L2 Informatique - Algorithmique - Correction session 1

lundi 30 novembre 2015. 10h00-12h00. T' 301.

La précision et la clarté de votre rédaction sont *fondamentales*. Documents interdits. Durée 2h15. Le barème indiqué est *approximatif.*

Exercice 1. [3pts] Calculez la somme des n premiers entiers naturels strictement positifs. Un *palindrome* est un mot $u = u_1u_2 \dots u_n$ sur un alphabet fini A qui est égal à son mot *miroir* $u_n u_{n-1} \dots u_1$. Le mot *radar* est un palindrome. Combien de palindromes de longueur n peut-on construire sur un alphabet A de cardinal q ? Justifiez votre réponse.

Solution. La question est rituelle, la somme S_n des n premiers entiers naturels strictements positifs est égale à $n(n+1)/2$. On le montre (ce n'était pas demandé) en écrivant les termes dans l'ordre croissant et dans l'ordre décroissant :

$$\begin{aligned} S_n + S_n &= \sum_{i=0}^{n} i + \sum_{i=0}^{n} (n-i) \\ 2S_n &= \sum_{i=0}^{n} (i + n - i) \\ 2S_n &= \sum_{i=0}^{n} n \\ 2S_n &= (n+1)n. \end{aligned}$$

Soit $u = u_1u_2 \dots u_n$ un palindrome. On note $k := \lfloor \frac{n+1}{2} \rfloor$ l'indice de la lettre centrale du palindrome u. Les k premières lettres déterminent la valeur des $n-k$ dernières. Pour compter le nombre de palindromes, ll suffit donc de compter combien de préfixes $u_1u_2 \dots u_k$ on peut construire. Comme les lettres sont indépendantes et que l'on a le choix parmi q, on a donc

$$\underbrace{q \times q \times \cdots \times q}_{k \text{ fois}} = q^{\lfloor \frac{n+1}{2} \rfloor}$$

palindromes possibles.

Exercice 2. [5pts] On appelle *élément* majoritaire d'une liste l'élément dont le nombre d'occurences est strictement supérieur à la moitié de la taille de la liste, s'il existe. Écrivez un algorithme Occurences(L, x) qui renvoie le nombre d'occurrences de la valeur x dans la liste L. Quelle est complexité de votre algorithme (justifiez) ?

Utilisez Occurences(L, x) pour écrire un algorithme Majoritaire(L) qui renvoie l'élément majoritaire de la liste L s'il existe et 0 sinon. On supposera donc que la liste ne contient pas la valeur nulle. Quelle est la complexité de votre algorithme (justifiez) ?

Supposons à présent que la liste L soit triée dans l'ordre croissant. Démontrez que s'il existe un élément majoritaire alors sa première occurence est nécessairement dans la première moitié de la liste. En déduire un nouvel algorithme MajoritaireTriee(L) qui suppose que la liste L est triée. Quelle est la complexité de votre algorithme (justifiez) ?

Solution. L'algorithme qui compte le nombre d'occurences de x dans une liste L est simplissime :

Algorithme Occurences(L, x) : entier
données
  L : liste de valeurs
  x : valeur
variables
  $nbo,\ i$: entiers

$nbo \leftarrow 0$
$i \leftarrow 1$
tantque $(i \leq \#L)$ **faire**
  si $(L[i] = x)$ **alors**
    $nbo \leftarrow nbo + 1$
  fsi
  $i \leftarrow i + 1$
ftq
renvoyer(nbo)

Algo 1. Nombre d'occurences d'une valeur x dans une liste L.

Les instructions hors de la boucle se font en temps constant $\Theta(1)$. La boucle parcourt l'intégralité de la liste L de taille n et les instructions dans la boucle se font en $\Theta(1)$, la complexité est donnée par

$$\begin{aligned} T(n) &= \Theta(1) + n \times \Theta(n) \\ &= \Theta(n) \end{aligned}$$

Déterminer l'élément majoritaire, s'il existe, s'en déduit immédiatement. Il suffit de compter le nombre d'occurences de chaque élément de la liste et de vérifier s'il dépasse la moitié de la taille de la liste.

Algorithme $\textsc{Majoritaire}(L)$: valeur
données
  L : liste de valeurs
variables
  i : entier

$i \leftarrow 1$
tantque $(i \leq \#L)$ **faire**
  si (Occurences$(L, L[i]) > \frac{\#L}{2}$) **alors**
    renvoyer$(L[i])$
  fsi
  $i \leftarrow i + 1$
ftq
renvoyer(0)

Algo 2. Recherche de l'élément majoritaire dans une liste L.

Les instructions en dehors de la boucle s'exécutent en temps constant $\Theta(1)$. Dans le meilleur des cas, le premier élément de la liste est l'élément majoritaire, l'algorithme Occurences n'aura été appelé qu'une seule fois, on a donc $\check{T}(n) = \Theta(1) + \Theta(n) = \Theta(n)$. Dans le pire des cas, il n'y a pas d'élément majoritaire et les n éléments de la liste seront testés par l'algorithme Occurences, La complexité de cet algorithme est donc $\hat{T}(n) = \Theta(1) + n \times \Theta(n) = \Theta(n^2)$.

Si x est l'élément majoritaire de la liste L, par définition ses occurences occupent $k > \lfloor \frac{n}{2} \rfloor$ cellules de la liste L soit strictement plus de la moitié. Il est donc impossible de ranger toutes les occurences de l'élément majoritaire dans la moitié des cellules, où qu'elles se trouvent, et en particulier dans la deuxième moitié de la liste. D'autre part, si la liste est triée, toutes les occurences d'une même valeur x se suivent. Ainsi, si x est majoritaire et i désigne la position de sa première occurence, les $\lfloor \frac{n}{2} \rfloor$ positions suivantes contiennent nécessairement x. Réciproquement s'il y a au moins $\lfloor \frac{n}{2} \rfloor + 1$ valeurs identiques qui se suivent, il s'agit par définition de l'élement majoritaire. Ainsi il est aisé de déterminer si l'élément à la position i est l'élément majoritaire ou non en comparant $L[i]$ à $L[i + \lfloor \frac{n}{2} \rfloor]$. L'algorithme s'en déduit :

Algorithme $\textsc{MajoritaireTriée}(L)$: valeur
données
  L : liste de valeurs triées
variables
  i : entier

$i \leftarrow 1$
tantque $(i \leq \#L/2)$ **faire**
  si $(L[i] = L[i + \lfloor \frac{n}{2} \rfloor])$ **alors**
    renvoyer$(L[i])$
  fsi
  $i \leftarrow i + 1$
ftq
renvoyer(0)

Algo 3. Recherche de l'élément majoritaire dans une liste triée L.

Dans le meilleur des cas l'élément majoritaire est le plus petit dans la liste et la complexité est donc en $\Theta(1)$. Dans le pire des cas, il faut avancer jusqu'à la fin de la première moitié de la liste et on réalise $\lfloor \frac{n}{2} \rfloor$ tests soit une complexité en $\Theta(n)$. Néanmoins la complexité globale dépend de celle de l'algorithme de tri. Si la situation le permet on pourrait trier la liste avec le tri par dénombrement et obtenir une complexité en $O(n)$, sinon on aura une complexité au mieux en $\Omega(n \log n)$.

Exercice 3. [5pts] Calculez la matrice des longueurs des plus longues sous-séquences communes des mots *matrice* et *aimable*. Exhibez deux chemins dans la matrice menant à deux plus longues sous-séquences communes différentes.

Solution. Notons que dans la matrice L ci-dessous, il existe d'autres chemins pour obtenir chacune des deux PLSC *aie* et *mae*.

L	ε	m	$\boldsymbol{a}$	t	r	$\boldsymbol{i}$	c	$\boldsymbol{e}$
ε	0	**0**	0	0	0	0	0	0
$\boldsymbol{a}$	0	0	**1**	**1**	**1**	1	1	1
$\boldsymbol{i}$	0	0	1	1	1	**2**	**2**	2
m	0	1	1	1	1	2	**2**	2
a	0	1	2	2	2	2	**2**	2
b	0	1	2	2	2	2	**2**	2
l	0	1	2	2	2	2	**2**	2
$\boldsymbol{e}$	0	1	2	2	2	2	2	**3**

L	ε	$\boldsymbol{m}$	$\boldsymbol{a}$	t	r	i	c	$\boldsymbol{e}$
ε	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1
i	**0**	0	1	1	1	2	2	2
$\boldsymbol{m}$	0	**1**	1	1	1	2	2	2
$\boldsymbol{a}$	0	1	**2**	2	2	2	2	2
b	0	1	**2**	**2**	**2**	2	2	2
l	0	1	2	2	**2**	**2**	**2**	2
$\boldsymbol{e}$	0	1	2	2	2	2	2	**3**

TABLE 4. En gras, à gauche un chemin pour la PLSC *aie*, à droite pour la PLSC *mae*.

Exercice 4. [3pts] Soient $u = u_1u_2\dots u_n$ et $v = v_1v_2\dots v_m$ deux mots sur un alphabet A. On dit que u est une *portion* de v, s'il existe un entier k tel que $1 \leq k \leq m-n+1$ et $\forall i \in [1,n],\ u_i = v_{k+i-1}$. Combien y-a-t-il de portions de longueur n d'un mot de longueur m ? Écrivez un algorithme ESTPORTION(u,v) qui renvoie `vrai` si u est une portion de v et `faux` sinon. On notera $u[i]$ le i-ème terme de la séquence u. Estimez la complexité de cet algorithme dans le meilleur des cas et dans le pire des cas en fonction des longueurs n et m des mots u et v.

Solution. Il s'agit de compter les différentes manières de couper une portion de largeur n dans un mot de longueur m. Cette portion peut débuter à n'importe quelle position entre les indices 1 et $m-n+1$, il y en a donc $m-n+1$.

Le principe de l'algorithme consiste de manière imagée à écrire le mot u au dessus du mot v et à trouver le bon décalage de u vers la droite pour que les symboles coïncident. Du point de vue algorithmique, il n'y a pas de décalage du motif u, mais on incrémente simplement l'index k du symbole de v à comparer. On commence évidemment par incrémenter k jusqu'à ce que $u_1 = v_k$, si c'est possible. On compare alors les symboles de u et v qui suivent à l'aide d'une variable d'indexation i pour le mot u jusqu'à ce que qu'ils soient distincts ou que l'on les ait tous reconnus, auquel cas on incrémente k à nouveau :

```
Algorithme EstPortion(u, v) : booléen
données
    u, v : mots
variables
    i, k : entier
-----------------------------------------
i ← 1
k ← 1
tantque (i ≤ |u| et k ≤ |v| − |u| + 1) faire
    si (u[i] = v[k + i − 1]) alors // on a trouvé k
        i ← i + 1
    sinon
        i ← 1 // on reprend au 1er symbole de u
        k ← k + 1 // on décale à droite
    fsi
ftq
renvoyer(i > |u|)
```

ALGO 5. Recherche d'une portion u dans v.

Dans le meilleur des cas le mot u est un préfixe de v et il suffit de comparer les n premiers symboles, donc $\check{T}(n,m) = \Theta(n)$. Dans le pire des cas, les deux mots u et v sont tous deux constitués par la répétition d'un même symbole, sauf le dernier symbole de u qui doit être différent, soit $u = x^{n-1}y$ et $v = x^m$ en notant x et y ces deux symboles. On recommence donc $m-n+1$ fois n comparaisons, soit une complexité quadratique $\hat{T}(n,m) = \Theta((n-m)m)$.

Exercice 5. [6pts] On suppose qu'une liste S contient une permutation des entiers $\{1,2,\dots,n\}$ et que l'indexation de la liste commence à 1. Soit k un entier, $k \geq 2$. On appelle *k-cycle* (ou cycle de longueur k) de la liste S, toute liste $[x_0, x_1, \dots, x_{k-1}]$ d'entiers tous distincts dans $\{1,2,\dots,n\}$ tels que

$$\forall i \in [0,k-1], \quad S[x_i] = x_{(i+1 \mod k)}.$$

Par exemple, la liste $[1,3,8,4]$ est un 4-cycle de la liste

$$S = [3,2,8,1,6,9,7,4,5].$$

Calculez l'autre cycle de cette liste.

Soit S une liste permutation de longueur n. Quelle est la longueur maximale d'un cycle ? Quelle liste ne contient aucun cycle ?

Écrivez un algorithme Cycles(S) qui renvoie la liste de tous les cycles de la liste S. Avant d'écrire votre algorithme, expliquez en français la méthode que vous allez employer. Quelle est la complexité de votre algorithme (justifiez) ?

Solution. L'autre cycle est $[5, 6, 9]$ puisque $S[5] = 6$, $S[6] = 9$ et $S[9] = 5$. La longueur maximale d'un cycle est n, par exemple le cycle $[2, 3, 4, \dots, n-1, n, 1]$. La liste identique $[1, 2, \dots, n]$ ne contient aucun cycle et il n'y en a pas d'autre puisque si l'on avait $S[i] = j$ avec $j \neq i$, alors i et j constitueraient le début d'un cycle dont la longueur serait ≥ 2.

L'algorithme consiste à calculer les itérés $S^k[x] = S[S[S[\dots[x]]]]$ jusqu'à ce que $S^k[x] = x$ ce qui détermine un cycle. Pour ne pas recalculer un même cycle plusieurs fois, on utilisera une liste *crible* de $\#S$ booléens pour éliminer les entiers des cycles obtenus. L'algorithme auxiliaire InitListe(L,n,x) crée une liste L de n éléments x, sa complexité est $\Theta(n)$. L'algorithme auxiliaire AjoutListe(L,x) rajoute l'élément x à la fin de la liste L, sa complexité est $\Theta(1)$.

La boucle principale à la ligne #5 vérifie qu'il reste encore des entiers à traiter dans la permutation, elle est décrémentée de la taille du cycle à la ligne #20. La boucle à la ligne #6 cherche le prochain début de cycle x_0 qui n'a pas encore été traité. La boucle à la ligne #12 construit le cycle courant avec les itérés de S. Le test à la ligne #17 ne rajoute le cycle dans la liste des cycles qu'à la condition que sa longueur est supérieure à 2.

Hors de la boucle principale, le coût est celui des initialisations sont celui des deux listes *crible* et *orbites*, soit $\Theta(n)$. Il est préférable de faire une analyse globale des instructions dans la boucle principale à la ligne #5, on ne sait pas combien de cycles seront calculés mais le coût reste largement indépendant du nombre de cycles de la permutation.

Notons que dans l'algorithme un cycle peut être de taille 1, c'est *a posteriori* que l'on vérifie (test #17) qu'il s'agit bien d'un cycle au sens de la définition pour le ranger ou non dans la liste des cycles. Compte tenu de cette remarque, tous les entiers de 1 à n auront été rangés dans leurs cycles respectifs. Notons k le nombre de cycles et p_i la taille du $i$ème cycle. On a

$$\sum_{i=1}^{k} p_i = n \tag{1}$$

Les instructions #9,#10 et #11 ainsi que #17, #18 et #20 ont un coût unitaire en $\Theta(1)$ et sont exécutées k fois donc en $\Theta(k)$. La boucle de la ligne #6 parcourt exactement une seule fois chaque booléen de la liste *crible*, soit un coût en $\Theta(n)$. Les instructions de la boucle #12 sont exécutées p_i fois pour le cycle i et donc globalement n fois d'après (1), donc avec un coût de $\Theta(n)$. Finalement, on a un coût global de

$$T(n) = \Theta(k) + \Theta(n) + \Theta(n) \tag{2}$$

$$= \Theta(k) + \Theta(n) \tag{3}$$

La complexité reste en $\Theta(n)$ quelle que soit le nombre de cycles k, dans le meilleur des cas 1 et dans le pire n.

```
Algorithme CYCLE(S) : liste
données
    S : liste // la permutation
variables
    crible : liste de booléens
    cycle : liste d'entiers // cycle courant
    orbites : liste de listes // liste des cycles
    n, x0 : entiers

n ← #S
INITLISTE(crible, n, faux)
INITLISTE(orbites, 0, 0)
x0 ← 1
tantque (n > 0) faire
    tantque (crible[x0]) faire
        x0 = x0 + 1
    ftq
    x ← x0
    INITLISTE(cycle, 1, x)
    crible[x] ← vrai
    tantque (S[x] ≠ x0) faire
        x ← S[x]
        AJOUTLISTE(cycle, x)
        crible[x] ← vrai
    ftq
    si (#cycle > 1) alors
        AJOUTLISTE(orbites, cycle)
    fsi
    n ← n − #cycle
ftq
renvoyer(orbites)
```

ALGO 6. Recherche des cycles d'une permutation