Mini-project : Sorting Algorithms

Sorting is one of the most classically studied families of algorithms, because they are among the modules essential for the good running of more advanced algorithms. The general principle of a sorting algorithm is to order (in ascending order for example) the objects of a collection of data (values), according to a comparison criterion (key – for us, values and keys are here confused: these are the elements of an array of integers). We generally carry out sorting by using « in-place » approach: the sorted values are stored in the same array as the initial values (which therefore becomes an input-output parameter).
There are many sorting algorithms.

- ➢ The objectives of this project are multiple:

- • Put into practice and test some sorting algorithms;
- • Study sorting algorithms and calculate their complexity;
- • Confront theoretical complexity and evaluation of running cost;
- • In order to study the real cost of the algorithms, you will test them on arrays of integers of increasing size n filled randomly. To be reliable, time measurement must be done several times (for example 5 times) for a given array size. It is up to you to choose the values of n that seem relevant to you. For each studied sort, you will draw a graph representing the running time as a function of n;
- • Compare the different methods from a theoretical and experimental point of view.

- ➢ Submit a detailed report on the study of the different sorting methods as well as the source code and a conclusion.
- ➢ The work can be done in pairs.

1. Bubble Sort

Principle : Traverse array T of size N from last to first element (almost. . .), with an index i. At each step, the part of the array to the right of i is considered sorted. We then traverse the left part (unsorted part) with an index j. For each j, if T[j − 1] > T[j], we swap them.

- ➢ Write the C program for the following BubbleSort procedure:

```
Procedure BubbleSort (I/O: rray A[n] of integers ; I/n :integer)
    Begin
    Change = true ; (boolean variable)
    while (Change=true) do
    Change ← false ;
    for i←1 to n-1 do
        if (A[i] > A[i+1]) then
          Swap(T[i], T[i+1]) ;
          Change ← true;
         endif ;
    done;
 done ;
End;
```

- ➢ After the ith traversal of the array, all the last i elements are in their final places. So at each table traversal, the traversal can stop one index before the previous one. The algorithm becomes:

```
Procedure BubbleSortOpt (I/O: Array A[n] of integers ; I/n ;integer)
    Begin
    M← n-1 ;
    Change = true ; (boolean variable)
    while (Change=true) do
        Change ← false ;
        for i←1 to m do
          if (T[i] > T[i+1]) then
                Swap(T[i], T[i+1]) ;
                Change ←true ;
          endif ;
        done;
        m←m-1;
    done;
    End;
```

Write the C program of bubbleSortOpt then compare the two algorithms BubbleSort and BubbleSortOpt in the best and worst cases.

2. Gnome Sort

Principle : In gnome sort, we start at the beginning of the array, we compare two consecutive elements (i, i+1): if they are in order we move one step towards the end of the array (increment) or we stop if the end is reached; otherwise, we swap them and move one step towards the start of the table (decrement) or if we are at the start of the table then we move one step towards the end (increment).
Write the C program and give its theoretical complexity in the best and worst case.

3. Radix Sort

We use radix sort (also called sort by distribution) to sort integers according to their least significant digit (units digit), then to sort the list obtained by the tens digit then by the hundreds digit, etc.
The list of integers 141, 232, 045, 112, 143 will be sorted according to the ones digit, we obtain the list 141, 232, 112, 143, 045 which in turn will be sorted according to the tens digit, we obtain the list 112, 232, 141, 143, 045 will then be sorted according to the hundreds digit and we obtain the list of integers sorted according to the ascending order 045, 112, 141, 143, 232.
Consider an array A containing n integers between 0 and $10k-1$, where k is an integer constant. We want to sort by distribution these integers.

- Write the key function (I/ x, i: integer): integer; which returns either the unis digit, or the tens digit, or the hundreds digit…

Example: key (143, 0)=3, key (143, 1)=4, key (143, 2)=1

- Write the function SortAux(T, n, i) which reorders the elements of A such that: key(A[1], i) ≤ key(A[2], i) ≤ . . . ≤ key(A[n],i). SortAux must execute in linear time depending on the size n of the array.
- Using the SortAux procedure, write the function RadixSort(T, n, k) to sort by distribution the array A.
- Write the C program using the functions defined previously.

4. Quick Sort

Quick sort is based on a “divide and conquer” approach that can be broken down into 3 steps.
We consider that we have a Tab array of size n. We note a "subarray" of Tab, Tab[p…r], p being the index of the beginning and r the index of the end of the subarray.

✓ **Divide**: the subarray Tab[p…r] is partitioned (i.e. rearranged) into 2 non-empty subarrays A[p..q] and A[q+1..r] in such a way that each element of the array A[p..q] is less than or equal to each element of A[q+1…r]. The q index is calculated during the partitionning procedure.

✓ **Conquer:** The 2 subarrays A[p..q] and A[q+1..r] are sorted by recursive calls to the main QuickSort method.

✓ **Combine**: QuickSort sorts in-place. This implies that there is no additional work to merge them: the entire Tab[p..r] subarray is now sorted.

Quick sort has 2 functions QuickSort and partition which returns an integer. The integer returned by partition is the index we are looking for in the "Divide" step. The initial call on the tab array will be QuickSort(tab, 0, n-1):

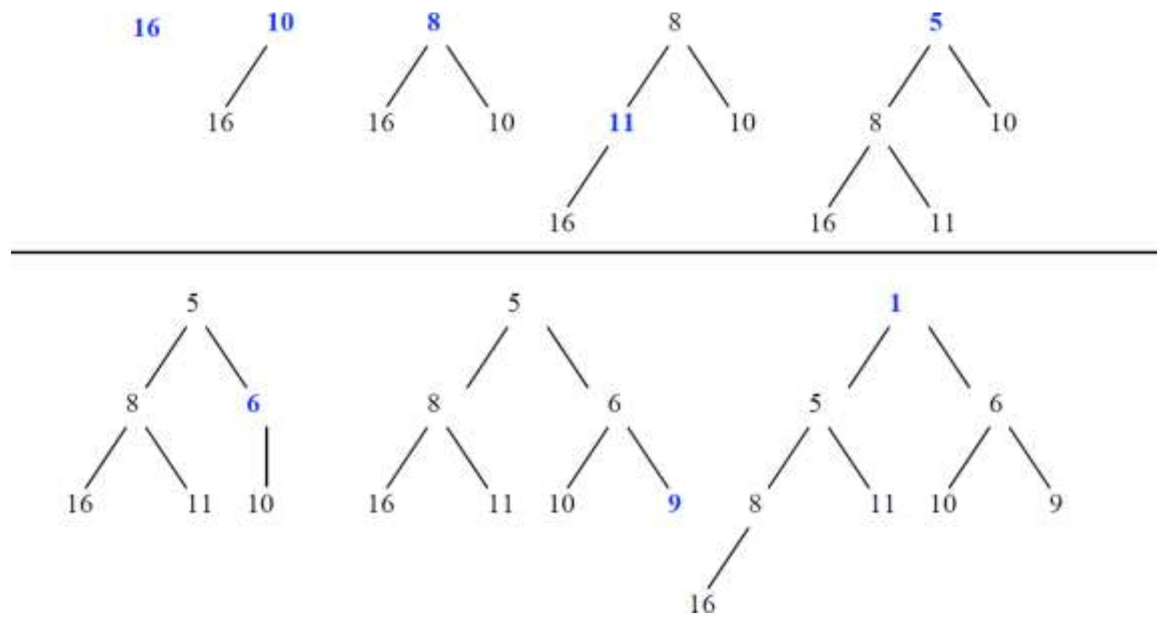
```
Procedure QuickSort(I/tab : array[n] of integers ; p, r :integer)
        Var q : integer;
   Begin
        if (p < r) then
        q ← partition(tab, p, r); QuickSort(tab, p, q);
        QuickSort (tab, q+1, r);
        endif ;
   End ;

function partition(I/tab : array[n] of integers ; d, f :integer) : integer
        var eltPivot : integer ; // we consider that tab is an array of integers
        i, j, x : integer ;
Begin
 eltPivot ← tab[p]; i← d ; j← f ;
Repeat
        while (j≤d) and (tab[j]≤ eltPivot) do
                j← j+1 ;
        done ;
        while (i≥f) and (tab[i]> eltPivot) do
                i← i-1 ;
        done ;
        if (j<i) then
                x←tab[j] ;
                tab[j]← tab[i] ;
                tab[i]←x;
                j←j+1; i←i-1;
                endif;
until (j>i)
return (i) ;
End ;
```

➢ Write the Quick Sort C program. Give and solve the recurrence equation of this sort.

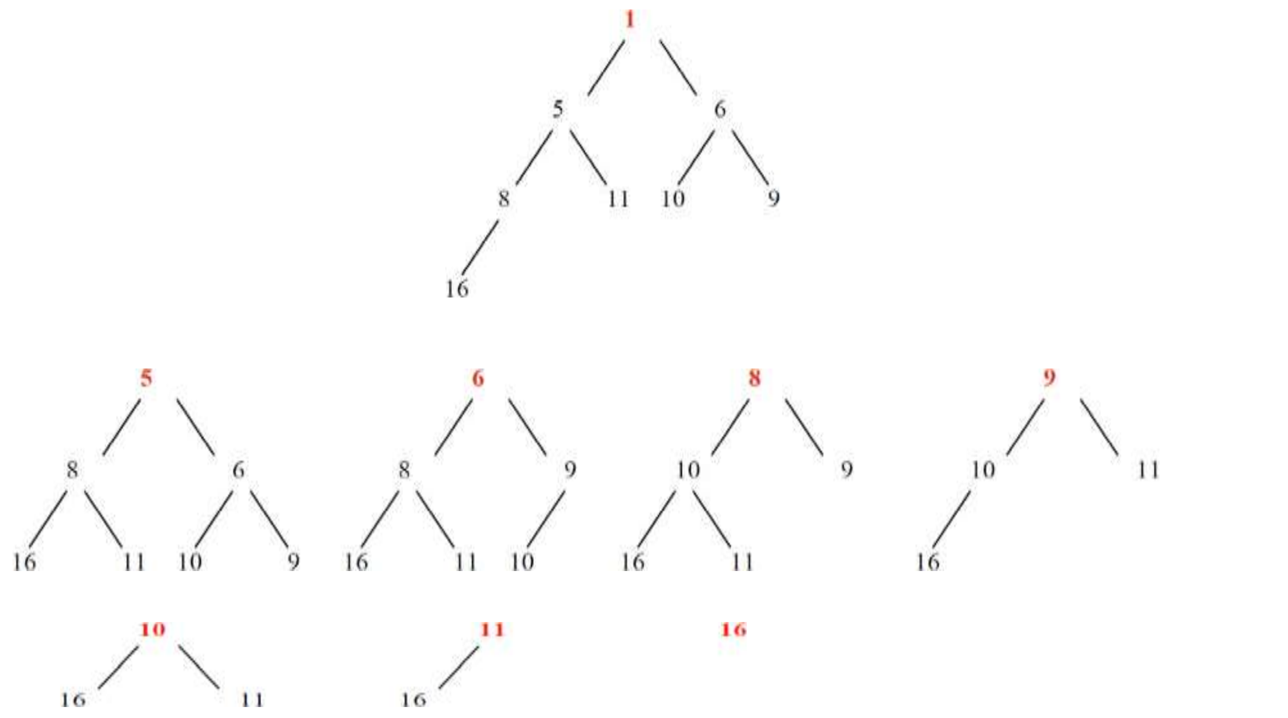
5. Heap Sort

Heap sort is based on a particular data structure: the heap. This is a representation of a binary tree in tabular form. The tree is almost complete: it is filled at all levels, except potentially the last. The tree is traversed by calculating the index on the table. For a node with index i, we have the father at index $\lfloor i/2 \rfloor$, the left child at index 2i and the right child at index 2i+1.
We will apply the procedures for inserting and deleting the root of a heap to the example so as to, firstly, construct a heap then by deleting the minimum reach the solution where all the elements are in the ascending order. We will first build the heap by inserting the elements of the example into this heap one after the other. Here is the representation of the different steps for the list 16-10-8-11-5-6-9-1.

At each step, the added element is indicated in blue.



By progressively deleting min by min, with the procedure delete_min, we go through the following heaps and draw the ordered list.



Write the C program for Heap Sort then evaluate the complexity of the algorithm.