# **M1.L3** : Série d'exercices sur les algorithmes / complexité

## 1 Ordre de complexité des algorithmes

1.1) Dans la série MOOC sem3, analyser la complexité de la détermination de la primalité d'un nombre entier n en fonction de n. On suppose ici qu'on travaille avec l'hypothèse habituelle d'un coût constant des opérateurs arithmétiques (on en reparlera au moment de M1.L5).

Indiquer l'ordre de complexité avec la notation O( ) pour chacun des algorithmes suivants étudiés dans la série M1.L2 :

1.2) les algorithmes de conversion travaillent tous en un nombre borné d'étapes quel que soit la donnée qui est fournie en entrée. Qu'en déduit-on comme ordre de complexité ?

1.3.1) multiplication égyptienne (Algo mystérieux 1) en fonction de y

1.3.2) Algorithme d'Euclide (Algo mystérieux 2) en fonction du max des paramètres x et y

1.4.1) Détermination de la plus petite valeur d'une liste L de taille n

1.4.2) Détermination de la plus petite différence dans une liste non-triée de taille n

1.4.3) **supplément à la question 1.4.2** : On suppose que la liste L fournie à l'algorithme est d'abord triée dans une première étape avant d'effectuer la recherche de la plus petite différence sur cette liste triée. Voici cet algorithme ci-dessous.

a) Exécutez-le sur une liste de plusieurs éléments dans le désordre pour vérifier son fonctionnement

| La plus petite différence-2 |
|---|
| entrée : Liste $L$ d'entiers, de taille n, contenant au moins 2<br>éléments sortie : $a,b$ nombres avec la plus petite différence |
| $L_{triee}$ = tri($L$)<br>$a \leftarrow L_{triee}(1)$<br>$b \leftarrow L_{triee}(2)$<br>$\delta_{min} \leftarrow \lvert L(2) - L(1)\rvert$<br>Pour $i$ de 1 à $n-1$<br>    $\delta \leftarrow \lvert L_{triee}(i+1) - L_{triee}(i)\rvert$<br>    Si $\delta < \delta_{min}$<br>        $\delta_{min} \leftarrow \delta$<br>        $a \leftarrow L_{triee}(i)$<br>        $b \leftarrow L_{triee}(i+1)$<br>sortir : $(a,b)$ |

b) Quel est son ordre de complexité ?

## 2 Manipulation de liste

On considère l'algorithme *algorithm*1. L'entrée de l'algorithme est une liste *L* et le nombre *n* d'éléments dans *L* avec *n* ≥ 1. L'algorithme utilise la fonction **swap(L[i],L[j])** qui échange les éléments dans la liste *L* aux positions *i* et *j*.

| algorithm1 |
|---|
| entrée : L, n<br>sortie : quelle transformation est effectuée sur L ? |
| **Pour** i de 1 à n – 1<br>    jMin ← i<br>    **Pour** j de i + 1 à n<br>        **Si** L[j] < L[jMin]<br>            jMin ← j<br>    **Si** jMin ≠ i<br>        **swap**(L[i],L[jMin]) |

Exécutez cet algorithme sur les listes de tailles différentes contenant des nombres entiers ; commencez par les cas limite (n valant 1) puis continuez avec des tailles de liste plus importantes jusqu'à être certain du but de l'algorithme. Aidez-vous d'un dessin pour suivre la progression du déroulement de l'algorithme.

a) Votre conclusion : de manière générale que fait cet algorithme?

b) Sachant que la fonction swap a un coût constant, quel est son ordre de complexité en fonction de n ?

## 3 Taille de liste

Soit L une liste d'entiers (pas forcément ordonnée et avec de possibles répétitions), comme par exemple {19, 31, 15, 21}.

On cherche ici à écrire un algorithme **taille($L$)** qui retourne le nombre d'éléments d'une liste $L$ donnée en entrée. On suppose que l'on dispose d'un autre algorithme **a_element( L , i )** qui nous dit (vrai ou faux) s'il existe un ***i*<sup></sup>**$^{e}$ élément dans la liste : il répond donc **vrai** si $i$ est inférieur ou égal la taille de la liste et **faux** sinon.[1]

[1] . Notez qu'il n'est pas évident de toujours avoir un tel algorithme (sans avoir taille, bien sûr!). En réalité cela dépend de la représentation effective de $L$. Mais nous faisons ici l'hypothèse qu'un tel algorithme existe pour $L$.

a) Comment utiliser l'algorithme a_element($L$,$i$) pour calculer la taille de $L$ en un temps linéaire (par rapport cette taille)? Ecrivez un algorithme pour le faire.

b) En vous inspirant de la recherche dichotomique (cf. cours), pourrait-on avoir une complexité moindre que linéaire en s'aidant par exemple de la suite des puissances de deux ?
Si oui, quelle serait cette complexité et expliquez intuitivement comment procéder.
Essayez d'écrire un tel algorithme.

## 4 Que font ces algorithmes?

Pour chacun des algorithmes suivants, indiquer (en mots) quelle est la sortie de l'algorithme.

Exprimer leur ordre de complexité avec la notation O( ) de Landau en fonction du paramètre d'entrée n (ou du max des paramètres a et b pour le second algorithme).

| Algorithme a) |
|---|
| entrée : $n$ nombre naturel<br>sortie :?? |
| $m \longleftarrow n$<br>$i \longleftarrow 1$<br>Tant que $m > 0$<br>    $i \leftarrow 2i$<br>    $m \leftarrow m - 1$<br>Sortir : $i$ |

| Algorithme b) |
|---|
| entrée : $a$,$b$ nombres naturels non-nuls<br>sortie :?? |
| $s \leftarrow 0$<br>Si $a < b$<br>    Pour $i$ allant de 1 à $a$<br>        $s \leftarrow s + b$<br>Sinon<br>    Pour $i$ allant de 1 à $b$<br>        $s \leftarrow s + a$<br>Sortir : $s$ |

| Algorithme c) |
|---|
| entrée : $L$ liste de nombres entiers, $n$ taille de la liste<br>sortie :?? |
| $s \leftarrow$oui<br>Pour $i$ allant de 1 à $n - 1$<br>    Si $L(i + 1) < L(i)$<br>        $s \leftarrow$non<br>Sortir : $s$ |

| Algorithme d) |
|---|
| entrée : $L$ liste de nombres entiers, $n$ taille de la liste, $d$ nombre naturel non-nul<br>sortie :?? |
| $s \leftarrow$non<br>Pour $i$ allant de 1 à $n - 1$<br>    Pour $j$ allant de $i + 1$ à $n$<br>        Si $\lvert L(j) - L(i) \rvert < d$<br>            $s \leftarrow$oui<br>Sortir : $s$ |