TP 9 : Algorithmes de tri

Objectifs

- Principe d'un algorithme de tri.
- Analyser leur complexité.
- Démontrer la correction d'un algorithme de tri par des outils simples.
- Notions de tri en place, comparatif, stable.

Étant donnés une liste de nombres L qui contient disons n éléments, un *algorithme de tri* consiste à ré-ordonner cette liste, ou à renvoyer une liste, contenant les mêmes n éléments que L, mais triés dans l'ordre croissant.

1 Tri par sélection

Le principe est de chercher le minimum de `L`, de l'échanger pour le mettre en début de liste, puis de recommencer avec la deuxième plus petite valeur de L, à mettre en deuxième position, etc jusqu'à avoir traité toutes les valeurs de L.

1. Écrire une fonction `indmin(L:list) -> int` qui renvoie un indice i tel que L[i] soit le minimum de L. Elle doit être de complexité linéaire. Combien de comparaisons effectue-t-elle ?
2. Écrire une fonction `trisel(L:list) -> list` qui trie un tableau de nombres L selon le tri par sélection.

   Dans un premier temps on *pourra* fabriquer une liste auxiliaire `Lnew` et on pourra utiliser `del L[q]` pour supprimer l'élément d'indice `q` de la liste `L`, mais il faudra ensuite remplacer cette instruction par quelque chose à la main, au programme.

Définition. On dit qu'un algorithme de tri est *en place* lorsqu'il n'utilise aucune liste auxiliaire. Un tel algorithme ne renvoie rien, mais *modifie* la liste qui lui est passée en entrée.

3. Ainsi, l'algorithme décrit dans la question précédente n'est pas en place. Écrire une version *en place* du tri par sélection, basé sur des échanges de variables. Par exemple pour la liste (514280) il doit effectuer les échanges suivants

$$(514820),\ (014825),\ (012845),\ (012485),\ (012458)$$

4. 🔁 Déterminer la complexité du tri par sélection.
5. 🔁 Proposer un invariant de boucle du tri par sélection (sans le démontrer) et en déduire la correction du tri par sélection.

2 Tri par insertion

On parcourt `L` dans l'ordre et chaque élément est « inséré » (en pratique il est déplacé) à la bonne place parmi les éléments déjà triés jusque là. On s'arrête lorsque l'on a parcouru tous les éléments de `L`. C'est le tri que vous appliquez naturellement pour trier votre main quand vous jouez aux cartes, ou qu'un prof applique pour trier un paquet de copies (pas trop gros, sinon il y a mieux).

Par exemple pour (514280) les états successifs sont

$$(514280),\ (154280),\ (145280),\ (124580),\ (012458)$$

6. Écrire une version *en place* de cet algorithme basé sur des échanges adjacents (pas de `del` ni de `insert`) pour repousser un élément à gauche jusqu'à la position désirée.
7. 🔁 Déterminer la complexité du tri par insertion. On ne comptera que les comparaisons.
8. 🔁 Trouver et démontrer un invariant de boucle simple pour prouver la correction du tri par insertion.

3 Tri par partition-fusion

On commence par une première observation : étant données deux listes triées, il n'est pas très compliqué de les *fusionner* en une seule liste triée.

9. Implémenter une fonction `fusion(L:list, M:list) -> list` qui renvoie une telle liste. On souhaite que cette fonction ait une complexité en $O(n+m)$ où n et m sont respectivement les longueurs de `L` et `M`. Par exemple si `L = [1,2,4,6,7,9]` et `M = [1,3,5,10,11]`, alors on souhaite que `fusion(L,M)` renvoie `[1,1,2,3,4,5,6,7,9,10,11]`.
10. Écrire une fonction *récursive* `trifusion(L:list) -> list`. L'idée est de *diviser pour mieux régner*, en remarquant que pour trier une liste il suffit de fusionner sa première et sa deuxième moitié, triées.
11. Étudier sa complexité.

4 Tri rapide

Le principe est le suivant : on choisit un élément de la liste (au hasard, mettons, on y reviendra). C'est le *pivot*. Puis on modifie la liste de façon à placer

- tous les éléments inférieurs au pivot (hormis le pivot lui-même) à gauche du pivot ;
- tous les éléments strictement supérieurs au pivot à droite du pivot.

Le pivot est alors bien placé, et ne bougera plus jamais. On dit qu'on a *partitionné* la liste par rapport au *pivot*. Sur les parties gauche et droite, on recommence...

12. Écrire une fonction `partitionnement(L:list, ind_piv:int) -> (list,list,list)` qui réalise le partitionnement de `L` et renvoie trois listes : la partie gauche, le singleton pivot, et la partie droite. Ces trois listes doivent former une partition (au sens mathématique) de `L`.
13. Écrire une fonction récursive `trirapide(L:list) -> list`. Justifier qu'elle termine.

    Remarque : pour tirer un entier au hasard uniformément entre a inclus et b exclu, on peut utiliser `random.randrange(a,b)` (après avoir importé le module `random`).

Remarque : le pire des cas est quand à chaque appel récursif, le pivot aléatoire est un minimum de la liste. Auquel cas, il faut effectuer $n-1$ appels récursifs pour arriver aux cas de base. Chacun de ces appels est en coût linéaire (coût de la concaténation), donc le coût total est quadratique.

Cependant : ce pire des cas est extrêmement rare. On peut montrer qu'en *moyenne* (hors-programme), la complexité est $O(n\log(n))$.

5 Tri par comptage

On suppose que l'on souhaite trier une liste L d'entiers naturels, et que l'on connait un majorant a priori de tous ces entiers, disons N. Le principe est tout simplement de compter combien de fois chacun des entiers entre 0 et N apparaît dans la liste, le tout en un seul balayage de L. On veut donc créer la liste auxiliaire $R = [R_0, \dots, R_N]$ où R_i est le nombre de fois que i apparaît dans L. Il suffit ensuite de renvoyer la liste qui contient R_0 zéros, puis R_1 uns, etc ...

14. Écrire une fonction `tricomptage(L:list, N:int) -> list`. Analyser sa complexité.
15. Proposer une solution si l'on ne connait pas de majorant a priori de L.
16. Quels sont les défauts du tri par comptage ? On pourra notamment regarder ce qu'il se passe en *mémoire*.

Remarque : ce tri n'est pas *comparatif*, c'est-à-dire qu'il n'effectue pas de comparaisons entre les valeurs de la liste.

6 Pour aller plus loin : notion de stabilité

Définition. Un tri est dit *stable* lorsque les valeurs égales (pour l'ordre avec lequel on travaille) sont présentées dans le même ordre d'apparition que dans la liste de départ.

Ceci n'est pas vraiment visible si l'on trie simplement des nombres pour l'ordre $\leq$ usuel, par exemple si un algorithme de tri reçoit la liste `L = [7,1,3,3,4]` et renvoie `[1,3,3,4,7]`, on ne peut pas savoir en regardant simplement la sortie si les deux occurences de 3 ont été échangées ou pas.

Imaginons plutôt que l'on souhaite trier une liste de couples (Nom, Note) par ordre croissant de Notes. Exemple : si

```
L = [ (John, 18), (George, 14), (Ringo, 12) (Paul, 18)]
```

alors un algorithme stable renverra

```
L = [ (Ringo, 12), (George, 14), (John, 18), (Paul, 18)]
```

car il ne change pas l'ordre d'apparition de John et Paul.

17. Parmi les tris précédents, lesquels sont stables ?