{EPITECH.}
PARIS GRADUATE SCHOOL OF DIGITAL INNOVATION

Introduction to Algorithms

Part I. Recursion, Dynamic Programming

B9 - Introduction to Algorithms

M-ALG-100

Overview of the module

- **Day 1** Concept of Algorithm, Cryptography, recursion, Knapsack, Shortest Path
- **Day 2** Complexity, Graph problems, Theory

Organisation of the module

- Theoretical course
- Small coding exercises,
- Paper + pen exercises
- Mini-project : explained tomorrow

Organisation

- The exercices will be in python
- Clone the following repository :
  `https://github.com/nlehir/ALGO1.git`
- For day 2 : Please install **matplotlib**, **numpy**
- Optional but useful : **ipdb** (python debug) or your favorite debugger

Objective of the course

- This course is more about the mathematical nature of algorithms
- It is not about optimizing the code itself, but about the mathematical reason why some methods are faster than others at solving problems

Day 1

What is an algorithm ?

Cryptography

- First examples
- Public-key cryptography and symmetric key algorithm
- RSA

Recursion

- Principle and examples
- Shortcomings

Two famous problems

- The Knapsack problem
- The Shortest Path problem

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, aranged in a determined order"

Simple example

- We have a stack of folders ranked 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.

Simple example

- We have a stack of folders ranked 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)

Simple example : solution

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

Simple example : solution

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

Simple 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**)

Simple example : speed

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

Simple 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 ?

Simple 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** ?

Simple 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)

Simple example : comparison

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

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

Simple 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)
- 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**

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)

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** .

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

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 $1.5 \times n$

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 $100 \times n^2$

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

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

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 alphabetical 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 algotithm should end after a **finite number** of computation steps

Cryptography

- We will study some cryptography algorithm as they will provide us examples that show why the complexity of an algorithm is a very important aspect of it.
- Thus, this section is not intended to be a cryptograpy course, but rather a course to focus on some mathematical aspects of the involved algorithms.

First example

- We want to be able to **cipher a text** by **permutating** the letters of the alphaet.

First example

- We want to be able to **cipher a text** by **permutating** the letters of the alphaet.

$$A \mapsto F, \quad B \mapsto P,$$

$$C \mapsto A, \quad D \mapsto \ldots$$

-

Figure: Permutation

Exercise 1

- **cd crypto_intro**
- Please modify the file **crypto_intro/cipher_1.py** so that the function *cipher_1(s)* produces a random key and ciphers the text s, which is a string.
- "cipher" means "chiffrer" in french

Breaking the code

- Please modify the file **crypto_intro/decipher_1.py** in order to attempt to find the key from a **coded message** and an **extract**

Breaking the code

- Please modify the file **crypto_intro/decipher_1.py** in order to attempt to find the key from a **coded message** and an **extract**
- Is it working ?

Breaking the code

- Please modify the file **crypto_intro/decipher_1.py** in order to attempt to find the key from a **coded message** and an **extract**
- Is it working ?
- Why is it taking such a long time ?

Number of permutations

- How many keys are possible ?

Number of permutations

- How many keys are possible ?
- $26! = 403291461126605635584000000$
- It is the number of permutations

Necessary time

- Let us evaluate the time that would be need to try all the keys.

Necessary time

- Let us evaluate the time that would be need to try all the keys.
- for instance from ipython, we can evaluate the time needed in **decipher_1.py** with **timeit**

```
In [3]: timeit import decipher_1
108 ns ± 0.226 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

In [4]:
```

Necessary time

- So I need 108 nanoseconds to try 100 permutations.

Necessary time

- So I need 108 nanoseconds to try 100 permutations.
- I need $\simeq 0.926$ nanosecond to try one permutation

Necessary time

- So I need 108 nanoseconds to try 100 permutations.
- I need $\simeq 0.926$ nanosecond to try one permutation
- Which means $\simeq 3.73 \times 10^{18}$ seconds for 26! permutations.
- This means more than 100 billion years.

First example

However, what would be a shortcoming of this method ?

First example

However, what would be a shortcoming of this method ?
It is vulnearble to **statistical attacks**.

Second example

- Let us do another example

C	H	A	Q	U	E	F	O	I	S	Q	U	U	N	H	O	M	M	E
B	V	A	B	V	A	B	V	A	B	V	A	B	V	A	B	V	A	B
E	D	B	S	G	F						...						N	G

-

Figure: Second coding method

Exercise 2

- Please modify the file **permutations/cipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way

Exercise 2 : Breaking the code

- Please modify the file **permutations/decipher_2.py** in order to attempt to find the key from a **coded message** and an **extract**

Exercise 2

- Please modify the file **permutations/decipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way
- Use a sentence with 100 characters. For which values does the algorithm break the code ?

Exercise 2

- Please modify the file **permutations/decipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way
- Use a sentence with 100 characters. For which values does the algorithm break the code ?
- What is the average running time ?

Exercise 2

- Please modify the file **permutations/decipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way
- Use a sentence with 100 characters. For which values does the algorithm break the code ?
- What is the average running time ? $26^{\text{key size}}$

Exercise 2

- Please modify the file **permutations/decipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way
- Use a sentence with 100 characters. For which values does the algorithm break the code ?
- What is the average running time ? $26^{\text{key size}}$
- So probably you won't be able to break the code for $k \geq 7$ (for instance)

The problem of complexity

- Complexity is key for security problems. However, in most other fields, it is an issue you have to understand and master for your algorithm to be efficient - or to work at all.

Private and public keys

- Before diving into complexity we will study a more complex cryptosystem
- It will allow us to study a more complex algorithm

Private and public keys

- Before diving into complexity we will study a more complex cryptosystem
- **RSA** is based on a Public-key system

Private and public keys

- Before diving into complexity we will study a more complex cryptosystem
- **RSA** is based on a Public-key system
- As opposed to symmetric key algorithms

Symmetric key algorithm

- In the first examples we saw, the same key is used to cipher and to decipher the message

Symmetric key algorithm

- In the first examples we saw, the same key is used to cipher and to decipher the message
- This is called a **symmetric key algorithm**

Symmetric key algorithm

- In the first examples we saw, the same key is used to cipher and to decipher the message
- This is called a symmetric key algorithm
- However would there be an advantage of using **two** keys ?

Public keys and private keys

- **Public key** : used to cipher a text
- **Private key** : used to decipher a text

Public keys and private keys

- **Public key** : used to cipher a text
- **Private key** : used to decipher a text
- There is no need to transmit the private key on the network.
- Whereas in a symmetric context, one needs a secure canal to transmit the key.

Asymmetric cryptosystem

How many keys do we need to generate for each case to enable n persons to communicate ?

Asymmetric cryptosystem

How many keys do we need to generate for each case ?

- Symmetric : each subset of 2 persons must have 1 key.
- Asymmetric : each person must have 1 public key and 1 private key.

Asymmetric cryptosystem

How many keys do we need to generate for each case ?

- Symmetric : each subset of 2 persons must have 1 key : $\binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n(n-1)}{2}$.
- Asymmetric : each person must have 1 public key and 1 private key : $2n$.

Examples :

- Symmetric : AES
- Asymmetric : RSA, ssh, sftp

RSA

- RSA is based on **modular exponentiation**.
- Let M be a message to cipher. We assume M is an integer. Let C be the code (also an integer)

RSA

- RSA is based on **modular exponentiation**.
- Let M be a message to cipher. We assume M is an integer. Let C be the code (also an integer)
- We work **modulo an integer** n (hence the name **modular** exponentiation)
- e.g. $17 \equiv 1 \mod 4$
- $25 \equiv 0 \mod 5$

RSA

- RSA is based on **modular exponentiation**.
- M : message to cipher. C : code.
- Public key : (n, a), Private key : b (a and b must be carefully chosen)
- $C \equiv M^a \mod n$

RSA

- RSA is based on **modular exponentiation**.
- M : message to cipher. C : code.
- Public key : (n, a), Private key : b (a and b must be carefully chosen)
- $C \equiv M^a \mod n$
- Let D be de **deciphered** message
- $D \equiv C^b \mod n$

RSA

- M : message to cipher. C : code.
- Public key : (n, a), Private key : b (a and b must be carefully chosen)
- $C \equiv M^a \mod n$
- Let D be de **deciphered** message
- $D \equiv C^b \mod n$
- In order for the algorithm to work, we must have $D \equiv M \mod n$.

RSA

- M : message to cipher. C : code.
- Public key : (n, a), Private key : b (a and b must be carefully chosen)
- $C \equiv M^a \mod n$
- Let D be de **deciphered** message
- $D \equiv C^b \mod n$
- In order for the algorithm to work, we must have $D \equiv M \mod n$.
- Which means : $M^{ab} \equiv M \mod n$

RSA

- M : message to cipher. C : code.
- Public key : (n, a), Private key : b
- $M^{ab} \equiv M \mod n$
- The construction of n, a, and b comes from **number theory** (Fermat theorem, Gauss theorem)

RSA

- M : message to cipher. C : code.
- Public key : (n, a), Private key : b
- $M^{ab} \equiv M \mod n$
- The construction of n, a, and b comes from **number theory** (Fermat theorem, Gauss theorem)

RSA : construction of the keys

- Choose p and q prime numbers
- $n = pq$
- $\phi = (p-1)(q-1)$
- Choose a coprime with ϕ
- Choose b inverse of a modulo ϕ.

Exercise 3

- **cd** to the **rsa** directory
- Please modify **rsa_functions.py** so that when calling **generate_rsa_keys** from **cipher_rsa**, a public key and a private key are created.
- You can change the prime numbers used.

Exercise 4

- Please modify **rsa_functions.py** so that when calling **cipher_rsa_** from **cipher_rsa**, a public key and a private key are created **and the text stored in texts is coded and stored in crypted_messages** .

Exercise 5

- Please modify **rsa_functions.py** so that when calling **decipher_rsa_keys** from **decipher_rsa**, the generated public key private key are used to **decipher the crypted text**.

Exercise 6 : breaking RSA

- Modify **rsa_functions.py** so that when calling **find_private_key** from **decipher_unknown_rsa** the secret private key is found from the public key and used to decipher the crypted message.

Conclusion on RSA

It is extremely hard to break RSA if n is sufficiently large, because you need to find the decomposition of n in **prime numbers.** This is another important example of a algorithmic that is too **complex** to be solved.

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

- **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** ?
- 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 **normal_exponentiation.py**

Fast exponentiation

- There is a faster method that uses recursion : **fast exponentiation** (backboard)

Exercise 7

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

Fast vs normal exponentiation

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

Fast vs normal exponentiation

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

Fast vs normal exponentiation

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

Fast vs normal 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 normal 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 $\mathcal{O}(\log n)$

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 :

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

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.

$$f_{n+2} = f_{n+1} + f_n \tag{3}$$

Exercise 8

- What happens with the function **bad_fibonacci.py** ?
- Write another method that uses a **generator** in **smarter_fibonacci.py**

Exercise 9 : last example with fibonacci

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

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.
- We have several **objects** each with a certain **weight** and **value**.

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.
- We have several **objects** each with a certain **weight** and **value**.
- We want to load the maximum possible value in the bag (which means respecting the wieght constrain)

The Knapsack problem : restricted variant

- We will focus on a **restricted variant**. The **value equals the size**.
- 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 focus on a **restricted variant**. The **value equals the size**.
- Each object i has a value v_i.
- The question is : "is it possible to fill the bag with a value exactly V ?"
- Mathematically speaking : "is there a sublist of total value V in the list of values ?"

Exercise 10

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

Optimization and decision

- 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

- 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, with a maximum number of objects (for instance)

Back to the knapsack : exhaustive search

We could also write the program in a non recursive way, by exploiting the correspondence with binary numbers : how ?

Back to the knapsack : exhaustive search

We could also write the program 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 :

$$\sum_{i=1}^{n} x_i v_i \tag{4}$$

Exhaustive search

$$\sum_{i=1}^{n} x_i v_i \tag{5}$$

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

Exhaustive search

$$\sum_{i=1}^{n} x_i v_i \tag{6}$$

How many vectors $(x_1, ...x_n)$ are possible ? 2^n
This is called **exponential complexity**

Recursivity vs dynamic programming

Run and compare **recursive_function.py** and **dynamic_programming.py**.
Why is one of them way faster than the other one ?

The Shortest Path problem

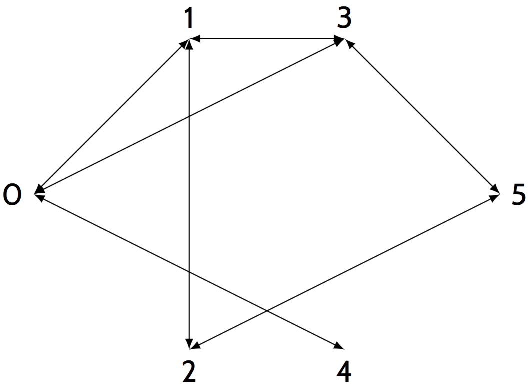


Figure: Toy graph

The Shortest Path problem

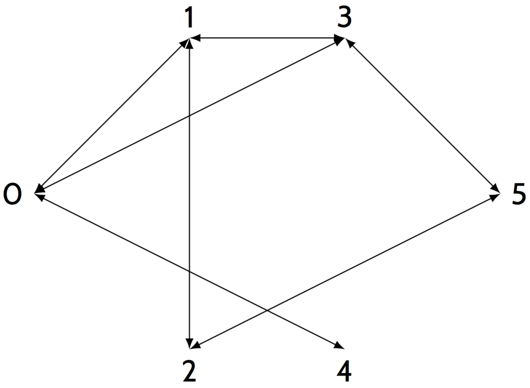


Figure: We will progressively build the list of all shortest paths from 0 to all points

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.

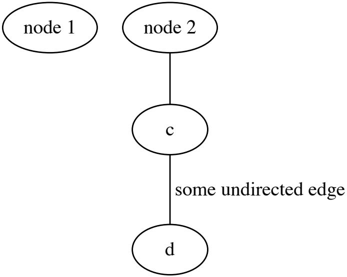


Figure: Simple graph (graphviz demo)

Reminders on graphs

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

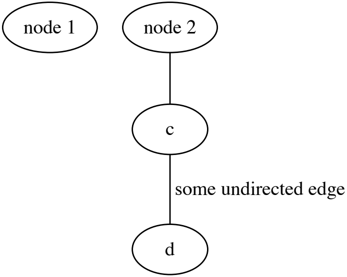


Figure: Simple graph (graphviz demo)

Reminders on graphs

Undirected graph

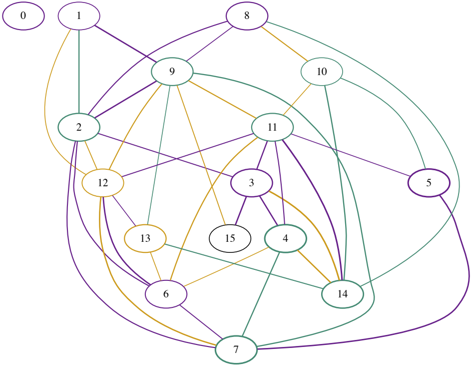


Figure: Undirected random graph generated with python

Reminders on graphs

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

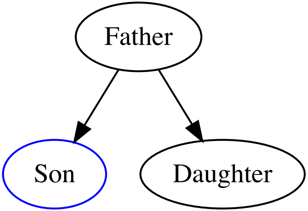
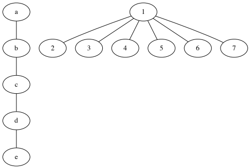


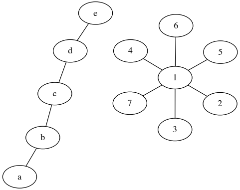
Figure: Digraph (graphviz demo)

Useful tool : graphviz

- A tool to visualize graphs
- Several **generator programs** : dot, neato



(a) Image generated with **dot**



(b) Image generated with **neato**

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)

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)

Exercise 11

Modify **build_all_paths** in order to build all the paths in the graph, under a certain length.

Exercise 12

Modify **build_paths_to_destination** in order to build all the paths to a destination, under a certain length.

Exercise 12

Modify **build_paths_to_destination_no_loops** in order to build all the paths to a destination, under a certain length, **avoiding loops.**

Complexity

Modify **build_paths_to_destination_no_loops** in order to build all the paths to a destination, under a certain length, **avoiding loops.** If we were using a 100×100 chessboard, how many paths would have to be tested to find the path from $(0, 0)$ to $(100, 100)$?

Complexity

Modify **build_paths_to_destination_no_loops** in order to build all the paths to a destination, under a certain length, **avoiding loops.**
If we were using a 100×100 chessboard, how many paths would have to be tested to find the path from $(0, 0)$ to $(100, 100)$?
4^{200} : this in an **exponential complexity**, it takes way too long to compute.

Exercise 13

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

Exercise 14

Modify **one_shortest_path** in order to recursively build one shortest path from 0 to a destination.

Exercise 15

Modify **all_shortest_paths** in order to recursively build all shortest paths from 0 to a destination.

Exercise 15

Modify **all_shortest_paths** in order to recursively build all shortest paths from 0 to a destination.
If we were using a 100×100 chessboard, how many paths would have to be tested to find the path from $(0, 0)$ to $(100, 100)$?

Exercise 15

Modify **all_shortest_paths** in order to recursively build all shortest paths from 0 to a destination.
If we were using a 100×100 chessboard, how many paths would have to be tested to find the path from $(0, 0)$ to $(100, 100)$?
A number of order 200^3 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.