**Beteab Gebru**

**April 18, 2017**

**CSCI 301 - Section 02**

**COMPARING SORTING ALGORITHMS**

**(MERGE-SORT, QUICK-SORT, BUBBLE SORT)**

**Project 9: Program Documentation**

### Introduction

INTRODUCTION

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. It can describe the **time complexity** or **memory complexity** of an algorithm to solve a given problem. Time complexity refers to the time (measured in number of characteristic operations) it takes to solve given problem. Space complexity refers to the amount of space required by the algorithm during run time. Time complexity of an algorithm is increasingly important aspect as we deal with bigger data sets as time moves forward.

Sorting is one of the most important operations in data processing. I will be looking at three sorting algorithms, namely Merge\_Sort, Quick\_Sort, Bubble\_Sort. I will be using random number generator to populate an array of integers and use the sorting algorithm to sort arrays with varying arrangements and analyze the times.

-max theoretical limit of sorting (Log2N)

**Design Document**

## **Insertion Sort [O(n2)]**.

The idea behind insertion sort is:

1. Put the first 2 items in correct relative order.
2. Insert the 3rd item in the correct place relative to the first 2.
3. Insert the 4th item in the correct place relative to the first 3.
4. :

Every time we consider an the next character we have to compare it to the sorted portions of the array

* 1st iteration of outer loop: inner loop executes 1 Characteristic operations
* 2nd iteration of outer loop: inner loop executes 2 Characteristic operations
* 3rd iteration of outer loop: inner loop executes 3 Characteristic operations
* :
* (n-1)st iteration of outer loop: inner executes n-1 times
  + O(n) = 1 + 2 + ... + n-1 = c(n2)

which is still -> O(N2).

## 

## **Merge Sort: [O(nlogn)].**

The idea with Merge-Sort algorithm is that we can merge two sorted arrays (each containing N/2 items). Sorting them separately and merging them takes shorter time than to sort the whole data set in one-go. The merging of sorted arrays takes O(N) steps. The height of the merge tree is O(log N). The whole operation takes O(NlogN) steps. The height of the tree (consecutive halving) \* The number of items in the divided part(N).

* Find the **pivot** ->middle index of the data set. Takes O(1). Finding the Pivot is done at every division level (1,2,3,4…….N/2 ) times. The last level involves the base case therefore the pivot is not needed. So at any stage the number of operations to find Pivot will be at most O(N).
* The time for the merging step at every level is proportional to the size of the array(portion of the whole). All the portions of the array take O(N) time the total time will add up to the N(the size of the original) array all the time

**In short**

* Recursively, sort the left half.
* Recursively, sort the right half.
* Merge the two sorted halves.

**Merge-Sort involves O(N) work done at each division level with recursive calls. Since there are O(log N) levels, the total worst-case time is O(N log N).**

## **Quick Sort: [O(nlogn)].**

Quick sort is classified as a divide and conquer algorithm; It divides up the data set into two halves and sorts the two halves recursively to combine them into one sorted set. Dividing the array into two involves more work than in Merge-Sort but when merging the sorted arrays, it simply puts them together.

Ideally the Pivot point should be the **median** value, however ensuring this case is an expensive operation, therefore we pick a random number to be the pivot, and put all larger values on the right, and smaller values to its left.

1. Choose a pivot value.
2. Partition the array (put all value less than the pivot in the left part of the array, then the pivot itself, then all values greater than the pivot).
3. Recursively, sort the values less than the pivot.
4. Recursively, sort the values greater than the pivot.

-Quick-Sort has best case scenario when the left and right of the pivot there are equal number of items.

-If the right and left side are not balanced then we will have worst case scenarios. If we encounter a sorted array and we chose the first or last item as the pivot it will cause the worst case of all scenarios(needing O(N)recursive calls therefore O(N2) overall time).How fast the algorithm solves the problem will be dictated by how balanced the partitions are . Ultimately finding the right Pivot to use will be the way to increase efficiency of this algorithm.

**The Main program**

My program looks to test and extract performance data for the three sorting algorithms in Question. I will be using arrays of various sizes that are copied from a large array of unsorted integers(int Unsorted[1000]). Using this copies we will be able to send the same profile of data to get accurate reflections of their performance.

The Main program calls the following functions

-void Merge\_Sort(int The\_Array[], int, int );

-void Merge(int The\_Array[], int, int, int);

-void Insertion\_Sort(int The\_Array[], int );

-void Quick\_Sort(int The\_Array[], int First, int Last);

-void Create\_Arrays();//we create the arrays to be sorted

We ask from the user **Array\_Size,** and choose if the array should be displayed. With the user’s input in we go ahead and select the said array sizes and send them to be sorted. Using the operations counters, we extract performance data from the algorithms. The program will ask the user if they want to continue running the sorting tester. We can do multiple runs of the same array size and that would allow us to observe how the Algorithms act when the array is already sorted.

### User Document

### How to run the program

The program is run using visual studio 2015 c++11. Simply compile the console program in the IDE or use g++ to run it from the folder directory it is stored in.

-User must put in program parameters to get performance statistics on the algorithms.

**Test Design**

-I will run the program to get counts for unsorted-array sizes as follows.

-results in table 1.0

-I will run the programs on the already sorted arrays of all sizes

-results in table 2.0

**Testing the sorting algorithms**

Table 1.0 Sorting algorithm on Unsorted array random numbers

|  |  |  |  |
| --- | --- | --- | --- |
| SIZE of array | Insertion-Sort  Number of char operations | Merge-Sort  Number of char operations | Quick-Sort  Number of char operations |
|  | Program Output | Program Output | Program Output |
| 10 | 20 | 34 | 19 |
| 50 | 588 | 286 | 136 |
| 100 | 2250 | 672 | 310 |
| 500 | 59321 | 4488 | 1727 |
| 1000 | 245591 | 9976 | 3813 |

Table 2.0 Sorting algorithm on Sorted array of random numbers

|  |  |  |  |
| --- | --- | --- | --- |
| SIZE of arrays | Insertion-Sort  Number of char operations | Merge-Sort  Number of char operations | Quick-Sort  Number of char operations |
|  | Program Output | Program Output | Program Output |
| 10 | 9 | 34 | 11 |
| 50 | 49 | 286 | 61 |
| 100 | 99 | 672 | 125 |
| 500 | 499 | 4488 | 521 |
| 1000 | 999 | 9976 | 1053 |

**Observations from the Numbers**

**Merge-Sort**

This algorithm doesn’t get affected much by the statistical profile of the data set. Sorted or not it takes the same time to complete its work.

In fact it takes 34 comparisons for 10 unordered or ordered elements in an array even if they are sorted already.

This algorithm performs better as the data sets become larger and more unordered, compared to the other two. For small data sets its unnecessarily cumbersome.

**Quick-Sort**

This algorithm becomes lot more efficient when the array/data is already sorted. It improved from 19 comparisons to sort 10 unordered elements 🡪 11 comparisons when array is already sorted.

This algorithm on average looks to perform better than the two considering it doesn’t increase comparisons dramatically like Insert-Sort as the number of items increases.

**Insert-Sort**

This algorithm is one that improves massively when the data set in question is sorted already. In fact, the algorithm goes from O(n2) to O(n) when the array is already sorted.

Overall this very inefficient algorithm to use to sort large sets of data as the number of operations will rise exponentially.

### Summary

Timing an algorithm’s time to solve a problem can be done using the system clock or using a counter to find the number of characteristic operations. Counting operations is a more portable way of answering Algorithm Efficiency Questions. There is no dependency on the hardware and compiler configuration affects the which affect speed of execution. This is effective way but there is ambiguity on what operations to count and which ones to ignore. Some operations such as assignment operations may be ignored but they do have real effect on how fast an algorithm executes.

The unsorted arrays are passed first and passed again after they are sorted, in the same runtime to extract the performance data for sorted arrays. We take advantage of the fact that arrays are passed by reference by default therefore we are able to pass the array that we sorted initially. This eliminated the need to build a separate sorted array.

**Conclusion**

The number of operations is used to [analyze the algorithm complexity](http://en.wikipedia.org/wiki/Analysis_of_algorithms). The idea is to have a rough idea how many operations are in the worst case and in the best-case scenarios of an algorithm. Having an idea of this allows us to seek ways to make our program predictable. Knowing the time needed to solve a sorting problem will help is resource planning and seeking efficiency improvements where necessary.

This algorithm, even though they are well known they can be altered to find efficiency gains. By choosing the right Pivot item we can improve the Quick-Sort massively. Assignment statements don’t take much resource when its integers but they r considerable when we are dealing with larger non-native datasets.

Choosing the sorting algorithm to use is a factor of many things amongst which; Size of problem, Type of data to be sorted and few others come into question.