Advanced Algorithms

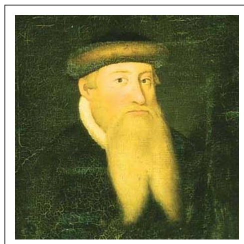
Floriano Zini

Free University of Bozen-Bolzano
Faculty of Computer Science

Academic Year 2013-2014

Lecture 1 - Preliminaries

Typography vs algorithms



Johann Gutenberg (c. 1398 – February 3, 1468)

- In 1448 in the German city of Mainz a goldsmith named **Johannes Gutenberg** discovered a way to **print books** by putting together movable metallic pieces
- Many **historians** claim that with the invention of **typography**, the literacy spread, the Dark Ages ended, the human intellect was liberated, science and technology triumphed, the Industrial Revolution happened
- But **others** insist that the **key development** was not typography, but **algorithms**

Typography vs algorithms

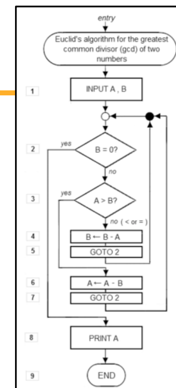
- Gutenberg would write the number 1448 as MCDXLVIII
  - How do you **add** two **Roman numerals**?
  - What is MCDXLVIII + DCCCXII?
    I.e., $(1448)_{10} + (812)_{10}$
- The **decimal system** was invented in India around AD 600
  - Using only **10 symbols**, even very large **numbers** could be **written** down **compactly**
  - **Arithmetic** could be done efficiently on them by following elementary steps

The "invention" of the algorithm

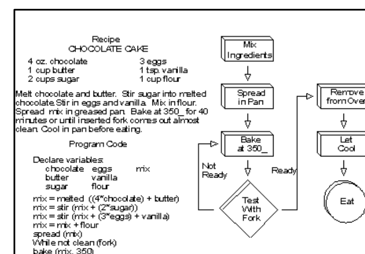
- **Al Khwarizmi** laid out the **basic methods** for
  - **adding**, **multiplying**, **dividing** numbers
  - extracting **square roots**
  - calculating **digits** of π
- These procedures were **precise**, **unambiguous**, **mechanical**, **efficient**, **correct**
- They were **algorithms**, a term coined to honor the wise man after the decimal system was adopted in Europe, in the 12th century

Al Khwarizmi (c.780 – c.850)

Algorithms



- No generally accepted *formal* definition of algorithm
- A "**mathematical**" definition
  - An algorithm is an **effective method** expressed as a finite **list** of well-defined **instructions** for calculating a **function**
- A **simpler** definition
  - an algorithm is a **step-by-step procedure,** a set of rules that precisely defines a **sequence** of **operations**



Computational problem

- The field of algorithms studies methods to **efficiently** solve computational problems
- In a computational problem, we are given an **input** and we want to return as **output** a solution satisfying some **properties**
- The **structure** of a computational problem is a **description** in terms of **parameters** and a wanted **result**
  - An **instance** of a problem can be obtained by **specifying** all the **parameters**

Computational problem

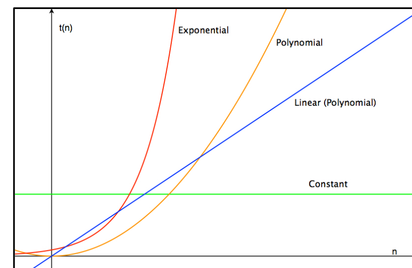
- **Decision problem**: a computational problem where the answer for every instance is either yes or no
  - **example**: *primality testing - Given a positive integer n, determine if n is prime*
- **Search problem**: a computational problem where answers can be arbitrary strings
  - **example**: *factoring is a search problem where the input is a positive integers and the solution is its factorization in primes*
- **Counting problem**: it asks for the number of solutions to a given search problem
  - **example**: *the counting problem associated with factoring - Given a positive integer n, count the number of prime factors of n"*
- **Optimization problem**: it asks for the "best possible" solution among the set of all possible solutions to a search problem
  - **example**: *Given a weighted graph G find a path between two nodes of G having the maximum weight*

1. Given the map of a city which is the shortest route for going from the city hall to the dome?
   Optimization
2. Given a graph can it be colored with 2 colors?
   Decision
3. Two dice are rolled, what is the number of possible outcomes?
   Counting
4. Given an array of integers which is the minimum element?
   Search

1. Given the map of a city which is the street with the shortest name?
   Search
2. We want to construct a box with a square base by using only 10 m^2 of material, what is the maximum volume that the box can have?
   Optimization
3. How many different seven-digit phone numbers can be used if no phone number begins with a 0 and no digit repeats?
   Counting
4. Given a positive integer, is it a product of two smaller positive integers?
   Decision

Computational cost

- The **execution** of an algorithm requires computational **resources**: **space**, **time**, etc.
- The cost or **computational complexity** of an algorithm is **usually** measured in terms of **time** required
- Generally, the **time** required **grows** with the size of the **input**
- A key factor is **how fast the time grows**



Leonardo Fibonacci

Leonardo Fibonacci (c. 1170 – c. 1250)

Fibonacci helped the **spread** of the **work of Al Khwarizmi** and the **decimal system** in **Europe**, primarily through the publication in the early 13th century of his Book of Calculation, the **Liber Abaci**

Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34,...

Formally

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

How can we exactly calculate F_{100} or F_{200}?

A first algorithm

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

Three (recurrent) questions

1. Is the algorithm **correct**?
2. How much **time** does it take, as a function of *n*?
3. Can we do **better**?

For *fib1* the answer to the first question is trivially true, as the algorithm precisely mimics the definition of F_n

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

How much time?

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

- Let $T(n)$ be the number of computer steps needed to compute *fib1(n)*
  - For $n \leq 1$: $T(n) \leq 2$
  - For $n > 1$: $T(n) = T(n-1) + T(n-2) + 3$
- Comparing with the recurrence relation of F_n

$$T(n) \geq F_n \approx 2^{0.649n}$$

- The computation time is **exponential** in *n*

Why exponential is bad?

$$T(200) \geq F_{200} \approx 2^{138} \approx 3.5 * 10^{41}$$

Let's consider the fastest computer in the world

As of June 2013, China's **Tianhe-2** supercomputer is the fastest in the world at $33.86*10^{15}$ FLOPS (**FL**oating-point **O**perations **P**er **S**econd)

To compute F_{200} Tianhe-2 needs roughly

$$10^{25} \text{ seconds} \approx 3*10^{17} \text{ years}$$

Even if we started computing from the Big Bang, which is roughly $1.4*10^9$ years ago, only a fraction had been completed!!

What about Moore's law?

Computer speeds have been doubling roughly every 18 months = 1.5 years

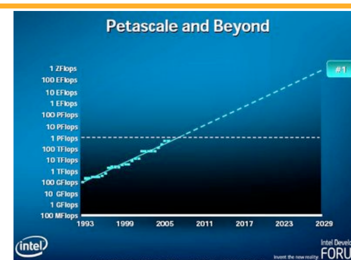
- The running time of *fib1(n)* is proportional to

  $$2^{0.694n} \approx (1.6)^n$$

- It takes 1.6 times longer to compute F_{n+1} than F_n
- if we can reasonably compute F_{100} with this year's technology, then next year we will manage F_{101}. And the year after, F_{102}. And so on:

  just one more Fibonacci number every year!

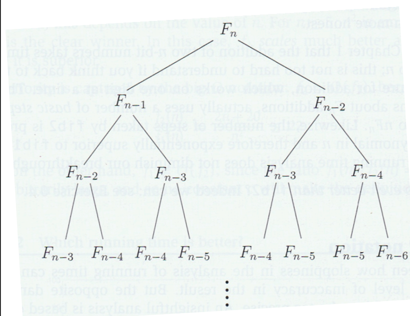
The improvement of computer speed does not cope with exponential complexity



A better (polynomial) algorithm

- Now we know that *fib1(n)* is **correct** but **inefficient**
- Can we do better?

Too many computations are repeated



Use iteration!

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

A better (polynomial) algorithm

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

- The **correctness** of *fib2(n)* is **trivial** as it directly use the definition of F_n
- **How long** does it take?
  - The **inner loop** consists of a **single computer step** and is executed ***n-1* times**
  - Therefore the number of **computer steps** used by *fib2* is **linear in *n***
- **From exponential to polynomial!**

A more careful analysis

- We have been counting the number of **basic computer steps** executed by each algorithm and thinking of these basic steps **as taking a constant amount of time**
- It is reasonable to treat **addition** as a **single computer step** if **small numbers** are being added, e.g., **32-bit numbers**
- But the nth **Fibonacci** number is about ***0.694n*** bits long, and this can far exceed *32* as *n* grows
- **Arithmetic operations on arbitrarily large numbers cannot be performed in a single, constant-time step**

Why is F_n about 0.694n bits long?

1. $F_n \approx 2^{0.649n}$
2. To represent $(x)_{10}$ in base 2 we need $log_2(x)$ bits
3. $log_2(2^{0.649n}) = 0.649n$

A more careful analysis (cont.)

- The **addition of two *n*-bit numbers** takes **time** roughly **proportional to *n*** (see next lecture)
- ***fib1***, which performs about F_n additions, uses a **number of basic steps** roughly **proportional** to nF_n

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

- The **number of steps** taken by ***fib2*** is **proportional** to n^2, still polynomial in n and therefore **exponentially superior to *fib1***

```
function fib2(n)
if n = 0 return 0
create an array f[0...n]
f[0] = 0, f[1] = 1
for i = 2...n:
    f[i] = f[i - 1] + f[i - 2]
return f[n]
```

Counting the number of steps

- **Sloppiness** in the analysis of running times can lead to unacceptable **inaccuracy**
- But it is also possible to be **too precise to be useful**
  - E.g., accounting for architecture-specific details is too complicated and yields a result that does not generalize from one computer to the next
- It makes more sense to seek a **machine-independent characterization** of an algorithm's **efficiency**
- To this end, we will always express **running time** by **counting the number of basic computer steps**, as a **function** of the size of the **input**

Counting the number of steps (cont.)

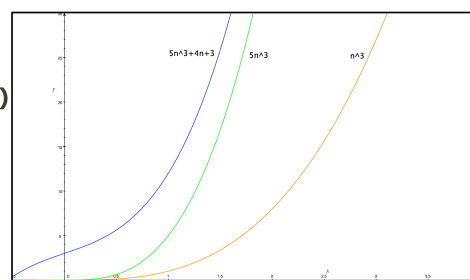
- **Execution time**:
  given an input and an algorithm A, the execution time of A is the number of its basic computer steps
- Two instances of the same problem having the **same size** may require **different execution times**
- Computational complexity
  - **Worst case**: The complexity is computed according to the instance that require the highest execution time
  - **Best case**: The complexity is computed according to the instance that require the lowest execution time
  - **Average case**: The complexity is computed as the mean value of the execution time of all the instances of that size

Counting the number of steps (cont.)

Instead of reporting that an algorithm takes, say,
$5n^3 + 4n + 3$ steps on an input of size n, it is much simpler to **leave out**

- **lower-order terms** such as 4n and 3 (which become insignificant as n grows)
- the **coefficient** 5 in the **leading term** (computers will be five times faster in a few years anyway)

and just say that the algorithm takes time $O(n^3)$ (**"big oh of n^3"**)

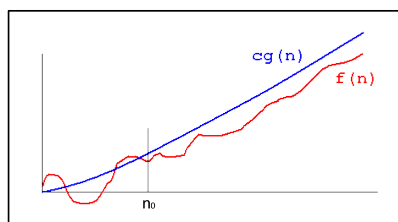


Big-O notation

- ***f(n)*** and ***g(n)*** are the **running times of two algorithms** (for the same computational problem) on **inputs** of **size *n***
- Let $f(n)$ and $g(n)$ be **functions from positive integers to positive reals**.
- We say $f = O(g)$ (which means that "f grows no faster than g") if there is a constant $c > 0$ such that

$$f(n) \leq c \cdot g(n)$$

- $f = O(g)$ is a almost analog of $f \leq g$
  - It differs from the usual notion because of the constant c so that for instance $10n = O(n)$



Big-O notation (cont.)

Example

$f_1(n) = n^2$ $f_2(n) = 2n+20$

- Is f_2 better than f_1 ?
  - $f_2=O(f_1)$ since $f_2(n)/f_1(n) = (2n+20)/n^2 \leq 22$ for $n>0$
    - $f_2(1)/f_1(1) = (2+20)/1^2 \leq 22$,
      $f_2(2)/f_1(2) = (4+20)/2^2 \leq 22$ $(n=2)$,
      …
  - $f_1 \neq O(f_2)$ since $f_1(n)/f_2(n) = n^2/(2n+20)$ is not bounded
- **YES! f_2 is better than f_1**

Big-O notation (cont.)

Example

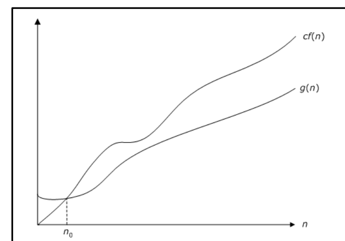
$f_2(n) = 2n+20$ $f_3(n) = n+1$

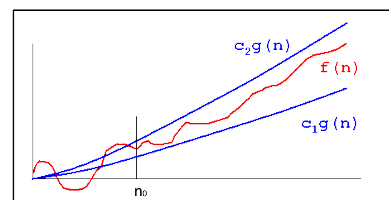
- Is f_3 better than f_2 ?
  - $f_2=O(f_3)$ since $f_2(n)/f_3(n) = (2n+20)/(n+1) \leq 11$ for $n>0$
  - $f_3=O(f_2)$ since $f_3(n)/f_2(n) = (n+1)/(2n+20) \leq 1$ for $n>0$
- f_3 is better than f_2 only by a constant factor
- **We treat functions as equivalent if they differ only by multiplicative constants**

Ω and Θ notations

Just as O(·) is an analog of ≤, we also define analogs of ≥ and = as follows

- $f = \Omega(g)$ means $g = O(f)$





- $f = \Theta(g)$ means $f = O(g)$ and $g = O(f)$

$f_1(n) = n^2$ $f_2(n) = 2n+20$ $f_3(n) = n+1$

Is $f_1=\Omega(f_2)$?

YES!

- $f_1=\Omega(f_2)$ if $f_2=O(f_1)$
- $f_2=O(f_1)$ since $f_2(n)/f_1(n) = (2n+20)/n^2 \leq 22$ for $n>0$

Is $f_3=\Theta(f_2)$?

YES!

$f_3=\Theta(f_2)$ since $f_3=O(f_2)$ and $f_2=O(f_3)$

Simplifications

- **Big-*O*** notation lets us focus on the **big picture**
- When faced with a complicated function like $3n^2 + 4n + 5$, we just replace it with $O(f(n))$, where $f(n)$ is as simple as possible
- In this particular example we'd use $O(n^2)$, because the quadratic portion of the sum dominates the rest
- **In general, it is possible to replace a complicated function $g(n)$ with $O(f(n))$ choosing a $f(n)$ as simple as possible**
- Some **rules** that help **simplify** functions by omitting dominated terms:
  1. **Multiplicative constants can be omitted**. E.g., $14n^2$ becomes n^2
  2. **n^a dominates n^b if $a > b$**. E.g., n^2 dominates n
  3. **Any exponential dominates any polynomial.** E.g., 3^n dominates n^5 (it even dominates 2^n)
  4. **Any polynomial dominates any logarithm**. E.g., n dominates $(\log n)^3$ This also means, for example, that n^2 dominates $n*\log n$

Simplifications - warning

- Programmers and algorithm developers are very interested in constants and would gladly stay up nights in order to make an algorithm run faster by a factor of 2
- But understanding algorithms at the high level would be impossible without the simplicity provided by big-*O* notation

Computational complexity

- Algorithms having complexity $O(n^k)$, with k being a constant, are said to be **polynomial**
  - Algorithms having complexity $O(n)$, are said to be **linear**
- Algorithms having complexity $O(log(n))$, are said to be **logarithmic**
- Algorithms having complexity $O(2^{nk})$ for some constant k are said to be **exponential**
- Algorithm having polynomial complexity are said to be **efficient**
- Problem that admits an efficient algorithm are said to be **tractable**

- **The invention of algorithms opened the way to progress (and to Computer Science!)**
- The field of algorithms studies methods to solve **computational problems**
- Given a computational problems and an algorithm to solve it, there are 3 questions:
  1. Is the algorithm **correct**?
  2. How much **time** does it take, as a function of *n*?
  3. Can we do **better**?
- The challenge is to find **efficient** algorithms, so it is important to have a way express their complexity (big-*O* notation) and a classification into **classes of complexity**