Algorithms and Data Structures

Asymptotic Time Complexity

Ulf Leser

Mit Beiträgen von Patrick Schäfer

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, …)
  - CPU cycles are directly correlated with time
- 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 skills of engineer
  - Are the datasets used typical for what we expect in an application?
  - Can we extrapolate results beyond the measured data sets?

Option 2: Computational Complexity

- Derive an estimate of the maximal (worst-case) number of operations as a function of the size of the input
  - "For an input of size n, the alg. will perform app. n^3 operations"
  - Abstraction: Based on a universal yet realistic machine model
- Advantages
  - Analyses the algorithm, not its concrete implementation
  - Independent of concrete hardware; future-proof
- Disadvantages
  - No real runtimes
  - 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 computational complexity
- We need to define what we count: Machine model
- We need to define how we estimate: O-notation
- Complexity analysis: Versatile & elegant yet coarse-grained

Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples

Our Machine Model: Random Access Machines

- Very simple model: Random Access Machines (RAM)
- Work: What a conventional CPU can execute in 1 cycle
  - Addition, comparison, jumps, …
  - Forget multi-core, disks, ALUs, GPUs, FPGA, cache levels, pipelining, hyper-threading, …
  - Note: There are proper machine models for such variations
- Space: Unlimited number of storage cells
  - Each cell holds one (possibly infinitely large) value (number)
    - Cells are addressed by consecutive integers
    - Access (read/write) to each cell in one CPU cycle
  - Special treatment of input and output
  - One special register (switch) storing results of a comparison

Programs

- Programs
  - Consists of atomic operations with 1-3 parameters
  - Are stored consecutively in memory cells
    - Separate address space – no interference with data
    - Each operation one cell
    - For simplicity, we assume that parameters take no space
  - Address of operations can be used for jumps

```
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, 3, 1;  # t := t*x
8.  ADDv 4, 4, 1;  # i := i+1
9.  GOTO 5;
10. RET 3;         # return t
```

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
  - **`CMP 4, 2;`** If equal, set switch to **`TRUE`**, otherwise to **`FALSE`**
- Conditional jumps
  - **`IFTRUE 10;`** Jump to position `10` if switch is **`TRUE`**
- Absolute jumps
  - **`GOTO 5;`** Jump to command at position 5
- 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, 3, 1;  # t := t*x
8. ADDv 4, 4, 1;  # i := i+1
9. GOTO 5;
10.RET 3;         # return t
```

4 cells:
- 1: x
- 2: y
- 3: t
- 4: i

Cost Models

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

Counting Operations in the RAM Model with UCM

```
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, 3, 1; # 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 (line 5) is passed y times
    - (y-1)-times costs 5 (lines 5-9)
    - 1-time costs 2 (lines 5-6)
  - Return costs 1
  - Total costs: $4 + (y-1) \cdot 5 + 3$
- If y=1
  - Total costs: $7 = 4 + (y-1) \cdot 5 + 3$

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 $f(n) \sim 3n^2+3n$
  - 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 a string S[i]
  - Then, line 5 costs m comparisons WC
  - Lines 6-8; additional cost for loops for copying char-by-char
- We did not consider super-long strings ($n>2^{64}$), or super-large alphabets (char comp always in 1 cycle?)

Conclusions

- We usually assume RAM with UCM, but will not give the RAM program itself
  - Translation from pseudo code is simple and adds only constant costs per operation – which we will (later) ignore anyways
- We assume UCM for primitive data types: numbers, strings
  - We will 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);
  - ...

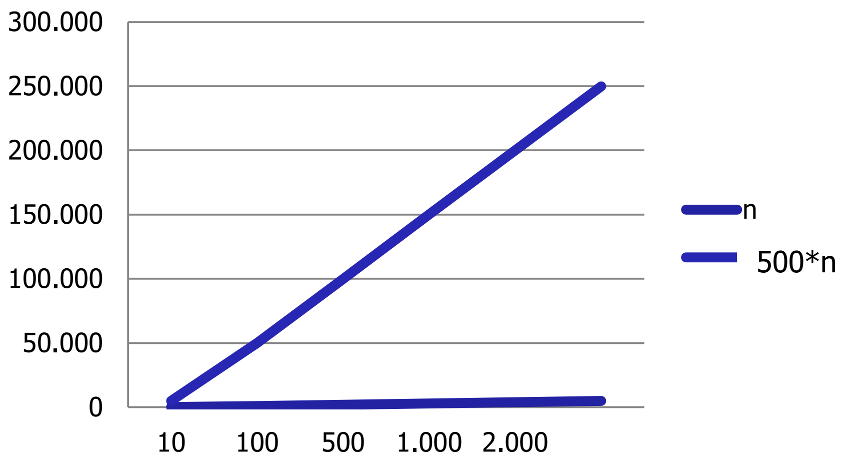
Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples

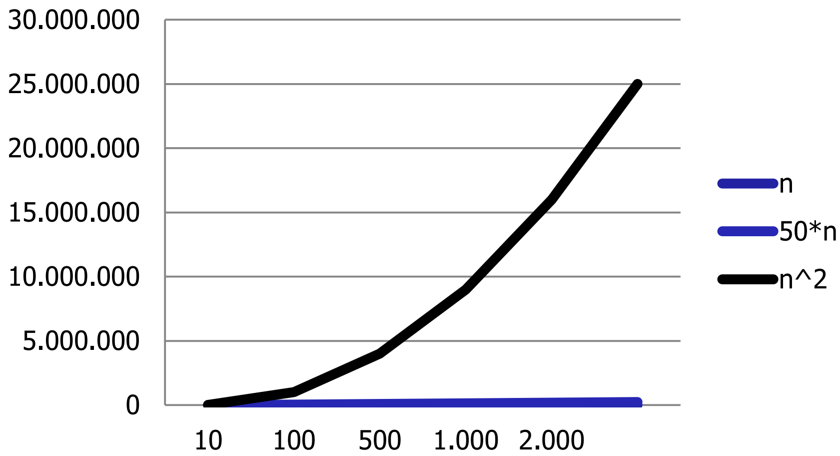
Computational Complexity

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

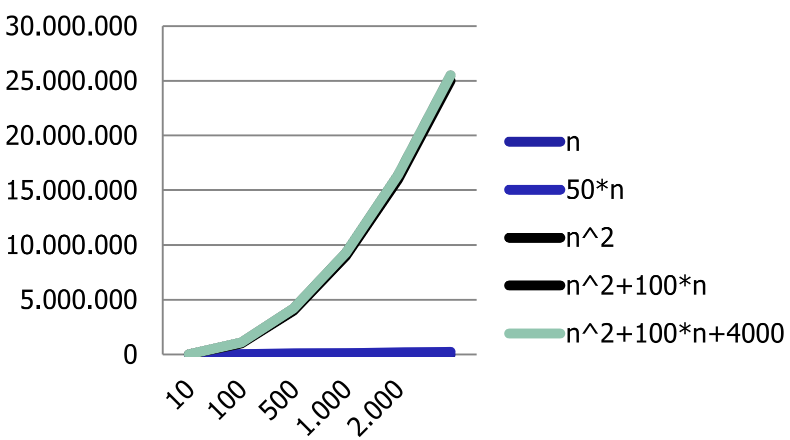
Examples



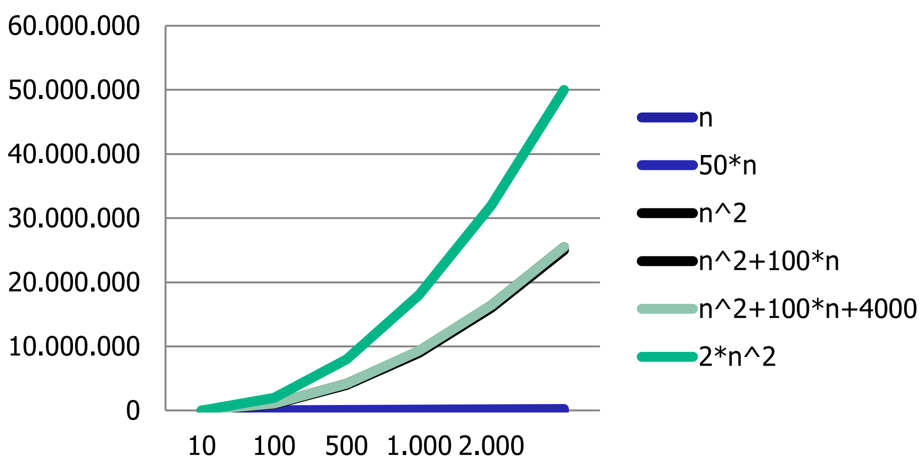
300.000
250.000
200.000
150.000
100.000
50.000
0
10
100
500
1.000
2.000
n
500*n



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

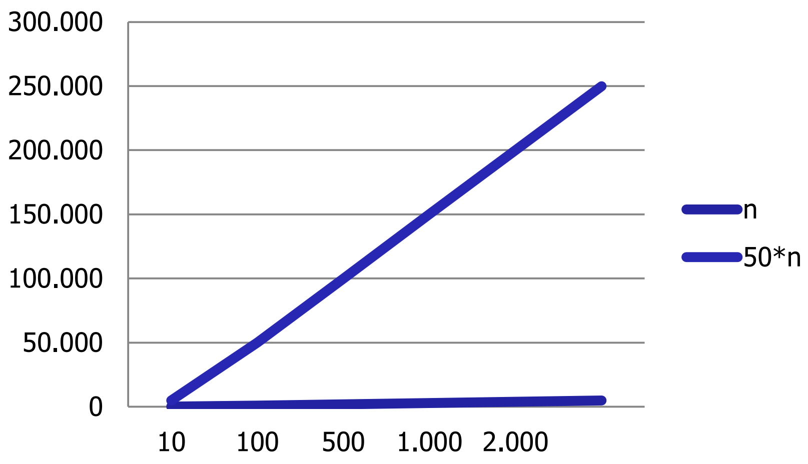


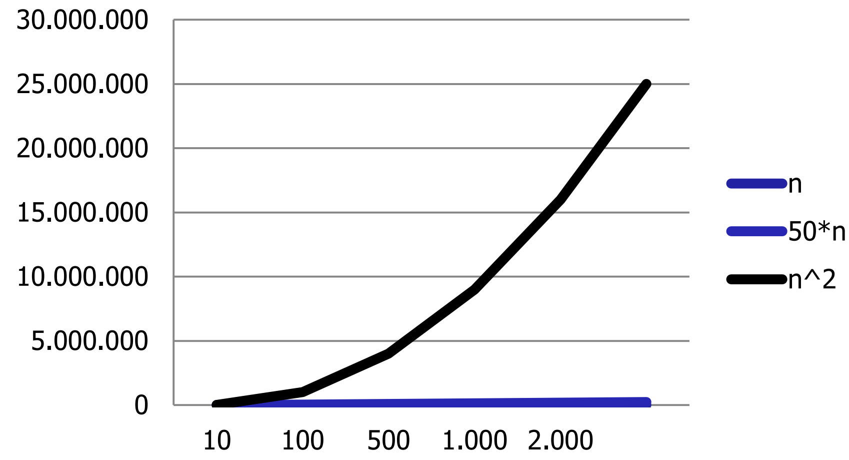
30.000.000
25.000.000
20.000.000
15.000.000
10.000.000
5.000.000
0
10
100
500
1.000
2.000
n
50*n
n^2
n^2+100*n
n^2+100*n+4000

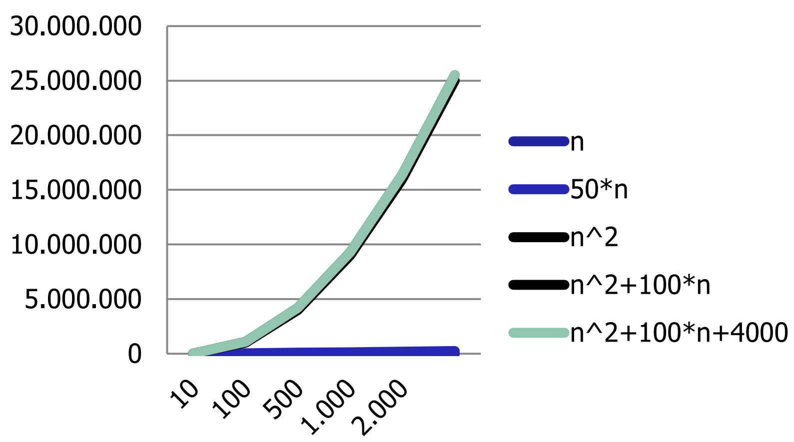


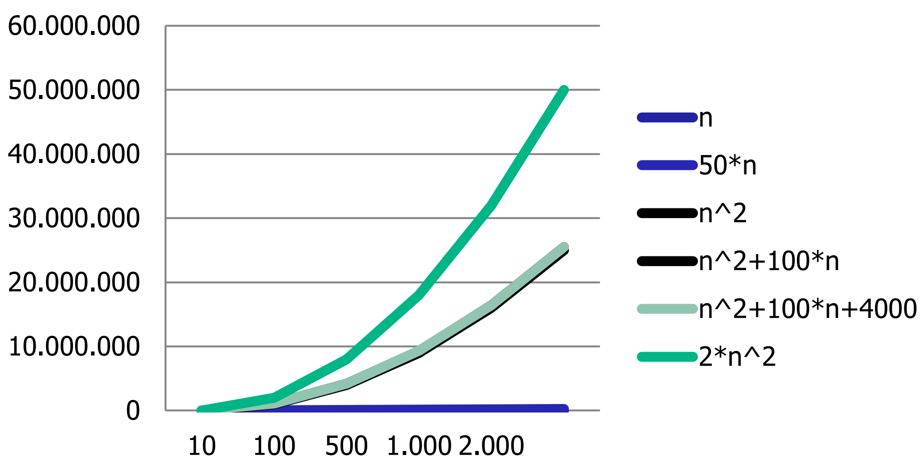
60.000.000
50.000.000
40.000.000
30.000.000
20.000.000
10.000.000
0
10
100
500
1.000
2.000
n
50*n
n^2
n^2+100*n
n^2+100*n+4000
2*n^2

Examples

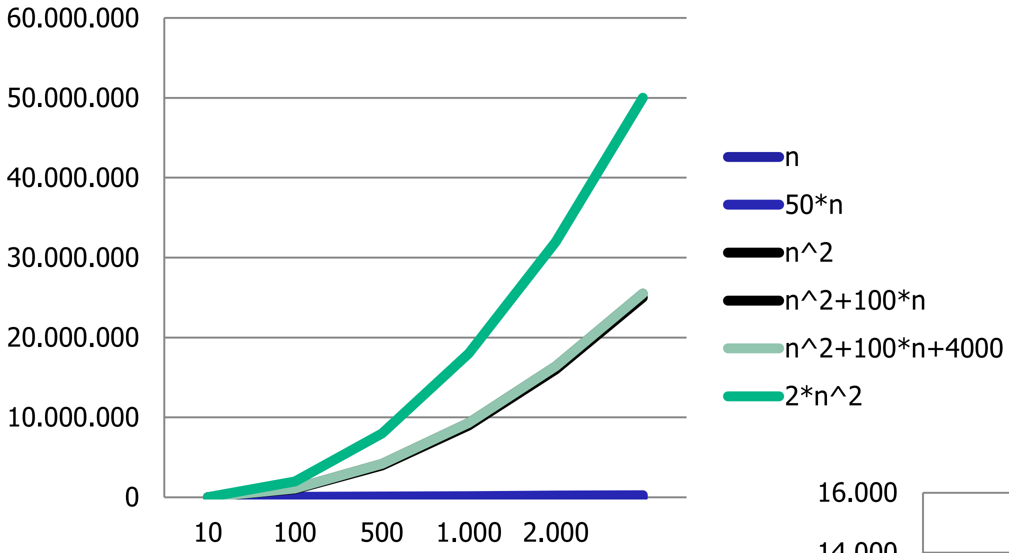

300.000
250.000
200.000
150.000
100.000
50.000
0
10
100
500
1.000
2.000
n
50*n

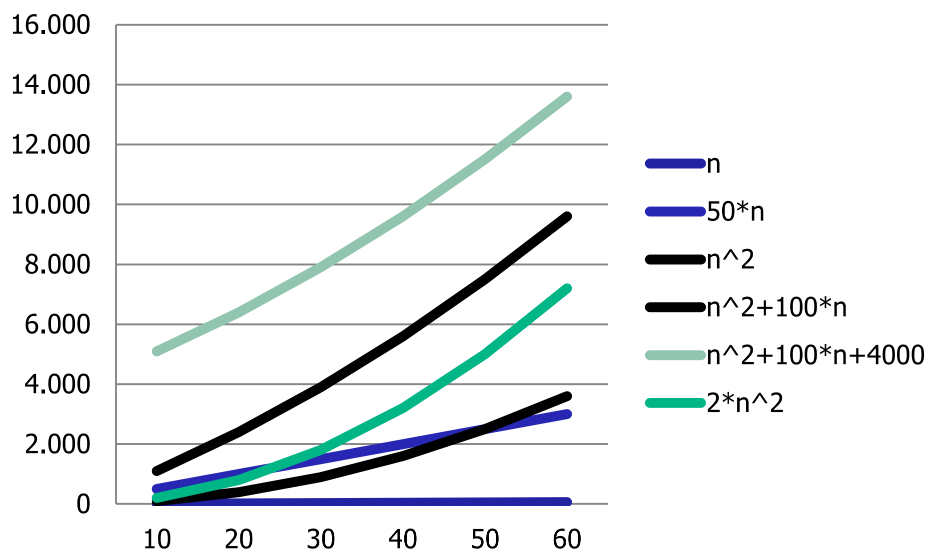

30.000.000
25.000.000
20.000.000
15.000.000
10.000.000
5.000.000
0
10
100
500
1.000
2.000
n
50*n
n^2


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 once n is large enough
  - Therefore, we will ignore all other
- This term can have a factor, but the effect of this factor usually can be outweighed by newer/more machines
  - Therefore, we will not consider it

- Assume we have developed a polynomial $f(n)$ capturing the exact cost of an algorithm A for input size n
- Intuitively, the complexity of A is the term in f with the highest exponent after stripping linear factors

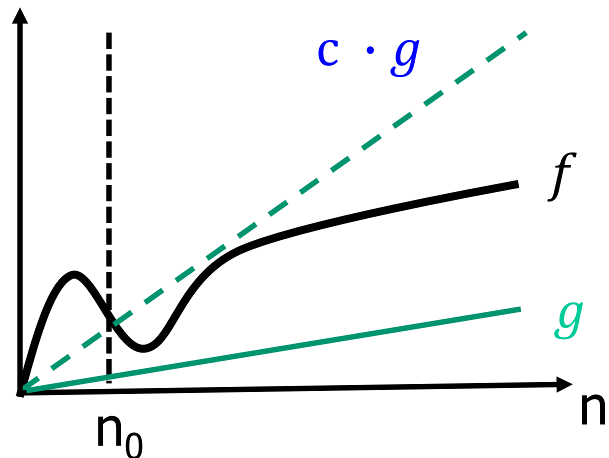
Overview

- Assume $f(n)$ gives the number of operations performed by algorithm 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 the set $O(g)$ as the class of functions that is asymptotically smaller than or equal to g
  - If $f \in O(g)$, then f will be asymptotically smaller than or equal to g
    - I.e.: for large input sizes, 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
    - We want a simple g; simpler than f
    - Not necessarily the lowest

Formally: O-Notation

- Definition
  Let $g\colon \mathbb{N}_0^+ \rightarrow \mathbb{R}_0^+$. $O(g)$ *is the class of functions defined as*
  $O(g) = \{f\colon \mathbb{N}_0^+ \rightarrow \mathbb{R}_0^+ \mid \exists c > 0, \exists n_0 \geq 0, \forall n \geq n_0\colon f(n) \leq c \cdot g(n)\}$
- 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

$$O(g) = \{f: \mathbb{N}_0^+ \rightarrow \mathbb{R}_0^+ \mid \exists c > 0, \exists n_0 \geq 0, \forall n \geq n_0: f(n) \leq c \cdot g(n)\}$$

1. $f(n) = 3n^2 + 6n + 7$ is $O(n^2)$
2. $f(n) = n^3 + 7000n - 300$ is $O(n^3)$
3. $f(n) = 4n^2 + 200n^2 - 100$ is $O(n^2)$
4. $f(n) = log(n) + 300$ is $O(log(n))$
5. $f(n) = log(n) + n$ is $O(n)$
6. $f(n) = n \cdot log(n)$ is $O(n \cdot log(n))$
7. $f(n) = 10$ is $O(1)$

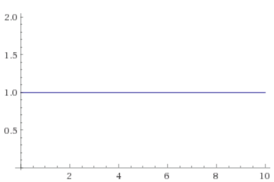
8. $f(n) = n^2$ is $O(n^3)$ but also $O(n^2)$

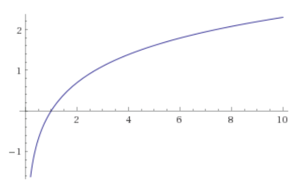
   or $O(n^4)$, $O(n^2 logn)$, …

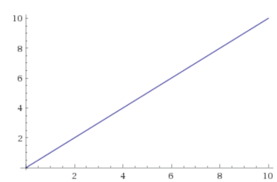
- Proof-Example: First f(n)
  - We need to show:
    $f(n) \in O(n^2) \Rightarrow \exists c \, \exists n_0: f(n) \leq cn^2$
  - Choose $c = 16$ and $n_0 = 1$
  - Now, for n>1=n_0:
    $\Rightarrow 3n^2 + 6n + 7$
    $\leq 3n^2 + 6n^2 + 7n^2$
    $= 16n^2 = cn^2$
  - Would also work for c=17,18, …
- Concrete choice of values of c and n_0 don't matter
  - Especially: No need to search for smallest values for proving complexity

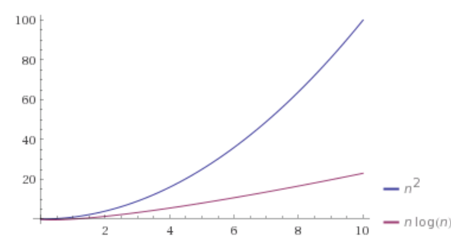
Common Complexity Classes

• $O(1)$:	constant	(Array Access)
• $O(\log n)$:	logarithmic	(Binary Search)
• $O(n)$:	linear	(Sequential Search)
• $O(n \log n)$:	loglinear	(MergeSort)
• $O(n^2)$:	quadratic	(Selection Sort, BubbleSort, QuickSort)
• $O(n^k)$:	polynomial	(Floyd-Warshall)
• $O(2^n)$:	exponential	(Knapsack Problem)

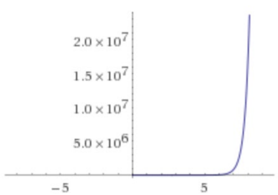

constant


logarithmic


linear



loglinear polynomial


exponential

General Result

- Lemma: *All constant functions are in* $O(1)$
  - All $f(n) = k$ for some constant $k > 0$
- Examples:
  - $f(n) = 10^6$ is $O(1)$
  - $f(n) = 3$ is $O(1)$
- Proof:
  - Let $g(n) = 1$
  - We need to show that $f \in O(g) \Leftrightarrow k \in O(1) \Rightarrow \exists c\ \exists n_0: k \leq c \cdot 1$
  - Chose $c = k$ and $n_0 = 0$
  - Clearly: $\forall n \geq n_0$, we now have $f(n) = k \leq c \cdot g(n) = k \cdot 1$
- Any part of an algorithm whose extend of work is independent of input size n is summarized as $O(1)$

Computational Complexity and Program Analysis

- Computational complexity not only leads to short formulas
- It also makes program analysis much easier
- We show that computing the complexity of a program p by aggregating the complexities of its individual steps is much simpler then first finding f, i.e., the real cost of p by aggregating the real cost of individual steps

- We need rules to combine the complexity of steps into the complexity of subprograms

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;
```

- 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 (line 3), we can forget the startup which is $O(1)$

Formally: O-Calculus

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

with "slight misuse of notations":

Let $f_0 \in O(f)$ *and* $g_0 \in O(g)$ then

- $f_0 + g_0 \in O(\max(f,g))$
- $f_0 \cdot g_0 \in O(f \cdot g)$

- 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
- Algorithm:
  - Note: There are more efficient ones
- Note: We have two inputs, and both must be considered

```
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;
```

Example

- Example
  - We use two counters: i, j
  - One (outer, i) runs through T
  - One (inner, j) runs through P

```
   123456789...
T  ctgagatcgcgta
P  gagatc
    gagatc
     gagatc
      gagatc
       gagatc
        gatatc
         gatatc
          gatatc
```

```
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) · O(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)
```

Transitivity of O-Membership

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

    $\forall n \geq n''_0\colon f(n) \leq c \cdot g(n) \leq c \cdot c' \cdot h(n) \leq c'' \cdot h(n)$
  - q.e.d.

Ω-Notation

- O-Notation denotes an upper bound for the amount of computations necessary to run an algorithm for asymptotically large inputs
  - "f will always be faster than g on large inputs"
- Sometimes, we also want lower bounds
  - "f will never be faster than g on large inputs"
- Definition
  Let $g: \mathbb{N}_0^+ \rightarrow \mathbb{R}_0^+$*. $\Omega(g)$ is the class of functions defined as*
  $\Omega(g) = \{f: \mathbb{N}_0^+ \rightarrow \mathbb{R}_0^+ \mid \exists c > 0, \exists n_0 \geq 0, \forall n \geq n_0: f(n) \geq c * g(n)\}$
- Explanation
  - $\Omega(g)$ is the class of functions that are asymptotically larger than g
  - Again: Not necessarily the largest smaller one

Examples

$$\Omega(g) = \{f: \mathbb{N}_0^+ \rightarrow \mathbb{R}_0^+ \mid \exists c > 0, \exists n_0 \geq 0, \forall n \geq n_0: f(n) \geq c * g(n)\}$$

$f(n) = 3n^2 + 6n + 7$ is $\Omega(n^2)$ but also $\Omega(n)$, $\Omega(1)$, …

$f(n) = n^3 + 7000n - 300$ is $\Omega(n^3)$ but also $\Omega(n^2)$, $\Omega(n)$, …

$f(n) = log(n) + 300$ is $\Omega(log(n))$ but also $\Omega(1)$, …

$f(n) = 10$ is $\Omega(1)$

$f(n) = n^2$ is $\Omega(n^2)$ but also $\Omega(n)$, $\Omega(\log n)$, …

Further Notation

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

- Interpretation: „f“ is asymptotically...
  1. $f \in O(g)$: … smaller than or equal to „g“
  2. $f \in \Omega(g)$: … larger than or equal to „g“
  3. $f \in \theta(g)$: … exactly like „g“
  4. $f \in o(g)$: … smaller than „g“
  5. $f \in \omega(g)$: … larger than „g“

Reads:

- Big O
- Big Omega
- Theta
- Small O
- Small Omega

Not Every Problem is Simple

- Definition
  We call an algorithm A with cost function f
  - *polynomial, iff there exists a polynomial p with $f \in O(p)$*
  - *exponential, iff $\exists\, \varepsilon > 0$ 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

Questions – Online Quiz

- Please go to **https://pingo.coactum.de**
- Enter ID: **729357**

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|+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;
```

- We showed that the algorithm's WC is $O((n-m) \cdot m) \sim O(n \cdot m)$
  - If we assume $m \ll n$
- What does a worst case look like?

Exact Substring Search: Beyond Worst Case

```
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;
```

- We showed that the algorithm's WC is $O((n-m)\cdot m) \sim O(n \cdot m)$
  - If we assume $m \ll n$
- What does a worst case look like?
  - $T=a^n$ / $P=a^m$

```
T   aaaaaaaaaaaaaa…
P   aaaaaa
     aaaaaa
      aaaaaa
       aaaaaa
         ...
```

- What about the average case?
  - The outer loop is passed by n-m+1 times, no matter what T/P look like
    - This already is in $\Omega(n-m)$
  - Let's look at the inner loop

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?
- Need 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

$$
\begin{aligned}
&= 1(1-p) + 2 \cdot p(1-p) + 3 \cdot p^2(1-p) + \cdots + m \cdot p^{m-1} \\
&= 1 - p \quad + 2p - 2p^2 \quad + \ 3p^2 - 3p^3 \quad + \cdots + \ m \cdot p^{m-1} \\
&= \ 1 \quad + p \quad + \quad p^2 \quad + \quad p^3 \quad + \ \ldots \quad + p^{m-1} = \sum_{i=1}^{m-1} p^i
\end{aligned}
$$

Cost 1 for mismatch at first position; probability is (1-p)

Differences On Real Data

- Assume |T|=50.000 and |P|=8 and |Σ|=29
  - German text, including Umlaute, excluding upper/lower case letters
  - Worst-case estimate: ~400.000 comparisons
    - Note: Here, $O(m \cdot n)$ 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 $T \subseteq S$ such that:

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

Algorithm and its Complexity

- Imagine an algorithm which enumerates all possible subsets 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 subsets 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 $2 \cdot 2 \cdot 2 \cdot \dots \cdot 2 = 2^{|S|}$ 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, $100n$, $n*\log(n)$, $n*2^{\log(n)}$, $\mathrm{sqrt}(n)$, $n!$
- 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 $f \notin O(n^3*\log(n))$