

# C16 Un peu de Python

## 1. Types de base et opérateurs associés

### Types de base

Type	Opérations	Commentaires
<code>int</code>		
<code>float</code>		
<code>bool</code>		
<code>str</code>		

# C16 Un peu de Python

## 1. Types de base et opérateurs associés

### Types de base

Type	Opérations	Commentaires
<code>int</code>	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code> , <code>//</code> , <code>%</code> .	Entiers de taille dynamique (limitée par la mémoire).
<code>float</code>		
<code>bool</code>		
<code>str</code>		

# C16 Un peu de Python

## 1. Types de base et opérateurs associés

### Types de base

Type	Opérations	Commentaires
<code>int</code>	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code> , <code>//</code> , <code>%</code> .	Entiers de taille dynamique (limitée par la mémoire).
<code>float</code>	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code> .	Représentation des nombres en virgule flottante (norme <code>ieee754</code> : mantisse sur 53 bits, exposant sur 11 bits). Fonctions élémentaires dans <code>math</code>
<code>bool</code>		
<code>str</code>		

# C16 Un peu de Python

## 1. Types de base et opérateurs associés


### Types de base

Type	Opérations	Commentaires
<code>int</code>	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code> , <code>//</code> , <code>%</code> .	Entiers de taille dynamique (limitée par la mémoire).
<code>float</code>	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code> .	Représentation des nombres en virgule flottante (norme <code>ieee754</code> : mantisse sur 53 bits, exposant sur 11 bits). Fonctions élémentaires dans <code>math</code>
<code>bool</code>	<code>or</code> , <code>and</code> , <code>not</code> , <code>all</code> , <code>any</code> .	Les deux valeurs possibles sont <code>True</code> et <code>False</code> . Évaluation séquentielle des expressions.
<code>str</code>		

# C16 Un peu de Python

## 1. Types de base et opérateurs associés


### Types de base

Type	Opérations	Commentaires
<code>int</code>	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code> , <code>//</code> , <code>%</code> .	Entiers de taille dynamique (limitée par la mémoire).
<code>float</code>	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code> .	Représentation des nombres en virgule flottante (norme <code>ieee754</code> : mantisse sur 53 bits, exposant sur 11 bits). Fonctions élémentaires dans <code>math</code>
<code>bool</code>	<code>or</code> , <code>and</code> , <code>not</code> , <code>all</code> , <code>any</code> .	Les deux valeurs possibles sont <code>True</code> et <code>False</code> . Évaluation séquentielle des expressions.
<code>str</code>	<code>+</code> , <code>*</code> , <code>len</code> , <code>[]</code>	Chaines de caractères.  La numérotation commence à 0. Extraction de tranches avec <code>[debut:fin:pas]</code>

# C16 Un peu de Python

## 1. Types de base et opérateurs associés

### Types de base

Type	Opérations	Commentaires
<code>int</code>	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code> , <code>//</code> , <code>%</code> .	Entiers de taille dynamique (limitée par la mémoire).
<code>float</code>	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code> .	Représentation des nombres en virgule flottante (norme <code>ieee754</code> : mantisse sur 53 bits, exposant sur 11 bits). Fonctions élémentaires dans <code>math</code>
<code>bool</code>	<code>or</code> , <code>and</code> , <code>not</code> , <code>all</code> , <code>any</code> .	Les deux valeurs possibles sont <code>True</code> et <code>False</code> . Évaluation séquentielle des expressions.
<code>str</code>	<code>+</code> , <code>*</code> , <code>len</code> , <code>[]</code>	Chaines de caractères.  La numérotation commence à 0. Extraction de tranches avec <code>[debut:fin:pas]</code>

Des conversions sont possibles entre ces différents types, par exemple `int("2024")` est l'entier 2024.

# C16 Un peu de Python

## 1. Types de base et opérateurs associés

### Exemples

- 1 Quel est le reste dans la division euclidienne de  $1970^{54}$  par 1515 ?

# C16 Un peu de Python

## 1. Types de base et opérateurs associés

### Exemples

- 1 Quel est le reste dans la division euclidienne de  $1970^{54}$  par 1515 ?
- 2 Quel est le nombre de chiffres de  $2024^{42}$  ?



# C16 Un peu de Python

## 1. Types de base et opérateurs associés

### Exemples

- 1 Quel est le reste dans la division euclidienne de  $1970^{54}$  par 1515 ?
- 2 Quel est le nombre de chiffres de  $2024^{42}$  ?
- 3 Comment obtenir le dernier caractère d'une chaîne de caractère ?

# C16 Un peu de Python

## 1. Types de base et opérateurs associés

### Exemples

- 1 Quel est le reste dans la division euclidienne de  $1970^{54}$  par 1515 ?
- 2 Quel est le nombre de chiffres de  $2024^{42}$  ?
- 3 Comment obtenir le dernier caractère d'une chaîne de caractères ?
- 4 On a dissimulé un message dans la chaîne de caractères suivante, pour le retrouver il faut lire un caractère sur 2. Quel est le message ?

"TArooapV cbbiqernz QlCeC JPEyItUhroknT"

### Définir une fonction en Python

Pour définir une fonction en Python :

### Définir une fonction en Python

Pour définir une fonction en Python :

- qui ne renvoie pas de valeur :

```
1 def <nom_fonction>(<arguments>):  
2   <instruction>
```

- qui renvoie une valeur :

```
1 def <nom_fonction>(<arguments>):  
2   <instruction>  
3   return <resultat>
```

# C16 Un peu de Python

## 3. Importation de librairies

### Utilisation de librairies

- On peut importer la totalité de la librairie `<lib>` à l'aide de `import <lib>`. Dans ce cas les fonctions de cette librairie doivent être utilisées en les faisant précéder du nom de la librairie

### Exemple

# C16 Un peu de Python

## 3. Importation de bibliothèques

### Utilisation de bibliothèques

- On peut importer la totalité de la bibliothèque `<lib>` à l'aide de `import <lib>`. Dans ce cas les fonctions de cette bibliothèque doivent être utilisées en les faisant précéder du nom de la bibliothèque
- Cet import peut se faire en donnant un *alias* : `import <lib> as <alias>`

### Exemple

# C16 Un peu de Python

## 3. Importation de bibliothèques

### Utilisation de bibliothèques

- On peut importer la totalité de la bibliothèque `<lib>` à l'aide de `import <lib>`. Dans ce cas les fonctions de cette bibliothèque doivent être utilisées en les faisant précéder du nom de la bibliothèque
- Cet import peut se faire en donnant un *alias* : `import <lib> as <alias>`
- Pour importer simple la fonction `<fonc>` de la bibliothèque `<lib>`, on utilise `from <lib> import <fonc>`. Le nom de la fonction est alors utilisé directement.

### Exemple

# C16 Un peu de Python

## 3. Importation de bibliothèques

### Utilisation de bibliothèques

- On peut importer la totalité de la bibliothèque `<lib>` à l'aide de `import <lib>`. Dans ce cas les fonctions de cette bibliothèque doivent être utilisées en les faisant précéder du nom de la bibliothèque
- Cet import peut se faire en donnant un *alias* : `import <lib> as <alias>`
- Pour importer simple la fonction `<fonc>` de la bibliothèque `<lib>`, on utilise `from <lib> import <fonc>`. Le nom de la fonction est alors utilisé directement.

### Exemple

```
1 import randint
2 de = randint(1,6)
```



# C16 Un peu de Python

## 3. Importation de bibliothèques

### Utilisation de bibliothèques

- On peut importer la totalité de la bibliothèque `<lib>` à l'aide de `import <lib>`. Dans ce cas les fonctions de cette bibliothèque doivent être utilisées en les faisant précéder du nom de la bibliothèque
- Cet import peut se faire en donnant un *alias* : `import <lib> as <alias>`
- Pour importer simple la fonction `<fonc>` de la bibliothèque `<lib>`, on utilise `from <lib> import <fonc>`. Le nom de la fonction est alors utilisé directement.

### Exemple

```
1 import randint
2 de = randint(1,6)
```

```
1 from random import randint
2 de = randint(1,6)
```

### Instructions conditionnelles

- Sans clause `else`

```
1  if <condition>:  
2  <instructions>
```

Exécute les <instructions> si la condition est vérifiée.

### Instructions conditionnelles

- Sans clause `else`

```
1  if <condition>:  
2  <instructions>
```

Exécute les <instructions> si la condition est vérifiée.

- Avec clause `else`

```
1  if <condition>:  
2  <instructions1>  
3  else:  
4  <instructions2>
```

Cela permet d'exécuter les <instructions1> si la condition est vérifiée, sinon on exécute les <instructions2>.

### Opérateurs de comparaison

- L'égalité se teste avec `==`

### Opérateurs de comparaison

- L'égalité se teste avec `==`
- La différence avec `!=`

### Opérateurs de comparaison

- L'égalité se teste avec `==`
- La différence avec `!=`
- Plus grand ou égal avec `>=`, plus petit ou égal avec `<=`

### Opérateurs de comparaison

- L'égalité se teste avec `==`
- La différence avec `!=`
- Plus grand ou égal avec `>=`, plus petit ou égal avec `<=`
- Plus grand strictement avec `>`, plus petit strictement avec `<`

### Boucles while



### Boucles while

- La syntaxe d'une boucle **while** en Python est :

```
1 while <condition>:  
2     <instruction>
```

Cela permet d'exécuter les <instructions> tant que la <condition> est vérifiée.

### Boucles while

- La syntaxe d'une boucle **while** en Python est :

```
1 while <condition>:  
2     <instruction>
```

Cela permet d'exécuter les <instructions> tant que la <condition> est vérifiée.

- L'instruction **break** permet de sortir de la boucle de façon anticipée.

### Boucles while

- La syntaxe d'une boucle **while** en Python est :

```
1 while <condition>:  
2     <instruction>
```

Cela permet d'exécuter les <instructions> tant que la <condition> est vérifiée.

- L'instruction **break** permet de sortir de la boucle de façon anticipée.
- On ne sait pas a priori combien de fois cette boucle sera exécutée (et elle peut même être infinie), on dit que c'est une boucle **non bornée**.

### Boucles for avec range

### Boucles for avec range

- Les instructions :

```
1  for <variable> in range(<entier>):  
2  <instructions>
```

créent une variable parcourant les entiers de 0 à <entier> (exclu).

### Boucles for avec range

- Les instructions :

```
1  for <variable> in range(<entier>):  
2  <instructions>
```

créent une variable parcourant les entiers de 0 à <entier> (exclu).

- Les <instructions> indentées qui suivent seront exécutées pour chaque valeur prise par la variable.

### Boucles for avec range

- Les instructions :

```
1  for <variable> in range(<entier>):  
2  <instructions>
```

créent une variable parcourant les entiers de 0 à <entier> (exclu).

- Les <instructions> indentées qui suivent seront exécutées pour chaque valeur prise par la variable.
- L'instruction **break** permet de sortir de la boucle de façon anticipée.

### Boucles for avec range

- Les instructions :

```
1  for <variable> in range(<entier>):  
2  <instructions>
```

créent une variable parcourant les entiers de 0 à <entier> (exclu).

- Les <instructions> indentées qui suivent seront exécutées pour chaque valeur prise par la variable.
- L'instruction **break** permet de sortir de la boucle de façon anticipée.
- La boucle for permet donc de répéter un nombre prédéfini de fois des instructions, on dit que c'est une boucle bornée.



### Boucles for pour parcourir un itérable

### Boucles for pour parcourir un itérable

- Les instructions :

```
1  for <variable> in <iterable>:  
2      <instructions>
```

permet à <variable> de prendre les valeurs présentes dans <iterable>.

### Boucles for pour parcourir un itérable

- Les instructions :

```
1  for <variable> in <iterable>:  
2      <instructions>
```

permet à <variable> de prendre les valeurs présentes dans <iterable>.

- Les <instructions> indentées qui suivent seront exécutées pour chaque valeur prise par la variable.

### Boucles for pour parcourir un itérable

- Les instructions :

```
1  for <variable> in <iterable>:  
2      <instructions>
```

permet à <variable> de prendre les valeurs présentes dans <iterable>.

- Les <instructions> indentées qui suivent seront exécutées pour chaque valeur prise par la variable.
- Une chaîne de ce caractère est un itérable, la variable prend alors comme valeur chacun des caractères de la chaîne.

### Exemple 1

Ecrire et tester une fonction syracuse qui prend en argument un entier naturel  $n$  et renvoie  $n/2$  si  $n$  est pair et  $3n + 1$  sinon.

### Exemple 1

Ecrire et tester une fonction `syracuse` qui prend en argument un entier naturel  $n$  et renvoie  $n/2$  si  $n$  est pair et  $3n + 1$  sinon.

```
1 def syracuse(n):  
2     if n%2 == 0:  
3         return n//2  
4     else:  
5         return 3*n+1
```

### Exemple 2

Ecrire une fonction `serie_harmonique` qui prend en argument un entier  $n$  et renvoie la somme  $\sum_{k=1}^n \frac{1}{k}$

### Exemple 2

Ecrire une fonction `serie_harmonique` qui prend en argument un entier  $n$  et renvoie la somme  $\sum_{k=1}^n \frac{1}{k}$

```
1 def serie_harmonique(n):  
2     somme = 0  
3     for i in range(1,n+1):  
4         somme = somme + 1/i  
5     return somme
```



### Exemple 3

Ecrire une fonction `pgcd` qui prend en argument deux entiers naturels  $a$  et  $b$  et renvoie leur PGCD.

### Exemple 3

Ecrire une fonction `pgcd` qui prend en argument deux entiers naturels  $a$  et  $b$  et renvoie leur PGCD.

🌀 on rappelle que l'algorithme consiste –tant que  $b$  n'est pas nul– à effectuer la division euclidienne de  $a$  par  $b$ . En remplaçant à chaque étape  $a$  par  $b$  et  $b$  par  $r$ .

### Exemple 3

Ecrire une fonction `pgcd` qui prend en argument deux entiers naturels  $a$  et  $b$  et renvoie leur PGCD.

🌀 on rappelle que l'algorithme consiste –tant que  $b$  n'est pas nul– à effectuer la division euclidienne de  $a$  par  $b$ . En remplaçant à chaque étape  $a$  par  $b$  et  $b$  par  $r$ .

- Version 1 :

```
1 def pgcd(a,b):  
2     while b!=0:  
3         a,b = b, a%b  
4     return a
```

### Exemple 3

Ecrire une fonction `pgcd` qui prend en argument deux entiers naturels  $a$  et  $b$  et renvoie leur PGCD.

🌀 on rappelle que l'algorithme consiste –tant que  $b$  n'est pas nul– à effectuer la division euclidienne de  $a$  par  $b$ . En remplaçant à chaque étape  $a$  par  $b$  et  $b$  par  $r$ .

- Version 1 :

```
1 def pgcd(a,b):  
2     while b!=0:  
3         a,b = b, a%b  
4     return a
```

- Version 2 :

```
1 def pgcd(a,b):  
2     if b == 0:  
3         return a  
4     return pgcd(b,a%b)
```

### Exemple 3

Ecrire une fonction `pgcd` qui prend en argument deux entiers naturels  $a$  et  $b$  et renvoie leur PGCD.

🌀 on rappelle que l'algorithme consiste –tant que  $b$  n'est pas nul– à effectuer la division euclidienne de  $a$  par  $b$ . En remplaçant à chaque étape  $a$  par  $b$  et  $b$  par  $r$ .

- Version 1 : **iterative**

```
1 def pgcd(a,b):  
2     while b!=0:  
3         a,b = b, a%b  
4     return a
```

- Version 2 : **réursive**

```
1 def pgcd(a,b):  
2     if b == 0:  
3         return a  
4     return pgcd(b,a%b)
```

### Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du même type).

### Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du même type).
- Une liste se note entre crochets : `[` et `]`

### Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du même type).
- Une liste se note entre crochets : `[` et `]`
- Les éléments sont séparés par des virgules



### Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du même type).
- Une liste se note entre crochets : `[` et `]`
- Les éléments sont séparés par des virgules
- Les éléments d'une liste sont repérés par leur position dans la liste, on dit leur **indice**. Attention, la numérotation commence à zéro.

### Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du même type).
- Une liste se note entre crochets : `[` et `]`
- Les éléments sont séparés par des virgules
- Les éléments d'une liste sont repérés par leur position dans la liste, on dit leur **indice**. Attention, la numérotation commence à zéro.

### Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du même type).
- Une liste se note entre crochets : [ et ]
- Les éléments sont séparés par des virgules
- Les éléments d'une liste sont repérés par leur position dans la liste, on dit leur **indice**. Attention, la numérotation commence à zéro.
- On peut accéder à un élément en indiquant le nom de la liste puis l'indice de cet élément entre crochet

### Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du même type).
- Une liste se note entre crochets : [ et ]
- Les éléments sont séparés par des virgules
- Les éléments d'une liste sont repérés par leur position dans la liste, on dit leur **indice**. Attention, la numérotation commence à zéro.
- On peut accéder à un élément en indiquant le nom de la liste puis l'indice de cet élément entre crochet
- L'erreur `IndexError` indique qu'on tente d'accéder à un indice qui n'existe pas.

### Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du même type).
- Une liste se note entre crochets : `[` et `]`
- Les éléments sont séparés par des virgules
- Les éléments d'une liste sont repérés par leur position dans la liste, on dit leur **indice**. Attention, la numérotation commence à zéro.
- On peut accéder à un élément en indiquant le nom de la liste puis l'indice de cet élément entre crochet
- L'erreur `IndexError` indique qu'on tente d'accéder à un indice qui n'existe pas.
- La longueur d'une liste (ie. son nombre d'éléments) s'obtient à l'aide de la fonction `len`.

### Opérations sur les listes

Les opérations suivantes permettent de manipuler les listes (ajout, suppression, insertion d'éléments). On fera bien attention à la syntaxe on met le nom de la liste suivi d'un point suivi de l'opération à effectuer (voir exemples)

- **append** : permet d'ajouter un élément à la fin d'une liste. Par exemple : `ma_liste.append(elt)` va ajouter `elt` à la fin de `ma_liste`.

### Opérations sur les listes

Les opérations suivantes permettent de manipuler les listes (ajout, suppression, insertion d'éléments). On fera bien attention à la syntaxe on met le nom de la liste suivi d'un point suivi de l'opération à effectuer (voir exemples)

- **append** : permet d'ajouter un élément à la fin d'une liste. Par exemple : `ma_liste.append(elt)` va ajouter `elt` à la fin de `ma_liste`.
- **pop** permet de récupérer un élément de la liste tout en le supprimant de la liste. Par exemple `elt=ma_liste.pop(2)` va mettre dans `elt` `ma_liste[2]` et dans le même temps supprimer cet élément de la liste.

### Opérations sur les listes

Les opérations suivantes permettent de manipuler les listes (ajout, suppression, insertion d'éléments). On fera bien attention à la syntaxe on met le nom de la liste suivi d'un point suivi de l'opération à effectuer (voir exemples)

- **append** : permet d'ajouter un élément à la fin d'une liste. Par exemple : `ma_liste.append(elt)` va ajouter `elt` à la fin de `ma_liste`.
- **pop** permet de récupérer un élément de la liste tout en le supprimant de la liste. Par exemple `elt=ma_liste.pop(2)` va mettre dans `elt` `ma_liste[2]` et dans le même temps supprimer cet élément de la liste.
  - ! On utilisera le plus souvent **pop** sans argument, dans ce cas c'est le dernier élément de la liste qui est supprimé



### ! Spécificité de Python

En Python, on manipule des *objets*, une variable est l'association entre un nom et l'objet qu'il référence. Certains objets, sont *mutables* et d'autres non et cela à des conséquences importantes :

- Cas *non mutable*, l'objet ne peut pas être modifié, c'est le cas des entiers, des chaînes de caractères, des flottants, des booléens. Par exemple,

```
1  x = 5      #x référence l'objet 5
2  y = x      #y référence aussi l'objet 5
3  x = x + 2  #en faisant x+2 on crée un nouvel objet, x référence ce nouvel
    ↪ objet (mais pas y)
```

- Cas *mutable*, l'objet peut être modifié et donc dans ce cas, toutes les références à cet objet vont désigner l'objet modifié. Par exemple,

```
1  x = [5]    #x référence [5]
2  y = x      #y référence aussi [5]
3  x.append(2) #On modifie [5] (on ne crée pas un nouvel objet, donc y
    ↪ contient aussi [5,2])
```

### Création de listes

On peut créer des listes de diverses façons en Python :

### Création de listes

On peut créer des listes de diverses façons en Python :

- Par ajout **succesif d'élément** on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction **append**.

### Création de listes

On peut créer des listes de diverses façons en Python :

- **Par ajout succesif d'élément** on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- **Par répétition du même élément** on utilise alors le caractère `*` pour indiquer le nombre de répétitions.

### Création de listes

On peut créer des listes de diverses façons en Python :

- **Par ajout succesif d'élément** on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- **Par répétition du même élément** on utilise alors le caractère `*` pour indiquer le nombre de répétitions.

Par exemple : `hesitation = ["euh"]*4`

### Création de listes

On peut créer des listes de diverses façons en Python :

- **Par ajout succesif d'élément** on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- **Par répétition du même élément** on utilise alors le caractère `*` pour indiquer le nombre de répétitions.

Par exemple : `hesitation = ["euh"]*4`

### Création de listes

On peut créer des listes de diverses façons en Python :

- **Par ajout succesif d'élément** on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- **Par répétition du même élément** on utilise alors le caractère `*` pour indiquer le nombre de répétitions.  
Par exemple : `hesitation = ["euh"]*4`
- **Par compréhension**, c'est-à-dire en indiquant la définition des éléments qui composent la liste.

### Création de listes

On peut créer des listes de diverses façons en Python :

- **Par ajout succesif d'élément** on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- **Par répétition du même élément** on utilise alors le caractère `*` pour indiquer le nombre de répétitions.

Par exemple : `hesitation = ["euh"]*4`

- **Par compréhension**, c'est-à-dire en indiquant la définition des éléments qui composent la liste.

Par exemple la liste `puissances2 = [1, 2, 4, 8, 16, 32, 64, 128]` est constitué des huit premières puissances de 2



### Création de listes

On peut créer des listes de diverses façons en Python :

- **Par ajout succesif d'élément** on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- **Par répétition du même élément** on utilise alors le caractère `*` pour indiquer le nombre de répétitions.

Par exemple : `hesitation = ["euh"]*4`

- **Par compréhension**, c'est-à-dire en indiquant la définition des éléments qui composent la liste.

Par exemple la liste `puissances2 = [1, 2, 4, 8, 16, 32, 64, 128]` est constitué des huit premières puissances de 2

Elle contient donc  $2^0, 2^1, 2^2, \dots, 2^7$ , ce qui se traduit en Python par :

### Création de listes

On peut créer des listes de diverses façons en Python :

- **Par ajout succesif d'élément** on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- **Par répétition du même élément** on utilise alors le caractère `*` pour indiquer le nombre de répétitions.

Par exemple : `hesitation = ["euh"]*4`

- **Par compréhension**, c'est-à-dire en indiquant la définition des éléments qui composent la liste.

Par exemple la liste `puissances2 = [1, 2, 4, 8, 16, 32, 64, 128]` est constitué des huit premières puissances de 2

Elle contient donc  $2^0, 2^1, 2^2, \dots, 2^7$ , ce qui se traduit en Python par :

`puissances2 = [2**k for k in range(8)]`

### Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.

### Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.  
Par exemple si la liste est `l=[2,3,5,7,11,13,17,19]` alors `l[2:4]` est une liste qui contient `[5,7]`.

### Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.  
Par exemple si la liste est `l=[2,3,5,7,11,13,17,19]` alors `l[2:4]` est une liste qui contient `[5,7]`.
- Si l'indice du premier est omis alors la tranche commence à l'indice 0.

### Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.  
Par exemple si la liste est `l=[2,3,5,7,11,13,17,19]` alors `l[2:4]` est une liste qui contient `[5,7]`.
- Si l'indice du premier est omis alors la tranche commence à l'indice 0.  
Avec la même liste `l`, on a `l[:5]` est une liste qui contient `[2,3,5,7,11]`.

### Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.  
Par exemple si la liste est `l=[2,3,5,7,11,13,17,19]` alors `l[2:4]` est une liste qui contient `[5,7]`.
- Si l'indice du premier est omis alors la tranche commence à l'indice 0.  
Avec la même liste `l`, on a `l[:5]` est une liste qui contient `[2,3,5,7,11]`.
- Si l'indice du dernier est omis alors la tranche va jusqu'à la fin de la liste.

### Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.  
Par exemple si la liste est `l=[2,3,5,7,11,13,17,19]` alors `l[2:4]` est une liste qui contient `[5,7]`.
- Si l'indice du premier est omis alors la tranche commence à l'indice 0.  
Avec la même liste `l`, on a `l[:5]` est une liste qui contient `[2,3,5,7,11]`.
- Si l'indice du dernier est omis alors la tranche va jusqu'à la fin de la liste.  
Avec la même liste `l`, on a `l[7:]` est une liste qui contient `[19]`.



# C16 Un peu de Python

## 8. Chaîne de caractères et tuples

### Tuples

- Les **tuples** sont le pendant non mutables des listes. Ils se notent entre parenthèses ( et ), les éléments sont aussi séparés par des virgules.
- De même que pour les listes, on peut accéder à la longueur avec **len**, aux éléments avec la notation crochet et le parcours avec une boucle **for** est aussi possible.
- La modification par contre n'est pas possible

### Exemple

```
1 anniv = (31,"Janvier",1956)
2 print("Mois de naissance = ",anniv[1])
3 anniv[2] = 1970 #provoque une erreur
```

### Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaînes de caractères.

### Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaînes de caractères.

Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre "n"

### Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaînes de caractères.  
Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre "n"
- Le parcours par élément peut aussi se faire sur une chaîne de caractères.

### Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaînes de caractères.  
Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre "n"
- Le parcours par élément peut aussi se faire sur une chaîne de caractères.  
Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

### Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaînes de caractères.

Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre "n"

- Le parcours par élément peut aussi se faire sur une chaîne de caractères. Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

```
1 for lettre in mot:  
2     print(lettre)
```

### Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaînes de caractères.

Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre "n"

- Le parcours par élément peut aussi se faire sur une chaîne de caractères. Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

```
1  for lettre in mot:  
2  print(lettre)
```

- Comme les tuples, les chaînes de caractères sont non mutables.

### Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaînes de caractères.

Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre "n"

- Le parcours par élément peut aussi se faire sur une chaîne de caractères. Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

```
1 for lettre in mot:  
2     print(lettre)
```

- Comme les tuples, les chaînes de caractères sont non mutables.
- ❗ Les variables lues au clavier (instruction `input`) ou issus de la lecture d'un fichier sont des chaînes de caractères. On doit les convertir dans le type approprié pour les utiliser comme nombre.



### Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaînes de caractères.

Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre "n"

- Le parcours par élément peut aussi se faire sur une chaîne de caractères. Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

```
1 for lettre in mot:
2     print(lettre)
```

- Comme les tuples, les chaînes de caractères sont non mutables.
- ❗ Les variables lues au clavier (instruction `input`) ou issus de la lecture d'un fichier sont des chaînes de caractères. On doit les convertir dans le type approprié pour les utiliser comme nombre.
- La fonction `split` permet de renvoyer une liste de sous chaînes en utilisant le séparateur donné en argument.

# C16 Un peu de Python

## 8. Chaîne de caractères et tuples

### Exemple

Ecrire une fonction `check_date` qui prend en argument une chaîne de caractères et renvoie `True` si cette chaîne est une date valide au format JJ/MM/AAAA et `False` sinon. Pour simplifier on testera simplement que le jour est entre 1 et 31 et le mois entre 1 et 12.

# C16 Un peu de Python

## 8. Chaîne de caractères et tuples

### Exemple

Ecrire une fonction `check_date` qui prend en argument une chaîne de caractères et renvoie `True` si cette chaîne est une date valide au format JJ/MM/AAAA et `False` sinon. Pour simplifier on testera simplement que le jour est entre 1 et 31 et le mois entre 1 et 12.

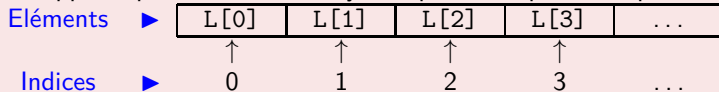
```
1 def check_date(date):
2     ldate = date.split("/")
3     if len(ldate)!=3:
4         return False
5     jour,mois = int(ldate[0]),int(ldate[1])
6     if jour<1 or jour>31 or mois<1 or mois>12:
7         return False
8     return True
```

# C16 Un peu de Python

## 8. Chaîne de caractères et tuples

### Parcours d'une liste

On rappelle qu'une liste `L`, en Python peut se représenter par le schéma suivant :



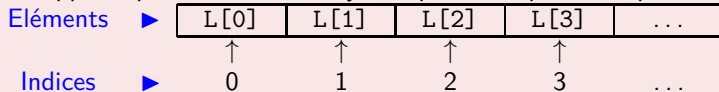
On peut parcourir cette liste :

# C16 Un peu de Python

## 8. Chaîne de caractères et tuples

### Parcours d'une liste

On rappelle qu'une liste `L`, en Python peut se représenter par le schéma suivant :



On peut parcourir cette liste :

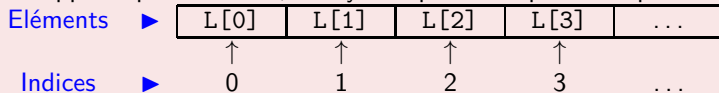
- **Par indice** (on se place sur la seconde ligne du schéma ci-dessus) et on crée une variable (un entier) qui va parcourir la liste des indices :

```
for indice in range(len(L))
```

Il faut alors accéder aux éléments en utilisant leurs indices.

### Parcours d'une liste

On rappelle qu'une liste `L`, en Python peut se représenter par le schéma suivant :



On peut parcourir cette liste :

- **Par indice** (on se place sur la seconde ligne du schéma ci-dessus) et on crée une variable (un entier) qui va parcourir la liste des indices :

```
for indice in range(len(L))
```

Il faut alors accéder aux éléments en utilisant leurs indices.

- **Par élément** (on se place sur la première ligne du schéma ci-dessus) et on crée une variable qui va parcourir directement la liste des éléments :

```
for element in L
```

La variable de parcours (ici `element`) contient alors directement les éléments).

# C16 Un peu de Python

## 8. Chaîne de caractères et tuples

### Exemple 1

Ecrire une fonction `est_dans` qui prend en argument un entier `n` et une liste d'entiers `l` et renvoie `True` si `n` est dans `l` et `False` sinon. On écrira une version utilisant un parcours par valeur et une version utilisant un parcours par indice.

# C16 Un peu de Python

## 8. Chaîne de caractères et tuples

### Exemple 1

Ecrire une fonction `est_dans` qui prend en argument un entier `n` et une liste d'entiers `l` et renvoie `True` si `n` est dans `l` et `False` sinon. On écrira une version utilisant un parcours par valeur et une version utilisant un parcours par indice.

- Parcours par élément :

```
1  def est_dans(n,l):  
2      for x in l:  
3          if x==n:  
4              return True  
5      return False
```



# C16 Un peu de Python

## 8. Chaîne de caractères et tuples

### Exemple 1

Ecrire une fonction `est_dans` qui prend en argument un entier `n` et une liste d'entiers `l` et renvoie `True` si `n` est dans `l` et `False` sinon. On écrira une version utilisant un parcours par valeur et une version utilisant un parcours par indice.

- Parcours par élément :

```
1  def est_dans(n,l):
2      for x in l:
3          if x==n:
4              return True
5      return False
```

- Parcours par indice :

```
1  def est_dans_ind(n,l):
2      for i in range(len(l)):
3          if l[i]==n:
4              return True
5      return False
```

### Exemple 2

Ecrire une fonction `max_liste` qui prend en argument une liste non vide d'entiers 1 et renvoie le maximum des éléments de cette liste

### Exemple 2

Ecrire une fonction `max_liste` qui prend en argument une liste non vide d'entiers `l` et renvoie le maximum des éléments de cette liste

```
1 def max_liste(l):  
2     # la liste doit être non vide  
3     assert len(l) != 0  
4     current_max = l[0]  
5     for elt in l:  
6         if elt > current_max:  
7             current_max = elt  
8     return current_max
```

### Gestions des fichiers en Python

En python, on peut ouvrir un fichier présent sur l'ordinateur à l'aide de l'instruction `open`. Cette instruction renvoie une variable appelée `descripteur de fichier` et prend un paramètre indiquant le mode d'ouverture du fichier :

### Gestions des fichiers en Python

En python, on peut ouvrir un fichier présent sur l'ordinateur à l'aide de l'instruction `open`. Cette instruction renvoie une variable appelée `descripteur de fichier` et prend un paramètre indiquant le mode d'ouverture du fichier :

- `"r"` (read) pour ouvrir le fichier en lecture. C'est le mode par défaut.

### Gestions des fichiers en Python

En python, on peut ouvrir un fichier présent sur l'ordinateur à l'aide de l'instruction `open`. Cette instruction renvoie une variable appelée `descripteur de fichier` et prend un paramètre indiquant le mode d'ouverture du fichier :

- `"r"` (read) pour ouvrir le fichier en lecture. C'est le mode par défaut.
- `"w"` (write) pour ouvrir le fichier en écriture. Attention, le contenu initial du fichier est alors perdu.

### Gestions des fichiers en Python

En python, on peut ouvrir un fichier présent sur l'ordinateur à l'aide de l'instruction `open`. Cette instruction renvoie une variable appelée `descripteur de fichier` et prend un paramètre indiquant le mode d'ouverture du fichier :

- `"r"` (read) pour ouvrir le fichier en lecture. C'est le mode par défaut.
- `"w"` (write) pour ouvrir le fichier en écriture. Attention, le contenu initial du fichier est alors perdu.
- `"a"` (append) pour ouvrir le fichier en ajout.

### Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier créé à l'aide de l'instruction `open` :

### Exemples



### Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier créé à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`

### Exemples

### Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier créé à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`

### Exemples

### Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier créée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`

### Exemples

### Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier créée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

### Exemples

### Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier créée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

### Exemples

### Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

### Exemples

Ouvrir le fichier "truc.txt", lire sa première ligne puis le refermer.

### Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier créée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

### Exemples

Ouvrir le fichier "truc.txt", lire sa première ligne puis le refermer.

```
fic = open("truc.txt","r")
```

### Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier créée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

### Exemples

Ouvrir le fichier "truc.txt", lire sa première ligne puis le refermer.

```
fic = open("truc.txt","r")  
lig1 = fic.readline()
```



### Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier créée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

### Exemples

Ouvrir le fichier "truc.txt", lire sa première ligne puis le refermer.

```
fic = open("truc.txt","r")  
lig1 = fic.readline()  
fic.close()
```

### Algorithmique

Dans l'étude des algorithmes ([algorithmique](#)), on s'intéresse aux trois problèmes suivants :

### Algorithmique

Dans l'étude des algorithmes ([algorithmique](#)), on s'intéresse aux trois problèmes suivants :

- 1 [terminaison](#) : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes avec des boucles non bornées et les récursions.)

### Algorithmique

Dans l'étude des algorithmes (**algorithmique**), on s'intéresse aux trois problèmes suivants :

- 1 **terminaison** : peut-on garantir que l'algorithme se termine en un temps fini ?  
(ne concerne que les algorithmes avec des boucles non bornées et les récursions.)
- 2 **correction** : peut-on garantir que l'algorithme fournit la réponse attendue ?

### Algorithmique

Dans l'étude des algorithmes (**algorithmique**), on s'intéresse aux trois problèmes suivants :

- 1 **terminaison** : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes avec des boucles non bornées et les récursions.)
- 2 **correction** : peut-on garantir que l'algorithme fournit la réponse attendue ?
- 3 **complexité** : évolution du temps d'exécution de l'algorithme en fonction de la taille des données. En particulier, le temps d'exécution d'un algorithme sur une entrée donnée sera-t-il « raisonnable » ?

### Algorithmique

Dans l'étude des algorithmes (**algorithmique**), on s'intéresse aux trois problèmes suivants :

- 1 **terminaison** : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes avec des boucles non bornées et les récursions.)
- 2 **correction** : peut-on garantir que l'algorithme fournit la réponse attendue ?
- 3 **complexité** : évolution du temps d'exécution de l'algorithme en fonction de la taille des données. En particulier, le temps d'exécution d'un algorithme sur une entrée donnée sera-t-il « raisonnable » ?

L'algorithme étudié doit avoir une **spécification** précise (entrées, sorties, préconditions, postconditions, effets de bord). On parle d'algorithmes (et non de programmes) car ces questions sont indépendantes de l'implémentation dans un langage de programmation quelconque.

### Exemple

Nous avons déjà rencontrés plusieurs algorithmes de tri (tri par insertion, tri par sélection, tri fusion). On cherche maintenant :

### Exemple

Nous avons déjà rencontrés plusieurs algorithmes de tri (tri par insertion, tri par sélection, tri fusion). On cherche maintenant :

- A obtenir une **preuve mathématique** que ces algorithmes se terminent et donc n'entrent jamais dans une boucle infinie quelques soient les données.



### Exemple

Nous avons déjà rencontrés plusieurs algorithmes de tri (tri par insertion, tri par sélection, tri fusion). On cherche maintenant :

- A obtenir une **preuve mathématique** que ces algorithmes se terminent et donc n'entrent jamais dans une boucle infinie quelques soient les données.
- A obtenir une **preuve mathématique** que ces algorithmes trient effectivement les listes de nombres données en argument et ce quelques soient leur taille et les valeurs qu'elles contiennent.

### Exemple

Nous avons déjà rencontrés plusieurs algorithmes de tri (tri par insertion, tri par sélection, tri fusion). On cherche maintenant :

- A obtenir une **preuve mathématique** que ces algorithmes se terminent et donc n'entrent jamais dans une boucle infinie quelques soient les données.
- A obtenir une **preuve mathématique** que ces algorithmes trient effectivement les listes de nombres données en argument et ce quelques soient leur taille et les valeurs qu'elles contiennent.
- A comparer ces algorithmes en quantifiant leur efficacité (qui peut être mesuré de diverses façons).

### Définitions

- On dit qu'un algorithme **termine** si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.

### Définitions

- On dit qu'un algorithme **termine** si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un **variant de boucle** est une quantité :

### Définitions

- On dit qu'un algorithme **termine** si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un **variant de boucle** est une quantité :
  - à valeurs dans  $\mathbb{N}$ .

### Définitions

- On dit qu'un algorithme **termine** si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un **variant de boucle** est une quantité :
  - à valeurs dans  $\mathbb{N}$ .
  - qui décroît strictement à chaque passage dans la boucle.

### Définitions

- On dit qu'un algorithme **termine** si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un **variant de boucle** est une quantité :
  - à valeurs dans  $\mathbb{N}$ .
  - qui décroît strictement à chaque passage dans la boucle.
- Pour un algorithme récursif, un **variant** est une quantité :

# C16 Un peu de Python

## 11. Rappels d'algorithmique : terminaison

### Définitions

- On dit qu'un algorithme **termine** si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un **variant de boucle** est une quantité :
  - à valeurs dans  $\mathbb{N}$ .
  - qui décroît strictement à chaque passage dans la boucle.
- Pour un algorithme récursif, un **variant** est une quantité :
  - à valeurs dans  $\mathbb{N}$ .



### Définitions

- On dit qu'un algorithme **termine** si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un **variant de boucle** est une quantité :
  - à valeurs dans  $\mathbb{N}$ .
  - qui décroît strictement à chaque passage dans la boucle.
- Pour un algorithme récursif, un **variant** est une quantité :
  - à valeurs dans  $\mathbb{N}$ .
  - qui décroît strictement à chaque appel récursif.

# C16 Un peu de Python

## 11. Rappels d'algorithmique : terminaison

### Définitions

- On dit qu'un algorithme **termine** si il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un **variant de boucle** est une quantité :
  - à valeurs dans  $\mathbb{N}$ .
  - qui décroît strictement à chaque passage dans la boucle.
- Pour un algorithme récursif, un **variant** est une quantité :
  - à valeurs dans  $\mathbb{N}$ .
  - qui décroît strictement à chaque appel récursif.

### Preuve de la terminaison d'un algorithme

Pour prouver la terminaison d'un algorithme, il suffit de trouver un **variant de boucle** pour chaque boucle non bornée qu'il contient. Et un **variant** pour chaque fonction récursive.

### Exemple 1

On considère la fonction ci-dessous :

```
1  # Renvoie le quotient dans la division euclidienne de a par b avec a  
   ↪ et b deux entiers naturels et b non nul  
2  def quotient(a, b):  
3      assert (a >= 0 and b > 0)  
4      q = 0  
5      while (a - b >= 0):  
6          a = a - b  
7          q = q + 1  
8      return q
```

### Exemple 1

On considère la fonction ci-dessous :

```
1  # Renvoie le quotient dans la division euclidienne de a par b avec a  
   ↪ et b deux entiers naturels et b non nul  
2  def quotient(a, b):  
3      assert (a >= 0 and b > 0)  
4      q = 0  
5      while (a - b >= 0):  
6          a = a - b  
7          q = q + 1  
8      return q
```

En trouvant un variant de boucle, prouver la terminaison de cette fonction.

### Correction de l'exemple 1

Montrons que la variable  $a$  est un variant de boucle, c'est-à-dire qu'elle prend ses valeurs dans  $\mathbb{N}$  et décroît strictement à chaque passage dans la boucle.

### Correction de l'exemple 1

Montrons que la variable  $a$  est un variant de boucle, c'est-à-dire qu'elle prend ses valeurs dans  $\mathbb{N}$  et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de  $a$  est un entier naturel (précondition)

### Correction de l'exemple 1

Montrons que la variable  $a$  est un variant de boucle, c'est-à-dire qu'elle prend ses valeurs dans  $\mathbb{N}$  et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de  $a$  est un entier naturel (précondition)
- A chaque tour de boucle la valeur de  $a$  diminue (de  $b > 0$ ).

### Correction de l'exemple 1

Montrons que la variable  $a$  est un variant de boucle, c'est-à-dire qu'elle prend ses valeurs dans  $\mathbb{N}$  et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de  $a$  est un entier naturel (précondition)
- A chaque tour de boucle la valeur de  $a$  diminue (de  $b > 0$ ).
- La nouvelle valeur de  $a$  est  $a-b$  qui est garantie positive par condition d'entrée dans la boucle



### Correction de l'exemple 1

Montrons que la variable  $a$  est un variant de boucle, c'est-à-dire qu'elle prend ses valeurs dans  $\mathbb{N}$  et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de  $a$  est un entier naturel (précondition)
- A chaque tour de boucle la valeur de  $a$  diminue (de  $b > 0$ ).
- La nouvelle valeur de  $a$  est  $a-b$  qui est garantie positive par condition d'entrée dans la boucle

Les trois éléments ci-dessus prouvent que la variable  $a$  est un variant de la boucle `while` de ce programme, par conséquent cette boucle se termine.

### Exemple 2

On considère la fonction ci-dessous :

```
1  # Renvoie True si elt est dans liste et False sinon
2  def est_dans(elt, liste):
3      if liste == []:
4          return False
5      return elt == liste[0] or est_dans(elt, liste[1:])
```

### Exemple 2

On considère la fonction ci-dessous :

```
1  # Renvoie True si elt est dans liste et False sinon
2  def est_dans(elt, liste):
3      if liste == []:
4          return False
5      return elt == liste[0] or est_dans(elt, liste[1:])
```

Prouver que cette fonction récursive termine

### Correction de l'exemple 2

Montrons que la longueur de la liste `liste` est un variant, c'est-à-dire qu'elle prend ses valeurs dans  $\mathbb{N}$  et décroît strictement à chaque appel récursif.

### Correction de l'exemple 2

Montrons que la longueur de la liste `liste` est un variant, c'est-à-dire qu'elle prend ses valeurs dans  $\mathbb{N}$  et décroît strictement à chaque appel récursif.

- La longueur d'une liste est à valeur dans  $\mathbb{N}$

### Correction de l'exemple 2

Montrons que la longueur de la liste `liste` est un variant, c'est-à-dire qu'elle prend ses valeurs dans  $\mathbb{N}$  et décroît strictement à chaque appel récursif.

- La longueur d'une liste est à valeur dans  $\mathbb{N}$
- A chaque appel récursif, on enlève un élément de `liste` (sa tête) et donc la taille de la liste diminue de 1.

### Correction de l'exemple 2

Montrons que la longueur de la liste `liste` est un variant, c'est-à-dire qu'elle prend ses valeurs dans  $\mathbb{N}$  et décroît strictement à chaque appel récursif.

- La longueur d'une liste est à valeur dans  $\mathbb{N}$
- A chaque appel récursif, on enlève un élément de `liste` (sa tête) et donc la taille de la liste diminue de 1.

Les deux éléments ci-dessus prouvent que la longueur de la liste est un variant et que donc cette fonction récursive termine.

### Remarque

Dans les exemples ci-dessus la mise en évidence du variant est facile (et ce sera le cas en général dans nos algorithmes). Mais, les preuves de terminaison sont loin d'être toujours aussi évidentes.



# C16 Un peu de Python

## 11. Rappels d'algorithmique : terminaison

### Remarque

Dans les exemples ci-dessus la mise en évidence du variant est facile (et ce sera le cas en général dans nos algorithmes). Mais, les preuves de terminaison sont loin d'être toujours aussi évidentes.

Par exemple, si on considère la fonction suivante :

```
1 def syracuse(n):  
2     while n != 1:  
3         if n % 2 == 0:  
4             n = n//2  
5         else:  
6             n = 3*n+1  
7     return True
```

# C16 Un peu de Python

## 11. Rappels d'algorithmique : terminaison

### Remarque

Dans les exemples ci-dessus la mise en évidence du variant est facile (et ce sera le cas en général dans nos algorithmes). Mais, les preuves de terminaison sont loin d'être toujours aussi évidentes.

Par exemple, si on considère la fonction suivante :

```
1 def syracuse(n):  
2     while n != 1:  
3         if n % 2 == 0:  
4             n = n/2  
5         else:  
6             n = 3*n+1  
7     return True
```

Prouver sa terminaison reviendrait à prouver la conjecture de syracuse qui résiste aux mathématiciens depuis un siècle !

### Correction d'un algorithme

On dira qu'un algorithme est **correct**

### Correction d'un algorithme

On dira qu'un algorithme est **correct** lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.

### Correction d'un algorithme

On dira qu'un algorithme est **correct** lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.

On parle de **correction partielle** quand le résultat est correct lorsque l'algorithme s'arrête. Et de **correction totale** lorsque la correction est partielle et que l'algorithme se termine.

### Tests et correction

### Correction d'un algorithme

On dira qu'un algorithme est **correct** lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.

On parle de **correction partielle** quand le résultat est correct lorsque l'algorithme s'arrête. Et de **correction totale** lorsque la correction est partielle et que l'algorithme se termine.

### Tests et correction

Des tests ne permettent **pas** de prouver qu'un algorithme est correct.

### Correction d'un algorithme

On dira qu'un algorithme est **correct** lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.

On parle de **correction partielle** quand le résultat est correct lorsque l'algorithme s'arrête. Et de **correction totale** lorsque la correction est partielle et que l'algorithme se termine.

### Tests et correction

Des tests ne permettent **pas** de prouver qu'un algorithme est correct. En effet, ils ne permettent de valider le comportement de l'algorithme que dans quelques cas particuliers et jamais dans le cas général

### Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme itératif, on utilise la notion **d'invariant de boucle**. C'est une propriété du programme qui



### Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme itératif, on utilise la notion **d'invariant de boucle**. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.

### Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme itératif, on utilise la notion d'**invariant de boucle**. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.
- reste vraie à chaque itération si elle l'était à l'itération précédente.

### Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme itératif, on utilise la notion d'**invariant de boucle**. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.
- reste vraie à chaque itération si elle l'était à l'itération précédente.

La méthode est similaire à une récurrence mathématique (les deux étapes précédentes correspondent à l'initialisation et à l'hérédité).

### Exemple 1

On considère la fonction ci-dessous :

```
1  # Renvoie le quotient dans la division euclidienne de a par b avec a  
   ↪ et b deux entiers naturels et b non nul  
2  def quotient(a, b):  
3      assert (a >= 0 and b > 0)  
4      q = 0  
5      while (a - b >= 0):  
6          a = a - b  
7          q = q + 1  
8      return q
```

### Exemple 1

On considère la fonction ci-dessous :

```
1  # Renvoie le quotient dans la division euclidienne de a par b avec a  
   ↪ et b deux entiers naturels et b non nul  
2  def quotient(a, b):  
3      assert (a >= 0 and b > 0)  
4      q = 0  
5      while (a - b >= 0):  
6          a = a - b  
7          q = q + 1  
8      return q
```

En trouvant un invariant de boucle, prouver la correction de cette fonction.

### Correction de l'exemple 1

On note  $a_0$  la valeur initiale de la variable  $a$ .

### Correction de l'exemple 1

On note  $a_0$  la valeur initiale de la variable  $a$ . Montrons que la propriété :  
«  $a_0 = bq + a$  » est un invariant de boucle.

### Correction de l'exemple 1

On note  $a_0$  la valeur initiale de la variable  $a$ . Montrons que la propriété :  
«  $a_0 = bq + a$  » est un invariant de boucle.



### Correction de l'exemple 1

On note  $a_0$  la valeur initiale de la variable  $a$ . Montrons que la propriété :

«  $a_0 = bq + a$  » est un invariant de boucle.

- La propriété est vraie avant d'entrée dans la boucle puisque on alors  $q = 0$  et  $a_0$  et donc  $bq + a = a_0$ .

### Correction de l'exemple 1

On note  $a_0$  la valeur initiale de la variable  $a$ . Montrons que la propriété :

«  $a_0 = bq + a$  » est un invariant de boucle.

- La propriété est vraie avant d'entrée dans la boucle puisque on alors  $q = 0$  et  $a = a_0$  et donc  $bq + a = a_0$ .
- Supposons la propriété vraie au début d'une itération de la boucle et montrons qu'elle est conservée à l'itération suivante. En notant  $a'$  (resp.  $q'$ ) la valeur de  $a$  (resp.  $q$ ) après l'itération, on a :

$$b \times q' + a' = b \times (q + 1) + a - b$$

$$b \times q' + a' = b \times q + a$$

et puisque la propriété est supposé vraie au début de l'itération

$$b \times q' + a' = a_0$$

### Correction de l'exemple 1

On note  $a_0$  la valeur initiale de la variable  $a$ . Montrons que la propriété :

«  $a_0 = bq + a$  » est un invariant de boucle.

- La propriété est vraie avant d'entrée dans la boucle puisque on alors  $q = 0$  et  $a = a_0$  et donc  $bq + a = a_0$ .
- Supposons la propriété vraie au début d'une itération de la boucle et montrons qu'elle est conservée à l'itération suivante. En notant  $a'$  (resp.  $q'$ ) la valeur de  $a$  (resp.  $q$ ) après l'itération, on a :

$$b \times q' + a' = b \times (q + 1) + a - b$$

$$b \times q' + a' = b \times q + a$$

et puisque la propriété est supposé vraie au début de l'itération

$$b \times q' + a' = a_0$$

En sortie de boucle, on on a donc  $bq + a = a_0$  avec  $a < b$  (condition de sortie), puisqu'on a déjà prouvé par ailleurs que  $a \geq 0$ , cela prouve que  $q$  est le quotient dans la division euclidienne de  $a_0$  par  $b$ . Donc la fonction est correcte.

### Principe

La preuve de correction d'une fonction récursive peut s'obtenir par récurrence :

### Principe

La preuve de correction d'une fonction récursive peut s'obtenir par récurrence :

- (initialisation) on vérifie que la fonction est correcte pour le cas de base

### Principe

La preuve de correction d'une fonction récursive peut s'obtenir par récurrence :

- (initialisation) on vérifie que la fonction est correcte pour le cas de base
- (hérédité) on prouve que si la fonction est correcte à un rang  $n \in \mathbb{N}$  alors elle l'est aussi au rang  $n + 1$ .

# C16 Un peu de Python

## 12. Rappels d'algorithmique : Correction

### Principe

La preuve de correction d'une fonction récursive peut s'obtenir par récurrence :

- (initialisation) on vérifie que la fonction est correcte pour le cas de base
- (hérédité) on prouve que si la fonction est correcte à un rang  $n \in \mathbb{N}$  alors elle l'est aussi au rang  $n + 1$ .

### Exemple

```
1  # Renvoie la factorielle de n (n>0)
2  def factorielle(n):
3      if n==0:
4          return 1
5      return n * factorielle(n-1)
```

# C16 Un peu de Python

## 12. Rappels d'algorithmique : Correction

### Principe

La preuve de correction d'une fonction récursive peut s'obtenir par récurrence :

- (initialisation) on vérifie que la fonction est correcte pour le cas de base
- (hérédité) on prouve que si la fonction est correcte à un rang  $n \in \mathbb{N}$  alors elle l'est aussi au rang  $n + 1$ .

### Exemple

```
1  # Renvoie la factorielle de n (n>0)
2  def factorielle(n):
3      if n==0:
4          return 1
5      return n * factorielle(n-1)
```

- est correcte pour  $n = 0$  puisqu'elle renvoie 1 et que  $0! = 1$ .



# C16 Un peu de Python

## 12. Rappels d'algorithmique : Correction

### Principe

La preuve de correction d'une fonction récursive peut s'obtenir par récurrence :

- (initialisation) on vérifie que la fonction est correcte pour le cas de base
- (hérédité) on prouve que si la fonction est correcte à un rang  $n \in \mathbb{N}$  alors elle l'est aussi au rang  $n + 1$ .

### Exemple

```
1  # Renvoie la factorielle de n (n>0)
2  def factorielle(n):
3      if n==0:
4          return 1
5      return n * factorielle(n-1)
```

- est correcte pour  $n = 0$  puisqu'elle renvoie 1 et que  $0! = 1$ .
- si elle est correcte au rang  $n$ , alors  $\text{fact}(n) = n!$ . Et comme,  $\text{fact}(n+1) = (n+1) * \text{fact}(n)$ , on en déduit  $\text{fact}(n+1) = (n+1)!$ .

# C16 Un peu de Python

## 12. Rappels d'algorithmique : Correction

### Exemple 2 : duplication des éléments d'une liste

On considère la fonction suivante qui renvoie la liste où chaque élément est dupliqué.

### Exemple 2 : duplication des éléments d'une liste

On considère la fonction suivante qui renvoie la liste où chaque élément est dupliqué.

```
1  # Renvoie la liste en dupliquant chaque élément
2  def duplique(liste):
3      if liste==[]:
4          return []
5      return [liste[0],liste[0]] + duplique(liste[1:])
```

# C16 Un peu de Python

## 12. Rappels d'algorithmique : Correction

### Exemple 2 : duplication des éléments d'une liste

On considère la fonction suivante qui renvoie la liste où chaque élément est dupliqué.

```
1  # Renvoie la liste en dupliquant chaque élément
2  def duplique(liste):
3      if liste==[]:
4          return []
5      return [liste[0],liste[0]] + duplique(liste[1:])
```

Pour prouver que cette fonction est correcte on peut faire le raisonnement inductif suivant :

# C16 Un peu de Python

## 12. Rappels d'algorithmique : Correction

### Exemple 2 : duplication des éléments d'une liste

On considère la fonction suivante qui renvoie la liste où chaque élément est dupliqué.

```
1  # Renvoie la liste en dupliquant chaque élément
2  def duplique(liste):
3      if liste==[]:
4          return []
5      return [liste[0],liste[0]] + duplique(liste[1:])
```

Pour prouver que cette fonction est correcte on peut faire le raisonnement inductif suivant : On note  $P(n)$ , la propriété : « pour une liste de taille  $n$ , `duplique` renvoie la liste (de taille  $2n$ ) où chaque élément est dupliqué. ». Alors :

# C16 Un peu de Python

## 12. Rappels d'algorithmique : Correction

### Exemple 2 : duplication des éléments d'une liste

On considère la fonction suivante qui renvoie la liste où chaque élément est dupliqué.

```
1  # Renvoie la liste en dupliquant chaque élément
2  def duplique(liste):
3      if liste==[]:
4          return []
5      return [liste[0],liste[0]] + duplique(liste[1:])
```

Pour prouver que cette fonction est correcte on peut faire le raisonnement inductif suivant : On note  $P(n)$ , la propriété : « pour une liste de taille  $n$ , `duplique` renvoie la liste (de taille  $2n$ ) où chaque élément est dupliqué. ». Alors :

- $P(0)$  est vérifiée d'après le cas de base

### Exemple 2 : duplication des éléments d'une liste

On considère la fonction suivante qui renvoie la liste où chaque élément est dupliqué.

```
1  # Renvoie la liste en dupliquant chaque élément
2  def duplique(liste):
3      if liste==[]:
4          return []
5      return [liste[0],liste[0]] + duplique(liste[1:])
```

Pour prouver que cette fonction est correcte on peut faire le raisonnement inductif suivant : On note  $P(n)$ , la propriété : « pour une liste de taille  $n$ , `duplique` renvoie la liste (de taille  $2n$ ) où chaque élément est dupliqué. ». Alors :

- $P(0)$  est vérifiée d'après le cas de base
- On suppose  $P(n)$  vérifié au rang  $n$ , et on considère une liste  $L$  de taille  $n + 1$ , alors, comme  $L[1:]$  est de taille  $n$ , on lui applique l'hypothèse de récurrence et `duplique(L[1:])` renvoie bien la liste avec chaque élément dupliqué.

### Exemple 2 : duplication des éléments d'une liste

On considère la fonction suivante qui renvoie la liste où chaque élément est dupliqué.

```
1  # Renvoie la liste en dupliquant chaque élément
2  def duplique(liste):
3      if liste==[]:
4          return []
5      return [liste[0],liste[0]] + duplique(liste[1:])
```

Pour prouver que cette fonction est correcte on peut faire le raisonnement inductif suivant : On note  $P(n)$ , la propriété : « pour une liste de taille  $n$ , `duplique` renvoie la liste (de taille  $2n$ ) où chaque élément est dupliqué. ». Alors :

- $P(0)$  est vérifiée d'après le cas de base
- On suppose  $P(n)$  vérifié au rang  $n$ , et on considère une liste  $L$  de taille  $n + 1$ , alors, comme  $L[1:]$  est de taille  $n$ , on lui applique l'hypothèse de récurrence et `duplique(L[1:])` renvoie bien la liste avec chaque élément dupliqué. La formule de récursivité permet alors de conclure que  $P(n + 1)$  est vérifiée puisqu'on renvoie le premier élément en double suivie de `duplique(L[1:])`.



### Définition

La **complexité** d'un algorithme est une mesure de son efficacité.

### Définition

La **complexité** d'un algorithme est une mesure de son efficacité. On parle notamment de :

### Définition

La **complexité** d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations nécessaire à l'exécution d'un algorithme.

### Définition

La **complexité** d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations nécessaire à l'exécution d'un algorithme.
- Complexité en mémoire : l'occupation mémoire en fonction de la taille des données.

### Définition

La **complexité** d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations nécessaire à l'exécution d'un algorithme.
- Complexité en mémoire : l'occupation mémoire en fonction de la taille des données.

Ces deux éléments varient en fonction de la taille et de la nature des données.

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Exemple : recherche simple dans un tableau

On considère la fonction ci dessous qui recherche si l'`elt` est ou non dans `liste`

```
1  # Renvoie true si elt est dans liste, et false sinon
2  def est_dans(elt, liste):
3      for x in liste:
4          if x == elt:
5              return True
6      return False
```

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Exemple : recherche simple dans un tableau

On considère la fonction ci dessous qui recherche si l'`elt` est ou non dans `liste`

```
1  # Renvoie true si elt est dans liste, et false sinon
2  def est_dans(elt, liste):
3      for x in liste:
4          if x == elt:
5              return True
6      return False
```

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Exemple : recherche simple dans un tableau

On considère la fonction ci dessous qui recherche si l'`elt` est ou non dans `liste`

```
1  # Renvoie true si elt est dans liste, et false sinon
2  def est_dans(elt, liste):
3      for x in liste:
4          if x == elt:
5              return True
6      return False
```

- 1 En donnant un exemple, montrer que cette fonction peut renvoyer le résultat en effectuant une seule comparaison.



# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Exemple : recherche simple dans un tableau

On considère la fonction ci dessous qui recherche si l'`elt` est ou non dans `liste`

```
1  # Renvoie true si elt est dans liste, et false sinon
2  def est_dans(elt, liste):
3      for x in liste:
4          if x == elt:
5              return True
6      return False
```

- 1 En donnant un exemple, montrer que cette fonction peut renvoyer le résultat en effectuant une seule comparaison.
- 2 En donnant un exemple, montrer que cette fonction peut renvoyer le résultat en effectuant  $n$  comparaison.

### Exemple : recherche simple dans un tableau

On considère la fonction ci dessous qui recherche si l'**elt** est ou non dans **liste**

```
1  # Renvoie true si elt est dans liste, et false sinon
2  def est_dans(elt, liste):
3      for x in liste:
4          if x == elt:
5              return True
6      return False
```

- 1 En donnant un exemple, montrer que cette fonction peut renvoyer le résultat en effectuant une seule comparaison.
- 2 En donnant un exemple, montrer que cette fonction peut renvoyer le résultat en effectuant  $n$  comparaison.
- 3 On suppose à présent qu'on cherche un élément  $a$  qui se trouve à un seul exemplaire dans le tableau et que les positions sont équiprobables. c'est-à-dire que pour tout  $i \in \llbracket 0; n-1 \rrbracket$   $a$  se trouve à l'indice  $i$  avec la probabilité  $\frac{1}{n}$ . Quel sera le nombre *moyen* de comparaison à effectuer avec de renvoyer le résultat ?

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

Exemple : recherche simple dans un tableau

### Exemple : recherche simple dans un tableau

- 1 Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.

### Exemple : recherche simple dans un tableau

- 1 Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
- 2 Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue  $n$  comparaison.

### Exemple : recherche simple dans un tableau

- 1 Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
- 2 Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue  $n$  comparaison.
- 3 on note  $X$  le nombre de comparaisons avant de trouver  $a$ , alors  $p(X = k) = \frac{1}{n}$ .  
Donc,

### Exemple : recherche simple dans un tableau

- 1 Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
- 2 Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue  $n$  comparaison.
- 3 on note  $X$  le nombre de comparaisons avant de trouver  $a$ , alors  $p(X = k) = \frac{1}{n}$ .

Donc,

$$E(X) = \sum_{k=1}^n k \frac{1}{n}$$

### Exemple : recherche simple dans un tableau

- ❶ Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
- ❷ Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue  $n$  comparaison.
- ❸ on note  $X$  le nombre de comparaisons avant de trouver  $a$ , alors  $p(X = k) = \frac{1}{n}$ .

Donc,

$$E(X) = \sum_{k=1}^n k \frac{1}{n}$$
$$E(X) = \frac{n+1}{2}$$



### Exemple : recherche simple dans un tableau

- 1 Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
- 2 Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue  $n$  comparaison.
- 3 on note  $X$  le nombre de comparaisons avant de trouver  $a$ , alors  $p(X = k) = \frac{1}{n}$ .

Donc,

$$E(X) = \sum_{k=1}^n k \frac{1}{n}$$
$$E(X) = \frac{n+1}{2}$$

Le nombre de comparaisons varie donc avec les données du problème.

### Types de complexité

On appelle :

### Remarque

Dans un calcul de complexité on ne fera pas de différences entre les différentes opérations (additions, soustractions, divisions, ...) ou tests (`if`) ou autres instructions (affectation, `return`) et on considère que toutes ont le même coût.

### Types de complexité

On appelle :

- **complexité dans le meilleur cas**, le nombre minimal d'opérations effectuées par un algorithme sur une entrée de taille  $n$ .

### Remarque

Dans un calcul de complexité on ne fera pas de différences entre les différentes opérations (additions, soustractions, divisions, ...) ou tests (**if**) ou autres instructions (affectation, **return**) et on considère que toutes ont le même coût.

### Types de complexité

On appelle :

- **complexité dans le meilleur cas**, le nombre minimal d'opérations effectuées par un algorithme sur une entrée de taille  $n$ .
- **complexité dans le pire cas**, le nombre maximal d'opérations effectuées par un algorithme sur une entrée de taille  $n$ .

### Remarque

Dans un calcul de complexité on ne fera pas de différences entre les différentes opérations (additions, soustractions, divisions, ...) ou tests (**if**) ou autres instructions (affectation, **return**) et on considère que toutes ont le même coût.

### Types de complexité

On appelle :

- **complexité dans le meilleur cas**, le nombre minimal d'opérations effectuées par un algorithme sur une entrée de taille  $n$ .
- **complexité dans le pire cas**, le nombre maximal d'opérations effectuées par un algorithme sur une entrée de taille  $n$ .
- **complexité en moyenne**, le nombre moyen d'opérations effectuées par un algorithme sur un ensemble d'entrées de taille  $n$ .

### Remarque

Dans un calcul de complexité on ne fera pas de différences entre les différentes opérations (additions, soustractions, divisions, ...) ou tests (**if**) ou autres instructions (affectation, **return**) et on considère que toutes ont le même coût.

### Types de complexité

On appelle :

- **complexité dans le meilleur cas**, le nombre minimal d'opérations effectuées par un algorithme sur une entrée de taille  $n$ .
- **complexité dans le pire cas**, le nombre maximal d'opérations effectuées par un algorithme sur une entrée de taille  $n$ .
- **complexité en moyenne**, le nombre moyen d'opérations effectuées par un algorithme sur un ensemble d'entrées de taille  $n$ .

En général, on s'intéresse à la **complexité dans le pire cas**, car on cherche à **majorer** le nombre d'opérations effectués par l'algorithme.

### Remarque

Dans un calcul de complexité on ne fera pas de différences entre les différentes opérations (additions, soustractions, divisions, ...) ou tests (**if**) ou autres instructions (affectation, **return**) et on considère que toutes ont le même coût.

### Calcul de complexité

- Donner la complexité d'un algorithme qui effectue  $C(n)$  opérations pour une entrée de taille de  $n$  c'est trouver un majorant **asymptotique** de  $C(n)$ .  
c'est-à-dire qu'on cherche à trouver une fonction qui majore la « vitesse de croissance » de  $C$ .

### Calcul de complexité

- Donner la complexité d'un algorithme qui effectue  $C(n)$  opérations pour une entrée de taille de  $n$  c'est trouver un majorant **asymptotique** de  $C(n)$ .  
c'est-à-dire qu'on cherche à trouver une fonction qui majore la « vitesse de croissance » de  $C$ .
- L'outil mathématique associé est la notion de **domination** d'une suite :  
On dit qu'une suite  $(u_n)_{n \in \mathbb{N}}$  est dominé par une suite  $(v_n)_{n \in \mathbb{N}}$  lorsqu'il existe un entier  $K > 0$  et un rang  $N \in \mathbb{N}$  tel que :



### Calcul de complexité

- Donner la complexité d'un algorithme qui effectue  $C(n)$  opérations pour une entrée de taille de  $n$  c'est trouver un majorant **asymptotique** de  $C(n)$ .  
c'est-à-dire qu'on cherche à trouver une fonction qui majore la « vitesse de croissance » de  $C$ .
- L'outil mathématique associé est la notion de **domination** d'une suite :  
On dit qu'une suite  $(u_n)_{n \in \mathbb{N}}$  est dominé par une suite  $(v_n)_{n \in \mathbb{N}}$  lorsqu'il existe un entier  $K > 0$  et un rang  $N \in \mathbb{N}$  tel que :  
 $\forall n \in \mathbb{N}, n > N, \text{ on a } |u_n| \leq k|v_n|.$

### Calcul de complexité

- Donner la complexité d'un algorithme qui effectue  $C(n)$  opérations pour une entrée de taille de  $n$  c'est trouver un majorant **asymptotique** de  $C(n)$ .  
c'est-à-dire qu'on cherche à trouver une fonction qui majore la « vitesse de croissance » de  $C$ .
- L'outil mathématique associé est la notion de **domination** d'une suite :  
On dit qu'une suite  $(u_n)_{n \in \mathbb{N}}$  est dominé par une suite  $(v_n)_{n \in \mathbb{N}}$  lorsqu'il existe un entier  $K > 0$  et un rang  $N \in \mathbb{N}$  tel que :  
 $\forall n \in \mathbb{N}, n > N, \text{ on a } |u_n| \leq k|v_n|.$   
On note alors  $u = O(v)$  et on dit que  $u$  est un grand  $O$  de  $v$ .

### Calcul de complexité

- Donner la complexité d'un algorithme qui effectue  $C(n)$  opérations pour une entrée de taille de  $n$  c'est trouver un majorant **asymptotique** de  $C(n)$ .  
c'est-à-dire qu'on cherche à trouver une fonction qui majore la « vitesse de croissance » de  $C$ .
- L'outil mathématique associé est la notion de **domination** d'une suite :  
On dit qu'une suite  $(u_n)_{n \in \mathbb{N}}$  est dominé par une suite  $(v_n)_{n \in \mathbb{N}}$  lorsqu'il existe un entier  $K > 0$  et un rang  $N \in \mathbb{N}$  tel que :  
 $\forall n \in \mathbb{N}, n > N, \text{ on a } |u_n| \leq K|v_n|.$   
On note alors  $u = O(v)$  et on dit que  $u$  est un grand  $O$  de  $v$ .
- Dans le cas de suite à valeurs positives (ce qui est la cas dans les calculs de complexités), on a :

### Calcul de complexité

- Donner la complexité d'un algorithme qui effectue  $C(n)$  opérations pour une entrée de taille de  $n$  c'est trouver un majorant **asymptotique** de  $C(n)$ .  
c'est-à-dire qu'on cherche à trouver une fonction qui majore la « vitesse de croissance » de  $C$ .
- L'outil mathématique associé est la notion de **domination** d'une suite :  
On dit qu'une suite  $(u_n)_{n \in \mathbb{N}}$  est dominé par une suite  $(v_n)_{n \in \mathbb{N}}$  lorsqu'il existe un entier  $K > 0$  et un rang  $N \in \mathbb{N}$  tel que :  
 $\forall n \in \mathbb{N}, n > N, \text{ on a } |u_n| \leq k|v_n|.$   
On note alors  $u = O(v)$  et on dit que  $u$  est un grand  $O$  de  $v$ .
- Dans le cas de suite à valeurs positives (ce qui est la cas dans les calculs de complexités), on a :  
 $u = O(v)$  ssi  $\exists K \in \mathbb{N}$  tel que  $\forall n \in \mathbb{N}, u_n \leq kv_n.$

### Exemples

Calculer le nombre d'opérations  $C(n)$  des fonctions suivantes en fonction de la taille des entrées  $n$ , et donner leur complexité avec la notation  $O$ .

#### ① Fonction $f$

```
1  def f(x):  
2      return x**2 + 7*x - 1
```

### Exemples

Calculer le nombre d'opérations  $C(n)$  des fonctions suivantes en fonction de la taille des entrées  $n$ , et donner leur complexité avec la notation  $O$ .

#### 1 Fonction `f`

```
1 def f(x):  
2     return x**2 + 7*x - 1
```

#### 2 Fonction `somme_f`

```
1 def somme_f(valeurs):  
2     sf = 0  
3     for x in valeurs:  
4         sf = sf + f(x)  
5     return sf
```

### Correction

#### 1 Fonction $f$

### Correction

#### 1 Fonction $f$

$C = 5$  quelque soit l'entrée  $x$ , et donc  $C$  est un grand  $O(1)$ , c'est-à-dire que le temps d'exécution est majoré par une constante quelque soit la taille des entrées.



### Correction

#### ① Fonction `f`

$C = 5$  quelque soit l'entrée  $x$ , et donc  $C$  est un grand  $O(1)$ , c'est-à-dire que le temps d'exécution est majoré par une constante quelque soit la taille des entrées.

#### ② Fonction `somme_f`

### Correction

#### ① Fonction `f`

$C = 5$  quelque soit l'entrée  $x$ , et donc  $C$  est un grand  $O(1)$ , c'est-à-dire que le temps d'exécution est majoré par une constante quelque soit la taille des entrées.

#### ② Fonction `somme_f`

$C = 8n + 2$ , et donc  $C$  est un grand  $O(n)$ , c'est-à-dire que le temps d'exécution est majoré par une fonction linéaire quelque soit la taille des entrées

### Correction

#### ① Fonction `f`

$C = 5$  quelque soit l'entrée  $x$ , et donc  $C$  est un grand  $O(1)$ , c'est-à-dire que le temps d'exécution est majoré par une constante quelque soit la taille des entrées.

#### ② Fonction `somme_f`

$C = 8n + 2$ , et donc  $C$  est un grand  $O(n)$ , c'est-à-dire que le temps d'exécution est majoré par une fonction linéaire quelque soit la taille des entrées

### A retenir

le nombre précis d'opérations effectué par l'algorithme n'est pas pertinent. On donnera toujours la complexité sous la forme d'un grand  $O$ , c'est-à-dire d'une majoration asymptotique du coût de l'algorithme.

### Tri par sélection

- 1 Rappeler le principe de l'algorithme du tri par sélection.

### Tri par sélection

- 1 Rappeler le principe de l'algorithme du tri par sélection.
- 2 Donner une implémentation itérative en Python.

### Tri par sélection

- 1 Rappeler le principe de l'algorithme du tri par sélection.
- 2 Donner une implémentation itérative en Python.
- 3 Prouver la correction de cet algorithme.

### Tri par sélection

- 1 Rappeler le principe de l'algorithme du tri par sélection.
- 2 Donner une implémentation itérative en Python.
- 3 Prouver la correction de cet algorithme.
- 4 Donner sa complexité.

### Correction : tri par sélection

- 1 Rappeler le principe de l'algorithme du tri par sélection.



### Correction : tri par sélection

- 1 Rappeler le principe de l'algorithme du tri par sélection.

A chaque étape d'indice  $i$  (pour  $i = 0 \dots n - 1$ ), on échange l'élément d'indice  $i$  avec le minimum des éléments de la liste depuis l'indice  $i$ .

### Correction : tri par sélection

- 1 Rappeler le principe de l'algorithme du tri par sélection.

A chaque étape d'indice  $i$  (pour  $i = 0 \dots n - 1$ ), on échange l'élément d'indice  $i$  avec le minimum des éléments de la liste depuis l'indice  $i$ .

- 2 Donner une implémentation itérative en Python.

```
1  # Renvoie l'indice du plus petit élément depuis l'indice start
2  def indice_min_depuis(liste,start):
3      imin = start
4      for i in range(imin+1,len(liste)):
5          if liste[i]<liste[imin]:
6              imin = i
7      return imin
8
9  # Tri en place par sélection
10 def tri_selection(liste):
11     for i in range(0,len(liste)):
12         imin = indice_min_depuis(liste,i)
13         liste[imin], liste[i] = liste[i], liste[imin]
```

### Correction : tri par sélection

- ③ Prouver la correction de cet algorithme.

### Correction : tri par sélection

- ③ Prouver la correction de cet algorithme.

Montrons que l'invariant  $I$  « Le sous tableau formé des  $i$  premiers éléments de liste est trié et contient les  $i$  plus petits éléments de liste »

### Correction : tri par sélection

- ③ Prouver la correction de cet algorithme.

Montrons que l'invariant  $I$  « Le sous tableau formé des  $i$  premiers éléments de liste est trié et contient les  $i$  plus petits éléments de liste »

- Cette propriété est vraie à l'entrée dans la boucle (le sous tableau est alors vide)

### Correction : tri par sélection

#### ③ Prouver la correction de cet algorithme.

Montrons que l'invariant  $I$  « Le sous tableau formé des  $i$  premiers éléments de liste est trié et contient les  $i$  plus petits éléments de liste »

- Cette propriété est vraie à l'entrée dans la boucle (le sous tableau est alors vide)
- Si on la suppose vraie au rang  $i$  alors au rang  $i + 1$ , on place à l'indice  $i + 1$  le minimum des éléments d'indice  $i + 1 \dots n$ . Cet élément étant forcément supérieur à ceux d'indice  $0, \dots, i - 1$ , la propriété est vraie au rang  $i + 1$ .

### Correction : tri par sélection

- ③ Prouver la correction de cet algorithme.

Montrons que l'invariant  $I$  « Le sous tableau formé des  $i$  premiers éléments de liste est trié et contient les  $i$  plus petits éléments de liste »

- Cette propriété est vraie à l'entrée dans la boucle (le sous tableau est alors vide)
- Si on la suppose vraie au rang  $i$  alors au rang  $i + 1$ , on place à l'indice  $i + 1$  le minimum des éléments d'indice  $i + 1 \dots n$ . Cet élément étant forcément supérieur à ceux d'indice  $0, \dots, i - 1$ , la propriété est vraie au rang  $i + 1$ .

- ④ Donner sa complexité.

### Correction : tri par sélection

#### ③ Prouver la correction de cet algorithme.

Montrons que l'invariant  $I$  « Le sous tableau formé des  $i$  premiers éléments de liste est trié et contient les  $i$  plus petits éléments de liste »

- Cette propriété est vraie à l'entrée dans la boucle (le sous tableau est alors vide)
- Si on la suppose vraie au rang  $i$  alors au rang  $i + 1$ , on place à l'indice  $i + 1$  le minimum des éléments d'indice  $i + 1 \dots n$ . Cet élément étant forcément supérieur à ceux d'indice  $0, \dots, i - 1$ , la propriété est vraie au rang  $i + 1$ .

#### ④ Donner sa complexité.

On peut se contenter de raisonner sur le nombre de comparaison, pour chaque recherche de minimum on fait  $n - i$  comparaisons, et on doit chercher le minimum pour  $i = 0 \dots n - 1$ . Donc, en notant  $C(n)$  le nombre de comparaisons



### Correction : tri par sélection

#### ③ Prouver la correction de cet algorithme.

Montrons que l'invariant  $I$  « Le sous tableau formé des  $i$  premiers éléments de liste est trié et contient les  $i$  plus petits éléments de liste »

- Cette propriété est vraie à l'entrée dans la boucle (le sous tableau est alors vide)
- Si on la suppose vraie au rang  $i$  alors au rang  $i + 1$ , on place à l'indice  $i + 1$  le minimum des éléments d'indice  $i + 1 \dots n$ . Cet élément étant forcément supérieur à ceux d'indice  $0, \dots, i - 1$ , la propriété est vraie au rang  $i + 1$ .

#### ④ Donner sa complexité.

On peut se contenter de raisonner sur le nombre de comparaison, pour chaque recherche de minimum on fait  $n - i$  comparaisons, et on doit chercher le minimum pour  $i = 0 \dots n - 1$ . Donc, en notant  $C(n)$  le nombres de comparaisons

$$C(n) = \sum_{i=0}^{n-1} n - i$$

### Correction : tri par sélection

#### ③ Prouver la correction de cet algorithme.

Montrons que l'invariant  $I$  « Le sous tableau formé des  $i$  premiers éléments de liste est trié et contient les  $i$  plus petits éléments de liste »

- Cette propriété est vraie à l'entrée dans la boucle (le sous tableau est alors vide)
- Si on la suppose vraie au rang  $i$  alors au rang  $i + 1$ , on place à l'indice  $i + 1$  le minimum des éléments d'indice  $i + 1 \dots n$ . Cet élément étant forcément supérieur à ceux d'indice  $0, \dots, i - 1$ , la propriété est vraie au rang  $i + 1$ .

#### ④ Donner sa complexité.

On peut se contenter de raisonner sur le nombre de comparaison, pour chaque recherche de minimum on fait  $n - i$  comparaisons, et on doit chercher le minimum pour  $i = 0 \dots n - 1$ . Donc, en notant  $C(n)$  le nombres de comparaisons

$$C(n) = \sum_{i=0}^{n-1} n - i = \sum_{k=1}^n k$$

### Correction : tri par sélection

#### ③ Prouver la correction de cet algorithme.

Montrons que l'invariant  $I$  « Le sous tableau formé des  $i$  premiers éléments de liste est trié et contient les  $i$  plus petits éléments de liste »

- Cette propriété est vraie à l'entrée dans la boucle (le sous tableau est alors vide)
- Si on la suppose vraie au rang  $i$  alors au rang  $i + 1$ , on place à l'indice  $i + 1$  le minimum des éléments d'indice  $i + 1 \dots n$ . Cet élément étant forcément supérieur à ceux d'indice  $0, \dots, i - 1$ , la propriété est vraie au rang  $i + 1$ .

#### ④ Donner sa complexité.

On peut se contenter de raisonner sur le nombre de comparaison, pour chaque recherche de minimum on fait  $n - i$  comparaisons, et on doit chercher le minimum pour  $i = 0 \dots n - 1$ . Donc, en notant  $C(n)$  le nombres de comparaisons

$$C(n) = \sum_{i=0}^{n-1} n - i = \sum_{k=1}^n k$$

$$C(n) = \frac{n(n+1)}{2} \text{ et donc cet algorithme est en } O(n^2).$$

### Complexités usuelles

Complexité	Nom	Exemple

### Complexités usuelles

Complexité	Nom	Exemple
$O(1)$	Constant	Accéder à un élément d'une liste

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Complexités usuelles

Complexité	Nom	Exemple
$O(1)$	Constant	Accéder à un élément d'une liste
$O(\log(n))$	Logarithmique	Recherche dichotomique dans une liste

### Complexités usuelles

Complexité	Nom	Exemple
$O(1)$	Constant	Accéder à un élément d'une liste
$O(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$O(n)$	Linéaire	Recherche simple dans une liste

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Complexités usuelles

Complexité	Nom	Exemple
$O(1)$	Constant	Accéder à un élément d'une liste
$O(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$O(n)$	Linéaire	Recherche simple dans une liste
$O(n \log(n))$	Linéaritmique	Tri fusion



# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Complexités usuelles

Complexité	Nom	Exemple
$O(1)$	Constant	Accéder à un élément d'une liste
$O(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$O(n)$	Linéaire	Recherche simple dans une liste
$O(n \log(n))$	Linéaritmique	Tri fusion
$O(n^2)$	Quadratique	Tri par insertion d'une liste

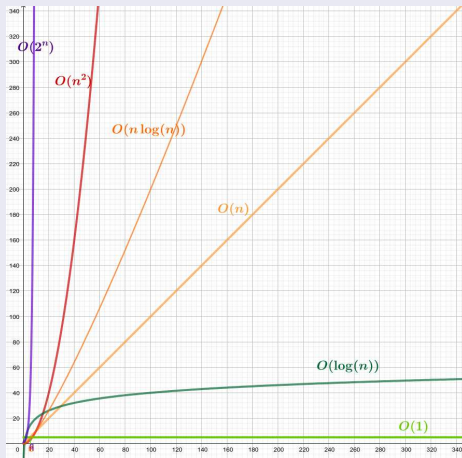
### Complexités usuelles

Complexité	Nom	Exemple
$O(1)$	Constant	Accéder à un élément d'une liste
$O(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$O(n)$	Linéaire	Recherche simple dans une liste
$O(n \log(n))$	Linéaritmique	Tri fusion
$O(n^2)$	Quadratique	Tri par insertion d'une liste
$O(2^n)$	Exponentielle	Algorithme par force brute pour le sac à dos

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Représentation graphique



# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde :

Complexité	$n = 10$	$n = 100$	$n = 1000$	$n = 10^6$	$n = 10^9$
$O(\log(n))$	✓	✓	✓	✓	✓
$O(n)$	✓	✓	✓	✓	$\simeq 10\text{s}$
$O(n) \log(n)$	✓	✓	✓	✓	$\simeq 1,5 \text{ mn}$
$O(n^2)$	✓	✓	✓	$\simeq 3 \text{ h}$	$\simeq 300 \text{ ans}$
$O(2^n)$	✓	✗	✗	✗	✗

### Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.

### Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.  
La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.

### Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.  
La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.  
 $0.015 \times 250 = 3.75$ , on peut donc prévoir un temps de calcul d'environ 3,75 secondes

### Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.  
La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.  
 $0.015 \times 250 = 3.75$ , on peut donc prévoir un temps de calcul d'environ 3,75 secondes
- Même question pour un algorithme de complexité quadratique qui traite une liste de 1 000 éléments en 0,07 secondes.



### Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.  
La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.  
 $0.015 \times 250 = 3.75$ , on peut donc prévoir un temps de calcul d'environ 3,75 secondes
- Même question pour un algorithme de complexité quadratique qui traite une liste de 1 000 éléments en 0,07 secondes.  
La taille des données a été multiplié par 250, la complexité étant quadratique le temps de calcul sera approximativement multiplié par  $250^2 = 62500$

### Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.  
La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.  
 $0.015 \times 250 = 3.75$ , on peut donc prévoir un temps de calcul d'environ 3,75 secondes
- Même question pour un algorithme de complexité quadratique qui traite une liste de 1 000 éléments en 0,07 secondes.  
La taille des données a été multiplié par 250, la complexité étant quadratique le temps de calcul sera approximativement multiplié par  $250^2 = 62500$   
 $0.07 \times 62\,500 = 4375$ , on peut donc prévoir un temps de calcul d'environ 4 375 secondes, c'est-à-dire près d'une heure et 15 minutes !

### Exemple : exponentiation rapide

- ❶ Combien faut-il faire de multiplications pour calculer  $a^{13}$  avec la fonction

```
1  # Calcule x puissance n, avec n>0
2  def puissance(x,n):
3      p = 1
4      for i in range(1,n):
5          p = p * n
6      return p
```

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Exemple : exponentiation rapide

- ❶ Combien faut-il faire de multiplications pour calculer  $a^{13}$  avec la fonction

```
1  # Calcule x puissance n, avec n>0
2  def puissance(x,n):
3      p = 1
4      for i in range(1,n):
5          p = p * n
6      return p
```

- ❷ Combien en faut-il si on procède de la façon suivante :

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Exemple : exponentiation rapide

- ❶ Combien faut-il faire de multiplications pour calculer  $a^{13}$  avec la fonction

```
1  # Calcule x puissance n, avec n>0
2  def puissance(x,n):
3      p = 1
4      for i in range(1,n):
5          p = p * n
6      return p
```

- ❷ Combien en faut-il si on procède de la façon suivante :
- Calculer  $a^6$ , l'élever au carré et le multiplier par  $a$ .

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Exemple : exponentiation rapide

❶ Combien faut-il faire de multiplications pour calculer  $a^{13}$  avec la fonction

```
1  # Calcule x puissance n, avec n>0
2  def puissance(x,n):
3      p = 1
4      for i in range(1,n):
5          p = p * n
6      return p
```

❷ Combien en faut-il si on procède de la façon suivante :

- Calculer  $a^6$ , l'élever au carré et le multiplier par  $a$ .
- Pour calculer  $a^6$ , calculer  $a^3$  et l'élever au carré.

### Exemple : exponentiation rapide

❶ Combien faut-il faire de multiplications pour calculer  $a^{13}$  avec la fonction

```
1  # Calcule x puissance n, avec n>0
2  def puissance(x,n):
3      p = 1
4      for i in range(1,n):
5          p = p * n
6      return p
```

❷ Combien en faut-il si on procède de la façon suivante :

- Calculer  $a^6$ , l'élever au carré et le multiplier par  $a$ .
- Pour calculer  $a^6$ , calculer  $a^3$  et l'élever au carré.
- Pour calculer  $a^3$ , élever  $a$  au carré et multiplier par  $a$ .

### Exemple : exponentiation rapide

- ❶ Combien faut-il faire de multiplications pour calculer  $a^{13}$  avec la fonction

```
1  # Calcule x puissance n, avec n>0
2  def puissance(x,n):
3      p = 1
4      for i in range(1,n):
5          p = p * n
6      return p
```

- ❷ Combien en faut-il si on procède de la façon suivante :

- Calculer  $a^6$ , l'élever au carré et le multiplier par  $a$ .
- Pour calculer  $a^6$ , calculer  $a^3$  et l'élever au carré.
- Pour calculer  $a^3$ , élever  $a$  au carré et multiplier par  $a$ .

- ❸ Généraliser la méthode précédente au cas d'un exposant quelconque et en déduire une relation de récurrence entre  $a^n$  et  $a^{\frac{n}{2}}$  si  $n$  est pair et  $a^{\frac{n-1}{2}}$  sinon.



### Exemple : exponentiation rapide

- ❶ Combien faut-il faire de multiplications pour calculer  $a^{13}$  avec la fonction

```
1  # Calcule x puissance n, avec n>0
2  def puissance(x,n):
3      p = 1
4      for i in range(1,n):
5          p = p * n
6      return p
```

- ❷ Combien en faut-il si on procède de la façon suivante :

- Calculer  $a^6$ , l'élever au carré et le multiplier par  $a$ .
- Pour calculer  $a^6$ , calculer  $a^3$  et l'élever au carré.
- Pour calculer  $a^3$ , élever  $a$  au carré et multiplier par  $a$ .

- ❸ Généraliser la méthode précédente au cas d'un exposant quelconque et en déduire une relation de récurrence entre  $a^n$  et  $a^{\frac{n}{2}}$  si  $n$  est pair et  $a^{\frac{n-1}{2}}$  sinon.

- ❹ Proposer une implémentation récursive de ce nouvel algorithme.

### Exemple : exponentiation rapide

- ❶ Combien faut-il faire de multiplications pour calculer  $a^{13}$  avec la fonction

```
1  # Calcule x puissance n, avec n>0
2  def puissance(x,n):
3      p = 1
4      for i in range(1,n):
5          p = p * n
6      return p
```

- ❷ Combien en faut-il si on procède de la façon suivante :

- Calculer  $a^6$ , l'élever au carré et le multiplier par  $a$ .
- Pour calculer  $a^6$ , calculer  $a^3$  et l'élever au carré.
- Pour calculer  $a^3$ , élever  $a$  au carré et multiplier par  $a$ .

- ❸ Généraliser la méthode précédente au cas d'un exposant quelconque et en déduire une relation de récurrence entre  $a^n$  et  $a^{\frac{n}{2}}$  si  $n$  est pair et  $a^{\frac{n-1}{2}}$  sinon.
- ❹ Proposer une implémentation récursive de ce nouvel algorithme.
- ❺ Déterminer la complexité de chacun des deux algorithmes, conclure.

### Exemple : exponentiation rapide

- ① Il faut faire 13 multiplications, puisque  $a^{13}$  est calculé avec :

$$a^{13} = 1 \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a$$

### Exemple : exponentiation rapide

- ❶ Il faut faire 13 multiplications, puisque  $a^{13}$  est calculé avec :

$$a^{13} = 1 \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a$$

- ❷ Dans ce cas, il ne faut que 5 multiplications en effet, on calcul  $a^{13}$  avec :

$$a^{13} = \left( (a^2 \times a)^2 \right)^2 \times a$$

# C16 Un peu de Python

## 13. Rappels d'algorithmique : Complexité

### Exemple : exponentiation rapide

- ❶ Il faut faire 13 multiplications, puisque  $a^{13}$  est calculé avec :

$$a^{13} = 1 \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a$$

- ❷ Dans ce cas, il ne faut que 5 multiplications en effet, on calcul  $a^{13}$  avec :

$$a^{13} = \left( (a^2 \times a)^2 \right)^2 \times a$$

- ❸ 
$$\begin{cases} a^n = \left( a^{\frac{n}{2}} \right)^2, & \text{si } n \text{ est paire} \\ a^n = \left( a^{\frac{n-1}{2}} \right)^2 \times a, & \text{sinon} \end{cases}$$

### Exponentiation rapide

#### 4 Implémentation en Python :

```
1 def puissance_rapide(x,n):
2     if n==0:
3         return 1
4     p = puissance_rapide(x,n//2)
5     if n%2==0:
6         return p*p
7     else:
8         return p*p*x
```

### Exponentiation rapide

#### 4 Implémentation en Python :

```
1 def puissance_rapide(x,n):
2     if n==0:
3         return 1
4     p = puissance_rapide(x,n//2)
5     if n%2==0:
6         return p*p
7     else:
8         return p*p*x
```

- 5 Le premier algorithme a une complexité linéaire, celui-ci a une complexité logarithmique. En effet, l'exposant est divisé par 2 à chaque appel récursif