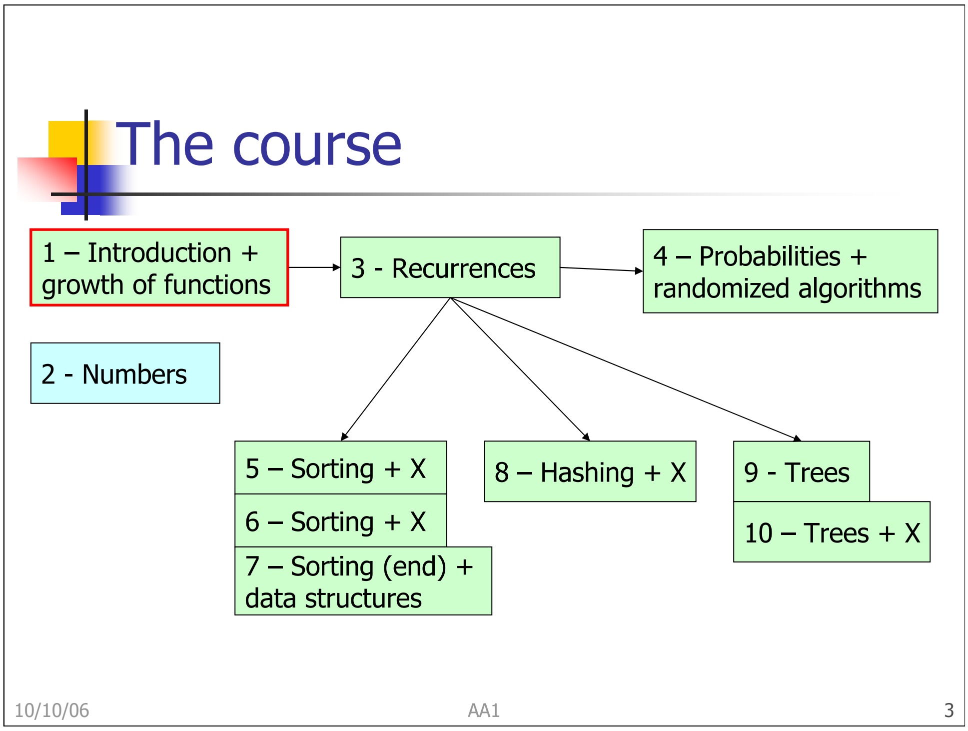
Algorithms & Architecture Introduction + Growth of Functions

Alexandre David
B2-206

Goals – The course

- Analysis (and design) of algorithms.
  - asymptotic behavior, complexity, recurrences
- Sorting algorithms
  - bubble/quick/merge/heap-sort
- Data structures
  - stacks, queues, lists, hash tables, binary/red-black trees
- Others – you ask (e.g. Linux scheduling algorithm, how to compute polynomials).

This is also your checklist for the end of the course. You should be familiar with all the concepts mentioned here.



That's a simple dependency between the lectures. You can ask for something earlier as long as it does not break the dependency.

Goals - Today

- Notion of algorithms.
  - GCD example.
- Algorithmic problem solving.
- Problem types.
  - Sorting example.
  - Numerical example.
- What it means to analyze algorithms.

Notion of Algorithms

- Why study algorithms?
- What is an algorithm?
- Example: GCD – greatest common divisor.
  - Simple and clear requirement.
  - Define range of inputs.
  - Different algorithms to solve it.
  - Different ideas & running speeds.

Why study? – Practical reason: to know a standard set of algorithms and not reinvent the wheel; Design new algorithms & analyze their efficiencies. Theoretical reason: cornerstone of computer science. There is no computer program without algorithm.

What is an algorithm? – Sequence of **unambiguous** instructions for solving a problem, i.e., for obtaining a required output for any **legitimate** input in a **finite** amount of time. Goal in the study: to understand what is happening and why it takes long or not to execute.

GCD: The problem

- Greatest common divisor of 2 non-negative integers, denoted *gcd(m,n)*, defined as the largest integer that divides both m and n with a remainder of zero.
- Algorithms:
  - Consecutive integer checking.
  - Euclid's algorithm.
  - Prime decomposition.

10/10/06 AA1 6

Give a clear specification of **inputs** and **outputs**.

Consecutive integer checking

- Idea: Solution cannot be greater than min(m,n). Let *t=min(m,n)*. Check *t* and try again by decreasing *t*.
- ? Correctness: greatest? Termination?
- ? Efficiency: (worst case) execution time?

```
1: t ← min(m,n)
2: if m mod t == 0 then 3: else 4:
3: if n mod t == 0 return t else 4:
4: t ← t-1
5: go to step 2:
```

!

Correctness: argue for right solution *and* termination.

Complexity question: size of the input or value of the input?

Oops: what if t == 0 from the beginning (m or n == 0)?

Euclid's algorithm

- Idea: Apply repeatedly *gcd(m,n)=gcd(n, m mod n)* until *m mod n == 0*. Ex: gcd(60,24)=gcd(24,12)=gcd(12,0)=12.
- ? Correctness? Termination? Efficiency?

```
Algorithm Euclid(m,n)
// Input: two non-negative, non both zero integers m and n.
// Output: gcd(m,n).
while n != 0 do
        r := m mod n
        m := n
        n := r
done
return m
```

!

There can be different description formats for algorithms.

Prime decomposition

- Idea: Decompose m and n into primes and pick the common factors.

1: Find the prime factors of m.
2: Find the prime factors of n.
3: Identify all the common factors – If p is a common factor occurring pm and p_n times in m and n, respectively, it should be repeated $\min(p_m, p_n)$ times.
4: Return the product of all the common factors.

- Problem: Non-trivial sub-problems to be solved.

Finding primes is expensive.

This is an expensive and complex algorithm (even if it is the one we learn at school).

Algorithmic problem solving

- Understand the problem.
- Choose exact/approximate solution.
- Decide on appropriate data structures.
- Apply an algorithm design technique.
- Specify the algorithm.
- Prove the correctness of the algorithm.
- Analyze the algorithm – time & space – simplicity – generality.
- Code the algorithm. ← Supposed to be at the end!

These are the steps to design and analyze algorithms. Do not underestimate understanding the problem – by examples & special cases.

Approximate technique when the problem cannot be solved exactly or it may be too expensive to get an exact solution – intrinsic difficult problems.

Algorithm design technique: divide-and-conquer, brute force, follow a proof technique, equation solving, …

Check the generality of the problem solved the accepted – Are 2 integers relatively primes? Checking for GCD is easier (gcd == 1).

Right choice of algorithm = several orders of magnitude of performance difference. Code tuning = constant factor improvement. Of course 2x faster is worth but it is minor.

Problem types ?

- Sorting
  - stable? in place?
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

10/10/06 AA1 11

Sorting: Rearrange items in ascending (or descending) order. There must be a total order on the set. Useful for other algorithms, used everyday for practical purposes. **Stable** algorithm: It preserves the order of 2 equal elements. **In place** algorithm: It does not require extra memory (apart from a constant overhead).

Searching: Given a key, find a value. How to organize big sets of data for efficient search?

Strings: string matching.

Graph problems: traversal, shortest path, coloring…

Combinatorial: Find a combinatorial object satisfying a set of constraints and has some property (a max/min cost). Difficult in general.

Geometric problems: closest pair, convex hull, circuit layout.

Numerical problems: equations and system of equations.

Sorting example

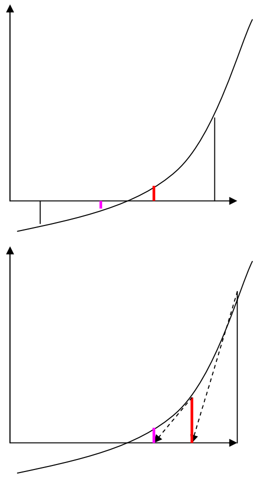
- Sorting problem:
  Input: a sequence of *n* numbers
  $\langle a_1, a_2, \ldots, a_n \rangle$.
  Output: a permutation (re-ordering)
  $\langle a_1', a_2', \ldots, a_n' \rangle$ of the input s.t.
  $a_1' \le a_2' \le \ldots \le a_n'$.
- Algorithms to solve it: insertion sort, merge sort, quicksort… Insertion sort takes $c_1 n^2$ in time, merge sort takes $c_2 n\lg(n)$.

Good algorithm vs. tuning

- Let's sort 10^6 elements (*only* 1 million).
- Optimized insertion sort @ 1GHz:
  $2n^2 \rightarrow 2(10^6)^2/10^9 = 2000s$.
- Average merge sort @ 10MHz:
  $50n\lg(n) \rightarrow 50*10^6*\lg(10^6)/10^7=100s$.
- Moore's law: 2x every 18 month is not enough.

Numerical example

- Find x s.t. f(x) = 0 for a continuous monotonic function.
  - Bisection algorithm:
    - Iterate on $[x,y]^0,[x,y]^1\ldots$ s.t. $f(x)<0$ and $f(y)>0$ (or opposite).
    - Reduce interval by 2 everytime.
  - Newton-Raphson algorithm:
    - Use derivative with $x_{i+1}=x_i-f(x_i)/f'(x_i)$. Faster convergence.
- Flat or exponential functions.

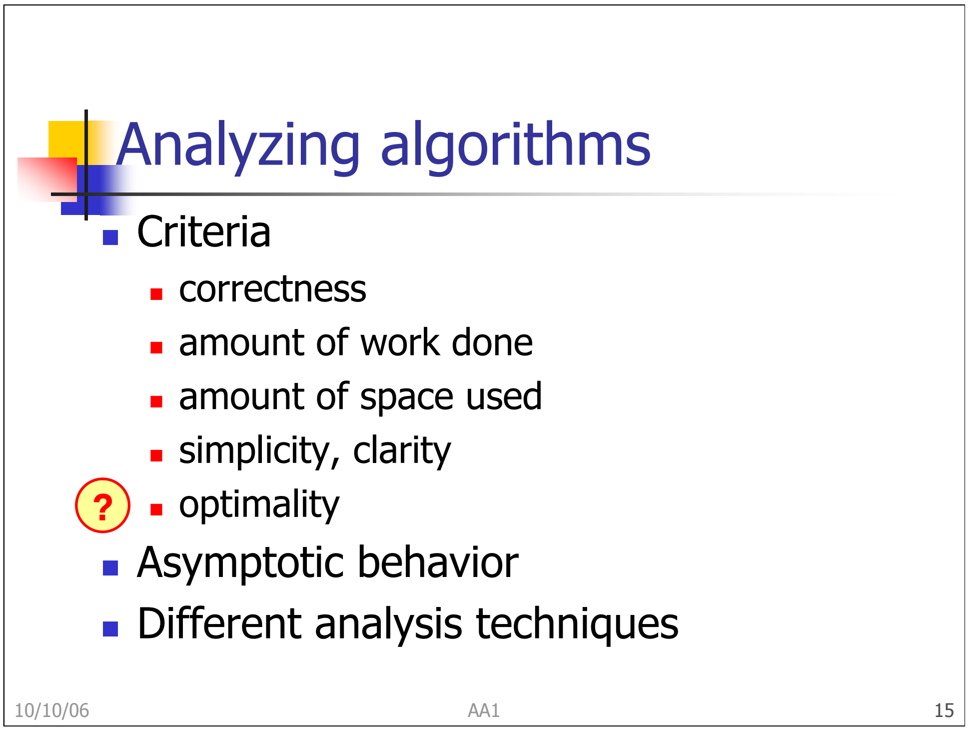


10/10/06 AA1 14

Particular cases:

•multiple zeros.

•exponential or very flat functions.



Correctness: What does “correct” mean? Input (pre-condition) and output (post-condition) are valid. Prove theorems if needed, check implementation.

Work: Efficiency of the method (not just execution time). We want a machine (and instruction + language) independent analysis technique.

Optimality: Problems have some inherent complexity. Optimal means *best possible*.

Analysis: Induction technique, recursion tree, straight-forward proof, math theorem… to match different kinds of algorithms, e.g., brute force, divide-and-conquer.

Asymptotic behavior

- Why do we care?
- What happens for *large* instances of the problems?
- How to compare different algorithm?

- Asymptotic running time of algorithm: $n \to +\infty$

Asymptotic notations

- Θ-notation: asymptotic tight bound.
  $\Theta(g(n))=\{f(n) \mid \exists c_1>0, c_2>0, n_0\geq 0.$
  $\forall n\geq n_0, 0\leq c_1 g(n)\leq f(n)\leq c_2 g(n)\}$
  $\Theta(g(n))$ is a set so we write $f(n)\in \Theta(g(n))$.
- O-notation: asymptotic upper bound.
  $O(g(n))=\{f(n) \mid \exists c>0, n_0\geq 0.$
  $\forall n\geq n_0, 0\leq f(n)\leq cg(n)\}$
  $O(g(n))$ is a set so we write $f(n)\in O(g(n))$.
- Ω-notation: asymptotic lower bound.
  $\Omega(g(n))=\{f(n) \mid \exists c>0, n_0\geq 0.$
  $\forall n\geq n_0, 0\leq cg(n)\leq f(n)\}$
  $\Omega(g(n))$ is a set so we write $f(n)\in \Omega(g(n))$.

Asymptotic notations

- Why is there a n_0 in the definition?

- O weaker than Θ: $\Theta(g(n)) \subseteq O(g(n))$.
- Ω weaker than Θ: $\Theta(g(n)) \subseteq \Omega(g(n))$.
- Theorem:
  $f(n)=\Theta(g(n))$ iff $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$.
  Abuse of notation = instead of $\in$.

10/10/06 AA1 18

It does not matter what happens before n_0. We are interested in the asymptotic behavior.

Insertion sort: $T(n)= \Theta(n^2)$.

Asymptotic notations

- o-notation: upper bound not asymptotically tight.
  $o(g(n))=\{f(n) \mid \forall c>0, \exists n_0, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$.
  $2n \in o(n^2)$ but not $2n^2$.
- ω-notation: lower bound not asymptotically tight.
  $\omega(g(n))=\{f(n) \mid \forall c>0, \exists n_0, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$.
  $n^2/2 \in \omega(n)$ but $n^2/2 \notin \omega(n^2)$.

10/10/06 AA1 19

Check properties of the different asymptotic notations p 49.

Standard notations and common functions

- Read section 3.2. Good to know.
- Fibonacci numbers.