AlPro
Tp2 - Sorting

1 Introduction

A sorting algorithm is an algorithm allowing to order a set of elements according to a determined order. There is a wide variety of sorting algorithms. Their performances vary according to the set considered: an algorithm can be very fast to order a completely disordered list whereas it will be particularly poor with a list which is almost already sorted. It is therefore important, for the designer of a computer program, to know the specificities of each of these algorithms. We propose to implement some of the best known sorting algorithms. The theoretical principle of these algorithms will be illustrated by an example that will be used as a guideline for this tutorial: sorting matches of different sizes and colors. Display functions will be provided for the translation into C with the ncurses library. In a first approach, it is asked to sort the matches only according to their size.

2 Input - Output

The goal of this part is to build two functions that will read and write the files related to the storage of matches information (color and size). The data about each match is read from an input file and stored in a record that can then be used in the body of the program. We also want to be able to save the configuration of the set of matches (for example, to keep a trace of a sorted configuration): it is therefore useful to have a function that writes in a file all information about a state of the match set.

File format: `matches_data.txt` *The file always starts with a line containing an integer indicating the number of matches to sort. Then follows the information specific to each match (one match per line): first an integer (between 1 and 6) corresponding to its color, then, separated with a space, another integer (between 1 and 20) representing its size. The order in which these data are read corresponds to the order in which the matches are arranged.*

Data format. The information on the matches will be stored in a record type named `match`. The sorting operations will be done by manipulating an array of matches, of type `m_array`. We give the definition of these types:

```
types:
  match: record
    Integer: color
    Integer: size

  m_array: record
    match array: matches
    Integer: length
```

Question 1. Write in C these two data types in a header file `matches.h`. Be careful to use these names for the file and the types, because they will be used in the given functions to display matches.

Question 2. Write the algorithm for a function that reads in a file the data of a set of matches and store them progressively in an `m_array` of matches. Implement this function in C in a file `utilities.c`.

Question 3. Write the algorithm for a function that writes an `m_array` in a file, following the format described above. Implement this function in C in a file `utilities.c`.

Question 4. To make sure these functions are well-defined and implemented, write a `main` function that reads a set of matches in a given file, rewrite them identically in another file. You may use for your tests the file `matches_data.txt` available on hippocampus.

Question 5. To display a set of matches, you have 3 functions in the file `display.c`:

- `init()`: to be called *once and for all* before the first display;
- `finish()`: to be called *only once* at the end of the program;
- `display_m(m_array)`; to be called whenever you want to display the match set on the screen.

The declarations (prototypes) of these functions are given in the file `display.h`. The display also requires functions defined in the ncurses library. We provide you with a Makefile allowing the compilation with this library. This file has to be in the same folder as your source files.

Question 6. Complete your `main` function so that the match set is displayed.

3 Insertion sort

This is the sorting generally used by a player to sort his hand during a card game. It consists in considering each element successively and inserting it in the right place in the set of already sorted elements. Here is an algorithm adapted for the display of matches in this TP:

Algorithm 1 insert_sort(matches)

```
Integer i,j
for all i ← 2 to size(matches)−1 do
    // initialization
    j ← 0
    while j<i AND matches[j].size ≤ matches[i].size do
        j ← j+1
    end while
    // insert the match at the right place
    matches ← insert(matches,i,j)
end for
```

The function `insert` inserts the match of index `i` at index `j`, with `j ≤ i`:

Question 7. Write the algorithm for function `insert` an implement it in C in a file `sorts.c`. This implementation will need to use **only one array of matches** that it will modify.

Question 8. Implement the function `instert_sort` in C in the file `sorts.c`. This implementation will need to use **only one array of matches** that it will modify. Complete and use your `main` function to demonstrate the proper operation of this sort.

Question 9. Modify the function `insert_sort` so that for every value of `i` and `j`, the set of matches is printed. Test again your `main` function.

4 Bubble sort

Let n be the size of the array to be sorted. The principle of the bubble sort is to compare two by two the successive elements m_k and m_{k+1} of the array. If they are not in the right order, they are swapped. Then the elements m_{k+1} and m_{k+2} are compared and so on. When the elements m_{n-1} and m_n have been compared, the largest (or smallest, depending on the order) element is at location n. We must then iterate this treatment on the $n-1$ unsorted values, then the $n-2$ values and so on until the whole vector is sorted.

Example: To sort the integer list 8,4,9,1, the result of the application of the algorithm is (to be read column by column):

$$\begin{array}{lll}
\boxed{8,\ 4,}\ 9,\ 1 & \boxed{4,\ 8,}\ 1,\ 9 & \boxed{4,\ 1,}\ 8,\ 9 \\
4,\ 8,\ 9,\ 1 & 4,\ 8,\ 1,\ 9 & 1,\ 4,\ 8,\ 9 \\
4,\boxed{8,\ 9,}\ 1 & 4,\ \boxed{1,\ 8,}\ 9 & \\
4,\ 8,\ 9,\ 1 & 4,\ 1,\ 8,\ 9 & \\
4,\ 8,\boxed{9,\ 1} & & \\
4,\ 8,\ 1,\ 9 & &
\end{array}$$

Question 10. Write a function swap that swap two consecutive matches of a m_array and implement it in C in the file sorts.c.

Question 11. Implement the bubble sort algorithm in the file sorts.c and test this function in your main.

Question 12. What needs to be modified to sort the matches not only by size, but also by color (assuming that an order on the colors has been defined beforehand)? Implement these modifications in C and illustrate the good working of the whole on some significant test sets.

Question 13. (for the fastest) What is the "worst case", i.e., the configuration of items to be sorted for which the algorithm makes the most permutations? Give a majorant (an upper bound) of the number of operations (assignments and comparisons) in this case. In order to evaluate the performance (in time) of the algorithms, we are generally interested in this number of operations in the worst case when the size of the data to be processed tends towards infinity. This is called the asymptotic worst-case complexity. We can also look at the asymptotic complexity in the best case and in average to complete the analysis.

5 Comparison

We can now compare these sorting algorithm with more realistic sets of data.

Question 14. Write an algorithm and implement a function randomGeneration in C in the file utilities.c to generate a m_array of matches containing a number of matches passed in parameter with an random distribution of size and color.

Question 15. Compare the execution times of your sorting algorithms with data sets of 1000, 10000, 100000 matches. Take care to remove the display functions to do these tests.

6 Quick sort (Bonus)

In the quick sort, we choose a pivot (often the last element of the array), we arrange the elements of the array in two sets smaller or larger than the pivot, without these elements being sorted between them. Then we put the pivot back in its rightful place (since we know the number of smaller elements). And we repeat the sorting operation with the two subsets (smaller and larger than the pivot).

Question 16. Write an algorithm and implement a quick sort function in the file sorts.c and compare it with the previous ones.