# TD5 : Algorithme, invariant de boucle et complexité (1/2)

## Exercice 1 : Analyse de la complexité d'un algorithme

On considère le pseudo-code suivant, comportant deux « tant que » imbriqués. On cherche à mesurer la complexité de cette imbrication en fonction de n. Pour cela, on utilise la variable « compteur », qui est incrémentée à chaque passage dans le « tant que » interne.

```
Variables :
  n : entier
  compteur : entier
  i, j : entiers
Début
  Afficher(« Quelle est la valeur de n ? »)
  Saisir(n)
  compteur ← 0
  i ← 1
  Tant que (i < n) Faire
      j ← i + 1
      Tant que (j ≤ n) Faire
          compteur ← compteur + 1
          j ← j + 1
      Fin tantque
      i ← i * 2
  Fin tantque
  Afficher(compteur)
Fin
```

a. Quelle est la valeur finale du compteur dans le cas où n = 16 ?

Pour i=1, j varie de 2 à 16 inclus, on fait donc 15 incrémentations du compteur.
Pour i=2, j varie de 3 à 16 inclus, on fait donc 14 incrémentations du compteur.
Pour i=4, j varie de 5 à 16 inclus, on fait donc 12 incrémentations du compteur.
Pour i=8, j varie de 9 à 16 inclus, on fait donc 8 incrémentations du compteur.
Ensuite, i vaut 16, donc on sort du « Tant que (i < n) ».
Au total, on a donc fait 15+14+12+8 = 49 incrémentations du compteur. Donc compteur vaut 49 en sortie du programme.

b. Considérons le cas particulier où n est une puissance de 2 : on suppose que $n = 2^p$ avec p connu. Quelle est la valeur finale du compteur en fonction de p ? Justifiez votre réponse.

i prend successivement les valeurs suivantes : $2^0$, $2^1$, $2^2$, … $2^{p-1}$, soit $2^k$ avec k variant de 0 à (p-1). Pour chacune de ces valeurs, on fait (n-i) incrémentations, soit ($2^p$ - $2^k$) incrémentations. Ensuite i vaut $2^p$, ce qui provoque la sortie du « Tant que (i < n) ». On ne fait pas d'incrémentations du compteur pour cette dernière valeur de i.
$\sum_{k=0}^{p-1}(2^p - 2^k) = p \times 2^p - \sum_{k=0}^{p-1} 2^k = p \times 2^p - (2^p - 1) = (p-1) \times 2^p + 1$
Rappel : $\sum_{k=0}^{p-1} 2^k = 2^0 + \cdots + 2^{p-1} = 2^p - 1$
Ainsi, la valeur finale du compteur est $(p-1) \times 2^p + 1$.

c. Réexprimez le résultat précédent en fonction de n.

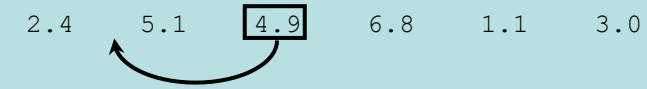
$(\log_2(n) - 1) \times n + 1$. (rappel : $n = 2^p \rightarrow p = \log n$)
On a donc une complexité en $O(n \log n)$.

## Exercice 2 : Tri par insertion

Le tri par insertion est l'algorithme utilisé par la plupart des joueurs lorsqu'ils trient leur « main » de cartes à jouer. Le principe consiste à prendre le premier élément du sous-tableau non trié et à l'insérer à sa place dans la partie triée du tableau.

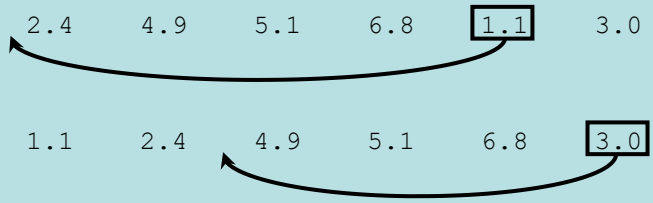
a. Dérouler le tri par insertion du tableau {5.1, 2.4, 4.9, 6.8, 1.1, 3.0}.


On commence par essayer de placer le deuxième élément, soit 2.4, dans le sous-tableau allant des cases 1 à 1. Comme 5.1 est supérieur à 2.4, on doit placer 2.4 avant 5.1. On passe ensuite au troisième élément, c'est-à-dire 4.9.




4.9 est inférieur à 5.1 mais supérieur à 2.4, on le place donc entre les deux. On passe ensuite au quatrième élément, soit 6.8.


6.8 ne bouge pas car il est supérieur à 5.1. On passe à 1.1 puis 3.0.




Le tableau est maintenant trié.

Le principe est simple à comprendre, mais la difficulté apparaît lorsqu'on essaie d'écrire l'algorithme avec des données structurées en tableau : il va parfois falloir déplacer plusieurs éléments pour en placer un, car il faut lui « faire de la place ».

b. Ecrire en langage algorithmique le corps de la procédure de tri par insertion, par ordre croissant, d'un tableau de réels :

**Procédure** tri_par_insertion (tab : tableau [1..n] de réels)
**Précondition** : tab[1], tab[2], … tab[n] initialisés
**Postcondition** : tab[1] ≤ tab[2] ≤ … ≤ tab[n]
**Paramètres en mode donnée** : aucun
**Paramètre en mode donnée-résultat** : tab

**Procédure tri_par_insertion** (tab : tableau [1..n] de réels)

```
Précondition : tab[1], tab[2], … tab[n] initialisés
Postcondition : tab[1] ≤ tab[2] ≤ … ≤ tab[n]
Paramètres en mode donnée : aucun
Paramètre en mode donnée-résultat : tab
Variables locales :
  i, j : entiers
  elt_a_placer : reel
Début
1     Pour i allant de 2 à n par pas de 1 Faire
2       elt_a_placer  ← tab[i]
3       j ← i – 1
4       Tant que (j > 0)  et (tab[j] > elt_a_placer) Faire
5          tab[j+1] ← tab[j]   {on pousse l'élément j vers la droite}
6          j ← j – 1
7       FinTantQue
8       tab[j+1] ← elt_a_placer
9     FinPour
Fin tri_par_insertion
```

c. Donner l'invariant de boucle correspondant à cet algorithme, en démontrant qu'il vérifie bien les 3 propriétés d'un invariant de boucle : initialisation, conservation, terminaison.

**Invariant de boucle :** Juste avant l'itération i, le sous-tableau tab[1...(i-1)] se compose des éléments qui occupaient initialement les positions tab[1...(i-1)], mais qui sont maintenant triés.

**Initialisation** : Il faut démontrer que la propriété est vraie juste avant l'itération i=2. Il faut donc montrer que « le sous-tableau tab[1...1] se compose des éléments qui occupaient initialement les positions tab[1...1], mais qui sont maintenant triés. » Avant l'itération i=2, on n'a pas modifié le tableau, donc l'élément tab[1] est bien l'élément tab[1] originel. Par ailleurs, un sous-tableau qui ne contient qu'un seul élément est forcément trié. La propriété est donc vérifiée.

**Conservation** : Il faut démontrer que si la propriété est vraie juste avant l'itération i, alors elle reste vraie juste avant l'itération i+1. On suppose donc que « le sous-tableau tab[1...(i-1)] se compose des éléments qui occupaient initialement les positions tab[1...(i-1)], mais qui sont maintenant triés », et on doit montrer qu'après avoir exécuté le corps de la boucle Pour, « le sous-tableau tab[1...i] se compose des éléments qui occupaient initialement les positions tab[1...i], mais qui sont maintenant triés. » On sait que l'élément tab[i] va être inséré quelque part entre les positions 1 et i incluses, mais qu'il ne sera en aucun cas placé après la case i. Les éléments des cases 1 à i-1 peuvent être déplacés vers la droite, mais d'un cran seulement : aucun d'entre eux ne se retrouvera après la case i. Donc les éléments situés initialement dans les cases 1 à i restent bien dans ce bloc de cases. Par ailleurs, l'élément i va bien être inséré de sorte que le sous-tableau tab[1...i] reste trié. On a donc bien conservation de la propriété.

**Terminaison** : Il faut montrer qu'une fois la boucle terminée, l'invariant de boucle fournit une propriété utile pour montrer la validité de l'algorithme. Ici, la boucle prend fin quand i dépasse n, c'est-à-dire pour i=n+1. On sait alors que « le sous-tableau tab[1...n] se compose des éléments qui occupaient initialement les positions tab[1...n], mais qui sont maintenant triés. » Le tableau est donc trié, ce qui montre que l'algorithme est correct.

d. Evaluer le nombre de comparaisons de réels et le nombre d'affectations de réels pour un tableau de taille *n*, dans le cas le plus défavorable (tableau trié dans l'ordre décroissant). Cet algorithme est-il meilleur que le tri par sélection (tri du minimum) vu en cours ?

**Affectations de réels** : Dans le cas le plus défavorable, on fait (i-1) décalages avant d'insérer chaque élément.

*Rappel mathématique utile pour (presque) tous les calculs de complexité* : $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$A = \sum_{i=2}^{n}(1 + (i-1) + 1)$ *(lignes 2, 5 et 8)*

$A = \sum_{i=2}^{n}(1 + i) = n - 1 + \sum_{i=2}^{n} i = n - 1 + \frac{n(n+1)}{2} - 1$

$A = \frac{n^2}{2} + \frac{3n}{2} - 2$

Le nombre d'affectations est donc $O(n^2)$ alors qu'il était $O(n)$ pour le tri par sélection.

**Comparaisons de réels** : Dans le cas le plus défavorable, on fait (i-1) comparaisons de réels avant d'insérer l'élément (on va de j=i-1 jusqu'à j=0, mais pour j=0 on ne fait pas la comparaison de réels, puisque j>0 est faux).

$C = \sum_{i=2}^{n}(i - 1)$ *(ligne 4 deuxième condition)*

$C = \sum_{i=2}^{n}(i) - (n-1) = \frac{n(n+1)}{2} - 1 - (n-1)$

$C = \frac{n^2}{2} - \frac{n}{2}$

On retrouve le même nombre de comparaisons que pour le tri par sélection en $O(n^2)$.