Algorithmique I
Année 2008-2009

A. Benoit, B. Depardon
C. Mouilleron, C. Rezvoy

TD n°8

1 Partiel 2008-2009, le retour !

1.1 Un jeu de patience

Question 1.1

On considère un jeu de n cartes, dans lequel à chaque carte est associée une valeur entière. Le jeu est mélangé et initialement posé à l'envers sur la table (on ne connaît alors la valeur d'une carte que lorsqu'on la retourne). On souhaite trier les cartes en utilisant des piles de cartes posées sur la table. La construction des piles doit respecter les règles suivantes.

- Initialement il n'y a aucune pile. La première carte tirée forme une nouvelle pile (on voit la valeur de la carte lorsqu'on la tire) : on pose la carte sur la table avec la face indiquant la valeur vers le haut (la valeur est donc visible).
- Chaque nouvelle carte peut être posée soit sur une pile existante dont la carte du dessus de pile a une valeur supérieure à la valeur de la carte tirée, soit sur une nouvelle pile à droite des autres.
- Lorsqu'il n'y a plus de cartes à tirer on s'arrête.

Proposez un algorithme qui trie le jeu de cartes. On commencera par créer des piles (le plus petit nombre possible) en se basant sur les règles précédentes. Une fois les piles de cartes créées, on ne peut voir que la valeur des cartes du dessus, puis on dépile pour effectuer le tri. On ne peut prendre à chaque étape qu'une carte se situant sur le dessus d'une pile, et on veut avoir à la fin en main les cartes triées par ordre croissant. Quelle est la complexité de cet algorithme ? Précisez les structures de données utilisées. Donnez un exemple où votre algorithme se comporte mal (*i.e.,* il atteint sa complexité dans le pire cas).

Question 1.2

Un arbre T de van Emde Boas **[?]** est une structure permettant de maintenir une liste triée d'entiers compris dans l'intervalle $[1, n]$ avec un temps $O(\log \log n)$ pour l'insertion et la suppression. On définit les opérations suivantes :

- $insérer(i)$: insérer i dans T, en $O(\log \log n)$
- $supprimer(i)$: supprimer i de T, en $O(\log \log n)$
- Si i est déjà inséré dans T, $suivant(i)/précédent(i)$: obtenir le suivant/précédent de i dans T (et retourne NIL si le suivant/précédent n'existe pas), en $O(1)$. $suivant(i)$ correspond à la valeur supérieure à i, la plus proche de i, présente dans T. $precedent(i)$ correspond à la valeur inférieure à i, la plus proche de i, présente dans T.

On dispose maintenant d'un jeu de cartes ayant des valeurs de 1 à n. En vous basant sur un arbre de van Emde Boas, proposez un algorithme en $O(n \log \log n)$ permettant de construire le plus petit nombre de piles de cartes respectant les règles de construction (on ne demande pas ici de trier les cartes, juste de construire les piles). La structure utilisée pour représenter les piles est laissée au choix.

Question 1.3

En utilisant votre algorithme précédent pour créer les piles de cartes avec des valeurs dans $[1, n]$, proposez un algorithme pour trier des cartes ayant des valeurs dans $[1, n]$ en $O(n \log \log n)$.

1.2 Multiplication de deux entiers

Soient deux entiers x et y codés sur n bits (on supposera que n est une puissance de 2). On souhaite multiplier ces deux entiers entre eux, en travaillant au niveau des bits. La multiplication de deux entiers de n bits se fait de manière triviale en $O(n^2)$, la multiplication d'un entier par une puissance de 2, et l'addition de 2 entiers se font en temps linéaire $O(n)$.

Question 1.4

La méthode diviser pour régner n'est pas toujours meilleure qu'un algorithme naïf. Pour illustrer cela, donnez un algorithme diviser pour régner ayant une complexité en $O(n^2)$ pour multiplier x et y, 2 entiers de n bits.

Question 1.5

Proposez un algorithme diviser pour régner ayant une complexité inférieure à $O(n^2)$. Donnez la formule de récurrence pour votre algorithme et sa complexité finale (en O). On supposera pour simplifier que l'addition/multiplication de deux nombres de taille n est un nombre de taille n.

1.3 Plus longue sous suite croissante

On considère une suite de n entiers : $a_1, a_2, \ldots, a_n$. Une sous séquence est un sous ensemble d'entiers de la suite, pris dans le même ordre que la suite : $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$, avec $1 \leq i_1 \leq i_2 \leq \cdots \leq i_k \leq n$. Une sous séquence croissante est une sous séquence où les entiers sont strictement croissants. Par exemple, la plus grande sous séquence croissante de la suite $5, 2, 8, 6, 3, 6, 9, 7$ est $2, 3, 6, 9$.

Question 1.6

Donnez une plus longue sous séquence croissante de $11, 6, 2, 24, 25, 12, 21, 41, 34, 30$.

Question 1.7

Donnez un algorithme de programmation dynamique permettant de trouver la plus grande sous séquence croissante d'une suite de n entiers et donnez sa complexité.

Question 1.8

On peut cependant faire mieux que par programmation dynamique. Proposez un algorithme qui tourne en $O(n \log n)$.

1.4 Alignement de séquence

Un problème récurrent en bioinformatique est l'alignement de séquences. Si l'on a deux mots u et v formés à partir d'un alphabet $\mathcal{A}$ (par exemple les nucléotides : $\mathcal{A} = \{G, T, A, C\}$), on appelle alignement de u et v un couple de mots u', v' sur l'alphabet $\mathcal{A} \cup \{e\}$ (où $e \notin \mathcal{A}$ est le caractère d'espacement). u' et v' satisfont les propriétés suivantes :

- u' et v' ont la même longueur ($|u'| = |v'| = n$)
- si on supprime tous les e dans u' et v' on obtient respectivement u et v
- les lettres e ne sont pas à la même position dans u' et v'

Par exemple un alignement de $u = CGATTAG$ et $v = GATCGA$ est

$$\begin{aligned} u' &= CGATTeAG \\ v' &= eGATCGAe \end{aligned}$$

Afin de déterminer le meilleur alignement, il nous faut une métrique. Soit p une fonction de similarité entre un couple de lettres. La valeur de $p(a, b)$ est un réel d'autant plus grand que les lettres sont considérées comme similaires d'un point de vue biologique. On prendra donc une valeur négative lorsque les lettres sont différentes. On a donc $b \neq a, p(a, a) > 0 > p(a, b)$, la fonction de similarité est également symétrique : $p(a, b) = p(b, a)$. De plus, on se donne un réel $q < 0$ qui exprime la similarité d'une lettre avec un espacement, elle est indépendante de la lettre, on a donc $\forall a \in \mathcal{A}, p(a, e) = p(e, a) = q$. Le score d'un alignement est simplement la somme des similarités entre les lettres de u' et v' :

$$\sum_{i=1}^{n} p(u'_i, v'_i)$$

Question 1.9

On pose $p(a, b) = -1$ pour $a \neq b$, $p(a, a) = 5$ et $q = -2$. Quel est le score de l'alignement donné en exemple ? Donnez le meilleur alignement et le score correspondant pour les chaînes suivantes : $GATTACA$ et $ATACGTA$.

Question 1.10

Donnez un algorithme permettant de trouver le meilleur alignement (celui ayant le score le plus élevé). Expliquez comment vous reconstruisez la solution.

Question 1.11

Quelle est la complexité en temps et en espace de votre algorithme ?

Question 1.12

Vous disposez maintenant d'une famille de k chaînes de caractères $S_1, S_2, \ldots, S_k$. On souhaite obtenir un alignement des k séquences, c'est à dire k nouvelles chaînes de caractères $S'_1, S'_2, \ldots, S'_k$ toutes de même longueur n. On veut en fait minimiser la somme des distances entre toutes les paires de séquences, en reprenant le principe d'alignement de séquences précédent : on fait correspondre au mieux les séquences en ajoutant au besoin des espacements. La distance entre deux séquences étant la somme des distances caractère par caractère :

$$\sum_{1 \leq i < j \leq k} \sum_{1 \leq l \leq n} \delta(S'_i[l], S'_j[l])$$

avec $\delta(S_i, S_i) = 0$, $\delta \geq 0$ et δ est symétrique et respecte l'inégalité triangulaire.

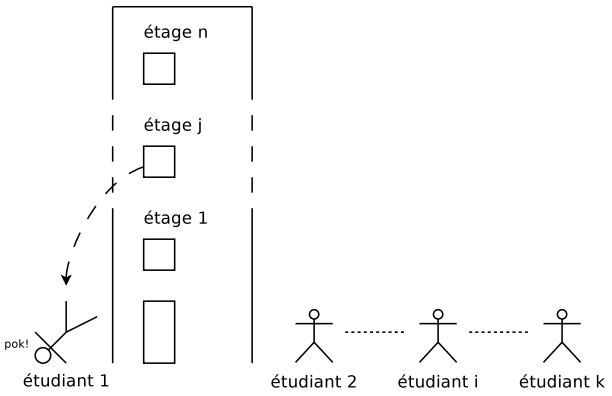
Pour $k = 2$, cela revient au problème précédent. Est-il possible d'étendre votre algorithme pour traiter le problème pour k quelconque ? Quelle serait alors la complexité temporelle de l'algorithme ? Est-ce polynomial en la taille des données (en le nombre de séquences à traiter, k ; et en la taille des chaînes, n) ?

2 Ce à quoi vous avez échappé

2.1 Le grand saut

Le problème est de déterminer à partir de quel étage d'un immeuble, sauter par une fenêtre est fatal. Vous êtes dans un immeuble à n étages (numérotés de 1 à n) et vous disposez de k étudiants. Les étudiants sont classés par notes de partiel croissantes. Il n'y a qu'une opération

possible pour tester si la hauteur d'un étage est fatale : faire sauter le premier étudiant de la liste par la fenêtre. S'il survit, vous pouvez le réutiliser ensuite (évidement, l'étudiant survivant reprend sa place initiale dans la liste triée), sinon vous ne pouvez plus.



Vous devez proposer un algorithme pour trouver la hauteur à partir de laquelle un saut est fatal (renvoyer $n+1$ si on survit encore en sautant du n-ème étage) en faisant le minimum de sauts.

Question 2.1 Si $k \geq \lceil \log_2(n) \rceil$, proposer un algorithme en $\mathcal{O}(\log_2(n))$ sauts.

Question 2.2 Si $k < \lceil \log_2(n) \rceil$, proposer un algorithme en $\mathcal{O}(k + \frac{n}{2^{k-1}})$ sauts.

Question 2.3 Si $k = 2$, proposer un algorithme en $\mathcal{O}(\sqrt{n})$ sauts.

2.2 Chercher la star

Dans un groupe de n personnes (numérotées de 1 à n pour les distinguer), une *star* est quelqu'un qui ne connait personne mais que tous les autres connaissent. Pour démasquer une star, s'il en existe une, vous avez juste le droit de poser des questions, à n'importe quel individu i du groupe, du type "est-ce que vous connaissez j ?" (noté "$i \to j$?"), on suppose que les individus répondent la vérité. On veut un algorithme qui trouve une star, s'il en existe, ou sinon qui garantit qu'il n'y a pas de star dans le groupe, en posant le moins de questions possibles.

Question 2.4 Combien peut-il y avoir de stars dans le groupe ?

Question 2.5 Ecrire le meilleur algorithme que vous pouvez et donner sa complexité en nombre de questions (on peut y arriver en $\mathcal{O}(n)$ questions).

Question 2.6 Donner une borne inférieure sur la complexité (en nombre de questions) de tout algorithme résolvant le problème. ((*Difficile*) prouver que la meilleure borne inférieure pour ce problème est $3n - \lfloor \log_2(n) \rfloor - 3$).