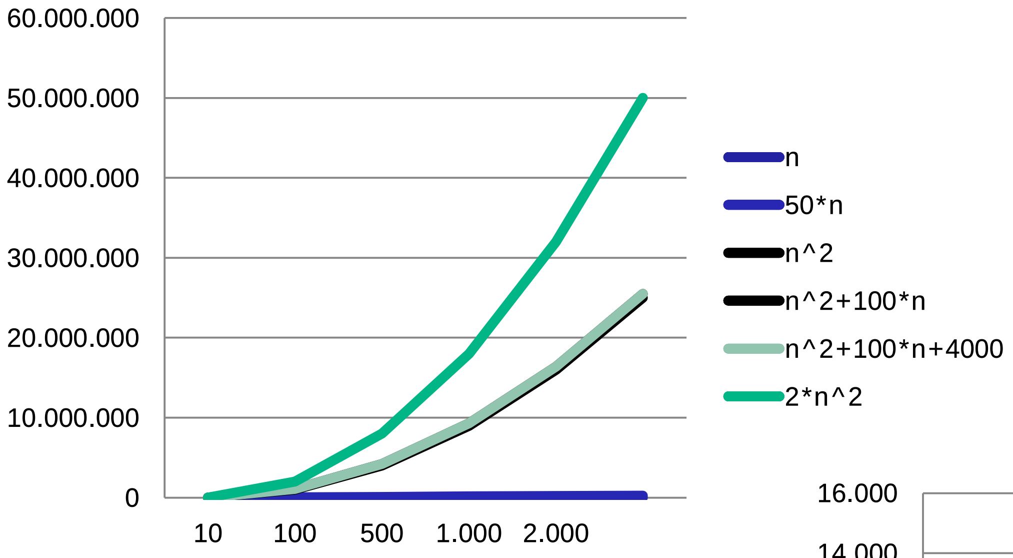

300.000
250.000
200.000
150.000
100.000
50.000
0
10
100
500
1.000
2.000
n
50*n

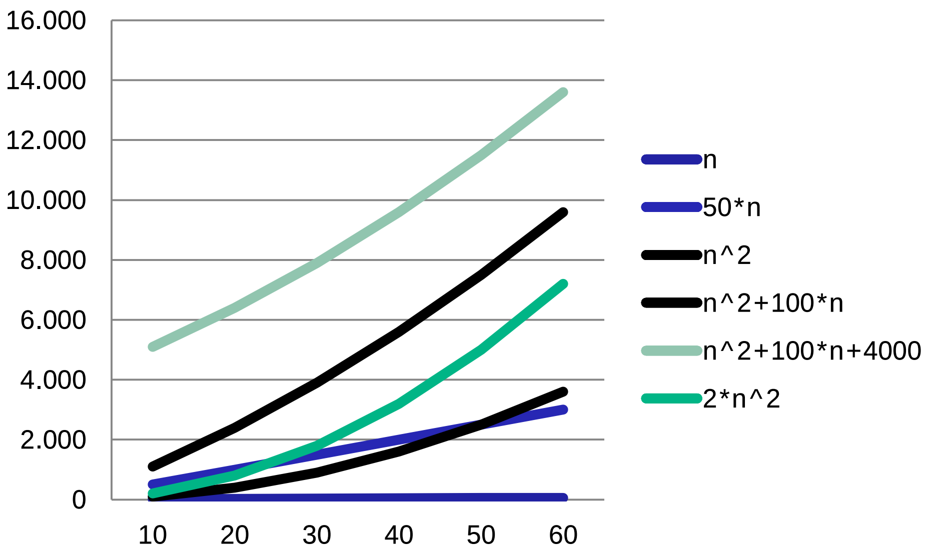

30.000.000
25.000.000
20.000.000
15.000.000
10.000.000
5.000.000
0
10
100
500
1.000
2.000
n
50*n
n^2


30.000.000
25.000.000
20.000.000
15.000.000
10.000.000
5.000.000
0
10
100
500
1.000
2.000
n
50*n
n^2
n^2+100*n
n^2+100*n+4000


60.000.000
50.000.000
40.000.000
30.000.000
20.000.000
10.000.000
0
10
100
500
1.000
2.000
n
50*n
n^2
n^2+100*n
n^2+100*n+4000
2*n^2

Small Values





Intuitive Observations

- Everything except the term with the highest exponent doesn't matter much, if n is large enough
- This term can have a factor, but the effect of this factor usually can be outweighed by newer/more machines
  - Therefore, we do not consider it
- Assume we have developed a polynomial f capturing the exact cost of an algorithm A
- Intuitively, the complexity of A is the term in f with the highest exponent after stripping linear factors

Overview

- Assume f(n) gives the number of operations performed by alg. A in worst case for an input of size n
- We are interested in the essence of f, i.e., the dominating factors when n grows large
- We do this by defining a hierarchy of classes of functions
  - For a function g, define O(g) as the class of functions that is asymptotically smaller or equal g
    - We want a simple g; simpler than f
  - If $f \in O(g)$, then f will be asymptotically smaller or equal g
    - I.e.: for large inputs, the number of ops counted by f will be smaller than or equal to the one estimated through g
  - Asymptotically, g is an upper bound for f
    - Not necessarily the lowest

Formally: O-Notation

- Definition
  Let g: $N \rightarrow R^+$. O(g) is the class of functions defined as
  $O(g) = \{f: N \rightarrow R^+ \mid \exists c, n_0: \forall n \geq n_0: f(n) \leq c*g(n)\}$
- Explanation
  - O(g) is the class of all functions which compute lower or equal values than g for any sufficiently large n, ignoring linear factors
  - O(g) is the class of functions that are asymptotically smaller than or equal g
- If $f \in O(g)$, we say that "f is in O(g)" or "f is O(g)" or "f has complexity O(g)"

Examples

`f(n)=3*n²+6*n+7 is O(n²)`

`f(n)=n³+7000*n-300 is O(n³)`

`f(n)=4*n²+200*n²-100 is O(n²)`

`f(n)=log(n)+300 is O(log(n))`

`f(n)=log(n)+n is O(n)`

`f(n)=n*log(n) is O(n*log(n))`

`f(n)=n² is O(n³)`

- Example: First f
  - Choose c=9 and n_0=7
  - Assume $n>7=n_0$:
    - Then, $n^2>6*n+7$
    - Thus: $3n^2+6n+7 \leq 3n^2 + n^2$
    - Finally: $3*n^2+n^2 \leq 9*n^2$
  - Would also work for c=8,7, ...
- Concrete values of c and n_0 don't matter
  - Especially: No need to search for smallest such values for proving complexity

General Result

- Lemma: *All constant functions are in O(1)*
  - Let f(n)=k for some k>0
  - Let g(n)=1
  - We need to show that $f \in O(g)$
- Proof
  - Chose c=k and $n_0=0$
  - Clearly: $\forall n \geq n_0$, we now have $f(n) \leq c * g(n) = k*1 = k$
- Any part of an algorithm whose extend of work is independent of n can be summarized as O(1)

Calculating with Complexities

```
1. S: array_of_names;
2. n := |S|
3. for i = 1..n-1 do
4.   for j = i+1..n do
5.     if S[i]>S[j] then
6.       tmp := S[i];
7.       S[i] := S[j];
8.       S[j] := tmp;
9.     end if;
10.  end for;
11.end for;
```

- Usually, we want to derive the complexity of a program without calculating its exact cost
  - Estimate a tight g without knowing f
- Some observations
  - Having many ops with cost 1 yields the same complexity as having only 1
    - Lines 5-8 cost 4 times 1 $\in$ O(1)
  - If we see a polynomial, we can forget terms except the largest
    - As we certainly need O(n) for the outer loop, we can forget the startup which is O(1)

Formally: O-Calculus

- Such observations can be cast in a set of rules
- Lemma
  Let k be a constant. The following equivalences are true
  - $O(k+f) = O(f)$;
  - $O(k*f) = O(f)$;
  - $O(f) + O(g) = O(\max(f,g))$
  - $O(f) * O(g) = O(f*g)$

with "slight misuse of notations"

- Explanations
  - Rule 3 (4) actually implies rule 1 (2), as $k \in O(1)$
  - Rule 3 is used for sequentially executed parts of a program
  - Rule 4 is used for nested parts of a program (loops)

Example

- There is a typo in this slide: Somewhere, I typed “und” instead of “and”. Where?
- Abstract problem: Given a string T (template) und a pattern P (pattern), find all occurrences of P in T
  - Exact substring search
- The following algorithm solves this problem
  - There are better ones

```
for i = 1..|T|-|P|+1 do
  match := true;
  j := 1;
  while match
    if T[i+j-1]=P[j] then
      if j=|P| then
        print i;
        match := false;
      end if;
      j := j+1;
    else
      match := false,
    end if;
  end while;
end for;
```

Complexity Analysis (n=|T|, m=|P|)

```
1.  for i = 1..|T|-|P|+1 do
2.    match := true;
3.    j := 1;
4.    while match
5.      if T[i+j-1]=P[j] then
6.        if j=|P| then
7.          print i;
8.          match := false;
9.        end if;
10.       j := j+1;
11.     else
12.       match := false,
13.     end if;
14.   end while;
15. end for;
```

```
1.  O(n-m)
2.    O(1)
3.    O(1)
4.    O(m)
5.      O(1)
6.        O(1)
7.          O(1)
8.          O(1)
9.        -
10.       O(1)
11.     -
12.       O(1)
13.     -
14.   -
15. -
```

Worst-Case

O(1)+O(1)=O(1)

```
1.  O(n-m)
2.    O(1)
3.    O(m)
4.      O(1)
```

O(1)*m)=O(m)

```
1.  O(n-m)
2.    O(1)
3.    O(m)
```

O(1)+O(m)=O(m)

```
1.  O(n-m)
2.    O(m)
```

O(n-m)*O(m)= O((n-m)*m)

```
1.  O((n-m)*m)
```

Deriving new Rules: Transitivity of O-Membership

- Lemma: If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$
- Proof
  - We know: $\exists c, n_0: \forall n \geq n_0: f(n) \leq c*g(n)$
  - We know: $\exists c', n'_0: \forall n \geq n'_0: g(n) \leq c'*h(n)$
  - We need to show: $\exists c'', n''_0: \forall n \geq n''_0: f(n) \leq c''*h(n)$
  - We chose: $n''_0 = \max(n_0, n'_0)$; $c''=c*c'$
  - This gives:
    $\forall n \geq n''_0: f(n) \leq c*g(n) \leq c*c'*h(n) \leq c''*h(n)$
  - qed.

Ω-Notation

- O-Notation denotes an upper bound for the amount of computation necessary to run an algorithm for asymptotically large inputs
  - "f will always be faster than g"
- Sometimes, we also want lower bounds
  - "f can never be faster than g"
- Definition
  Let $g: N \rightarrow R^+$. $\Omega(g)$ is the class of functions defined as
  $\Omega(g) = \{f:N \rightarrow R^+ | \exists c, n_0: \forall n \geq n_0: g(n) \leq c*f(n)\}$
- Explanation
  - Ω(g) is the class of functions that are asymptotically larger than g
  - Again: Not necessarily the largest smaller one

Further Notation

- $O(g) = \left\{ f: \mathbb{R}_0^+ \to \mathbb{R}_0^+ \middle| \begin{array}{c} \exists c \in \mathbb{R}^+ > 0 \;\; \exists n_0 \in \mathbb{R}_0^+ > 0 \\ \forall n \geq n_0:\; f(n) \leq c \cdot g(n) \end{array} \right\}$
- $\Omega(g) = \left\{ f: \mathbb{R}_0^+ \to \mathbb{R}_0^+ \middle| \begin{array}{c} \exists c \in \mathbb{R}^+ > 0 \;\; \exists n_0 \in \mathbb{R}_0^+ > 0 \\ \forall n \geq n_0:\; f(n) \geq c \cdot g(n) \end{array} \right\}$
- $\Theta(g) = \left\{ f: \mathbb{R}_0^+ \to \mathbb{R}_0^+ \middle| \begin{array}{c} \exists c_1, c_2 \in \mathbb{R}^+ > 0 \;\; \exists n_0 \in \mathbb{R}_0^+ > 0 \\ \forall n \geq n_0:\; c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \end{array} \right\}$
- $o(g) = \left\{ f: \mathbb{R}_0^+ \to \mathbb{R}_0^+ \middle| \begin{array}{c} \forall c \in \mathbb{R}^+ > 0 \;\; \exists n_0 \in \mathbb{R}_0^+ > 0 \\ \forall n \geq n_0:\; f(n) < c \cdot g(n) \end{array} \right\}$
- $\omega(g) = \left\{ f: \mathbb{R}_0^+ \to \mathbb{R}_0^+ \middle| \begin{array}{c} \forall c \in \mathbb{R}^+ > 0 \;\; \exists n_0 \in \mathbb{R}_0^+ > 0 \\ \forall n \geq n_0:\; f(n) > c \cdot g(n) \end{array} \right\}$

Not Every Problem is Simple

- Definition
  We call an algorithm A with cost function f
  - *polynomial, if there exists a polynomial p with $f \in O(p)$*
  - *exponential, if $\exists \varepsilon > 0$ with $f \in \Omega(2^{n^{\varepsilon}})$*
- General assumption: If A is exponential, it cannot be executed in reasonable time for non-trivial input
  - But: If A is exponential, this does not imply that the problem solved by A cannot be solved in polynomial time
  - Of course: If A is bounded by a polynomial, then also the problem solved by A can be solved in polynomial time (by A)
  - Much research in finding good solutions for difficult problems

Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples
  - Exact substring search (average-case versus worst-case)
  - Knapsack problem (exponential problem)

Exact Substring Search: Average Case

```
for i = 1..|T|-|P| do
  match := true;
  j := 1;
  while match
    if T[i+j-1]=P[j] then
      if j=|P| then
        print i;
        match := false;
      end if;
      j := j+1;
    else
      match := false,
    end if;
  end while;
end for;
```

- We showed that the algorithm's WC is O((n-m)*m)~O(n*m)
  - Since m<<n
- How does a worst case look like?

Exact Substring Search: Beyond Worst Case

```
for i = 1..|T|-|P| do
  match := true;
  j := 1;
  while match
    if T[i+j-1]=P[j] then
      if j=|P| then
        print i;
        match := false;
      end if;
      j := j+1;
    else
      match := false,
    end if;
  end while;
end for;
```

- We showed that the algorithm's WC is O((n-m)*m)~O(n*m)
  - Since m<<n
- How does a worst case look like?
  - $T=a^n$; $P=a^m$
- What about the average case?
  - The outer loop is always passed by n times, no matter how T / P look like
  - This already is in Ω(n-m) in all cases
    - Worst, best, average, ...

Exact Substring Search: Average Case

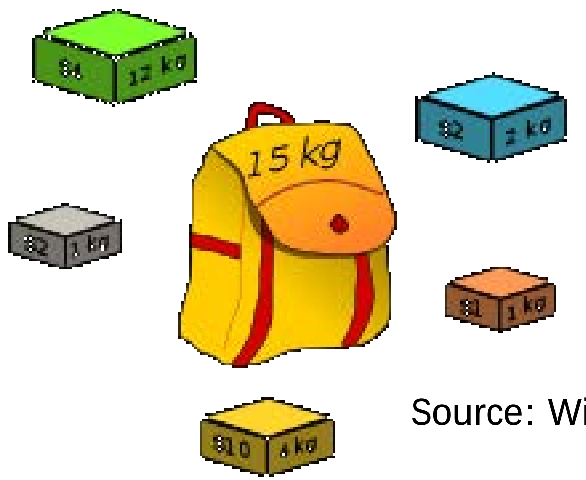
```
1.  O(n)
2.    while match
3.      if T[i+j-1]=P[j] then
4.        O(1)
5.      else
6.        O(1);   # end loop
7.    -
```

- How often do we pass by the inner loop?
- Needs a model of "average strings"
- Simplest model:
  T and P are randomly generated from the same alphabet Σ
  - Every character appears with equal probability at every position
- Gives a chance of $p=1/|\Sigma|$ for every test "T[i+j-1]=P[j]"
- Derive the expected number of comparisons in line 3
  - $1(1-p)+2*p(1-p)+3*p^2(1-p)+\ldots+m*p^{m-1}=$
    $1-p \quad + 2p-2p^2+ 3p^2-3p^3+ \ldots m*p^{m-1}=$
    $1 \quad + \quad p \quad + \quad p^2 \quad + \quad p^3 \quad + \ldots p^{m-1} = \sum_{i=0}^{m-1} p^i$

Differences On Real Data

- Assume |T|=50.000 and |P|=8 and |Σ|=28
  - German text, including Umlaute, excluding upper/lower case letters
  - Worst-case estimate: 400.000 comparisons
    - Note: Here, O(m*n) is quite tight, no linear factors ignored
  - Average-case estimate: ~51.851 comparisons
    - We expect a mismatch after every 1,03 comparisons
- Assume |T|=50.000, |P|=8, |Σ|=4 (e.g., DNA)
  - Worst-case: 400.000 comparisons
  - Average-case: 65.740
- Best algorithms are O(m+n) ~ 50.008 comparisons
- Much better WC result, but not much better AC result
- But: Are German texts random strings?

Example 2: Knapsack Problem



Source: Wikipedia.de

- Given a set S of items with weights w[i] and value v[i] and a maximal weight m; find the subset T⊆S such that:

$$\sum_{i \in T} w[i] \leq m \quad \text{and} \quad \sum_{i \in T} v[i] = \max$$

Algorithm and its Complexity

- Imagine an algorithm which enumerates all possible T
- For each T, computing its value and its weight is in O(|S|)
  - Testing for maximum is O(1)
- But how many different T exist?

Algorithm and its Complexity

- Imagine an algorithm which enumerates all possible T
- For each T, computing its value and its weight is in O(|S|)
  - Testing for maximum is O(1)
- But how many different T exist?
  - Every item from S can be part of T or not
  - This gives 2*2*2* .... *2=$2^{|S|}$ different options
- Together: This algorithm is in $O(2^{|S|})$

- Actually, the knapsack problem is NP-hard
- Thus, very likely no polynomial algorithm exists

Exemplary Questions for Examination

- Given the following algorithm: ... Analyze its worst case and average case complexity
- Prove that O(f*g) = O(f)*O(g)
- Order the following functions by their complexity class: n^2, 100n, n*log(n), $n*2^{log(n)}$, sqrt(n), n!
- Let $f \in \Omega(g)$ and $g \in \Omega(h)$. Show that $f \in \Omega(h)$
- Find a function f such that: $f \in \Omega(n)$ and $f \notin O(n^3 * log(n))$