TD n°3 : LES TRIS, 1ère partie

I. Une optimisation du tri par insertion

Je rappelle le principe du tri par insertion d'une liste `L` vu en cours :

> On parcourt le tableau à trier du début à la fin. Au moment où on considère le i-ème élément, les éléments qui le précèdent sont déjà triés, et il s'agit d'insérer cet élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion.

Une optimisation de cette méthode consiste à utiliser la recherche dichotomique pour trouver l'emplacement où il faut insérer l'élément `L[i]` parmi les éléments de la liste *déjà triée* `L[0:i]`.

On écrira donc une fonction `dicho(liste, x, d, f)` qui renvoie le plus grand indice i de l'intervalle $[\![d\,;f]\!]$ tel que $L[i] > x$, en supposant que la liste `L[d:f+1]` est déjà triée par ordre croissant.

On utilisera ensuite cette fonction pour écrire une procédure `TriInsertionOptimise(L)`, et l'on comparera son temps d'exécution avec l'algorithme classique vu en cours.

Remarque : vous noterez que le gain en temps est assez spectaculaire ; néanmoins, cet algorithme reste de complexité $\mathcal{O}(n^2)$ car il faut quand même à chaque étape faire un décalage d'une partie de la liste, et cette opération reste en $\mathcal{O}(n)$.

II. Tri par dénombrement *(counting sort)*

Cet algorithme, qui est en temps linéaire, est applicable lorsque l'on sait par avance que les éléments du tableau à trier ne peuvent prendre qu'un nombre fini de valeurs dans un intervalle donné.

On supposera donc que l'on dispose d'une liste `L` de longueur n, **formée de nombres entiers**.

Un premier parcours de la liste, qui sera en temps $\mathcal{O}(n)$, permet d'obtenir le minimum m et le maximum M de ces nombres. Les valeurs des éléments de la liste sont donc au nombre, au maximum, de $k = M - m + 1$.

En faisant un second parcours du tableau, on peut ainsi construire une liste `Occ` de longueur k, où pour tout $i \in [\![0\,;k-1]\!]$, `Occ[i]` désigne le nombre d'occurrences dans `L` de l'élément $m+i$.

Il suffit alors de parcourir le tableau `Occ`, ce qui se fera en temps $\mathcal{O}(k)$, pour créer une liste formée des éléments de `L` et triée.

La complexité totale de l'algorithme est donc en $\mathcal{O}(2n+k)$.

Remarque : ce qui a été vu dans le 1er paragraphe du cours, à savoir que la complexité minimale d'un algorithme de tri est en $\mathcal{O}(n\ln(n))$, ne s'applique pas ici, car on ne procède pas par comparaisons entre éléments de la liste.

III. Tri par répartition *(pigeonhole sort)*

Il s'agit d'une généralisation du tri par dénombrement, qui permet de trier les éléments d'une liste selon une certaine **clé**. Ce tri calque le mode opératoire d'un facteur dans un centre de tri postal : celui-ci dispose de casiers correspondant à des adresses, rangées dans l'ordre de sa tournée. Le facteur récupère alors sa liste de lettres à distribuer, puis range chacune des enveloppes dans le casier correspondant. À la fin, il ne lui reste plus qu'à rassembler les enveloppes en les prenant dans l'ordre des casiers.

On dispose donc d'une liste `L` formée de n éléments d'un certain ensemble E, et d'une fonction `clé` qui à tout $x \in E$ associe un entier naturel $c(x)$ (le numéro du casier dans le cas du facteur !). On veut trier la liste `L` selon cette clé de façon que le tri soit stable (c'est-à-dire que des éléments de même clé se retrouveront au final dans le même ordre que dans la liste initiale).

Pour cela, on parcourt une 1ère fois la liste `L` afin de connaître la clé maximum ; cela permet de créer un tableau `Occ` de longueur k dont les éléments seront des listes (les casiers !). Puis on parcourt une seconde fois la liste `L` ; on calcule alors la clé c de chaque élément `L[i]`, puis on met cet élément `L[i]` dans le casier numéro c, autrement dit on l'ajoute à la liste `Occ[c]`. À la fin, il ne reste plus qu'à concaténer les listes `Occ[c]` pour obtenir la liste triée.

Comme exemple, on pourra trier un tableau de nombre entiers selon leur dernier chiffre en base 10. Ainsi le tableau

$$[752, 268, 496, 384, 307, 945, 25, 563, 300, 483, 820, 369, 235, 660, 208, 575, 585, 686, 383, 656]$$

sera trié en :

$$[300, 820, 660, 752, 563, 483, 383, 384, 945, 25, 235, 575, 585, 496, 686, 656, 307, 268, 208, 369]$$

Là encore, la complexité de cet algorithme est en $O(2n+k)$. Notez que, si l'on connaît à l'avance la valeur maximum que peut prendre la clé (10 dans l'exemple précédent), la 1ère étape de l'algorithme n'est plus nécessaire, et la complexité est en $O(n+k)$.

IV. Tri par base *(radix sort)*

Le tri par base permet de trier des listes de nombres entiers (mais on peut facilement l'adapter pour trier des chaines de caractères). Si la liste est formée de nombres de d chiffres au maximum en base 10, il consiste à faire d tris par répartition, en commençant par trier selon le chiffre le plus à droite de chaque nombre, puis selon l'avant-dernier etc... Il est bien sûr impératif que chaque tri soit stable.

Si k désigne la base choisie, on a vu que le tri par répartition d'une liste de n éléments est en $O(n+k)$; la complexité du tri par base sera donc de $O(d(n+k))$. Donc pour que cet algorithme soit plus intéressant que les meilleurs algorithmes que l'on verra dans la seconde partie, qui sont en $O(n\ln(n))$, il est nécessaire que d soit inférieur à $\ln(n)$. Cet algorithme est donc surtout adapté au tri de données de taille maximum fixe (code postal, n° de sécurité sociale, etc...).

V. Tri par paquets (ou baquets) *(bucket sort)*

Le tri par paquets permet de trier une liste de n nombres réels uniformément distribués dans l'intervalle $[0\,;1[$ (pour simplifier).

Pour cela, on divise l'intervalle $[0\,;1[$ en n sous-intervalles de même taille (les *paquets*) dans lesquels on répartit les éléments du tableau initial. Comme les nombres sont uniformément distribués, chaque paquet ne devrait pas contenir beaucoup de nombres. Ensuite, on trie les éléments de chaque paquet par un autre algorithme de tri (par exemple par insertion), et il ne reste plus qu'à parcourir tous les paquets dans l'ordre pour obtenir la liste triée.

On peut montrer que le temps d'exécution moyen de ce tri est en $O(n)$.

VI. Tri de Shell

Imaginé par Donald Shell en 1959, cet algorithme est une amélioration du tri par insertion. L'idée ce ce tri est la suivante.

Le principal inconvénient du tri par insertion est que les éléments sont déplacés d'une position à la fois, donc si le k-ième éléemnt de la liste doit se retrouver en 1ère position, il faudra déplacer k éléments ! Mais un gros avantage du tri par insertion est qu'il est très efficace sur des tableaux « presque » triés.

La méthode de Shell consiste donc à trier par insertion les sous-tableaux `L[0], L[h], L[2h], ..` puis `L[1], L[h+1], L[2h+1], ...` jusqu'à `L[h-1], L[2h-1], L[3h-1], ...`, où h est une constante >0, puis de recommencer avec une constante $h'<h$, etc.., jusqu'à un dernier passage avec $h=1$. L'idée est que le premier tri par distance de h (on parlera de « h-tri ») accélère le tri par distances de h', puisque un certain ordre a déjà été réalisé dans le tableau.

On procèdera donc ainsi :

1. Dans un premier temps, en s'inspirant de l'algorithme du tri par insertion, écrire une fonction `h_tri_insertion( L, h,i)` qui trie par insertion la sous-liste formée des éléments `L[i], L[i+h], L[i+2h], ...`.
2. La procédure de tri de Shell s'écrira ensuite simplement :

```
def TriShell(L):
    for h in liste_des_incréments
        for i in range(h):
            h_tri_insertion(L, h, i)
```

La difficulté consiste à trouver une suite (h_n) satisfaisante (`liste_des_increments` dans le programme ci-dessus), et ce problème n'est pas encore complètement résolu.

Vous pourrez essayer les listes suivantes, et comparer les temps d'exécution :

- $h_1=1$ puis $h_{k+1}=3*h_k+1$ avec $h_k\leqslant\frac{n}{3}$, où n est la longueur de la liste à trier.

  Cette séquence a été proposée par Donald Knuth vers 1968, et donne un algorithme de complexité $O(n^{3/2})$.
- $h_1=1$ puis $h_k=4^k+3\cdot2^{k-1}+1$ pour $k\geqslant2$.

  Cette séquence a été proposée par Sedgewick en 1986 , et donne un algorithme de complexité $O(n^{4/3})$.

– $h_{max} = n$ puis $h_{k-1} = \left\lfloor \frac{5h_k}{11} \right\rfloor$ avec $h_1 = 1$.

Cette séquence a été proposée par Gonnet et Baeza-Yates en 1991 , et donne un algorithme de complexité théorique inconnue.

– La liste `[1, 4, 10, 23, 57, 132, 301, 701, 1750]`, complétée par $h_{k+1} = \text{round}(h_k * 2.3)$ a été trouvée expérimentalement par Ciura en 2001.

Elle donne de très bons résultats mais a une complexité théorique inconnue.