# Algorithmique avancée et Complexité (1h30) Corrigé

**Exercice 1 : (08 pts) (2, 2, 2, 2)**

1. Déterminer l'invariant des différentes boucles et en déduire l'ordre de complexité des algorithmes suivants :

```
Algo_A1()
Début
x=0 ;
i=1 ;
Tant que (i*i < n)
Faire   x= x+i ;
        i=i*2 ;
fait ;
Fin.
```

```
Algo_A2()
Début
x=0 ;
i=1 ;
Tant que (i*i < n)
Faire   x= x+i ;
        i=i+1 ;
fait ;
Fin.
```

```
Algo_A3 (N)
Début
Si N < B alors Retourner (-1) ;
        Sinon Si N = B
                Alors Retourner (0) ;
                Sinon
                Retourner ( A3 (N – A)) ;
                Fsi ;
         Fsi ;

Fin.
```

Avec A, B deux entiers qui peuvent être positifs ou négatifs.

*Algorithme A1 :*

Boucle tant qui a le comportement suivant :

- Borne Inferieure : i=1.
- Borne supérieures : i*i >= n ➔ i >= sqrt(n).
- Pas = i=i*2, pas clairement défini mais il n'est pas constant et est croissant en fonction de i donc on ne peut pas utiliser la formule pour déterminer le nombre d'itérations.

Nous allons donc déterminer le nombre d'itérations autrement en déroulant :

$1^{ère}$ itération : i=1 ➔ i= 2 = $2^1$

$2^{ème}$ itération : i=2 ➔ i=4= $2^2$

$3^{ème}$ itération : i=4 ➔ i=8 = $2^3$

A la kième itération : i= $2^{k-1}$ ➔ $2^k$ **(1 pt pour l'analyse)**

Pour sortir de la boucle, il faut que i*i>= n. Donc pour sortir de la boucle au bout de k itération il suffit de considérer i=$2^k$ **(0,5)**

Ainsi, i*i = $2^k$*$2^k$ = $2^{2*k}$ ; i*i >= n ➔ i >= sqrt(n) ➔ $2^k$ >= sqrt(n) ➔ k >= ½ $\log_2(n)$.

Donc la complexité de l'algorithme A1 est de l'ordre de O(log n). **(0,5)**

*Algorithme A2 :*

Boucle tant qui a le comportement suivant :

- Borne Inferieure : i=1.
- Borne supérieure : i*i >= n ➔ i >= sqrt(n).

Pas = i=i+1. Pas clairement défini et constant. **(1 pt pour l'analyse)**

Donc, le nombre d'itération peut directement être calculé comme suit :

(Borne supérieure – Borne inférieure)/ pas = sqrt(n) **(0,5)**

Ainsi, la complexité est de l'ordre de O(sqrt(n)). **(0,5).**

*Algorithme A3 :* **(1 pt pour l'analyse, (0,5) pour la résolution de l'équation de récurrence et (0,5) pour le résultat).**

Fonction récursive qui dépend de A et B. Plusieurs situations possibles :

- A < 0 et B < 0 : N étant positif à la base donc N est toujours supérieur à B. Ceci d'une part, d'autre part, la récursivité est en fonction de N – A= N -- |A| = N + |A|. Ce qui signifie qu'au fil des appels récursifs N ne va pas cesser de croitre et ne pourra jamais atteindre la condition d'arrêt. Dans ce cas la terminaison de l'algorithme est impossible.
- A < 0 et B >= 0. Deux cas possibles dans cette situation :
  - Si N > B alors terminaison impossible parce que la condition d'arrêt ne sera jamais atteinte comme la situation précédente.
  - Si N <= B, alors on est directement dans la condition d'arrêt, donc la complexité dans ce cas est O(1).
- A > 0 et B > 0 :
  - N <= B ➔ O(1) comme dans la situation précédente.
  - N > B. Dans ce cas, la fonction récursive sera appelé k fois jusqu'à atteindre une valeur de n <= B.

  F(N)= $\alpha\, F(N-A) + \beta, Si\ N > B$

  F(N) = $\beta, sinon$. Avec $\alpha et\ \beta$ deux entiers positifs qui représente respectives le coût d'un appel récursif et le coût de la sortie de l'algorithme.

  Dans notre algorithme : $\alpha = 1\ et\ \beta = 1$.

  Il suffit de résoudre d'équation de récurrence pour déterminer le nombre d'appel recursif k en fonction de A et B et ainsi en déduire l'ordre de complexité.

  F(N)= F(N- A)+ 1 = F(N- A – A) + 2 = …. = F(N – k*A) + k.

  Pour sortir de la fonction il suffit que N –k*A <= B

  Donc, $k \leq \frac{N-B}{A}$ ; puisque A et B < N donc la complexité est de l'ordre de O(N).

- A> 0 et B <= 0. Idem que la situation précédente et la complexité est de l'ordre de O(N).

2. Montrer que pour tout entier k > 0 on a : $\sum_{i=1}^{n} i^k = \emptyset(n^{k+1})$

Il s'agit de démontrer que :

$\exists C1, C2 > 0, \exists n0 \geq 0, C1 * n^{k+1} \leq \sum_{i=1}^{n} i^k \leq C2 * n^{k+1}$ **(0,5 : formalisme de Landau)**

Pour la borne inferieure : $C1 * n^{k+1} \leq \sum_{i=1}^{n} i^k$ **(0,75 : 0,5 calcul + 0,25 résultat)**

$C1 * n^{k+1} \leq 1^k + 2^k + \cdots + n^k$, *Si on ignore la première moitié on obtient*:

$$\underbrace{\left\lceil \frac{n}{2} \right\rceil^k + \left\lceil \frac{n}{2} \right\rceil^k + \cdots + \left\lceil \frac{n}{2} \right\rceil^k}_{\text{N/2 fois}} \leq 1^k + 2^k + \cdots + \underbrace{\left\lceil \frac{n}{2} \right\rceil^k + \cdots + n^k}$$

$\left\lceil \frac{n}{2} \right\rceil^k + \left\lceil \frac{n}{2} \right\rceil^k + \cdots + \left\lceil \frac{n}{2} \right\rceil^k = \frac{n}{2} * \left\lceil \frac{n}{2} \right\rceil^k = \frac{1}{2^{k+1}} * n^{k+1}$. D'où C1= $\frac{1}{2^{k+1}}$

Pour la borne supérieure il faut que : $\sum_{i=1}^{n} i^k \leq C2 * n^{k+1}$ **(0,75 : 0,5 calcul + 0,25 résultat)**

$1^k + 2^k + \cdots + n^k \leq C2 * n^{k+1}$.

Il est évident que : $1^k + 2^k + \cdots + n^k \leq C2 * (n^k + n^k + \cdots + n^k) \leq C2 * n * n^k$

$\leq C2 * n^{k+1}$.

Il suffit donc de considérer C2= 1 et n0= 1.

**Exercice 2 : (12 pts) (2, (4, 2, 1), 3)**

Soit en entrée un texte sous forme d'une chaine de caractères de longueur N (ou un tableau de caractères de dimension N). On cherche à déterminer si un mot donné X de longueur M est caché dans ce texte. Un mot peut être caché dans un texte de manière directe (sous la forme d'une sous-chaine) ou de manière indirecte, c'est-à-dire, que les caractères qui forment le mot X ne sont pas contigus mais espacés d'une distance fixe k (à déterminer) dans le texte.

**Exemple :** 1er cas : gsiqidbk**usthb**jhasdladh 2ème cas : fcsah**u**hgs**s**jhq**t**jsl**h**sfd**b**qjqlu

1) Expliquer à l'aide de l'exemple comment déterminer l'existence ou non du mot caché.
   - Parcours séquentiel de 1 à N-M à la recherche du premier caractère du mot, s'il n'existe pas alors le mot n'est pas caché dans le texte. Sinon, à partir de la position où est apparu le premier caractère en continue de chercher le caractère suivant (2ème caractère) qui peut être juste après ou bien éloigné de k positions. Donc, à partir de la position suivante on continue le parcours séquentiel mais cette fois-ci à la recherche du second caractère, s'il est rencontré on définit la distance k qu'on va utiliser pour

chercher les caractères suivants. Si tout le mot est retrouvé dans le texte on dira que le mot est bel est bien caché dans le texte. Autrement, on reprend la recherche du premier caractère à partir de la position qui suit le premier caractère précédemment trouvé. **(2 pts)**

2) Proposer un algorithme itératif (ou récursif) permettant de répondre au problème posé.

```
Algorithme :
    Début
    i= 1 ; h= 1 ; trouve = faux;
    T[N] : tableau de caractères qui représente le texte ;
    A[M] : tableau de caractères qui représente le mot caché ;
    Tant que (i <= N-M) et (trouve = faux) ……//1 pt
        Faire Si T[i] <> A[1] alors i= i+1 ;
            Sinon /*premier caractère trouver reste à chercher le second*/
             k=1 ;
            Tant que (i+k<=N) et (trouve =faux) ……..//1 pt
                Faire si T[i+k]=A[2] alors trouve =vrai ;
                    Sinon si T[i+k] == A[1] alors i=i+k ;
                        Sinon k=k+1 ;
                        Fsi ;
                Si trouve = vrai ;
            /* Reste maintenant à trouver les caractères restant de 3 jusqu'à M*/
                Alors j= i+ 2*k ; h=3 ; …….. //0,5
                Tant que (j <= N) et (h <=M) et (trouve =vrai) ……//1 pt
                    Faire
                        Si T[j]==A[h] alors j=j+k ; h=h+1 ;
                            Sinon trouve =faux ; i=i+k+1 ;…..//0,5
                    /* Pour reprendre la recherche du 1er caractère là où on
                    s'est déjà arrêté*/
                        fsi ;
                    Fait ;
                Fsi ;
            Fsi ;
        Fait ;
Fin.
```

{Barème de l'algorithme : 1 pt la boucle externe. 1pt pour la boucle de recherche du 2ème caractère. 1 pt pour la boucle de recherche du reste du mot. 0,5 pour l'initialisation de la dernière boucle. 0,5 pour la reprise de la recherche à la fin de la dernière boucle pour préparer la nouvelle itération de la boucle externe.}

a) Calculer la complexité de l'algorithme proposé. **(1 pt pour l'analyse et 1 pt pour le calcul et le résultat).**

Bon courage !

Nous avons dans notre algorithme 3 boucles, dont, une boucle externe et deux boucles séquentielles internes.

Les 3 boucles sont coopérantes (travaillent ensemble) dans le but de trouver le mot caché et le font en plusieurs étape : recherche du 1er caractère, ensuite recherche du second caractère et définition de la distance k qui sépare les caractères du mot M dans le texte, et en dernière étape, la recherche du reste du mot à partir de la position 3 jusqu'à M.

L'enchainement des étapes par ces boucles est conditionné par le même booléen et donc par la même hypothèse de trouver le mot en question.

Si le premier caractère du mot M n'existe pas dans le texte, le nombre d'itération de la boucle externe = N-M. Dans ce cas, à aucun moment les boucles internes vont s'exécuter.

Si maintenant le 1er caractère est trouvé à une certaine position j l'étape suivante (la seconde boucle) peut entamer sa cherche du deuxième caractère et si celui-ci est trouvé la dernière étape est également lancée. Si à une position donnée les caractères ne correspondent pas la boucle externe devra reprendre sa recherche là où les deux boucles internes se sont arrêtées dans le parcours. Cela exprime bien la coopération entre les boucles pour l'exploration du texte de 1 à N.

Si le 1er caractère est trouvé à la position j.

La première boucle interne fera au maximum N-j itération et dans ce cas le mot M n'existe pas dans le texte. Autrement, la boucle n'exécute en k itération, k étant la distance qui sépare le 1er du 2ème caractère dans le texte. La dernière boucle quant à elle devra exécuter M-3 itérations au maximum lorsque le mot est entièrement trouvé dans le texte. Au cas où la dernière étape échoue, la boucle externe devra reprendre à partir de la position qui suit le 2ème caractère trouvé.

Donc, le nombre d'itération au totale = N-M + M = N. Ainsi la complexité de l'algorithme est en O(N).

b) Est-ce qu'il est nécessaire de spécifier s'il s'agit d'une complexité au pire cas ou au meilleur cas ? Justifier votre réponse. **(1pt)**

Il s'agit de la complexité au pire cas puisque nous avons envisagé qu'au maximum on doit parcours les N-M positions du texte pour pouvoir retrouver ou pas le premier caractère du mot caché. Au meilleur cas, la situation est différente, nous pouvons supposer que le mot caché est au tout début du texte à partir de la position 1. Dans ce cas, le parcours séquentiel concerne

uniquement le mot caché et donc l'ordre de complexité ne dépend pas de N mais de M, donc O(M).

**3)** Supposons maintenant, que le mot X peut être également caché dans le texte sous son format miroir (si X = « USTHB », miroir(X)= « BHTSU ».

a) Modifier l'algorithme précédent pour prendre en compte, **en plus**, cette situation. Quel sera son impact sur la complexité ?

```
Algorithme :
Début
Tant que (i <= N) et (trouve = faux)
        Faire Si T[i] <> A[1] alors i= i+1 ;
                Sinon /*premier caractère trouver reste à chercher le second*/
 k=1 ;
Tant que (i+k<=N) et (trouve =faux)
        Faire si T[i+k]=A[2] alors trouve =vrai ; sens=0 ;
                Sinon si T[i+k] == A[1] alors i=i+k ;
/*chercher le deuxième caractère*/
                        Sinon si (i- k >M) alors si T[i-k]==A[2] alors trouve= vrai ;sens =1 ;
                                                Sinon k=k+1 ;
                                                Fsi ;
                                Sinon k=k+1 ;
                        Fsi ;
/*sens est un entier qui prend 0 si le 2eme caractère est trouvé à gauche et 1 s'il est
retrouvé à droite*/
Si trouve = vrai ;
/* Reste maintenant à trouver les caractères restant de 3 jusqu'à M*/
Alors
        Si sens (== 0)
        Alors j= i+ 2*k ; h=3 ;
                Tant que (j <= N) et (h <=M) et (trouve =vrai)
                Faire
                        Si T[j]==A[h] alors j=j+k ; h=h+1 ;
                                Sinon trouve =faux ; i=i+k+1 ;
/* Pour reprendre la recherche du 1er caractère là où on s'est déjà arrêté*/
                        fsi ;
                Fait ;
        Sinon //sens = 1 le 2eme caractère est à droite
                j=i-2*k ; h=3 ;
                Tant que (j >= 1) et (h <=M) et (trouve =vrai)
                Faire
                        Si T[j]==A[h] alors j=j-k ; h=h+1 ;
```

```
                    Sinon trouve =faux ; i=i +1 ;
                    Fsi ;
                Fsi ;
            Fait ;
        Fsi ;
Fin.
```

**{Barème du second algorithme : 1 pt pour la recherche du second caractère à gauche. 1 pt pour la boucle de recherche du reste du mot à gauche. 1 pt pour la complexité de l'algorithme.}**

Aucun impact sur la complexité parce que même si le mot existe sous son format miroir à partir de la fin du tableau c'est-à-dire que T[N]= A[1], nous devrons re parcourir le tableau dans le sens inverse à la recherche des autres caractères du mot.

Donc au pire cas nous devrons faire deux passage sur le texte ➔ 2*N = O(N).