Advanced Programming Techniques
PART I
Introduction

Beniamin Bogosel

Ecole Polytechnique
Department of Applied Mathematics

Contact

Beniamin BOGOSEL: `beniamin.bogosel@polytechnique.edu`

Site Web: `http://www.cmap.polytechnique.fr/~beniamin.bogosel/APT_UAV.html`

- slides
- lab subjects
- codes: Python, Jupyter Notebook

Bibliography:

- `https://people.montefiore.uliege.be/geurts/Cours/PA/2018/pa2018_2019.html`
- Steven S. Skiena, *The Algorithm Design Manual*, Springer (available online, search it!)

Course objectives

Introduction to the systematic study of algorithms and data structures

Two objectives:

- Provide a toolbox containing:
  - data structures allowing to organize and easily access data sets
  - popular algorithms
  - generic methods for the modelization, analysis and solving algorithmic problems
- Use elements of this toolbox to solve new algorithmic problems

Organization

First part: (7 weeks)

⋆ Theoretical/algorithmical aspects

⋆ Implementation in Python

Second part: (7 weeks)

⋆ More applied aspects taught by **Marcela Florea**

An evaluation will be given after each half of the course.

1. Intro: Algorithms + Data structures = Programs

2. Recursivity: recall

Algorithms

- An **Algorithm** is a *finite* and *non-ambiguous* set of instructions or operations allowing to solve a *problem*
- Comes from the name of the mathematician *Al-Khawarizmi* ($\pm$820), the father of the algebra
- An algorithmic problem is formulated by transforming a sequence of values, **inputs**, into a series of values, **outputs**
- Examples of algorithms:
  - a cooking recipe (ingredients $\longrightarrow$ meal/cake)
  - searching in a dictionary (word $\longrightarrow$ definition)
  - integer division (two integers $\longrightarrow$ their quotient)
  - sorting a sequence (sequence $\longrightarrow$ ordered sequence)

Algorithms

⋆ We will study algorithms which are **correct**.

- An algorithm is totally correct if for every given instance, the algorithm terminates producing the expected output
- There are *partially correct algorithms* (termination not guaranteed)
- *approximate algorithms*, producing an inexact output, which is close enough to the desired result

⋆ Algorithms are evaluated in terms of ressource usage:

- computational time
- memory usage

Algorithm descriptions

An algorithm may be specified in multiple ways

- natural language
- graphical illustration
- pseudo code
- a program in a programming language
- ...

The only condition is that the description is precise enough.

Example: sorting algorithms

⋆ sorting problem:

- Input: a sequence of n numbers $\langle a_1, ..., a_n\rangle$
- Output: a permutation of the initial sequence $\langle a'_1, ..., a'_n\rangle$ such that $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Permutation: same values but in a different order.

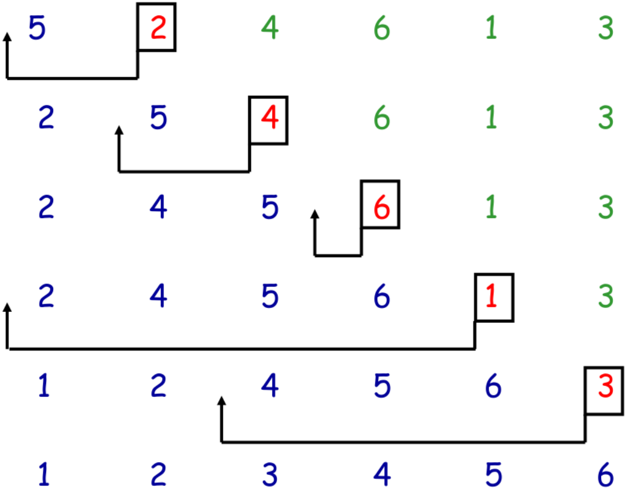
⋆ Example:

- Input: $\langle 31, 41, 59, 26, 41, 58\rangle$
- Output: $\langle 26, 31, 41, 41, 58, 59\rangle$

Insertion Sort

Description in natural language:
Go through the sequence from left to right
For every element a_j:
$\star$ insert it in the corresponding position in a newly ordered sequence containing all previous values of the sequence
Stop when the last element of the sequence was inserted in its place in the new sequence.

Insertion sort: graphical representation



Pseudo-code

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Pseudo-code

Objectives:

- Describe algorithms such that they can be understood by humans
- Render the description independent of the implementation
- Leave out details: error handling, type declaration, etc

Can contain instructions in natural language if necessary

Pseudo-code: Some rules

- Block structures indicated by indentation
- loops (**for**, **while**, **repeat**) and conditions (**if**, **else**, **elseif**)
- Comments indicated by double slash: //
- Variables in a function are local
- $A[i]$ designates the ith element in an array A. $A[i..j]$ represent an interval of values in A, $A.length$ is the size of the array.
- Indexing begins at 1 (note that when coding indices often start at 0)
- when exiting a loop the counter keeps its value

Three questions when facing an algorithm

1. Is my algorithm correct? Does it finish?
2. What is the execution speed
3. Is it possible to do better?

Example: **insertion sort**

1. Yes, analysis, induction
2. $O(n^2)$: complexity analysis
3. Yes: there are algorithms of complexity $O(n \log n)$

Recall: $O(f(n)) \leq C(f(n))$ for some constant C, arbitrary, but fixed.

Correctness: Insertion Sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

- Observation: Before every iteration: the interval $1..j-1$ of A is sorted
- After every iteration the interval $1..j$ of A is sorted

Correctness: Insertion Sort

⋆ Before the first iteration $A[1]$ is trivially sorted

⋆ Before iteration j $A[1..j-1]$ is sorted.

- The inner loop displaces $A[j-1], A[j-2], ...$ a step towards the right until the right position for $A[j]$ is found

⋆ when exiting the main loop $A[1, ..., A.length]$ is ordered!

Complexity of Insertion-sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

- How many comparisons $T(n)$ to sort an array of size n?
- In worst case:
  - The for loop is executed $n - 1$ times $n = A.length$
  - The while loop is executed $j - 1$ times

Complexity of Insertion-sort

- The number of comparisons is bounded by

$$T(n) \leq \sum_{j=2}^{n} (j-1).$$

- Since $\sum_{i=1}^{n} i = n(n+1)/2$ we have

$$T(n) \leq n(n-1)/2$$

- Finally $T(n) = O(n^2)$.

Question: What about the lower bound?

Data structures

- method for storing and organizing data to facilitate access and modification
- A data structure regroups:
  - a certain number of data to maintain
  - a set of operations that may be applied to the data
- In most cases there are
  - multiple ways to represent data and
  - multiple ways to manipulate data
- We distinguish between the interface(abstract representation/description) of the data structures and an implementation

- An abstract data structure (ADS) represents the interface of a data structure
- An ADS specifies precisely:
  - the nature and proprieties of the data
  - the usage and operations that ca be performed
- An ADS admits **different implementations**! (multiple ways of representing the data, multiple ways of performing the operations – more or less efficient)

Example: priority queue

- Data that can be handled: objects with attributes:
  - a key, with a comparison operator, each two keys can be compared (e.g. positive integers)
  - an arbitrary value
- Operations:
  - create an empty queue
  - INSERT(S,x): insert element x in the queue S
  - EXTRACT-MAX(S): remove and output the element of S with the largest priority key
- Possible implementation of this ADS:
  - non-ordered table (insert cheap, extract-max expensive)
  - ordered list (insert costs a bit, extract cheap)
  - etc...

Each implementation leads to different complexities for INSERT and EXTRACT-MAX

Data structures and algorithms in practice

- Solving algorithmic problems almost always requires a good combination of data structures and algorithms (more or less sophisticated) to manage and search in these structures
- The importance of efficient implementation grows with the size of the data
- Real life examples:
  - routing in computer networks
  - search engines
  - aligning DNA sequences in bio-informatics

An example

- A genetics laboratory wants to develop a program capable of finding repetitions of length M in a sequence of nucleotides S of length N with $N \gg M$:

$$ACTG\,\underline{CGAC}\,GGTACGCTT\,\underline{CGAC}\,TTAG...(M = 4)$$

- First idea:
  - An index i goes from 2 to $N - M + 1$
  - Another index j goes from 1 to $j - 1$
  - For $k \in [0, ..., M - 1]$ test if $S[i + k] = S[j + k]$
- Efficiency: number of comparison equal to

$$M \cdot (1 + ... + (N - M)) = \frac{M(N - M + 1)(N - M)}{2}$$

$$\approx 4.5 \cdot 10^{21} \text{ for } N = 3 \cdot 10^9 \text{ and } M = 1000$$

$$\approx 143.000 \text{ years assuming } 10^9 \text{ operations/s}$$

A better solution

1. Build a table of $N-M+1$ lines and M columns for which the k-th line contains the subsequence of length M starting at position k in S

$$\begin{pmatrix} ACTG \\ CTGC \\ TGCG \\ GCGA \\ CGAC \\ \vdots \end{pmatrix}$$

2. Sort the lines of this table in lexicographic order
3. Go through the sorted table and test if there are two identical consecutive lines

Note: when comparing two lines stop at the first difference. Less than $4/3$ comparisons on average.

Effectiveness

- Constructing the table: $M(N-M+1)$ copy operations
- Lexicographic sorting (fast sorting)

$$\leq \frac{8}{3} N \ln N \text{ comparison operations on average}$$

- Detection of consecutive lines

$$\leq \frac{4}{3}(N-M) \text{ comparison operations on average}$$

Assuming identical cost for all operations we get:

$$N(M + \frac{8}{3}\ln N + \frac{4}{3}) - M(M+1/3)$$
$$\approx 3.179 \cdot 10^{12} \text{ operations for } N = 3 \cdot 10^9 \text{ and } M = 1000$$
$$\approx 53 \text{ minutes assuming } 10^9 \text{ operations/s}$$

Remarks

- Using a bigger computer does not improve efficiency problems! Having a computer 1000 more effective: 143 years for the first approach 3.2s for the second
- The second solution is faster, but uses a lot of memory (M times more than the first one)
- (for later) Find an even more efficient solution given the data structures that you will learn in this course

1. Intro: Algorithms + Data structures = Programs

2. Recursivity: recall

Recursive algorithms

An algorithm is recursive if it calls itself directly or indirectly

Motivation: Simplicity of expression for some algorithms

Example: Factorial function

$$n! = \begin{cases} 1 \text{ if } n = 0 \\ n \cdot (n-1)! \text{ if } n > 0 \end{cases}$$

$\textsc{Factorial}(n)$

```
if n == 0
      return 1
return n · FACTORIAL(n − 1)
```

```
FACTORIAL(n)
if n == 0
        return 1
return n · FACTORIAL(n − 1)
```

Rules for defining a recursive solution:

- Define a base case ($n == 0$)
- Each step must decrease the "size" of the problem $n \mapsto n-1$
- If the recursive calls work on the same structure, the sub-problems must not overlap (avoid boundary effects)

Example of multiple recursion

Computing the n-th Fibonacci number

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ \forall n \geq 2 : F_n &= F_{n-2} + F_{n-1} \end{aligned}$$

Algorithm:

```
FIBONACCI(n)
if n ≤ 1
    return n
return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

Example of multiple recursion

$\textsc{Fibonacci}(n)$

1 **if** $n \leq 1$

2 **return** n

3 **return** $\textsc{Fibonacci}(n-2) + \textsc{Fibonacci}(n-1)$

1. Is the algorithm correct?
2. What is the speed of execution?
3. Can we do better?

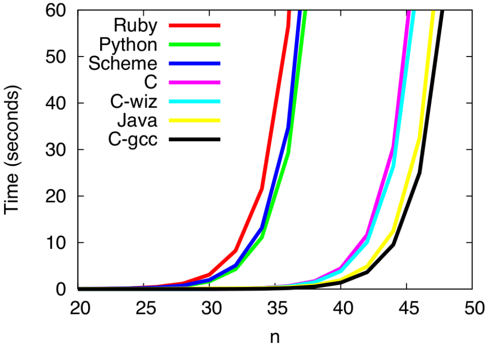
Example of multiple recursion

```
FIBONACCI(n)
1  if n ≤ 1
2      return n
3  return FIBONACCI(n − 2) + FIBONACCI(n − 1)
```

1. Is the algorithm correct?
   - Obviously, the algorithm is correct
   - Proof by induction
2. What is the speed of execution?
3. Can we do better?

Execution speed

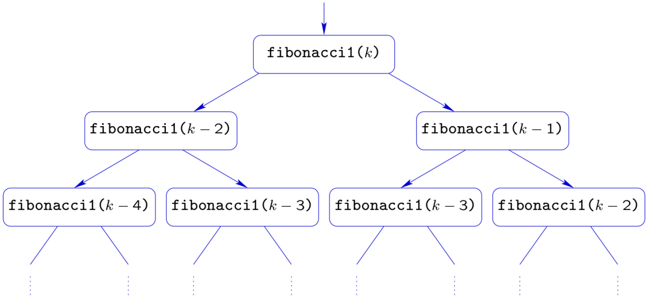
- number of operations for computing FIBONACCI(n) in function of n
- Doing some tests



(Carzaniga)

- Exponential complexity: all implementation reach their limit very fast
- A bigger computer or a faster programming language do not fix a bad algorithm!

Keeping track of the execution



- A naive implementation (like the one presented) computes the same thing multiple times!!
- Keeping track of computed instances could help improve efficiency: recursion with memoization (the recursive algorithm should interact with a data structure; store and quickly retrieve computed values)

Complexity of the naive implementation

```
Fibonacci(n)
1  if n ≤ 1
2      return n
3  return Fibonacci(n − 2) + Fibonacci(n − 1)
```

⋆ $T(n)$ number of basic operations for computing $\textsc{Fibonacci}(n)$

$$T(0) = 2, T(1) = 2, T(n) = T(n-1) + T(n-2) + 2.$$

⋆ therefore $T(n) \geq F_n$

Complexity: how fast does F_n grow?

Elementary observation: Note that $F_n \geq F_{n-1} \geq F_{n-2} \geq \dots$. Therefore for n even we have

$$F_n \geq 2F_{n-2} \geq 2^2 F_{n-4} \geq 2^{n/2-1} F_2$$

and for n odd

$$F_n \geq 2F_{n-2} \geq \dots \geq 2^{\frac{n-1}{2}} F_1.$$

Direct formula: $F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n.$

Conclusion: F_n grows exponentially with n and so does $T(n)$.

Can we do better?

Iterative solution

Yes: simplest approach is better than recursion!

```
FIBONACCI-ITER(n)
if n ≤ 1
        return n
else
        pprev = 0
        prev = 1
        for i = 2 to n
                f = prev + pprev
                pprev = prev
                prev = f
        return f
```

Complexity: time $O(n)$, space $O(1)$

Merge sort

Sort idea based on recursion:

- separate the array into two sub-arrays of the same size
- sort (recursively) each one of the sub-tables
- merge the sorted tables into the big sorted table

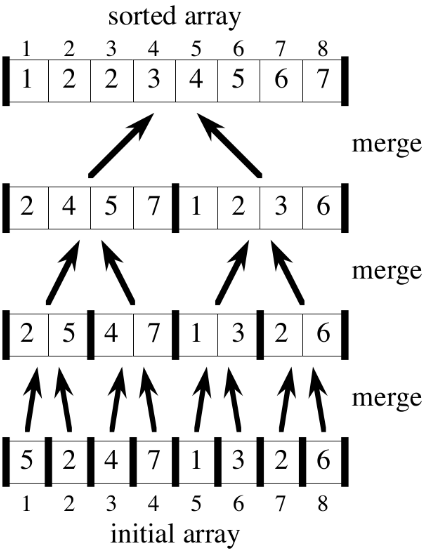
The base case is a table with only one element!

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p+r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERGE(A, p, q, r)
```

Initial call: $\textsc{Merge-sort}(A, 1, A.length)$

General principle: divide and conquer, divide et impera!

Merge-Sort: example



The MERGE function

MERGE(A, p, q, r)

- **input**: the array A and indices p, q, r such that
  - $p \leq q < r$ (no void tables)
  - The sub tables $A[p..q]$ and $A[q+1..r]$ are ordered
- **output**: the two sub-tables are fusioned into a single ordered sub-table $A[p..r]$

Idea

- keep a pointer for the beginning of the tables
- Compare the two smallest elements
- Put it in the fusioned table
- advance the pointer

Fusion: the algorithm

MERGE(A, p, q, r)

$n_1 = q - p + 1$; $n_2 = r - q$
New arrays $L[1..n_1+1] \leftarrow A[p..q]$, $R[1..n_2+1] \leftarrow A[q+1..r]$
$L[n_1+1] = \infty, R[n_2+1] = \infty$
$i = 1$; $j = 1$
for $k = p$ to r **do**
    if $L[i] \leq R[j]$ **then**
        $A[k] = L[i]$
        $i = i + 1$
    else
        $A[k] = R[j]$
        $j = j + 1$

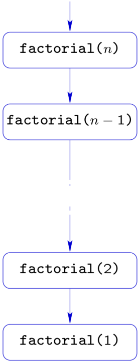
Remarks

- Complexity of merge-sort: $O(n \log n)$ – see next part of the course
- The MERGE function uses $O(n)$ memory space. Exercise (difficult): write a Merge function which does not use additional memory!
- Recursive version of insertion sort:

```
INSERTION-SORT-REC(A, n)
if n > 1 then
    INSERTION-SORT-REC(A, n − 1)
    MERGE(A, 1, n − 1, n)
```

Note on implementing recursivity

- execution trace of the factorial



- each recursion call must memorize the invocation context
- The memory space is $O(n)$ (n recursive calls)

Terminal recursivity

- A procedure is tail recursive if it does not make any other operations after it is being invoked recursively
- Advantages:
  - the memory space is reduced since the invocation context does not need to be memorized
  - Tail recursive procedures can be converted into iterative procedures

Tail recursive version of the factorial

FACTORIAL2(n)

```
return FACTORIAL2-REC(n, 2, 1)
```

FACTORIAL2-REC(n, i, f)

```
if i > n then
    return f
return FACTORIAL2-REC(n, i + 1, f)
```

⋆ Memory space used $O(1)$: the factorial is kept in f which is an input argument for the recursive function

⋆ A little bit less straightforward

We have seen...

- general definition: algorithms, data structures
- analysis of an iterative algorithm (INSERTION-SORT)
- notions regarding recursivity
- analysis of a recursive algorithm (FIBONACCI)
- merge sorting (MERGESORT)