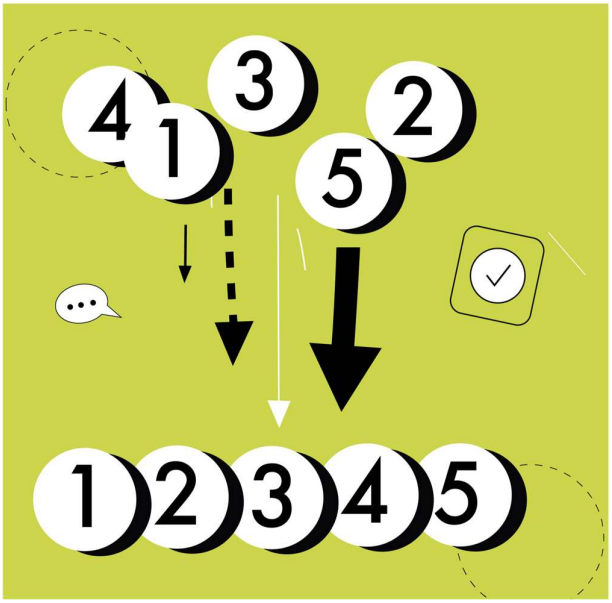

ETUDE VISUELLE SUR LES ALGORITHMES DE TRI

En utilisant le langage de programmation C

YOUNESS HATTABI
Étudiant de première année filière GLSID

YOUSSEF NAZIH
Étudiant de première année filière GLSID

Table des matières

Chapitre I : Programmation des tris et de graphes

Introduction

Dans ce chapitre, nous allons explorer le code source qui alimente à la fois les algorithmes de tri et l'affichage graphique. Nous allons programmer en C et utiliser Gnuplot pour les graphes.

1. Programmation des tris

Nous avons choisi de programmer six algorithmes de tri : Bubble, Insertion, Sélection, Merge, Quick et Counting :

1.1. Le tri à bulles

Le tri à bulles (ou "Bubble sort") est un algorithme de tri simple qui fonctionne en parcourant un tableau plusieurs fois. À chaque passage, il compare deux éléments adjacents et les échange s'ils sont dans le mauvais ordre. Ce processus est répété jusqu'à ce que le tableau soit entièrement trié. Bien qu'il soit facile à comprendre, cet algorithme est inefficace pour de grands tableaux à cause de sa complexité en $O(n^2)$.

```c
// Bubble sort
void bubbleSort(long int a[], long int n)
{
    for (long int i = 0; i < n - 1; i++)
    {
        for (long int j = 0; j < n - 1 - i; j++)
        {
            if (a[j] > a[j + 1])
            {
                swap(&a[j], &a[j + 1]);
            }
        }
    }
}
```

Figure 1: Code de tri à bulles (Bubble sort)

1.2. Le tri par insertion

Le tri par insertion est un algorithme de tri qui construit progressivement un tableau trié en déplaçant les éléments un par un. À chaque itération, il prend un élément du tableau non trié et le place à la position correcte dans la partie triée. Ce processus se répète jusqu'à ce que tous les éléments soient triés. Le tri par insertion est efficace pour les petits tableaux ou les tableaux presque triés, avec une complexité en $O(n^2)$.

```c
void insertionSort(long int arr[], long int n)
{
    long int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Figure 2: Code de tri par insertion (Insertion Sort)

1.3. Le tri par sélection

Le tri par sélection est un algorithme de tri qui divise le tableau en deux parties : une partie triée et une non triée. À chaque itération, il recherche l'élément le plus petit (ou le plus grand) dans la partie non triée, puis l'échange avec le premier élément de cette partie. Ce processus est répété jusqu'à ce que tout le tableau soit trié. Le tri par sélection a une complexité en $O(n^2)$ et est peu efficace pour les grands tableaux.

```
void selectionSort(long int arr[], long int n)
{
    long int i, j, midx;
    for (i = 0; i < n - 1; i++)
    {
        midx = i;
        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[midx])
                midx = j;
        swap(&arr[midx], &arr[i]);
    }
}
```

Figure 3: Code de tri par sélection (Sélection Sort)

Ces trois premiers algorithmes de tri sont très faciles à programmer et à implémenter dans la plupart des langages, mais souffrent de performances médiocres. En revanche, les trois suivants sont beaucoup plus difficiles à programmer, mais offrent des temps d'exécution bien plus rapides.

1.4. Le tri fusion

Premièrement, il y a le tri fusion. Cet algorithme est basé sur la méthode "diviser pour régner" et divise récursivement le tableau en sous-tableaux jusqu'à ce qu'ils aient une taille de 1, puis les fusionne tout en les triant. Il est efficace avec une complexité en O(nlog n), mais nécessite plus de mémoire pour la fusion.

Pour l'implémenter dans notre code, nous divisons la fonction en deux parties : une fonction de fusion (`merge`) :

```
void merge(long int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0; j = 0; k = l;
    while (i < n1 && j < n2){
        if (L[i] <= R[j]){
            arr[k] = L[i]; i++;
        }
        else{
            arr[k] = R[j]; j++;
        } k++;
    }
    while (i < n1){
        arr[k] = L[i];
        i++; k++;
    }
    while (j < n2){
        arr[k] = R[j];
        j++; k++;
    }
}
```

Figure 4: Code de la fonction de fusion

Et pour la deuxième partie on a une fonction principale de tri fusion (`mergeSort`) :

```
void mergeSort(long int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Figure 5: Code de tri par fusion (Merge Sort)

1.5. Le tri rapide

Le tri rapide (Quick Sort) est un algorithme de tri efficace aussi basé sur la méthode "diviser pour régner". En C, il fonctionne en sélectionnant un "pivot" à partir du tableau et en réorganisant les éléments de manière que les éléments plus petits que le pivot soient placés à gauche et ceux plus grands à droite. Ensuite, il applique récursivement le même processus aux sous-tableaux à gauche et à droite du pivot. Ce processus permet de trier le tableau en divisant successivement le problème en sous-problèmes plus petits. L'algorithme a une complexité moyenne de O (n log n).

Pour l'implémenter dans notre code, nous divisons la fonction en deux parties : une fonction de partition (`partition`) :

```
int partition(long int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}
```

Figure 6: Code de la fonction de partition

Et pour la deuxième partie on a une fonction principale de tri rapide (`quickSort`) :

```
void quickSort(long int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Figure 7 : Code de tri rapide (Quick Sort)

1.6. Le tri par dénombrement

Le tri par dénombrement (Counting Sort) est un algorithme qui trie des nombres en comptant combien de fois chaque valeur apparaît (**fréquence**). On crée un tableau pour stocker ces fréquences, puis on utilise ces informations pour remettre les nombres dans le bon ordre. Il est rapide si les nombres à trier sont dans une plage de valeurs **limitée**, car il parcourt les données en fonction de ces fréquences.

Ce qui différencie le tri par dénombrement des cinq premiers algorithmes mentionnés, c'est qu'il est non comparatif. Sa complexité est O(n + r), où n est le nombre d'éléments et r la différence entre la plus grande et la plus petite valeur.

```
void countSort(long int inputArray[], int N)
{
    int M = 0;
    for (int i = 0; i < N; i++)
        if (inputArray[i] > M)
            M = inputArray[i];
    int *countArray = (int *)calloc(M + 1, sizeof(int));
    for (int i = 0; i < N; i++)
        countArray[inputArray[i]]++;
    for (int i = 1; i <= M; i++)
        countArray[i] += countArray[i - 1];
    int *outputArray = (int *)malloc(N * sizeof(int));
    for (int i = N - 1; i >= 0; i--)
    {
        outputArray[countArray[inputArray[i]] - 1] = inputArray[i];
        countArray[inputArray[i]]--;
    }

    for (int i = 0; i < N; i++)
        inputArray[i] = outputArray[i];
    free(countArray);
    free(outputArray);
}
```

Figure 8 : Le code de tri par dénombrement

2. Fonction main

2.1. Calcule des temps d'exécution en microsecondes

Dans la première partie de la fonction main, une boucle while générera des nombres aléatoires pour remplir des tableaux de tailles variant de 10 000 à 100 000. Ces tableaux seront triés à l'aide des algorithmes de tri mentionnés précédemment, et leurs temps d'exécution seront mesurés. Chaque algorithme sera exécuté 10 fois, et les résultats, sont affichés à la fois à l'écran et enregistrés dans le fichier sort.txt, serviront ensuite à une analyse avec GNUPlot.

Pour obtenir le temps pris par chaque algorithme de tri, la fonction `clock()` a été utilisée. Elle renvoie des valeurs en ticks de processeur, qui varient d'un ordinateur à l'autre. Pour obtenir des valeurs en microsecondes, on divise cette valeur par `CLOCKS_PER_SEC` et on multiplie par 1 000 000.

```c
int main(){
    FILE *fichierOutput = fopen("sort.txt", "w");
    if (fichierOutput == NULL) return -1;
    long int n = 10000; int it = 0;
    double tim1[10], tim2[10], tim3[10], tim4[10], tim5[10], tim6[10];
    fprintf(fichierOutput,"Taille de tableau, Bubble, Insertion, Selection, Merge, Quick, Counting\n");
    printf("Taille de tableau, Bubble, Insertion, Selection, Merge, Quick, Counting\n");
    while (it++ < 10){
        long int a[n], b[n], c[n], d[n], e[n], f[n];
        for (int i = 0; i < n; i++){ long int no = rand() % n + 1; a[i] = no; b[i] = no; c[i] = no; d[i] = no; e[i] = no; f[i] = no;}
        clock_t start, end;
        start = clock(); bubbleSort(a, n); end = clock();
        tim1[it] = ((double)(end - start) / CLOCKS_PER_SEC) * 1000000;
        start = clock(); insertionSort(b, n); end = clock();
        tim2[it] = ((double)(end - start) / CLOCKS_PER_SEC) * 1000000;
        start = clock(); selectionSort(c, n); end = clock();
        tim3[it] = ((double)(end - start) / CLOCKS_PER_SEC) * 1000000;
        start = clock(); mergeSort(d, 0, n - 1); end = clock();
        tim4[it] = ((double)(end - start) / CLOCKS_PER_SEC) * 1000000;
        start = clock(); quickSort(e, 0, n - 1); end = clock();
        tim5[it] = ((double)(end - start) / CLOCKS_PER_SEC) * 1000000;
        start = clock(); countSort(f, n); end = clock();
        tim6[it] = ((double)(end - start) / CLOCKS_PER_SEC) * 1000000;
        fprintf(fichierOutput, "%li, %li, %li, %li, %li, %li, %li\n",
                n,
                (long int)tim1[it], (long int)tim2[it], (long int)tim3[it], (long int)tim4[it], (long int)tim5[it], (long int)tim6[it]);
        printf("%li, %li, %li, %li, %li, %li, %li\n",
                n,
                (long int)tim1[it], (long int)tim2[it], (long int)tim3[it], (long int)tim4[it], (long int)tim5[it], (long int)tim6[it]);
        n += 10000;
    }
```

Figure 9: Première partie de la fonction main

2.2. Affichage des graphes

Le tracé en échelle normale permet de mieux observer la complexité en temps des algorithmes, mais les trois derniers tris sont si efficaces qu'ils deviennent presque invisibles sur le graphe, pour resourdre cette probleme on donne a l'utilisateur trois options dans la deuxième partie de la fonction `main()` : tracer le graphe en échelle normale, utiliser l'option logscale y pour mieux visualiser tous les algorithmes, ou afficher les deux versions pour une comparaison complète.

```c
    fclose(fichierOutput);
    int choix;
    printf("\n\nType de graphe: [1] Pour normale, [2] Pour temps logarithmique, [3] Pour les deux:  ");
    scanf("%d", &choix);
    printf("\nAttendez 3 secondes...");
    sleep(3);
    if (choix == 1){
        system("gnuplot -p -e \"set title \\\"Performance d\'algorithmes de tri\\\"; set xlabel \'Taille de tableau\'; set ylabel \'Temps
    }
    else if (choix == 2){
        system("gnuplot -p -e \"set title \\\"Performance d\'algorithmes de tri\\\"; set xlabel \'Taille de tableau\'; set ylabel \'Temps
    }
    else{
        system("gnuplot -p -e \"set title \\\"Performance d\'algorithmes de tri\\\"; set xlabel \'Taille de tableau\'; set ylabel \'Temps
        system("gnuplot -p -e \"set title \\\"Performance d\'algorithmes de tri\\\"; set xlabel \'Taille de tableau\'; set ylabel \'Temps
    }
    return 0;
}
```

Figure 10: Deuxième partie de la fonction main

La fonction `system()` exécute le script GNUPlot via le terminal de commandes.

```
set title "Performance d'algorithmes de tri"
set xlabel "Taille de tableau"
set ylabel "Temps en ms"
set grid
set key outside
set xtics rotate by -90
set logscale y
plot "sort.txt" using 1:2 with lines title "A bulles", \
"sort.txt" using 1:3 with lines title "Insertion",\
"sort.txt" using 1:4 with lines title "Selection",\
"sort.txt" using 1:5 with lines title "Fusion",\
"sort.txt" using 1:6 with lines title "Rapide",\
"sort.txt" using 1:7 with lines title "Denombrement"
```

Figure 11: Script de GNUPlot

Chapitre II : Resultats et comparaisons

1. Résultats sous forme de texte

Après l'exécution du programme, les résultats sont enregistrés dans un fichier afin que l'utilisateur puisse les consulter au format brut et que GNUPlot puisse les utiliser pour créer des graphes.

```
Taille, Bubble, Insertion, Selection, Merge, Quick, Counting
10000, 403143, 80180, 149384, 2041, 1684, 301
20000, 1627232, 580321, 579857, 5571, 2814, 443
30000, 3921502, 754860, 1303615, 7112, 4710, 796
40000, 6668857, 1293043, 2268414, 8268, 7941, 2203
50000, 10814630, 2067703, 3669423, 11059, 10663, 1527
60000, 15578818, 4999102, 5283715, 15327, 8874, 1396
70000, 21337426, 4039185, 7300198, 17748, 14060, 2268
80000, 29169008, 9356952, 9257077, 16982, 14732, 1967
90000, 36406466, 11640960, 11589935, 23766, 18641, 2275
100000, 44112398, 8269380, 14354502, 24480, 20984, 2646
```

Figure 12: Les temps d'exécution en forme texte

2. Résultats sous forme de graphes GNUPlot

Nous avons deux graphes, comme mentionné précédemment : un graphe à échelle normale et un graphe à échelle logarithmique. Sur le graphe normal, la complexité temporelle des trois premiers tris, qui est $O(n^2)$ est très évidente. En revanche, nous avons du mal à observer le reste des algorithmes de tris en raison de leur temps d'exécution très faible.

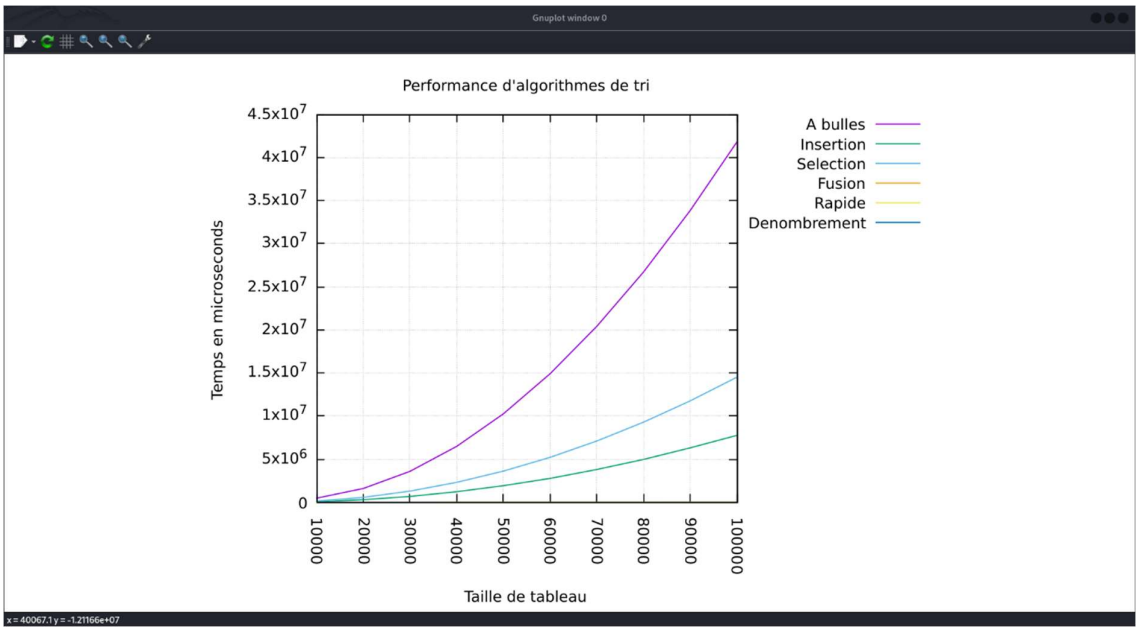

Figure 13: Graphe à échelle normale

Pour comparer les algorithmes de tri avec l'augmentation de la taille de tableau, nous utiliserons un graphe à échelle logarithmique.

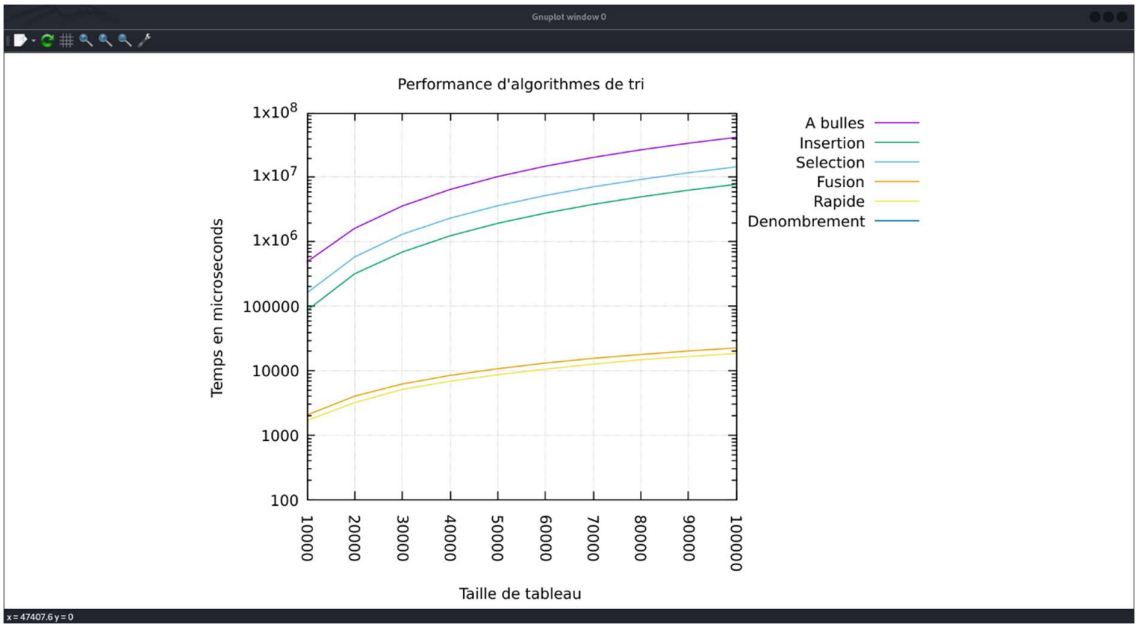

Figure 14: Graphe à échelle logarithmique

3. Observations et comparaisons

Les trois premiers algorithmes de tri sont beaucoup plus lents que les trois derniers en raison de leur complexité temporelle, qui est de $O(n^2)$. Cependant, on remarque que le tri par insertion et le tri par sélection offrent de bien meilleures performances que le tri à bulles à mesure que la taille des données augmente. Cela montre que bien que la complexité temporelle soit un facteur important, il ne faut pas s'y fier exclusivement dans ce type d'expérimentations.

Quant aux trois derniers algorithmes, ils sont extrêmement rapides, avec le tri par dénombrement qui se distingue comme le plus rapide et bénéficiant de la meilleure complexité temporelle, $O(n+r)$. Cependant, comme mentionné auparavant, la complexité temporelle ne fait pas tout. Cela se reflète dans le fait que le tri par dénombrement, malgré son efficacité théorique, n'est pas aussi répandu que d'autres algorithmes de tri.

Conclusion

En conclusion, les algorithmes de tri diffèrent largement en termes d'efficacité selon la taille et la structure des données, ainsi que la complexité temporelle de l'algorithme. Bien que la complexité théorique fournisse des indications utiles, les performances réelles peuvent être influencées par d'autres facteurs, tels que l'utilisation de la mémoire et les spécificités de l'implémentation. Il est donc important de choisir l'algorithme adapté au contexte et aux besoins de la tâche, plutôt que de se fier uniquement à l'analyse de la complexité