Rapport projet 2

Analyse d'algorithmes de tri

DAVID Matthias
LE BRIS Ilan
MARCHERON Bastien
PARCHEMINER Nolann

29 mars 2024

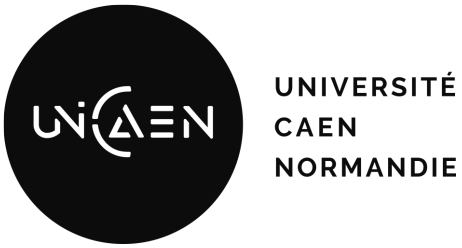


Table des matières

1 Introduction

1.1 Description générale du projet

Dans le cadre de l'unité d'enseignement *Projet 2*, nous devions réaliser un projet en lien avec l'UE de spécialité que nous avions choisi au premier semestre. Nous avions donc le choix entre 2 projets : *Analyse des algorithmes de tris* ou *Simulation du problème à N Corps*.

Après s'être mis d'accord, nous avons choisi l'*Analyse des algorithmes de tris*. Le sujet consistait à développer une application qui permettrait d'analyser et de comparer les performances de certains algorithmes de tris en fonction des données à trier.

1.2 Choix du Projet

Les algorithmes de tri sont parmi les concepts les plus fondamentaux en informatique et en algorithmique, certains existant depuis très longtemps. Ils représentent un domaine d'étude essentiel pour saisir les bases de l'efficacité des algorithmes c'est pourquoi nous nous sommes orientés vers ce choix.

De plus, les algorithmes de tri sont très diversifiés, chacun ayant ses propres caractéristiques, avantages et inconvénients en fonction de la nature des données à trier ainsi qu'au désordre de celles-ci. Ainsi, ce projet nous offre l'opportunité d'explorer et de comparer différentes techniques de tri.

1.3 Pourquoi le langage C ?

Pour notre projet, nous avons opté pour l'utilisation du langage C car c'est un langage bas niveau avec des performances optimales. Étant donné que les algorithmes de tris peuvent être coûteux en termes de temps d'exécution, en particulier ceux avec des complexités quadratiques ou plus élevées, nous avons jugé judicieux d'utiliser ce langage afin de minimiser ces temps d'exécution.

Par ailleurs, le langage C nous confère un contrôle total sur la gestion de la mémoire, ce qui se traduit par la capacité à développer des programmes à l'exécution rapide et efficiente. Cette caractéristique en fait un choix idéal

pour les projets dans lesquels les performances ont une importance majeure.

De plus, le langage C offre une souplesse remarquable dans la manipulation des données, notamment grâce aux pointeurs ainsi qu'aux structures, permettant ainsi une représentation et une manipulation efficace de données complexes.

En résumé, le choix du langage C découle principalement de notre volonté d'améliorer nos compétences en programmation dans ce langage, tout en bénéficiant d'une combinaison unique de contrôle, de performance et de flexibilité.

2 Objectif du projet

2.1 Problématique

La question posée pour ce projet explore l'efficacité des algorithmes de tri par rapport au niveau de désordre des données à trier ainsi qu'à leur nombre, en prenant en compte le nombre de comparaisons, l'accès aux données et le temps d'exécution. Cette question met en évidence l'impact du type de données sur les performances des algorithmes. En effet, un algorithme peut être considéré comme moyen voire très mauvais dans la plupart des cas, mais il pourrait se révéler être un choix judicieux pour un type de données spécifique en fonction du degré de désordre de la structure donnée en entrée. Cette aspect souligne l'importance d'une analyse approfondie des caractéristiques des données lors du choix d'un algorithme de tri pour une application donnée.

2.2 Découpage du projet

Pour mener à bien la réalisation de ce projet, nous avons identifié trois parties majeures

— Implémentation des algorithmes de tris

— Implémentation d'un générateur de tableaux aléatoire

— Lancement des algorithmes avec des paramètres variables et analyse des résultats

Et comme parties secondaires :

— Visualiser les algorithmes de tri
— Paramétrer l'aléatoire dans le générateur

3 Fonctionnalités implémentées

3.1 Description des fonctionnalités

Après avoir identifié les parties majeures du projet, nous les avons subdivisées et priorisées.

— Implémentation des algorithmes de tri
— Implémentation du générateur d'aléatoire
— Implémentation des structures
— Implémentation des logs
— Personnalisation de l'environnement de travail
— Écriture de tests
— Implémentation d'une interface pour lancer l'app

3.2 Organisation du projet

Chaque membre du groupe a choisi les fonctionnalités sur lesquelles il voulait travailler. Voici comment elles ont été réparties :

DAVID Matthias :

— Gestion de l'aléatoire
— Gestion des erreurs
— personnalisation de l'environnement de travail

LE BRIS Ilan :

— Implémentation des structures de données
— Gestion des tests

MARCHERON Bastien :

— Implémentation des algorithmes de tri
— Rapport et README

PARCHEMINER Nolann :

— Implémentation de l'interface et des Logs
— Lancement des expérimentations
— Personnalisation de l'environnement de travail

Nous nous étions certes répartis les tâches cependant, nous avons souvent été amené à modifier le code des autres pour l'adapter à nos besoins

4 Éléments techniques

4.1 Sélection des algorithmes de tri

Nous avons pour ce projet choisi d'implémenter les algorithmes de tris les plus connus et qui permettent de trier tous sortes de type de données. Ils sont listés ci-dessous :

— **Merge Sort :** Divise la liste en deux moitiés égales, trie chaque moitié récursivement, puis fusionne les deux moitiés triées.
— **Quick Sort :** Choisit un élément pivot, partitionne la liste autour de ce pivot en déplaçant les éléments plus petits à gauche et les éléments plus grands à droite. Ensuite, il trie récursivement les deux partitions.
— **Selection Sort :** Recherche le plus petit élément dans la liste et l'échange avec l'élément à la première position. Ensuite, il recherche le deuxième plus petit élément et l'échange avec l'élément à la deuxième position, et ainsi de suite jusqu'à ce que la liste soit entièrement triée.
— **Insertion Sort :** Construit une liste triée en insérant un élément à la fois de la liste non triée à la bonne position dans la liste triée.

Et nous en avons sélectionné d'autre moins connu :

— **Bogo Sort :** Place les éléments de manière aléatoire dans la liste jusqu'à ce qu'ils soient triés. C'est un algorithme inutilisable en condition réelle car sa compléxité sur des très grands tableaux est infinie
— **Bubble Sort :** Compare deux à deux les éléments adjacents de la liste et effectue une permutation si l'élément de gauche est plus grand que l'élément de droite. Se répète ce processus jusqu'à ce qu'il n'y ait plus de permutation.
— **Gnome Sort :** Permute les éléments de la liste un par un vers leur place correcte en parcourant la liste. Si un élément est plus petit que

son prédécesseur, il est échangé avec son prédécesseur, puis revient en arrière pour vérifier le nouvel élément avec son prédécesseur.

— **Heap Sort :** Transforme la liste en un tas, puis extrait successivement l'élément maximum du tas et reconstitue le tas.

— **Odd-Even Sort :** Trie les éléments en les comparant et en les échangeant alternativement, en traitant d'abord les éléments d'index pairs puis les éléments d'index impairs.

— **Shaker Sort :** Similaire à Bubble Sort, mais va dans les deux sens pour trier les plus grands éléments d'un côté et les plus petits de l'autre.

— **Shell Sort :** Divise la liste en sous-listes plus petites, puis trie chaque sous-liste avec un tri par insertion amélioré. Ensuite, il fusionne progressivement les sous-listes jusqu'à ce que la liste entière soit triée.

— **Tim Sort :** Combinaison de Merge Sort et Insertion Sort. Il divise la liste en blocs, trie chaque bloc avec Insertion Sort, puis fusionne les blocs triés avec Merge Sort. Cela le rend efficace pour trier des données partiellement triées.

4.2 Descriptions des structures de données

Pour parvenir à gérer les données de manière efficace, nous avons du implémenter nos propres types de données

4.2.1 Structures de données pour les expérimentations

Nous avons implémenté deux structures principales pour lancer une expérimentation qui sont *entry* et *result*.

La structure *entry* est passée en paramètre aux algorithmes de tri et contient les tableau de données ainsi que toutes les fonctions relatives à celles-ci.

— *array :* pointeur sur un tableau d'objets pré-alloués.

— *cmp_fn :* pointeur sur une fonction de comparaisons entre deux objets de *array*.

— *copy_fn :* pointeur sur une fonction de copie sur un objet de *array*.

— *free_fn :* pointeur sur une fonction pour libérer l'espace mémoire d'un objet de *array*.

— *size :* la taille de *array*.

La structure *result* permet de représenter le résultat après le tri et permet ainsi de pouvoir exploiter les donnée numériquement. Elle est structurée comme suit :

— array_in : pointeur sur un tableau d'objets pré-alloués.
— *array_out :* pointeur sur un tableau d'objets pré-alloués triés.
— *array_size :* taille de *array_in* et *array_out*.
— *nb_cmp :* nombre de fois où l'algorithme appelle la fonction de comparaison.
— *nb_access :* nombre de fois où l'algorithme a accédé à *array_out*.
— *misplaced_value :* nombre de valeurs mal placées dans *array_in* par rapport à *array_out*.
— *avg_offset :* moyenne des décalages entre la position d'une valeur dans *array_in* et sa position dans *array_out*.
— *process_time :* temps nécessaire à l'algorithme pour trier *array_in*.
— *algo_name :* nom de l'algorithme utilisé.
— *free_fn :* pointeur sur une fonction pour libérer l'espace mémoire d'un objet de *array_in* et de *array_out*.

De plus, nous avons implémenté une structure *experiment* qui représente les résultats des expériences sur un algorithme de tri. Une expérience c'est lorsque l'on exécuté l'algorithme n fois avec différents paramètres de randomization et que l'on garde les valeurs minimales, maximales ainsi que les moyennes obtenues.

4.2.2 Structures de données pour les types et les tests

Les algorithmes de tri, pouvant trier différents types de données, nous avions besoin (dû à la manière dont nous avons construit le projet) de déclarer des fonctions de comparaison, copie et libération mémoire des types primitifs du c.

Pour les tests nous avons implémenté une structure qui permet de créer des étudiants avec des attributs tels qu'un nom, un âge et un genre..

Cette structure de test étant composée de différents types, elle offre la possibilité d'utiliser les algorithmes pour trier par nom, par genre ou encore par âge.

4.3 Descriptions des données

5 Architecture du projet

Au début du projet nous avions convenu des répertoires suivant :

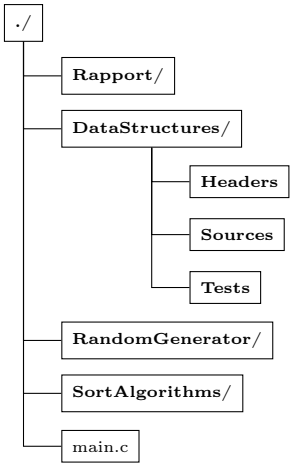
— `DataStructures` : nécessaire aux structures implémentées.

— `RandomGenerator` : nécessaire à la génération de type et de tableau aléatoire.

— `SortAlgorithms` : nécessaire aux algorithmes de tri.
Chacun répertoire contient 3 répertoires : `Headers`, `Sources` et `Tests`.

— `Rapport` : contenant les fichiers de ce rapport

Cela nous donne l'architecture suivante :



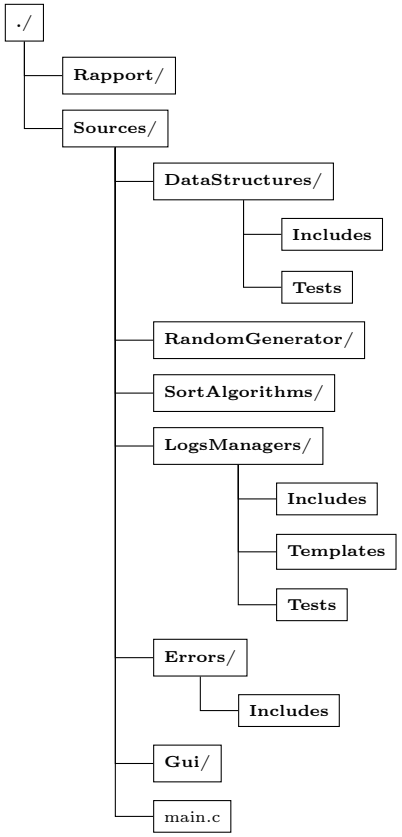
Cependant pour le bon fonctionnement de *CMAKE* nous avons du complètement la revoir. C'est pour cela que nous avons décidé de mettre tous les répertoires contenant des fichiers sources dans un répertoire `Sources` à la racine du projet, et de placer tous les fichiers sources à la racine de leurs répertoires respectif. De plus, à l'exécution de l'application un répertoire `Logs` est créé à la racine. C'est ici que les fichiers logs sont créés. Et dans `Sources`, nous avons ajouter les répertoires :

— `LogsManager` : nécessaire à la gestion des logs.
Ce répertoire contient un répertoire de plus par rapport aux précédant, qui contient les patterns des logs pour les expérimentations.

— `Gui` : nécessaire à l'interface graphique.

— `Errors` : nécessaire à la gestion des erreurs.
Ces 2 repertoires, en plus de leurs fichiers sources, ne contiennent qu'un répertoire `Includes`.

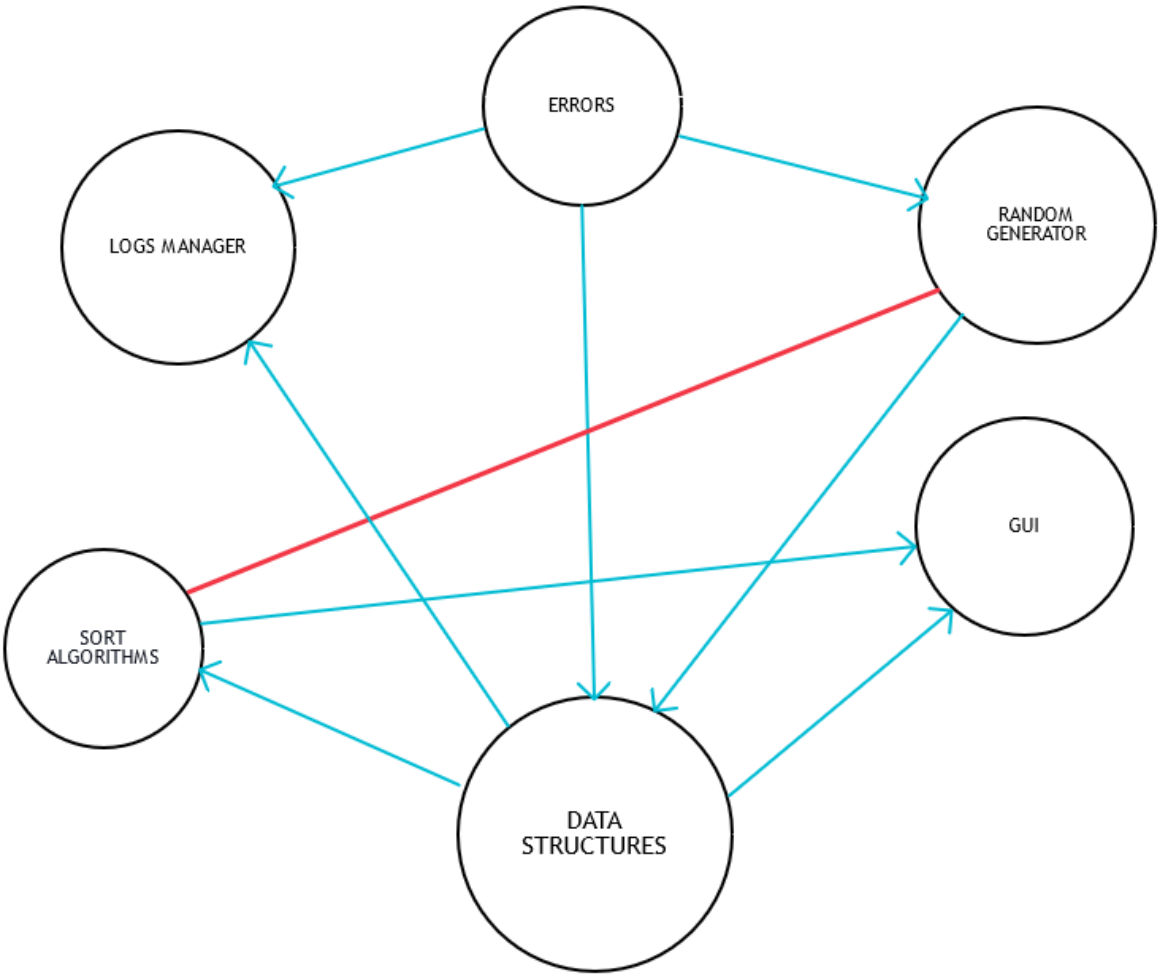
et renommer les répertoires `Headers` en `Includes`.
Ce qui nous donne l'architecture suivant :



5.1 Interaction entre les différentes parties

Chaque composante est destinée à interagir avec les autres. Le schéma ci-dessous illustre cette dynamique, il y est représenté les interconnexions entre les différentes parties du projet.

Les liaisons rouge représentent un lien réciproque entre 2 parties du projet. L'endroit où pointent les flèches bleues représente leur présence dans le programme d'une des parties.



5.2 Description des package non standards utilisés

Pour ce projet, le seul package exotique nous avons utilisé est *ncurses*. Il nous a permis de créer l'interface. C'est un package portable sur toutes les distributions car il est disponible dans les librairies standards.

6 Expérimentations

6.1 Cas d'utilisation

Notre application est utile lorsque l'on souhaite trier un type de données spécifique (dans notre cas *student*), car il renvoie des données claires sur les performances et le comportement de chaque algorithme de tri. Cependant, pour pouvoir effectuer des tests personnalisés, il sera nécéssaire d'implémenter la structure de données à trier ansi que tous ses algorithmes de gestion (*cmp*,*copy*,*free*.

6.2 Analyse des résultats

Grâce à la trace d'exécution, nous avons pu classer les algorithmes du meilleur au moins performant, en attribuant le rang 1 au meilleur et le rang 12 au moins performant. Nous avons décidé de placer Bogo Sort en dernière position d'office, car l'exécution de cet algorithme sur un tableau de taille supérieure à 10 prend un temps excessivement long pour être résolu.

Pour le lancement des algorithmes, nous utilisons des tableaux de taille 1000 que nous relançons 100 fois afin d'obtenir une moyenne fiable de chaque algorithme.

Voici les résultats obtenus : Temps d'exécution (en secondes)

— HEAP SORT 0.000001
— MERGE SORT 0.000001
— QUICK SORT ITER 0.000001
— QUICK SORT REC 0.000001
— SHELL SORT 0.000001
— TIM SORT 0.000001
— GNOME SORT 0.000027
— ODD EVEN SORT 0.000027
— INSERTION SORT 0.000011
— SELECTION SORT 0.000023
— BUBBLE SORT 0.000030

— SHAKER SORT 0.000030

Accès au tableau

— MERGE SORT 5875
— TIM SORT 6642
— HEAP SORT 7040
— SHELL SORT 9807
— INSERTION SORT 63556
— ODD EVEN SORT 122459
— GNOME SORT 125544
— SELECTION SORT 499500
— QUICK SORT ITER 378298
— QUICK SORT REC 378262
— SHAKER SORT 128651
— BUBBLE SORT 499500

Comparaisons

— SHELL SORT 21800
— HEAP SORT 26162
— TIM SORT 39024
— MERGE SORT 51655
— INSERTION SORT 189673
— ODD EVEN SORT 494180
— GNOME SORT 500182
— QUICK SORT REC 388388
— QUICK SORT ITER 388414
— SHAKER SORT 506569
— SELECTION SORT 1002996
— BUBBLE SORT 1248289

Pour minimiser le nombre d'accès au tableau, les choix les plus appropriés sont le Merge Sort, le Tim Sort et le Heap Sort. En revanche, pour réduire le nombre de comparaisons, nous recommandons l'utilisation du Shell Sort, du Heap Sort et du Tim Sort.

En général, le Merge Sort se distingue comme l'algorithme de tri le plus efficace car il offre des performances supérieures en termes de temps d'exécution et d'accessibilité au tableau par rapport aux autres algorithmes présentés. D'autre part, le Shell Sort se démarque particulièrement dans sa capacité à réduire le nombre de comparaisons pour trier les éléments, ce qui en fait bon choix lorsque la réduction de la complexité en termes de comparaisons est importante. Bien que son temps d'exécution puisse ne pas être aussi rapide que celui du Merge Sort dans certains cas, sa performance en termes de comparaisons le place comme une bonne option si l'on souhait limiter le nombre de comparaison.

7 Conclusion

7.1 Récapitulatif de la problématique et de la réalisation

Pour conclure, notre application répond à la problématique. Nous avons implémenté des algorithmes de tris que l'on peut lancer avec des paramètres de randomization variables. Nous pouvons par la suite utiliser les données retournées afin de les traiter et de comparer de manière plus approfondie les points forts et les points faibles de chaque algorithme. On a de plus remarqué que les algorithmes sont certes efficaces pour certains, mais ils restent extrêment dépendants des fonctions associées aux données qu'ils doivent trier. On peut donc par déduction et après plusieurs tests, dire que certains algorithmes qui paraissent inefficaces dans certaines situations le sont dans d'autres, et que pour maximiser la vitesse de tri d'un ensemble de données, il faut choisir les algorithmes de tris les plus adaptés à ces données et ensuite leur répartir le travail. Afin que chaque donnée soit traitée le plus efficacement possible.

7.2 Propositions d'améliorations

De manière générale nous sommes satisfaits de notre travail. Cependant, si nous devons garder un aspect objectif sur le travail fournit, nous pouvons

facilement lister plusieurs d'améliorations :

- Ajouter d'autres algorithmes de tri pour élargir les choix disponibles.
- Optimiser les algorithmes existants pour améliorer les performances.
- Ajouter des fonctionnalités de personnalisation pour permettre aux utilisateurs de configurer les paramètres des algorithmes.
- Optimiser la gestion de la mémoire pour réduire les fuites et améliorer l'efficacité.
- Proposer un paramétrage plus poussée de la randomization
- Rendre le projet plus dynamique (réduire au maximum les dépendances inter-packages)