Algorithmes sur les tableaux

1. Algorithmes de base Parcours d'un tableau

```
Algorithme AfficheTableau (t : tableau)
{ Affiche tous les éléments d'un tableau }
Variable i : entier
Début
  Pour i ← 1 à taille(t) faire
    Écrire(t[i])
  Fin Pour
Fin
```

Recherche des plus petit et grand éléments d'un tableau

```
Algorithme Maximum (t : tableau d'entiers)
{ Recherche l'élément le plus grand d'un tableau de taille n non nulle }
Variables i, max : entier
Début
  max ← t[1]
  Pour i ← 2 à taille(t) faire
    Si (t[i] > max)
    Alors max ← t[i]
    Fin si
  Fin Pour
  Écrire(max)
Fin
```

Déroulement du programme sur le tableau
4 2 5 1 3

i	max
-	4
2	4
3	5
4	5

5 5
et l'algorithme affiche la valeur 5

Pour mesurer la complexité d'un algorithme, il faut tout d'abord désigner une ou plusieurs opérations élémentaires utilisées par l'algorithme. Dans le cas de la recherche du maximum d'un tableau, ces opérations pourront être :

- la comparaison entre le maximum connu et un élément du tableau (`t[i] > max`) ;
- l'affectation d'une valeur à la variable contenant le maximum (`max ← t[1]` et `max ← t[i]`).

Mesurer la complexité de l'algorithme revient alors à compter le nombre de fois où ces opérations sont effectuées par l'algorithme.

Ainsi, pour un tableau de taille *n*, l'algorithme *Maximum* effectue *n*-1 comparaisons.

Naturellement, nous le voyons avec l'algorithme *Maximum* et le nombre d'affectations qu'il effectue, la complexité peut varier avec les données fournies en entrée. Nous allons donc distinguer trois notions de complexité : au mieux, au pire et en moyenne.

- Le cas le plus favorable pour notre algorithme *Maximum* est le cas où le maximum du tableau se trouve en première position : la variable `max` prend cette valeur au début et n'en change plus ensuite. La complexité au mieux vaut donc 1.
- Au pire, le tableau est trié en ordre croissant et la variable `max` doit changer de valeur avec chaque case. Le complexité au pire vont donc *n*.
- La complexité en moyenne est plus difficile à calculer. Il ne s'agit pas comme on pourrait le penser de la moyenne des complexités au mieux et au pire. Nous pouvons observer que si nous choisissons un tableau au hasard, il est beaucoup plus probable d'avoir un tableau dont le maximum est en première place (cas le mieux) qu'un tableau complètement trié (cas le pire). Par conséquent, la complexité en moyenne est plus proche de 1 que de *n* et, en effet, il est possible de montrer que cette complexité en moyenne vaut $log(n)$.

En résumé, nous avons pour la complexité de l'algorithme *Maximum* en nombre d'affectations sur un tableau de taille *n* :

au mieux	**en moyenne**	**au pire**
1	$log(n)$	n

(variation) On cherche la position du minimum dans un tableau et entre deux cases repérées par leurs numéros *d*(début) et *f* (fin):

```
Algorithme PositionMinimum (t : tableau d'entiers ; d,f : entier)

Variable i,imin : entier

Début

  imin ← d

  Pour i ← d+1 à f faire :

    Si t[i] ≤ t[imin] alors

      imin ← i

    Fin si

  Fin pour
```

```
  Retourner imin
Fin
```

Existence d'un élément dans un tableau

Algorithme général :

```
Algorithme Recherche (e : entier ; t : tableau d'entiers)
{ Indique si l'élément e est présent ou non dans le tableau t }
Variable i : entier
Début
  i ← 1;
  Tant que (i ≤ taille(t)) et (t[i] ≠ e) faire
    i ← i+1
  Fin tant que
  Si (i>taille(t))
  Alors Écrire("l'élément recherché n'est pas présent")
  Sinon Écrire("l'élément recherché a été découvert")
  Fin si
Fin
```

Mais si les éléments du tableau sont ordonnés, nous pour vous pouvons tirer parti de cette caractéristique.

```
Algorithme RechercheO (e : entier ; t : tableau d'entiers)
Variable i  : entier
         trouve : booléen
Début
  i ← 1
  Tant que (i ≤ taille(t)) et (t[i] < e) faire:
    i ← i+1
  Fin TQ
  Si (i ≤ taille(t)) et (t[i]=e) alors
    Écrire('oui')
  Sinon
    Écrire('non')
  Fin si
Fin
```

ou encore mieux avec une recherche dite *dichotomique* :

```
Algorithme RechercheDichotomique
(e : entier ; t : tableau d'entiers)
Variable g,d,m  : entier
         trouve : booléen
Début
  g ← 1
  d ← taille(t)
  trouve ← faux
  Tant que (g ≤ d) et NON(trouve) Faire
    m = (g+d) div 2
    Si t[m] = e
    Alors trouve ← vrai
    Sinon Si e < t[m]
          Alors d ← m-1
          Sinon g ← m+1
          Fin si
    Fin si
  Fin Tant que
  Si trouve
  Alors Écrire('Trouvé !')
  Sinon Écrire('Pas trouvé...')
  Fin si
  Écrire(m)
Fin
```

Pour continuer à bénéficier de ces algorithmes, il faut être capable d'insérer un nouvel élément directement à sa place (ici *n* indique le numéro de la dernière case):

```
Algorithme Insertion (t : tableau d'entiers ; n,e : entier)
Variable i : entier
Début
  i ← n
  Tant que (i>0) et (t[i]>e) faire :
    t[i+1] ← t[i]
    i ← i-1
```

```
  Fin TQ
  t[i+1] ← e
Fin
```

2. Algorithmes de tri de tableaux

Algorithme d'échange

```
Algorithme Échange (t : tableau d'entiers ; i,j : entiers)
{ Échange le contenu des cases i et j dans le tableau t }
Variable pro : entier
Début
  pro  ← t[i]
  t[i] ← t[j]
  t[j] ← pro
Fin
```

Tri insertion

```
Algorithme TriInsertion (t : tableau d'entiers)
{ Trie par ordre croissant le tableau t }
Variable i : entier
Début
  Pour i ← 2 à taille(t) faire
    Insertion(t,i-1,t[i])
  Fin Pour
Fin
```

Tri extraction

Aussi nommé *tri sélection*, il utilise l'algorithme *PositionMinimum*

1. Extraire l'élément le plus petit du tableau à trier.
2. Échanger cette valeur minimale avec la première case du tableau à trier.
3. Trier le reste du tableau (le tableau initial sans la première case) de la même manière.

```
Algorithme TriExtraction (t : tableau d'entiers)
{ Trie par ordre croissant le tableau t }
Variables i,imin : entier
Début
  Pour i ← 1 à taille(t)-1 faire
```

```
    imin ← PositionMinimum(t,i,taille(t))
    Échange(t,i,imin)
  Fin Pour
Fin
```

Tri à bulles

```
Algorithme TriBulles (t : tableau d'entiers)
{ Trie par ordre croissant le tableau t contenant n éléments }
Variables i,j : entier
Début
  Pour i ← 1 à taille(t)-1 faire
    Pour j ← 1 à taille(t)-i faire
      Si t[j] > t[j+1]
      Alors Échange(t,j,j+1)
      Fin Si
    Fin Pour
  Fin Pour
Fin
```

Principe du tri rapide

1. Choisir un élément du tableau, élément que l'on nomme ensuite *pivot*.
2. Placer le pivot à sa position finale dans le tableau : les éléments plus petits que lui sont à sa gauche, les plus grands à sa droite.
3. Trier, toujours à l'aide de cet algorithme, les sous-tableaux à gauche et à droite du tableau.

Pour que cette méthode soit la plus efficace possible, il faut que le pivot coupe le tableau en deux sous-tableaux de tailles comparables.

Ainsi, si l'on choisit à chaque le plus petit élément du tableau comme pivot, on se retrouve dans le cas de l'algorithme de tri par extraction : la taille du tableau de diminue que d'un à chaque alors que le but est de diviser cette taille par deux.

Cependant, bien choisir le pivot peut être coûteux en terme de complexité.

Aussi on suppose que le tableau arrive dans un ordre aléatoire et on se contente de prendre le premier élément comme pivot.

```
Algorithme Placer (t,d,f):
Variables l,r : entiers
Début
  l ← d+1
  r ← f
```

```
  Tant que l ≤ r:
        Tant que t[r]>t[d]
                r ← r-1
        Fin TQ
        Tant que l ≤ f et t[l] ≤ t[d]
                l ← l+1
        Fin TQ
        Si l<r:
                Échange(t,l,r)
                r ← r-1
                l ← l+1
        Fin si
  Fin TQ
  Échange(t,d,r)
  Renvoyer r
Fin

Algorithme TriRapideBoucle (t,d,f)
Variable k : entier
Début
  Si d<f alors
        k ← Placer(t,d,f)
        TriRapideBoucle(t,d,k-1)
        TriRapideBoucle(t,k+1,f)
  Fin si
Fin

Algorithme TriRapide (t,n)
Début
  TriRapideBoucle(t,1,n)
Fin
```