L3-INFO Algorithmique et Modélisation

Quick – 23 février 2024 – durée 1 h

Sont interdits : les documents, les ordinateurs, les téléphones (incluant smartphones, tablettes,... tout ce qui contient un dispositif électronique).
Seuls les dictionnaires papier pour les personnes de langue étrangère sont autorisés.
Il sera tenu compte de la qualité de la rédaction et de la clarté de la présentation (2 pts).
Une phrase courte et claire vaut mieux qu'une expression longue confuse avec des ratures.
En cas d'incompréhension du sujet, préciser les hypothèses de travail que vous faites et continuer.
Le barème **indicatif** : Exercice 1 : 5 pts (5 à 10 mn) ; Exercice 2 : 5 pts (10 à 20 mn), Exercice 3 : 8 pts (20 à 30 mn) Relecture ($\sim 5mn$)
Les 3 exercices sont indépendants.

I : Duplication

On considère un tableau de n éléments (type abstrait défini en amont) parmi lesquels il y a des éléments dupliqués (doublons). On souhaite identifier les éléments qui sont dupliqués. Par exemple, pour le tableau suivant, les éléments sont des caractères,

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	b	r	a	c	a	d	b	r	a		d	i	t	$-$	e	l	l	e

on construit un nouveau tableau constitué des éléments apparaissant au moins 2 fois dans le tableau donné

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	b	r	d	e	l	/	/	/	/	/	/	/	/	/	/	/	/	/

I.1. Écrire un algorithme efficace qui renvoie un tableau contenant les éléments qui sont dupliqués, le tableau renvoyé ne contiendra pas de doublons, on ne pourra parcourir le tableau en entrée qu'une seule fois. Préciser avec soin les structures de données auxiliaires utilisées.

Éléments de réponse : Comme on ne fait qu'un seul passage sur le tableau, on doit déterminer si l'élément considéré est vu pour la première fois, ou la deuxième fois ou plus, on peut utiliser une ou des tables de hachage. Une première solution serait de compter le nombre d'occurrences et de sélectionner au fur et à mesure les éléments doublons. Cela suppose la possibilité de hacher les éléments, c'est à dire que chaque élément possède une unique clé qui servira pour le hachage.
On suppose que l'on dispose de tables de hachage avec les opérateurs suivants :
- `Créer_Table_Hachage` () : Crée une nouvelle table de hachage de capacité suffisante
- `Recherche_Table` (t, e) : recherche si un élément e est présent dans la table t
- `Insérer_Table_Hachage` (t, e) : Insère un élément e dans la table t

Extrait_Doublon (T)
Données : Un tableau $T = [e_1, e_2, \cdots, e_n]$ de n éléments
Résultat : Un tableau contenant tous les éléments au moins en double dans T

```
// Initialisation des structures auxiliaires
t_élem = Créer_Table_Hachage ()
t_doublons = Créer_Table_Hachage ()
T_res = init (taille (T),"/")
i = 1 // indice du prochain élément doublon détecté dans T_res
// parcours du tableau d'éléments
for i = 1 to taille (T) do
    if Recherche_Table (t_élem,T[i])
        if not Recherche_Table (t_doublons,T[i])
            Insérer_Table_Hachage (t_doublons,T[i])
            T_res[i] = e
            i = i + 1
    else
        Insérer_Table_Hachage (t_élem,T[i])
return T_res
```

I.2. Évaluer la complexité en moyenne de cet algorithme en nombre d'opérations effectuées (on précisera les opérations choisies pour l'évaluation).

Éléments de réponse : En supposant que la table de hachage est de taille suffisante (taille supérieure au nombre d'éléments du tableau), la recherche et l'insertion d'un élément sont des opérations en $\mathcal{O}(1)$ en moyenne.
À chaque itération on effectue une ou deux opérations de recherche ou d'insertion, donc comme on itère une et une seule fois sur chaque élément du tableau on obtient un coût moyen de $\mathcal{O}(n)$. Comme chaque élément du tableau doit être examiné (il peut potentiellement être un doublon) le coût au mieux est n, donc en moyenne la complexité est en $\Theta(n)$.

I.3. Évaluer la complexité en mémoire de cet algorithme.

Éléments de réponse : On utilise dans notre cas 2 tables de hachage, la première $t_{\text{élem}}$ devra contenir tous les éléments de T et assurer un temps constant pour les opérations de recherche et d'insertion, sa taille doit être de l'ordre au moins de n, de même la table t_{doublons} peut contenir au plus $\frac{n}{2}$ éléments qui sont des doublons, sa taille doit également être de l'ordre de n. De même le tableau résultat a la taille du tableau initial (on aurait pu prendre la taille divisée par 2), cela reste de l'ordre de $\mathcal{O}(n)$
En conclusion, pour notre choix d'algorithme, on utilise un espace mémoire en $\mathcal{O}(n)$.

II : Une histoire de dés

On dispose d'un dé avec 6 faces et on souhaite simuler un dé à 8 faces à partir de lancers de dés à 6 faces. Pour formaliser le problème, on dispose d'une fonction `de6()` qui renvoie une valeur entière entre 1 et 6. On modélise une séquence d'appels à la fonction `de6` par une suite de variables aléatoires indépendantes de même loi uniforme sur $\{1, 2, 3, 4, 5, 6\}$.

II.1. À partir de la fonction `de6()`, écrire une fonction `de8()` simulant un dé avec 8 faces équiprobables.

Éléments de réponse : Une solution élémentaire serait de considérer le dé à 6 faces comme une pièce de monnaie, c'est à dire fournissant un bit aléatoire. Il faudrait alors reprendre la preuve faite en cours qui génère un dé à 8 faces à partir d'une pièce

`pièce` ()
Résultat : Une valeur dans $\{0, 1\}$ distribuée uniformément, les appels successifs sont indépendants
return `dé6` () $\leqslant 3$

On aurait pu également renvoyer la parité de la valeur du dé.

Pour la preuve de cet algorithme voir le cours.

`dé8` ()
Résultat : Une valeur dans $\{1, \cdots, 8\}$ distribuée uniformément, les appels successifs sont indépendants
return `pièce` () + 2*`pièce` () + 4*`pièce` () +1

Idem, c'est dans le cours.

II.2. Évaluer le nombre moyen d'appels à `de6()` pour simuler une valeur d'un dé à 8 faces. Commenter votre résultat.

Éléments de réponse : Dans la solution proposée ci-dessus on utilise 3 appels à `dé6`, en fait on n'utilise pas tout l'aléatoire contenu dans les 6 valeurs. Il est clair que l"on a besoin d'au moins 2 appels à `dé6` pour générer un dé à 8 faces (il n'y a pas assez d'information dans un dé à 6 faces pour produire 8 valeurs de même probabilité.

2 appels à `dé6` produit un couple de loi uniforme sur $\{1, \cdots, 6\} \times \{1, \cdots, 6\}$, c'est à dire, modulo un calcul en base six un nombre entre 0 et 35. On peut ainsi appliquer une méthode à base de rejet

`dé8` ()
Résultat : Une valeur dans $\{0, 1\}$ distribuée uniformément, les appels successifs sont indépendants
repeat
1 $x =$ (`dé6` ()-1)+6$*$ (`dé6` () -1)
until $x \leqslant 31$
2 `// fin du rejet`
3 $y = x$ modulo 8
4 **return** $y + 1$

La preuve de cet algorithme se fait en 3 étapes
ligne 1 : x valeur générée suit une loi uniforme sur $\{0, \cdots, 35\}$ (cf cours)
ligne 2 : à la fin de l'itération x suit une loi uniforme sur $\{0, \cdots, 31\}$ (cf cours)
ligne 3 : x suit une loi uniforme sur $\{0, \cdots, 31\}$, donc y suit une loi uniforme sur $\{0, \cdots, 7\}$
ligne 4 : il suffit d'ajouter 1 pour avoir une valeur distribuée uniformément sur $\{1, \cdots, 8\}$
Dans ce cas la probabilité d'acceptation est $p_a = \frac{32}{36} = \frac{8}{9}$, le nombre moyen d'appels à `dé6` sera donc

$$2 * \frac{1}{p_a} = \frac{9}{4} = 2,25$$

ce qui est meilleur que la solution initiale. On peut encore réduire la complexité en remarquant que lorsque l'on rejette on dispose de 4 valeurs uniformément distribuées sur $32, 33, 34, 35$ soit l'équivalent de 2 bits aléatoires. Il suffit de tirer un 3ième bit aléatoire pour avoir un dé à 8 faces.

```
dé8 ()
    Résultat : Une valeur dans {0, 1} distribuée uniformément, les appels successifs sont
               indépendants
    x = dé6 () +6* dé6 ()
    if x ⩽ 31
      return 1 + x modulo 8
    else
      if dé6 ()⩽ 3
        return x − 30
      else
        return x − 30 + 4
```

Dans ce cas la complexité est

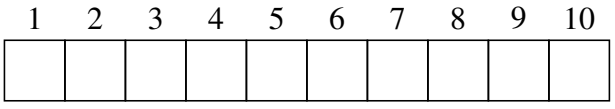
$$2p_a + 3(1 - p_a) = 2\frac{32}{36} + 3\frac{4}{36} = \frac{19}{9} = 2,1111...$$

Peut-on faire mieux ? sans doute, car on perd de l'aléatoire sur le dernier dé.

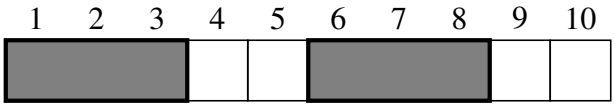
Par contre on est certain de faire au plus 3 alors que le nombre de rejets pour la méthode de base suit une loi géométrique donc peut prendre des valeurs au delà de 3.

III : Places de parking

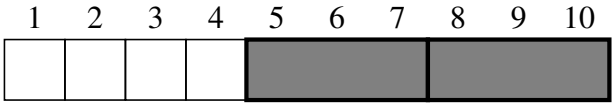
On considère un parking constitué de places en ligne. On peut garer des camions sur ce parking sauf qu'ils prennent 3 places de voiture.



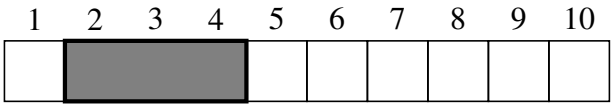
Parking vide



Parking avec 2 camions garés en position 1 et 6



Parking avec 2 camions garés en position 5 et 8



Parking avec 1 camion garé en position 2

Une configuration du parking correspond au placement de différents camions sur le parking, évidemment 2 camions ne peuvent pas occuper une même place et il n'est pas nécessaire de remplir le parking au maximum. On présente, dans les exemples ci-dessus, 4 configurations possibles pour un parking de 10 places. Ce ne sont que des exemples, il y a d'autres configurations, un parking de 10 places permettant de garer au maximum 3 camions.

On note n la longueur du parking en nombre de places et C_n le nombre de configurations possibles du parking.

III.1. Proposer un algorithme pour calculer C_n et calculer C_{10}. Pour cela on pourra écrire une équation de récurrence vérifiée par les C_n. Pour construire cette équation de récurrence considérer la place 1 du parking elle est soit occupée par un camion soit elle n'est pas occupée par un camion.

Éléments de réponse : On peut remarquer que si la dernière place est occupée par un camion, cela revient à regarder un parking de taille $n-3$

En reprenant cette idée le nombre total de configurations vérifie l'équation de récurrence

$$C_n = \underbrace{C_{n-1}}_{\text{il a une voiture sur la place } n} + \underbrace{C_{n-3}}_{\text{il a un camion sur la place } n \text{ ainsi que sur les places } n-1 \text{ et } n-2}$$

Cette équation de récurrence n'est pas valide lorsque l'on ne peut pas placer un camion, c'est à dire $n = 0,\ 1$ ou 2. On obtient ainsi

$$\begin{aligned} &C_0 = C_1 = C_2 = 1 \text{ cas de base} \\ &C_n = C_{n-1} + C_{n-3} \text{ pour } n \geqslant 3 \end{aligned} \tag{1}$$

On peut alors écrire un programme récursif pour calculer C_n

```
Configurations_Parkings (n)
    Données : n taille du parking
    Résultat : Nombre de configurations possibles du parking

    if n ⩽ 2
        return 1
    else
        return Configurations_Parkings (n − 3) +
         Configurations_Parkings (n − 1)
```

Évidemment ce n'est pas très efficace car on répète des calculs lors des appels récursifs. On peut alors mémoïser les calculs intermédiaires.

Ce schéma de récurrence se prête bien à de la programmation dynamique,

```
Configurations_Parkings (n)
    Données : n taille du parking
    Résultat : Nombre de configurations possibles du parking

    T = init (n,1)          // tableau indicé de 0 à n initialisé à 1
    if n ⩾ 3
        for i = 3 to n do
            T[i] = T[i − 1] + T[i − 3]
    return T[n]
```

et on peut même replier la mémoire (comme pour le calcul des nombres de Fibonacci), il suffit de travailler avec un tableau circulaire de taille 4

III.2. Évaluer la complexité de votre algorithme et commenter cet algorithme.

Éléments de réponse : Lorsque l'implémentation utilise la programmation dynamique ou la mémoïsatiion la complexité est de l'ordre de $\Theta(n)$. On est de l'ordre de $\Theta(n)$ en mémoire et si on effectue un repliement de la mémoire on passe en $\Theta(1)$.

Si on effectue le schéma récursif sans la mémoïsation le nombre d'opérations d'additions est exponentiel (supérieur à $2^{\frac{n}{3}}$

III.3. Écrire un algorithme qui énumère toutes les configurations possibles d'un parking de taille n.

Éléments de réponse : On procède de la même manière que pour l'énumérations des parties. On va généraliser à une liste de tailles de véhicules possibles

```
def enumerer_parkings(taille_parking, tailles_vehicules,
                      parking = None, i = None):
    """ enumere toutes les configuration de parkings
    paramètres:
        taille_parking : taille du parking
        taille_vehicules: liste des tailles de véhicule
        parking : liste des places de parking occupées
                  avec la longueur des véhicules
        i : indice de la prochaine place disponible
    pré-conditions :
        0<= i <=  taille_parking
        les places de parking de 0 à i-1 sont
            marquées par des valeurs de t
    valeur de retour:
        liste de toutes les configurations possibles de parking
    """
    # initialisation
    parking = [0]*taille_parking if parking is None else parking
    i = 0 if i is None else i

    if i == taille_parking: # cas de base
        res = [parking[:]]
        # copie de l'état du parking dans la liste résultat
    else:
    # énumération de la partie restante du parking
    # pour les différentes tailles de véhicules possibles
    # en place i
        res = []
        for t in tailles_vehicules:
            if i + t -1 < taille_parking:
                # on peut placer un véhicule de longueur t en i
                for j in range(i, i + t):
                    parking[j] = t
                # on marque les t places
                res = res + enumerer_parkings(taille_parking,
                            taille_vehicule, parking, i + t)
                # énumération à partir de i+t
    return res
```

III.4. Rédiger la preuve de cet algorithme.

Éléments de réponse : Pour la preuve de cet algorithme il faut montrer que l'appel initial `enumerer_parkings(n,tailles_véhicules)` énumère toutes les configurations du parking dans lesquelles les véhicules ont une longueur dans la liste `tailles_véhicules`.

Comme l'algorithme est récursif il faut d'une part faire la preuve de la correction partielle et d'autre part prouver la terminaison.

— **Correction partielle**
On généralise la propriété demandée par l'appel `enumerer_parkings(taille_parking, tailles_vehicules, parking, i)` énumère toutes les configurations de parking de taille `taille_parking` dont le préfixe de 0 à $i-1$ est fixé par la variable parking.

Lorsque la configuration du parking est fixée de 0 à $i-1$ la boucle for divise l'ensemble des configurations en fonction de la taille du véhicule t dont le début est placé en i et la fin en $i+t-1$, on vérifie dans le if que le véhicule ne "dépasse" pas du parking. Lors de l'appel récursif la pré-condition est vérifiée, l'appel récursif renvoie la liste de toutes les configurations dont le préfixe est donné par parking de 0 à $i-1$. Le cumul se fait dans la variable `res`, les configurations trouvées sont différentes car les appels récursifs produisent des configurations différentes. Toutes les configurations sont traitées car les appels récursifs renvoient toutes les configurations de préfixe donné et les configurations traitées dans chaque itération sont différentes car de préfixe différent, et tous les préfixes possibles sont traités. Ce qui assure la correction partielle, c'est à dire que si l'algorithme se termine et que les appels internes sont corrects alors l'appel `enumerer_parkings` est correct.

— **Terminaison**
La quantité `taille_parking`$-i$ est positive strictement décroissante le long des branches de l'arbre des appels. Comme l'algorithme atteint le cas de base lorsque $n-i \leqslant 2$ l'algorithme se termine.

— **appel initial**
`enumerer_parkings(n,tailles_véhicules,`$[0,\cdot,0]$`,0)` énumère toutes les configurations dont le préfixe est vide et la première place est disponible c.q.f.d

III.5. Que faudrait-il modifier à cet algorithme pour énumérer les configurations ayant exactement 2 camions ?

Éléments de réponse : Il suffit de tenir à jour un vecteur de comptage des véhicules de taille t et d'effectuer les appels récursif lorsque l'existence d'une solution est possible.