Lycée César Baggio — PT1/PT2 — 2021/2022
Devoir de contrôle d'informatique
Mardi 16 novembre 2021, 16:00 – 18:00

- Sans document, sans calculatrice ou autre dispositif électronique.
- Lorsque l'on demande d'écrire un code `Python`, on prendra soin de numéroter les lignes et de faire apparaître très clairement l'indentation.
- L'utilisation des propriétés `min`, `max` ou encore `sort` sont interdites.
- La complexité, ou le temps d'exécution, d'un programme `P` est le nombre d'opérations élémentaires nécessaires à l'exécution de `P`. Lorsqu'il est demandé de donner une certaine complexité, cette dernière sera donnée par une borne supérieure asymptotique (en $O(\dots)$) et sera justifiée.

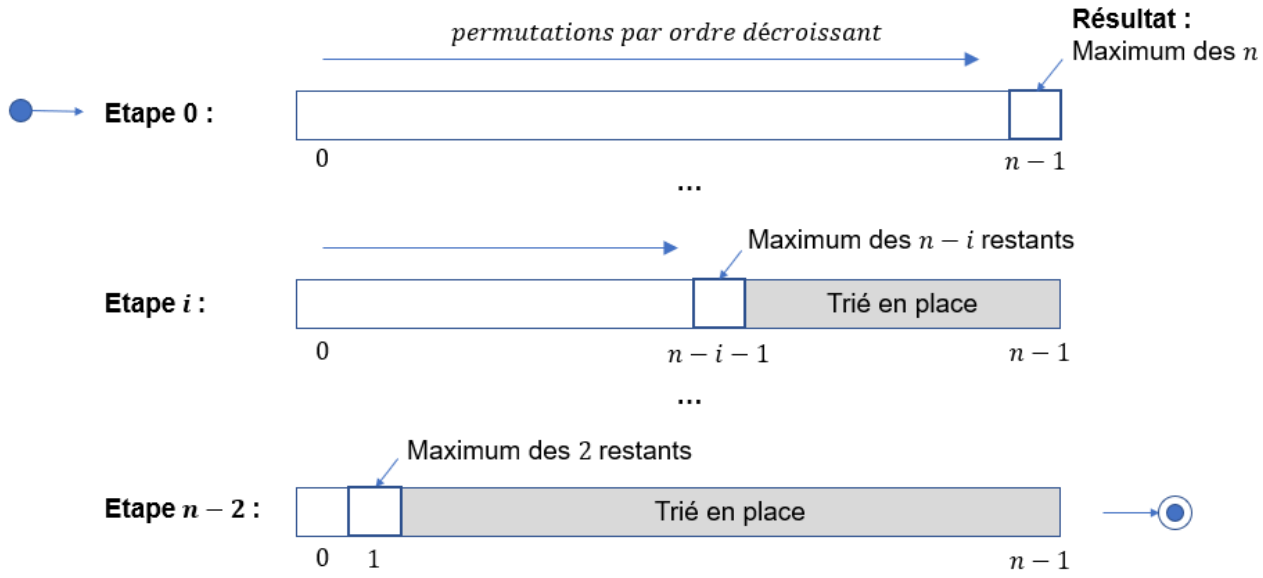
---

## Exercice 1 - Tri à bulles

**Rappels :**

Le **tri à bulles** ou **tri par propagation** est un algorithme de tri qui consiste à faire remonter progressivement les plus grands éléments d'un tableau, comme les bulles d'air remontent à la surface d'un liquide. Son fonctionnement s'appuie sur des permutations répétées d'éléments contigus qui ne sont pas dans le bon ordre.

L'algorithme parcourt le tableau, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet du tableau, l'algorithme recommence l'opération. L'image ci-dessous résume les différentes étapes d'un tel tri.




Après un premier parcours complet du tableau, le plus grand élément est forcément en fin de tableau, à sa position définitive. En effet, aussitôt que le plus grand élément est rencontré durant le parcours, il est mal trié par rapport à tous les éléments suivants, donc échangé à chaque fois jusqu'à la fin du parcours. À la seconde passe, il suffit donc de parcourir à nouveau le tableau, en s'arrêtant à l'avant-dernier élément. Après ce deuxième parcours, les deux plus grands éléments sont à leur position définitive. Et ainsi de suite de passe en passe...

La fonction réalisant ce tri est donnée ci-dessous.

Implémentation en langage Python du tri à bulles

```
1  """ ------------------------------------------------------------
2          Tri à bulles
3      ------------------------------------------------------------
4      Entrée : une liste L de n éléments à trier
5      Sortie : la liste initiale passée en argument
6               est triée par ordre croissant
7
8      Principe : à l'étape i, boucle sur les n - i premiers éléments et
9      permutation de deux éléments successifs en ordre décroissant.
10     ------------------------------------------------------------"""
11 def triBulle(L):
12     n = len(L)
13     for i in range(n-1):
14         for j in range(n-i-1):
15             if L[j] > L[j+1]:
16                 L[j],L[j+1]=L[j+1],L[j]
```

1. Comment s'appelle une séquence dont les données sont séparées par des virgules ? Que réalise l'instruction de la ligne 16 ?

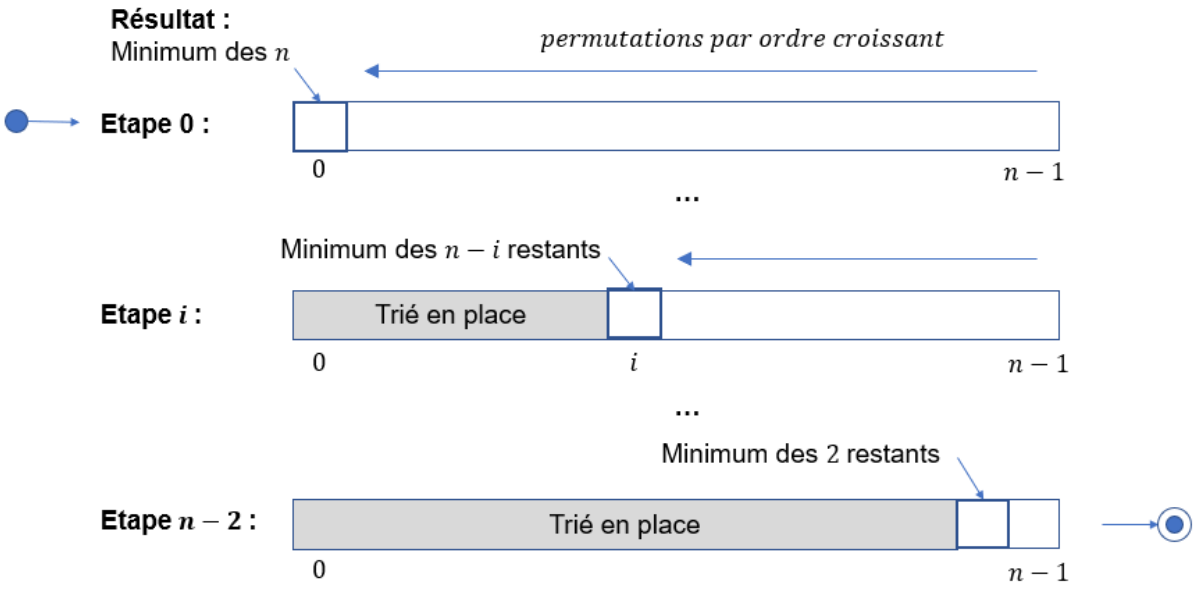
On rappelle que pour un tri, la complexité est calculée en comptant le nombre de comparaisons effectuées.

2. À quelles données correspond le meilleur des cas ? Le pire des cas ? Donner dans ces deux cas la complexité en temps en fonction de $n$.

3. Comparer cette complexité avec celle du tri par insertion.

### 1.1 Tri à bulles en arrière (tri à plombs 🙂)

Le principe du tri à bulles étant de remonter progressivement les valeurs maximales vers le haut du tableau (on parle de tri en avant), on peut réaliser le même tri en descendant progressivement les valeurs minimales vers le bas du tableau (on parle de tri en arrière), comme illustré ci-dessous.




4. Écrire la fonction `triBullesArriere(liste)`, prenant en argument une liste non triée, et triant en place cette liste selon ce principe.

### 1.2 Tri à bulles avant/arrière (cocktail sort)

On cherche maintenant à exploiter ces deux principes, en alternant à chaque étape le tri avant, puis le tri arrière. Cette technique offre l'avantage de mieux trier les éléments restants à chaque étape, et d'obtenir bien souvent une liste triée avant la fin de l'algorithme.

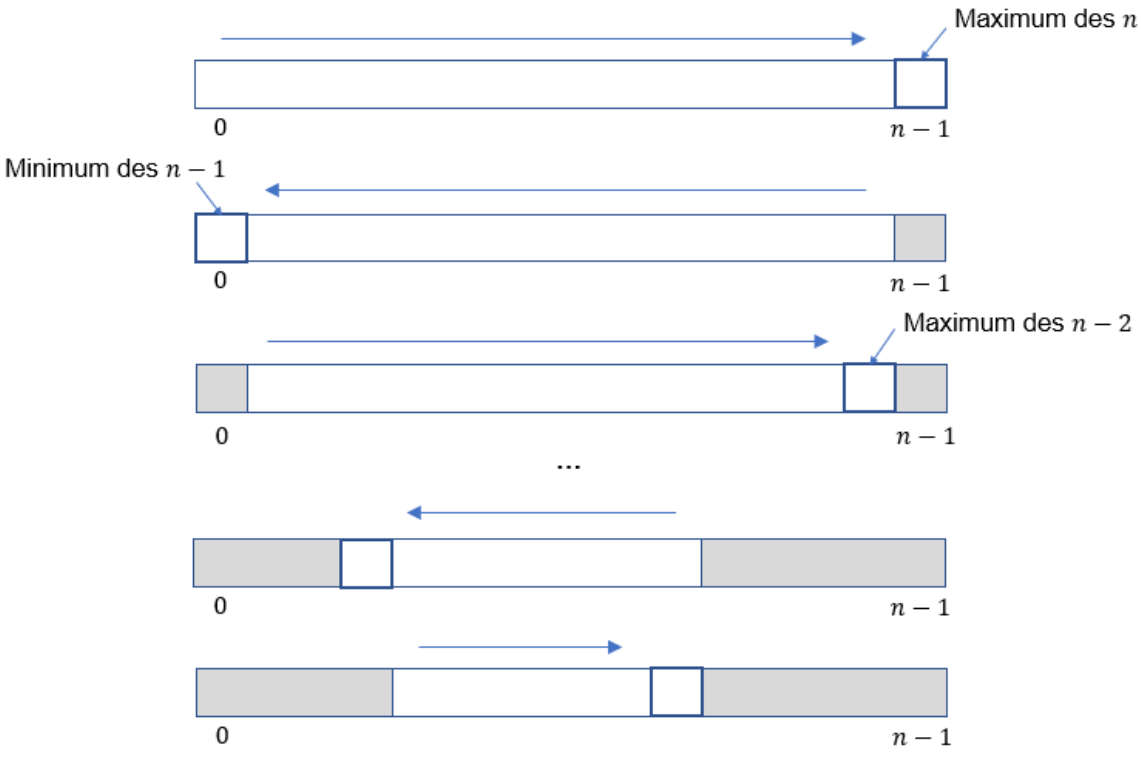
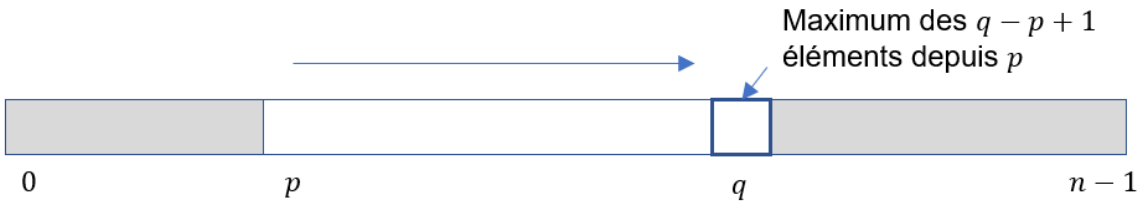
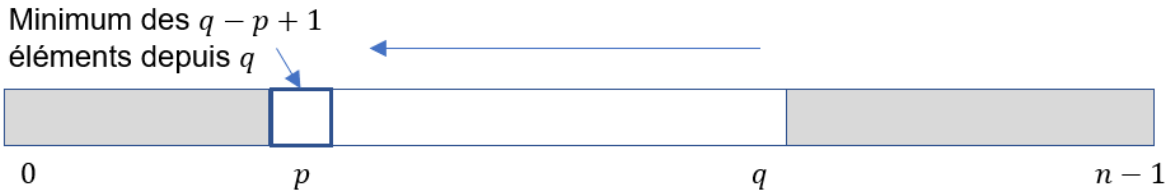



*Illustration de l'alternance des tris avant et arrière.*

Il faut pour cela effectuer une étape unitaire de chacun de ces algorithmes, d'un emplacement quelconque $p$ jusqu'à un emplacement quelconque $q$, soit donc pour un nombre de données $q-p+1$ défini, et donc pouvoir remonter (redescendre) la valeur maximum (minimum) de $q-p+1$ données en avant (en arrière) à partir d'un emplacement $p$ ($q$), dans une liste donnée.




*Remontée du maximum de $q-p+1$ données en avant à partir de l'emplacement $p$.*




*Descente du minimum de $q-p+1$ données en arrière à partir de l'emplacement $q$.*

5. Sur le principe d'une étape du tri à bulle en avant, écrire la fonction `remonterMax(L,p,q)`, qui effectue la remontée vers le haut du maximum des `q-p+1` valeurs, à partir de l'index bas `p`, d'une liste `L` donnée.

6. Sur le principe d'une étape du tri à bulle en arrière, écrire la fonction `descendreMin(L,p,q)`, qui effectue la descente vers le bas du minimum des `q-p+1` valeurs, à partir de l'index haut `q`, d'une liste `L` donnée.

7. Écrire une fonction **itérative** `triAvantArriereIter(L)` prenant comme argument une liste `L` non triée, et triant en place cette liste sur le principe du tri avant/arrière utilisant les fonctions `remonterMax(L,p,q)` et `descendreMin(L,p,q)`.

## 1.3 Cocktail sort récursif

On peut écrire ce tri de manière plus simple, en utilisant le principe de récursivité croisée : il suffit qu'en fin de chacune des deux fonctions de remontée et de descente, l'une fasse à appel à l'autre et réciproquement, en positionnant comme il se doit les différents arguments. Cela suppose de définir deux nouvelles fonctions, `remonterMaxRec(L,p,q)` et `descendreMinRec(L,p,q)`, et l'algorithme à implémenter est alors celui-ci :

- on commence à l'emplacement `p = 0` pour une liste `L` de `q+1 = n` données ;
- on appelle la fonction `remonterMaxRec(L,p,q)`, qui réalise la remontée vers le haut du maximum des `q-p+1` valeurs, à partir de l'index bas `p`, de la liste `L` donnée, puis appelle la fonction `descendreMinRec(L,p,q-1)` pour effectuer la descente de minimum suivante ;
- la fonction `descendreMinRec(L,p,q)` réalise la descente vers le bas du minimum des `q-p+1` valeurs , à partir de l'index haut `q`, de la liste `L`, puis appelle la fonction `remonterMaxRec(L,p+1,q)` pour effectuer la remontée de maximum suivante ;
- la récursivité prend fin quand le nombre de données devient égal à 1 (il n'y a plus rien à trier).

8. Écrire les fonctions récursives croisées `remonterMaxRec(L,p,q)` et `descendreMinRec(L,p,q)`. Écrire enfin la fonction `triBulleRec(L)` qui effectue le tri à bulle récursif croisé.

## Exercice 2 - Fonction WilliamGeorge

On décrit un polynôme

$$P(X) = a_0 + a_1 X + \cdots + a_n X^n = \sum_{k=0}^{n} a_k X^k$$

par la liste $L$ de ses coefficients $a_k$, par exemple le polynôme $3X^2 + X - 1$ est représenté par la liste

$$[-1, 1, 3]$$

On donne les fonctions suivantes :

```
1  def aux(P, x, d):
2      if d == len(P):
3          return 0
4      else:
5          return P[d] + x * aux(P, x, d + 1)
6
7  def WilliamGeorge(P, x):
8      return aux(P, x, 0)
```

1. Soit `x` un flottant quelconque.
   Dessiner l'arbre des appels récursifs pour l'appel `WilliamGeorge([-1, 1, 3], x)`. On précisera, pour chaque appel la valeur de retour obtenue.
   Quelle valeur retourne l'appel `WilliamGeorge([-1, 1, 3], x)` ?

2. Que fait la fonction `WilliamGeorge` ? Concevoir son en-tête (ou sa spécification).

3. Justifier la terminaison de la fonction `WilliamGeorge`.

Dans la suite, on note $n =$`len(P)`.

4. Combien d'appels récursifs sont effectués lors de l'appel de `WilliamGeorge(P, x)` ?
   En déduire une borne supérieure asymptotique de la complexité en espace de la fonction `WilliamGeorge` en fonction de $n$.

5. Donner une relation de récurrence vérifiée par la complexité en temps de la fonction `aux` en fonction de $n =$`len(P)`.
   En déduire une borne supérieure asymptotique de la complexité en temps de la fonction `WilliamGeorge` en fonction de $n =$`len(P)`.

## Exercice 3 - Suite de Thue-Morse

On appelle *suite de Thue-Morse* la suite définie récursivement par

$$\begin{cases} t_0 &= 0 \\ t_n &= t_{n/2} & \text{si } n > 0 \text{ et } n \text{ est pair} \\ t_n &= 1 - t_{(n-1)/2} & \text{si } n \text{ est impair.} \end{cases}$$

On peut montrer que cette suite ne contient que les valeurs 0 et 1. Les premières valeurs de la suite sont : $0,1,1,0,1,0,0,1,1,\ldots$.

1. Écrire une fonction de prototype `estPair(n)` qui retourne `True` ou `False` suivant que l'entier `n` est pair ou non.

2. Donner la complexité en temps de l'appel `estPair(n)`.

3. Écrire une fonction itérative `ThueMorseTableau` ayant un paramètre entier `n` et qui retourne un tableau contenant les chiffres d'indice inférieur ou égal à `n` de la suite de Thue-Morse. Par exemple,

**Python 3 Shell**

```
>>> ThueMorseTableau(6)
[0, 1, 1, 0, 1, 0, 0]
```

4. Donner la complexité en temps de l'appel `ThueMorseTableau(n)`. Justifier votre réponse.

5. Écrire une fonction récursive `chiffreThueMorseRecursive` ayant un paramètre entier `n` et qui retourne $t_n$ ($n \geq 0$). Par exemple

**Python 3 Shell**

```
>>> chiffreThueMorseRecursive(9)
0
```

6. Donner la liste des appels récursifs lors de l'exécution de `chiffreThueMorseRecursive(9)`.

7. Quelle est la complexité en temps de votre fonction `chiffreThueMorseRecursive` ? Justifier votre réponse.

Étant donné un tableau `s` contenant une suite binaire, on veut savoir si cette suite est une sous-suite des `n` premiers chiffres de la suite de Thue-Morse.

8. Pour cela, écrire une fonction `estSousSuiteThueMorse(s, n)` qui d'abord construit un tableau représentant les `n` premiers chiffres de la suite de Thue-Morse, ensuite compare le contenu de ce tableau avec le contenu du tableau `s`. La fonction renvoie `True` si `s` correspond à une sous-suite des des `n` premiers chiffres de la suite de Thue-Morse et renvoie `False` sinon. Par exemple, les 10 premiers chiffres étant $011\underbrace{0100110}0$

**Python 3 Shell**

```
>>> estSousSuiteThueMorse([0, 1, 0, 0, 1, 1], 10)
True
>>> estSousSuiteThueMorse([1, 0, 1], 4)
False
```