Algorithms and Data Structures

Asymptotic Complexity

Ulf Leser

News

- We are approaching 180
- Exercises start today
  - Have a look at the first exercise soon
- New times for tutorial
  - Di. 11-13 Uhr, RUD25 4.113
  - Do. 11-13 Uhr, RUD25 4.112

Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples

Efficiency of Algorithms

- Research in algorithms focuses a lot on efficiency
  - Find fast/space-efficient algorithms for a given problem
  - Best-case, on average, in the worst-case
- 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
  - Not: Which day is today? Are nuclear power plants evil?
- How can we measure efficiency for different inputs?
  - Also: How can we compare the efficiency of two algorithms for different inputs?

Option 1: Using Reference Machine

- Using a reference machine
  - Define a concrete machine (CPU, RAM, BUS, ...)
  - Chose a set of different inputs
  - Run algorithm on all inputs and measure times
- Pro: Gives real runtimes
- Contra
  - Only one machine for the entire world?
  - Time dependent on programming language and skill of engineer
  - Times between measured points can only be extrapolated
  - Are these datasets typical for what we expect in the real world?
    - Uniformly distributed over all possible inputs?
  - Reference machines are always out-of-date

Option 2: Computational Complexity

- Derive an estimate of the maximal (worst-case) number of operations performed as a function of the input
  - "For an input of size n, the alg. will perform $n^3+3n/5$ operations"
- Advantages
  - Independent of machine
  - Independent of implementation of the algorithm
    - If we make certain assumptions on the cost of primitive operations
- Disadvantages
  - No real runtimes
  - What is an operation? What do we count?

Outline

- In this lecture, we focus on complexity
  - Note: 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 - Famous O-notation

Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples

Machine Model

- Very simple model: Random Access Machines (RAM)
- Roughly: What a traditional CPUs can execute in 1 cycle
  - Forget pipelining, registers, multi-core, disks, arithmetic units, ...
- Storage
  - Infinite amount of storage cells
    - Each cell holds one (possibly infinitely large) number
    - Cells are addressed by integers
  - Separate program storage – no interference with data
  - Special treatment of input and output
  - Special register (switch) storing results of comparisons

Operations

- Load value into cell, move value from cell to cell
  - LOADv 3, 5: Load value “5” in cell 3
  - LOAD 3, 5: Load 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 cell
  - 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 Model

- 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 cell needs space 1
    - “1” has no unit – we concentrate on the change in cost
  - Independent of size of operands
    - Clearly not realistic: Every CPU has only a certain number of bits per operation, thus can only compute with values up to a certain limit
- Alternative: Machine costs (or logarithmic cost model)
  - Consider machine representation of input and all operands
  - More realistic, yet more complex
  - Often not necessary (“values in sensible range”)

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 costs 4
  - Loop (lines 5-9) costs 5
  - Loop is passed by y-1 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 names
  - Then, line 5 costs m comps in WC
  - Lines 6-8; additional cost for loops for copying char-by-char
- In 5-8, AC≠WC
  - Given two strings, how many characters do we have to compare on average to see which is greater?

Conclusions

- We usually assume RAM with uniform cost, but will not give the RAM program itself
  - Translation from pseudo code to our language is simple and adds only constant costs per operation
- We mostly assume UCM for ops on numbers and strings
  - We sometimes look at strings in more details
  - More complex data type (lists, sets) will be analyzed in detail
- When analyzing real programs, many more issues arise
  - Cost of library functions? Internal management of inherited methods? Management of free space? Etc.

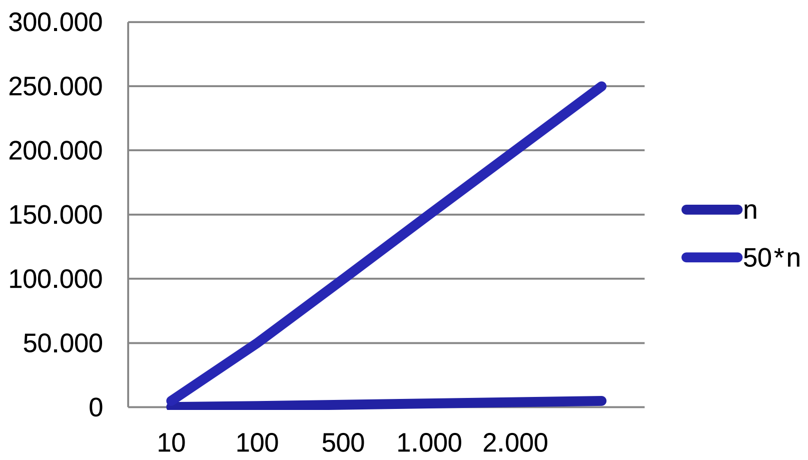
Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples

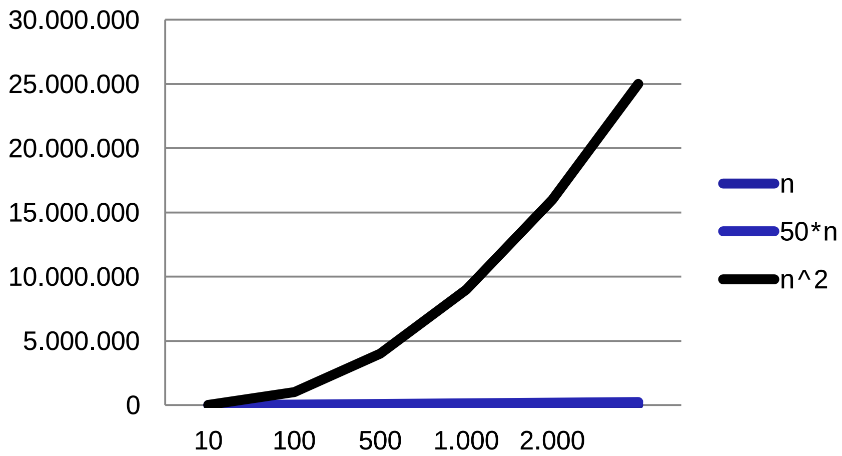
Complexity

- 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 lightening-fast anyway
- We concentrate on the major factors influencing runtime when the input gets "large"
  – Asymptotic complexity – behavior if input size goes to infinity

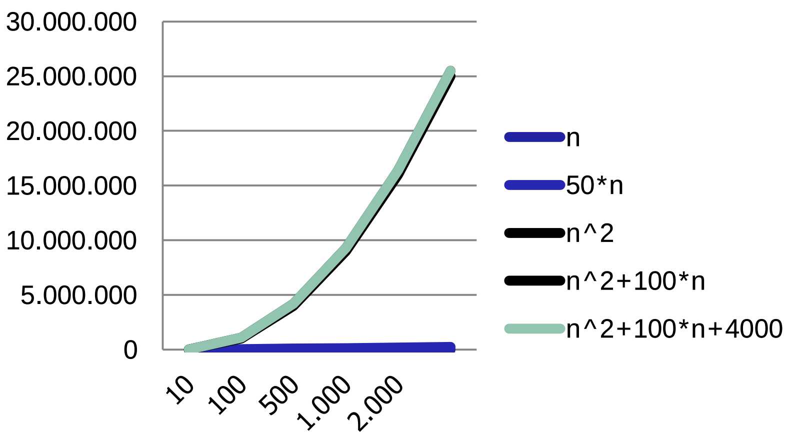
Examples



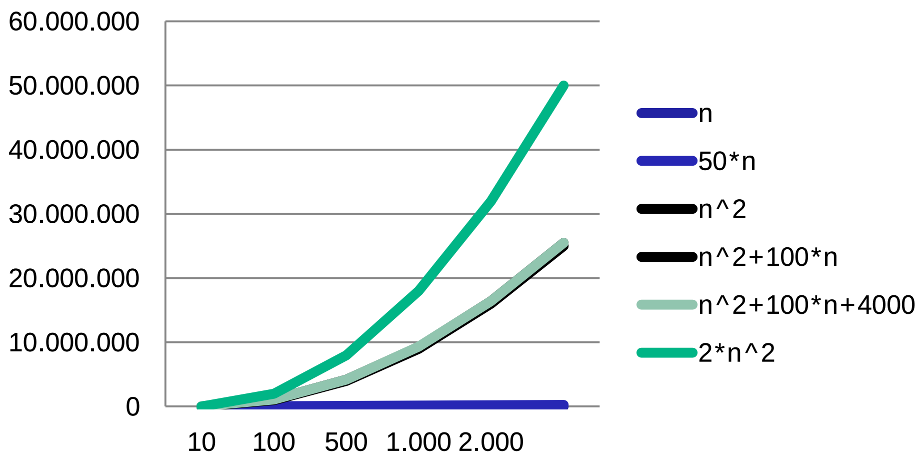
300.000
250.000
200.000
150.000
100.000
50.000
0
10
100
500
1.000
2.000
n
50*n



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

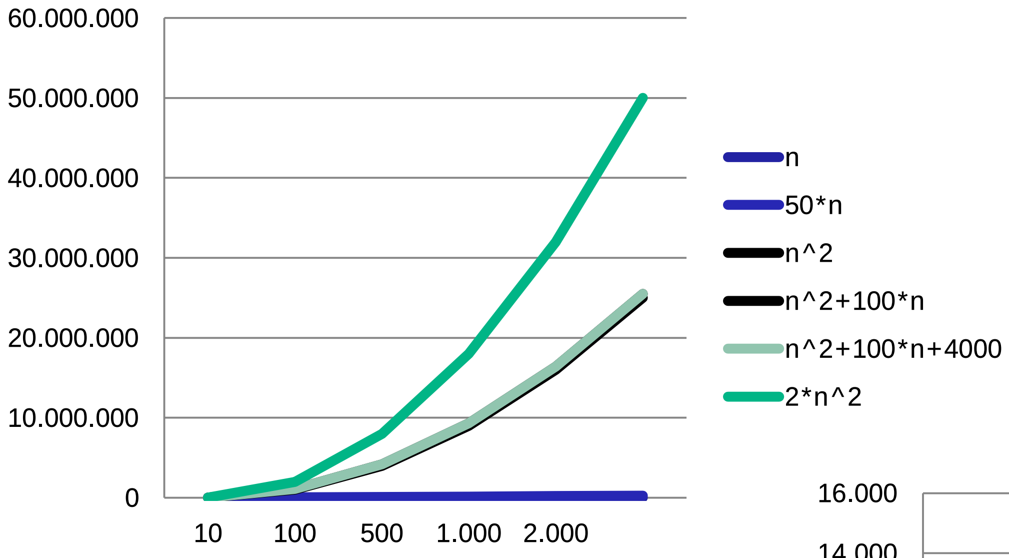


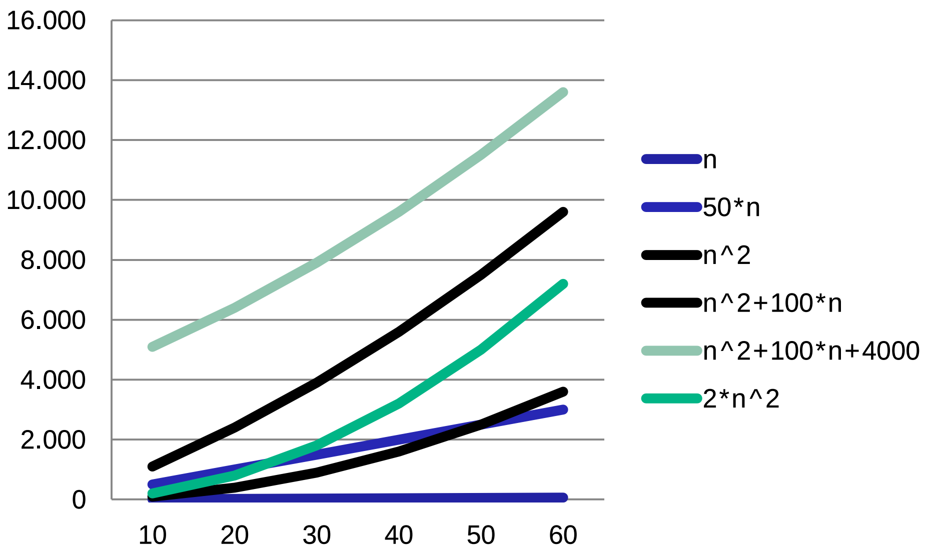
30.000.000
25.000.000
20.000.000
15.000.000
10.000.000
5.000.000
0
10
100
500
1.000
2.000
n
50*n
n^2
n^2+100*n
n^2+100*n+4000



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 out-weighted 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 of f with the highest exponent after striping constant factors

More Formally

- For now, let’s 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 describing the essence of f, i.e., the factors which will dominate the runtime if n grows large
- To this end, we define a hierarchy of classes of functions
- For a function g, O(g) is the class of functions that is asymptotically smaller than g
  – We want a simple g; simpler than f
- Now, if $f \in O(g)$, than f will be asymptotically smaller than g; for large inputs, the number of ops will be smaller than estimated through g
  – We don’t know how much smaller

Formally: O-Notation

- Definition
  Let $g: N \rightarrow R^+$. O(g) is the class of functions defined as
  $$O(g) = \{f: N \rightarrow R^+ \mid \text{there exist positive constants } c, n_0 \text{ with } f(n) \le c*g(n)\ \forall n \ge n_0\}$$
- Explanation
  - O(g) is the class of all functions which compute lower values than g for any sufficiently large n, ignoring linear factors
  - O(g) is the class of functions that are asymptotically smaller than 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)"
- Algorithms also can have more than one input – definition of complexity is analogous

Examples

$f(n)=3*n^2+6*n+7$ is $O(n^2)$

$f(n)=n^3+7000*n-300$ is $O(n^3)$

$f(n)=4*n^2+200*n^2-100$ is $O(n^2)$

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

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

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

$f(n)=n^2$ is $O(n^3)$

- Example: First f
  - Chose $c=9$ and $n_0=7$
  - For any $n>7$: $n^2>6*n+7$
  - With $3*n^2+6*n^2=9*n^2$ we have $3*n^2+6*n+7 \leq 9*n^2$
- Values of c and n_0 don't matter
  - Especially: No need to search for smallest such values
- Now, we can formally state the complexity of selection sort: $O(n^2)$
  - Exact cost was $4n^2-3n+1$

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[j];
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 ~ 1
  - If we see a polynomial, we can forget about all smaller or equal ones
    - Will only lead to constant factors
    - As we certainly need O(n) for the outer loop, we can forget the startup

O-Calculus

- These 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 occurrence of P in T
  – Exact substring search
- The following algorithm solves this problem
  – Note: There are better algorithms

```
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;
```

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

O(1*m)=O(m)

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

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

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

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

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

Ω-Notation

- O-Notation denotes an upper bound for the amount of computation necessary to run an algorithm for asymptotically large inputs
  - Not necessarily the lowest upper bound
- Sometimes, we also want lower bounds
- Definition
  Let $g: N \rightarrow R^+$. $\Omega(g)$ is the class of functions defined as
  $\Omega(g) = \{f: N \rightarrow R^+ \mid$ *there exist positive constants* c, n_0 *with*
  $$g(n) \leq c * f(n) \; \forall n \geq n_0 \}$$
- Explanation
  - $\Omega(g)$ is the class of functions that are asymptotically larger than g

Not Every Problem is Simple

- Definition
  We call an algorithm A with cost function f
  - *bounded by a 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)

Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples
  - Exact substring search (average-case versus worst-case)
  - Knapsack 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 is O(n*m) in worst-case
- How does a worst case look like?

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 is O(n*m) in worst-case
- How does a worst case look like?
  - $T=a^n$; $P=a^m$
- What about the AC complexity?
  - The outer loop is always passed by n times, no matter how T / P look like
  - This already gives $\Omega(n)$

Exact Substring Search: Average Case

```
O(n)
  while match
    if T[i+j-1]=P[j] then
      O(1)
    else
      match := false,
  -
```

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

On Real Data

- Assume $|T|=50.000$ and $|P|=8$ and $|\Sigma|=28$
  - German text, including Umlaute, excluding upper/lower case letters
  - Worst-case upper bound: 400.000 comparisons
  - Average-case: ~51.851 comparisons
    - We expect a mismatch after 1,03 comparisons
- Assume $|T|=50.000$, $|P|=8$, $|\Sigma|=4$ (e.g., DNA)
  - Worst-case: 400.000 comparisons
  - Average-case: 65.740
- Best algorithms are $O(m+n)$ ~ 50.008 comparisons
  - Beware: We ignore constant factors
- Not much better than the average case
- But: Are German texts random strings?

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 $T \subseteq S$ 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 at least in O($2^{|S|}$)

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