2015-2016

SINF-1121 Algorithmique et structures de données

Synthèse

Damien Deprez

18 décembre 2015

Table des matières

Chapitre 1 - Algorithme de Tri

1.1 Selection Sort

L'algorithme de selection sort consiste à chercher dans le tableau le premier plus petit élément et le placer à la première place du tableau. Répéter l'opération pour le deuxième plus petit élément et ainsi de suite. La complexité de cet algorithme est en $\mathcal{O}(N^2)$.

Algorithm 1: Selection Sort

```
Input: T est un tableau de taille N
Result: Trie le tableau T
for i de 0 à N-1 do
    min ← i
    for j de i+1 à N-1 do
        if T[j] < T[min] then
            min ← j
        end
    end
    if min ≠ i then
        T[i] ↔ T[min]
    end
end
```

1.2 Insertion Sort

L'algorithme d'insertion sort consiste à parcourir le tableau une fois. Si l'élément i est plus petit que celui à la position $i-1$, alors on le fait avancer jusqu'à ce qu'il soit plus grand où égal à l'élément $i-j$. La complexité de cet algorithme est en $\mathcal{O}(N^2)$ dans le pire cas. Dans le meilleur cas, c'est en $\mathcal{O}(N)$.

Algorithm 2: Insertion Sort

```
Input: T est un tableau de taille N
Result: Trie le tableau T
for i de 0 à N-1 do
    j ← i
    while j > 0 et T[j-1] ≥ T[i] do
        T[j] ↔ T[j-1]
        j← j-1
    end
end
```

1.3 Shell Sort

L'algorithme de Shell Sort est une amélioration de l'algorithme d'Insertion Sort. Le Shell Sort parcours plusieurs fois le tableau mais compare l'élément de la position i avec celui de la position $i-g$ avec g un écart fixé (petit par rapport à la taille du tableau). Chaque parcours de tableau, g diminue pour atteindre 1 à la fin. La complexité de cet algorithme dépends de la séquence de g prise. La séquence 1, 4, 13, 40, 121, 364 donne une complexité de $\mathcal{O}(N^{3/2})$.

Algorithm 3: Shell Sort

```
Input: T est un tableau de taille N
Result: Trie le tableau T
h ← 1
while h<N/3 do
    h ← 3*h+1
end
while h ≥ 1 do
    for i de h à N-1 do
        j ← i
        while j ≥ h et T[j-h] ≥ T[i] do
            T[j] ↔ T[j-h]
            j← j-h
        end
    end
    h ← h/3
end
```

1.4 Merge sort

L'algorithme de Merge Sort est basé sur le principe de *Diviser pour régner*. L'opération principale de cet algorithme est donc de fusionner deux tableaux triés en un seul. Cet étape ce fait en temps linéaire. On effectue donc cette fusion sur les deux moitiés du tableau entré et ainsi de suite. La complexité de cet algorithme est en $\mathcal{O}(N \log(N))$.

Algorithm 4: Merge Sort

```
Function MergeSort(T)
    Input: T, un tableau de taille N
    Data: R, un tableau vide de taille N
    Output: R, un tableau trié de taille N
    if N>1 then
        A ← MergeSort(T[ 0,...,N/2])
        B ← MergeSort(T[N/2+1,...,N − 1])
        a ← 1
        b ← 1
        for i de 0 à N-1 do
            if (a ≥ size(A) et b > size(B)) ou A[a] ≥ B[b] then
                R[i] ← A[a]
                a ← a+1
            else
                R[i] ← B[b]
                b ← b+1
            end
        end
    end
    return R
end
```

1.5 Quick Sort

L'algorithme de Quick Sort est basé sur le principe de *Diviser pour régner*. L'opération principale de cet algorithme est de partitionner un tableau en deux autour d'un éléments appelé le pivot. Ensuite, on recommence l'opération sur les deux tableaux obtenus. Le pivot peut-être choisi de manière aléatoire dans intervalle 0 N. La complexité de cet algorithme est en $\mathcal{O}(N \log(N))$.

Algorithm 5: Quick Sort

```
Function QuickSort(T, lo, hi)
    Input: T, un tableau de taille N
    Input: lo entier index de début du tableau
    Input: hi entier index de fin du tableau
    if lo<hi then
        pivot ← Random(lo,hi)
        T[pivot] ↔ T[hi]
        j ← lo
        for i de lo à hi-1 do
            if T[i] ≤ T[hi] then
                T[i] ↔ T[j]
                j ← j+1
            end
        end
        T[hi] ↔ T[j]
        pivot ← j
        QuickSort(T, lo, pivot − 1)
        QuickSort(T, pivot + 1, hi)
    end
end
```

Chapitre 2 - Algorithme de Recherche

2.1 Recherche Séquentielle

L'algorithme de recherche séquentielle permet de trouver une clé dans une table de symbole non-triée représentée par une liste chaînée. Cette recherche scanne toutes les clés de la table de symbole jusqu'à trouver celle qui est recherchée. La complexité de cet algorithme est en $\mathcal{O}(n)$.

2.2 Recherche Binaire

L'algorithme de rechercher binaire permet de trouver une clé dans une table de symbole tirée représentée par un tableau. Si l'élément est plus grand que la clé située au milieu du tableau, alors il se trouve dans le sous tableau de gauche sinon, il se trouve dans le sous tableau de droite. On recommence l'opération de comparaison en déplaçant les indices début, milieu et fin au bord du bon sous tableau. Si l'indice de fin et plus petit que l'indice de début du tableau c'est que l'élément ne se trouve pas dedans. La complexité de cet algorithme est en $\mathcal{O}(\log(n))$.

2.3 Structure de données pour les Tables de Symboles

2.3.1 Arbres binaires

La représentation sous forme d'arbre binaire une table de symboles permet d'allier la flexibilité des listes chaînées dans l'ajout et la suppression d'élément et la rapidité de la recherche binaire. La recherche dans un arbre binaire dépends de la forme de l'arbre. Si l'arbre est équilibré, la rechercher est en $\mathcal{O}(\log(n))$. Si l'arbre n'est pas équilibré, le pire cas est en $\mathcal{O}(n)$. La forme de l'arbre dépend de l'ordre d'insertion des clés.

2.3.2 Arbres balancés

L'avantage de l'arbre balancés par rapport à l'arbre binaire est une garantie d'une complexité de recherche en $\mathcal{O}(\log(n))$. Pour ce faire, on va utiliser un arbre 2-3. Chaque nœud de l'arbre 2-3 est soit constitué d'une clé et deux liens vers deux autres nœuds fils (*2-Node*), soit de 2 clés avec trois liens vers des trois autres nœuds fils (*3-Node*). Dans le cadre de trois 3 liens, le lien du milieu veut dire que ce qui est en dessous est entre la clé 1 et la clé 2.

Insertion

2-Node *3-Node* *3-Node* parent *2-Node* *3-Node* parent *3-Node*