Semaine 3 : Série d'exercices sur les algorithmes [Solutions]

1 Quel est le bon algorithme ? – le retour

Le bon algorithme est le **c**. Le **a** a deux problèmes : 1) si n est pair, il ne calcule que la somme des $n/2$ premiers nombres pairs ; 2) si n est impair, la condition de terminaison n'est jamais recontrée et l'algorithme ne s'arrête jamais ; le **b** est problématique car pour $n \geq 1$, sa sortie est un nombre plus grand ou égal à 2^n, et le **d** est encore plus problématique, car il ne s'arrête jamais pour toute valeur de $n \geq 2$.

2 Que font ces algorithmes ?

algo1 : a) 36 ; **b)** pas récursif ; **c)** sortie : la somme des n premiers nombres impairs (qui vaut n^2) ; **d)** $\Theta(n)$;.
algo2 : a) 36 ; **b)** pas récursif ; **c)** sortie : n^2 ; **d)** Répondre correctement à cette question est difficile car :

- si l'on s'en tient à la stricte définition, il faut calculer la complexité comme une fonction de « la taille de l'entrée » ; il faut alors pour cela savoir comment représenter n (ce qui est l'objet de la leçon de la semaine suivante) ;
- il vous faudrait de plus savoir quel est le meilleur algorithme pour effectuer le calcul de n^2.

Pour ces raisons, nous avons décidé (axiomatique) que pour ce cours toutes les opérations arithmétiques sont considérées comme des *instruction élémentaires*. Sous cette hypothèse, la réponse serait alors $\Theta(1)$.

Notes :

- nous ne poserons évidemment pas une telle question à l'examen ;
- l'hypothèse que les opérations arithmétiques sont élémentaires reste raisonnable tant que l'on considère des algorithmes dont les complexités sont bien plus grandes que cela, à l'exemple des trois autres algorithmes de cet exercice ;
- la vraie complexité des opérations arithmétiques a été donnée pour information en annexe du cours de la semaine passée.

algo3 : a) 36 ; **b)** récursif. ; **c)** sortie : la somme des n premiers nombres impairs (qui vaut n^2) ; **d)** $\Theta(n)$.
algo4 : a) 12 ; **b)** récursif ; **c)** sortie : si n est impair, la somme des $(n+1)/2$ premiers nombres impairs ; si n est pair, la somme des $n/2$ premiers nombres pairs ; **d)** $\Theta(n)$.
e) algo7 est le seul algorithme qui fonctionne correctement ; les autres ne s'arrêtent jamais.
f) Il recalcule plusieurs fois la même chose inutilement et prend également un temps exponentiel pour s'exécuter.
g) Si l'entrée est n, alors la sortie sera n (tout ça pour ça...).

3 Au temps des Grecs

Algorithme d'Euclide - version récursive :

pgcdRecursif

entrée : a, b *deux entiers naturels non nuls*
sortie : *pgcd* (a, b)

```
Si b = 0
    sortir : a
sortir : pgcdRecursif (b, a mod b)
```

4 Au temps des Egyptiens, deuxième partie

La version récursive de l'algorithme est donnée par :

multRecursif

entrée : a, b *deux entiers naturels non nuls*
sortie : $a \times b$

```
Si b = 1
    sortir : a
Si b est pair
    sortir : multRecursif(2a, b/2)
Sinon
    sortir : a + multRecursif(a, b − 1)
```

5 Création d'algorithmes

a) La solution la plus simple consiste ici à compter les espaces et à ajouter 1 :

nombre de mots

entrée : *chaîne de caractères* A
sortie : *le nombre de mots de* A

```
n ⟵ taille(A)
s ⟵ 0
Pour i allant de 2 à n − 1
    Si A(i) = ' '
        s ⟵ s + 1
sortir : s + 1
```

La complexité de cet algorithme est clairement $\Theta(n)$ (**taille** est toujours $\Theta(n)$, voire peut être moins).

Pour les séquences d'espaces, il suffit :

— soit d'ajouter une boucle dès qu'on a trouvé une espace ;

— soit de ne compter un mot que si le caractère précédent (une espace) n'est pas une espace.

Pour les espaces initiales et finales, la seconde solution ci-dessus convient aussi ; elle est donc la plus simple. Attention cependant à la fin : il faut maintenant aller jusqu'au bout (n au lieu de $n-1$) et n'ajouter 1 que si la dernière lettre n'est pas une espace :

nombre de mots

entrée : *chaîne de caractères* A
sortie : *le nombre de mots de* A

```
n ⟵ taille(A)
s ⟵ 0
Pour i allant de 2 à n
    Si A(i) = ' ' et A(i−1) ≠ ' '
        s ⟵ s + 1
Si A(n) ≠ ' '
    s ⟵ s + 1
sortir : s
```

b) Il suffit de regarder tour à tour toutes les valeurs de la liste, en mémorisant la plus grande vue jusqu'ici :

la plus grande valeur

entrée : *liste* L *(non vide) de nombres entiers positifs*
sortie : *la plus grande valeur de* L

```
n ⟵ taille(L)
x ⟵ L(1)
Pour i allant de 2 à n
    Si L(i) > x
        x ⟵ L(i)
sortir : x
```

A nouveau, l'ordre de complexité de l'algorithme est $\Theta(n)$.
Et pour une version récursive :

plus_grande_valeur_r

entrée : *liste* L *(non vide) de nombres entiers positifs*
sortie : *la plus grande valeur de* L

```
n ⟵ taille(L)
Si n = 1
    sortir : L(1)
sortir : max( L(1), plus_grande_valeur_r( L(2, n) ) )
```

en notant « $L(2,n)$ » la sous-liste de L constituée des éléments $L(2), L(3), \cdots, L(n)$.

Notons $C(n)$ la complexité de cette version récursive. Supposons de plus ici que la complexite de **taille** est $\Theta(1)$, disons a (sinon, il suffirait de changer un tout petit peu l'algorithme en lui

ajoutant un paramètre de plus, la taille, pour retrouver ce même résultat). Nous avons :

$$C(n) = a + 1 + 1 + 1 + C(n-1)$$

et $C(1) = a + 3$. Donc : $C(n) = (a+3)\,n$, qui est donc aussi en $\Theta(n)$.

c) Ici, plusieurs possibiltés s'offrent à nous, qui sont plus ou moins efficaces. Une première possibilité est de parcourir toutes les paires de nombres de la liste et de garder la trace du plus grand des produits de ces paires de nombres. La complexité d'un tel algorithme est $\Theta(n^2)$ et n'est pas optimale. Une meilleure option est de parcourir une fois la liste et de garder la trace des deux plus grands nombres rencontrés :

le plus grand produit

entrée : *liste L de nombres entiers positifs*
sortie : *le plus grand produit de deux valeurs de L*

```
Si L(1) > L(2)
|   x1 ⟵ L(1)
|   x2 ⟵ L(2)
Sinon
|   x1 ⟵ L(2)
|   x2 ⟵ L(1)
n ⟵ taille(L)
Pour i allant de 3 à n
|   Si L(i) > x1
|   |   x2 ⟵ x1
|   |   x1 ⟵ L(i)
|   Sinon
|   |   Si L(i) > x2
|   |   |   x2 ⟵ L(i)
sortir : x1 × x2
```

L'ordre de complexité de l'algorithme est $\Theta(n)$ (car à chaque itération, le nombre d'opérations effectuées est constant).

Une troisième possibilité serait de d'abord trier (dans l'ordre croissant) la liste avec un algorithme de tri efficace, puis de sortir le produit des deux derniers nombres de la liste. Mais la complexité serait alors au mieux de $\Theta(n \log_2(n))$ en fonction de l'algorithme de tri utilisé.

6 Taille de liste

1) Pour la solution de complexité linéaire : il suffit d'essayer toutes les valeurs une à une :

Taille

entrée : *Liste L*
sortie : *n le nombre d'éléments de L*

$n \longleftarrow 1$
Tant que a_ element*(L, n)*
  $n \longleftarrow n + 1$
$n \longleftarrow n - 1$
sortir : n

L'algorithme met effectivement $n+1$ étapes à s'arrêter. C'est bien un algorithme de complexité linéaire.

2) Pour une version sous-linéaire, toute la difficulté est de trouver une borne supérieure à la taille, car une fois que l'on a une telle borne supérieure, on peut simplement rechercher *par dichotomie* entre par exemple 1 et cette borne, ce qui donnera un algorithme de complexité logarithmique.

La question est donc de savoir si l'on peut trouver une borne supérieure à la taille n de L en un temps logarithmique en n. La réponse est oui : prenons une constante K (par exemple $K = 2$) et demandons si **a_ element**(L, K^i), pour i partant de 1 et augmentant.

Nous aurons besoin de poser $\lfloor \log_K(n) \rfloor + 1$ fois cette question. Une fois i trouvé, nous pouvons rechercher la taille par dichotomie entre K^{i-1} et K^i.

Au total, notre algorithme effectuera $\Theta(\log n)$ opérations. Formellement, avec $K = 2$, on peut écrire l'algorithme comme ceci :

TailleLog

entrée : *Liste L*
sortie : *le nombre d'éléments de L*

$t \longleftarrow 1$
Tant que a_ element*(L, t)*
  $t \longleftarrow 2\,t$
sortir : TailleDichotomique$(L, \lfloor t/2 \rfloor, t)$

avec

TailleDichotomique

entrée : *L, a, b*
sortie : *le nombre d'éléments de L*

Si $a \geq b - 1$
  sortir : a
$c \longleftarrow a + \left\lfloor \frac{b-a}{2} \right\rfloor$
Si a_ element*(L, c)*
  sortir : TailleDichotomique*(L, c, b)*
Sinon
  sortir : TailleDichotomique*(L, a, c)*

L'idée de ce dernier algorithme est de chercher la taille entre a *inclus* et b *exclu*.

Pour aller plus loin

7 Devinette

Cet algorithme récursif effectue une *recherche par dichotomie* presque exactement comme vu en cours. La seule différence avec le cours, c'est qu'ici on compare aussi avec « l'élément du milieu » (c dans l'algorithme).

Rappel du fonctionnement de l'algorithme : Étant donnés une liste L triée par ordre croissant, un objet x et des bornes $a \leq b$, il cherche l'emplacement dans L entre les indices a et b qui contient x. Une manière de chercher dans *toute* la liste est de choisir $a = 1$ et $b =$ taille de L.

L'algorithme procède en affinant successivement l'intervalle de recherche. À chaque étape, x est comparé à l'élément c situé au milieu de l'intervalle actuel, et un appel récursif est effectué sur un nouvel intervalle réduit de moitié. Lorsque l'intervalle a été réduit à un seul élément ($a = b$), soit l'élément est équivalent à x et l'indice est retourné, soit la liste ne contient pas x et l'algorithme retourne 0 (indice impossible pour un élément).

Exactitude : Sans autre contraintes sur a et b, cet algorithme n'est pas correct dans tous les cas.

Si en entrée $a > b$, il peut arriver qu'il ne termine pas en raison d'un nombre infini d'appels récursifs.

Par ailleurs, si a est plus grand que le nombre d'éléments dans la liste, « le a-ième » ou « le c-ième élément de L » ne sont pas définis.

De plus, la contrainte $a \leq b$ n'est même pas garantie dans tous les cas dans l'algorithme lui-même ! En effet, si $x < L(a)$ et $b = a + 1$ alors on lancera devinette$(L, x, a, a - 1)$! Essayez par exemple avec $L = \{2, 3\}$, $x = 1$, $a = 1$ et $b = 2$.

Le plus simple serait donc d'ajouter un test au début de l'algorithme pour garantir $1 \leq a \leq b \leq$ taille(L) et répondre 0 sinon.

8 Deviner l'affichage

Lorsque $m =$ «3 : » et $n = 3$:

```
3 : ,3
3 : ,2,1
3 : ,1,2
3 : ,1,1,1
```

Lorsque $m =$ «4 : » et $n = 4$:

```
4 : ,4
4 : ,3,1
4 : ,2,2
4 : ,2,1,1
4 : ,1,3
```

```
4 : ,1,2,1
4 : ,1,1,2
4 : ,1,1,1,1
```

L'algorithme affiche toutes les permutations avec répétition de séquences de nombres entre 1 et n dont la somme est n.

Pour le fun...

$P(1)$ est vraie, en effet. L'erreur n'est pas là.

$P(n) \implies P(n+1)$ en effet si $n \geq 2$; le problème n'est pas là.

Le problème, c'est la démonstration de $P(1) \implies P(2)$: en effet, « le crayon que l'on a sorti est aussi de la même couleur que les autres restés dans la boîte, donc tous les 2 crayons sont bien tous de la même couleur, » est erroné car « les autres restés dans la boîte » est un ensemble vide...