Complexité (suite)

Exercice 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 (évidemment, 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 (l'algorithme doit renvoyer $(n+1)$ si on survit encore en sautant du n-ème étage) en faisant le minimum de sauts.

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

Exercice 2. *Min et max*

Dans cet exercice, on s'intéresse au calcul (simultané) du maximum et du minimum de n entiers. On mesure la **complexité dans le pire des cas et en nombre de comparaisons** des algorithmes.

1. Donner un algorithme naïf et sa complexité.

Une idée pour améliorer l'algorithme est de regrouper *par paires* les éléments à comparer, de manière à diminuer ensuite le nombre de comparaisons à effectuer.

2. Décrire un algorithme fonctionnant selon ce principe et analyser sa complexité.

Nous allons étudier l'optimalité d'un tel algorithme en fournissant une borne inférieure sur le nombre de comparaisons à effectuer. Nous utiliserons la méthode de l'*adversaire*.
Soit A un algorithme qui trouve le maximum et le minimum. Pour une donnée fixée, au cours du déroulement de l'algorithme, on appelle *novice* (N) un élément qui n'a jamais subi de comparaisons, *gagnant* (G) un élément qui a été comparé au moins une fois et a toujours été supérieur aux éléments auxquels il a été comparé, *perdant* (P) un élément qui a été comparé au moins une fois et a toujours été inférieur aux éléments auxquels il a été comparé, et *moyens* (M) les autres. Le nombre de ces éléments est représenté par un quadruplet d'entiers (i, j, k, l) qui vérifie bien sûr $i + j + k + l = n$.

3. Donner la valeur de ce quadruplet au début et à la fin de l'algorithme. Exhiber une stratégie pour l'adversaire, de sorte à maximiser la durée de l'exécution de l'algorithme. En déduire une borne inférieure sur le nombre de tests à effectuer.

Exercice 3. *Star*

Dans un groupe de n personnes (numérotées de 1 à n pour les distinguer), une *star* est quelqu'un qui ne connaît 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 ? ». On suppose que les individus répondent toujours la vérité. On veut un algorithme qui trouve une star s'il en existe une, et qui garantit qu'il n'y en a pas sinon, en posant le moins de questions possibles.

1. Combien peut-il y avoir de stars dans le groupe ?
2. Écrire le meilleur algorithme que vous pouvez et donner sa complexité en nombre de questions (on peut y arriver en $\mathcal{O}(n)$ questions).
3. Donner une borne inférieure sur la complexité (en nombre de questions) de tout algorithme résolvant le problème.
   (*Difficile*) Prouver que la complexité exacte de ce problème est $3n - \lfloor \log_2(n) \rfloor - 3$.

Exercice 4. *Recherche dans un tableau*

Dans cet exercice, on s'intéresse à deux problèmes de sélections d'éléments dans un tableau.

1. Étant donné un tableau trié d'entiers deux à deux distincts $A[1,\dots,n]$, on veut décider s'il existe un entier i tel que $A[i] = i$. Donner un algorithme en temps $\mathcal{O}(\log n)$ pour ce problème.
2. Soit $A[1,\dots,n]$ un tableau d'entiers triés dont on ne connaît pas la longueur (n est inconnu). On peut demander la valeur de $A[i]$ pour n'importe quelle valeur entière de i : si $i \leq n$, on reçoit la valeur de $A[i]$, sinon on reçoit ∞. Donner un algorithme en temps $\mathcal{O}(\log n)$ qui prend en entrée un entier x et trouve l'indice du tableau où se trouve x (s'il en existe un).

Exercice 5. *Élément majoritaire*

Soit E une liste de n éléments rangés dans un tableau numéroté de 1 à n. On suppose que la seule opération qu'on sait effectuer sur les éléments est de vérifier si deux éléments sont égaux ou non. On dit qu'un élément $x \in E$ est *majoritaire* si l'ensemble $E_x = \{y \in E | y = x\}$ a strictement plus de $n/2$ éléments. Sauf avis contraire, on supposera que n est une puissance de 2. On s'intéressera à la complexité dans le pire des cas.

1. **Algorithme naïf**
   Écrire un algorithme calculant le cardinal c_x de E_x pour un x donné. En déduire un algorithme pour vérifier si E possède un élément majoritaire. Quelle est la complexité de cet algorithme ?
2. **Diviser pour régner**
   (a) Donner un autre algorithme récursif basé sur un découpage de E en deux listes de même taille. Quelle est sa complexité ?
   (b) Même question quand n n'est pas une puissance de 2.

Exercice 6. *Optimisation de la mémoire*

On souhaite enregistrer sur une mémoire de taille L un groupe de fichiers $P = (P_1,\dots,P_n)$. Chaque fichier P_i nécessite une place a_i. Supposons que $\sum a_i > L$: on ne peut pas enregistrer tous les fichiers. Il s'agit donc de choisir le sous ensemble Q des fichiers à enregistrer.
On pourrait souhaiter le sous-ensemble qui contient le plus grand nombre de fichiers. Un algorithme glouton pour ce problème pourrait par exemple ranger les fichiers par ordre croissant des a_i.
Supposons que les P_i soient ordonnés par taille ($a_1 \leq \dots \leq a_n$).

1. Écrire un algorithme pour la stratégie présentée ci-dessus. Cet algorithme doit renvoyer un tableau booléen S tel que $S[i] =$ true si P_i est dans Q et $S[i] =$ false sinon. Quelle est sa complexité en nombre de comparaisons et en nombre d'opérations arithmétiques ?
2. Montrer que cette stratégie donne toujours un sous-ensemble Q maximal tel que $\sum_{P_i \in Q} a_i \leq L$.
3. Soit Q le sous-ensemble obtenu. À quel point le quotient d'utilisation $(\sum_{P_i \in Q} a_i)/L$ peut-il être petit ?

Supposons maintenant que l'on souhaite enregistrer le sous-ensemble Q de P qui maximise ce quotient d'utilisation, c'est-à-dire celui qui remplit le plus de disque. Une approche *gloutonne* consisterait à considérer les fichiers dans l'ordre décroissant des a_i et, s'il reste assez d'espace pour P_i, on l'ajoute à Q.

4. On suppose toujours les P_i ordonnés par taille croissante. Écrire un algorithme pour cette nouvelle stratégie.
5. Montrer que cette nouvelle stratégie ne donne pas nécessairement un sous-ensemble qui maximise le quotient d'utilisation. À quel point ce quotient peut-il être petit ?

Exercice 7. *Impression équilibrée*

Le problème est l'impression équilibrée d'un paragraphe sur une imprimante. Le texte d'entrée est une séquence de n mots de longueurs $\ell_1, \ell_2, \dots, \ell_n$ (mesurées en caractères). On souhaite imprimer ce paragraphe de manière équilibrée sur un certain nombre de lignes qui contiennent un maximum de M caractères chacune. Le critère d'*équilibre* est le suivant. Si une ligne donnée contient les mots i à j (avec $i \leq j$) et qu'on laisse exactement une espace [1] entre deux mots, le nombre de caractères d'espacement supplémentaires à la fin de

1. En typographie *espace* est un mot féminin.

la ligne est $M - j + i - \sum_{k=i}^{j} \ell_k$, qui doit être positif ou nul pour que les mots tiennent sur la ligne. L'objectif est de minimiser la somme, sur toutes les lignes *hormis la dernière*, des cubes des nombres de caractères d'espacement présents à la fin de chaque ligne.

1. Est-ce que l'algorithme glouton consistant à remplir les lignes une à une en mettant à chaque fois le maximum de mots possibles sur la ligne en cours, fournit l'optimum ?
2. Donner un algorithme de programmation dynamique résolvant le problème. Analyser sa complexité en temps et en espace. *Indication : remarquer (et démontrer !) que si une solution est optimale, alors la solution privée de sa première ligne est optimale pour le texte privé de ses premiers mots.*
3. Supposons que pour la fonction de coût à minimiser, on ait simplement choisi la somme des nombres de caractères d'espacement présents à la fin de chaque ligne. Est-ce que l'on peut faire mieux en complexité que pour la question 2 ?
4. (*Plus informel*) Qu'est-ce qui à votre avis peut justifier le choix de prendre les cubes plutôt que simplement les nombres de caractères d'espacement en fin de ligne ?