Theoretical and practical work

Implémentation and analysis of compexities of sorting algorithms

Objectives of the work

- Initiating students to sorting methods and the notion to algorithms' complexity
- Evaluation and comparison of theoretical and experimental complexities of different sorting algorithms

Sending Work

- The work must be done **in pairs** (binôme)
- The report must be in **pdf** format.The source code must be in **wordPad**
- Source code + report must be sent to the address recupspace@gmail.com, in **two separate files** specifying in the object of the mail « Devoir-ALGO-ING2C – students names)
- Deadline to send the works: Saturday **28/12/2024**
- Dispalys on the screen must be well aligned and readable.
- **Cases of copying will be penalized with Zero.**

Problematic

A large number of real problems resort to searching for elements in a vector or any other data structure (matrix, linked list, tree, etc.). This search becomes almost cumbersome when it is carried out in a large (several thousand of elements) and disordered structure, hence the need to sort these structures in order to facilitate the search. In this work, it involves implementing some vector sorting methods (of equal or different complexities). And then generalize them to matrices and linked lists.

Description of the methods

Selection Sorting

this technique consists of sequentially traversing the vector to be sorted. At iteration **i**, the smallest value (respectively the largest if it is a sort in descending order) of the table is permuted with the value located in the box with index i. At the end of an iteration, one element of the vector is well placed (i.e., it is in its correct position).

Bubble Soring

In this method, the smallest elements (respectively the biggest in the descending case) of the array "rise" (like bubbles) towards the beginning of the array to reach their final position. Once all the elements have been reassembled, the table is sorted. This technique is done by successive permutations of consecutive elements t[i] and t[i+1] after comparing them.

Insertion Soring

This technique consists of considering at each step, an element t[i] of the vector and searching for its insertion position, by shifting the other elements (found before the element of index i) so as to "free" the insertion position of the element. At each step, part of the array (until position i) is already ordered and a new value is inserted in the appropriate place.

Quick Sort

It is a sorting algorithm based on the **"divide and conquer"** principle. It is generally used on arrays, but can also be adapted to linked lists. The method consists of placing an element of the array (called **pivot**) in its final place, permuting all the elements such that all those which are lower than the pivot are to the left of the pivot and all those which are higher (to the pivot) are to its right.
This operation is called partitioning. For each of the sub-arrays, we define a **new pivot** and we repeat the partitioning operation. This process is repeated recursively until all elements are sorted.

Merging Sort

The algorithm is naturally described recursively. It is also based on the principle of **"divide and conquer"**. It consists of dividing the array (in the middle) into two sub-arrays at several levels (until having sub-arrays made up of 1 element, each) and then merging the sub-arrays sorted at several levels until reconstructing the initial array fully sorted. The general idea is based on merging sorted subarrays.

Comb Sort

The basic idea of "Comb sort" is to compare an element with another more distant element spaced from it by a certain interval, (not necessarily two consecutive elements like bubble sort). Initially, this interval is relatively large, then we gradually reduce it with each pass of the algorithm until we arrive at an interval of size 1. Bubble sort can be considered as a variant of "comb sort" in which the interval is always 1.

Requested Work

Part I

Implement (in C language) the sorting methods described above for a **vector of integers**.

Part II

Adapt two methods for sorting matrices and linked lists, the program must answer the following questions:

- Consider a **matrix of characters** where each line represents String of characters (this gives an array of character strings) and sort this matrix in alphabetical order of words. Consider two sorting algorithms to choose from: ***bubble sort (or insertion sort)*** and ***merge sort (or quick sort)***.
- Consider **a linked list of words** and apply the ***insertion sort*** algorithm and the ***Bubble sort*** algorithm on the list.

Requirements

1. Each sorting program (or algorithm) must display the state of the vector (matrix or list) after each iteration in order to see the sorting progress. (Write a display function to call at each iteration).
2. Each sorting program (or algorithm) must display **at the end** the number of comparisons made (nbComp) and the number of permutations made (nbPerm). These two numbers are calculated in each algorithm (via instructions added in the algorithm) and vary from one execution to another.
3. A theoretical analysis of the different programmed methods is required, this will involve calculating the **time complexity of the different algorithms, using Landau notations**, considering the best case, the average case and the worst case.
4. An analysis of experimental complexity is also requested (see details below)

Indications

The main interface of the program must be in the form of a multiple choice menu to answer any of the user's questions.

1. Vector Sort	2. Matrix Sort	3. Linked List Sort

When the user selects « Vector Sort », another menu must be displayed to choice one sorting method. Idem if the user selects « Matrix Sort » or « Linked List Sort ».

- The program must be written as a set of functions (C language).
- Names of functions and variables must be significant
- Comments must be incorporated in the code for more clarity and understanding.
- Intermediary results must be displayed in order to show the progression of the sorting method (with the iteration number).

Report : Average of 10 pages

A report must be written following this plan:

1. **Introduction**
2. **Objectives of the work**
3. **Description of the different algorithms** (with comments), for each sorting method, while specifying its theoretical complexity.
4. **Experimental evaluation**

   Choice three (03) methods with different complexities and fill in a table for experimental evaluation to report time complexities where considering vectors of integers with different sizes **10, 20, 50, 100, 200, 500 et 1000.**

Size n	Nb permutations	Nb comparisons	Nb iterations	**Execution time (Tps)**
10				
20				
…				
1000				

PS : Data for test (vectors of integer) can be filled in by program to avoid entering values in keyboard

The execution time (Tps) must be evaluated by the program (specific instruction)

5. **Conclusion**

Appendix: Screenshots illustrating some executions must be included in appendix

---

Implémentation et analyse de complexité des algorithmes de Tri

Problématique

Un grand nombre de problèmes réels ont recours à la recherche d'éléments dans un vecteur ou toute autre structure de données (matrice, liste chainée, arbre, ...). Cette recherche devient quasiment lourde lorsqu'elle s'effectue dans une structure volumineuse (plusieurs milliers d'éléments) et *désordonnée*, d'où le besoin de *trier ces structures* afin de faciliter la recherche. Dans ce travail, il s'agit d'implémenter quelques méthodes de tri de vecteurs (de complexités égales ou différentes). Et de les généraliser par la suite, aux matrices et aux listes chainées.

Description des méthodes

Tri par sélection : cette technique consiste à parcourir séquentiellement le vecteur à trier. A l'itération i, la plus petite valeur (respectivement la plus grande s'il s'agit d'un tri dans l'ordre décroissant) du tableau est permutée avec la valeur située dans la case d'indice i. Au bout d'une itération, un élément du vecteur est bien placé (i.e, il est à sa bonne position).

Tri par bulle : dans cette méthode, les petits éléments (respectivement les plus grands dans le cas décroissant) du tableau « remontent » (comme des bulles) vers le début du tableau pour atteindre leur position finale. Une fois tous les éléments remontés, le tableau est trié. Cette technique se fait par permutations successives d'éléments consécutifs t[i] et t[i+1] après les avoir comparé.

Tri par insertion : cette technique consiste à considérer à chaque étape, un élément t[i] du vecteur et à chercher sa position d'insertion, en décalant les autres éléments (se trouvant avant l'élément d'indice i) de façon à « libérer » la position d'insertion de l'élément. À chaque étape, une partie du tableau est déjà ordonnée et une nouvelle valeur est insérée à l'endroit approprié.

Tri rapide ou tri pivot : (en anglais quicksort)

C'est un algorithme de tri fondé sur le principe « diviser pour régner ». Il est généralement utilisé sur des tableaux, mais peut aussi être adapté aux listes chainées. La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à gauche du pivot et que tous ceux qui sont supérieurs (au pivot) soient à sa droite.

Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

Tri fusion

L'algorithme est naturellement décrit de façon récursive. Il est aussi fondé sur le principe « diviser

pour régner ». Il consiste à diviser le tableau (au milieu) en deux sous-tableaux à plusieurs niveaux (jusqu'à avoir des sous-tableaux constitués de 1 élément, chacun) et fusionner ensuite les sous-tableaux triés à plusieurs niveaux jusqu'à reconstruire le tableau initial entièrement trié. L'idée générale repose sur la fusion de sous-tableaux triés.

Tri peigne (comb sort)

L'idée de base du « tri peigne » est de comparer un élément avec un autre élément plus lointain espacé de celui-ci d'un certain intervalle, (pas nécessairement deux éléments consécutifs comme le tri à bulles). Au départ, cet intervalle est relativement grand, puis on le réduit progressivement à chaque passe de l'algorithme jusqu'à arriver à un intervalle de taille 1. Le tri à bulles peut être considéré comme une variante du « tri peigne » sort dans lequel l'intervalle est toujours de 1.

Travail demandé

Partie I

Implémenter (en langage C), les méthodes de tri su-décrites pour un vecteur d'entiers.

Partie II

Généraliser deux méthodes au tri de matrices et de listes chaînées, le programme doit répondre aux questions suivantes :

- Considérer une matrice de caractères où chaque ligne représente une chaîne de caractères (il s'agit d'un tableau de chaines de caractères) et trier cette matrice dans l'ordre alphabétique des mots. Considérer deux algorithmes de tri au choix: *tri par bulles (ou par insertion)* ***et*** *tri fusion (ou tri rapide)*.
- Considérer une liste chainée de mots et appliquer l'algorithme du *tri par insertion* et l'algorithme *du tri à Bulles* sur la liste.

Exigences

2. Chaque programme (ou algorithme) de tri doit afficher l'état du vecteur (matrice ou liste) après chaque itération afin de voir la progression du tri. (Ecrire une fonction d'affichage).
3. Chaque programme de tri (ou algorithme) doit afficher à la fin le nombre de comparaisons effectuées (**nbComp**) et le nombre de permutations effectuées (**nbPerm**). Ces deux nombres sont calculés dans chaque algorithme (via des instructions ajoutées dans l'algorithme) et varient d'une exécution à l'autre.

4. Une analyse théorique des différentes méthodes programmées est demandée, il s'agira de calculer la **complexité temporelle** des différents algorithmes, en utilisant les notations de Landau, en considérant le meilleur des cas, le cas moyen et le pire des cas.
5. Une analyse de complexité expérimentale est aussi demandée (voir détails ci-dessous)

Indications

- L'interface principale du programme doit se présenter sous forme d'un menu à choix multiples pour répondre à l'une ou l'autre des questions de l'utilisateur.

1. Tri d'un vecteur	**2.** Tri d'une matrice	**3.** Tri d'une liste chainée

Une fois que l'utilisateur choisit le « Tri d'un vecteur », un autre menu doit être affiché pour sélectionner l'une des méthodes de tri. Idem si l'utilisateur choisit « Tri d'une matrice » ou « Tri d'une liste ».

- Le programme doit être écrit sous forme d'un ensemble de fonctions (en langage C).
- Les noms des fonctions et des variables doivent être significatives
- Les commentaires doivent figurer dans le code source pour plus de clarté et de compréhension.
- Des affichages de résultats intermédiaires sont demandés, en précisant le numéro de l'itération (étape).

Rapport à remettre : Environ **10 pages**

Un rapport doit être rédigé en suivant le plan ci-après :

6. **Introduction**
7. **Objectifs du travail**
8. **Description des différents algorithmes** (avec commentaires), pour chaque méthode de tri, en précisant les différentes **complexités théoriques** de chaque algorithme.
9. **Evaluation expérimentale** :

   Choisir trois (03) algorithmes de **complexités différentes** et remplir un tableau d'évaluation expérimentale de la complexité des algorithmes en considérant des vecteurs d'entiers de taille 10, 20, 50, 100, 200, 500 et 1000.

Taille n	Nb permutations	Nb comparaisons	Nb itérations	**Temps d'exécution (Tps)**
10				
20				
…				
1000				

   PS : Les données de test (vecteurs d'entiers) peuvent être remplies par programme afin d'éviter la saisie trop longue de valeurs pour les tableaux de grande taille.
   Le Temps d'exécution (Tps) devra être évalué par la machine
10. **Conclusion**

Annexe : Des captures d'écran illustrant l'exécution du programme (choisir les captures les plus significatives et sans redondance).