TD6 : Algorithmique sur les tris

Compétences

- Dessiner l'arbre d'appels d'un tri récursif
- Évaluer de manière empirique la complexité d'un tri

Un peu d'histoire

Durant la seconde Guerre mondiale, La *Moore School of Engineering* à Philadelphie embauche des femmes pour effectuer des calculs de trajectoires. L'une d'elle Betty Holberton est vite remarquée pour ses aptitudes mathématiques et fut choisie pour être l'une des six programmatrices de l'ENIAC (acronyme de l'expression anglaise Electronic Numerical Integrator And Computer). Elle développe en 1952 le premier algorithme de tri connu.

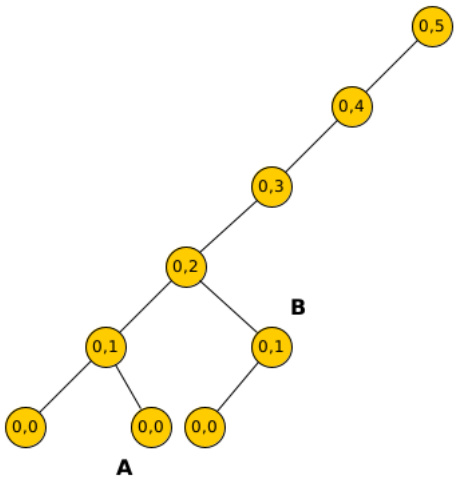
In 1952, Holbetton developed the Sort-Merge Generator" for the UNIVAC I which produced a program to sort and merge files. This was the first step toward actually using a computer to write programs (i.e., a precursor to the concept o f a compiler) and was called "the first major *software routine* ever developed for automatic programming

1 Le XSort un tri insolite

Les lignes de code de cet alogorithme de tri ont été retrouvées dans un état qui ne permet pas de l'utiliser. Les seules indices que l'on possède pour essayer de retrouver le code d'origine est une partie de l'abre des appels simplifié pour le tableau suivant : 3 1 2 7 8 2 et le contenu du tableau pour les noeuds notés A et B :

- A : 1 3 2 7 8 2
- B : 1 2 3 7 8 2 .

```
1  def XSort(t, i, j):
2
3  tmp = t[j-1]
4  XSort(t, i, j-1)
5  XSort(t, i, j-1)
6  return
7  if i >= j:
8  if t[j-1] > t[j]:
9  t[j-1] = t[j]
10 t[j]  = tmp
```



1. À l'aide des indices donnés dans l'énoncé remettre dans l'ordre les lignes de code de la fonction `XSort`
2. Compléter l'arbre des appels.
3. Est-ce que cet algorithme de tri est stable ?
4. Etudier et justifier la complexité en temps au meilleur cas de *XSort* et expliquer à quoi correspond ce meilleur cas.
5. En cherchant de l'information sur ce tri vous trouvez que dans le pire cas il est en $O(2^n)$ avec n la taille du tableau qu'en pensez-vous ? Justifier.

2 Quelques propriétés des tris

Theorem 1 *Le tri par insertion a une durée d'exécution égale à $O(n)$ sur une séquence triée*

Ce théorème suggère que si des sous parties d'un tableau sont déjà triées, il peut être avantageux d'utiliser le tri par insertion qui a une complexité en $O(n)$ dans ce cas.

Theorem 2 *Le tri fusion est exécuté en $O(n\log(n))$*

Le tri fusion fonctionne en divisant à plusieurs reprises le tableau initial en deux jusqu'à ce que les sous-séquences ne possèdent plus qu'un unique élément, puis en fusionnant les sous-séquences par paire jusqu'à obtenir tous les éléments triés.
Sur la base de ces deux idées, nous vous proposons d'étudier deux nouveaux algorithmes.

3 Tri insertion modifié

Étudions un algorithme qui exploite le premier théorème pour obtenir une complexité en $O(n^{1.5})$.

3.1 Principe

L'idée est d'appliquer plusieurs fois le tri par insertion à des sous séquences d'un tableau sur la base suivante :

1. Initialiser $d = 1$
2. Répéter l'étape 3 jusqu'à ce que $9d > n$
3. d prend la valeur $3d + 1$
4. Répéter les étapes 5 et 6 jusqu'à ce que $d = 0$
5. Appliquer le tri par insertion à chacune des d sous-séquences d'incrément d du tableau de départ.
6. Affecter la partie entière $d/3$ à d

3.2 Exemple

- Supposons que le tableau de départ soit composé de $n = 50$ éléments. La boucle de l'étape 2 serait alors répétée 2 fois ce qui ferait augmenter la valeur de départ de $d = 1$ à $d = 4,\ 13$.
- La boucle de l'étape 4 :
  - La première itération applique le tri insertion à chacune des 13 sous-séquences d'incrément 13 $[e_0, e_{13}, e_{26}, e_{39}]$, $[e_1, e_{14}, e_{27}, e_{40}]$, …, $[e_{11}, e_{24}, e_{37}, e_{50}]$, $[e_{12}, e_{25}, e_{38}]$
  - L'étape 6 affecte $d = 4$
  - La deuxième itération applique le tri insertion à chacune des 4 sous-séquences d'incrément 4 $[e_0, e_4, e_8, \ldots, e_{48}]$, $[e_1, e_5, e_9, e_{13}, \ldots, e_{49}]$, …
  - L'étape 6 affecte $d = 1$
  - La troisème et dernière itération applique le tri insertion à tout le tableau.

3.3 Analyse théorique

1. Écrire une fonction `newSort1(T, n)` implémentant l'algorithme décrit ci-dessus.
2. La première impression est de se dire qu'il est beaucoup plus long d'utiliser le tri newSort1 plutôt que d'appliquer directement le tri par insertion à toute la séquence en seul fois. Donner le nombre totale de comparaisons nécessaire pour l'exemple étudier si on considère qu'à chaque étape la fonction de complexité est en n^2
3. Comparer au nombre de comparaisons effectuées par le tri insertion sur toute la séquence.
4. Quelle hypothèse peut-on faire après la première itération de l'étape 4 pour le tri newSort1 ?
5. Évaluer avec cette hypothèse le nombre de comparaisons effectuées.
6. Comparer la complexité en temps et en espace de cet algorithme avec celui du tri insertion.

4 Diviser pour régner

On va maintenant essayer de tirer avantage de sous-séquences triées mais avec la fonction `fusionner`.

4.1 Principe

L'idée est d'appliquer plusieurs fois la fonction `fusionner` à des sous séquences d'un tableau sur la base suivante :

1. Découper le tableau en sous-séquences consécutives constituées chacune d'éléments triés en partant de l'extrémité gauche du tableau.
2. Répéter l'étape 3 jusqu'à ce qu'il n'y ait plus qu'une seule sous-séquence.
3. En partant de la gauche du tableau fusionner deux à deux les sous-séquences consécutives triées pour ne former qu'une seule sous séquence. S'il y a un nombre impair de sous-séquences la dernière reste inchangée.

4.2 Exemple

- Soit le tableau `[2, 7, 3, 8, 5, 3, 12]`. La première étape découpe le tableau en 4 sous séquences consécutives triées : `[2, 7]`, `[3, 8]`, `[5]`, `[3, 12]`
- La boucle de l'étape 2 engendre les resultats suivants :
  — `[2, 3, 7, 8, 5, 3, 12]`
  — `[2, 3, 7, 8, 3, 5, 12]`
  — `[2, 3, 3, 5, 7, 8, 12]`

4.3 Analyse théorique

1. Écrire une fonction `newSort2(T, n)` implémentant l'algorithme décrit ci-dessus.
2. Faire une analyse de la complexité en temps et en espace de cet algorithme dans le pire et le meilleur cas. Expliquez précisément à quoi correspondent les pire et meilleur cas.
3. Quelle est la complexité temporelle dans le pire cas de l'algorithme `newSort2` si le tableau de taille n est constitué de $k < n$ sous-séquences triées de taille identique ? Par exemple `[1, 5, 2, 6, 4, 8, 3, 12]`.