CORRECTION ET TERMINAISON DES ALGORITHMES

Matériel en partie du à Michel Syska.
Lien web :
http://www-sop.inria.fr/members/Stephane.Perennes/TD1/Fiche1.pdf

Rendu de TD

1 Simple Boucle

Soit la fonction python suivante :

```
1  def sommeNormale(L):
2    Res=0
3    for i in range(len(L)):
4      Res+= L[i]
5  return(Res)
```

1. Quelle précondition (en terme de somme partielle), peut on placer juste avant la ligne 4 ?
   Réponse La précondition est que Res est égal à la somme de $j = 0$ jusque $j = i - 1$ des $L[j]$. Formellement ceci signifie que $Res = \sum_{j\in[0,i-1]} L[j]$. Notons enfin que pour $i = 0$ l'intervale $[0, i - 1] = [0, -1]$ est vide et $Res = 0$ (de part l'initialisation) la condition est vérifiée quand $i = 0$.
2. Quelle postcondition, peut on affecter à la ligne 4 ?
   Réponse La postcondition est que Res est égal à la somme de $j = 0$ jusque $j = i$ des $L[j]$. Autrement dit, $Res = \sum_{j\in[0,i]} L[j]$.
   Pour conclure nous remarquons que si la précondtion est vraie pour i alors la post-condition est vraie pour i du fait de la ligne $Res+ = L[i]$. Comme la post-condition est située à la fin du bloc de boucle la post-condition pour i est aussi la pré-condition pour $i + 1$. On peut donc conclure (par induction sur i) que les deux conditions sont vraies pour tout i et il s'en suit qu'à la fin de la boucle on a $Res = \sum_{i=0,i=lenL-1} L[i]$.

```
1 def sommeLente(L) {
2   if len(L==0)
3      return 0
4    return L.pop() + sommeLente(L)
```

1. Que calcule sommeLente ? le démontrer par induction.
   Réponse L'algorithme calcule la somme des éléments de L. Pour le démontrer par induction (sur la longueur de la liste L) on remarque tout d´abord que la propriété est vraie quand L est vide (de longueur 0. Maintenant si la propriété est vraie pour toute liste de longueur $< n$, alors pour une liste de longueur n sa somme vaut $L.pop$ + la somme de la liste restante, qui est de longueur $n - 1$ et qui est contenue désormais dans L. L'hypothèse d'induction implique que $sommeLente(L)$ retourne la somme des $n - 1$ éléments restants. La ligne 4 retourne donc la somme de la liste.

2. Comparer les vitesses d'execution de sommeNormale et de sommeLente, expliquer la différence.
   Réponse : SommeLente est certainement plus lente que somme normale, dans les deux cas la complexité est en $\Theta(n)$ mais du fait des constantes cachées du Θ sommeLente devrait être nettement plus lente avec un facteur de ralentissement qui devrait être assez constant. En effet pour chaque élément de la liste, SommeNormale effectue une addition, une comparaison et une incrémentation d'indice tandis que SommeLente va effectuer un appel de fonction plus une addition.
3. tester sommeLente(list(range(10**5)), que ce passe t'il ?
   Réponse :
   Le nombre maximal d'appels récursifs imbriqués (La hauteur de la pile d'appel des fonctions) de la machine va probablement être dépassé. Ce nombre dépend du langage de programmation et de la machine utilisée, il peut être accru au besoin.

Remarque :
L'objectif de l'exercice était triple :

- présenter une utilisation de la récursivité élémentaire.
- montrer qu'un algorithme récursif (somme lente) est souvent plus lent qu'un algorithme itératif (somme normale). Cependant la version récursive est souvent bien plus simple à exprimer.
- présenter une *mauvaise* utilisation de la récursivité, en effet ici l'arbre d'appel est de profondeur n car il est $f(n), f(n-1), \ldots, f(k), \ldots f(1)$. Or on utilise en général la récursivité (par exemple pour le tri fusion aussi nommé tri dichotomique) quand pour chaque appel la *taille* du problème n diminiue d'un facteur constant tel que 2. En ce cas l'arbre d'appel sera :

  $$f(n), f(n/2), \ldots f(n/2^k), \ldots, f(1)$$

  Ainsi sa profondeur ne sera que $\log 2(n)$. Plus généralement si la taille est multipliée par $\alpha < 1$ alors l'arbre d'appel sera :

  $$f(n), f(\alpha n), \ldots f(\alpha^k n), \ldots f(1)$$

  La récursion se termine alors pour un certain k_0 tel que $n\alpha^{k_0} = 1$ soit $\frac{1}{\alpha^{k_0}} = n$ et donc $k_0 = \log_{\frac{1}{\alpha}}(n)$, notez enfin que de ce fait $k_0 = \Theta(\log(n))$.

2 Boucles imbriquées

Soit les deux méthodes Java suivante :

```java
public static void bonjour1(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println("Bonjour 1");
        }
    }
}
```

```java
public static void bonjour2(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            System.out.println("Bonjour 2");
```

```
5               }
6           }
7   }
```

1. Combien de fois la méthode `bonjour1(5)` affiche « Bonjour 1 » ?
   Réponse : bonjour est affiché 25 fois.
2. Combien de fois la méthode `bonjour2(5)` affiche « Bonjour 2 » ?
   Réponse : bonjour est affiché 10 fois.
3. Combien de fois la méthode `bonjour1(n)` affiche « Bonjour 1 » ? Exprimez la réponse en fonction de n.
   Réponse : bonjour est affiché $n \times n = n^2$ fois. En effet le corps de première boucle est effectuée n fois. Celui-ci contient une boucle interne dont le comportement ne dépend pas de la boucle externe et celle-ci affiche bonjour n fois.
4. Combien de fois la méthode `bonjour2(n)` affiche « Bonjour 2 » ? Exprimez la réponse en fonction de n.
   Réponse : bonjour est affiché $\frac{n(n-1)}{2}$ fois. Dans ce second cas la boucle interne dépend de la boucle externe. Si i est l'indice courant de la boucle interne le corps de la boucle interne est effectué i fois et donc bonjour est affiché i fois. Au total bonjour est donc affiché $B(n) = 0 + 1 + 2 + \ldots + (n-1)$ fois, soit $B(n) = \sum_{i \in [0,n-1]} i$. La somme $B(n)$ est classique et vaut $\frac{n(n-1)}{2}$. On peut le montrer facilement en utilisant l'astuce suivante :

$$\begin{array}{rc} & 0 + 1, \ldots + (n-1) \\ B(n) + B(n) = & (n-1) + (n-2) + \ldots + 0 \\ & n \times (n-1) \end{array}$$

   Et donc $2B(n) = n(n-1)$. Notons que **il n'était pas demandé de démontrer ce résultat** qui est présenté dans le cours et est *classique*. On peut aussi démontrer que $B(n) = \frac{n(n-1)}{2}$ par induction. Effet si $B(n) = \frac{n(n-1)}{2}$ pour $n < n_0$ on aura :

$$B(n_0) = n_0 - 1 + B(n_0 - 1) = n_0 - 1 + \frac{(n_0 - 1)(n_0 - 2)}{2} = \frac{(n_0 - 1)(n_0)}{2}$$

   .

Ceci valide l'hypothèse d'induction (récurrence). Pour que la preuve soit complète il faut aussi vérifier que la formule est correcte pour $n = 0$, C'est le le cas car $B(0) = 0$.

3 Tri de tableaux par sélection

Cette méthode de tri consiste à itérativement :

— selectionner le minimum x du tableau, et à le supprimer.

— le placer en tête du tableau résultat.

— recommencer.

Dans la méthode suivante on utilise le résultat est placé au début du tableau ce qui optimise l espace.

Soit la méthode `triSelection` qui implante l'algorithme de tri suivant :

```
1  /**
2   * Preconditions :
3   * Postconditions :
4   */
5  public static void triSelection(int[] tab) {
6      for (int i = 0; i < tab.length; i++) {
7          // A1 : tab[0..i-1] est trié
8          int indiceMin = i;
9          for (int j = i + 1; j < tab.length; j++){
10             // A2 : tab[indiceMin] <= tab[k] pour tout i <= k <j
11             if (tab[indiceMin] > tab[j]) {
12                 indiceMin = j;
13             }
14         }
15         int aux = tab[i];
16         tab[i] = tab[indiceMin];
17         tab[indiceMin] = aux;
18     }
19 }
```

Préliminaire : L'algorithme fonctionne comme suit à l étape i :

- **le sous tableau $[0, i-1]$ est trié (Assertion A1).**
- **On cherche le minimum du sous tableau $[i, n-1]$ et on l'échange avec l'élément $T[i]$ (Assertion A2).**

1. Déroulez « à la main » `triSelection(tab)` avec le tableau d'entiers `tab` suivant : `int[] tab = { 3, 25, 50, 8, 1, 3, 49 };`
   Vous devez donner les valeurs du tableau `tab` à chaque étape de l'algorithme (ici ligne 7 du code).
   Réponse :

Index i	Tableau	IndiceMin	Nombre de boucles j
0	3, 25, 50, 8, 1, 3, 49	4	6
1	1, 25, 50, 8, 3, 3, 49	4	5
2	1, 3, 50, 8, 25, 3, 49	5	4
3	1, 3, 3, 8, 25, 50, 49	3	3
4	1, 3, 3, 8, 25, 50, 49	4	2
5	1, 3, 3, 8, 25, 50, 49	6	1
6	1, 3, 3, 8, 25, 49, 50	5	0

2. On considère qu'à chaque étape de l'algorithme, on peut lire un élément du tableau, le comparer à une valeur stockée en mémoire et procéder à des affectations d'un nombre constant de variables

(en gros un passage dans une boucle). Quel est le nombre maximum d'étapes pour trier un tableau de taille n avec la méthode `triSelection` ? On compte donc le nombre de fois où on passe dans la comparaison `if` ligne 11.

Réponse :

Dans le cas du tableau donné en exemple de taille 7 on trouve $6+5+4+3+2+1 = \frac{7(7-1)}{2} = 21$.

Dans le cas général on trouve $n + (n-1) + \ldots + 1 = \frac{n(n-1)}{2}$.

3. Donnez un exemple de tableau avec $n = 7$ qui atteint ce nombre.

   Réponse :

   La tableau de l'exemple fait l'affaire, comme tout tableau de taille 7.

4. Quel est le nombre minimum d'étapes pour trier un tableau de taille n avec la méthode `triSelection` ?

   Réponse :

   Le nombre d'étapes ne dépend pas du tableau, contrairement à d'autre algorithme le temps de tri ne dépend pas des données.

5. Précisez les preconditions, postconditions et expliquez les assertions (ici A_1 et A_2 données dans le code) utiles à se convaincre que la méthode termine et donne la bonne réponse au probléme posé (le tri).

   Réponse :

   Les assertions permettent de valider les hypothèses indiquées dans la présentation de l'algorithme. L assertion A1 indique que le début du tableau jusque $i-1$ est trié, elle est vraie au début car ce tableau est vide et l assertion A2 permet de maintenir la validité de l'assertion A1 lors du déroulement de l'algorithme. L'assertion A2 établit IndexMin à la fin de deuxième boucle retourne bien l'index de l'élément minimal de la seconde partie du tableau (i.e. les indices $[i, n-1]$). Notons qu'en cas d'égalité si le premier indice d'un des plus petits éléments qui est retourné. Comme cet élément minimal est inséré à la fin de le première section du tableau celle-ci demeure triée.

Addendum :

En python on pourrait procéder ainsi, avec une version moins optimisée mais plus simple et synthétique. Cette dernière utilise plus d'espace, de plus la liste non triée est parcourue deux fois, une fois pour cherche l'index min et une fois pour extraire l'élément associé.

```
1  def Tri(mL):
2      L=list(mL)
3      Res=list()
4      while(len(L)>0):
5          print (Res, "//", L)
6          a=Findmin(L)
7          myMin=L.pop(a)
8          Res.append(myMin)
9      return (Res)
10
11     def Findmin(L):
12     if len(L)=0:
13       return(None)
14     indexmin=0
15     val=L[0]
16     for i in range(len(L)):
17         if L[i]<L[indexmin]:
18             indexmin=i
19     return(indexmin)
```

4 Tri Pivot

Dans ce tri on selectionne un élément du tableau x et on scinde le tableau en deux parties

— T_{petit} qui contient les éléments strictement inférieurs à x.

— T_{grand} qui contenant éléments supérieurs ou egal à x.

— on trie T_{petit}, T_{grand}

On appelle x le pivot, il peut être le premier élément du tableau ou un élément pris au hasard.

```
1  import random
2
3  def TriPivot(L):
4      if len(L)<=1 :
5          return (L)
6      pivot=random.choice(L)
7      Petits= [ x for x in L if x < pivot]
8      Grands= [ x for x in L if x >= pivot]
9
10      Res= list(TriPivot(Petits) + TriPivot(Grands) )
11      return (Res)
```

1. En quoi la condition le la ligne 4 importe ?
   Réponse :
   Cette condition permet de terminer la récursion, en effet quand la liste est de taille 1 Petit sera vide et grand la liste elle même et le programme va boucler à l'infini.
2. Que ce passe t-il si le pivot est le minimum (resp. le maximum) de la liste ?
   Réponse :
   Si le pivot est l'élément minimal on va avoir $Petit = \emptyset, Grand = L$, si le pivot est l'élement maximal x on aura $Grand = [x]$ et petit sera une liste de longueur $len(L) - 1$.
3. montrer que 1 fois sur 2 on on a $len(grand) <= 3len(L)/4$ et $len(Petit) <= 3len(L)/4$.
   Réponse :
   Appelon le *rang* d'un élément son index dans la liste triée. Ainsi si la liste est de longueur n le minimum est de rang 0, le maximum de rang n et l'élément médian de rang $n/2$. Supposons que le rang du pivot soit r. Par définition Petit contient les élément de rang inférieur à r et grand ceux de rang $\geq r$. Ainsi à 1 rès $|Petit| = r, |Grand| = n - r$. Si donc $r \in [\frac{n}{4}, \frac{3n}{4}]$ on aura $|Petit|, |Grand| \leq \frac{3n}{4}$. Pour conclure ajoutons que l'intervale $[\frac{n}{4}, \frac{3n}{4}]$ est de longueur $n/2$ et donc que le pivot a une chance sur 2 d'être dans cet intervale.
4. montrer que le pire temps d exécution peut être quadratique (i.e en n^2). Si à chaque appel le pivot choisi est de rang n peut avoir n niveaux de recursion. Or chaque niveau coute de l'ordre de $Theta(n)$ car on va comparer tout les éléments de la liste au pivots associés a ce niveau de récursion. Au total on va donc avoir un nombre d'opérations (i.e. un complexité de) $n \times \Theta(n) = \Omega(n^2)$
5. faire de nombreux test et afficher la distribution du temps d'exécution.

Le problème des éléments répétés : Si des élements sont répétés (par exemple la liste est 3333333333) le pivot ne va pas diviser la liste. Pour résoudre ce problème on peut par exemple retourner un résultat aléatoire quand on compare deux éléments identique. Autrement dit si $t[i] == pivot$ on place $t[i]$ dans Grand avec probabilité $1/2$ (une chance sur 2) et dans Petit avec probabilité $1/2$. Pour ce faire on peut utiliser (pex.) la méthode NextBoolean de la classe Java.util.Random, on

peut aussi utiliser NextFloat (nombre flottant aléatoire entre $[0, 1]$)et comparer le résultat avec 0.5.

On peut aussi supprimer les répétitions et associer dans un dictionnaire à chaque élément son nombre d'occurences.

Remarque : (ceci ne vous était pas demandé)

- La question 4 nous montre qu'à chaque appel récursif il y a une chance sur 2 que la taille du problème soit reduite d'un facteur $3/4$. Appelons ce cas un *bon cas* alors la loi des grands nombres implique que si on observe k appels récursifs on aura avec forte probabilité environ $k/2$ bons cas. En ce cas la taille n est multipliée par $(3/4)^k$ et vaut 1 pour $(3/4)^k\, n = 1$ soit $k = \log_{4/3}(n)$. On peut alors conclure que la complexité est en général $\Theta(n) \times \log_{4/3}(n) = \Theta(n \log 2(n)$.
- Un autre facon d'analyser l'algorithme est de compter le nombre de comparaisons effectuées. Les comparaisons sont uniquement effectuées quand on détermine les sous tableaux $Petit$ et $Grand$. Pour que deux éléments a et b, $a < b$ soient comparés il faut que
  - Que a ou b soit choisi comme pivot.
  - qu'aucun pivot situé entre a et b n'ai été choisi avant.

  Ainsi si le rang de a et i et celui de b est j on se note $P_c(i, j)$ la probabilité de comparer a et b. Du fait de la remarque ci-dessus $P(i, j)$ est aussi lr probabilité le premier pivot choisi dans $[i, j]$ soit i ou j. Comme il y a $j - i + 1$ élements dans l'intervale $[i, j]$ et deux cas favorables (choisir i ou j) cette probabilité vaut $\frac{2}{j-i+1}$. Ainsi si $j = i + 1$ on trouve i car il faut nécéssairement comparer deux éléments consécutif afin de terminer le tri.

  Pour poursuivre on compte simplement la moyenne du nombre total de comparaison :

$$sum_{i,j\in[0,n-1]^2,i<j} P(i, j) = sum_{i,j\in[0,n-1]^2,i<j} \frac{2}{j - i + 1}$$

  Enfin pour conclure on utilise le fait que

$$\sum_{i,j\in[0,n-1]^2,i<j} \frac{2}{j - i + 1} = \Theta(\log(n))$$

5 Recherche dichotomique dans un tableau

On considère la méthode Java `rechercheVite(int[] tab, int x)` donnée dans le listing de la figure 5 qui renvoie la position de x dans `tab` si $x \in$ `tab`, -1 sinon. Cette méthode s'applique uniquement à un tableau trié (ordre croissant).

1. Exécutez « à la main » la méthode `rechercheVite` du nombre 17 sur le tableau suivant :
   `int[] tab = {3, 6, 15, 21, 30, 33, 35, 40};`
   Quelles sont les valeurs des variables $x, gauche, droite$ et $milieu$ au cours de l'exécution de cette méthode ? Donnez un tableau avec une ligne par étape et une colonne par variable.
   Réponse

Index Gauche	Val Gauche	Index Droit	Val Droite	Index milieu	Val Milieu
0	3	7	40	3	21
0	3	2	15	1	6
2	15	2	15	2	15
3	21	2	15	2	15

2. Quel est le nombre minimum d'étapes pour la `rechercheVite` d'un entier x dans un tableau de taille n ? Donnez au moins un exemple.
   Réponse :
   Si on a de la chance l'élément que l'on recherche se trouve au milieu du tableau et l'algorithme se termine à la première étape. C'est le cas si dans le tableau donné en exemple on recherche 21.

3. Quel est le nombre maximum d'étapes pour la `rechercheVite` d'un entier x dans un tableau de taille n ? Donnez des exemples.
   Réponse :
   Notons n la longueur du tableau. L'algorithme assure que la condition suivante est maintenue : Si x se trouve dans le tableau il est entre $gauche$ et $droite$. Si nous notons $n_t = droite - gauche + 1$ la taille de ce tableau à l'étape t, il se trouve que si $k \geq 2$ alors k_t est au moins divisé par deux à chaque étape de l'algorithme. Plus précisément :

   (a) si $n_t = 2p + 1$ le tableau sera virtuellement coupé en un tableau de taille p (éléments avant milieu), le milieu (+1) et un second tableau de taille (éléments après milieu) et sauf si l'algorithme termine car milieu est la solution nous aurons $k_{t+1} = p$.

   (b) si $n_t = 2p$, le tableau sera virtuellement coupé en un tableau de taille $p - 1$ (éléments avant milieu), le milieu (+1) et un tableau de taille p (éléments après milieu) et sauf si l'algorithme termine car milieu est la solution nous aurons $k_{t+1} = p$.

   Quand $k_t = 1$, $gauche = droite = milieu$ et si x n'est pas égal à $tab[milieu]$ on aura lors de l'itération suivante : soit $droite = milieu - 1$, $gauche = milieu$ ou $droite = milieu, gauche = milieu + 1$. Dans ces cas nous n'aurons plus $gauche \leq droite$ ce qui indique un espace de recherche vide.

   Il faut donc au plus $\log 2(n)$ étapes pour que k_t devienne ≤ 1, plus l'étape supplémentaire quand $k = 1$. Ce qui donne $\log 2(n) + 1$. Comme chaque étape effectue un nombre constant d'opérations la complexité de la recherche dichotomique est $\Theta(\log 2(n))$.

   Donc après t itération $k \leq \frac{n}{2^t}$

4. Dans la méthode `rechercheVite` on remplace la ligne 10 « `droite = milieu - 1;` » par « `droite = milieu;` ». Expliquez quel problème de correction peut se poser. Par exemple, vous exécuterez la méthode `rechercheVite` du nombre 34 sur ce tableau :
   ```
   int[] tab = {3, 6, 15, 21, 30, 33, 35, 40};
   ```
   Réponse :
   Notons que l'assertion concernant la présence de l'élément dans l'intervale $[gauche, droite]$ reste vraie, le problème vient de la conditon de terminaison. Par exemple si $droite = gauche = milieu$ (à la toute fin de l'algorithme). En effet en ce cas

5. Montrer que si $n = 2^i$ alors la boucle 5 est effectuée au plus i fois. En déduire que le temps d'exécution est $Constante \times \log(n)$.
   Réponse : La taille de la partie du tableau est divisée par deux à chaque itération, quand le tableau est de taille 1 l algorithme termine en une itération. AU final on a donc au plus $\log 2(n)+1$ itérations. Pour plus voir question 3.

6. Qu'elle condition insvariante est maintenue dans le corps de la boucle "while"de la ligne 5.
   La condition est que l'élement recherché x vérifie $x \in tab[gauche, droite]$.

Addendum : On peut utiliser l'implémentation récursive suivante (ici en Python). Celle ci est **moins efficace**, non seulement de part les appels de fonction mais aussi parce que Python est interprété.

```java
public static int rechercheVite(int[] tab, int x) {
    int gauche = 0;
    int droite = tab.length - 1;
    int milieu;
    while (gauche <= droite) {
        milieu = (gauche + droite) / 2;
        if (x == tab[milieu])
            return milieu;
        if (x < tab[milieu])
            droite = milieu - 1;
        else
            gauche = milieu + 1;
        }
    return -1;
}
```

FIGURE 1 – Méthode rechercheVite en Java

Attention l'affichage de la liste coûte $\Theta(n)$ il faut donc le commenter une fois le principe compris. Cependant il n'y pas de problème avec la profondeur de l'arbre d'appel car celle-ci est logarithmique.

```python
def DichoFind(L,x,gauche,droite):
    print ([L[i] for i in range(gauche,droite+1)])
    if (gauche>droite):
        return(False)
    milieu= int((gauche+droite)/2)
    print(" milieu=" , milieu)
    if x== L[milieu] :
        return (milieu)
    elif (x< L[milieu]):
        return DichoFind( L,x, gauche,milieu-1)
    elif (x> L[milieu]):
        return DichoFind( L,x, milieu+1,droite)

def DichoSearch(L,x):
    return DichoFind(L,x,0, len(L)-1)
```

Attention aussi aux opérations cachées, par exemple dans la version suivante – quasi identique – le tableau est en partie récopié. les deux opérations générant les deux sous tableaux ($L[0, milieu]$,$L[milieu + 1 : len(L)]$) situées ligne 12 et 14 cachent une recopie qui va être de coût n.

```python
  def Dichorec(L,x):
    print ("L ", L)
    if len(L)==0:
        return(False)

    milieu= int(len(L)/2)
    print(milieu)
    if x== L[milieu] :
        return (milieu+shift)
    elif (x< L[milieu]):
        return Dichorec( L[0:milieu],x)
    elif (x> L[milieu]):
        return Dichorec( L[milieu+1:len(L)],x) +  milieu+1
```

6 Recherche du maximum dans un tableau

1. Écrivez la méthode Java `max(int tab[])` qui respecte les preconditions et postconditions données dans le listing suivant. L'algorithme attendu est un simple parcours linéaire du tableau.

```
/*
 * max
 *
 * Preconditions : tab est un tableau de n entiers quelconques
 *
 * Postconditions : le résultat est l'indice de l'élément le plus grand
 * de tab
 *
 */
public static int max(int[] tab) {

    // à compléter

    return max;
}
```

2. Soit un tableau d'entiers distincts ordonnés avec un premier segment croissant et un second segment décroissant. Trois exemples de tableaux de ce type sont : `{ 5, 8, 9, 11, 7, 6, 4 }`, `{ 5, 8, 9, 11 }` et `{ 7, 6, 4 }`.

   Écrivez la méthode Java `maxTrie(int[] tab)` qui respecte les preconditions et postconditions données dans le listing suivant. L'algorithme attendu est un simple parcours linéaire du tableau.

   Réponse :

```
import java.io.*;
import java.util.*;
class exo6
 {
     public static int Mymax(int[] tab)
     {
     int BestIdx=0;
     for(int i=0; i < tab.length; i++)
         {
             if (tab[i]> tab[BestIdx])
             {
             BestIdx=i;
             }

         }
     return BestIdx;
     }
 public static void main(String args[]) throws Exception
     {
     int[] t1 = {1, 2, 4, 6, 7, 12, 11, 8, 4, 1};
     System.out.println("t1 = " + Arrays.toString(t1) + " of length " + t1.
         length);
     System.out.println("max in t1 = " + t1[Mymax(t1)]);
     }
}
```

3. Écrivez la méthode Java `maxTrieDicho(int[] tab)` qui respecte les preconditions et postconditions données dans le listing suivant. L'algorithme attendu est un parcours dichotomique du tableau.

   Réponse :
   Il suffit d'adapter la dichotomie classique, mais on cherche pas un élément mais une inversion. Dans le parcours dichotomique quand l'espace des indices est $[gauche, droite]$ on doit déterminer si le "milieu" $tab[m]$ est avant ou après l'inversion. Si $tab[m+1] > tab[m]$ alors l'indice de l'inversion est dans $[m+1, droite]$, sinon il se trouve dans $[gauche, m]$.

4. Donnez des tests significatifs pour ces méthodes (tests unitaires Junit ou simple appels de méthodes depuis `main()`.

7 Fusion de deux tableaux triés

1. Écrivez la méthode `int[] fusionTrie(int[] tab1, int[] tab2)` qui respecte les preconditions et postconditions données dans le listing suivant et qui minimise le nombre d'étapes nécessaires à cette fusion.
2. Donnez ce nombre d'étapes en fonction de n_1 et n_2.

Réponse :
Il faut $\Theta(n_1 + n_2)$ étapes.

Dans le code les indices i_1, i_2 parcourent les deux listes. Le minimum de $tab1[i1], tab2[i2]$ (quand les deux éléments existent) est placé dans $tab3$. Les deux premiers "If" sont présents afin de gérer le cas où l'un des deux indices est arrivé à la fin de la liste. La condition "current <n1+n2" assure la bonne terminaison qui advient quand on a parcouru $n1 + n2$ éléments ; notez enfin que $current$ est incrémenté à chaque passage dans la boucle. Les deux premiers if assurent que lorsqu'ils sont faux alors $tab1[i1], tab2[i2]$ existent.

```
1  /*
2   *
3   * Preconditions : deux tableaux d'entiers triés, tab1 de taille
4   * n1 et tab2 de taille n2
5   *
6   * Postconditions : un tableau d'entiers trié tab3 de taille n1+n2
7   * contenant tous les éléments de tab1 et tab2
8   *
9   */
10 public static int[] fusionTrie(int[] tab1, int[] tab2) {
11   int[] tab3 = new int[tab1.length + tab2.length];
12   int n1=tab1.length
13   int n2=tab2.length
14
15   int i1=0; int i2=0
16   for (int current=0; current < n1+n2; current++)
17   {
18     if (i2>= n2)
19     {
20       tab3[current]=tab1[i1];
21       i1++;
22     }
23     else if (i1>= n1)
```

```
24      {
25        tab3[current]=tab2[i2];
26        i2++;
27      }
28      else if   (tab1[i1] <= tab2[i2])
29      {
30        tab3[current]= tab1[i1];
31        i1++
32      }
33     else
34      {
35      tab3[current]= tab2[i2];
36      i2++
37      }
38    }
39      return tab3;
40
41 }
```

8 Palindrome

Un palindrome est un mot qui se lit aussi bien de gauche à droite que de droite à gauche. Par exemple, « rever » et « ressasser » sont des palindromes, « carotte » n'est pas un palindrome. Notez que dans cet exercice on ne traite ni les accents ni la casse.

Complétez et tester la méthode donnée dans le listing suivant qui détermine si la chaîne de caractères c est un palindrome. Pour écrire cette méthode vous devez obligatoirement compléter les « ... » dans le code avec des opérations élémentaires sur les indices ou les caractères de la chaîne c (méthode `charAt(int i)` de la classe String).

Attention : à chaque étape de la boucle, vous devez respecter l'assertion A_1

Version bonus (uniquement si vous avez fini tous les exercices du TP) : modifiez la méthode `palindrome(c)` pour produire une version `phrasePalindrome(c)` qui ignore les séparateurs comme les espaces. Dans ce cas, les chaînes telles que « elu par cette crapule » ou « engage le jeu que je le gagne » sont valides (ainsi que les exemples de Georges Perec `http://homepage.urbanet.ch/cruci.com/lexique/palindrome.htm`).

Réponse :

L'algorithme compare simplement et itérativement le caractère $c[i]$ et le caractère symètrique $c[n-1-i]$ où $n = c.length$, et on peut s arreter quand les deux indices i et $j = n-1-i$ se croisent.

```
1  public boolean palindrome(String c) {
2    int i = 0;
3    int j = c.length()-1;
4    while ((i<j))
5    {
6      //  A1 : si C1 est la sous-chaîne de c allant de l'indice 0 à l'indice i et
7      //  si C2 est la sous-chaîne de c allant de l'indice j à c.length()-1
8      //  alors C1C2 est un palindrome
9      if (c[i]!=c[j])
10       {return False;}
11       i++;
12       j--;
13   }
```

```
14
15    return (True);
16 }
```

Pour répondre à la question bonus le plus simple est de supprimer les espaces ou les caractères de séparation. Par exemple en utilisant string.replace ou

```
string.replaceAll("\\s+'' , "")
```

qui remplace des expressions régulières.

9 Tri à bulle

Soit la méthode Java `triBulle` qui implante l'algorithme de tri donné ci-après.

1. Étudiez l'algorithme de la méthode `triBulle` en montrant le déroulement des étapes sur l'exemple du tableau `int[] tab = { 3, 25, 50, 8, 1, 3, 49 }`. Vous devez simplement donner les différentes valeurs de `tab`, étape par étape (entrée de la boucle `while`)
2. Complétez les lignes Preconditions, Postconditions ainsi que les assertions A1, A2 et A3 avec ce qui vous paraît le plus adapté pour nous convaincre de la terminaison et de la correction de cet algorithme. A1 concerne `tab[j]`, A2 et A3 concernent `tab` par rapport à l'indice j.
3. Améliorez l'algorithme pour qu'il termine dès qu'une itération (boucle `for`) sans échange est effectuée.
4. Donnez un exemple de tableau qui représente un pire cas possible pour ce dernier algorithme (qui maximise le nombre d'étapes).
5. Donnez un exemple de tableau qui représente un meilleur cas possible pour ce dernier algorithme (qui minimise le nombre d'étapes).
6. Est-ce que le tri à bulle vous semble meilleur que le tri par sélection présenté dans l'exercice 3 ?

```
1  // Preconditions : ..........................................................
2  // Postconditions : .........................................................
3  public static void triBulle(int[] tab) {
4      int j = tab.length - 1;
5      while (j > 0) {
6          for (int i = 0; i < j; i++) {
7              if (tab[i] > tab[i + 1]) {
8                  int tmp = tab[i];
9                  tab[i] = tab[i + 1];
10                 tab[i + 1] = tmp;
11             }
12         }
13         // A1 : ..................................................................
14         // A2 : ..................................................................
15         // A3 : ..................................................................
16         j--;
17     }
18 }
```

Attention à ne pas oublier la ligne 16 !

Réponses

L'algorithme est inspiré de bulles remontant à la surface, dans cette implémentation ce sont les éléments les plus grand qui montent.

- L'invariant est que la fin du tableau est triée. Plus précisément au début de la boucle externe (while) le sous tableau $]j, tab.length - 1]$ est correctement trié et contient les plus grands éléments.
- Le corps de la boucle fait remonter le maximum du sous tableau $[0, j]$ et va placer ce maximum en $tab[j]$. Ceci assure le maintient de la première condition.
- La ligne 16 incrémente de 1 la partie triée en fin du tableau.
- Pour améliorer l'algorithme il suffit d'introduire un *drapeau* stop mis à vrai entre lignes 5 et 6 et de mettre stop à faux si un échange est effectu´é. On peut alors ajouter entre les lignes 11 et 12 une ligne type "if stop return".
- Le cas le pire est celui d'une liste inversée (triée à l'envers) ou l'on va effectuer $\Theta(n + n - 1 + n - 2 + \ldots 1) = \Theta(n^2)$ opérations. Le cas le plus favorable est celui d'une liste déjà triée en ce cas le corps de la boucle est exécuté une seule fois et l'algorithme effectue $\Theta(n)$ opérations.

— Les deux tris : selection et à bulle (selection sort, bubble sort) ont des complexités dans le pire cas qui ont le même ordre de grandeur $\Theta(n^2)$, les différences de performance peuvent donc dépendre de l'implémentation. Le seul avantage du bubble sort est qu'il plus rapide sur les listes qui sont à peu près triées.