Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
- Examples
  - Multiplication of two binary numbers (unit cost?)
  - Exact substring search (average case versus worst case)
  - Selection sort: Best case = average case in unit cost model
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

• Problems being studied typically have an input and a clearly defined solution
  – 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?

• We always study efficiency wrt. size of the input
Question

• How can we measure efficiency of an algorithm for different input sizes?
Using Reference Machine

- Reference machine
  - Define a **concrete machine** (CPU, RAM, BUS, ...)
  - Chose a **set of different inputs**
    - Uniformly distributed over all possible inputs?
    - Distributed such as we expect real problems to be?
  - Run algorithm on all inputs and **measure times**

- **Pro**: Gives real times
- **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
  - Reference machines are **always out-of-date**
Computational Complexity

• Derive an estimate of the number of performed operations 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?
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
  - What do we count?
- Complexity
- Examples
Machine Model

• Random Access Machines (RAM)
  – Very simple model of a machine
• Roughly covers what traditional CPUs can execute in one clock cycle
  – Forget pipelining, registers, multi-core, disks, arithmetic units, ...

• Storage
  – Infinite amount of storage cells
    • Each cell holds one number
    • Cells are addressed by integers
  – Separate program storage – not 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;     # if 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

- Essentially, we count the number of operations (time) performed and the number of cells (space) required
  - Cells hold infinitely large values
- This is called uniform cost model
  - 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

- If $y > 1$
  - Startup costs 4
  - Loop (lines 5-9) costs 5
  - Loop is passed by $y - 1$ times
  - Return costs 1
  - Total costs: $4 + (y - 1) \times 5 + 1$

- If $y = 1$
  - Total costs: $7 = 4 + (y - 1) \times 5 + 1$

- Here: Best-case = average-case = worst-case

- Shorter: $f(y) \sim c_0 + c_1 \times y$ for some constants $c_0, c_1$
Uniform versus Machine Cost

- 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 $m$ comparisons in worst-case
    - Lines 6-8; additional cost for loops for copying char-by-char
- Now, average-case makes sense
  - Given two strings, how many characters do we have to compare on average to see which is greater?
Conclusions

- We usually assume RAM model and uniform cost, but will not give the RAM program itself
  - Translation from pseudo code to our “language” is simple
- Counting has an end
  - We usually will assume UCM for operations on numbers and strings
  - All more complex data type (lists, sets) will be analyzed in detail
- When analyzing Java programs, be more careful
Content of this Lecture

- Efficiency of Algorithms
- Machine Model
- Complexity
  - How do we estimate?
- Examples
Complexity

- Counting the exact number of operations for an algorithm (wrt. input size) seems overly complicated
  - We want to concentrate on the major factors influencing runtime
  - 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 want to focus on behavior for large inputs
  - Asymptotic complexity – behavior if input size goes to infinity

- This allows to focus on the most cost-intensive operations
Examples

- Graph 1: Comparison of $n$ and $50n$
  - $n$: Linear growth
  - $50n$: Steep linear growth

- Graph 2: Comparison of $n^2$ and $n(n+100)$
  - $n^2$: Quadratic growth
  - $n(n+100)$: Quadratic growth with a linear term

- Graph 3: Comparison of $n^2$, $n^2+100n$, and $n^2+100n+4000$
  - $n^2$: Quadratic growth
  - $n^2+100n$: Quadratic growth with a linear term
  - $n^2+100n+4000$: Quadratic growth with a linear term and a constant term

- Graph 4: Comparison of $n^2$, $2n^2$, and $n^2+100n+4000$
  - $n^2$: Quadratic growth
  - $2n^2$: Quadratic growth with a constant factor of 2
  - $n^2+100n+4000$: Quadratic growth with a linear term and a constant term
Small Values
Intuitive Approach

• Apparently, 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 (in theory)
• Assume we have developed a formula $f$ 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
**Formally: O-Notation**

- **Definition**
  
  Let \( g: \mathbb{N} \to \mathbb{R}^+ \). \( O(g) \) is the class of functions defined as
  
  \[
  O(g) = \{ f: \mathbb{N} \to \mathbb{R}^+ | \text{there exist positive constants } c_1, n_0 \text{ with } f(n) \leq c_1 g(n) \ \forall n \geq n_0 \}
  \]

- **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

- **Example: First f**
  - Chose $c_1=9$ and $n_0=7$
  - For any $n>7$: $n^2 > 6n + 7$
  - With $3n^2 + 6n + 7 = 9n^2$ we have $3n^2 + 6n + 7 \leq 9n^2$

- **Note: The exact values of $c_1$ and $n_0$ are not important**
  - Especially: No need to search for minimal 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

- Usually, we want to know the complexity of a program without calculating its exact cost

- Some observations
  - Having many operations with cost 1 is the same as having only one
    - Lines 5-8 cost 4 times 1 ~ 1
  - As soon as we see a higher 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
  \textit{Let k be a constant. The following \textit{equivalences are true}}
  \begin{itemize}
  \item \(O(k+f) = O(f)\);
  \item \(O(k*f) = O(f)\);
  \item \(O(f) + O(g) = O(\max(f,g))\)
  \item \(O(f) * O(g) = O(f*g)\)
  \end{itemize}
- Explanations
  \begin{itemize}
  \item Rule 3 (4) actually implies rule 1 (2), as \(k \in O(1)\)
  \item Rule 3 is used for \textit{sequentially executed parts} of a program
  \item Rule 4 is used for \textit{nested parts} of a program (loops)
  \end{itemize}
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

```
1. for i = 1..|T|-|P| 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;
```
Complexity Analysis \((n=|T|, m=|P|)\)

1. for \(i = 1..|T|-|P|\) 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. \(O(m)\)

1. \(O(n)\)
2. \(O(1)\)
3. \(O(1)\)
4. \(O(1)\)

1. \(O(n)\)
2. \(O(1)\)
3. \(O(m)\)
4. \(O(1)\)

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

1. \(O(n)\)
2. \(O(1)\)
3. \(O(m)\)
4. \(O(m)\)

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

\[O(1)+O(1)=O(1)\]

\[O(1)+O(m)=O(m)\]

\[O(1)+O(m)=O(m)\]

\[O(n)+O(m)=O(n+m)\]

\[O(n)\times O(m)=O(n\times m)\]

\[O(n)\times O(m)=O(n\times m)\]
**Ω-Notation**

- **O-Notation** gives us an **upper bound** for the amount of computation necessary to run an algorithm for asymptotically large inputs
  - Note: Not necessarily the **lowest upper bound**
- Sometimes, we also want **lower bounds**
- **Definition**
  \[
  \Omega(g) \text{ is the class of functions defined as } \Omega(g) = \{ f: N \rightarrow \mathbb{R^+} \mid \text{there exist positive constants } c_1, n_0 \text{ with } g(n) \leq c_1 * f(n) \ \forall n \geq n_0 \} \]
- **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

- We showed that the algorithm is $O(n \times m)$ in worst-case.
- How does a worst case look like?

```
1. for i = 1..|T|−|P| 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;
```
Exact Substring Search: Average Case

- 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 its average-case complexity?
  - Again, we can forget about all assignments etc.
  - The outer loop is always passed by $n$ times, no matter how $T / P$ look like
    - This already gives $\Omega(n)$

```plaintext
1. for $i = 1..|T|-|P|$ 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;
```
Exact Substring Search: Average Case

- How often do we pass by the inner loop?
- Needs a model of “average strings”
- Simplest model:
  Strings are randomly generated from alphabet $\Sigma$
  - 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 gives the expected number of comparisons:
  $$1(1-p)+2*p(1-p)+3*p^2(1-p)+...+m*p^{m-1}(1-p)=
  1 - p + 2p - 2p^2 + 3p^2 - 3p^3 + ... m*p^{m-1} - m*p^m =
  1 + p + p^2 + p^3 + ... p^{m-1} - m*p^m =
  -mp^m + \sum_{i=0}^{m-1} p^i$$
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: $\sim 51,851$ 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 $O(m+n) \sim 50,008$ comparisons
  - Beware: We ignore constant factors
- Not much better than the average case
- But: Are German texts random strings?
Knapsack Problem

- 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
  - See exercise
- 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 $T$
  - See exercise
- 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 $2 \times 2 \times 2 \times \ldots \times 2 = 2^{|S|}$ 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