Algorithmique et Structure discrète : TP2

Ayoub BARICH, Camille VEDANI

Résumé

On s'intéresse au problème de Tri qui consiste à trier les éléments d'un tableau d'entiers relatifs donné, dans l'ordre croissant. Il existe plusieurs méthodes de tri. L'objectif de ce TP est de comparer expérimentalement différentes méthodes de tri :
- Tri par insertion
- Tri par sélection
- Tri rapide (avec choix aléatoire ou arbitraire du pivot)
- Tri par fusion
- Tri par tas

La fonction Array génère un tableau aléatoirement.

Vous pouvez consulter la mise en œuvre de chaque méthode ainsi que les tests sur le lien suivant :

Préambule :

Pour des raison de praticité, nous avons eu une approche orientée objet pour notre TP. Nous avons créé une classe ArrayFunc qui regroupe toutes les méthodes portant sur la manipulation des tableaux.

- **iterate** : Cette fonction sert à échanger deux termes dans le tableau.
- **GetMin** : Cette fonction est utilisée pour obtenir le plus petit élément d'un tableau d'entiers relatifs en comparant les éléments du tableau.

Pseudo-code :

```
1    iterate(liste Array,firstIndex,secondIndex):
2        tmp<-Array[firstIndex]                                C1
3        Array[firstIndex]<-Array[secondIndex]                 C2
4        Array[secondIndex]<-tmp                               C3
5    Retourner Array
```

```
1    getMin(liste Array, firstIndex):
2        min<-firstIndex                                        C1
3        Pour i allant de firstIndex+1 jusqu'à len(Array) :     nC2
4            Si Array[min] < Array[i]:                          (n-1)C3
5            min<-i                                             (n-1)C4
6    Retourner min
```

Complexités des méthodes :

Pour calculer la complexité de notre algorithme, on cherche une majoration en fonction de n (la taille de notre tableau) , telle que :

$$\forall n \in \mathbb{N}\ \exists C > 0\ et\ n_0,\ \forall n > n_0\ \frac{f(n)}{g(n)} \leq C$$

Pour chaque question du TP, on se basera sur cette définition. En ce qui concerne cette première méthode, on pose $T(n)$ *notre temps d'execution totale*. Pour simplifier notre travail, on prend $C = MAX_{i\in[1,k]}\ (C_i)$, k étant le numéro de ligne de notre algorithme.

iterate
On a :

$$T_{iterate}(n)\ \leq\ 3C\ =\ O(1)$$

getMin
On a :

$$T_{getMin}(n)\ \leq\ C(2(n-1)+n+1)\ =\ O(n)$$

1 Méthode 1 : Tri par insertion (Insertion Sort)

Pseudo-code :

```
1  Main(Array) :
2      Pour k allant de 1 jusqu'à la longueur de de Array, faire:  C1(n+1)
3          Key<-Array[k]                                           C2(n)
4          i = k-1                                                 C3(n)
5          Tant que i>=0 et Array[i]> key, faire :                 C4(n(n+1))
6              Array[i+1] <- Array[i]                              C5(nxn)
7              i <- i-1                                            C6(nxn)
8              Array[i+1] <- key                                   C7(nxn)
```

Complexité :

On a :

$$T_{Main}(n) \leq\ C(4n^2+4n)\ =\ O(n^2)$$

Structure utilisée :
Cet algorithme utilise des tableaux comme structure.

2 Méthode 2 : Tri par selection (Selection Sort)

Pseudo-code :

```
1  Main( liste Array) :
2      lenght <- len(Array)                                  C1
3      Pour i allant de 0 à lenght :                        (n+1)C2
```

```
        min <- getMin(Array,i)                                  nO(GetMin)
        Si Array[i]<Array[min] :                                  nC4
            échanger Array[min]<->Array[i]                        nC5
    Retourner Array
```

Compléxité :

Pour la fonction **Main** de la méthode 2, en prenant $C = MAX_{i\in[1,k]}\ (C_i)$ on a :

$$T_{Main}(n)\ \leq\ C(nO(GetMin) + 3n + 2) = O(nO(n) + 3n + 2) = O(n^2)$$

Structure utilisée :

Cette méthode utilise aussi les tableaux comme structure .

3 Méthode 3 : Tri rapide (QuickSort)

Pseudo-code :

```
Partition(liste Array, firstIndex,lastIndex):

    pivot <- un l'index d'un terme de la liste Array                C1
    index <- firstIndex-1                                           C2
    Array <- ArrayFunc.iterate(Array,lastIndex,pivot)               C3
    Pour i allant de firstIndex à lastIndex, faire:                nC4
            Si Array[i]<=Array[lastIndex]:                         nC5
                index+=1                                            C6
                échanger Array[index]<->Array[i]                C8nO(iterate)
        échanger Array[lastIndex]<->Array[index+1]                  C9
        retourner index+1
```

```
quickSort(liste Array,firstIndex, lastIndex):

    Si firstIndex<lastIndex:                                        C1
        pivot <- Partition(Array,firstIndex,lastIndex)          O(Partition)
        quickSort(Array,firstIndex,pivot-1)                     O(quickSort)
        quickSort(Array,pivot+1,lastIndex)                      O(quickSort)
```

Complexité :

Partition :

On a

$$T_{Partition}(n)\ \leq\ C(nO(iterate) + 2n + 5) = O(nO(1) + 3n + 2) = O(n)$$

QuickSort :

Cette complexité dépend des tableaux utilisés par la méthode. On étudie le cas le plus défavorable, c'est-à-dire partitionner le tableau de façon complètement déséquilibrée, par exemple si le tableau est déjà trié à l'inverse. Dans ce cas, on a :

$$T(n) = 2T(n-1) + O(Partition) = 2T(n-1) + Cn = \sum_{k=1}^{n-1} Ck = C.\frac{n(n-1)}{2} = O(n^2)$$

Dans le pire des cas, cet algorithme a une complexité $O(n^2)$

Dans le cas le plus favorable, c'est-à-dire chaque fois que le tableau est partitionné dans deux tableaux de même taille, on a :

$$T(n) \leq 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(nlog(n))$$

D'après le cours, en considérant que les n ! permutations ont la même probabilité d'être exécutées, l'espérance de cette loi serait O(nlog(n)).
Donc la complexité de cette méthode est O(nlog(n)).

Structure utilisée :

Cet algorithme utilise les tableaux comme structure.

4 Méthode 4 : Tri par fusion (Merge Sort)

Pseudo-code : On pose S(tn

```
main(Array):
   Si len(Array)>1:                                                    C1
       midPoint <- len(Array)//2                                       C2

       rightArray <- Array[midPoint:len(Array)]                        C3
       leftArray <- Array[0:midPoint]                                  C4

       main(rightArray)                                               0(main)/2
       main(leftArray)                                                0(main)/2
       i=0                                                              C7
       j=0                                                              C8
       k=0                                                              C9
       Tant que i <len(leftArray) et j <len(rightArray), faire :        n
           Si leftArray[i]<=rightArray[j]:                           C11 x (n-1)/2
               Array[k] <- leftArray[i]                              C12 x (n-1)/2
               i+=1                                                  C13 x (n-1)/2
           Sinon:
               Array[k] <- rightArray[j]                             C14 x (n-1)/2
               j+=1                                                  C15 x (n-1)/2
           k+=1                                                      C16 x (n-1)/2
       Tant que i < len(leftArray), faire :                              n/2
           Array[k] <- leftArray[i]                                  C18 x (n-1)/2
           i+=1                                                      C19 x (n-1)/2
           k+=1                                                      C20 x (n-1)/2
       Tant que j<len(rightArray), faire :                             C21 n
           Array[k] <- rightArray[j]                                 C22 x (n-1)/2
           j+=1                                                      C23 x (n-1)/2
           k+=1                                                      C24 x (n-1)/2
```

Complexité :

Le calcule de notre complexité sera un peu laborieux sans simplification ,donc on introduit deux constantes A et B telles que :

On définit notre complexité totale comme :

$$T_{main}(n) = \begin{cases} A & n = 1 \\ 2T_{main}(\frac{n}{2}) + An + B & n > 1 \end{cases}$$

On va faire une preuve itérative pour trouver une majoration de notre complexité totale.

$$T_{main}(n) = 2T_{main}(\frac{n}{2})+An+B = 4T_{main}(\frac{n}{4})+2An+2B = ... = \log_2(n)(An+B)+2^{\log_2(n)}T(1)$$
$$T_{main}(n) = \log_2(n)An + \log_2(n)(B) + An = O(n\log_2(n))$$

En considérant que chaque permutations à une probabilité égale de se produire, la moyenne de cela vaut O(nlog(n))

Structure utilisée :

Cet algorithme utilise aussi les tableaux comme structures.

5 Méthode 5 : Tri par tas (Heap Sort)

Pseudo-code :

```
1  EntasserMax(Array,lenght,index):
2
3      root<-index                                                  C1
4
5      leftChild<-(2*index)+1                                       C2
6      rightChild<-(2*index) +2                                     C3
7
8      Si leftChild < lenght and Array[root]<Array[leftChild]:      C4
9          root <- leftChild                                        C5
10
11     Si rightChild < lenght and Array[root]<Array[rightChild]:    C6
12         root <- rightChild                                       C7
13
14     Si root != index:                                            C8
15         Array <- ArrayFunc.iterate(Array,root,index)             C9
16         EntasserMax(Array,lenght,root)                    O(EntasseMax)C10
17
18 main (Array):
19
20     lenght <- len(Array)                                                  C1
21     Pour i allant de (lenght//2) à -1 (dans l'ordre décroissant), faire:  nC2
22         EntasserMax(Array,lenght,i)                              nO(EntasserMAX)
23     Pour i allant de lenght-1 à 0 (dans l'ordre décroissant), faire :     nC4
24         Array <- ArrayFunc.iterate(Array,i,0)                          (n-1)C5
25         EntasserMax(Array,i,0)                                nO(EntasserMax)C6
```

Complexité :
Avant de calculer la complexité, intéressons-nous à la hauteur de l'arbre
D'après le lemme du cours, un arbre binaire parfait avec n noeud a une hauteur de

$$log_2(n+1) - 1$$

EntasserMax

On appelle n le nombre de noeuds de T qui ne sont pas au dernier niveau et T' l'arbre parfait défini par ces noeuds.
Alors : $h(t) = h(t') + 1 = log_2(n' + 1) - 1 + 1 = log_2 \leq log_2(n + 1)$
D'après le cours, si le noeud de la valeur "index" dans EntasserMax a une hauteur h, alors la complexité est O(h). Comme son sous-arbres est un tas avec $n' \leq n$ noeuds, d'après le calcul précédent, on a : $h = log_2(n' + 1) \leq log_2(n + 1)$

On a donc une complexité de O(log(n)) pour EntasserMax.

Main

Pour la complexité, on a :

$$T_{main}(n) = C(n+n+1+1-1+2nO(EntasserMax)) = C(2n+1+2nO(log_2(n))) = O(nlog_2(n))$$

Structure utilisée :

Cet algorithme utilise les arbres binaires comme structures.

6 Synthèse

Sur les graphiques ci-dessous, les complexités trouvées précédemment correspondent avec les temps d'exécution de celles-ci. On remarque que le tri le plus efficace est le tri par tas.

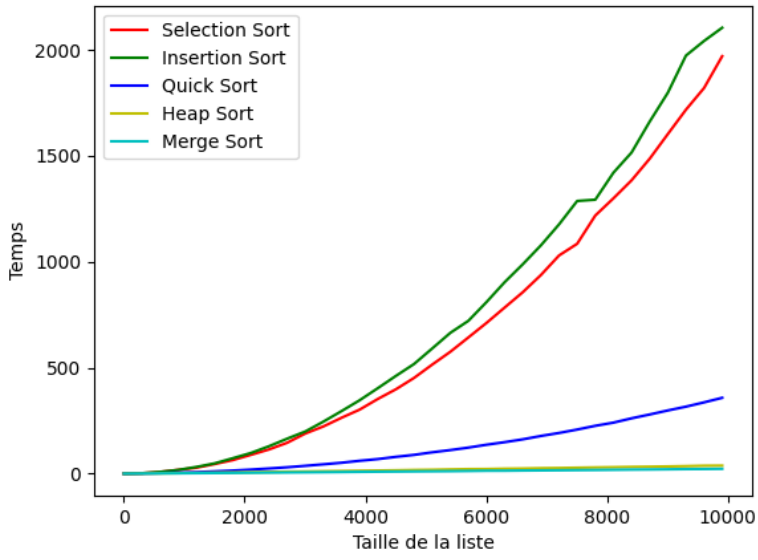

Figure 1 Temps d'exécution des différentes méthodes pour des tableaux de taille 10000