École normale supérieure **2009-2010**
Département d'informatique

Algorithmique et Programmation
TD n° 2 : Tris et hachage

Un élève devra rédiger la correction de ce TD et envoyer sa correction à l'adresse `Damien.Vergnaud@ens.fr` *avant le* ***mercredi 21 octobre 2009****.*

Exercices théoriques

Exercice 1. Nombre moyen d'affectations dans la recherche de maximum

Nous voulons calculer le nombre moyen d'affectations $\mathrm{Moy}(n)$ de l'algorithme naïf de recherche du maximum sur un tableau de n éléments distincts. Soit $P_{n,k}$ le nombre de permutations ayant k maximum provisoire de gauche à droite (MPGD).

1. Montrer que
$$\mathrm{Moy}(n) = \sum_{k=1}^{n} k \frac{P_{n,k}}{n!}.$$
2. Montrer que le nombre de permutations ayant k MPGD vérifie :
   - $P_{1,1} = 1$, $P_{n,0} = 0$, pour $n \geq 1$ et $P_{n,k} = 0$ pour $k > n$.
   - $P_{n,k} = (n-1)P_{n-1,k} + P_{n-1,k-1}$, pour $n \geq 2$, $k \geq 1$.
3. Pour résoudre cette relation de récurrence, utiliser la série génératrice associée aux $(P_{n,k})_{k\in\mathbb{N}}$ et montrer que si $G_n(z) = \sum_{k\geq 0} P_{n,k} z^k$, alors pour $n \geq 1$, nous avons
$$G_n(z) = z(z+1)\dots(z+n-1).$$
4. En conclure que $\mathrm{Moy}(n) = G_n'(1)/G_n(1)$ et que $\mathrm{Moy}(n) = H_n$ le n-ième nombre harmonique
$$H_n = 1 + \frac{1}{2} + \dots + \frac{1}{n}$$
de l'ordre de $\Theta(\log n)$.

Exercice 2. Maximum & Minimum

1. Écrire un algorithme naïf qui calcule le minimum et le maximum sur un tableau de n éléments et donner sa complexité dans le pire cas.
2. 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. Décrire un algorithme fonctionnant selon ce principe et analyser sa complexité.
3. Montrer l'optimalité d'un tel algorithme en fournissant une borne inférieure sur le nombre de comparaisons à effectuer.

   Indication : On pourra utiliser la méthode de l'*adversaire*.

   Soit $\mathcal{A}$ 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,
– moyens (M) les autres éléments.

Le nombre de ces éléments est représenté par un quadruplet d'entiers (i, j, k, l) qui vérifient $i + j + k + l = n$. 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. Utiliser une approche de type *diviser pour régner* pour écrire un algorithme récursif qui trouve la somme maximale de valeurs situées dans des cases consécutives d'un tableau de n entiers de $\mathbb{Z}$.

Exercice 4. k-IÈME ÉLÉMENT D'UN TABLEAU

Dans cet exercice, nous allons utiliser une variante du tri rapide pour déterminer le k-ième élément d'un ensemble de n éléments munis d'une clé. Nous supposons que les n clés sont distinctes deux à deux. Nous considérons dans un premier temps, la fonction SÉLECTIONNER suivante :

```
Fonction SÉLECTIONNER(t, i, j, k) : élement
// on suppose que k ∈ [[1, n]].
si i = j alors retourner t[i] fin si
p := PIVOTER(t, i, j) ;
si k < p
    alors retourner SÉLECTIONNER(t, i, p, k)
    sinon retourner SÉLECTIONNER(t, p + 1, j, k − p)
fin si
```

où la fonction PIVOTER retourne un entier $p \in \{i + 1, \ldots, j - 1\}$ et réorganise en temps $O(n)$ le tableau $t[i..j]$ en deux sous-tableaux $t[i..p]$ et $t[p + 1..j]$ tels qu'un élément quelconque de $t[i..p]$ possède une clé inférieure ou égale à celle d'un élément quelconque de $t[p + 1..j]$.

1. Montrer que la procédure SÉLECTIONNER détermine le k-ième élément du tableau.
2. Monter que que sans hypothèse particulière sur l'algorithme de pivotage, la fonction SÉLECTIONNER peut réaliser $O(n^2)$ comparaisons.
3. Montrer que si l'algorithme de pivotage garantit que la taille du sous-tableau de l'appel récursif ne dépasse pas αn où $\alpha < 1$, la complexité en nombre de comparaisons de la fonction Sélectionner est $O(n)$.

On considère l'algorithme de choix du pivot suivant :

– Découper le tableau en $\lfloor n/5 \rfloor$ blocks $\{B_1, \ldots, B_{\lfloor n/5 \rfloor}\}$ de cinq éléments ; les élements restants (au plus 4) ne seront pas considérés dans la suite de l'algorithme ;
– Déterminer les éléments médians m_k des B_k, $k \in \{1, \ldots, \lfloor n/5 \rfloor\}$;
– Utiliser la fonction SÉLECTIONNER pour déterminer l'élement d'ordre $\lfloor (n+5)/10 \rfloor$ de la liste $m_1, \ldots, m_{\lfloor n/5 \rfloor}$; (si $\lfloor n/5 \rfloor$ est pair, l'élement selectionné est l'élement médian de la liste $m_1, \ldots, m_{\lfloor n/5 \rfloor}$).

4. Montrer que le pivot choisi est strictement supérieur à au moins $3\lfloor (n - 5)/10 \rfloor$ éléments de t et est inférieur ou égal à au moins $3\lfloor (n - 5)/10 \rfloor$ éléments de t. En déduire que pour $n \geq 75$, le sous tableau de l'appel récursif est de taille au plus égale à $3n/4$.

On considère alors la fonction SÉLECTIONNER suivante :

```
Fonction Sélectionner(t, i, j, k) : élément
si j − i ≤ 74 alors
    Trier(t, i, j) ;
    retourner l'élément d'ordre k de t ;
sinon
    pour q de 1 à ⌊(j − i)/5⌋ faire
        m(q) := Médian(t, 5(q − 1) + i, 5(q − 1) + i + 4)
    fin pour
    r := Sélectionner(m, 1, ⌊(j − i)/5⌋, ⌊(j − i + 5)/10⌋)
    p := Partition(t, i, j, r)
    si k < p
        alors retourner Sélectionner(t, i, p, k)
        sinon retourner Sélectionner(t, p + 1, j, k − p)
    fin si fin si
```

La procédure Trier est un algorithme de tri quelconque. La fonction Médian fournit l'élement de rang 3 d'une liste de 5 éléments. La procédure Partition réorganise le tableau t en prenant $t[r]$ comme pivot ; après son exécution, un élément quelconque de $T(i..p]$ a une clé inférieure ou égale à $t[r]$, un élément quelconque de $t[p+1..j]$ a une clé strictement supérieure à $t[r]$.

5. Démontrer la validité de la fonction Sélectionner.
6. Montrer que la complexité de la fonction Sélectionner est $O(n)$.

Exercices avec programmation

Exercice 5. Filtre de Bloom

Le filtre de Bloom, conçu par Burton H. Bloom en 1970, est une structure de données probabiliste qui optimise l'espace utilisé. Cette structure est utilisée pour tester si un élément fait partie d'un ensemble. Les faux positifs sont possibles, mais les faux négatifs ne le sont pas. Les éléments peuvent être ajoutés à la série, mais pas supprimés.

Un filtre de Bloom consiste en un tableau T de m bits initialisé à zéro. Nous disposons de k fonctions de hachage aléatoire et indépendantes $h_1, \ldots, h_k$ à image dans $[\![0, m-1]\!]$ et nous faisons l'hypothèse que les fonctions de hachage envoient chaque élément vers un élément uniformémemt aléatoire de $[\![0, m-1]\!]$. Pour placer un élément x dans cette structure, nous appliquons chaque fonction de hachage et nous plaçons un 1 dans $T[h_i(x)]$. Ensuite, pour tester si un ensemble y appartient à cet ensemble, il suffit de calculer tous les $h_i(y)$ et de vérifier qu'ils valent tous 1.

1. Déterminer la probabilité de faux positifs en fonction de m et k.
2. Programmer à l'aide d'un filtre de Bloom un détecteur de faute d'orthographe dans un fichier texte non formaté.
   (Le fichier `http://www.pallier.org/ressources/dicofr/liste.de.mots.francais.frgut.txt` contient 336531 mots français.)