3ème année InfoTronique

ITC313

Programmation C/C++

Nicolas TOURRETTE

TP6 – Promotion Pascal

ESIREM Dijon

Plan

CHAPITRE 1

Introduction

I Objectifs du TP

Ce TP a pour objectifs d'implémenter et de comparer différents algorithmes de tri pour un tableau.

On s'intéresse à l'implémentation de ces algorithmes dans un premier temps, puis nous essayerons d'analyser les performances de chacun des algorithmes implémentés afin de définir lequel est le plus adapté et le plus performant pour des tableaux de grande taille. On veillera donc à faire fonctionner ces algorithmes sur des tableaux comportant un nombre élévé d'éléments. Ici, nous analyserons des tableaux comportant jusqu'à 2 000 ou 5 000 éléments selon la méthodologie définie plus bas en sections 1-IV.1 et 2-II.

II Architecture générale

À titre d'aperçu, on utilisera la structure présentée sur la figure 1.1 pour le programme complet :

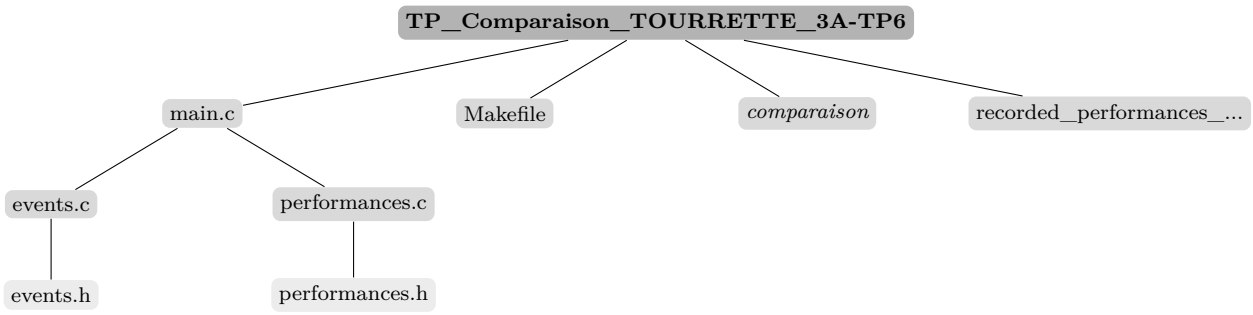

FIGURE 1.1 – Architecture du programme complet

Une fois dans le répertoire `TP_Comparaison_TOURRETTE_3A-TP6`, on utilisera la commande `make` pour produire le fichier exécutable `comparaison` et la commande `make exe` pour lancer le programme. On peut également lancer directement la compilation puis le programme en tapant cette commande.

III Création des tableaux de données

III.1 Structure de données `date`

On définit ici le type de données qui constitueront la donnée essentielle des événements créés en section 1-III.2 : la date. Une date sera ainsi composée d'un jour (entier), d'un mois (chaîne de caractères) et d'une année (entier). Ces trois éléments sont membres d'une structure de données C appelée `date`.

III.2 Structure de données `event`

On définit ici le type des données qui constitueront nos tableaux. On crée notre propre structure de données qui est de type `event`. Cet événement est de type structure de données du C. On y renseigne une date (type `date` défini en section 1-III.1) et une description de l'événement qui contiendra la mention `Event-<date>` où `date` est la date de l'événement.

Ces événements viendront remplir un tableau d'événements et seront générés de manière complètement aléatoire par le programme (les dates iront du 1er janvier 0 au 31 décembre 2018 pour une plus grande cohérence).

IV Implémentation d'algorithmes de tri

IV.1 Méthodologie d'implémentation

On implémentera les cinq algorithmes de tri classiques en langage C que sont le tri basique, le tri par sélection, le tri à bulle, le tri par insertion et enfin le tri rapide. On suivra les consignes données dans le sujet du TP concernant la méthode de tri pour chaque algorithme. Chaque algorithme fera l'objet d'une fonction particulière qui permettra d'être utilisée de manière générique pour n'importe quel tableau. On définira ainsi cinq fonctions qui porteront chacune le nom de l'algorithme de tri qu'elle implémente.

On aura également besoin de plusieurs autres fonctions.

La première est une fonction d'échange de deux éléments d'un tableau, car elle sera grandement utile à chaque méthode car le tri consiste finalement en l'échange de deux valeurs d'un tableau si elles sont mal positionnées. Cette fonction est nommée `swap_events`.

La seconde est une fonction de comparaison. En effet, nos tableaux seront de types `event` (voir section 1-III.2). On a donc besoin d'une fonction qui compare deux événements et qui renvoie une valeur en fonction de la position de chaque événement l'un par rapport à l'autre dans le temps. Cette fonction est nommée `compare_events`.

IV.2 Structure du code

Dans cette partie, on se concentrera uniquement sur la partie "events" de la structure générale du programme présentée sur la figure 1.1. Ainsi, on peut détailler que nous aurons dans le fichier :

main.c l'ensemble des tests et création des tableaux de données à trier avec les différents algorithmes ;

events.c l'ensemble des définitions des fonctions relatives aux événements et à leur tri ;

events.h l'ensemble des prototypes des fonctions relatives aux événements et à leur tri ainsi que les structures de données relatives aux événements.

Cette structure a été adoptée pour correspondre aux standards des règles de bonne programmation. On notera bien entendu l'utilisation des gardes dans `events.h` afin d'éviter toutes les inclusions en boucle conduisant à des erreurs de compilation.

CHAPITRE 2

Analyse des performances

I Structure de performances

Pour la sauvegarde des performances d'un algorithme, on va créer une structure de type `perf`. On a défini cette structure dans le fichier `performances.h` comme suit :

```
typedef struct perf{
    char algorithme[14] ;
    int comparaison, echange ;
    double temps_execution ;
} perf ;
```

On aura donc la possibilité de sauvegarde de manière propre et ordonnée nos résultats et on pourra les utiliser ensuite grâce à cette structure de données.

II Méthodologie d'implémentation

On aura recours à plusieurs fonctions pour gérer l'analyse de performances de notre programme :

reset_performances Fonction mettant à zéro les compteurs globaux du nombre de comparaison et d'échanges faits, ainsi que des temps processeur nécessaires à la mesure du Δt de chaque tri ;

save_performances Fonction permettant d'enregistrer dans un tableau de performances les résultats de l'analyse pour un algorithme donné ;

print_performances Fonction permettant d'afficher une performance donnée ;

record_performances Fonction permettant la sauvegarde des résultats d'une analyse de performance dans un fichier texte ;

moyenne Fonction de calcul de la moyenne et de l'écart-type pour plusieurs données de la performance : moyenne du nombre de comparaisons et d'échanges, moyenne et écart-type du temps d'exécution de l'algorithme.

Chacune de ces fonctions utilisera la structure de données de types `perf`.

III Résultats des analyses

Je commencerai dans un premier temps par montrer les résultats des analyses pour chaque algorithme à partir des fichiers texte que j'ai pu obtenir après les tris des tableaux générés selon les indications du sujet, et je ferai ensuite une comparaison directe afin d'établir un classement des algorithmes selon leurs résultats.

On précise également qu'on a fait deux séries de tris pour des tailles jusqu'à 5000 éléments (deux premiers résultats) et jusqu'à 2000 éléments (deux suivants).

III.1 Tri basique

On a obtenu les résultats suivants pour le tri basique :

```
######## 5000 ########
Algorithme utilisé                                    : BASIC_SORT
Nombre moyen de comparaisons effectuées               : 12502500
Nombre moyen d'échanges effectués                     : 6237774
Temps moyen d'exécution de l'algorithme               : 623557.330000
Ecart-type du temps moyen d'exécution de l'algorithme : 36962.891887

Algorithme utilisé                                    : BASIC_SORT
Nombre moyen de comparaisons effectuées               : 12502500
Nombre moyen d'échanges effectués                     : 6236574
Temps moyen d'exécution de l'algorithme               : 599018.970000
Ecart-type du temps moyen d'exécution de l'algorithme : 5872.209220

######## 2000 ########
Algorithme utilisé                                    : BASIC_SORT
Nombre moyen de comparaisons effectuées               : 2001000
Nombre moyen d'échanges effectués                     : 995610
Temps moyen d'exécution de l'algorithme               : 75338.280000
Ecart-type du temps moyen d'exécution de l'algorithme : 1488.627341

Algorithme utilisé                                    : BASIC_SORT
Nombre moyen de comparaisons effectuées               : 2001000
Nombre moyen d'échanges effectués                     : 998701
Temps moyen d'exécution de l'algorithme               : 75524.910000
Ecart-type du temps moyen d'exécution de l'algorithme : 1497.331540
```

Pour cet algorithme, on remarque qu'on effectue plus de 12 millions de comparaisons pour la série de tableaux comportant au maximum 5000 éléments et plus de 2 millions pour ceux allant jusqu'à 2000 éléments. On effectue 6 millions d'échanges et 600 000 coups d'horloge processeur pour le premier cas et presque un million d'échanges pour le second cas pour 75 000 coups d'horloge processeur. On peut déjà établir que cet algorithme aura du mal à concurrencer les autres...

III.2 Tri par sélection

On a obtenu les résultats suivants pour le tri par sélection :

```
######## 5000 ########
Algorithme utilisé                                    : SELECTION_SORT
Nombre moyen de comparaisons effectuées               : 12497500
Nombre moyen d'échanges effectués                     : 6235151
Temps moyen d'exécution de l'algorithme               : 589202.040000
Ecart-type du temps moyen d'exécution de l'algorithme : 7777.404832

Algorithme utilisé                                    : SELECTION_SORT
Nombre moyen de comparaisons effectuées               : 12497500
Nombre moyen d'échanges effectués                     : 6240946
Temps moyen d'exécution de l'algorithme               : 593990.080000
Ecart-type du temps moyen d'exécution de l'algorithme : 6123.802723
```

```
######## 2000 ########
Algorithme utilisé                                    : SELECTION_SORT
Nombre moyen de comparaisons effectuées               : 1999000
Nombre moyen d'échanges effectués                     : 1000410
Temps moyen d'exécution de l'algorithme               : 75965.530000
Ecart-type du temps moyen d'exécution de l'algorithme : 1556.555026

Algorithme utilisé                                    : SELECTION_SORT
Nombre moyen de comparaisons effectuées               : 1999000
Nombre moyen d'échanges effectués                     : 997710
Temps moyen d'exécution de l'algorithme               : 75301.750000
Ecart-type du temps moyen d'exécution de l'algorithme : 1301.391781
```

On constate que moins de comparaisons et d'échanges ont été effectués par cet algorithme même si les chiffres restent du même ordre de grandeur que ceux du cas précédent. En revanche les temps d'exécution restent du même ordre de grandeur, soit 600 000 et 75 000 coups d'horloge processeur pour chacun des cas. Cet algorithme n'est donc pas forcément plus performant que le premier.

III.3 Tri à bulle

On a obtenu les résultats suivants pour le tri à bulle :

```
######## 5000 ########
Algorithme utilisé                                    : BUBBLE_SORT
Nombre moyen de comparaisons effectuées               : 12497500
Nombre moyen d'échanges effectués                     : 6248760
Temps moyen d'exécution de l'algorithme               : 789757.210000
Ecart-type du temps moyen d'exécution de l'algorithme : 33476.784166

Algorithme utilisé                                    : BUBBLE_SORT
Nombre moyen de comparaisons effectuées               : 12497500
Nombre moyen d'échanges effectués                     : 6248521
Temps moyen d'exécution de l'algorithme               : 780036.260000
Ecart-type du temps moyen d'exécution de l'algorithme : 8736.499605

######## 2000 ########
Algorithme utilisé                                    : BUBBLE_SORT
Nombre moyen de comparaisons effectuées               : 1999000
Nombre moyen d'échanges effectués                     : 998643
Temps moyen d'exécution de l'algorithme               : 93299.980000
Ecart-type du temps moyen d'exécution de l'algorithme : 1907.236944

Algorithme utilisé                                    : BUBBLE_SORT
Nombre moyen de comparaisons effectuées               : 1999000
Nombre moyen d'échanges effectués                     : 1000479
Temps moyen d'exécution de l'algorithme               : 91918.360000
Ecart-type du temps moyen d'exécution de l'algorithme : 1548.593591
```

D'après les chiffres obtenus, le tri à bulle est encore très comparable aux deux précédents algorithmes, et on obtient même des temps de tri supérieurs avec un écart-type plus important. De plus, on remarque qu'il effectue le même nombre de comparaisons que l'algorithme de tri par sélection mais que pour arriver au bon résultat, il effectue plus d'échanges. Ainsi on peut établir que cet algorithme est moins performant que les deux précédents.

III.4 Tri par insertion

On a obtenu les résultats suivants pour le tri par insertion :

```
######## 5000 ########
Algorithme utilisé                                  : INSERTION_SORT
Nombre moyen de comparaisons effectuées             : 6256070
Nombre moyen d'échanges effectués                   : 6251071
Temps moyen d'exécution de l'algorithme             : 148653.030000
Ecart-type du temps moyen d'exécution de l'algorithme : 1960.772269

Algorithme utilisé                                  : INSERTION_SORT
Nombre moyen de comparaisons effectuées             : 6248018
Nombre moyen d'échanges effectués                   : 6243019
Temps moyen d'exécution de l'algorithme             : 147904.480000
Ecart-type du temps moyen d'exécution de l'algorithme : 1739.656055

######## 2000 ########
Algorithme utilisé                                  : INSERTION_SORT
Nombre moyen de comparaisons effectuées             : 1002144
Nombre moyen d'échanges effectués                   : 1000145
Temps moyen d'exécution de l'algorithme             : 23645.750000
Ecart-type du temps moyen d'exécution de l'algorithme : 624.868232

Algorithme utilisé                                  : INSERTION_SORT
Nombre moyen de comparaisons effectuées             : 1003478
Nombre moyen d'échanges effectués                   : 1001479
Temps moyen d'exécution de l'algorithme             : 23671.500000
Ecart-type du temps moyen d'exécution de l'algorithme : 744.850126
```

On constate que cet algorithme de tri effectue un nombre similaire de comparaisons et d'échanges, mais que toutes ces opérations sont faites dans un temps relativement restreint par rapport aux algorithmes précédents, et y compris pour des tableaux de grandes tailles, et avec très peu d'amplitude (l'écart-type est faible). Cet algorithme offre donc des performances significativement meilleures que les trois précédents.

III.5 Tri rapide

On a obtenu les résultats suivants pour le tri rapide :

```
######## 5000 ########
Algorithme utilisé                                  : QUICK_SORT
Nombre moyen de comparaisons effectuées             : 71215
Nombre moyen d'échanges effectués                   : 38667
Temps moyen d'exécution de l'algorithme             : 2729.710000
Ecart-type du temps moyen d'exécution de l'algorithme : 182.412516

Algorithme utilisé                                  : QUICK_SORT
Nombre moyen de comparaisons effectuées             : 70536
Nombre moyen d'échanges effectués                   : 38429
Temps moyen d'exécution de l'algorithme             : 2716.930000
Ecart-type du temps moyen d'exécution de l'algorithme : 160.598771
```

```
######## 2000 ########
Algorithme utilisé                                    : QUICK_SORT
Nombre moyen de comparaisons effectuées               : 24622
Nombre moyen d'échanges effectués                     : 13631
Temps moyen d'exécution de l'algorithme               : 881.920000
Ecart-type du temps moyen d'exécution de l'algorithme : 86.632059

Algorithme utilisé                                    : QUICK_SORT
Nombre moyen de comparaisons effectuées               : 24719
Nombre moyen d'échanges effectués                     : 13502
Temps moyen d'exécution de l'algorithme               : 873.870000
Ecart-type du temps moyen d'exécution de l'algorithme : 85.803223
```

On utilise maintenant l'algorithme de tri rapide (*quick sort*) qui offre des performances très élevées en s'appuyant sur le partitionnement du tableau initial. On constate que ces performances sont réelles car on obtient un nombre de comparaisons extrêmement bas pour celui-ci (70 000 comparaisons contre 6 millions soit un facteur 100) et un nombres d'échanges très bas également (40 000 contre six millions). Le temps est également très faible et est environ divisé par 100 (ou 30 dans le second cas) ce qui n'est nullement négligeable. L'écart-type est également très faible ce qui signifie que l'algorithme est relativement régulier dans ses temps de tri.

III.6 Conclusion

On peut donc établir que l'algorithme de tri rapide est le plus performant des cinq car il est plus régulier, plus rapide car il effectue moins d'opérations de comparaison ou d'échange.

Ensuite, on peut utiliser l'algorithme de tri par insertion même s'il est moins performant que le tri rapide...

En revanche, les autres algorithmes auront beaucoup de mal à être appliqués à des tableaux de plus de 5 000 éléments car ils peinent déjà à trier ceux-ci.

Il vaut mieux donc privilégier toujours l'algorithme de tri rapide qui offre des performances bien supérieures aux quatre autres algorithmes de tri que nous avons implémenté dans ce TP.