2h — documents autorisés.

Toutes les réponses doivent être *argumentées* !

Par *algorithme* on entend le pseudo-code **et** sa description en langage naturel.

.....
Dans un nombre trop important de copies (15% ?), on ne fait pas la différence entre afficher et renvoyer/retourner (qui met fin prématurément aux boucles... et à la correction).

Plus le code est incompréhensible, moins il y a d'explication de l'algorithme...

Trop de copies sales où il faut chercher entre ratures et nappages de correcteur blanc. Les parties illisibles n'ont pu être lues (lapalissade).

.....
1) EST_PREMIER. On considère la fonction suivante : (6 pts)

```
1: function EST_PREMIER(n)
2:   i ← 2
3:   while i * i ≤ n do
4:     if n % i == 0 then
5:       return false
6:     end if
7:     i ← i + 1
8:   end while
9:   return true
10: end function
```

a) On considère que toutes les opérations arithmétiques se font en temps constant.
En fonction de n , quelle est la complexité (pire cas) de cet algorithme ?

.....
Il s'agit de savoir combien de fois on fait la boucle While.

Au mieux, on ne la fait qu'une fois (si n est pair). Au pire, on la fait à partir de 2 tant que $i^2 < n$ par pas de 1. On la fait donc $\mathcal{O}(\sqrt{n})$ fois.

.....
b) Même question, mais cette fois-ci, on considère que les opérations arithmétiques ont les complexités suivantes :

- comparaisons et additions : linéaires en la taille du plus grand entier (en $\mathcal{O}(t)$), et
- multiplications et divisions (quotient comme reste) : quadratiques en la taille du plus grand entier (en $\mathcal{O}(t^2)$).

.....
On part de la complexité au pire cas et on multiplie par le coût des opérations. À chaque fois, il y a un sur-coût de $\mathcal{O}(t^2)$, où t est la taille de l'entier n . Comme $t \approx \log_2(n)$, le sur-coût est de l'ordre de $\mathcal{O}(\log_2^2(n))$.

En tout, la complexité est donc de $\mathcal{O}(\log_2^2(n)\sqrt{n})$.

2) Tableaux différents (6 pts)

On dispose de deux tableaux, A et B , de même taille n . Ils ne contiennent que des valeurs entières comprises entre 1 et k (une valeur peut être présente plus d'une fois).

On veut afficher les valeurs dans A qui ne sont pas ou pas autant de fois dans B (en comptant les répétitions manquantes). Par exemple, avec $k = 9$,

- A :

1	8	2	3	3	8
---	---	---	---	---	---

 et
- B :

7	1	3	4	4	9
---	---	---	---	---	---

,

il sera affiché : 2388 (ou 8238 ou 3882 ou... l'ordre d'affichage est sans importance).

a) En supposant que k est très grand, proposer un algorithme de meilleure complexité possible (expliciter cette complexité) pour résoudre ce problème ($\mathcal{O}(n^2)$ est assez trivial, $\mathcal{O}(n \cdot \log(n))$ serait mieux).

.....
Ne pas oublier les multiplicités, sur l'exemple, 8 est affiché deux fois alors que 3 ne l'est qu'une seule fois!

On trie les deux tableaux par ordre croissant et ensuite on les parcourt simultanément. Si les valeurs sont identiques, on les passe (une occurrence de chaque côté s'annulent). On passe la plus petite, si elle est dans A on l'affiche.

S'il ne reste plus d'éléments dans B alors on doit afficher ceux de A .

```

trier ( A )
trier ( B )
i ← 1
j ← 1
while i ≤ n do
    if ( ( n < j ) or ( A[i] < B[j] ) ) then                ▷ Teste n < j avant
        afficher ( A[i] )
        i ← i + 1
    else if B[i] < A[j] then
        j ← j + 1
    else
        i ← i + 1
        j ← j + 1
    end if
end while

```

En ce qui concerne la complexité (en temps), il faut additionner celle des tris et celle de la boucle While. Les tris peuvent chacun se faire en temps $\mathcal{O}(n \ln n)$, n étant la taille de A (et de B).

Pour la boucle While, au minimum, on parcourt entièrement A sans parcourir B (par exemple, si la condition du premier if est toujours vérifiée) donc en temps linéaire en la taille de A . Au pire on parcourt également entièrement B (par exemple, tant que $j \leq n$ on passe dans le second if puis, B parcouru, on ne fait plus que le premier if). Comme A et B ont la même taille, on est toujours en temps linéaire.

En additionnant, comme le linéaire est négligeable, il ne reste que $\mathcal{O}(n \ln n)$.

.....
b) Même question, mais en supposant cette fois que k est très petit (par exemple borné par une constante) et en calculant le nombre de fois que chaque valeur apparaît.

Si k est petit, il est plus simple de compter combien de fois apparaît chaque valeur dans A , de décompter le nombre de fois où elle apparaît dans B et d'afficher ensuite.

```

compte : newint[ $k$ ]
for  $i \leftarrow 1$  to  $k$  do
    compte[ $i$ ]  $\leftarrow 0$                                 ▷ Initialisation
end for
for  $i \leftarrow 1$  to  $n$  do
    compte[ $A[i]$ ]  $\leftarrow$  compte[ $A[i]$ ] + 1                ▷ Compte des apparitions dans  $A$ 
end for
for  $i \leftarrow 1$  to  $n$  do
    compte[ $B[i]$ ]  $\leftarrow$  compte[ $B[i]$ ] - 1                ▷ Décompte des apparitions dans  $B$ 
end for
for  $i \leftarrow 1$  to  $k$  do                                ▷ Affiche les valeurs restantes
    for  $j \leftarrow 1$  to compte[ $i$ ] do
        print (  $i$  )
    end for
end for

```

Les trois premières boucles sont de complexités k , n et n respectivement. Comme k est borné, elles sont une complexité en $\mathcal{O}(n)$ (linéaire).

La dernière boucle est formée de deux boucles imbriquées. On ignore a priori la valeur de *compte*[i], mais on sait que la somme est au maximum de n (le nombre total de valeurs présentes dans A).

En tout, on obtient donc un algorithme en temps linéaire.

3) Naviguer à moindre prix (8 pts)

Le but est de descendre entièrement une rivière en canoë pour le prix le plus bas.

Il y a n postes de dépôt et location de canoë le long de la rivière (numérotés en descendant la rivière, le n°1 en haut). On peut déposer un canoë et en reprendre un à n'importe quel poste. On paye à chaque fois le prix correspondant à l'utilisation du canoë.

Les prix selon les postes de location et de dépôt sont renseignés dans un tableau à deux dimensions. Par exemple (pour $n = 4$) :

	↗	n°1	n°2	n°3	n°4
<i>prix</i> :	n°1	—	20	30	50
	n°2	—	—	20	30
	n°3	—	—	—	10
	n°4	—	—	—	—

le moins cher est ici de prendre un canoë au n°1 pour le laisser au n°3 (coût 30) puis d'en reprendre un (donc au n°3) pour aller jusqu'au n°4 (coût 10) ; soit un coût total de 40.

a) On peut résoudre ce problème récursivement en faisant un sous-appel pour chaque avant-dernier arrêt possible. La fonction a pour nom `PP` et prend un seul argument : k , un entier compris entre 1 et n . Elle retourne le meilleur prix pour aller du poste n°1 au poste n° k (en considérant tous les cas de dernier arrêt avant a entre 2 et $k - 1$).

On applique donc la formule :

$$PP(k) = \min(\text{prix}[1][k], PP(2)+\text{prix}[2][k], PP(3)+\text{prix}[3][k], \dots, PP(k-1)+\text{prix}[k-1][k]) .$$

et on résout donc le problème en appelant $PP(n)$.

Écrire l'algorithme récursif correspondant. Donner une équation de récurrence permettant de calculer sa complexité (sans chercher à résoudre cette récurrence!).

```
.....  
1: function PP(k)  
2:   v ← prix[1][k]  
3:   for i ← 2 to k-1 do  
4:     v ← min(v, PP(i) + prix[i][k])  
5:   end for  
6:   return v  
7: end function
```

Soit $cx(k)$ la complexité en temps. Elle vérifie l'équation de récurrence suivante :

$$\begin{cases} ck(1) = a \\ ck(k) = b + \sum_{i=2}^{k-1} (c + ck(i)) \end{cases}$$

où a , b et c représentent des constantes de temps (accès à un élément, somme, min...)

.....
b) Beaucoup de coûts sont recalculés inutilement (par exemple, $PP(n-3)$ est calculé 4 fois et $PP(n-4)$ 8 fois!). L'idée est donc de stocker les valeurs dans un tableau ($pp[]$). Celui-ci sera rempli en considérant les postes par numéros croissants et en utilisant la même formule qu'au dessus.

Écrire l'algorithme correspondant (qui n'est pas récursif).

Quelle est la complexité de cet algorithme ?

.....
Plutôt que de recalculer, on met dans un tableau (q) les valeurs. Chaque case ne dépendant que des cases avec des indices plus petits, on le remplit donc de 1 à k .

```
1: function PP(k)  
2:   q ← new int[k]  
3:   for i ← 2 to k do  
4:     q[i] ← prix[1][i] ▷ Initialisation avec premier terme du min  
5:     for j ← 2 to i-1 do ▷ On fait tous les autres min  
6:       q[i] ← min(q[i], q[j] + prix[i][j])  
7:     end for  
8:   end for  
9: end function
```

Soit $cx(k)$ la complexité en temps.

$$cx(k) \approx \sum_{i=2}^k \left(\sum_{j=2}^{i-1} 1 \right) = \sum_{i=2}^k (i-2) = \left(\sum_{i=2}^k i \right) - 2(k-1) = \frac{k(k+1)}{2} - 1 - 2(k-1) \in O(k^2)$$

Il s'agit d'un algorithme en temps quadratique.

.....
c) (hors barème) Résoudre l'équation de récurrence de la question a) (2 pts).

.....
(hors correction)
.....