Header Files

Sorting Technique's functions in a Header File

Aditya Raj & Akash Hebbar

R.V. COLLEGE OF ENGINEERING

(Autonomous Institution Affiliated to VTU, BELAGAVI)

**R.V. Vidyaniketan Post, Mysore
Road, Bengaluru - 560 059**

Department of Master of Computer Applications

DATA STRUCTURE USING C

18MCA13

LITERATURE SERVEY

Submitted To: -

Dr. Andhe Dharani

Mr. Krishnan R.

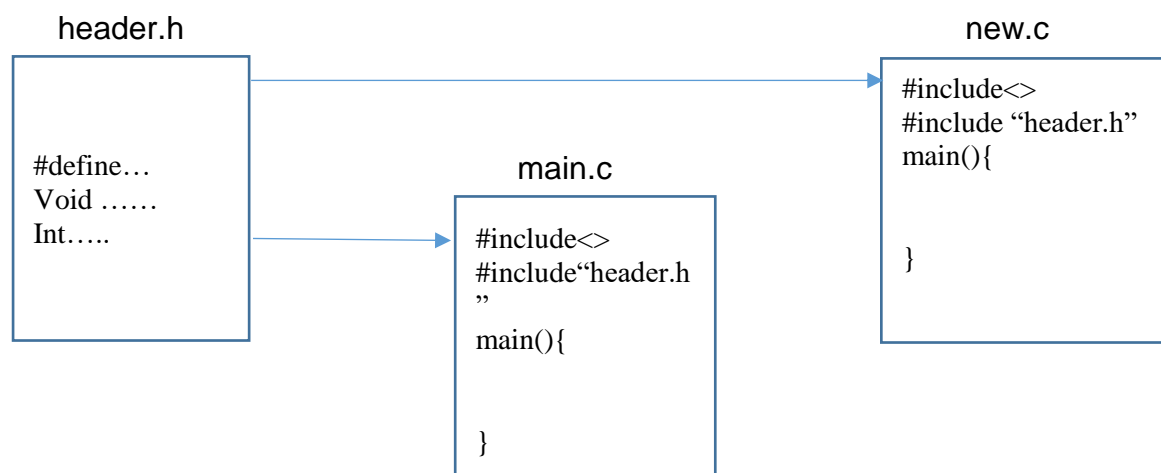**Submitted By**: -

Akash Hebbar P.

Aditya Raj

Analysis and Review of Sorting Algorithms

Header files

A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.

The application programming interface (API) of the C standard library is declared in a number of header files. Each header file contains one or more function declarations, data type definitions, and macros.

Header Files are included in the beginning of the program. It is a predefined code and we have to simply include the file. Thus, writing less is an advantage. It has .h extension. It contains C declarations and macro definition.

header.h

```
#define…
Void ……
Int…..
```

main.c

```
#include<>
#include"header.h"
main(){


}
```

new.c

```
#include<>
#include "header.h"
main(){


}
```

What should be in the header files for a complex project?

In C, the contents of a module consist of structure type (struct) declarations, global variables, and functions. The functions themselves are normally defined in a source file (a ".c" file). Except for the main module, each source (.c) file has a header file (a ".h" file) associated with it that provides the declarations needed by other modules to make use of this module.

The idea is that other modules can access the functionality in module X simply by #include "**X.h**" for the header file, and the linker will do the rest. The code in **X.c** needs to be compiled only the first time or if it is changed; the rest of the time, the linker will link X's code into the final executable without needing to recompile it, which enables the Unix make utility and IDEs to work very efficiently.

A well-organized C program has a good choice of modules, and properly constructed header files that make it easy to understand and access the functionality in a module. They also help ensure that the program is using the same declarations and definitions of all of the program components. This is important because compilers and linkers need help in enforcing the One Definition Rule.

Furthermore, well-designed header files reduce the need to recompile the source files for components whenever changes to other components are made. The trick is reduce the amount of "coupling" between components by minimizing the number of header files that a module's header file itself #includes. On very large projects, minimizing coupling can make a huge difference in "build time" as well as simplifying the code organization and debugging.

Sorting Algorithms

1. Selection Sort

This algorithm works by **selecting** the **smallest unsorted** item and then **swapping** it with the item in the **next position** to be filled.

    The selection sort works as follows: you look through the entire array for the **smallest** element, once you find it you **swap** it (the smallest element) with the **first** element of the array. Then you look for the **smallest** element in the remaining array (an array without the first element) and swap it with the **second** element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on. Here is an example,

```
void selectionSort(int[] ar){
 for (int i = 0; i ‹ ar.length-1; i++)
 {
 int min = i;
 for (int j = i+1; j ‹ ar.length; j++)
        if (ar[j] ‹ ar[min]) {
     min = j;
     int temp = ar[i];
     ar[i] = ar[min];
     ar[min] = temp;

 }}}
```

Algorithm

```
Step 1 – Set MIN to location 0
Step 2 – Search the minimum element in
the list
Step 3 – Swap with value at location
MIN
Step 4 – Increment MIN to point to next
element
Step 5 – Repeat until list is sorted
```

Example.
    29, 64, 73, 34, **20**,
    20, **64**, 73, 34, **29**,
    20, 29, **73**, **34**,64
    20, 29, 34, **73**, **64**
    20, 29, 34, 64, 73

This algorithm is not suitable for large data sets as its average and worst-case complexities are of O(n2), where n is the number of items.

2. Insertion Sort

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position. The resulting array after k iterations has the property where the first k + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position.

Example. We color a sorted part in green, and an unsorted part in black. Here is an insertion sort step by step. We take an element from unsorted part and compare it with elements in sorted part, moving from right to left.

29, 20, 73, 34, 64

29, 20, 73, 34, 64

20, 29, 73, 34, 64

20, 29, 73, 34, 64

20, 29, 34, 73, 64

20, 29, 34, 64, 73

Algorithm

```
Step 1 - If it is the first element, it is
already sorted. return 1;
Step 2 - Pick next element
Step 3 - Compare with all elements in the sorted
sub-list
Step 4 - Shift all the elements in the sorted
sub-list that is greater than the
       value to be sorted
Step 5 - Insert the value
Step 6 - Repeat until list is sorted
```

Let us compute the worst-time complexity of the insertion sort. In sorting the most expensive part is a comparison of two elements. Surely that is a dominant factor in the running time. We will calculate the number of comparisons of an array of N elements:

we need 0 comparisons to insert the first element we need 1 comparison to
insert the second element we need 2 comparisons to insert the third element
...
we need (N-1) comparisons (at most) to insert the last element
Totally,

　1　　+ 2 + 3 + ... + (N-1) = $O(n^2)$

　2　Best Case Time Complexity [Big-omega]: **O(n)**

　3　Average Time Complexity [Big-theta]: **O(n²)**

4   Space Complexity: **O(1)**

3. Bubble Sort

In this task, the goal is to sort an array of elements using the bubble sort algorithm. This uses the divide and conquer method of algorithm.The elements must have a total order and the index of the array can be of any discrete type. For languages where this is not possible, sort an array of integers. The bubble sort is generally considered to be the simplest sorting algorithm. Because of its simplicity and ease of visualization, it is often taught in introductory computer science courses. Because of its abysmal $O(n^2)$ performance, it is not used often for large (or even medium-sized) datasets.

void BubbleSort(int a[])

{

int i,j;   Algorithm

for (i=MAXLENGTH; --i >=0;) {

swapped = 0;

for (j=0; j<i;j++) {

 if (a[j]>a[j+1]) {

 Swap[a[j],a[j+1]); swapped=1;

}}

if (!swapped) return;

}}

```
begin BubbleSort(list)
    for all elements of list
        if list[i] > list[i+1]
            swap(list[i], list[i+1])
        end if
    end for
    return list
end BubbleSort
```

4. Quick Sort

Quick sort is a comparison sort developed by Tony Hoare. Also, like merge sort, it is a divide and conquer algorithm, and just like merge sort, it uses recursion to sort the lists. It uses a pivot chosen by the programmer, and passes through the sorting list and on a certain condition, it sorts the data set. Quick sort algorithm can be depicted as follows:

QUICKSORT (A)

1:   step←m;

2:   while step > 0

3:   for (i←0 to n with increment 1)

4:   do temp←0;

5:   do j←i;

6:   for (k←j+step to n with increment step)

7:   do temp←A[k];

8:   do j←k-step;

9:   while (j>=0 && A[j]>temp)

10: do A[j+step]=A[j];

11: do j←j-step;

12: do Array[j]+step←temp;

13: do step←step/2;

5. Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merg() function** is used for merging two halves. The merge (arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while

   while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
   end while

   return c

end procedure
```

Complexity of Mergesort

Suppose T (n) is the number of comparisons needed to sort an array of n elements by the Merge Sort algorithm. By splitting an array in two parts we reduced a problem to sorting two parts but smaller sizes, namely n/2. Each part can be sort in T(n/2). Finally, on the last step we perform n-1 comparisons to merge these two parts in one. All together, we have the following equation

$$T(n) = 2*T(n/2) + n - 1$$

The solution to this equation is beyond the scope of this course. However, I will give you a reasoning using a binary tree. We visualize the merge sort dividing process as a tree.

Figure1: Example of Merge Sort
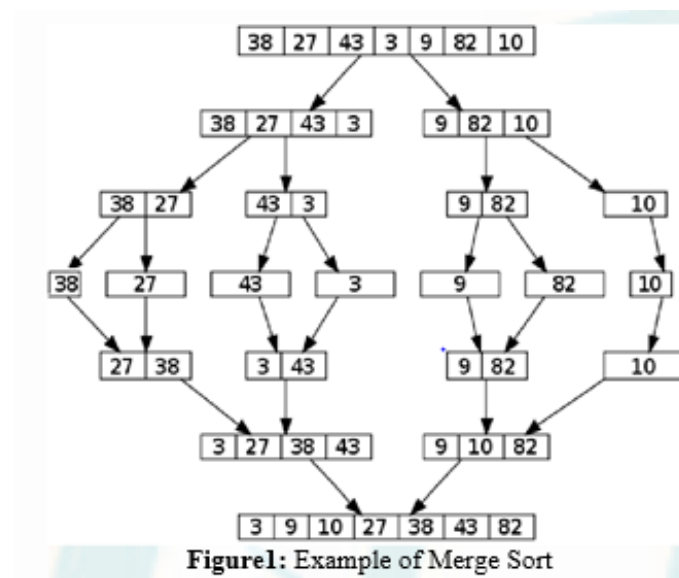
6. **Heap Sort**

Heapsort is a comparison-based sorting algorithm. Heapsort is part of the selection sort family; it improves on the basic selection sort by using a logarithmic-time priority queue rather than a linear-time search. Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case $O(n \log n)$ runtime. Heapsort is an in-place algorithm, but it is not a stable sort. It was invented by J. W. J. Williams in 1964.

	Worst Case	Average Case	Best Case
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Figure 2: Comparison of sorting algorithms

• <u>**Conclusion**</u>

This paper discusses comparison-based sorting algorithms function using user defined **Header File**. It analyses the performance of these algorithms for the same number of elements. It then concludes that selection sort shows better performance than Quick sort but being the simple structure selection sort is more preferred and widely used. It is clear that both sorting techniques are not that popular for the large arrays because as the arrays size increases both timing is slower for integer and string, but generally the study indicate that integer array have faster CPU time than string arrays. Both have the upper bound running time O (n2). Bubble Sort is a very simple algorithm that is only suitable for small lists. There are lots of alternative sorting algorithms that are more performant than Bubble Sort. Bubble Sort is widely considered to be the least performant of all the established sorting algorithms.

The sorting algorithm Mergesort produces a sorted sequence by sorting its two halves and merging them. With a time complexity of $O(n \log(n))$ Mergesort is optimal. However, selection sort has the property of minimizing the number of swaps. In applications where the cost of swapping items is high, selection sort very well may be the algorithm of choice.

• <u>**References**</u>

1. Introduction to Algorithms is a book by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. C Header File Guidelines David Kieras, EECS Dept., University of Michigan December 19, 2012.
3. https://www.tutorialspoint.com/data_structures_algorithms/index.htm