Outils logiques et algorithmiques – TD 7 – Récursion

Exercise 1 (Tours de Hanoï) Rappel du problème : sur un emplacement de départ, on a une tour formée par un empilement de disques de différents diamètres, chaque disque étant posé sur un disque plus grand. On veut déplacer l'ensemble de la tour vers un emplacement d'arrivée. On déplace les disques un par un, chaque disque doit toujours être posé sur le sol ou sur un disque plus grand, et on a droit à un emplacement intermédiaire. Voici un algorithme récursif, pour déplacer une tour de n disques d'un emplacement A à un emplacement C, en utilisant l'emplacement intermédiaire B.

```
static void deplacePile(int n, int A, int C, int B) {
    if (n != 0) {
        deplacePile(n-1, A, B, C);
        deplaceDisque(A, C);
        deplacePile(n-1, B, C, A);
    }
}
```

On note $C(n)$ le nombre d'opérations `deplaceDisque` effectuées par l'algorithme `deplacePile`.

1. Donner des équations récursives pour $C(n)$ et les résoudre.
2. Combien de temps faut-il approximativement pour résoudre `deplacePile(64, x, y, z)` si l'on réalise un milliard d'opérations `deplaceDisque` par seconde ? *Rappel :* 2^{10} *est légèrement supérieur à* 1 000.

□

Exercise 2 (Multiplication de grands nombres) On cherche une méthode efficace pour multiplier deux nombres a et b ayant chacun n chiffres (si l'un des deux a moins de n chiffres, on le complète avec des zéros à gauche pour uniformiser). Notre mesure de complexité sera le nombre $C(n)$ de multiplications de chiffres réalisées lors de la multiplication de a par b.

1. Avec la méthode traditionnelle, combien doit-on effectuer de multiplications de chiffres pour calculer $a \times b$?

Remarque : si on prend deux décompositions $a = a_1 \times 10^k + a_2$ et $b = b_1 \times 10^k + b_2$ (autrement dit, a_1 et b_1 sont les nombres formés avec les $n - k$ premiers chiffres de a et b, et a_2 et b_2 sont formés avec les k derniers chiffres), alors on peut proposer les formules suivantes pour calculer le produit $a \times b$:

— Décomposition (A) $a \times b = a_1 \times b_1 \times 10^{2k} + (a_1 \times b_2 + a_2 \times b_1) \times 10^k + a_2 \times b_2$

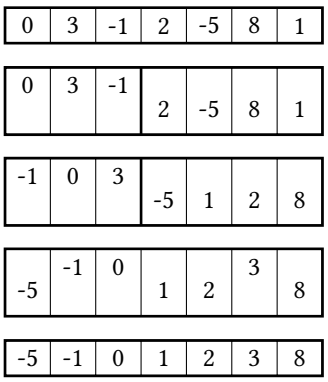
— Décomposition (B) $a \times b = a_1 \times b_1 \times 10^{2k} + (a_1 \times b_1 + a_2 \times b_2 - (a_1 - a_2) \times (b_1 - b_2)) \times 10^k + a_2 \times b_2$

Si $0 < k < n$, chacune de ces décompositions utilise des produits d'entiers avec strictement moins que n chiffres. On continue à décomposer ainsi (récursivement) jusqu'à obtenir des multiplications de nombres à un chiffre.

2. Si on utilise systématiquement $k = 1$, donner des équations récursives pour $C(n)$ en suivant la décomposition (B), et déduire qu'on obtient une complexité exponentielle.
3. Supposons que n est une puissance de 2, et prenons systématiquement $k = \frac{n}{2}$. Donner des équations récursives pour $C(n)$ en suivant la décomposition (A), et montrer que $C(n) = n^2$.
4. Supposons que n est une puissance de 2, et prenons systématiquement $k = \frac{n}{2}$. Donner des équations récursives pour $C(n)$ en suivant la décomposition (B), et montrer que $C(n) = 3^{\log_2(n)}$. En déduire que $C(n) = n^c$, pour une certaine constante c à déterminer (*rappel d'identités utiles sur les exponentielles :* $(x^y)^z = x^{(yz)}$ *et* $x = 2^{log_2(x)}$). A-t-on gagné quelque chose ?

□

Exercise 3 (Tri fusion) Le *tri fusion* procède, comme le tri rapide, en découpant un tableau en deux parties à trier indépendamment, et en poursuivant récursivement le découpage pour chaque partie. La différence tient dans le découpage : on définit les deux parties à trier indépendamment comme la première moitié et la deuxième moitié du tableau (sans toucher préalablement aux éléments de ces parties), et on ajoute à la fin une opération de *fusion*, pour entrelacer les deux moitiés triées en un unique segment trié.



Voici une réalisation de cette idée en java. La fusion est réalisée par la fonction `merge`, qui prend en entrée :

— un tableau `tab`,

— trois indices `lo`, `mid`, et `hi` tels que $0 \leq$ `lo` $\leq$ `mid` $\leq$ `hi` $\leq$ `tab.length`,

et suppose en outre que les deux segments `tab[lo,mid[` et `tab[mid,hi[` sont triés par ordre croissant, et qui réécrit le segment `tab[lo,hi[` de sorte à ce qu'il soit entièrement trié par ordre croissant. Cette fonction `merge` utilise un tableau temporaire `tmp`, dans lequel on place en alternance des éléments de l'une ou l'autre moitié triée du tableau `tab` donné en entrée, avant de les copier dans l'ordre dans `tab`.

```
static void merge(int[] tab, int lo, int mid, int hi) {
    /* fusion de tab[lo,mid[ et tab[mid,hi[ dans tab[lo,hi[,
       en passant par tmp[0,hi-lo[ */
    int[] tmp = new int[hi-lo];
    int i = lo, j = mid, k = 0;
    while (i < mid || j < hi) {
        if (i < mid && (j == hi || tab[i] <= tab[j]))
            tmp[k++] = tab[i++];
        else
            tmp[k++] = tab[j++];
    }
    for (k=0; k < hi-lo; k++) tab[lo+k] = tmp[k];
}

static void mergeSort(int[] tab, int lo, int hi) {
    /* tri de tab[lo,hi[ */
    if (lo >= hi-1) return;
    int mid = lo + (hi-lo)/2;
    mergeSort(tab, lo, mid);
    mergeSort(tab, mid, hi);
    merge(tab, lo, mid, hi);
}

static void mergeSort(int[] tab) {
    mergeSort(tab, 0, tab.length);
}
```

1. Donner des invariants pour la boucle `while` de la fonction `merge`.
2. Quand s'arrête la boucle de la fonction `merge` ? Combien cette boucle fait-elle de tours ? Combien cette fonction effectue-t-elle d'accès à un élément de tableau au minimum ? au maximum ? Peut-on préciser le résultat si `mid` est précisément au milieu de `lo` et `hi` ?
3. On note $C_{\text{merge}}(n)$ le nombre maximum d'accès réalisés par `merge(tab, lo, mid, hi)` lorsque `hi-lo` $= n$. Donner des équations récursives définissant $C(n)$, le nombre maximal d'accès réalisés par `mergeSort(tab, lo, hi)` lorsque `hi-lo` $= n$.
4. Montrer que la fonction `mergeSort`, appliquée à un tableau de longueur 2^k, réalise au maximum de l'ordre de $k \times 2^k$ accès. Quel est l'ordre de grandeur de la complexité de `mergeSort` ?

□

Exercise 4 (Tri-tri) Voici un nouvel algorithme de tri récursif.

```
public static void stoogeSort(int[] tab, int lo, int hi) {
    if (tab[lo] > tab[hi-1])
        swap(tab, lo, hi-1);
    if (hi-lo > 2) {
        int a = lo + (hi-lo)/3;
        int b = hi - (hi-lo)/3;
        stoogeSort(tab, lo, b);
        stoogeSort(tab, a, hi);
        stoogeSort(tab, lo, b);
    }
}
```

On note $C(n)$ le nombre de comparaisons d'éléments du tableau nécessaires pour trier un segment de longueur n.

1. Donner une expression récursive de $C(n)$ et la résoudre (*master theorem* autorisé).
2. Comparer avec les autres tris que vous connaissez.

□

Exercise 5 (Élément majoritaire) Un élément majoritaire dans un tableau de taille n est un élément qui apparaît au moins $\lceil \frac{n}{2} \rceil$ fois. Voici un algorithme déterminant s'il existe un élément majoritaire dans le segment [`lo`, `hi` [d'un tableau.

```
static Integer majority(int[] tab, int lo, int hi) {
    if (hi-lo == 0) return null;
    if (hi-lo == 1) return tab[lo];
    int mid = lo + (hi-lo)/2;
    Integer a = majority(tab, lo, mid);
    Integer b = majority(tab, mid, hi);
    if (a == b)
        return a;
    else
        return checkMaj(tab, lo, hi, a, b);
}
```

Il fait appel à une routine auxiliaire qui, étant donnés deux éléments potentiellement majoritaires, vérifie si un des deux l'est effectivement.

```
static Integer checkMaj(int[] tab, int lo, int hi, int a, int b) {
    int na=0, nb=0;
    for (int i=lo; i<hi; i++) {
        if (tab[i] == a) na++;
        if (tab[i] == b) nb++;
    }
    if (na >= (hi-lo+1)/2)
        return a;
    else if (nb >= (hi-lo+1)/2)
        return b
    else
        return null;
}
```

1. Combien `checkMaj` réalise-t-elle d'accès au tableau `tab` ?
2. En déduire des équations récursives pour le nombre d'accès à des éléments de `tab` réalisés par `majority` en fonction de la longueur n du segment et résoudre.
3. Comparer avec ce qu'aurait donné l'approche naïve consistant à compter le nombre d'occurrences de chaque élément puis chercher l'élément avec le nombre d'occurrences maximal (et vérifier si ce nombre est bien au moins la moitié).

□

Exercise 6 (Quickselect) Les algorithmes suivants cherchent l'élément qui serait à l'indice `k` dans le tableau `tab`, si le tableau `tab` était trié. Mais cela sans trier le tableau, ou du moins sans le trier complètement.

```
static int select(int k, int[] tab) {
    for (int i=0; i <= k; i++) {
        int jMin = i;
        for (int j=i+1; j < tab.length; j++)
          if (tab[j] < tab[jMin]) jMin = j;
        swap(tab, i, jMin);
    }
    return t[k];
}
```

```
static int quickSelect(int k, int[] tab, int lo, int hi) {
    assert (lo <= k && k < r);
    int a=lo, b=lo+1, c=hi;
    int pivot = tab[lo];
    while (b < c) {
        if      (tab[b] < pivot) swap(tab, b++, a++);
        else if (tab[b] > pivot) swap(tab, b, --c);
        else /* tab[b] == pivot */ b++;
    }
    if      (k <  a)   return quickSelect(k, tab, lo, a);
    else if (k >= b)   return quickSelect(k, tab, b, hi);
    else  /* a<=k<b */ return t[k];
}
```

Dans le programme `quickSelect`, remarquez que la boucle **`while`** est exactement celle qui apparaît dans `quickSort` pour répartir les éléments du tableau de part et d'autre d'un pivot.

1. Combien de comparaisons effectue `select`, en fonction de `k` et de la longueur n de `tab` ?
2. Combien de comparaisons effectue `quickSelect` dans le meilleur cas ? Dans le pire cas ?
3. On veut estimer le nombre de comparaisons effectué en moyenne par `quickSelect` en moyenne, en supposant que `tab` et `k` sont aléatoires et que tous les éléments de `tab` sont distincts.
   (a) On fait d'abord une hypothèse simplificatrice : les n scénarios « on a trouvé l'élément » et « on fait un appel récursif dans un segment de longueur k » (pour les $n - 1$ valeurs crédibles de k) sont tous équiprobables. Quelle serait la complexité moyenne dans ce cas ?
   (b) Notre hypothèse est-elle de nature à surestimer ou à sous-estimer la véritable moyenne ?
   (c) Défi : estimer la vraie moyenne.

□