Master 1 STL 4I505 CPA – Examen 31 Mai 2018

Les documents distribués dans le cadre du cours et les notes personnelles sont autorisés.

Partie B

0.1 Un mauvais algorithme pour *k-means* (1 point)

Le problème de *k-means* consiste en un ensemble donné X de n points dans l'espace $\mathbb{R}^2$ et un entier $k \geq 1$. On veut partitionner X en k ensembles $X_1, \ldots, X_k$ et trouver k centres $c_1, \ldots, c_k \in \mathbb{R}^2$ tel que la valeur objectif suivante

$$\sum_{i=1}^{k} \sum_{x \ in X_i} ||x - c_i||^2$$

soit minimale. Ici $||x - c_i||$ dénote la distance Euclidienne entre les points x et c_i.

Pour approcher ce problème NP-dur, généralement on commence par générer une première solution, qu'on améliore ensuite par une recherche locale. Comme on ne sait pas évaluer les améliorations obtenues par la recherche locale, on se contente d'analyser la génération de la solution initiale. Voici deux algorithmes de génération possibles.

k-means++

1. Choisir c_1 uniformément au hasard parmi X.
2. Pour tout $i = 2, \ldots, k$, choisir $c_i = x \in X$ avec probabilité proportionnelle à $D^2(x)$, où $D(x)$ dénote la distance entre x et le centre le plus proche parmi ceux déjà sélectionnés, donc $D(x) := \min_{\{j=1,\ldots,i-1\}} \|x - c_j\|$ dans ce cas.

k-means-stupide

1. Choisir c_1 uniformément au hasard parmi X.
2. Pour tout $i = 2, \ldots, k$, choisir $c_i = x \in X$ tel qu'il maximise $D(x)$.

Pour le premier algorithme k-means++ il a été montré qu'il produit une solution dont la valeur objectif est en espérance au plus $8(\ln k + 2)$ fois plus grande que la solution optimale.

Montrez à l'aide d'un exemple que le deuxième algorithme k-means-stupide peut produire une solution avec une valeur objectif arbitrairement grande par rapport à la solution optimale.

0.2 Déplacement droite-bas dans une grille (1 points)

Soit une grille composée de n colonnes et m lignes, voir Figure 1. Chaque case est identifiée par une coordonnée (i, j) où i est l'indice de ligne et j l'indice de colonne. La case $(1, 1)$ se trouve en haut à gauche et la case (m, n) en bas à droite.

Un fromage se trouve dans la case (m, n), et seulement dans cette case. Il existe une souris qui initialement se trouve dans la case $(1, 1)$. Elle veut atteindre la case avec le fromage en effectuant des déplacements successifs très simples, soit vers la case adjacente droite soit vers la case adjacente en dessous. Formellement si la souris est dans la case (i, j), alors elle peut atteindre en un pas la case $(i + 1, j)$ pourvu que $i < m$ et la case $(i, j + 1)$ pourvu que $j < n$.

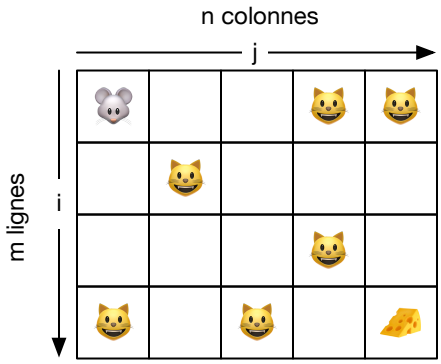


Figure 1: Une instance du problème de la question 0.2

L'histoire serait trop ennuyeuse sans les chats. Certaines cases contiennent des chats. Les chats ne se déplacent pas, mais la souris doit éviter les cases avec les chats. Les cases $(1, 1)$ et (m, n) sont dépourvus de chats.

Concrètement on vous donne $n \geq 2$ et $m \geq 2$ ainsi qu'une matrice booléenne C de dimension $m \times n$ tel que $C[i, j]$ =Vrai si et seulement si la case (i, j) contient un chat. Vous devez concevoir un algorithme qui décide si la souris peut atteindre le fromage avec seulement des pas vers la droite ou vers le bas en évitant les chats.

Ce problème pourrait se modéliser comme un problème de connectivité dans un graphe orienté, mais une solution par programmation dynamique est attendue, de complexité $O(nm)$. Donnez un algorithme pour ce problème, pas du code.

Indice: déterminez pour chaque case (i, j) si la souris peut atteindre cette case avec les restrictions mentionnées.

0.3 Déplacement gauche-droite-bas dans une grille (2 points)

Cette fois-ci la souris est plus intelligente et peut faire des déplacements vers la gauche, la droite ou vers le bas. Par contre elle ne peut pas remonter, voir Figure 2. Trouvez de nouveau une solution par programmation dynamique de complexité $O(nm)$.

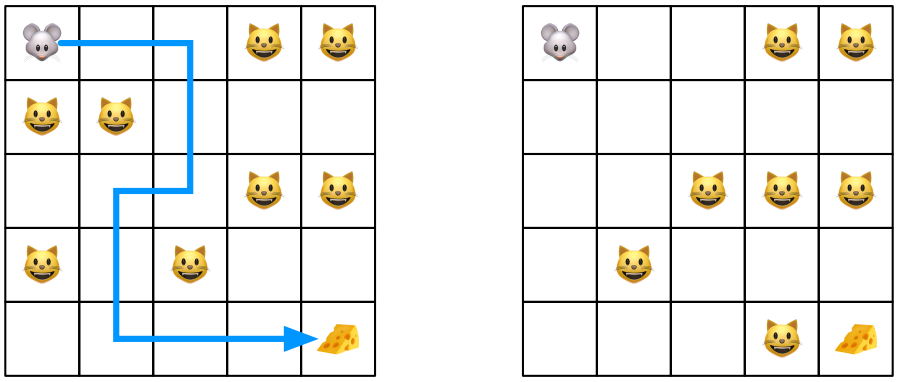


Figure 2: Des instances au problème de la question 0.3. L'instance de gauche a une solution, alors que l'instance de droite n'en a pas, car la souris ne peut faire des pas vers le haut, ni se déplacer en diagonale.

Indice: déterminez pour chaque case (i, j) et direction $d \in \{\text{GAUCHE,DROITE,BAS}\}$ si la souris peut atteindre cette case par un chemin qui satisfait les restrictions mentionnées et qui termine par un pas de la direction d. Indiquez clairement dans quel ordre les variables doivent être calculées.

0.4 Pavage par des dominos (2 points)

On vous donne une grille composée de 4 lignes et de n colonnes avec $n \geq 2$. On veut *paver* cette grille avec des *dominos*, chaque domino couvre exactement 2 cases adjacentes verticalement ou horizontalement. Paver veut dire que chaque case de la grille est couverte par exactement un domino.

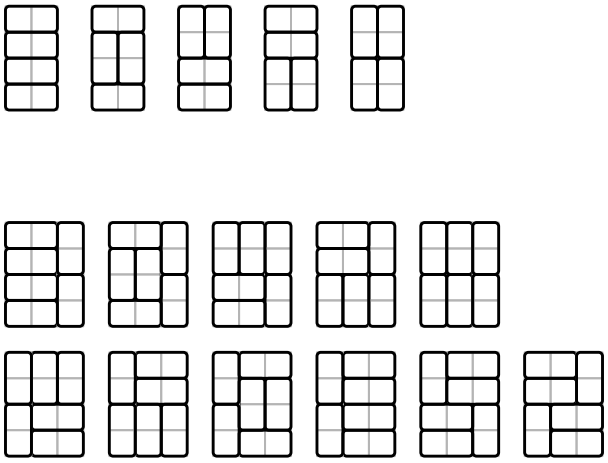


Figure 3: Les 5 pavages pour $n = 2$ et les 11 pavages pour $n = 3$.

Étant donné n, calculez par programmation dynamique le nombre de pavages possibles,

modulo 1000007. Votre algorithme devrait avoir une complexité $O(n)$. Notez: une solution en temps $O(\log n)$ est possible, mais c'est hors sujet. Notez: le résultat est attendu modulo 1000007 juste pour s'assurer qu'il tient dans un entier 32 bits.

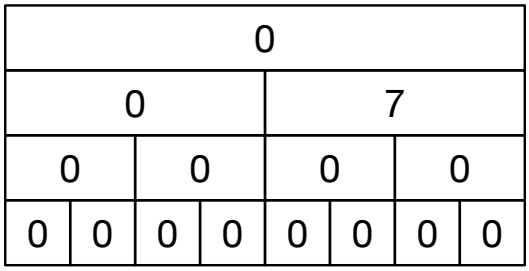
Donnez le cas de base, la récursion et expliquez votre solution.

0.5 Arbre de segments (4 points)

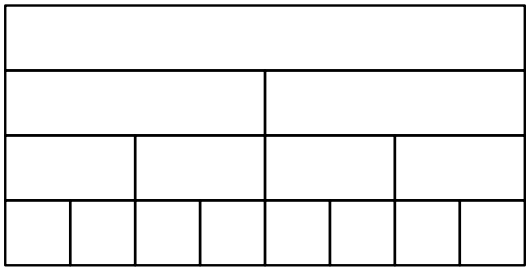
Rappel Un arbre de segments est une structure de donnée qui permet de stocker un tableau t de n entiers — avec n étant une puissance entière de 2 — et qui permet les opérations suivantes chacune en temps $O(\log n)$: (1) demander la valeur $t[i]$ pour un indice i donnée, et (2) ajouter une valeur Δ à toutes les entrées $t[i], t[i+1], \ldots, t[j-1]$ pour $\langle i, j, \Delta \rangle$ données. Cette structure consiste en un arbre binaire complet avec n feuilles. Chaque nœud est responsable d'un segment du tableau t. Formellement un *segment* est un intervalle de la forme $[i, j)$ tel que $j - i$ soit une puissance entière de 2 et que i soit multiple de $j - i$. La racine est responsable de tout le tableau, les feuilles sont responsables d'un seul indice de t, et les fils d'un nœud interne p sont responsables respectivement des deux moitiés du segment dont p est responsable. Chaque nœud p contient une étiquette représentant une valeur à ajouter à toutes les entrées du segment de t concerné par p.

(1) La valeur $t[i]$ pour un indice i est obtenu en sommant les étiquettes le long du chemin de la racine vers la feuille responsable de l'élément à l'indice i. (2) Une mise à jour avec les paramètres $\langle i, j, \Delta \rangle$ est faite en décomposant l'intervalle demi-ouvert $[i, j)$ en un nombre minimal de segments, et la valeur Δ est ajoutée aux nœuds correspondants.

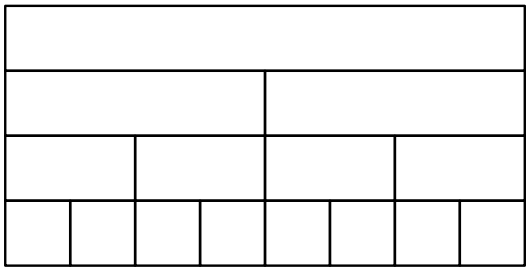
Exemple: l'arbre de segments suivant représente un tableau t composé de 4 fois le nombre 0 suivi de 4 fois le nombre 7.



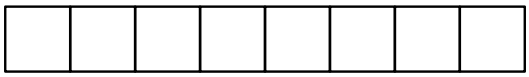
Quel sera le tableau après une mise à jour aux paramètres $\langle 0, 2, +3 \rangle$? Répondez directement sur cette feuille.



Quel sera le tableau après une mise à jour aux paramètres $\langle 1, 6, -5 \rangle$? Répondez directement sur cette feuille.



Quel est le contenu du tableau t représenté par l'arbre obtenu après ces opérations?



Expliquez pourquoi l'opération de mise à jour peut être effectuée en temps $O(\log n)$.

1 Éléments de correction

1.1 Un mauvais algorithme pour *k-means* (1 point)

Pour cette question il faut bien comprendre que les points sont affectés aux centres les plus proches et il faut montrer que le choix des centres n'est pas bien.

Fixons $k = 2$ pour simplifier. Considérons un premier exemple. Par $[i]$ on dénote une collection de i points à distance au plus ϵ les uns des autres.

```
[1]           [n]            [1]
 <----- n ----> <---- n ----->       (distances)
```

Une solution optimale consiste à choisir un centre dans l'amas du centre et un centre sur le point tout à gauche. Son coût est de $\Theta(n + \epsilon n)$.

Mais si l'algorithme choisit le premier centre tout à gauche et l'autre du coup tout à droite, le coût sera $\Theta(n^2)$. Le rapport entre les coût est de l'ordre de n qui peut être arbitrairement grand, quand n tend vers l'infini.

Le problème avec cet exemple est que l'algorithme k-means-stupide choisira un point extrême pour le premier centre seulement avec une probabilité $2/(n + 2)$, donc très petite. Un meilleur exemple force l'algorithme à faire un mauvais choix avec grande probabilité. Ici avec probabilité $1/2$ il choisir un centre dans l'amas tout à gauche et le deuxième centre tout à droite, pour un coût espéré de $\Omega(n^2)$, ce qui donne de nouveau un rapport $\Omega(n)$.

```
[n]           [n-1]           [1]
 <----- n ----> <---- n ----->       (distances)
```

1.2 Déplacement droite-bas dans une grille

Ceci est un classique de la programmation dynamique. La souris peut atteindre une case (i, j) si elle y est déjà (donc $(i, j) = (1, 1)$) où si elle peut venir de la gauche ou du haut. Formellement notons $A[i, j]$ l'accessibilité de la case (i, j). Le but est de calculer $A[m, n]$. La clé est de remplir la matrice dans l'ordre des ligne, puis colonnes de telle sorte à ce que les entrées nécessaires au calcul de $A[i, j]$ aient déjà été calculés.

```
pour toutes les lignes i de 1 à m
   pour toute les colonnes j de 1 à n
      Si pas de chat en (i,j) et ( (i,j)=(1,1) ou
                                   (i > 1 et A[i - 1, j]) ou
                                   (j > 1 et A[i, j - 1])
      alors A[i,j] := Vrai
      sinon A[i,j] := Faux
retourner A[m,n]
```

Il y a mn variables, chacune se calcule en temps constant, la complexité du programme dynamique est $O(mn)$.

1.3 Déplacement gauche-droite-bas dans une grille

Si on garde le même principe, on rencontre une difficulté. Le calcul de $A[i,j]$ dépendra de $A[i,j+1]$ qui à son tour dépendra de $A[i,j]$. Pour casser cette dépendance circulaire, on peut calculer plusieurs variables par case. Notons $D[i,j]$ la propriété que la case (i,j) peut être atteinte avec une série de pas dont le dernier est un pas vers la droite. Similairement on note $B[i,j]$ et $G[i,j]$ pour un dernier pas vers le bas ou la gauche. Alors on a $A[i,j] = G[i,j] \vee B[i,j] \vee D[i,j]$. La clé est que $D[i,j]$ ne dépend que de D et B pour des entrées de la même ligne, mais des colonnes précédentes. Similairement $G[i,j]$ ne dépend que de G et B pour des entrées de la même ligne, mais des colonnes suivantes. Et finalement $B[i,j]$ ne dépend que de B, G, D de la case $(i-1,j)$. Il faut donc procéder ligne par ligne, en calculant B, G, D dans des boucles indépendantes.

```
Dans ce qui suit A[i,j] est un raccourci pour
          G[i,j] ou B[i,j] ou D[i,j] ou (i,j)=(1,1)

pour toutes les lignes i de 1 à m
   pour toute les colonnes j de 1 à n
      B[i,j] = non Chat[i,j] et i > 1 et A[i-1,j]
   D[i,1] = faux
   pour toute les colonnes j de 2 à n
      D[i,j] = non Chat[i,j] et (D[i,j-1] ou B[i,j-1])
      // plusieurs ou un seul dernier pas à droite
   G[i,n] = faux
   pour toute les colonnes j de n-1 à 1 en sens descendant
      G[i,j] = non Chat[i,j] et (G[i,j+1] ou B[i,j+1])
      // plusieurs ou un seul dernier pas à gauche

retourner A[m,n]
```

L'analyse de la complexité est identique à la question précédente.

1.4 Pavage par des dominos (2 points)

Pour faciliter les notations on tourne la grille, qui a donc 4 colonnes et n lignes. Notons par O, E les cases d'un domino horizontal et N, S ceux d'un domino vertical, pour Ouest, Est, Nord, Sud. Alors une ligne d'un pavage valide correspond à un mot de 4 lettres sur l'alphabet $\{O, E, N, S\}$ avec la restriction que chaque O est suivi d'un E et vice-versa. Soit $\mathcal{C}$ l'ensemble de ces mots. On peut définir une matrice M de dimension $|\mathcal{C}| \times |\mathcal{C}|$ tel que $M[a,b]$ est 1 si la configuration a peut être suivi de la configuration b et $M[a,b] = 0$ sinon. Formellement $M[a,b] = 1$ si pour tout $k \in \{1,2,3,4\}$, $a[k] = N \Leftrightarrow b[k] = S$.

Maintenant calculons $A[n,a]$ le nombre de pavages étendues d'une grille $4 \times n$ où chaque case est couverte par exactement un domino et les dominos peuvent dépasser la dernière ligne, et la dernière ligne a la configuration a.

Pour le cas de base: $A[1, a]$ est 1 si a est sans la lettre S et 0 sinon. Pour la récursion et $n > 1$ on a: $A[n, a] = \sum_b A[n-1, b]$ où la somme est prise sur tous les $b \in \mathcal{C}$ tel que $M[a, b] = 1$.

Le but est de calculer $\sum_a A[n, a]$ où la somme est prise sur $a \in \mathcal{C}$ sans la lettre N.

Comme $\mathcal{C}$ a une taille constante, chacune des $O(n)$ variables peut être calculée en temps constant.

Bonus: Pour une solution en $O(\log n)$ il suffit d'observer que le nombre de pavages de la grille $4 \times n$ est égal au nombre de pavages de la grille $4 \times (n+2)$ avec la configuration $a = OEOE$ dans la première et dernière ligne. Et ce nombre est précisément $M_{a,a}^{n+1}$. La matrice M^{n+1} peut-être calculée par exponentiation rapide en temps $O(\log n)$. Il suffit de calculer successivement les matrices $M, M^2, M^4, M^8, \ldots$ et de multiplier ceux dont l'exposant apparaît dans la décomposition binaire de $n+1$.

1.5 Arbre de segments

Attention les intervalles $[0, 2)$ et $[1, 6)$ sont demi-ouverts.

<table>
<tr><td colspan="8">0</td></tr>
<tr><td colspan="4">0</td><td colspan="4">7</td></tr>
<tr><td colspan="2">2</td><td colspan="2">0</td><td colspan="2">0</td><td colspan="2">0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

<table>
<tr><td colspan="8">0</td></tr>
<tr><td colspan="4">0</td><td colspan="4">7</td></tr>
<tr><td colspan="2">2</td><td colspan="2">-5</td><td colspan="2">-5</td><td colspan="2">0</td></tr>
<tr><td>0</td><td>-5</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

<table>
<tr><td>2</td><td>-3</td><td>-5</td><td>-5</td><td>2</td><td>2</td><td>7</td><td>7</td></tr>
</table>

Pour l'analyse de la complexité, on considère le sous-arbre A composé des nœuds traités lors des appels récursifs de la mise à jour. Comme tout arbre les nœuds de A de décomposent en feuilles et en nœuds internes. Pour l'analyse il suffit de borner le nombre de nœuds internes de A, le nombre de feuilles étant juste un nœud de plus.

Soit p un nœud responsable de l'intervalle A appelé pour une mise à jour sur un intervalle I. Si $A \subseteq I$ ou $A \cap I = \emptyset$ alors le parcours d'arbre termine ici, dans les autres cas, deux appels récursifs sont effectués à partir du nœud p sur les descendants de p.

Pour un niveau donnée dans l'arbre soit p le nœud responsable de l'intervalle contenant $\min I$ et q le nœud responsable de l'intervalle contenant $\max I$. Il est possible que $p = q$. Alors par l'observation précédente ce sont les seuls nœuds de l'arbre où le parcours de l'arbre ne termine pas nécessairement. Donc il y a au plus $2 \log n$ nœuds internes dans A, ce qui termine la preuve.