Bonnes pratiques de programmation

- Les *commentaires* s'écrivent en faisant commencer la ligne par le caractères #

Bonnes pratiques de programmation

- Les *commentaires* s'écrivent en faisant commencer la ligne par le caractères #
- Les noms de variables et de fonction doivent être explicites.

Bonnes pratiques de programmation

- Les *commentaires* s'écrivent en faisant commencer la ligne par le caractères `#`
- Les noms de variables et de fonction doivent être explicites.
- L'instruction `assert <condition>` permet de vérifier que `<condition>` est vérifiée avant de continuer l'exécution du programme. On peut ainsi tester des fonctions ou vérifier des *préconditions* sur des arguments.

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

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
- Cet import peut se faire en donnant un *alias* : `import <lib> as <alias>`

Exemple

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
- Cet import peut se faire en donnant un *alias* : `import <lib> as <alias>`
- Pour importer simple la fonction `<fonc>` de la librairie `<lib>`, on utilise `from <lib> import <fonc>`. Le nom de la fonction est alors utilisé directement.

Exemple

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
- Cet import peut se faire en donnant un *alias* : import <lib> as <alias>
- Pour importer simple la fonction <fonc> de la librairie <lib>, on utilise from <lib> import <fonc>. Le nom de la fonction est alors utilisé directement.

Exemple

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

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
- Cet import peut se faire en donnant un *alias* : import <lib> as <alias>
- Pour importer simple la fonction <fonc> de la librairie <lib>, on utilise from <lib> import <fonc>. Le nom de la fonction est alors utilisé directement.

Exemple

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

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

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `//`, `%`	Entiers signés ou non signés. Taille dynamique limitée par la mémoire

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `//`, `%`	Entiers signés ou non signés. Taille dynamique limitée par la mémoire
`float`	`+`, `-`, `*`, `/`, `**`	Représentation des nombres en virgule flottante (norme ieee754 : mantisse sur 53 bits, exposant sur 11 bits). Fonctions élémentaires dans `math.h`

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `//`, `%`	Entiers signés ou non signés. Taille dynamique limitée par la mémoire
`float`	`+`, `-`, `*`, `/`	Représentation des nombres en virgule flottante (norme ieee754 : mantisse sur 53 bits, exposant sur 11 bits). Fonctions élémentaires dans `math`
`bool`	`or` `and`, `not`, `all`, `any`	Evaluations paresseuses des expressions.

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 :

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

- qui renvoie une valeur :

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

Instructions conditionnelles

- Sans clause `else`

```
if <condition>:
<instructions>
```

Exécute les `<instructions>` si la `condition` est vérifiée.

Instructions conditionnelles

- Sans clause `else`

```
if <condition>:
<instructions>
```

Exécute les `<instructions>` si la `condition` est vérifiée.

- Avec clause `else`

```
if <condition>:
<instructions1>
else:
<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 :

```
while <condition>:
    <instruction>
```

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

Boucles `while`

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

```
while <condition>:
<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 :

```
while <condition>:
    <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`

Boucles `for`

- Les instructions :

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

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

Boucles for

- Les instructions :

```
for <variable> in range(<entier>):
    <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

- Les instructions :

```
for <variable> in range(<entier>):
    <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`

- Les instructions :

```
for <variable> in range(<entier>):
    <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.

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.

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

Exemple 2

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

Exemple 2

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

```python
def serie_harmonique(n):
    somme = 0
    for i in range(1,n+1):
        somme = somme + 1/i
    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 :

```
def pgcd(a,b):
    while b!=0:
        a,b = b, a%b
    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 :

```python
def pgcd(a,b):
    while b!=0:
        a,b = b, a%b
    return a
```

- Version 2 :

```python
def pgcd(a,b):
    if b == 0:
        return a
    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

```python
def pgcd(a,b):
    while b!=0:
        a,b = b, a%b
    return a
```

- Version 2 : récursive

```python
def pgcd(a,b):
    if b == 0:
        return a
    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 mêmte type).

Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du mêmte 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 mêmte 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 mêmte 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 mêmte 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 mêmte 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 mêmte 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 mêmte 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 élement 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 élement 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

Les listes de Python sont mutables, c'est à dire que les modifications faites sur une liste passée en argument à une fonction sont effectivement réalisées sur la liste. Ce n'est pas le cas sur les arguments de type entier ou flottants.

Exemples

- Ce programme affiche 42 car `n` étant de type entier l'opération effectuée sur `n` ne se répercute pas sur l'argument de la fonction.

```python
def carre(n):
    n = n * n

n = 42
carre(n)
print(n)
```

Exemples

- Ce programme affiche 42 car `n` étant de type entier l'opération effectuée sur `n` ne se répercute pas sur l'argument de la fonction.

```python
def carre(n):
n = n * n

n = 42
carre(n)
print(n)
```

- Ce programme modifie la liste passée en argument et donc affichera `[5,7]`

```python
def ajoute(liste,valeur):
liste.append(valeur)

liste = [5]
liste.ajoute(7)
print(liste)
```

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'élement 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'élement 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'élement 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'élement 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'élement 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'élement 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 huits premières puissances de 2

Création de listes

On peut créer des listes de diverses façons en Python :

- Par ajout succesif d'élement 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 huits 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'élement 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 huits 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 commmence 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 commmence 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 commmence 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 commmence 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]`.

Tuples

- Les tuples sont le pendant non mutables des listes. Ils se notent entre parenthèses (et), les éléments sont aussi séparés par des virgules.

Exemple

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

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.

Exemple

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

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

```python
anniv = (31,"Janvier",1956)
print("Mois de naissance = ",anniv[1])
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 chaines de caractères.

Chaines de caractères

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

Chaines de caractères

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

```python
for lettre in mot:
    print(lettre)
```

Chaines de caractères

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

```python
for lettre in mot:
print(lettre)
```

- Comme les tuples, les chaines 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 chaines de caractères.
  Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre `"n"`
- Le parcours par élément peut aussi se faire sur une chaine de caractères.
  Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

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

- Comme les tuples, les chaines de caractères sont non mutables.
- ! Les variables lues au clavier (instruction `input`) ou issus de la lecture d'un fichier sont des chaines 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 chaines de caractères.
  Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre `"n"`
- Le parcours par élément peut aussi se faire sur une chaine de caractères.
  Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

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

- Comme les tuples, les chaines de caractères sont non mutables.
- ! Les variables lues au clavier (instruction `input`) ou issus de la lecture d'un fichier sont des chaines 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 chaines en utilisant le séparateur donné en argument.

Exemple

Ecrire une fonction `check_date` qui prend en argument une chaine de caractères et renvoie `True` si cette chaine 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.

Exemple

Ecrire une fonction `check_date` qui prend en argument une chaine de caractères et renvoie `True` si cette chaine 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.

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

Parcours d'une liste

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

Eléments ▶	`L[0]`	`L[1]`	`L[2]`	`L[3]`	...
	↑	↑	↑	↑	
Indices ▶	0	1	2	3	...

On peut parcourir cette liste :

Parcours d'une liste

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

Eléments ▶	L[0]	L[1]	L[2]	L[3]	...
	↑	↑	↑	↑	
Indices ▶	0	1	2	3	...

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 :

Eléments ▶	`L[0]`	`L[1]`	`L[2]`	`L[3]`	...
	↑	↑	↑	↑	
Indices ▶	0	1	2	3	...

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).

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.

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 :

```python
def est_dans(n,l):
    for x in l:
        if x==n:
            return True
    return False
```

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 :

```python
def est_dans(n,l):
    for x in l:
        if x==n:
            return True
    return False
```

- Parcours par indice :

```python
def est_dans_ind(n,l):
    for i in range(len(l)):
        if l[i]==n:
            return True
    return False
```

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

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

```python
def max_liste(l):
    # la liste doit être non vide
    assert len(l)!=0
    current_max = l[0]
    for elt in l:
        if elt>current_max:
            current_max = elt
    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 crée à l'aide de l'instruction `open` :

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`

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`

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`

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

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

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 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.

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

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.

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

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.

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

Algorithmique

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

Algorithmique

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

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 suivantes :

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 suivantes :

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 : evolution 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 suivantes :

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 : evolution 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 :

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.

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 :

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

Exemple 1

On considère la fonction ci-dessous :

```python
# Renvoie le quotient dans la division euclidienne de a par b avec a et b deux entiers naturels et b non nul
def quotient(a, b):
    assert (a >= 0 and b > 0)
    q = 0
    while (a - b >= 0):
        a = a - b
        q = q + 1
    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 :

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

Exemple 2

On considère la fonction ci-dessous :

```python
# Renvoie True si elt est dans liste et False sinon
def est_dans(elt, liste):
    if liste == []:
        return False
    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.

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 :

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

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 :

```
def syracuse(n):
    while n != 1:
        if n % 2 == 0:
            n = n//2
        else:
            n = 3*n+1
    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 :

```python
# Renvoie le nombre d'occurences d'elt dans liste
def nb_occ(elt, liste):
    cpt = 0
    for x in liste:
        if x == elt:
            cpt = cpt + 1
    return cpt
```

Exemple 1

On considère la fonction ci-dessous :

```python
# Renvoie le nombre d'occurences d'elt dans liste
def nb_occ(elt, liste):
    cpt = 0
    for x in liste:
        if x == elt:
            cpt = cpt + 1
    return cpt
```

En trouvant un invariant de boucle, montrer qu'à la sortie de la boucle, la variable `cpt` contient le nombre de fois où `elt` apparaît dans `liste`

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `liste` et k le nombre de tours de boucle

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `liste` et k le nombre de tours de boucle
Montrons que la propriété :
« `cpt` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `liste` » est un invariant de boucle.

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `liste` et k le nombre de tours de boucle
Montrons que la propriété :
« `cpt` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `liste` » est un invariant de boucle.

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `liste` et k le nombre de tours de boucle
Montrons que la propriété :
« `cpt` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `liste` » est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `cpt` vaut 0.

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `liste` et k le nombre de tours de boucle
Montrons que la propriété :
« `cpt` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `liste` » est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `cpt` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt`$= e_{k+1}$

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `liste` et k le nombre de tours de boucle
Montrons que la propriété :
« `cpt` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `liste` » est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `cpt` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt`$= e_{k+1}$

Cette propriété est donc bien un invariant de boucle.

Correction de l'exemple 1

On note $\{e_1, e_2, \ldots, e_n\}$ les élements de `liste` et k le nombre de tours de boucle
Montrons que la propriété :
« `cpt` *contient le nombre de de fois où* `elt` *apparaît dans les* k *premiers éléments de* `liste` » est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `cpt` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt`$= e_{k+1}$

Cette propriété est donc bien un invariant de boucle. L'invariant de boucle reste vraie en sortie de boucle ce qui prouve que l'algorithme est correct.

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir :

- Directement à partir des identités mathématiques issus du code de la fonction.

Exemple 2 : factoriel récursif

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir :

- Directement à partir des identités mathématiques issus du code de la fonction.
- Par un raisonnement inductif, c'est à dire similaire à une récurrence.

Exemple 2 : factoriel récursif

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir :

- Directement à partir des identités mathématiques issus du code de la fonction.
- Par un raisonnement inductif, c'est à dire similaire à une récurrence.

Exemple 2 : factoriel récursif

```python
# Calcule n!
def fact(n):
    if n==0:
        return 1
    return n * fact(n-1)
```

Correction d'un algorithme récursif

Dans le cas d'un algorithme récursif, la preuve de la correction peut s'obtenir :

- Directement à partir des identités mathématiques issus du code de la fonction.
- Par un raisonnement inductif, c'est à dire similaire à une récurrence.

Exemple 2 : factoriel récursif

```python
# Calcule n!
def fact(n):
    if n==0:
        return 1
    return n * fact(n-1)
```

Les identités `fact 0 = 1` et `fact n = n * fact (n-1)` si $n > 0$, correspondent bien à la définition mathématique de la factorielle c'est à dire $0! = 1$ et pour tout $n \in \mathbb{N}^*$, $n! = n \times (n-1)!$, donc cette fonction est correcte

Exemple 2 : duplication des éléménts d'un liste

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

Exemple 2 : duplication des éléménts d'un liste

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

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

Exemple 2 : duplication des éléménts d'un liste

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

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

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

Exemple 2 : duplication des éléménts d'un liste

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

```python
# Renvoie la liste en dupliquant chaque élément
def duplique(liste):
    if liste==[]:
        return []
    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 :

Exemple 2 : duplication des éléménts d'un liste

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

```python
# Renvoie la liste en dupliquant chaque élément
def duplique(liste):
    if liste==[]:
        return []
    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 éléménts d'un liste

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

```python
# Renvoie la liste en dupliquant chaque élément
def duplique(liste):
    if liste==[]:
        return []
    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 éléménts d'un liste

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

```python
# Renvoie la liste en dupliquant chaque élément
def duplique(liste):
    if liste==[]:
        return []
    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.

Exemple : recherche simple dans un tableau

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

```python
# Renvoie true si elt est dans liste, et false sinon
def est_dans(elt, liste):
    for x in liste:
        if x == elt:
            return True
    return False

```

Exemple : recherche simple dans un tableau

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

```python
# Renvoie true si elt est dans liste, et false sinon
def est_dans(elt, liste):
    for x in liste:
        if x == elt:
            return True
    return False

```

Exemple : recherche simple dans un tableau

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

```python
# Renvoie true si elt est dans liste, et false sinon
def est_dans(elt, liste):
    for x in liste:
        if x == elt:
            return True
    return False

```

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

Exemple : recherche simple dans un tableau

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

```python
# Renvoie true si elt est dans liste, et false sinon
def est_dans(elt, liste):
    for x in liste:
        if x == elt:
            return True
    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`

```python
# Renvoie true si elt est dans liste, et false sinon
def est_dans(elt, liste):
    for x in liste:
        if x == elt:
            return True
    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 $i \in [\![0; n-1]\!]$ a se trouve à l'indice i avec la probabilité $\dfrac{1}{n}$. Quel sera le nombre *moyen* de comparaison à effectuer avec de renvoyer le résultat ?

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 $p(X = k) = \dfrac{1}{n}$.
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 $p(X=k)=\dfrac{1}{n}$. Donc,

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

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 $p(X=k)=\dfrac{1}{n}$.
   Donc,

$$\begin{aligned} E(X) &= \sum_{k=1}^{n} k\,\frac{1}{n} \\ E(X) &= \frac{n+1}{2} \end{aligned}$$

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 $p(X=k)=\dfrac{1}{n}$.
   Donc,
$$\begin{aligned} E(X) &= \sum_{k=1}^{n} k\,\frac{1}{n} \\ E(X) &= \frac{n+1}{2} \end{aligned}$$

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

Types de complexité

On appelle :

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 entres 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'operations effectuées par un algorithme sur une entrée 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 entres 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'operations effectuées par un algorithme sur une entrée de taille n.
- complexité dans le pire cas, le nombre maximal d'operations effectuées par un algorithme sur une entrée 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 entres 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'operations effectuées par un algorithme sur une entrée de taille n.
- complexité dans le pire cas, le nombre maximal d'operations effectuées par un algorithme sur une entrée de taille n.
- complexité en moyenne, le nombre moyen d'operations 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 entres 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 :
  $\forall n \in \mathbb{N}, n > N$, 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 :
  $\forall n \in \mathbb{N}, n > N$, 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 :
  $\forall n \in \mathbb{N}, n > N$, 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 :
  $\forall n \in \mathbb{N}, n > N$, 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 $\exists K \in N$ 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

```python
def f(x):
    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`

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

2. Fonction `somme_f`

```python
def somme_f(valeurs):
    sf = 0
    for x in valeurs:
        sf = sf + f(x)
    return sf
```

Correction

1. Fonction `f`

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.

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.

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.

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`

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.

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.

```python
# Renvoie l'indice du plus petit élément depuis l'indice start
def indice_min_depuis(liste,start):
    imin = start
    for i in range(imin+1,len(liste)):
        if liste[i]<liste[imin]:
            imin = i
    return imin

# Tri en place par sélection
def tri_selection(liste):
    for i in range(0,len(liste)):
        imin = indice_min_depuis(liste,i)
        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 la propriété « après i tour de boucle, les i premiers éléments du tableau sont les i premiers éléments du tableau trié » est un invariant de boucle.

Correction : tri par sélection

3. Prouver la correction de cet algorithme.
   Montrons que la propriété « après i tour de boucle, les i premiers éléments du tableau sont les i premiers éléments du tableau trié » est un invariant de boucle.
   - Cette propriété est vraie à l'entrée dans la boucle

Correction : tri par sélection

3. Prouver la correction de cet algorithme.
   Montrons que la propriété « après i tour de boucle, les i premiers éléments du tableau sont les i premiers éléments du tableau trié » est un invariant de boucle.
   - Cette propriété est vraie à l'entrée dans la boucle
   - 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 $i+1 \ldots n$. Cet élément étant forcément supérieur à ceux d'indice $0, \ldots, i-1$, la propriété est vraie au rang $i+1$.

Correction : tri par sélection

3. Prouver la correction de cet algorithme.
   Montrons que la propriété « après i tour de boucle, les i premiers éléments du tableau sont les i premiers éléments du tableau trié » est un invariant de boucle.
   - Cette propriété est vraie à l'entrée dans la boucle
   - 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 la propriété « après i tour de boucle, les i premiers éléments du tableau sont les i premiers éléments du tableau trié » est un invariant de boucle.
   - Cette propriété est vraie à l'entrée dans la boucle
   - 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

Correction : tri par sélection

3. Prouver la correction de cet algorithme.
Montrons que la propriété « après i tour de boucle, les i premiers éléments du tableau sont les i premiers éléments du tableau trié » est un invariant de boucle.
   - Cette propriété est vraie à l'entrée dans la boucle
   - 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 la propriété « après i tour de boucle, les i premiers éléments du tableau sont les i premiers éléments du tableau trié » est un invariant de boucle.
   - Cette propriété est vraie à l'entrée dans la boucle
   - 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 la propriété « après i tour de boucle, les i premiers éléments du tableau sont les i premiers éléments du tableau trié » est un invariant de boucle.
   - Cette propriété est vraie à l'entrée dans la boucle
   - 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

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

$C(n) = \dfrac{n(n+1)}{2}$ 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

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

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

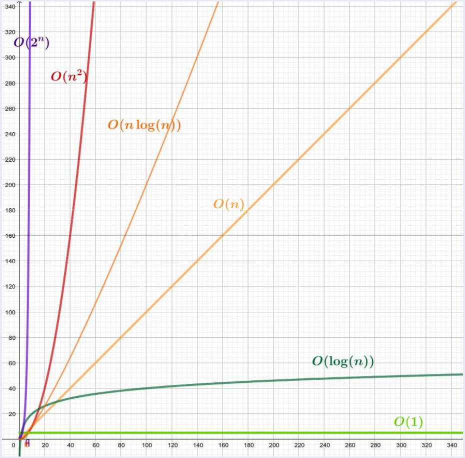
Complexités usuelles

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

Représentation graphique



Temps de calcul effectif

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

Complexité	$n=10$	$n=100$	$n=1000$	$n=10^6$	$n=10^9$
$O(\log(n))$	✔	✔	✔	✔	✔
$O(n)$	✔	✔	✔	✔	$\simeq 10$s
$O(n)\log(n)$	✔	✔	✔	✔	$\simeq 1,5$ mn
$O(n^2)$	✔	✔	✔	$\simeq 3$ h	$\simeq 300$ 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

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

```python
# Calcule x puissance n, avec n>0
def puissance(x,n):
    p = 1
    for i in range(1,n):
        p = p * n
    return p
```

Exemple : exponentiation rapide

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

```python
# Calcule x puissance n, avec n>0
def puissance(x,n):
    p = 1
    for i in range(1,n):
        p = p * n
    return p
```

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

Exemple : exponentiation rapide

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

```python
# Calcule x puissance n, avec n>0
def puissance(x,n):
    p = 1
    for i in range(1,n):
        p = p * n
    return p
```

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

Exemple : exponentiation rapide

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

```python
# Calcule x puissance n, avec n>0
def puissance(x,n):
    p = 1
    for i in range(1,n):
        p = p * n
    return p
```

2. 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

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

```python
# Calcule x puissance n, avec n>0
def puissance(x,n):
    p = 1
    for i in range(1,n):
        p = p * n
    return p
```

2. 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

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

```python
# Calcule x puissance n, avec n>0
def puissance(x,n):
    p = 1
    for i in range(1,n):
        p = p * n
    return p
```

2. 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.

3. 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 et pair et $a^{\frac{n-1}{2}}$ sinon.

Exemple : exponentiation rapide

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

```python
# Calcule x puissance n, avec n>0
def puissance(x,n):
    p = 1
    for i in range(1,n):
        p = p * n
    return p
```

2. 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.
3. 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 et pair et $a^{\frac{n-1}{2}}$ sinon.
4. Proposer une implémentation récursive de ce nouvel algorithme.

Exemple : exponentiation rapide

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

```python
# Calcule x puissance n, avec n>0
def puissance(x,n):
    p = 1
    for i in range(1,n):
        p = p * n
    return p
```

2. 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.
3. 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 et pair et $a^{\frac{n-1}{2}}$ sinon.
4. Proposer une implémentation récursive de ce nouvel algorithme.
5. 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

1. Il faut faire 13 multiplications, puisque a^{13} est calculé avec :
   $a^{13} = 1 \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a$
2. Dans ce cas, il ne faut que 5 multiplications en effet, on calcul a^{13} avec :
   $a^{13} = \left(\left(a^2 \times a\right)^2\right)^2 \times a$

Exemple : exponentiation rapide

1. Il faut faire 13 multiplications, puisque a^{13} est calculé avec :
$a^{13} = 1 \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a \times a$
2. Dans ce cas, il ne faut que 5 multiplications en effet, on calcul a^{13} avec :
$a^{13} = \left(\left(a^2 \times a\right)^2\right)^2 \times a$
3. $\begin{cases} a^n & = & \left(a^{\frac{n}{2}}\right)^2, \text{ si } n \text{ est paire} \\ a^n & = & \left(a^{\frac{n-1}{2}}\right)^2 \times a, \text{ sinon} \end{cases}$

Exponentiation rapide

4. Implémentation en Python :

```
def puissance_rapide(x,n):
    if n==0:
        return 1
    p = puissance_rapide(x,n//2)
    if n%2==0:
        return p*p
    else:
        return p*p*x
```

Exponentiation rapide

4. Implémentation en Python :

```python
def puissance_rapide(x,n):
    if n==0:
        return 1
    p = puissance_rapide(x,n//2)
    if n%2==0:
        return p*p
    else:
        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