Fundamental of Computer Programing

{Sir, Muhammad Muneeb Ullah}



More efficient sort than bubble sort

Assignment # 3



Prepared by:

Hamad Nasir (120312)

Abdul Ghaffar Kalhoro (194699)

Date:

January 17, 2017

# Sorting in Programming:

Sorting basically means to sort data in increasing or decreasing fashion by comparing elements of that data according to some linear relationship between them. Data can be of any form for example names, numbers, and records.

For understanding consider that it is to find the phone number of a friend from a telephone dictionary when the names in the phone book are sorted into alphabetical order. If the names are not arranged, then it will take incredibly long to find that number. This example clearly illustrates one of the main reasons that sorting large quantities of [information](http://ecomputernotes.com/fundamental/information-technology/what-do-you-mean-by-data-and-information) is desirable.

Sorting algorithms are an important part of managing data in Programming. There are different types of sorting algorithms. Each algorithm has its own strengths and weaknesses. For sorting, there is need of a quick and dirty sorting algorithm, there are variety of choices for this purpose.

Most of the sorting algorithms work by comparing the data that is being sorted. In some cases, it may be desired to sort a large chunk of data for example a struct containing a name and address based on only a portion of that data. The piece of data that is used to determine the sorted order is called the key.   
Sorting algorithms are mostly judged by their efficiency. In this case, efficiency refers to the algorithmic efficiency as the size of the input grows large and is generally based on the number of elements to sort means the number of elements that needs to be sorted. Many algorithms that have the same efficiency do not have the same speed on the same input.

First, algorithms must be judged based on their average case, best case, and worst case efficiency. Some algorithms perform exceptionally well for some inputs, but not much efficient for others. A second criterion for judging algorithms is their space requirement – do they require scratch space or can the array be sorted in place without additional memory beyond a few variables? Some algorithms never require extra space, whereas some are most easily understood when implemented with extra space. Space requirements may even depend on the data structure used merge sort on arrays versus merge sort on linked lists, for instance.   
A third criterion is stability -- does the sort preserve the order of keys with equal values? Most simple sorts do just this, but some sorts, such as heap sort, do not. 

# Types of Sorts:

The Sorts other than the Bubble sort are listed below:

* Selection Sort
* Insertion Sort
* Shell Sort
* Merge Sort
* Quick Sort

From above list of sorts, there is a sort called as ‘Quick Sort’ which leads all other sorts with its marvelous advantages in use case.

It was 1st proposed by Tony Hoare. It is a commonly used algorithm for sorting. Quicksort can operate in-place on an array, in compare with merge sort, it requires small additional amounts of memory to perform the sorting.

Mathematical analysis of quicksort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items.

**Quick Sort Algorithm Steps:**

* Select a pivot element (pick last element in array): array of the elements with the greater values.
* Partition the input array based on the pivot element:
* Sort both halves of the partitioned arrays:
* Concatenate the sorted halves along with the pivot element:

**Note:** In the below implemented quick sort program, always first element is set as Pivot element. But if array is already sorted, then the time complexity becomes worse, as quick sort will take quadratic time. So pivot element should be chosen in such a way, that partitioning of array becomes half.

# Comparison of Bubble Sort and Quick Sort:

Quicksort algorithm is a divide and conquer algorithm which sorts the given sequence in place meaning that it doesn't require extra storage as we need in bubble sort. Quick sort requires less steps to sort the sequence then bubble sort and thus is faster. These are based on following steps:

1. First it breaks a large array or sequence into 2 sub array.

2. Then take a pivot point.

3. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After this partitioning, the pivot is in its final position.

4. Apply these above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values until the complete list is sorted.

Whereas, the bubble sort makes multiple passes within a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. This process is carried out until the all list is sorted. The basic method used in quick sort is portioning whereas in bubble sort there is exchanging.

Example:

**Source code of simple quick sort implementation using array ascending order in c programming language**

#include<stdio.h>

#define SIZE 9

void quicksort(int[SIZE], int, int);

int main(){

//Declaring variables.

int array[SIZE] = { 54, 26, 93, 17, 77, 31, 44, 55, 20 }, i;

printf("The unsorted elements are: ");

for (i = 0; i<SIZE; i++)

printf(" %d", array[i]);

//calling quickSort function defined below.

quicksort(array, 0, SIZE - 1);

printf("\n\nSorted elements: ");

for (i = 0; i<SIZE; i++)

printf(" %d", array[i]);

puts("");

puts("");

puts("");

return 0;

} //end main

//quick Sort function to Sort Integer array list

void quicksort(int array[], int first, int last){

int pivot, j, temp, i;

if (first<last){

//assigninh first element as pivot element

pivot = first;

i = first;

j = last;

//Sorting in Ascending order with quick sort

while (i<j){

while (array[i] <= array[pivot] && i<last)

i++;

while (array[j]>array[pivot])

j--;

if (i<j){

//Swapping opertation

temp = array[i];

array[i] = array[j];

array[j] = temp;

}

}

//At the end of first iteration, swap pivot element with 'j' element

temp = array[pivot];

array[pivot] = array[j];

array[j] = temp;

//Recursive call for quick sort, with partiontioning

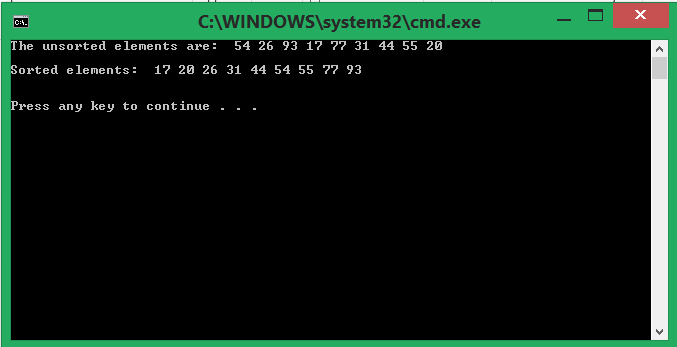
quicksort(array, first, j - 1);

quicksort(array, j + 1, last);

}

}

**OUTPUT:**

****

# 