

Entraînement : conception d'algorithme (partie 1)

Les solutions peuvent être rédigées en **pseudo-code, python, C, C++ ou Java**. La syntaxe du langage n'a pas d'importance tant que celle-ci reste **cohérente** et **compréhensible**. (Dans les exemples, les solutions sont données en pseudo-code).

Grille d'évaluation

A (20)	Les deux algorithmes répondent correctement aux problèmes posés, ils sont écrits de façon claire et compacte et sont de complexité optimale.
B (16)	Le principe général des deux algorithmes est le bon pour obtenir une complexité optimale cependant certains cas ont été oubliés ou des éléments non essentiels rajoutés (tests inutiles, écriture compliquée) ou bien on trouve des "petites" erreurs type indice, signes, etc. Ou alors, un seul algorithme a été trouvé mais c'est le cas "difficile".
C (11)	Le principe général du premier algorithme est le bon pour obtenir une complexité optimale mais pas le second.
D (8)	Il y a une tentative de réponse sur au moins le premier algorithme avec une tentative de complexité améliorée mais la solution est fausse ou la complexité non optimale.
E (1)	Soit les deux algorithmes sont faux ou manquant soit ils sont justes mais avec une complexité non optimale sans aucune tentative d'amélioration.

Exercice 1.

Soit T un tableau trié d'entiers, par exemple

2	45	120	150	220	250	368	410	582	1000
---	----	-----	-----	-----	-----	-----	-----	-----	------

Donner les deux algorithmes suivant avec complexité optimale. (Le second ne compte que si vous avez écrit le premier)

- (1) Un algorithme qui prend en entrée le tableau trié T de taille n et un entier v et qui renvoie le plus petit indice i tel que $T[i] > v$. Par exemple, pour le tableau ci-dessus et $v = 160$, l'algorithme répond 4. Si toutes les valeurs sont plus inférieures ou égales à v , l'algorithme renvoie n .
- (2) Un algorithme qui prend en entrée le tableau trié T et deux entiers min et max et qui renvoie le nombre de valeurs v du tableau telle que $min < v \leq max$. Par exemple, pour $min = 200$ et $max = 300$, l'algorithme doit répondre 2. Pour $min = 0$ et $max = 1000$, l'algorithme doit répondre 10.

Solution

(Remarque : je donne la solution itérative mais ça marche aussi en récursif)

```

Algo1
Input : Un tableau d'entier trié T de taille n, un entier v
Procédé :
    i <- 0
    j <- n
    Tant que i < j :
        m <- (i+j)/2
        Si v < T[m] :
            Si m = 0 ou T[m-1] <= v :
                Retourner m

```

```

        j <- m
    Sinon :
        i <- m+1
    Retourner n

```

Algo2

Input : Un tableau d'entier trié T de taille n, deux entiers min et max

Procédé :

```

    a <- Algo1(T, min)
    b <- Algo1(T, max)
    Retourner b - a

```

La complexité optimale est donc $O(\log(n))$, on résout le problème par dichotomie.

Exemple de "petites" erreurs type *B* : vous avez oublié de retourner n à la fin, vous avez oublié de tester $m = 0$, vous avez inversé les comparaisons.

Exercice 2.

Le problème est le suivant : on a accès en lecture à un certain tableau de données qui contient une suite de 0 suivi d'une suite de 1. Exemple : un tableau de taille 10 qui contiendrait trois 0 suivi de sept 1. Une fois qu'on a lu un 1, il n'y aura plus que des 1. **On cherche à connaître la position du premier 1.**

Pour les deux cas suivant, donnez un algorithme optimal qui donne la position du premier 1.

- (1) **Premier cas** : on prend entier le tableau *Tab* ainsi qu'un entier n qu'on suppose être la taille du tableau. Les positions sont indexées à partir de 0.

Cas particuliers : si le tableau ne contient que des 0, on retourne sa taille, s'il ne contient que des 1, on retourne 0.

- (2) **Deuxième cas** : on considère que le tableau est de taille infinie, c'est-à-dire que $\text{Tab}[i]$ donne toujours une valeur, même si i est très grand.

Cas particulier : si le tableau ne contient que des 1, on retourne 0. On a la certitude que le tableau ne contient pas que des 0.

Solution

Algo1

Input : Tab, n

Procédé :

```

    i <- 0
    j <- n
    Tant que i < j :
        m <- (i+j)/2
        Si T[m] = 1 :
            Si m = 0 ou T[m-1] = 0
                Retourner m
            j <- m
        Sinon :
            i <- m+1
    Retourner n

```

Algo2

Input : Tab

Procédé :

```

    i <- 1
    Tant que Tab[i] = 0 :
        i <- i*2

```

Retourner Algo1(Tab, i)

Remarque : dans les deux cas la complexité est $O(\log(n))$.

Exercice 3 (Partiel 2017-18).

Problème : tester si un entier n est un carré parfait, c'est-à-dire est-ce qu'il existe k tel que $k \times k = n$. Par exemple : 0, 1, 4, 9 et 16 sont des carrés parfaits mais pas 2 et 5.

Donner **deux** algorithmes qui répondent au problème (seules les opérations mathématiques de base sur des entiers sont autorisées). Les deux doivent avoir une complexité **sous-linéaire**, c'est-à-dire inférieure stricte à $O(n)$. Cependant l'un des deux sera beaucoup plus efficace que l'autre.

Remarque : si n n'est pas un carré parfait, il existera k tel que $k \times k < n$ et $(k+1) \times (k+1) > n$.

Solution

Algorithme 1 en $O(\sqrt{n})$

```

Algo1 :
i ← 0
Tant que i*i < n :
    i ← i+1
Si i*i = n :
    Retourner Vrai
Retourner Faux
    
```

Algorithme 2 en $O(\log(n))$

On cherche l'entier k tel que $k \times k = n$ dans les entiers plus petits ou égaux à n : on peut utiliser la dichotomie.

```

Algo2
i ← 0
j ← n+1
Tant que i < j :
    m ← (i+j) / 2
    Si m*m = n :
        Retourner Vrai
    Si m*m < n :
        i ← m+1
    Sinon :
        j ← m
Retourner Faux
    
```

Question donnée au partiel 1 2017-2018, résultats obtenus :

A	B	C	D	E
9.5%	9.5%	62%	19%	0%

2 étudiants ont obtenu *A* : ils ont donné les deux algos. 2 étudiants ont obtenu *B* : ils ont donné l'algo 2 dichotomique mais pas l'algo 1 (remarque : ce cas ne faisait pas partie du barème d'origine, je l'ai rajouté). Les étudiants qui ont obtenu *C* sont ceux qui ont donné une version "plus ou moins" correcte de l'algo 1. Les étudiants qui ont obtenus *D* ont soit donné un algorithme linéaire, soit un algorithme avec une erreur qui ne me permettait pas de décider si la complexité finale de l'algo corrigé serait $O(n)$ ou $O(\sqrt{n})$.

Exemple d'un algorithme qui ne marche pas mais qui obtient *C* :

```

Algo1 :
i ← 0
Tant que i*i < n :
    i ← i+1
Si (i+1)*(i+1) > n :
    
```

```

    Retourner Faux
Sinon :
    Retourner Vrai

```

La boucle s'arrête au bon moment cependant le test final ne marche pas (l'algo renvoie toujours "Faux"). Le principe général pour obtenir une complexité $O(\sqrt{n})$ est bien là malgré l'erreur : l'étudiant obtient C .

Exemple de deux algorithmes qui obtiennent D :

```

Algo1 :
Pour i allant de 0 à n/2 :
    Si i*i = n :
        Retourner Vrai
Retourner Faux

```

Quand n n'est pas un carré parfait, cet algorithme a une complexité $O(n)$, il est donc linéaire. On teste de nombreuses valeurs "inutiles" car une fois que $i * i > n$, ça ne sert plus à rien de continuer la boucle.

```

Algo1
Pour i allant de 0 à n/2 :
    Si (i*i < n) et ((i+1)*(i+1)) > n :
        Retourner Faux
Sinon
    Retourner Vrai

```

Dans ce cas, la boucle s'arrête immédiatement. Il est impossible de savoir si la version corrigée effectuerait la boucle jusqu'au bout ou non, donc le "principe général" n'est pas bon.

Exercice 4 (Partiel 2018-2019).

On cherche à évaluer une donnée numérique non entière $x > 1$ par un entier (le plus grand entier n tel que $n < x$). On ne peut pas lire directement la donnée, on n'a seulement accès à la fonction suivante :

```

infX(k) :
Renvoie True si k est strictement inférieur à la donnée x et False sinon.

```

- (1) **Premier cas** : on sait que x est compris entre deux entiers $v1 < x < v2$. Écrivez un algorithme optimal qui prend en paramètre $v1$ et $v2$ et renvoie k , l'estimation entière de x . **Donnez sa complexité.** (On suppose que `infX` a une complexité $O(1)$).
- (2) **Deuxième cas** : on ne sait rien de la donnée x . Écrivez un algorithme optimal qui renvoie k , l'estimation entière de x . **Donnez sa complexité.**

Remarque : vous avez le droit d'utiliser la première question pour résoudre la deuxième.
Solution

```

Algo1 :
Input v1, v2
k <- (v1 + v2) / 2
Tant que infX(k) == infX(k+1) :
    Si infX(k) :
        v1 <- k+1
    Sinon :
        v2 <- k
    k <- (v1 + v2) / 2
Renvoyer k

```

```

Algo2 :
k ← 1
Tant que infX(k) :
    k ← k*2
Renvoyer Algo1(k/2, K)

```

Complexité des 2 algorithmes : $O(\log(x))$.

Question donnée au partiel 1 2018-2019, résultats obtenus :

A	B	C	D	E
18%	27%	40%	13%	0%

Exercice 5 (2019-2020).

On a codé un dictionnaire à l'aide d'un tableau D contenant des mots rangés par ordre alphabétique.

- (1) **Écrire une fonction qui renvoie l'indice d'un mot donné dans le dictionnaire (ou -1 si le mot est absent) de complexité optimale et donner sa complexité.** La fonction prend en paramètre le tableau D de taille n ainsi que le mot v . On a accès à une fonction de comparaison `compare(v1, v2)` de complexité $O(1)$ qui prend en paramètre 2 mots et renvoie -1 si $v1$ est avant dans l'ordre alphabétique, 0 si les deux mots sont égaux et 1 si $v1$ est après dans l'ordre alphabétique.
- (2) La taille n du dictionnaire correspond au nombre de mots présents. Pour pouvoir rajouter des mots, le dictionnaire a aussi une capacité $c > n$. Lorsqu'on rajoute un mot et que la capacité est atteinte, elle doit être augmentée. Décrire une stratégie d'augmentation de la capacité qui permette d'obtenir une complexité optimale.

Solution

```

Algo1 :
Input D tableau de taille n, v un mot
i ← 0
j ← n
Tant que i < j :
    m ← (i+j)//2
    w = D[m]
    c ←
    Si c = 0 :
        Renvoyer m
    Sinon si c < 0 :
        j ← m
    Sinon :
        i ← m+1
Renvoyer -1

```

Algo2 : On commence avec une capacité donnée (par exemple 1) et on double la capacité quand le dictionnaire est plein.

Complexité de Algo1 : $O(\log(n))$.

Question donnée au partiel 1 2019-2020, résultats obtenus :

A	B	C	D	E
41%	44%	7%	4%	4%