Algorithmic complexity and graphs: problem statement

28 septembre 2022

Need for speed

Speed is important for most applications of computer science, e.g. :

- games
- apps
- scientifc calculus, machine learning
- artistic software (Digital Audio Workstations, graphic software, etc.)

Speed bottleneck

The speed at runtime depends on the complete processing chain :

- 1) computing power / hardware (drives, connectors, wires)
- 2) low-level implementation (programming language, data structure optimization, ressource management)
- 3) algorithms used

Speed bottleneck

- 1) computing power / hardware (drives, connectors, wires)
- 2) low-level implementation (programming language, data structure optimization, ressource management)
- 3) algorithms used

Remarks I :

- the separation between these 3 aspects is somewhat artificial, as the low-level implementation also depends on the hardware and might be considered as part of the algorithm.
- this separation is nonetheless reasonable and it also makes sense to study each aspect separately, in the right context.

Speed bottleneck

- 1) computing power / hardware (drives, connectors, wires)
- 2) low-level implementation (programming language, data structure optimization, ressource management)
- 3) algorithms used

Remarks II :

- 1) between an average machine and and an excellent machine, we most of the time obtain a speedup that is ≤ 10.
- 2) new libraries, languages, protocols often yields important speedups, however it is not possible to give general figures here. But most of the time, for a given task, the speedup is ≤ 100.
- 3) some algorithms are fundamentally unusable because of the time necessary to run them on some problems.

Speed bottleneck

- 1) computing power / hardware (drives, connectors, wires)
- 2) low-level implementation (programming language, data structure optimization, ressource management)
- 3) algorithms used

Remarks III :

- quantum computing **might** behave differently (faster) than classical algorithms in the future, on some tasks, and change the landscape in some fields. However, it is hard to know what the extent of this change will be (in research and industry). Right now, it is still limited by the hardware possibilities, although some companies exist in cryptography.

Speed bottleneck

- 1) computing power / hardware (drives, connectors, wires)
- 2) low-level implementation (programming language, data structure optimization, ressource management)
- 3) algorithms used

Remarks IV :

- our topic will be mostly 3), and we will sometimes focus on 2) slightly more.

What is an algorithm ?

- How could we define it ?

What is an algorithm ?

- **Proposed definition** "A method to solve a problem based on a sequence of elementary operations, arranged in a determined order"

First example

- We have a stack of folders **sorted** by alphabetical order on their name. We look for an algorithm that determins whether a given person **X** has a folder with his or her name in the stack.

First example

- We have a stack of folders **sorted** by alphabetical order on their name. We look for an algorithm that determins whether a given person **X** has a folder with his or her name in the stack.
- Please propose the simplest possible algorithm to complete this task (without worrying about the formalization yet)

First example : solution

- most intuitive solution : check each folder one by one, in their order

First example : solution

- most intuitive solution : check each folder one by one, in their order (**linear search**)
- Could you think of a better (faster) solution ?

First example : solution

- most intuitive solution : check each folder one by one, in their order (**linear seach**)
- Could you think of a better (faster) solution ?
- We could split the folders stack in two parts, then check the first folder of the lower part of the stack (**dichotomic search**)

First example : speed

- Why is the **dichotomic seach** faster than the **linear search** ?

First example : speed

- Why is the **dichotomic seach** faster than the **linear search** ?
- At each dichotomic cut, how many folders can we remove from the stack ?

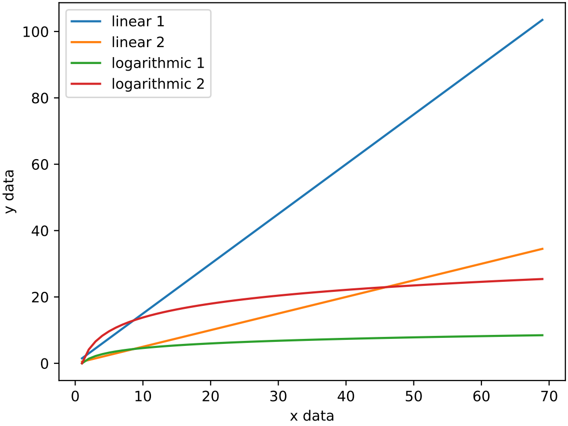
First example : speed

- Why is the **dichotomic seach** faster than the **linear search** ?
- At each dichotomic cut, how many folders can we remove from the stack ?
- Is S is the size of the stack, the new size after the cut is (roughly) $\frac{S}{2}$. Hence, around how many checks are needed, if n is the **initial number of folders** ?

First example : speed

- Why is the **dichotomic seach** faster than the **linear search** ?
- At each dichotomic cut, how many folders can we remove from the stack ?
- Is S is the size of the stack, the new size after the cut is (roughly) $\frac{S}{2}$. Hence, around how many checks are needed, if n is the **initial number of folders** ?
- We need at most $\log_2 n$ checks (backboard)

Logarithms and linear functions



First example : comparison

- How many checks does the **linear search need** at most ? (n is still the initial number of folders)

First example : comparison

- How many checks does the **linear search need** at most ? (n is still the initial number of folders)
- It needs n checks.
- So we have to algorithms that perform the same task, but one of them is faster ($\log n$ versus n)

Simple search example

Exercice 1 : **Comparison between linear and dichotomic search**
If the stack contains 10 folders, how faster is the dichotomic search compared to the linear seach ?

Simple search example

Exercice 1 : **Comparison between linear and dichotomic search**
And if the stack contains 50, 100, 1000, 10000 folders ?

Simple example : comparison

- We say that they have a different **complexity**, which will be the topic of the course

Simple example : comparison

- We say that they have a different **complexity**, which will be the topic of the course
- The complexity is an **approximation** : we are interested in **orders of magnitude**

Time Complexity

- We will study the complexity of algorithms.
- More specifically we will focus on the **time complexity** of algorithms. It is an order of magnitude of the number elementary operations required to solve a problem, given a method (ie : given an algorithm)
- Several computations are possible : for instance worst-case complexity, average complexity.

Time Complexity

- The order of magnitude, in our case, is a way to give an upper bound on the number of elementary operations needed **as a function of the size of the input**.

Time Complexity

- The **order of magnitude** , in our case, is a way to give an upper bound on the number of elementary operations needed **as a function of the size of the input**.
- In this context, we do not mean that 100 and 101 are of the same order of magnitude, but that for instance

Time Complexity

- The **order of magnitude** , in our case, is a way to give an upper bound on the number of elementary operations needed **as a function of the size of the input** .
- In this context, we do not mean that 100 and 101 are of the same order of magnitude, but that for instance
  - n is of the same order of magnitude as $1.5 \times n$

Time Complexity

- The **order of magnitude** , in our case, is a way to give an upper bound on the number of elementary operations needed **as a function of the size of the input** .
- In this context, we do not mean that 100 and 101 are of the same order of magnitude, but that for instance
  - n is of the same order of magnitude as $1.5 \times n$
  - $2 \times n^2$ is of the same order of magnitude as $100 \times n^2$

Time Complexity

- The order of magnitude, in our case, is a way to give an upper bound on the number of elementary operations needed **as a function of the size of the input** .
- In this context, we do not mean that 100 and 101 are of the same order of magnitude, but that for instance
  - n is of the same order of magnitude as $1.5 \times n$
  - $2 \times n^2$ is of the same order of magnitude as $100 \times n^2$
  - 10×2^n is of the same order of magnitude as 500×2^n

Time Complexity

- The order of magnitude, in our case, is a way to give an upper bound on the number of elementary operations needed **as a function of the size of the input** .
- In this context, we do not mean that 100 and 101 are of the same order of magnitude, but that for instance
  - n is of the same order of magnitude as $1.5 \times n$
  - $2 \times n^2$ is of the same order of magnitude as $100 \times n^2$
  - 10×2^n is of the same order of magnitude as 500×2^n
  - but 3^n is **NOT** the same order of magnitude as 2^n

What is the size of the input ?

What we mean by size of the input will depend on the context. Actually, this size will often be represented by several numbers !

- in machine learning, we always express the complexity as a function of <u>at least</u> the number of samples n and the dimension of each sample d.
- in graph theory, the complexity is often expressed as function of the number of nodes n and edges p of the graph.

Space complexity

- There is another time of complexity : the **space complexity**

Space complexity

- There is another time of complexity : the **space complexity**
- It is a measure of the memory used by the algorithm in order to run.

Space complexity

- There is another time of complexity : the **space complexity**
- It is a measure of the memory used by the algorithm in order to run.
- We will also discuss it tomorrow, but this aspect also depends more programming language.

Formalization

- Let us define how we should **specify** an algorithm. It needs :
  - inputs
  - outputs
  - preconditions
  - postconditions

Formalization

- In the case of our folders checking algorithm, this means :
  - inputs
  - outputs
  - preconditions
  - postconditions

Formalization

- In the case of our folders checking algorithm, this means :
  - inputs : stack of folder
  - outputs :
  - preconditions :
  - postconditions :

Formalization

- In the case of our folders checking algorithm, this means :
  - inputs : stack of folder
  - outputs : boolean (**True** if the stack contains the given name X, **False** otherwise)
  - preconditions :
  - postconditions :

Formalization

- In the case of our folders checking algorithm, this means :
  - inputs : stack of folder
  - outputs : boolean (**True** if the stack contains the given name X, **False** otherwise)
  - preconditions : the folders stack is sorted in the alphebetical order
  - postconditions :

Formalization

- In the case of our folders checking algorithm, this means :
  - inputs : stack of folder
  - outputs : boolean (**True** if the stack contains the given name X, **False** otherwise)
  - preconditions : the folders stack is sorted in the alphebetical order
  - postconditions : the output is **True** if and only if the folders stack contains a folder whose name is X.

Final general comments

Once an algorithm is specified, one should also study :

-

Final general comments

Once an algorithm is specified, one should also study :

- its correctness

Final general comments

Once an algorithm is specified, one should also study :

- its correctness
- the termination : the algorithm should end after a **finite number** of computation steps

Theoretical notions

In order to study algorithms from a mathematical point of view and give the intuitive notion a solid grounding, several **axiomatic systems** (plusieurs "axiomatiques") have been built.

Theoretical notions

In order to study algorithms from a mathematical point of view and give the intuitive notion a solid grounding, several **axiomatic systems** (plusieurs "axiomatiques") have been built.

- Turing Machines (Turing, 1936)
- Lambda calculus (Alonzo Church, 1930s)
- Recursive functions (Kurt Gödel, 1930s)