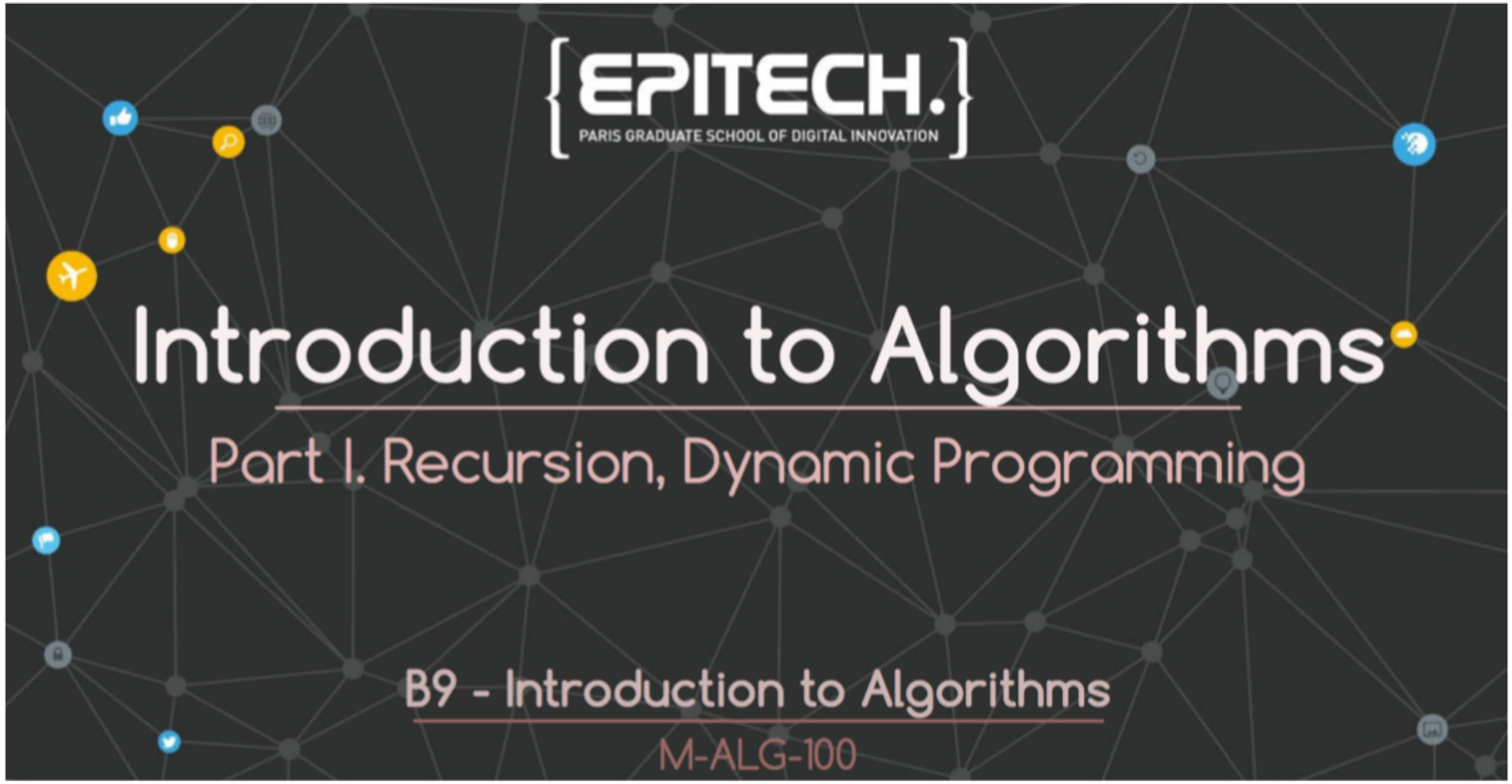
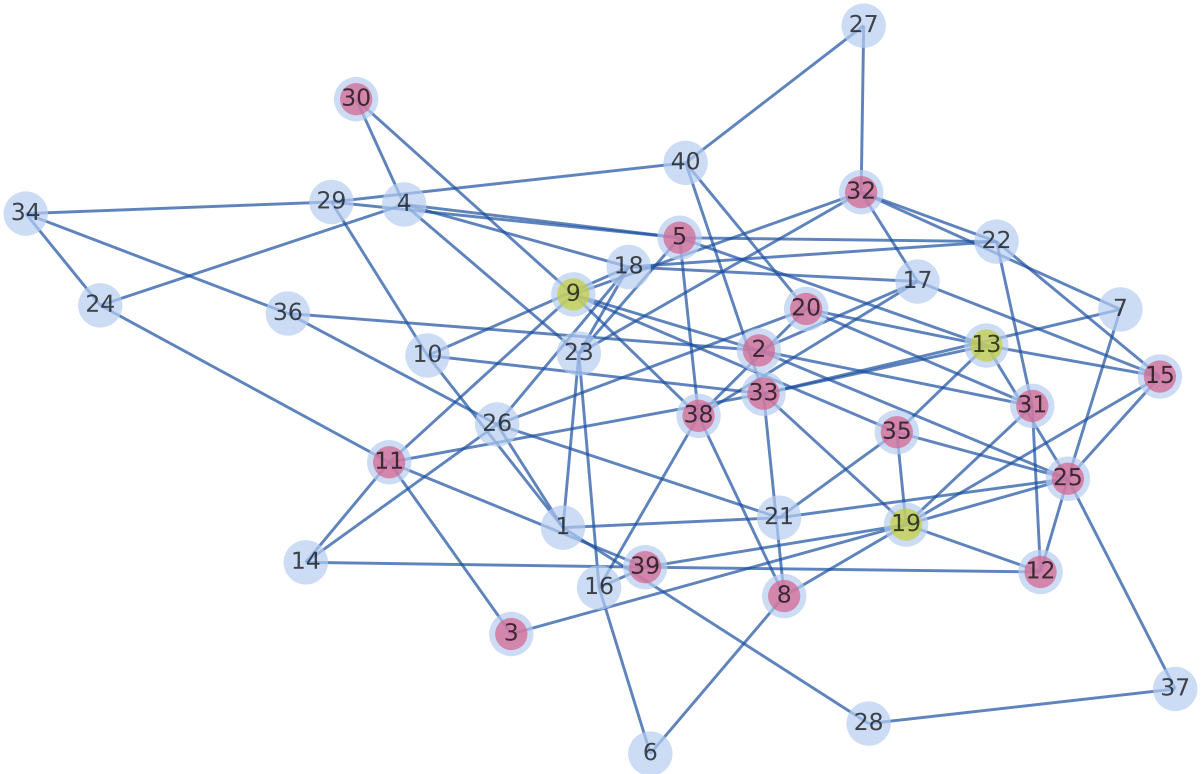
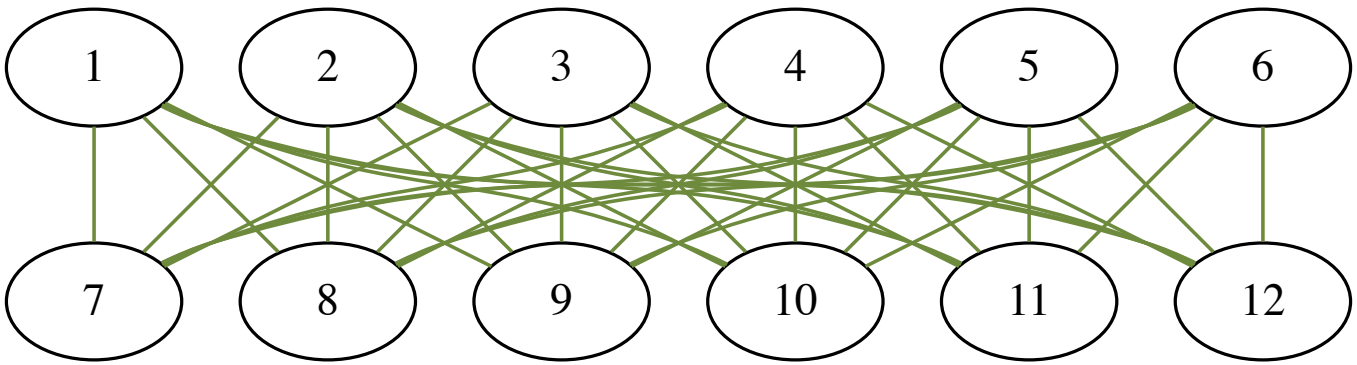
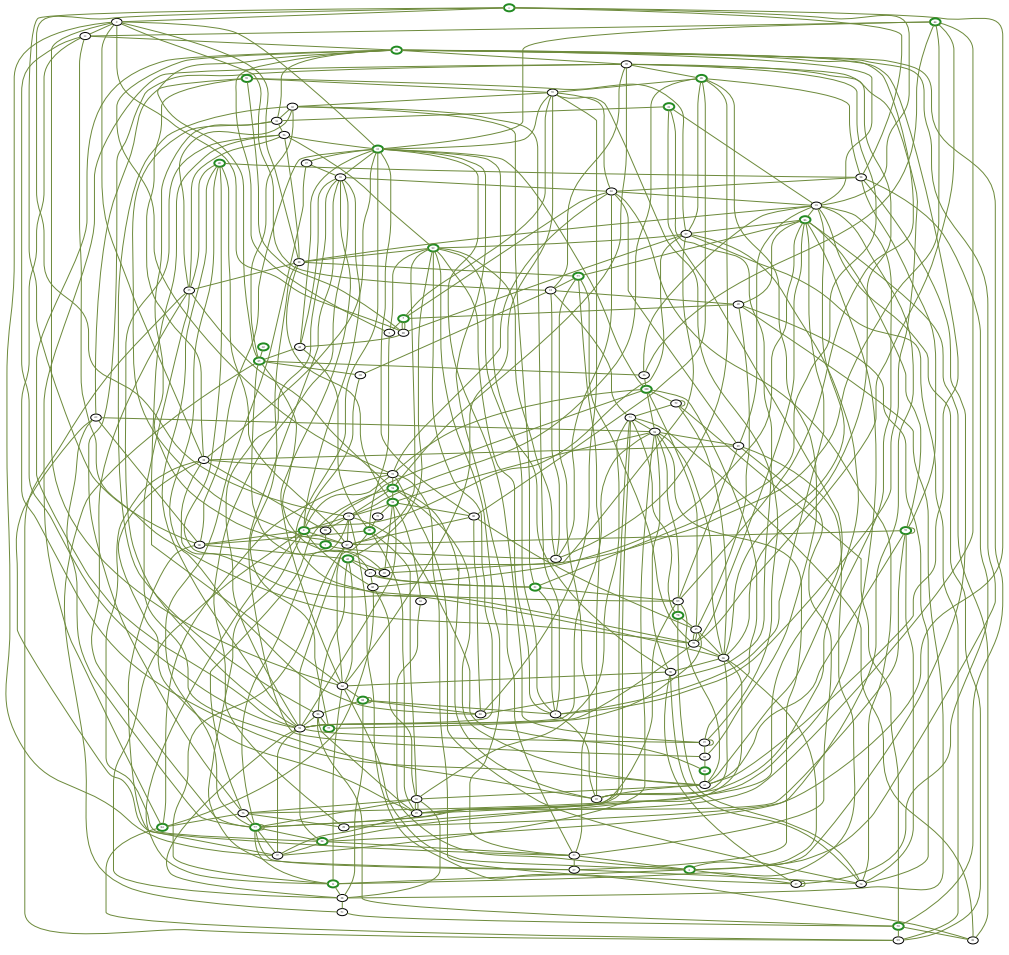

EPITECH.
PARIS GRADUATE SCHOOL OF DIGITAL INNOVATION
Introduction to Algorithms
Part I. Recursion, Dynamic Programming
B9 - Introduction to Algorithms
M-ALG-100

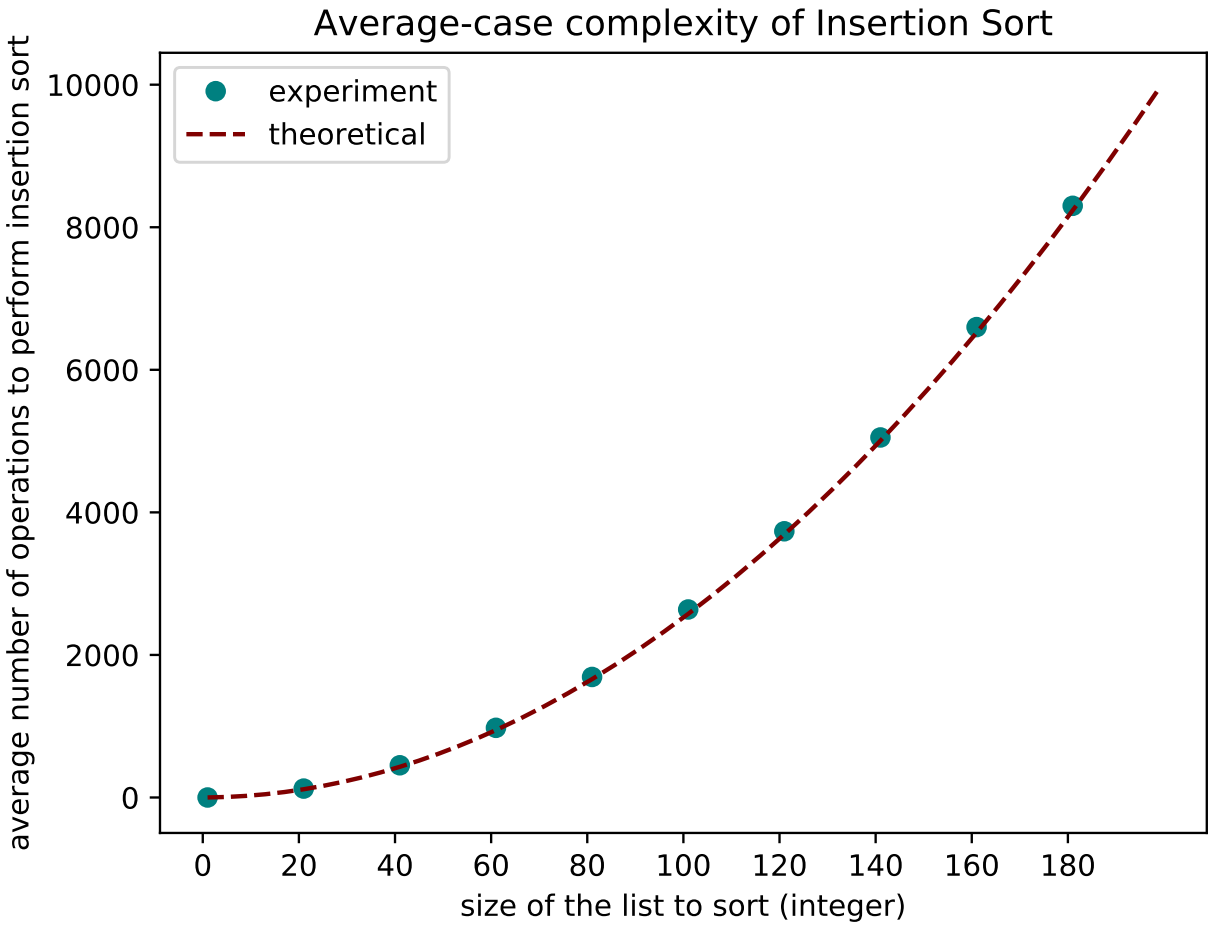
Subset size: 3
Algo step: 3




1
2
3
4
5
6
7
8
9
10
11
12



```
➜  rsa git:(master) ✗ pp cipher_rsa.py
public key : (187, 153)
private key : 137
keys are ok : b is the inverse of a modu
cipher rsa
a (97) becomes 124
l (108) becomes 113
g (103) becomes 86
o (111) becomes 111
r (114) becomes 141
i (105) becomes 150
t (116) becomes 139
h (104) becomes 70
m (109) becomes 10
  (32) becomes 32
c (99) becomes 88
o (111) becomes 111
u (117) becomes 134
r (114) becomes 141
s (115) becomes 81
e (101) becomes 118
code : 124,113,86,111,141,150,139,70,10,
```


Average-case complexity of Insertion Sort
experiment
theoretical
average number of operations to perform insertion sort
size of the list to sort (integer)
0
2000
4000
6000
8000
10000
0
20
40
60
80
100
120
140
160
180

```
for i in range(n):
    for j in range(i):
        l = [i+j+k for k in range(n)]
```

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
- Project : explained tomorrow

Organization

- The exercices will be in python
- Please clone the following repository : `https://github.com/nlehir/ALGO1.git`
- Third party libs : **matplotlib**, **numpy**, **networkx**
- Optional but useful : **ipdb** (python debug) or another debugger

Organization

- In order to make installation easier and to make the course more interactive, you are encouraged to use **docker** and **jupyter-lab**. Please see the **readme.txt** in the github repo.

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 ?

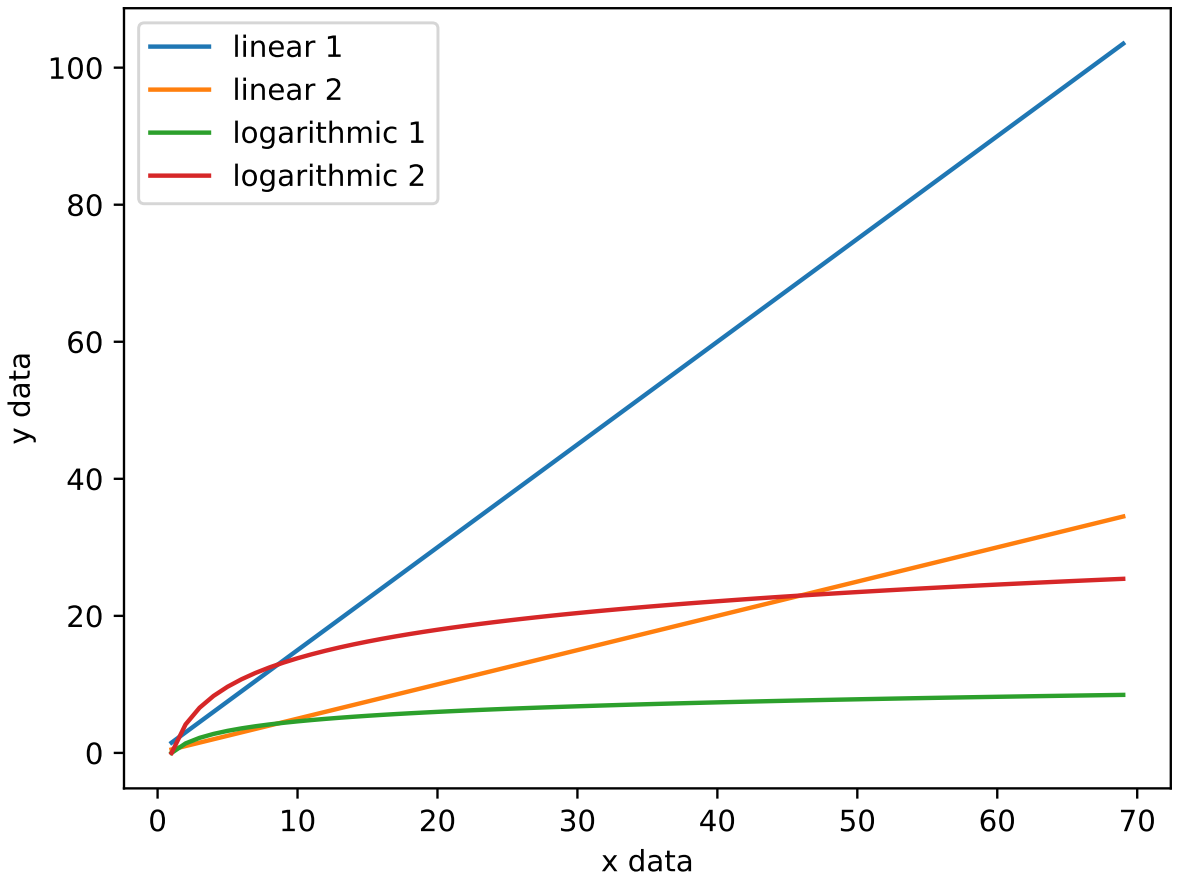
Simple example : speed

- 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)

Logarithms and linear functions



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 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 the stack contains 50 folders ?

Simple search example

Exercice 1 : **Comparison between linear and dichotomic search**

What if the stack contains 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)

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

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.

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 Godel 1930s)

Cryptography

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

First example (Chiffrage par substitution)

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

First example (Chiffrage par substitution)

- 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 – Example permutation

Ciphering

Exercice 2 : First ciphering example

- **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

Exercice 2 : First ciphering example part II

- 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

Exercice 2 : First ciphering example part II

- 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

Exercice 2 : First ciphering example part II

- 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 ?
- Let us count them.

Number of permutations

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

Exercice 3 : How many keys would actually stop the program ?

Necessary time

Exercice 4 : Please evaluate the time that would be necessary on your machine to evaluate all possible keys.

Necessary time

- I need 3.6 milliseconds to try 100 keys.

Necessary time

- I need 3.6 milliseconds to try 100 keys.
- So I need 0.036 millisecond to try 1 key.

Necessary time

- I need 3.6 milliseconds to try 100 keys.
- So I need 0.036 millisecond to try 1 key.
- Which means $\simeq 1.45 \times 10^{25}$ seconds for 26! permutations.

Necessary time

- I need 3.6 milliseconds to try 100 keys.
- So I need 0.036 millisecond to try 1 key.
- Which means $\simeq 1.45 \times 10^{25}$ seconds for 26! permutations.
- Or $\simeq 4.6 \times 10^{17}$ 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

Second example

Exercice 5 : Please modify the file **crypto_intro/cipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way

Second example : Breaking the code

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

Second example

- Please modify the file **crypto_intro/decipher_2.py** so that the function *cipher_2(s)* ciphers the text in the same way
- Use a sentence with 50 characters. For key sizes does the algorithm break the code ?

Second example

- Please modify the file **crypto_intro/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 number ok keys that are to be tried, as a function of the size of the key ?

Second example

- Please modify the file **crypto_intro/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 number ok keys that are to be tried, as a function of the size of the key ? $26^{\text{key size}}$

Second example

- Please modify the file **crypto_intro/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 number ok keys that are to be tried, as a function of the size of the key ? $26^{\text{key size}}$
- So probably you won't be able to break the code for $k \geq 7$ or so.

The problem of complexity

- Complexity is key for security problems. However, it is important in most other fields as well. 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 : construction of the keys

- Choose p and q prime numbers
- $n = pq$
- $\phi = (p-1)(q-1)$
- Choose a coprime with ϕ (entiers premiers entre eux)
- Choose b inverse of a modulo ϕ, which means

$$ab \equiv 1 \mod \phi \tag{1}$$

Setting up a RSA system

Exercice 6 : Building RSA I : choosing keys

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

Setting up a RSA system

Exercice 6 : Building RSA II : ciphering the text

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

Setting up a RSA system

Exercice 6 : Building RSA III : deciphering the text

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

Breaking RSA

Exercice 6 : Trying to break RSA

- Modify **rsa_functions.py** so that when calling **find_private_key()** from **decipher_unknown_rsa.py 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

Exercice 7 : Facorial 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 **normal_exponentiation.py**

Fast exponentiation

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

Fast exponentiation

Exercice 8 : Using recursion to perform fast exponentiation

- 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 :

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

- Then :

$$a^n = (a)^{\alpha_0}(a^2)^{\alpha_1}(a^{2^2})^{\alpha_2}...(a^{2^d})^{\alpha_d} \tag{3}$$

Remark on fast exponentiation

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

$$n = \sum_{k=0}^{d} \alpha_k 2^k \tag{4}$$

- Then :

$$a^n = (a)^{\alpha_0}(a^2)^{\alpha_1}(a^{2^2})^{\alpha_2}...(a^{2^d})^{\alpha_d} \tag{5}$$

- 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 9 : **Algebraic fast exponentiation :** Use the file **fast_exponentiation_algebraic.py** in order to use this method. You can use the file **./maths** in order to have information on how to decompose n in binary.

- If we write the binary decomposition of n :

$$n = \sum_{k=0}^{d} \alpha_k 2^k \tag{6}$$

- Then :

$$a^n = (a)^{\alpha_0}(a^2)^{\alpha_1}(a^{2^2})^{\alpha_2}...(a^{2^d})^{\alpha_d} \tag{7}$$

- 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.

$$f_{n+2} = f_{n+1} + f_n \tag{8}$$

Non optimized Fibonacci

Exercice 10 : Memory and Fibonacci

- What happens with the function **bad_fibonacci.py** ?

Non optimized Fibonacci

Exercice 10 : Memory and Fibonacci

- What happens with the function **bad_fibonacci.py** ?
- Let's look at an example.

Fibonacci and memoization

Exercice 11 : 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 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 first focus on a **restricted variant**. The **value equals the weight** $w_i = v_i$.
- 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.

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 12 : 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 a list of values

$$L = [v_1, \ldots, v_n] \tag{9}$$

- Please try to reformulate the problem in terms of **sublists** of L.

Solving the problem

Exercice 13 : 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

- Let us analyze of this solution works on an example.

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.

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 :

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

Exhaustive search

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

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

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**

Exhaustive search

$$\sum_{i=1}^{n} x_i v_i \tag{13}$$

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 14 : 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 14 : Finding a heuristic II : heuristics and bad solutions

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

Shortest path problem

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

The Shortest Path problem

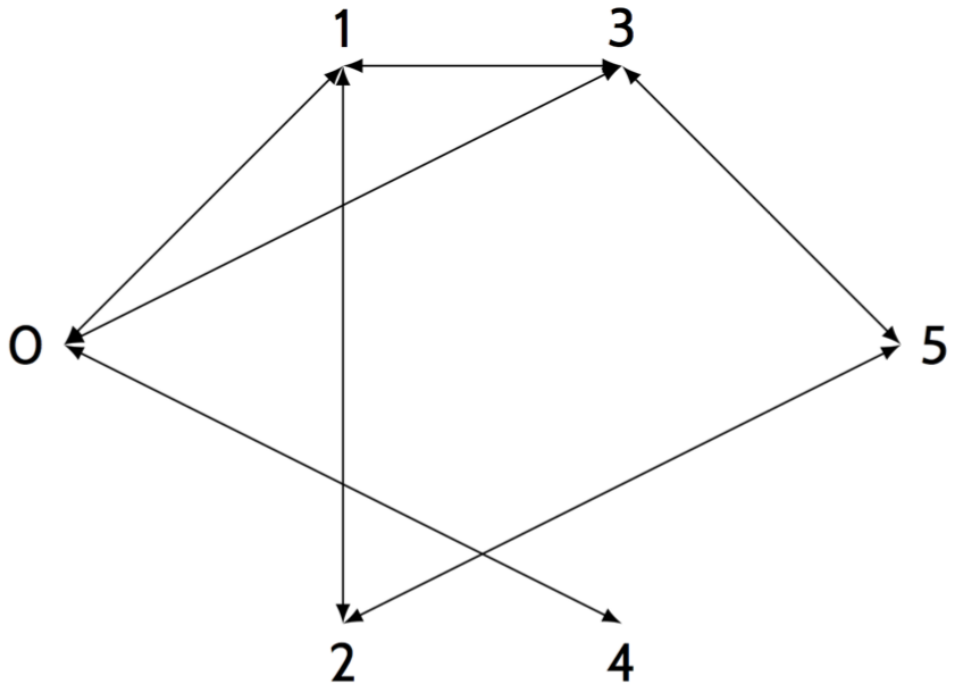


FIGURE – Toy graph

The Shortest Path problem

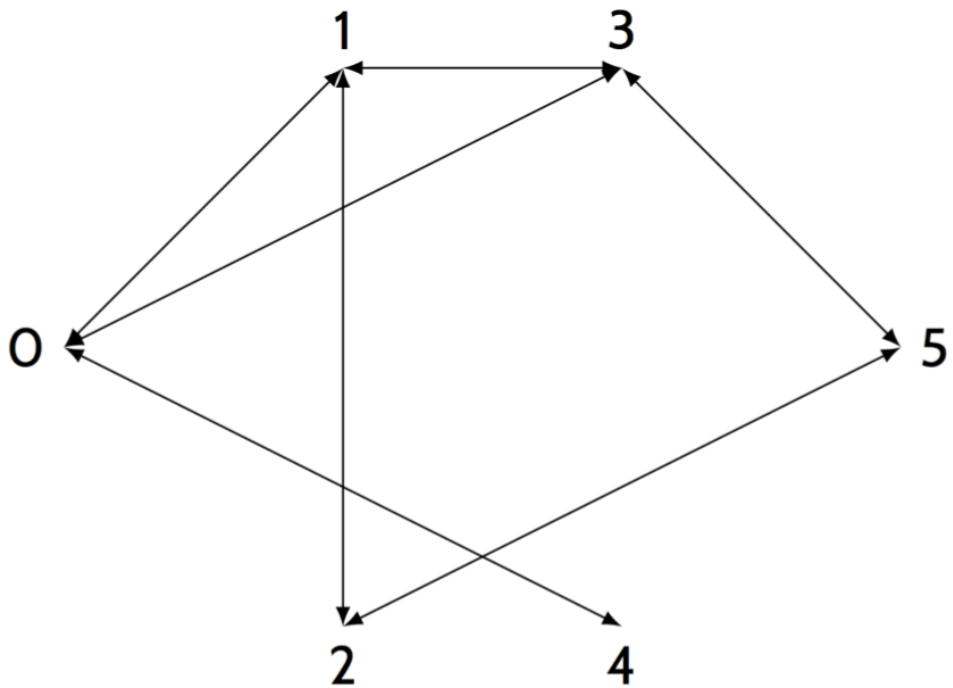


FIGURE – We will progressively build the list of all shortest paths from 0 to all points

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.

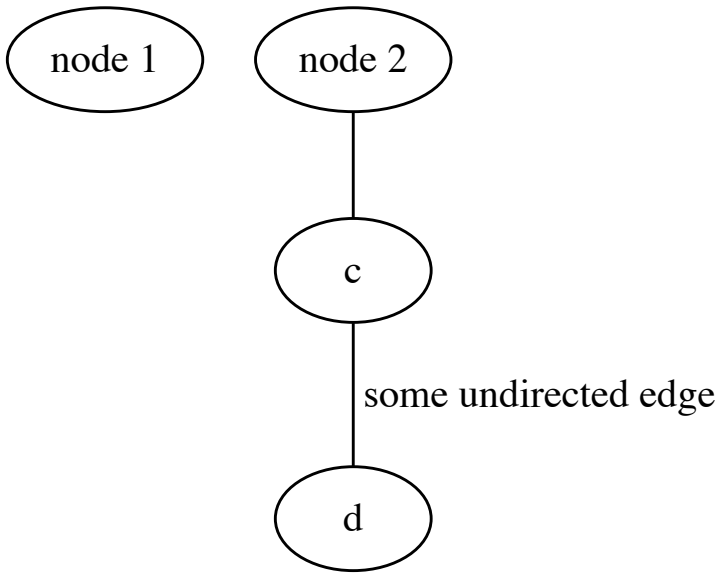


FIGURE – Simple graph (graphviz demo)

Reminders on graphs

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

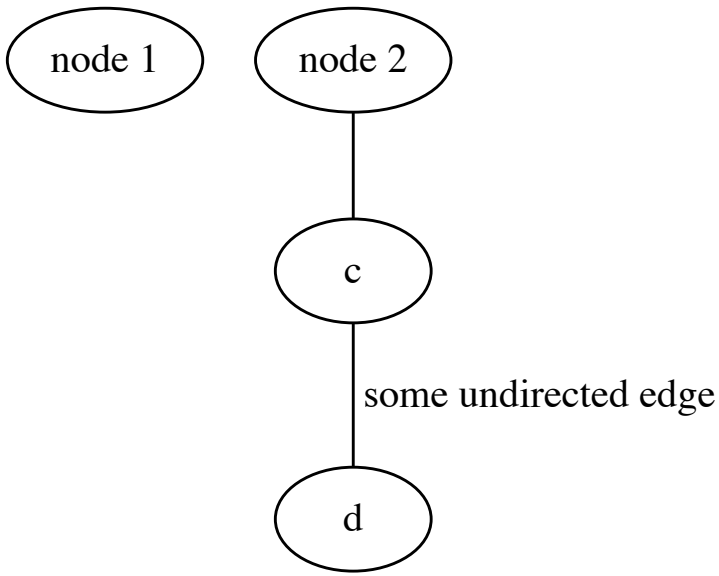


FIGURE – Simple graph (graphviz demo)

Reminders on graphs

Undirected graph

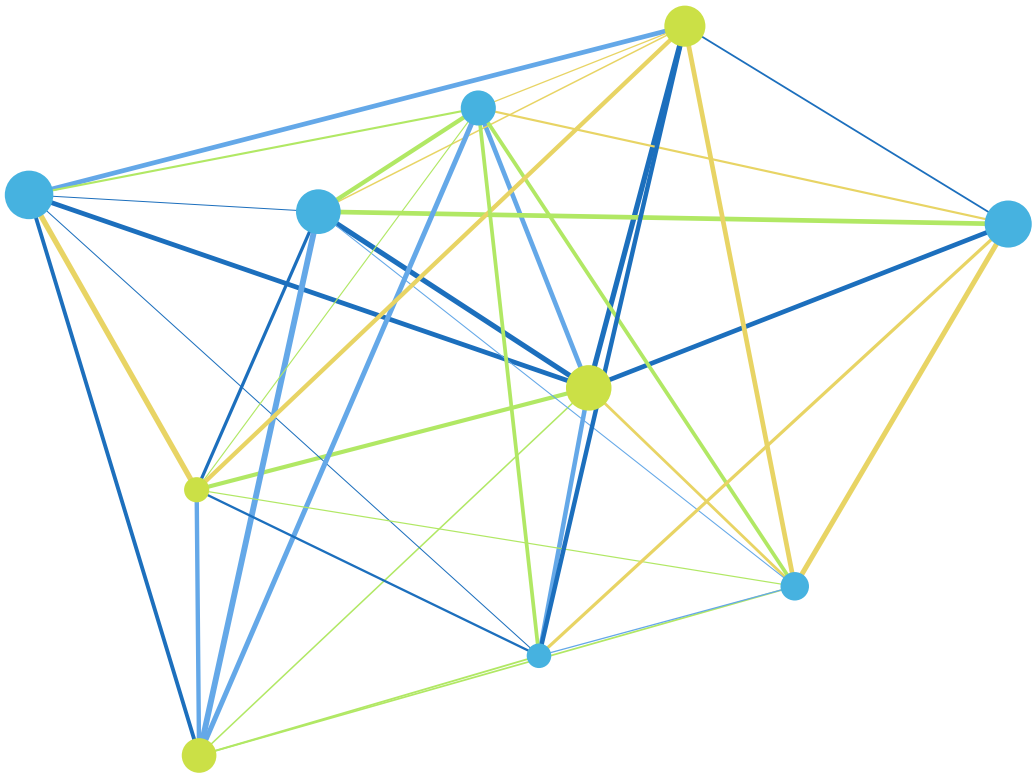


FIGURE – Undirected random graph generated with python

Reminders on graphs

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

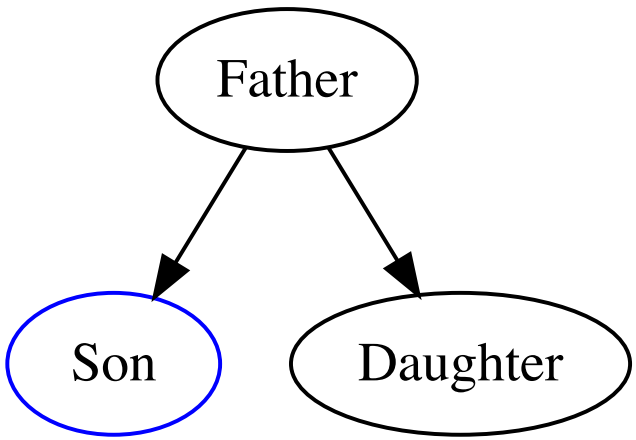


FIGURE – Digraph

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 (Dijkstra algorithm)
- We will develop more tomorrow on what "polynomial complexity" is, but roughly speaking, it is way faster than exponenial.

Paths and graphs

Exercice 15 : 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 16 : Build all the paths to a destination

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

Paths and graphs

Exercice 16 : Build all the paths to a destination II

Modify **build_paths_to_destination_no_loops** in order to build all the paths to a destination, 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) ?

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) ?
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 17 : 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 18 : Recursion and shortest path

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

Shortest paths

Exercice 18 : Recursion and shortest path

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

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) ?

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.