M1.L3 : Série d'exercices sur les algorithmes / complexité [Solutions]

1 Ordre de complexité des algorithmes de la série M1.L2

1.1) Détermination si un nombre entier n est premier en fonction de n : dans le pire des cas on teste tous les entiers entre 2 et $n^{1/2}$. L'ordre de complexité est donc $O(n^{1/2})$. La série M1.L5 offrira une autre analyse de ce problème.

1.2) conversions : Le nombre maximum d'étapes dans le pire des cas étant constant, il s'agit d'un ordre de complexité en O(1) qui est indépendant de la donnée en entrée.

1.3.1) multiplication égyptienne (devinette1) en fonction de y : le nombre maximum de passage est déterminé par le nombre de fois qu'on peut diviser y par 2, ce qui est donné par $\log_2(y)$. D'où $O(\log_2(y))$.

1.3.2) Algorithme d'Euclide en fonction du max des paramètres x et y. Dans le pire des cas on a x valant 1 et y est un entier positif ; le nombre de passage dans la boucle sera proportionnel à y. D'où un ordre de complexité linéaire.

1.4.1) Détermination de la plus petite valeur d'une liste L de taille n. Il faut comparer le premier terme avec les (n-1) autres termes. Le terme dominant est linéaire, d'où O(n).

1.4.2) Détermination de la plus petite différence dans une liste de taille n. Le nombre de comparaisons contient un terme dominant en n^2, d'où $O(n^2)$.

1.4.3) a) une simple itération suffit grâce à l'ordre produit par le tri ; l'ordre de complexité de la partie qui exploite la liste trié est linéaire car le nombre de passage est proportionnel à la taille n de la liste L.

b) Pour avoir l'ordre de complexité total, il faut aussi considérer l'ordre de complexité du tri de la liste L qui permet d'ordonner les éléments de cette liste.

En utilisant une approche naïve, la complexité temporelle d'un tri est $\mathrm{O}(n^2)$, et celle de notre algorithme est alors la même que pour la première version (question 1.4.2).

Des méthodes plus efficaces permettent d'ordonner une liste de n nombres en $\mathrm{O}(n\log(n))$ opérations. La complexité temporelle de notre nouvel algorithme devient alors de la forme :

$$C_1 + C_2\, n + C_3\, n\, \log(n)$$

Dont le terme dominant est le troisième. Donc au final cet algorithme est en $\mathrm{O}(n\log(n))$; ce qui pour des listes de grande taille fait une grosse différence avec la premiere version en $O(n^2)$!

2 Manipulation de liste

Il trie les éléments de la liste L dans l'ordre croissant (tri-par-sélection).

L'ordre de complexité est $O(n^2)$ du fait de la double boucle imbriquée qui produit un nombre total de comparaisons du même type que pour l'exercice 1.4.2 (voir solution de M1.L2), avec un terme dominant en n^2.

3 Taille de liste

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

Taille
entrée : Liste L sortie : n le nombre d'éléments de L
$n \leftarrow 1$ **Tant que** a_element(L, n) $n \leftarrow n + 1$ $n \leftarrow n - 1$ **sortir** : n

L'algorithme met effectivement $n + 1$ étapes à s'arrêter. C'est bien un algorithme linéaire en O(n).

b) Pour une version sous-linéaire (= moins coûteuse que la version 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 $\text{partie\_entière}(\log_K(n))+1$ 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 $O(\log n)$ opérations. Formellement, avec $K = 2$, on peut écrire l'algorithme comme ceci :

TailleLog
entrée : Liste L sortie : le nombre d'éléments de L
$t \leftarrow 1$ Tant que a_element(L, t) $t \leftarrow 2t$ sortir : TailleDichotomique(L, $t/2$, t)

avec

TailleDichotomique
entrée : L, a, b sortie : le nombre d'éléments de L
Tant que $a < b - 1$ $c \longleftarrow a + \lfloor \frac{b-a}{2} \rfloor$ **Si a_element(L , c)** $a \leftarrow$ c **Sinon** b $\leftarrow$ c **sortir** a

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

3. Que font ces algorithmes?

a) La sortie de l'algorithme vaut $\mathbf{2}^{n}$ (on répète n fois l'opération $i \leftarrow 2i$ en partant de $i = 1$), et le nombre d'opérations effectuées est donc de l'ordre de n, d'où un ordre de complexité en O(n)

b) Quelles que soient les valeurs de a et b, la sortie de l'algorithme vaut $\boldsymbol{a \cdot b}$, et le nombre d'opérations effectuées est de l'ordre du minimum de a et b. En e et, si $a < b$, alors on répète a fois l'opération $s \leftarrow s + b$; dans le cas contraire, on répète b fois l'opération $s \leftarrow s + a$. Vu qu'on part de $s = 0$, le résultat est le même dans les deux cas, et le nombre d'additions effectuées est toujours le mininum. L'ordre de complexité est donc O(min(a,b)).

c) L'algorithme détecte si la liste L est **ordonnée dans l'ordre croissant ou non.** L'algorithme part de l'a priori positif que c'est le cas et change d'avis dès que deux nombres consécutifs ne respectent pas cet ordre. Le nombre d'opérations effectuées est proportionnel la taille de la liste (n). D'où O(n).

d) L'algorithme détecte s'il existe **deux nombres dans la liste L dont la différence est plus petite que d.** Il part du principe que ce n'est pas le cas, et change d'avis dès qu'il trouve deux nombres dont la différence est plus petite que d (en particulier, avec $d = 1$, il détecte s'il existe deux nombres entiers identiques dans la liste). Dans le pire des cas, le nombre d'opérations effectuées est proportionnel n^2. D'où O(n^2).