S1.02 Comparaisons d'approches algorithmiques

Marielle Vallée Gaspard Pons TD A

Question Préliminaire :

Temp de calcul pour 200 000 données :
1er essai : 105 ms
2eme essai : 35 ms
3eme essai : 73 ms
4eme essai : 29 ms
5eme essai : 40 ms
6eme essai : 87 ms
7eme essai : 32 ms
8eme essai : 70 ms
9eme essai : 76 ms
10eme essai : 28 ms

moyenne : 57,5 millisecondes

Temp de calcul pour 10 000 000 données :
1er essai : 1230 ms
2eme essai : 1213 ms
3eme essai : 1247 ms
4eme essai : 1163 ms
5eme essai : 1191 ms
6eme essai : 1162 ms
7eme essai : 1242 ms
8eme essai : 1285 ms
9eme essai : 1202 ms
10eme essai : 1180 ms

moyenne : 1211,5 ms = 1 seconde 21

Question 1 :

Tri par insertion

Temp de calcul pour 200 000 données :
1er essai : 4000 ms
2eme essai : 3932 ms
3eme essai : 3964 ms
4eme essai : 3907 ms
5eme essai : 3988 ms
6eme essai : 4017 ms
7eme essai : 3986 ms
8eme essai : 4004 ms
9eme essai : 3939 ms
10eme essai : 4004 ms

moyenne : 3 974,1 ms = 3 secondes 97
moyenne théorique pour 10 000 000 : 3 974,1 x 50^2 = 9 935 250 ms = 2 heures

Tri Cocktail

Temp de calcul pour 200 000 données :
1er essai : 55416 ms
2eme essai : 59359 ms
3eme essai : 57158 ms
4eme essai : 53302 ms
5eme essai : 53547 ms
6eme essai : 54222 ms
7eme essai : 58214 ms
8eme essai : 49327 ms
9eme essai : 49114 ms
10eme essai : 48828 ms

moyenne : 53 848,7 ms = 53 secondes
moyenne théorique pour 10 000 000 : 134 621 750 ms = 1 jour et demi

Conclusion : plus le tableau est grand, moins ces méthodes de tri sont adaptées. Elles ne sont donc pas un choix judicieux.

Question 2 : Tri à peigne

Temp de calcul pour 200 000 données :
1er essai : 32 ms
2eme essai : 34 ms
3eme essai : 35 ms
4eme essai : 32 ms
5eme essai : 34 ms
6eme essai : 32 ms
7eme essai : 31 ms
8eme essai : 35 ms
9eme essai : 34 ms
10eme essai : 31 ms

moyenne : 33 millisecondes
moyenne théorique pour 10 000 000 : 82 500 ms = 82.5 secondes

Conclusion : on observe que le tri à peigne est bien plus rapide que le tri cocktail. Ces deux versions sont supérieures au tri à bulles, le tri à peigne est cependant une version bien plus efficace que le tri cocktail.

Question 3 : Tri comptage

Temp de calcul pour 200 000 données comprise entre 0 et 1000 :
1er essai : 4 ms
2eme essai : 3 ms
3eme essai : 5 ms
4eme essai : 3 ms
5eme essai : 3 ms
6eme essai : 2 ms
7eme essai : 6 ms
8eme essai : 6 ms
9eme essai : 2 ms
10eme essai : 6 ms

moyenne : 4 millisecondes
moyenne théorique pour 10 000 000 : 10 000 ms = 10 secondes

Conclusion : on observe que cette méthode de tri est très rapide et efficace, d'autant plus qu'elle fonctionne avec des données à plusieurs chiffres. Cependant elle ne marche qu'avec des données supérieures à 0.

Question 5 : Tri à peigne sur une autre machine

Temp de calcul pour 200 000 données : Java vs C
1er essai : 40 / 53 ms
2eme essai : 70 / 178 ms
3eme essai : 61 / 170 ms
4eme essai : 51 / 84 ms
5eme essai : 73 / 104 ms
6eme essai : 100 / 50 ms
7eme essai : 61 / 268 ms
8eme essai : 82 / 111 ms
9eme essai : 80 / 77 ms
10eme essai : 35 / 78 ms

moyenne Java : 65.3 ms
moyenne C : 117.3 ms

Conclusion : on observe que l'algorithme de tri est deux fois plus rapide en moyenne codé en Java que en C sur ces tests.

Description des algorithmes

leur fonctionnement, leurs caractéristiques et leurs performances

Explication : les performances d'un algorithme

Celles-ci dépendent du temps d'exécution de l'algorithme et de l'espace (i.e des ressources matérielles utilisées comme la mémoire ou le microprocesseur). Cependant pour calculer les performances d'un algorithme on quantifie le temps d'exécution ce qui permet d'obtenir la complexité en temps. Etant donné que la capacité d'un ordinateur influence le temps d'exécution, on souhaite l'ignorer. Pour cela on utilise le nombre d'opérations effectuées par le tri dans le pire des cas, un cas moyen et le meilleur des cas. Pour l'exprimer on utilise 'O' qui représente le temps par itération de la machine et 'n' le nombre d'itérations dans les boucles. Ainsi O(n) représente le temps d'exécution de n itérations. C'est pourquoi lorsqu'on a des boucles imbriquées, l'algorithme a une temps d'exécution de $O(n^2)$.

Explication : la notion de stabilité

On dit qu'un algorithme de tri est *stable* s'il ne modifie pas l'ordre initial des valeurs identiques. Par exemple, si un tonneau A et B ont le même poids et qu'on les trie du plus léger au plus lourd, si le tonneau B était situé avant le tonneau A dans l'espace de stockage, alors après le tri, le tonneau B doit être avant le tonneau A pour que le tri soit considéré stable. L'intérêt d'un tri stable est qu'on peut appliquer ce tri successivement à un même tableau, avec des critères différents.

Les algorithmes implantés

Le tri par insertion

Le tri par insertion consiste à comparer une valeur du tableau (qu'on stocke en mémoire) avec celles précédentes et à positionner celle-ci en fonction des valeurs précédentes dans le tableau. On décale les éléments précédents tant qu'ils sont plus grands que la valeur stockée jusqu'à ce qu'on tombe sur un élément plus petit ou le premier indice du tableau.

Le tri par insertion est stable, il stocke 2 variables d'entiers en mémoire et utilise 2 boucles (un while et un for). C'est un algorithme simple qui est plus efficace si le tableau possède déjà un certain ordre étant donné qu'il n'aura que quelques valeurs à échanger. Autrement dit, il rentrera moins souvent dans la boucle while et n'aura dans ce cas que le temps de calcul d'un passage de boucle for et 2 modifications de variables à faire en mémoire.

En théorie, dans le pire des cas : son temps d'exécution est de $O(n^2)$. Dans le meilleur des cas : O(n). En moyenne : $O(n^2)$. Dans nos cas pratiques (10 fois des tableaux générés aléatoirement) le temps moyen pour trier un tableau de 200 000 valeurs était d'environ 3 secondes 97.

Cependant si le tri par insertion est bien moins efficace que d'autres algorithmes de tri pour les grands tableaux, il est plus efficace sur des tableaux de petite taille ou déjà pré-triés. C'est pourquoi il est parfait pour finir le tri d'autres méthodes de tri qui perdent en efficacité quand le tableau est quasi trié.

Avant d'aller plus loin : le tri à bulles.

Celui-ci consiste à comparer les valeurs de 2 indices côte à côte. Ainsi il parcourt le tableau depuis l'indice 0 et compare les deux valeurs des cases adjacentes au fur et à mesure et échange l'une avec l'autre si la condition est remplie. Il passe autant de fois que nécessaire dans une boucle qui parcourt l'intégralité du tableau pour vérifier s'il reste des échanges à faire. S'il n'y a plus d'échange à faire, il parcourra une dernière fois pour pouvoir sortir de la boucle.

Le tri par cocktail (ou tri shaker)

Le tri par cocktail est une variante du tri à bulles. Dans son premier tour, il va prendre le premier élément et le déplacer vers la droite jusqu'à ce qu'il trouve un élément plus grand grâce à la méthode du tri à bulles. Une fois arrivé à une extrémité du tableau, il parcourt le tableau en sens inverse et effectue la même chose mais en décalant vers la gauche l'élément tant qu'il est plus petit . Ainsi il ne revient pas au tout début à chaque fois comme le tri à bulles qui ne fait toujours que parcourir dans un sens.

Le tri par cocktail est un tri stable. Il stocke 2 variables (un booléen et un entier) et utilise 3 boucles (2 for et un while). Le while utilise le booléen pour permettre de savoir quand arrêter de parcourir le tableau i.e quand il n'y a plus d'échange qui a été réalisé lors des 2 tours de boucle for. Le premier for permet de parcourir de gauche à droite et le second de droite à gauche. Dans chaque for on compare et échange les valeurs si besoin est et on précise s'il y a eu un échange (false or true).

En théorie, dans le pire des cas : son temps d'exécution est $O(n^2)$. Dans le meilleur des cas : O(n). En moyenne : $O(n^2)$. Dans nos cas pratiques (10 fois des tableaux générés aléatoirement) le temps moyen pour trier un tableau de 200 000 valeurs était d'environ 53 secondes.

Cependant, bien que le tri par cocktail permette de gagner en efficacité par rapport au tri à bulles, il ne gagne pas beaucoup de temps d'exécution par rapport à celui-ci. Pourtant dans un tableau trié en ordre décroissant, le tri par cocktail n'aura besoin de passer qu'une fois dans la boucle contrairement au tri à bulles. Il gagnerait donc théoriquement beaucoup plus de temps d'exécution dans ce cas là que le tri à bulles.

Le tri à peigne (ou comb sort)

Le tri à peigne est un algorithme de tri par comparaison qui améliore les performances du tri à bulles. Il consiste à comparer en premier lieu des éléments lointains puis de progressivement réduire cet intervalle jusqu'à comparer deux éléments adjacents comme dans un tri à bulles. Le facteur de réduction idéal pour l'espace entre les éléments est de 1.3.

Le tri à peigne est un algorithme de tri non stable. On utilise 4 variables (3 entiers et un booléen) et 2 boucles while. On initialise en premier lieu l'intervalle à la taille du tableau qu'on va ensuite modifier à chaque passage de boucle au début pour définir et réduire l'espace entre les 2 indices qu'on compare. Si l'espace est trop petit (inférieur à 1) on le met à 1 pour retourner sur du tri à bulles classique. On initialise un entier à 0 pour avoir un indice qu'on va progressivement augmenter de 1 pour parcourir le tableau tant que l'indice avec lequel il est comparé (via l'intervalle défini précédemment) existe. On continue à faire les comparaisons et réduire l'intervalle entre les indices tant qu'on a pas fait un tour de tri à bulles (donc d'intervalle 1) où il n'y a pas eu d'échanges (comme lors du tri à bulles classique).

En théorie, dans le pire des cas : son temps d'exécution est $O(n^2)$. Dans le meilleur des cas : O(n log n). Dans nos cas pratiques (10 fois des tableaux générés aléatoirement) le temps moyen pour trier un tableau de 200 000 valeurs était d'environ 33 millisecondes.

Le tri à peigne est plus rapide que les algorithmes de tri vus précédemment dans le meilleur des cas et prend le même temps qu'eux dans le pire des cas. De plus, bien qu'il utilise plus de variables il ne les modifie finalement que quand il y a un échange ce qui fait qu'il effectue moins de modifications des variables stockées en mémoire.

Différence de performances entre java et le C pour le tri à peigne

Sur la même machine, lors du premier test le programme en C prenait 80 ms pour trier un tableau de 200 000 valeurs et le programme en Java 40ms soit deux fois plus de temps en C. Cela a été confirmé ensuite par une série de tests. Nous avons été étonnés du résultat car nous pensions que le C était plus proche de la machine que le Java donc plus performant. Nous avons remis en doute nos résultats. Cependant une recherche internet nous a prouvé que nous ne nous étions pas trompés. Le langage C est en effet normalement plus rapide : la lenteur vient des paramètres du compilateur du langage C et non du langage en lui-même (le fait de poser des flags i.e des 'points de contrôle' le ralentit énormément). Cependant si on optimise GCC on se rend compte qu'un programme va plus vite en C qu'en Java.

Si ça vous intéresse, nous vous invitons à prendre connaissance de cet article :

https://medium.com/swlh/a-performance-comparison-between-c-java-and-python-df3890545f6d

Le tri comptage

Le tri comptage que nous avons trouvé est particulier car il ne fonctionne que pour les tableaux d'entiers positifs. Il consiste à compter le nombre d'occurrences de chaque nombre présent. On cherche donc le plus grand nombre. On crée un tableau de même longueur que la valeur de celui-ci + 1 (car on aura besoin de l'indice de valeur plus grand nombre et de l'indice 0). Dans ce tableau de taille plus grand nombre, on indiquera pour chaque indice, le nombre d'occurrences de la valeur de l'indice. Une fois cela fait on pourra re-remplir le tableau pris en paramètres en suivant les indications du tableau de nombre d'occurrences.

Le tri comptage est un algorithme de tri non stable. On utilise 2 variables d'entiers et un tableau ainsi que 4 boucles for. En premier lieu on cherche la plus grande valeur du tableau pour pouvoir créer le tableau d'occurrences avec une première boucle for. On parcourt le tableau pris en paramètres et pour chaque valeur d'une case de celui-ci on ajoute +1 à l'indice du tableau d'occurrences correspondant à cette valeur (deuxième boucle for). Enfin, on parcourt le tableau d'occurrences et on initialise une variable à 0 avant la boucle for qui va permettre de re-remplir au fur et à mesure le tableau pris en paramètres en fonction du nombre d'occurrences. Ainsi on parcourt le tableau d'occurrences puis chaque case de ce tableau si sa valeur n'est pas nulle. Tant qu'il y a une valeur, il y a une occurrence de cette valeur (définie par l'indice) donc on remplace la case du tableau pris en paramètres par cette valeur et on augmente de 1 l'indice pour pouvoir parcourir à la prochaine valeur à entrer la prochaine case.

En théorie, le tri comptage a un temps d'exécution dans le pire des cas de O(n+k). Dans nos cas pratiques avec le tri comptage que nous avons programmé le temps moyen pour trier un tableau de 200 000 valeurs de 0 à 1000 était d'environ 4 ms. Par contre avec le même nombre de valeurs mais un écart de 99999999 nous avons obtenu environ 250 ms (lancé une seule fois).

Le tri comptage est très gourmand en mémoire car il demande pour les tableaux avec des grandes valeurs de créer un nouveau tableau de taille plus grande valeur+1. On ajoute à cela la contrainte que les entiers ont besoin d'être positifs et on se rend compte que le tri comptage est rapide mais uniquement dans des conditions bien précises (entiers positifs). On ne peut pas comparer sa performance à celle des autres.