Algorithmes de tri

2ème partie du cours

Cette partie du cours concerne 2 algorithmes de tri performants. Tous deux sont récursifs et basés sur la stratégie « diviser pour régner ». Ils sont plus efficaces que le tri par insertion dès lors que la taille n des données dépasse environ 50. Leur complexité moyenne est en $O(n \ln n)$.

I. La stratégie « diviser pour régner »

Le principe de cette stratégie est le suivant :

- Diviser le problème initial en sous-problèmes similaires de tailles inférieures ;
- Traiter récursivement les sous-problèmes ;
- Recomposer la solution du problème initial à partir des solutions des sous-problèmes.

Pour trier une liste à l'aide de cette stratégie, on va se ramener au tri de deux listes de tailles inférieures.

- Dans le cas du *tri fusion*, on coupe simplement la liste en deux, on trie récursivement les deux parties, puis on fusionne ces deux parties en une liste triée. Le défaut de cette méthode est qu'elle oblige à créer une nouvelle liste de même taille que la liste initiale (du moins dans sa version élémentaire).
- Dans le cas du *tri rapide*, on commence par partitionner la liste autour d'un élément appelé *pivot* : les éléments à gauche du pivot sont inférieurs, ceux à droite sont supérieurs. On n'utilise pas de liste auxiliaire. On trie alors récursivement les deux sous-listes, et il n'y a rien à faire ensuite.

II. Tri fusion (merge sort)

La méthode consiste à séparer les données en deux parties presque égales, puis à traiter chaque partie et enfin à les rassembler.

L'efficacité de l'algorithme vient du fait que deux listes triées peuvent être fusionnées en temps linéaire.

Pseudo-code : programme principal

Algorithme 1 : Tri fusion : programme principal

Données : t liste des éléments à trier
Résultat : la liste triée

fonction *TriFusion(t)*
  $n \leftarrow$ longueur(t);
  si $n <= 1$ **alors**
    renvoyer t
  sinon
    $p \leftarrow \lfloor \frac{n}{2} \rfloor$;
    renvoyer Fusionner$(TriFusion(t[0:p]),\ TriFusion(t[p:]))$
  finsi
findef

On pourrait envisager une implémentation récursive de la procédure `Fusionner`, mais cela n'est pas du tout efficace à cause des nombreuses recopies de listes qui sont faites (chaque recopie étant en temps linéaire, on peut arriver à une complexité quadratique dans le pire des cas !), et surtout, le nombre d'appels récursifs est égal dans le pire des cas à la taille du plus grand des deux tableaux, ce qui est inacceptable :

Algorithme 2 : Fusionner : version récursive moche

Données : deux listes triées t_1 et t_2
Résultat : liste triée formée de la fusion des deux

```
fonction Fusionner(t1 , t2 )
    si t1 est vide alors
        renvoyer t2
    sinon
        si t2 est vide alors
            renvoyer t1
        sinon
            si t1[0] < t2[0] alors
                renvoyer [t1[0]] + Fusionner(t1[1 :], t2)
            sinon
                renvoyer [t2[0]] + Fusionner(t1, t2[1 :])
            finsi
        finsi
    finsi
findef
```

Une version itérative, qui utilise une liste auxiliaire, est de loin préférable :

Algorithme 3 : Fusionner : version itérative

Données : deux listes triées a et b
Résultat : liste triée formée de la fusion des deux

```
fonction Fusionner(a , b )
    tmp ← liste vide ;
    i ← 0;
    j ← 0 ;
    tant que (i < len(a) ) et (j < len(b) ) faire
        si a_i ⩽ b_j alors
            ajouter a_i à la fin de tmp;
            i ← i + 1
        sinon
            ajouter b_j à la fin de tmp ;
            j ← j + 1
        finsi
    fintq
    si i = len(a) alors
        ajouter à la fin de tmp les éléments de la liste b à partir de b_j
    sinon
        ajouter à la fin de tmp les éléments de la liste a à partir de a_i
    finsi
    renvoyer tmp;
findef
```

Complexité maximale :

Supposons d'abord que la taille du tableau initial est $n = 2^p$. On a donc alors (complexité en nombre de tests) :

$$C(Nn = 2C\left(\frac{n}{2}\right) + n \quad (\text{ et } C(1) = 1)$$

(deux appels récursifs avec des tableaux de taille divisée par 2, plus le coût de la fusion). Si on réécrit cette relation en posant $u_p = \dfrac{C(2^p)}{2^p}$ on a alors :

$$u_p = \begin{cases} 1 & \text{si } p = 0 \\ u_{p-1} + 1 & \text{sinon} \end{cases}$$

On reconnaît une suite arithmétique, dont on peut calculer le terme général $u_p = 1 + p$, d'où il vient $C(2^p) = 2^p u_p = p.2^p + 2^p$. Puisque $p = \log_2(n)$ on en tire :

$$C(n) = O(n \ln(n)) .$$

Si maintenant le tableau n'est pas de taille 2^p, il suffit d'encadrer n entre deux puissances de 2 successives :

$$2^p \leqslant n < 2^{p+1}$$

La fonction de complexité étant croissante, on a donc

$$C(2^p) \leqslant C(n) \leqslant C(2^{p+1})$$

ce qui permet d'en déduire que l'on a encore $C(n) = O(n\ln(n))$.

III. Tri rapide (quicksort)

Le tri rapide est une autre méthode de tri récursive s'appuyant sur le principe « diviser pour régner ». Il a été inventé par Hoare en 1961.

La méthode consiste à placer un élément du tableau (appelé *pivot*) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite.

Cette opération s'appelle le *partitionnement*. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

La difficulté est ici que l'on exige d'avoir un tri *en place* ; sinon on pourrait facilement créer deux nouveaux tableaux avec les éléments plus petits/plus grands que le pivot, et on en déduirait un tri similaire au tri fusion.

Pseudo-code : programme principal

Algorithme 4 : Tri rapide : programme principal

Données : t, debut, fin : on trie le sous-tableau de t[debut] à t[fin] inclus
Résultat : la même liste triée en place

```
fonction TriRapide(t,debut,fin)
    si debut < fin alors
        pivot ← Partition(t,debut,fin);
        TriRapide(t, debut,pivot−1);
        TriRapide(t,pivot+1,fin)
    finsi
findef
/* L'appel initial sera: TriRapide(t,0, len(t) − 1)          */
```

Dans cet algorithme, la procédure `Partition` s'occupe de partager le sous-tableau autour d'un pivot, et renvoie l'indice de ce pivot.

Un problème de *complexité spatiale* peut se poser : en effet, lors de la récursion, les appels non encore satisfaits sont empilés ; dans un cas particulièrement défavorable, il se pourrait que les partitions successives donnent deux sous-tableaux très inégaux, l'un vide et l'autre de taille maximum ; à partir du tableau initial, on obtiendrait donc une pile de taille n, soit une complexité spatiale en $O(n)$, ce qui est inacceptable.

Une solution à ce problème est de commencer systématiquement par le tableau de plus petite taille. Ainsi, la taille maximale $P(n)$ de la pile lors des appels récursifs vérifiera une relation de récurrence de la forme

$$P(n) \leqslant 1 + P\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

d'où $P(n) = O(\ln n)$ ce qui est beaucoup mieux. La modification à apporter est simple :

Algorithme 5 : Tri rapide : programme principal amélioré

Données : t, debut, fin : on trie le sous-tableau de t[debut] à t[fin] inclus
Résultat : la même liste triée en place

```
fonction TriRapide(t,debut,fin)
    si debut < fin alors
        pivot← Partition(t,debut,fin);
        si pivot-debut ⩽ fin - pivot alors
            TriRapide(t, debut,pivot−1);
            TriRapide(t,pivot+1,fin);
        sinon
            TriRapide(t,pivot+1,fin);
            TriRapide(t, debut,pivot−1);
        finsi
    finsi
findef
/* L'appel initial sera: TriRapide(t,0, len(t) − 1)                    */
```

Enfin, on peut remarquer que, puisque le second appel récursif est la dernière instruction (récursivité terminale), il peut être avantageusement remplacé par une structure itérative (boucle `tant...que`). Cela donne le programme suivant :

Algorithme 6 : Tri rapide : programme principal encore amélioré

Données : t, debut, fin : on trie le sous-tableau de t[debut] à t[fin] inclus
Résultat : la même liste triée en place

```
fonction TriRapide(t,debut,fin)
    tant que debut < fin faire
        pivot← Partition(t,debut,fin);
        si pivot-debut ⩽ fin - pivot alors
            TriRapide(t, debut,pivot−1);
            debut ← pivot +1;
        sinon
            TriRapide(t,pivot+1,fin);
            fin ← pivot−1;
        finsi
    fintq
findef
/* L'appel initial sera: TriRapide(t,0, len(t) − 1)                    */
```

Pseudo-code : programme partitionnement

Le partitionnement est la partie la plus délicate.

Une approche assez simple est la suivante :

- on choisit arbitrairement comme pivot le dernier élément du sous-tableau ;
- on place tous les éléments inférieurs au pivot en début du sous-tableau ;
- on place le pivot à la fin des éléments déplacés, et on retourne sa position.

Cela donne l'algorithme suivant :

Algorithme 7 : Partitionnement : première version

Données : sous-tableau de t[debut] à t[fin] inclus
Résultat : indice du pivot

```
fonction Partition(t,debut,fin)
    valeur ← t[fin];
    p ← debut;
    pour i de debut à fin−1 faire
        si t[i] ⩽ valeur alors
            échanger t[i] et t[p];
            p ← p + 1
        finsi
    finpour
    échanger t[p] et t[fin];
    renvoyer p
findef
```

Complexité :

On s'intéresse ici au nombre de comparaisons effectuées, qui majore de toutes façons le nombre d'échange.
La complexité, en nombre de comparaisons, de la fonction de partition `Partition(t,debut,fin)` est exactement $fin - debut$.

Étudions la complexité en moyenne du tri rapide, en supposant qu'à chaque étape la position du pivot est équiprobable pour toutes les valeurs. Notons $C(k)$ le nombre de comparaisons pour un tableau de taille k.
Lors de l'appel initial de `Trirapide`, la fonction `Partition` effectue $n-1$ comparaisons ; puis, si le pivot est situé à la position k, il y aura $C(k)$ comparaisons pour la partie gauche du tableau et $C(n-k-1)$ pour la partie droite. Cela donne, en moyenne :

$$C(n) = \frac{1}{n}\sum_{k=0}^{n-1}(C(k)+C(n-1-k)) + n - 1 = \frac{2}{n}\sum_{k=0}^{n-1}C(k) + n - 1$$

d'où

$$nC(n) = 2\sum_{k=0}^{n-1}C(k) + n(n-1)\,.$$

En considérant cette égalité aussi au rang $n-1$ et en la soustrayant à la précédente on obtient :

$$nC(n) - (n-1)C(n-1) = 2C(n-1) + n(n-1) - (n-2)(n-1)$$

d'où

$$nC(n) - (n+1)C(n-1) = 2(n-1)$$

puis

$$\frac{C(n)}{n+1} - \frac{C(n-1)}{n} = 2\frac{n-1}{n(n+1)} = \frac{4}{n+1} - \frac{2}{n}$$

d'où en additionnant, après télescopage, et en notant H_n la n-ième somme partielle de la série harmonique :

$$\frac{C(n)}{n+1} = 4(H_{n+1}-1) - 2H_n = 4(\ln(n+1)+\gamma-1) - 2(\ln(n)+\gamma) + o(1) \underset{n\to+\infty}{\sim} 2n\ln(n)\,.$$

La complexité en moyenne du tri rapide pour les comparaisons est donc en $O(n\ln(n))$.

Cependant : dans le pire des cas, c'est-à-dire quand le pivot choisi est le plus petit élément, ou le plus grand, la partition se fera en laissant un des cotés vide, et l'autre avec $n-1$ éléments. La complexité vérifie alors la relation de récurrence

$$C(n) = C(n-1) + n - 1$$

qui se résout (assez) facilement, et donne $C(n) \sim \frac{n^2}{2}$! Oups !

Pour cette raison, il existe pléthore d'autres méthodes de partitionnement ; la plus facile à écrire consiste à choisir un pivot aléatoire. On obtiendra donc un algorithme du genre :

Algorithme 8 : Partitionnement : deuxième version

Données : sous-tableau de t[debut] à t[fin] inclus
Résultat : indice du pivot

fonction *Partition(t ,debut,fin)*
  $x \leftarrow$ entier aléatoire entre debut et fin;
  échanger t_x et t_{fin};
  valeur $\leftarrow t_{fin}$;
  $p \leftarrow$ debut;
  pour i *de debut à fin*-1 **faire**
    si $t_i \leqslant$ *valeur* **alors**
      échanger t_i et t_p;
      $p \leftarrow p+1$
    finsi
  finpour
  échanger t_p et t_{fin};
  renvoyer p
findef

Une autre solution possible pour le choix du pivot, elle aussi très simple à mettre en œuvre, est de choisir l'indice de l'élément médian parmi les trois éléments d'indices `debut`, `fin` et `(debut + fin)/2`. Je vous encourage vivement à le faire !

Un ultime raffinement :

Il semble clair que le programme exposé ci-dessus est trop lourd pour des petits tableaux ; pour cette raison, il convient d'arrêter la récursion dès que la taille des sous-tableaux devient inférieur à un certain seuil. Lorsque ce seuil est atteint, on fera appel au tri par insertion. Certaines études théoriques et des essais pratiques ont situés ce seuil entre 8 et 20.
Le programme définitif sera donc le suivant :

Algorithme 9 : Tri rapide : programme principal, version finale.

Données : t, debut, fin : sous-tableau t[debut :fin]
Résultat : la même liste triée en place

fonction *TriRapide(t ,debut,fin)*
  seuil = 10;
  tant que *fin - debut* $>$ *seuil* **faire**
    pivot$\leftarrow$ Partition(t,debut,fin);
    si *pivot-debut* $\leqslant$ *fin - pivot* **alors**
      TriRapide(t, debut,pivot-1);
      debut $\leftarrow$ pivot $+1$;
    sinon
      TriRapide(t,pivot$+1$,fin);
      fin $\leftarrow$ pivot-1;
    finsi
  fintq
  TriInsertion(t,debut,fin)
findef
`/* L'appel initial sera: TriRapide(`t`,0,` $len(t)-1$ `)  */`