Algorithmic complexity and graphs: recursion, dynamic programming

22 septembre 2024

Classic algorithmic methods

- We will study classical programming paradigms
- Recursivity
- Dynamic programming

Recursion

- **Proposed definition** : a method to solve a problem based on smaller instances of the same problem.

First Recursion example

Exercice 1 : Factorial recursion

- **cd recursion/**
- **factorial_rec.py**
- $n! = 1 \times 2 \times \ldots \times n$

Recursion

A recursive function always has :

- a base case
- a recursive case

Warning

- Decrease does not mean terminate !
- Example : **bad_recursion.py**
- In python, you we get the recursion deptch limit with **sys.getrecursionlimit()** (and also set it **sys.setrecursionlimit()**)

Second example : exponentiation

- We will study the case of **exponentiation** (that we used in RSA)
- Given an integer a, and another integer n, we want to compute a^n.
- If we had to code it ourselves, we could naively do a method similar to **naive_exponentiation.py**

Fast exponentiation

- There is a faster method that can be written recursively : **fast exponentiation** (backboard)

Fast exponentiation

Exercice 2 : Using recursion to perform fast exponentiation

- Modify **fast_exponentiation.py** so that it performs the fast exponentiation algorithm.

Fast vs naive exponentiation

- Compute 5^{300000} with naive exponentiation and fast exponentiation : which one is faster ?

Fast vs naive exponentiation

- Let us compute the number of operations performed in fast exponentiation.

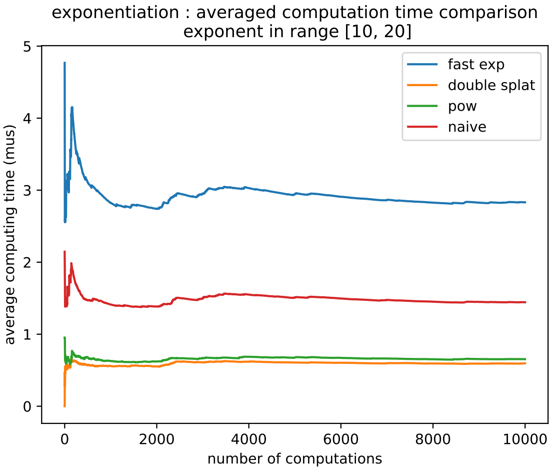
Fast vs naive exponentiation

- Let us compute the number of operations performed in fast exponentiation.
- Say we compute a^n.
- We call the function d times, where $2^d = n$

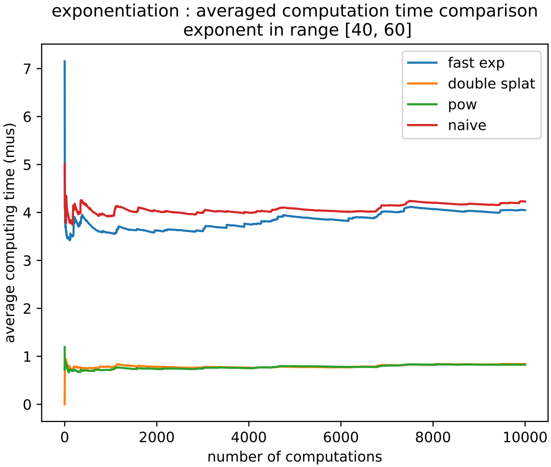
Fast vs naive exponentiation

- Let us compute the number of operations performed in fast exponentiation.
- Say we compute a^n.
- We call the function d times, where $2^d = n$
- This means that $d = \log_2(n)$.
- We say that fast exponentiation has a **logarithmic complexity**, and we denote it $\mathcal{O}(\log n)$

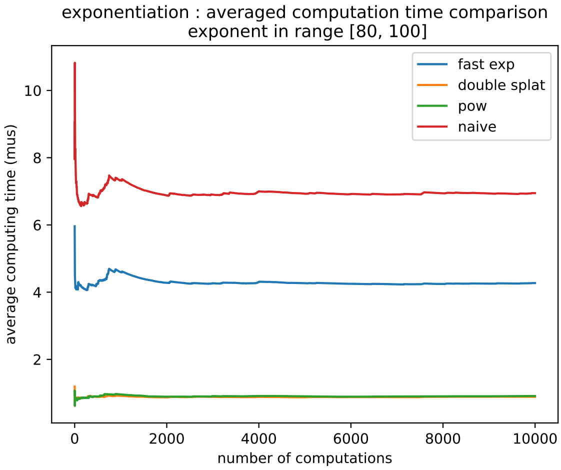
Comparison of methods



Comparison of methods



Comparison of methods



Remark on fast exponentiation

- The fact that fast exponentiation is logarithmic is not related to recursivity.

Remark on fast exponentiation

- The fact that fast exponentiation is logarithmic is not related to recursivity.
- If we write the binary decomposition of n :

$$n = \sum_{k=0}^{d} \alpha_k 2^k \tag{1}$$

- Then :

$$a^n = (a)^{\alpha_0}(a^2)^{\alpha_1}(a^{2^2})^{\alpha_2}\ldots(a^{2^d})^{\alpha_d} \tag{2}$$

Remark on fast exponentiation

- The fact that fast exponentiation is logarithmic is not related to recursivity.
- If we write the binary decomposition of n :

$$n = \sum_{k=0}^{d} \alpha_k 2^k \tag{3}$$

- Then :

$$a^n = (a)^{\alpha_0}(a^2)^{\alpha_1}(a^{2^2})^{\alpha_2}...(a^{2^d})^{\alpha_d} \tag{4}$$

- This allows us to use dynamic programmming and compute only the powers of the form a^{2^i} for $i \leq d$ and then compute the result with at most d multiplications.

Remark on fast exponentiation

Exercice 3 : **Algebraic fast exponentiation :** Use the file **fast_exponentiation_algebraic.py** in order to use this method. You can use the file **./slides/X maths.pdf** in order to have information on how to decompose n in binary (section 8).

- If we write the binary decomposition of n :

$$n = \sum_{k=0}^{d} \alpha_k 2^k \tag{5}$$

- Then :

$$a^n = (a)^{\alpha_0}(a^2)^{\alpha_1}(a^{2^2})^{\alpha_2}...(a^{2^d})^{\alpha_d} \tag{6}$$

- This allows us to use dynamic programmming and compute only the powers of the form a^{2^i} for $i \leq d$ and then compute the result with at most d multiplications.

Shortcomings of recursion

- Recursion can be an elegant way to write algorithms but when not made carefully, the memory usage can explode.
- Let's compute for instance the 100e term of the Fibonnacci sequence.

$$f_{n+2} = f_{n+1} + f_n \tag{7}$$

Non optimized Fibonacci

Exercice 4 : Memory and Fibonacci

- What happens with the function **bad_fibonacci.py** ?
- Let us decompose an example.

Fibonacci and memoization

Exercice 5 : Optimizing Fibonacci

- Modify **memoized_fibonacci.py** so that it uses memoization to compute the sequence without uselessly computing several times the same terms.

Remark

- In python, we can also use a **generator** in order to perform this kind of task.
- See **fibonacci_with_generator.py**

Functools

See also **fibonacci_with_functools.py**, that uses caching through a **decorator**.
https://docs.python.org/3/library/functools.html

Tail recursion

https://en.wikipedia.org/wiki/Tail_call

The Knapsack problem

- We will apply the concept of recursion to a classical problem : **The Knapsack problem**

The general Knapsack problem

- We will apply the concept of recursion to a classical problem : **The Knapsack problem**
- We have a bag of maximal capacity. It can not contain more than a certain weight, say W.
- We have several **objects** i each with a certain **weight** w_i and **value** v_i.

The general Knapsack problem

- We will apply the concept of recursion to a classical problem : **The Knapsack problem**
- We have a bag of maximal capacity. It can not contain more than a certain weight, say W.
- We have several **objects** i each with a certain **weight** w_i and **value** v_i.
- We want to load the maximum possible value in the bag (which means respecting the weight constraint)

The Knapsack problem : restricted variant

- We will focus on a **restricted variant**, without the weight constraint.
- Each object i has a value v_i.
- The question is : "is it possible to fill the bag with a value exactly V ?"

The Knapsack problem : restricted variant

- We will first focus on a **restricted variant**, without the weight constraint.
- Each object i has a value v_i.
- The question is : "is it possible to fill the bag with a value exactly V ?"
- This is called the subset sum problem (problème de la somme de sous-ensembles)

The Knapsack problem : restricted variant

- The question is : "is it possible to fill the bag with a value exactly V ?"
- **example :** values $= [1, 8, 3, 7]$
  - Can we fill the bag with the value 16 ?
  - Can we fill the bag with the value 10 ?
  - Can we fill the bag with the value 5 ?

The Knapsack problem : restricted variant

Exercice 6 : Reformulation of the problem

- Each object i has a value v_i. We have n objects.
- "is it possible to fill the bag with a value exactly V ?"
- We have an list of values

$$L = [v_1, \ldots, v_n] \tag{8}$$

- Please try to reformulate the problem in terms of **sublists** of L.
- Remark : Here we should maybe call L an array. In Python the objets called "lists" are neither arrays nor linked lists but a complex combination of the two. However, here we work with "python lists".

Solving the problem

Exercice 7 : A recursive solution

- Modify **knapsack_recursive.py** so that it searches for a sublist of total value V **in a recursive way**.

Breaking down an instance of the problem

- Using **knapsack_recursive_detailed.py** we can decompose the algorithm.

Optimization and decision

- We say that our solution solves a **decision problem**. The answer provided is "yes" or "no".
- Given a contraint, how could we transpose our solution to an **optimization problem** ? Which means optimizing the total value put inside de bag.

Optimization and decision

- We say that our solution solves a **decision problem**. The answer provided is "yes" or "no".
- Given a contraint, how could we transpose our solution to an **optimization problem** ? Which means optimizing the total value put inside de bag.
- We could search for the maximum V such that there exists a sublist of total value V (in the case of the standard knapsack problem, the constraint of the maximum weight implies that the solution consisting in taking the sum of all positive values does not work, if the weight is small enough).

Exhaustive search

In the general knapsack problem (not the subset sum problem) we could also write a program to find the optimal solution in a **non** recursive way, by exploiting the correspondence with binary numbers : how ?

Back to the knapsack : exhaustive search

In the general knapsack problem (not the subset sum problem) we could also write a program to find the optimal solution in a **non** recursive way, by exploiting the correspondence with binary numbers : how ?
If x_i is a boolean coding the fact that object i is selected, the value of the selected sublist is :

$$\sum_{i=1}^{n} x_i v_i \tag{9}$$

Exhaustive search

$$\sum_{i=1}^{n} x_i v_i \tag{10}$$

How many vectors $(x_1, \ldots x_n)$ are possible ?

Exhaustive search

$$\sum_{i=1}^{n} x_i v_i \tag{11}$$

2^n vectors $(x_1, ...x_n)$ are possible : this number is **exponential** in the problem size n. If n is not very small (e.g. $n >= 20$), we cannot use this solution.

Exploration problems

The Knapsack problem is an **exploration problem** : find an optimal solution in a large set of possible solutions (here, the set of all choices of objects), respecting some constraints (here, the capacity of the knapsack).
There are three approaches to solve exploration problems :

- exhaustive search (often intractable if the problem is not small). Test all possible solutions (e.g. with **backtracking**)
- dynamic programming
- heuristics (specific algorithms designed to attempt to find an approximate solution in a reasonable time)

A heuristic for the general Knapsack problem

- A heuristic is an **approximate solution** that is **possible to compute in a reasonable time** (as opposed to an exhaustive search).
- However a heuristic does most of the time not yield an optimal solution.
- It is often necessary to use heuristics : when it is not possible to find an optimal solution in reasonable time, which, as we will see, happens in many real world situations (such as the Knapsack problem, and more generally NP-hard problems).

Heuristic for the Knapsack problem

Exercice 8 : Finding a heuristic

- Each object i hax value v_i and weight w_i.
- We want to put the maximum value in the bag, keeping the total weight smaller than W.
- Can you propose a **heuristic** that gives an approximate solution to the Knapsack ptoblem ?

Heuristic for the Knapsack problem

Exercice 8 : Finding a heuristic II : heuristics and bad solutions

- Can you find a situation where the solution given by the heuristic is bad ?

Optimality of greedy algorithms

- The heuristic we just discussed belongs to the family of **greedy algorithms** (algorithmes gloutons). Greedy algorihtms exploit information that is available "locally", potentially ignoring better solutions that they would find if they explored the solution space in a different way.
- Most of the time, greedy algorithms lead to **suboptimal solutions**. Often, it is however possible to give a quality bound on the solution they provide (e.g. : "the solution obtained with the greedy algorithm is half as good as the optimal solution").
- In some cases, greedy algorithms yield an optimal solution !

An optimal greedy algorithm : single ressource allocation

We consider the following problem :
A set E of n of clients indexed by $i \in [1, n]$ would like to rent a car (there is only 1 car available). Client i would like to rent the car starting on day $b(i)$ and ending on day $e(i)$.
We look for the largest subset $F \subset E$ of clients that can be satisfied, without renting the car to two clients at the same time.
Formally,

$$\forall i, j \in F, b(i) \leq b(j) \Rightarrow e(i) \leq b(j) \tag{12}$$

Let us look for a greedy optimal solution.

Shortest path problem

- We will now study a famous **graph problem** : the shortest path in a graph.
- This is an exploration problem as well, solved this time by dynamic programming !

The Shortest Path problem

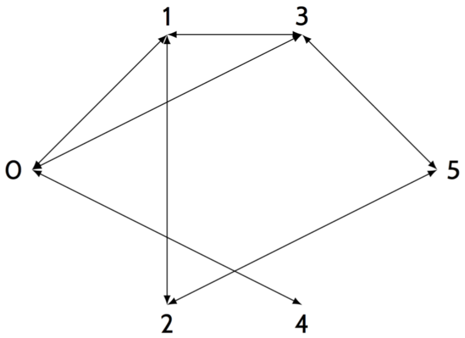


Figure – Toy graph

The Shortest Path problem

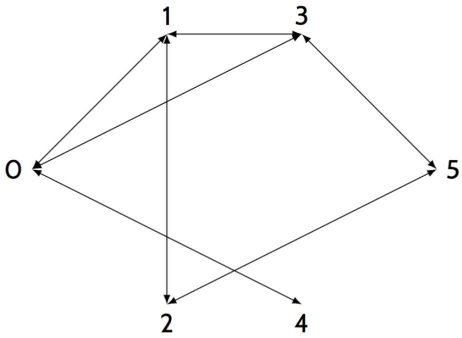


Figure – We will progressively build the list of all shortest paths from 0 to all points

Reminders on graphs

- A graph is defined by set of vertices V and a set of edges E.

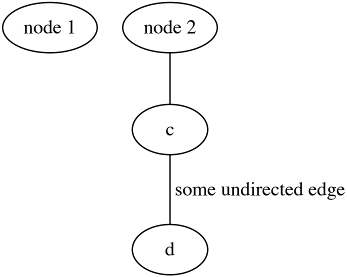


Figure – Simple graph (graphviz demo)

Reminders on graphs

- It can be **undirected**, as this one :

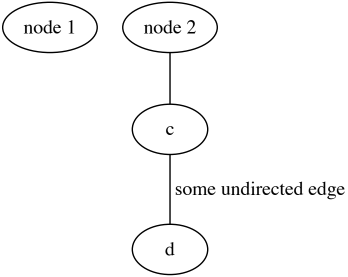


Figure – Simple graph (graphviz demo)

Reminders on graphs

Undirected graph

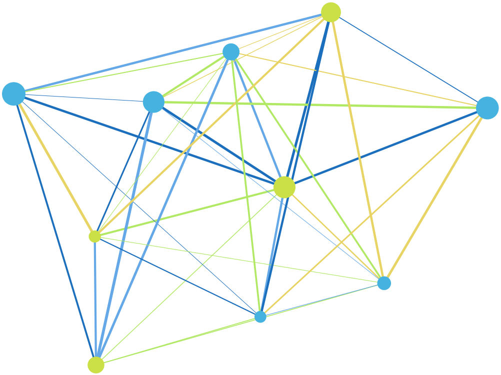


Figure – Undirected random graph generated with python (using networkx)

Reminders on graphs

- Or **directed**, as this one. (it is then called a **digraph**)

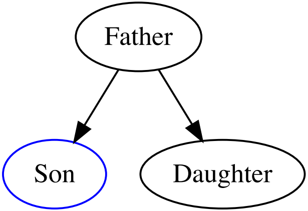


Figure – Digraph

The shortest path problem

- We can code a graph with :

The shortest path problem

- We can code a graph with :
  - a set of edges
  - or a set of neighbors for each node (we will use this solution in the exercices)

The shortest path problem

- the shortest path problem is considered an "easy" problem in terms of algorithmic complexity (as opposed to the knapsack problem).
- It has solutions that are polynomial in the size of the graph and rather intuitive (Dijkstra algorithm)

The shortest path problem

- the shortest path problem is considered an "easy" problem in terms of algorithmic complexity (as opposed to the knapsack problem).
- It has solutions that are polynomial in the size of the graph and rather intuitive such as the famous Dijkstra algorithm and A^*. Those algorithms are slightly more general than the ones we will study as they are also used on **weighted** graphs.
- We will develop more tomorrow on what "polynomial complexity" is, but roughly speaking, it is way faster than exponenial.

Input graphs

cd shortest_path/
In the following exercises, we use graphs that are defined in **input_graphs.py**. You can modify them or use different graphs.

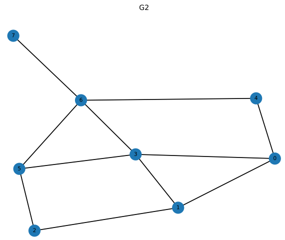
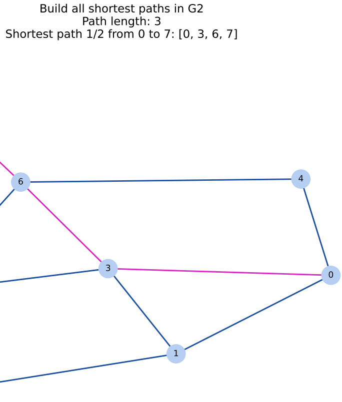


Figure – Input graph G2

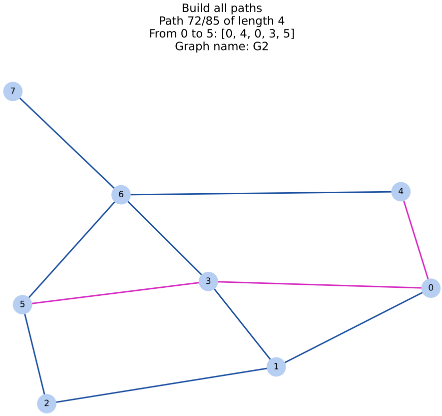
Images

Each exercise stores images in a dedicated subfolder of the **images**/ folder, in order to have a visual feedback.



Exercice 9 : Building all the paths in a graph
Modify **1_all_paths.py** in order to build all the paths in the graph, under a certain length.

All paths


Build all paths
Path 72/85 of length 4
From 0 to 5: [0, 4, 0, 3, 5]
Graph name: G2
7
6
4
3
0
5
1
2

Complexity

If we were using a $n \times n$ chessboard, how many paths would have to be tested to find the path from $(0, 0)$ to (n, n) ? (including loops)

Complexity

If we were using a $n \times n$ chessboard, how many paths would have to be tested to find the path from $(0,0)$ to (n,n) ? (including loops) A number of order 4^{2n} : this in an **exponential complexity**, it takes way too long to compute, and we need another approach.

Path existence

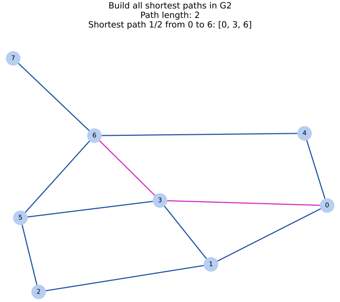
Exercice 10 : Recursion and paths of fixed length
Please modify **4_path_existence.py** in order to recursively check if there exists a path of length l from 0 to a destination.

Exercice 11 : One shortest path

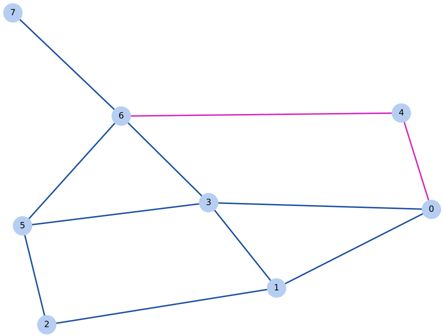
Modify **6_one_shortest_path.py** in order to build one shortest path from 0 to each node, using dynamic programming.

Exercice 11 : All shortest paths

Modify **7_all_shortest_paths.py** in order to build all shortest paths from 0 to each node, using dynamic programming.



Build all shortest paths in G2
Path length: 2
Shortest path 2/2 from 0 to 6: [0, 4, 6]



Shortest paths : complexity

If we were using a $n \times n$ chessboard, how many paths would have to be tested to find the path from $(0, 0)$ to (n, n) ?

In the case of the **one_shortest_path.py** variant, if we were using a $n \times n$ chessboard, how many paths would have to be tested to find the path from $(0, 0)$ to (n, n) ?
A number of order $(2n)^2$ which is a **polynomial complexity :** it is possible to compute it for larger values n than the exponential case.

Conclusion

We experimentally saw that some algorithms (e.g. polynomial ones) run way faster than others (e.g. exponential ones). This is the key phenomenon behind algorithmic complexity.
Tomorrow we will discuss more examples and more theoretical notions about this.