Algorithmic complexity and graphs: recursion, dynamic programming

29 septembre 2022

Classic algorithmic methods

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

Recursion

- How would you define recursion ?

Recursion

- How would you define 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/**
- Please modify **factorial_rec.py** so that it computes the factorial
- $n! = 1 \times 2 \times ... \times n$

Recursion

A recursive function always has :

- a base case
- a recursive case

Warning

- Decrease does not mean terminate !
- What happens with the example **bad_recursion.py** ?
- In python, you can see the recursion limit with **sys.getrecursionlimit()**

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 would naively do a method similar to **naive_exponentiation.py**

Fast exponentiation

- There is a faster method that uses recursion : **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

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

Fast vs naive exponentiation

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

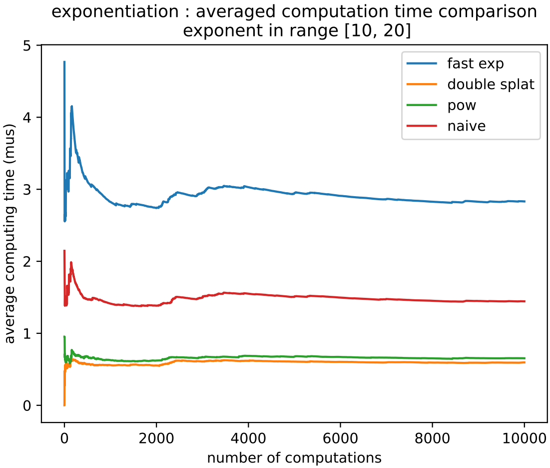
Fast vs naive exponentiation

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

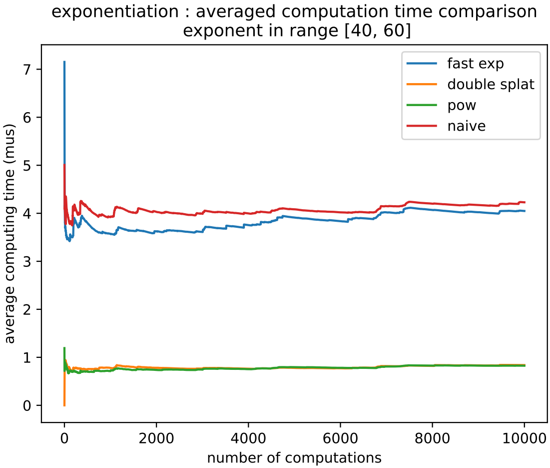
Fast vs naive exponentiation

- 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 $\mathcal{O}(\log n)$

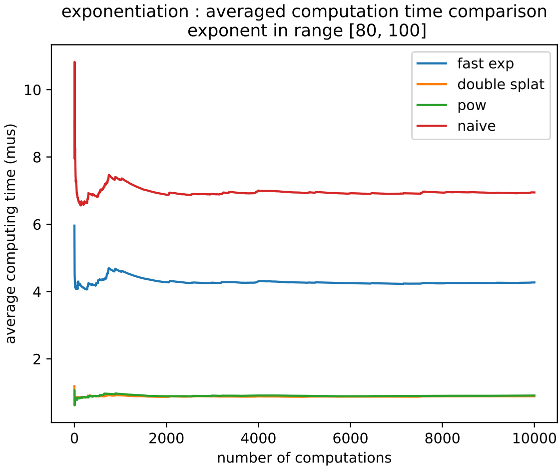
Comparison of methods



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 :

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

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** 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** ?

Non optimized Fibonacci

Exercice 4 : Memory and Fibonacci

- What happens with the function **bad_fibonacci.py** ?
- Let's decompose at 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, you can also use a **generator** in order to perform this king of task.
- See **smarter_fibonacci.py**

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

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

How many vectors $(x_1, \ldots x_n)$ are possible ? 2^n

This is called **exponential complexity**

Exhaustive search

$$\sum_{i=1}^{n} x_i v_i \tag{12}$$

How many vectors $(x_1, ... x_n)$ are possible ? 2^n

This is called **exponential complexity**. If n is large, can we use this solution ?

A heuristic for the general Knapsack problem

- A very important notion regarding algorithms if that of a **heuristic**.

A heuristic for the general Knapsack problem

- A very important notion regarding algorithms if that of a **heuristic.**
- A heuristic is an **approximate solution** that is **possible, or easier to compute**.
- It is sometimes necessary when it is too hard to find the optimal solution, which, as we will see, happens in some real world situations (such as the Knapsack problem).

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 ?

Shortest path problem

- We will now study a famous **graph problem** : the shortest path.

The Shortest Path problem

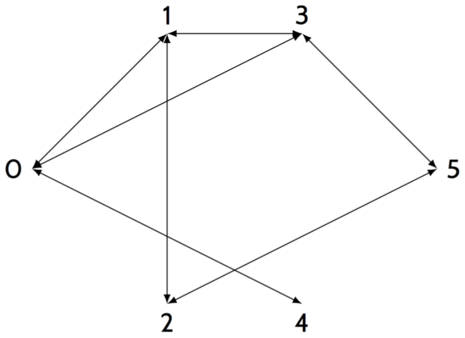


Figure – Toy graph

The Shortest Path problem

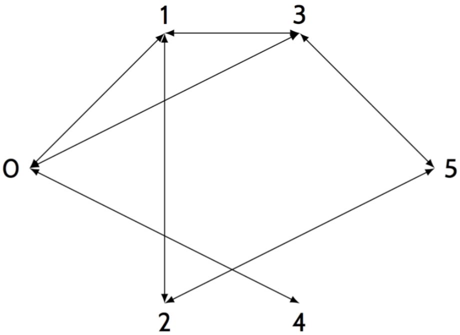


Figure – We will progressively build the list of all shortest paths from 0 to all points

Reminders on graphs

- A graph is defined by ?

Reminders on graphs

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

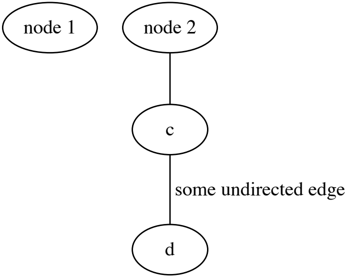


Figure – Simple graph (graphviz demo)

Reminders on graphs

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

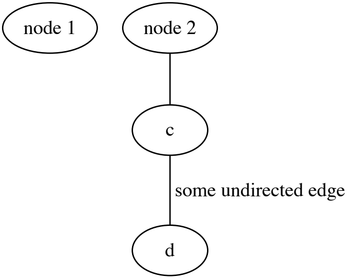


Figure – Simple graph (graphviz demo)

Reminders on graphs

Undirected graph

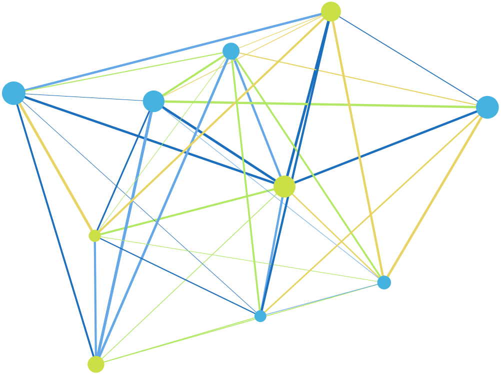


Figure – Undirected random graph generated with python (using networkx)

Reminders on graphs

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

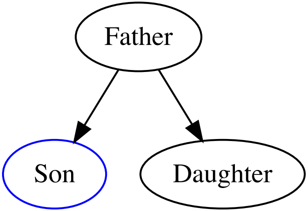


Figure – Digraph

Back to the shortest path problem

- We can code a graph with :

Back to 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)

Back to 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 is considered an "easy" problem in terms of algorithmic complexity.
- Is has solutions that are polynomial in the size of the graph and rather intuitive (Dijkstra algorithm)

Back to 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 is considered an "easy" problem in terms of algorithmic complexity.
- Is has solutions that are polynomial in the size of the graph and rather intuitive such as the famous Dijkstra algorithm, 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.

Paths and graphs

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

Paths and graphs

Exercice 10 : Build all the paths to a destination
Modify **build_paths_to_destination.py** in order to build all the paths that lead to 5, under a certain length.

Paths and graphs

Exercice 10 : Build all the paths to a destination II
Modify **build_paths_to_destination_no_loops** in order to build all the paths that lead to 5, under a certain length, **avoiding 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)

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.

Other approach

- We need another approach.

Path existence

Exercice 11 : Recursion and paths of fixed length

Please modify **path_existence.py** in order to recursively check if there exists a path of length l from 0 to a destination.

Shortest paths

Exercice 12 : Recursion and shortest path
Modify **one_shortest_path.py** in order to recursively build one shortest path from 0 to each node.

Shortest paths

Exercice 12 : Recursion and shortest path
Modify **all_shortest_paths.py** in order to recursively build all shortest paths from 0 to each node.

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 ok to compute it.

Conclusion

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