Corrigé série 2 : Complexité des algorithmes

Exercice 1: *Complexité en fonction de deux paramètres*

Déterminer la complexité des algorithmes suivants (par rapport au nombre d'itérations effectuées), où m et n sont deux entiers positifs.

Algorithme A

$i \leftarrow 1$; $j \leftarrow 1$

tant que $(i \leq m)$ *et* $(j \leq n)$ faire

 $i \leftarrow i+1$

 $j \leftarrow j+1$

fin tant que

O(min(m,n))

Algorithme B

$i \leftarrow 1$; $j \leftarrow 1$

tant que $(i \leq m)$ *ou* $(j \leq n)$ faire

 $i \leftarrow i+1$

 $j \leftarrow j+1$

fin tant que

O(max(m,n))

Algorithme C

$i \leftarrow 1$; $j \leftarrow 1$

tant que $(j \leq n)$ faire

 si $i \leq m$

 alors

 $i \leftarrow i+1$

sinon

 $j \leftarrow j+1$

fin si

fin tant que

O(m+n)

Algorithme D

$i \leftarrow 1$; $j \leftarrow 1$

tant que $(j \leq n)$ faire

 si $i \leq m$

 alors

 $i \leftarrow i+1$

sinon

 $j \leftarrow j+1$; $i \leftarrow 1$

fin si

fin tant que

O(mxn)

Exercice2 :

Déterminer un algorithme qui teste si un tableau de taille n est un "tableau de permutation" (i.e. tous les éléments sont distincts et compris entre 1 et n).

1. Donner un premier algorithme naïf qui soit quadratique.
2. Donner un second algorithme linéaire utilisant un tableau auxiliaire.
3. Donner un troisième algorithme linéaire sans utiliser un tableau auxiliaire.

1/ fonction TableauDePermutation(E/ t :tableau[1..n] d'entiers;E/n :entier) :booleen

```
i, j :entier ; b :booleen ;
debut
i:=1 ; b :=vrai ;
tantque (i<=n) et (b=vrai) faire
     si (t[i]<1) ou (t[i]>n) alors b :=faux ;  finsi ;
    i :=i+1 ;
fintq ;
i :=1 ;
tantque (i<=n-1) et (b=vrai) faire
       j :=i+1 ;
      tantque (j<=n) et (b=vrai) faire
              si (t[i]=t[j]) alors b :=faux
             sinon j :=j+1 ;
            finsi ;
     fintq ;
   i :=i+1;
fintq ;
retourner b ;
fin ;
```

Complexité : $O(n^2)$ solution quadratrique

```
2/ fonction TableauDePermutation(E/ t :tableau[1..n] d'entiers;E/n :entier) :booleen
i, j :entier ; b :booleen ; temp :tableau[1..n] d'entiers ;
debut
pour i :=1 à n faire temp[i]=0 ; finpour ;    ---------- O(n)
i:=1 ; b :=vrai ;                                         +
tantque (i<=n) et (b=vrai) faire              ---------  O(n)
     si (t[i]<1) ou (t[i]>n) alors b :=faux ;  finsi ;
    i :=i+1 ;
fintq ;                                                   +
pour i :=1 à n faire temp[t[i]]=t[i] ;  finpour ;  ---- O(n)
i :=1 ;                                                   +
tantque (i<=n)  et (b=vrai) faire      ----------------- O(n)
          si (temp[i] ≠i) alors b :=faux   // ou bien si temp[i]=0
         sinon i :=i+1 ;
        finsi ;
 fintq ;
retourner b ;
fin ;
```

Complexité : $O(4n) \approx O(n)$ solution linéaire avec un tableau supplémentaire

```
3/ fonction permutation(E/ T : tableau [1..n] d'entiers ; n :entier) :booleen ;
      i, x :entier ;  b:booleen;
```

```
debut
 i:=1 ; b :=vrai ;
tantque (i<=n) et (b=vrai) faire
    si (T[i]<1) ou (T[i]>n) alors b :=faux ;  finsi ;
   i :=i+1 ;
fintq ;
 i :=1
tantque (i<=n) et (b=vrai) faire
   si (T[T[i]]=T[i]) alors  b :=faux    finsi ;
   sinon si (T[i] ≠ i) alors
                        x=T[i] ;
                        T[i]=T[T[i]] ;
                        T[T[i]]=x
            sinon i=i+1 ;
            finsi ;
   finsi ;
fait ;
Retourner b ;
fin;
```

Complexité : O(n) solution linéaire sans utiliser un deuxième tableau

```
3b/ fonction permutation(E/ T : tableau [1..n] d'entiers ; n :entier) :booleen ;
     i, x :entier ;  b:booleen;
debut
```

```
 i:=1 ; b :=vrai
tq (i<=n) et (b=vrai) faire
   si (T[i]<1) ou (T[i]>n) alors b :=faux  finsi
  i :=i+1
fait
 i :=1
tq (i<=n) et (b=vrai) faire
      si (T[i]]=i) alors  i :=i+1
      sinon
            si (T[i]=T[T[i]]) alors b=FAUX
            sinon si (i<T[i] et T[i]>T[T[i]]) ou (i>T[i] et T[i]<T[T[i]]) alors
                  x=T[i]
                  T[i]=T[T[i]]
                  T[T[i]]=x
            finsi
      finsi
fait
retourner b
fin
```

Complexité : O(n) solution linéaire sans utiliser un deuxième tableau

Exercice 3 : *Produit matriciel*

On considère deux matrices carrées (d'entiers) d'ordre n, A et B. Le produit de A par B est une matrice carrée C définie par :

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$$

1. Donner un algorithme calculant le produit de deux matrices représentées sous forme d'un tableau à deux dimensions. Calculer la complexité de cet algorithme.
   Doit-on préciser dans quels cas (pire cas, meilleur des cas, cas moyen) cette complexité est obtenue ?
2. Modifier l'algorithme précédent lorsque la matrice A est de dimension (m,n) et la matrice B de dimension (n, p). Quelle est alors la complexité de l'algorithme ?

Corrigé :

1/ i, j, k :entier ;
pour i :=1 à n faire
  pour j :=1 à n faire

```
    c[i,j]=0 ;
    pour k :=1 à n faire
        c[i,j]=c[i,j]+A[i,k]*B[k,j] ;
    finpour
  finpour
finpour ;
```

Complexité est de $\Theta(n^3)$. Il n'y a donc ici ni meilleur cas ni pire cas.

2/ i, j, k :entier ;

```
pour i :=1 à m faire
  pour j :=1 à p faire
    c[i,j]=0 ;
    pour k :=1 à n faire
        c[i,j]=c[i,j]+A[i,k]*B[k,j] ;
    finpour
  finpour
finpour ;
```

Complexité de l'algorithme est de $\Theta(npm)$ et croît linéairement par rapport à la dimension de chacune des deux matrices d'entrée prises individuellement. Il à noter cependant que cet algorithme ne sera pas globalement qualifié de « linéaire »

<u>Exercice 4</u> : Tri d'un tableau ne contenant que des 0 et des 1

On souhaite trier un tableau T de n entiers appartenant à l'ensemble {0,1} de façon à ce que les valeurs nulles soient rangées au début du tableau. Par exemple:

T=

0	1	1	0	0	…	1	1	1

Après l'application de l'algorithme de tri on aura :

T=

0	0	0	0	1	…	1	1	1

Au cours du traitement, une partie des données est déjà triée et le tableau est organisé de la façon suivante :

T=

1			i			j				n
0	0	0	?	?	?	?	1	1	1	1

Les ? représentent les données non encore traitées.

1. Que représentent i et j ?
2. Selon la valeur de T[i] quel traitement doit-on effectuer ?
3. Quand doit-on arrêter l'algorithme ?
4. Écrire l'algorithme de tri et donner sa complexité en temps.

Corrigé :

1. i et j représentent l'intervalle des positions des éléments non encore triés.
2. Si T[i]=0 on incrémente i sinon si T[i]=1 on le remplace par T[j] et T[j]=1
3. Arrêt quand i>=j
4. *Procedure Tri(ES/ T :tableau [1..n] d'entiers ; E/n :entier)*

```
i, j :entier ;
Debut
  i:=1 ; j:=n ;
tantque (i<j) faire
    si (T[i]=0) alors i :=i+1
    sinon T[i] :=T[j] ; T[j] :=1 ; j :=j-1 ;
    finsi ;
fintq ;
Fin ;
```

Exercice 5 : *Interclassement de deux tableaux triés*

On dispose de deux tableaux *T1[1..n]* et *T2[1..n]* dont les éléments sont triés de façon croissante. On veut créer un tableau trié *T3[1..2n]* contenant tous les éléments de *T1* et *T2*. Pour cela on propose deux algorithmes Fusion_A et Fusion_B.

Fusion_A : initialise T3 avec T1 (déjà trié) et y insère un à un les éléments de T2 de façon à ce que l'ordre soit respecté.

Fusion_B : remplit T3 en parcourant simultanément T1 et T2 du début jusqu'à leur fin. Soit i1 et i2 les indices courant dans T1 et T2, on a 3 cas possible :

Si T1[i1]<T2[i2] alors mettre T1[i1] à la fin de T3 et avancer dans T1
Si T1[i1]>T2[i2] alors mettre T2[i2] à la fin de T3 et avancer dans T2
Sinon mettre T1[i1] puis T2[i2] à la fin de T3 et avancer dans T1 et T2

1. Ecrire les deux algorithmes et déroulez sur l'exemple :

   T1= | 1 | 3 | 5 | et T2 = | 2 | 3 | 4 |

2. Donnez la complexité, au pire des cas, des algorithmes en fonction de la taille des données.
3. Quel algorithme choisissez-vous d'implémenter ?

Corrigé :

```
Action  Fusion_A(E/ t1, t2 :tableau[1..n] d'entiers ;E/n :entier ;
                                        S/ t3 :tableau[1..2n]d'entiers)
i, j, k,m :entier ;
Debut
Pour i :=1 à n faire t3[i]=t1[i] ; finpour ;
```

```
m:=n ;
Pour j :=1 à n faire
    i :=1 ;
    tantque (i<=m et t3[i]<t2[j]) i :=i+1 ; finTq//  recherche de la position
    k :=m ;
    tantque (k>=i) faire t3[k+1] :=t3[k] ; k :=k-1 ; finTq ; // décalage à droite
    t3[i] :=t2[j] ;
Finpour ;
Fin ;
```

Complexité de Fusion_A est de l'ordre de $O(n^2)$

```
Action  Fusion_B(E/ t1, t2 :tableau[1..n] d'entiers ;E/n :entier ;
                                        S/ t3 :tableau[1..2n]d'entiers)
i, j, k,m :entier ;
Debut
    i :=1; j :=1; k :=1 ;
    tantque (i<=n et j<=n) faire
        si (t1[i]<t2[j]) alors t3[k] :=t1[i]; i :=i+1;
        sinon si (t1[i]>t2[j]) alors t3[k] :=t2[j] ; j :=j+1 ;
                sinon t3[k] :=t1[i]; k:=k+1; t3[k]:=t2[j]; i:=i+1; j:=j+1 ;
                finsi ;
        finsi ;
      k :=k+1 ;
    finTq;
    tantque (i<=n) faire t3[k] :=t1[i] ; k :=k+1; i :=i+1 ; finTq ;
    tantque (j<=n) faire t3[k] :=t2[j] ; k :=k+1; j :=j+1 ; finTq ;
Finpour ;
Fin ;
```

Complexité de Fusion_B est de l'ordre de $O(n)$

On choisira l'algorithme Fusion_B car il a une complexité linéaire.

<u>Exercice 6</u> : Recherche séquentielle

On considère un tableau *A* de *n* éléments, que l'on suppose trié en ordre croissant.

On cherche à construire un algorithme permettant de savoir à quel endroit du tableau se trouve une valeur *clé*. (On suppose que *clé* est bien dans le tableau et on recherchera la première occurrence de cette valeur.)

1. Donner un algorithme itératif qui résout ce problème. Indiquer et démontrer un invariant de boucle pour cet algorithme.
2. A quoi correspond le pire des cas ? En déduire la complexité en *O* de l'algorithme.
3. A quoi correspond le meilleur des cas ? En déduire la complexité en *O* de l'algorithme.
4. Ecrire cet algorithme sous forme récursive et l'exécuter sur le tableau suivant avec *clé*=18.

1	7	8	9	12	15	18	22	30	31

5. Prouver l'algorithme récursif.
6. Comment faut-il modifier ces versions (itérative et récursive) de l'algorithme si l'on n'est pas sûr que *clé* appartienne au tableau ?

Corrigé :

1. *fonction Rech_seqIt(E/ A :tableau[1..n]d'entiers ;E/ n, clé :entier) :entier*

   pos :entier ;

   debut

   pos :=1 ;

   tantque (A[pos]≠clé) faire pos :=pos+1 ; fintq ;

   retourner pos ;

   fin ;

Il est à noter que l'algorithme ne tire pas profit du fait que le tableau soit trié.

Il faut d'abord démontrer que cet algorithme se termine. Or si l'on suppose que clé appartient bien au tableau A :

$$\exists k \leq n | A[k] = clé\ et\ \forall j < k, A[j] \neq clé$$

L'algorithme, par construction parcourt tous les éléments du tableau, en partant du premier, tant que A[pos]≠ *clé*. Il s'arrête donc pour pos=k.

Si pour tout i $\in \{1, \dots, n\}$, pos_i désigne la valeur de pos à la fin de la i-ième itération, un invariant de boucle est :

« A la fin de la i^{ème} itération, soit A[i]=clé et la fonction se termine, soit pos_i=i+1 et ∀k ∈ {1,…,i}, A[k]≠clé »

On montre sa validité par récurrence sur i.

L'invariant est facilement vérifié pour i=1. Si l'on suppose cette propriété vraie à la fin de l'itération i, alors si il y a une itération i+1, c'est que pos_i=i+1 et ∀k∈{1,…,i}, A[k]≠clé. Au début de l'itération i+1, on a soit A[i+1]=clé et la boucle se termine, soit A[i+1]≠clé et pos est incrémentée. L'invariant à la fin de la boucle i+1 est donc vérifié.

2. Le pire des cas correspond à une valeur clé en dernière position du tableau. En effet, l'algorithme doit alors parcourir tout le tableau pour la trouver. Il est clair que le nombre d'itérations (donc de comparaisons de clés) est alors égal à n. On en déduit que l'algorithme est en O(n).

3. Le meilleur des cas correspond à une valeur clé en première position du tableau. L'algorithme n'effectue alors qu'une seule comparaison de clés. On en déduit que l'algorithme est en O(1).

4. *fonction Rech_recur(A :tableau ; n, clé, pos :entier) : entier ;*

```
Début                                  //1er appel pos=1
   si (A[pos]=clé) alors  retourner(pos)
   sinon  retourner(Rech_recur(A, n, clé, pos+1)) ;
   finsi ;
fin ;
```

Il est cependant peu satisfaisant de voir une fonction récursive se rappeler avec un paramètre plus grand. On peut donc transformer cette fonction de la façon suivante :

```
fonction Rech_recur(A :tableau ; n, clé, k :entier) : entier ;
Début
   si (A[n-k+1]=clé) alors  retourner(n-k+1)
   sinon  retourner(Rech_recur(A, n, clé, k-1)) ;
   finsi ;
fin ;
```

Le paramètre k de la fonction correspond au nombre d'éléments restant à tester. L'appel initial de la fonction est alors Rech_recur(A, 10, 18, 10) et l'exécution est :

Rech_recur(A, 10, 18, 10) comme A[1]<18

    Rech_recur(A, 10, 18, 9) comme A[2]<18

    …

        Rech_recur(A, 10, 18, 4) comme A[7]=18

        retourne 7

    …

    retourne 7

retourne 7

Preuve : On montre par récurrence sur ***k*** $\in$ ***{1, …, n}*** que si ***clé*** $\in$ ***A[n-k+1 . . n]***, alors rech_recur renvoie le plus petit indice ***i*** $\in$ ***{n-k+1, …, n}*** tel que ***A[i]=clé***.

Pour ***k=1***, on a alors ***A[n]=clé*** et la propriété est vérifiée. Supposons maintenant que la propriété est vérifiée pour une valeur ***k<n*** et que ***clé*** $\in$ ***A[n-k . . n]***. On a alors deux cas :

- si ***clé*** $\in$ ***A[n-k+1 . . n]***, alors par hypothèse de récurrence, la fonction renvoie le plus petit indice ***i*** $\in$ ***{n-k+1,…,n}*** tel que ***A[i]=clé***.
- sinon, ***A[n-k]=clé*** et la fonction renvoie ***n-k***.

On en déduit que, si ***clé*** $\in$ ***A[n-k . . n]*** alors Rech_recur renvoie le plus petit indice

i $\in$ ***{n-k, …, n}*** tel que ***A[i]=clé***. La propriété est donc vérifiée.

Exercice 7 : Recherche dichotomique

On se place dans les mêmes conditions que celles de l'exercice précédent (*A* est un tableau trié de *n* éléments).

1. Donner un algorithme itératif qui recherche une clé par la méthode dichotomique.
2. Développer cet algorithme sur le tableau suivant avec *clé* = 30.

1	7	8	9	12	15	18	22	30	31

3. Indiquer et démontrer un invariant de boucle pour cet algorithme.

4. On suppose que le tableau *A[1..n]* contient n=2^k éléments (où *K* est un entier positif). Combien d'itérations l'algorithme effectuera-t-il au maximum ?
5. En déduire la complexité (en *O*) de l'algorithme.
6. Pour *k*=100 comparer les complexités des algorithmes de recherche séquentielle et dichotomique.
7. Ecrire l'algorithme sous forme récursive et l'exécuter sur l'exemple de la question 1.
8. Déterminer la complexité (en *O*) de l'algorithme récursif.
9. Comment faut il modifier ces versions (itérative et récursive) de l'algorithme si l'on n'est pas sûr que *clé* appartienne au tableau ?

***Exercice 8* : Tri sélection**

Le tri sélection d'un tableau T[1..n] de n éléments consiste, pour i variant de 1 à n-1, à déterminer l'élément minimum du sous-tableau T[i..n] et à échanger cet élément avec T[i].

a. On note T[d..f] le sous-tableau de T compris entre les indices d et f. Ecrire une fonction itérative *rech_min(T, d, f)* qui retourne l'indice du plus petit élément de T[d..f]. Prouver la terminaison et la validité de cette fonction.

b. Déterminer la complexité de la fonction *rech_min*.

c. On considère la procédure suivante :

```
Procédure  Tri_Sélection(T : tableau ; n :entier)
{  i, k : entier ;
      pour i :=1 à n-1 faire
            k=rech_min(T, i, n) ;
            si (i ≠ k) alors échanger (T[i], T[k])  finsi;
     fait ;  }
```

Déterminer la complexité de la procédure *Tri_sélection*.

Corrigé :

a. *Fonction Rech_min(E/t :tableau[d..f] d'entiers ; E/ d, f :entier) :entier*
pos :entier ;
Debut
Pos :=d ;
Pour i :=d+1 à f faire
si (t[i] < t[pos]) alors pos :=i ; finsi ;
Fait ;
retourner pos ;
Fin ;

Terminaison : elle est triviale puisque si d<=f la boucle est exécutée f-d fois sinon 0 fois aussi l'instruction conditionnelle (si) ainsi que l'affectation sont des instructions finies donc la boucle pour se termine.

Invariant : *« A la fin de l'itération i, pos contient l'indice du plus petit élément de t[d..i] »*

Preuve par récurrence :

« A l'itération 1 c'est-à-dire i=d+1, si t[i]<t[pos] pos_1=d+1, donc pos_1 contient l'indice du plus petit élément de t[d..d+1] » donc vrai

On suppose que l'invariant est vrai à l'itération i et on montre qu'il est vrai à l'itération i+1.

« A l'itération i+1, si t[i+1]<t[pos] pos_{i+1}=i+1 et conservera sa valeur sinon, pos contient donc bien l'indice du plus petit élément de t[d..i+1] »

Il en résulte que l'invariant est vrai. La fonction Rech_min(t,d,f) retourne bien l'indice du plus petit élément de t[d..f].

b. Complexité : Rech_min est linéaire et égale à O(f-(d+1)+1)=O(f-d) au pire cas et même chose au meilleur cas donc précisément la complexité de Rech_min est de Θ(f-d)

c. La complexité de la procédure *Tri_sélection* est au pire cas de l'ordre de O(n^2) car :

- quand *i=1* *Rech_min* s'exécute en *n-1* itérations
  " *i=2* " *n-2* "
  . " .
  . .
  " *i=n-1* " *n-(n-1)=1* "

  Ce qui correspond à $\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ = O(n^2)
- La fonction *échanger* est de l'ordre de O(1)

Donc la complexité de la fonction *tri_sélection* est au pire cas de l'ordre de O(n^2).