Bilkent Univeristy

Computer Engineering

CS202 – Section 2

Fundamental Structures of Computer Science II

Section TA: Ilkin Safarli

Supervisor: Uğur Doğrusöz

Homework

By: Faaiz Ul Haque

21503527

10/8/2017

Question 1 – 25 points

(a) [10 points] Sort the functions below in the increasing order of their asymptotic complexity: f1(n) = 10π,f2(n) = n,f3(n) = √n,f4(n) = logn,f5(n) = n^0.0001, f6(n) = nlogn,f7(n) = 2^n,f8(n) = n!,f9(n) = log(n!),f10(n) = n^n

1. f1(n) = 10π
2. f4(n) = logn
3. f5(n) = n^0.0001
4. f3(n) = √n
5. f2(n) = n
6. f9(n) = log(n!)
7. f6(n) = nlogn
8. f7(n) = 2^n
9. f8(n) = n!
10. f10(n) = n^n

(b) [10 points] Express the running time complexity (using asymptotic notation) of each loop separately. Show all the steps clearly.

Listing 2: Loops

```
// Loop A - 3 points
for (i = 1; i <= n; i++)
    for (j = 1; j <= i; j++)
        sum = sum + 1;
```

Answer: O(n^2)

For this case there is a double for loop with one simple operation that takes one unit of time. Since there is a double nested for loop we can calculate the time complexity accordingly. The outer for loop will execute a total number of n times and the inner loop will execute on a value dependent on i. We know that if i = 1, second loop will iterate 1 times, if i = 2, second loop will iterate 2 times and so on..

So (i = 1, j = 1), (i=2, j=2)… all the way up to n times (i=n, j = n). Therefore the total times that the single statement sum = sum + 1; will iterate is.. 1 + 2 + 3 + 4.. + n = (n(n+1)) / 2 which is equal to (n^2 + n) / 2. We can get rid of the constant to get n^2 + n, and since n^2 is dominate over n, we will be left with n^2.

```
// Loop B - 3 points
    i = 1;
    while (i < n) {
        for (j = 1; j <= i; j++)
            sum = sum + 1;
        i *= 2;
    }
```

Answer: O(nlog(n))

The sum= sum+1; line is a single operation and runs in one unit time. The while loop will execute n – 1 times, and the for loop will execute dependent on the i variable. For this reason when i = 1, for loop will execute one time, if i = 2 the for loop will execute two times and so on. Since i is incrementing by a factor of 2, the rate at which it increases is log n. So we have the following equation of times that the sum will be incremented.

1 + 2 + 4 + 8 + 16 + 32.. which is increasing by a logarithmic value we get n*log(n)

```
// Loop C - 4 points
    i = 1;
    while (i < n) {
        j = 1;
        while (j < i*i) {
                for (k = 1; k <= n; k++)
                        sum = sum + 1;
                j *= 2;
        }
        i *= 2;
    }
```

Answer: O(n*log(n) *log(n))

For the first outer while loop it will increment n times but i is increasing by log(n) we get one log(n) from this loop, the second while loop also is incrementing in the same way, and since its dependant on i, we can assume this also happens n times, so another log(n). Finally the for loop is linearly increasing n times, where we get the n from. Multiplying the values together since they are nested, we get n*log(n)*log(n)

(c) [5 points] Write the recurrence relation of merge sort and quick sort algorithms for the worst case, and solve them. Show all the steps clearly.

Merge sort worst case recurrence relation: T(n) = 2T(n/2) + O(n)

We can solve it using recursion trees.

T(n) and O(n) can be split into two halves to T(n/2) and O(n/2) which can be further split into four quarters T(n/4) O(n/4). Since each level is equal to n, and there are log(n) levels the time complexity is equal to nlog(n).

Answer: O(N*log(N))

Quick sort worst case recurrence relation: T(n) = T(L) + T(R) + O(n) where T(L) and T(R) represent the left and right sides of the partitioning. Since the worst case could be partitioning from the first element, we get T(n) = T(n-1) + O(n) as the worst case

We can solve it using repeated substitution.

1. **T(N) = T(N-1) + O(N)**
2. **T(N-1) = T(N-2) + O(N-1)**
3. **T(N-2) = T(N-3) + O(N-2)**

…

4. **T(3) = T(2) + 3**
5. **T(2) = T(1) + 2**

Sum up every term to get

T(N) + T(N-1) + T(N-2) … + T(3) + T(2) =

T(N-1) + O(N) + T(N-2) + O(N-1) + T(N-3) + O(N-2) .. T(2) + 3 + T(1) + 2

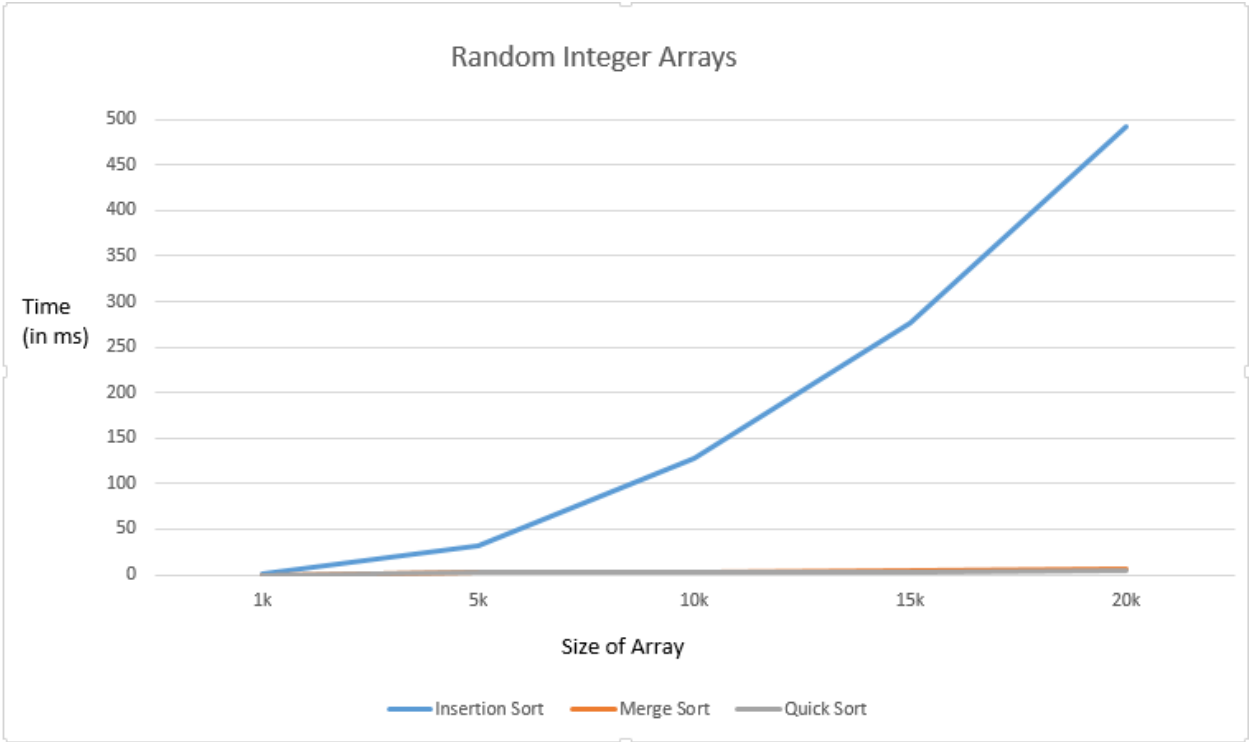
Cancelling common terms on both sides we get:

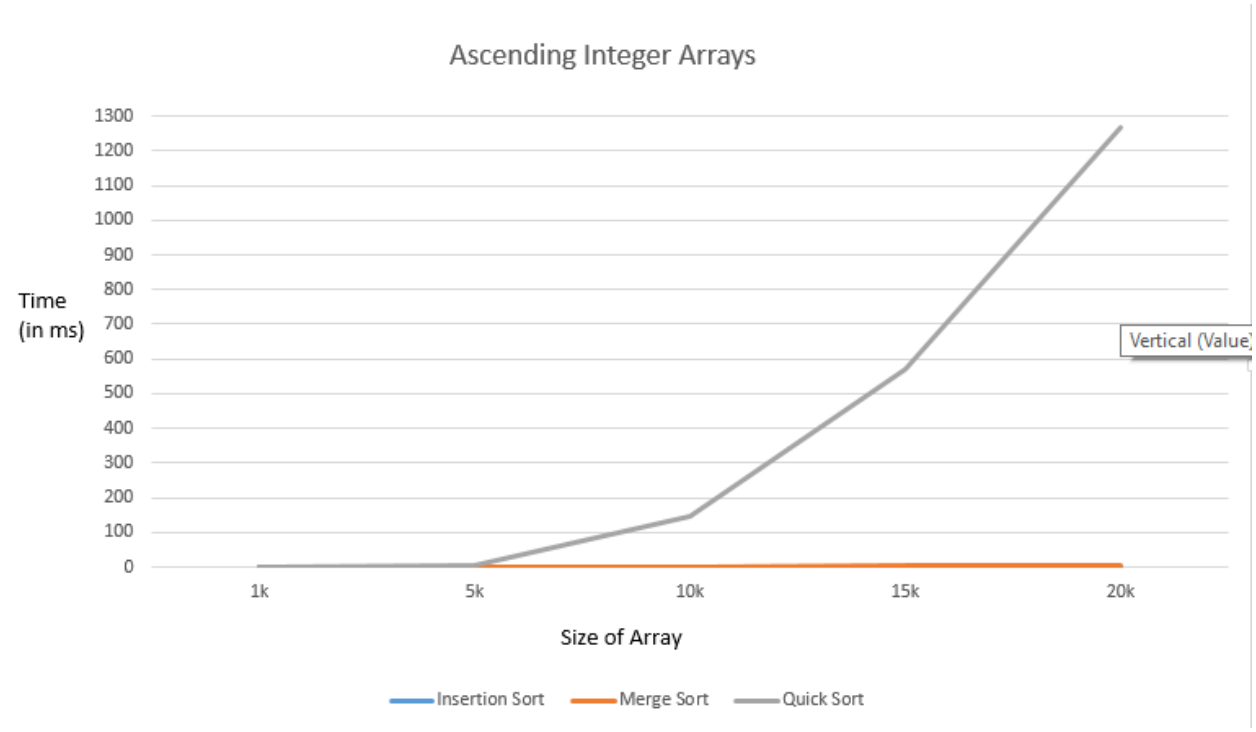
T(N) = O(N) + O(N-1) + O(N-2) .. + 3 + 2

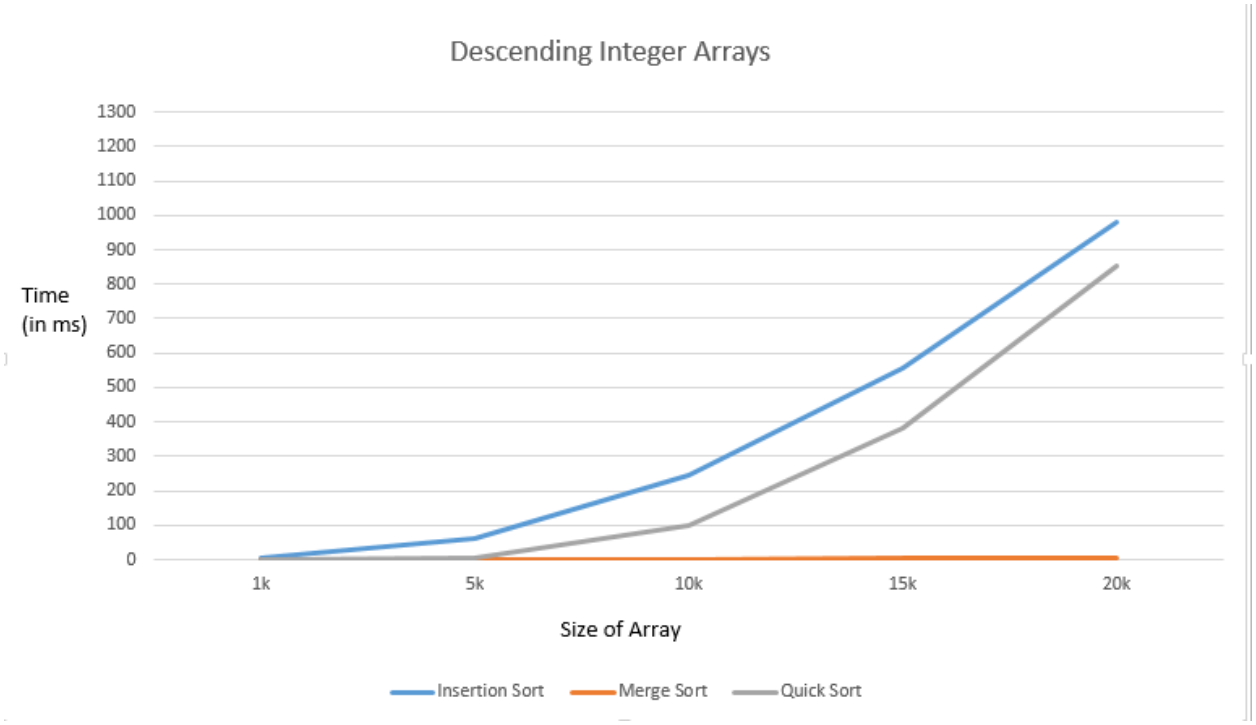
Answer: T(N) = O(N^2)

Question 3 – 25 points

Array	Elapsed Time (in ms)			Number Of Comparisons			Number of Data Moves		
	Insertion	Merge	Quick	Insert	Merge	Quick	Insert	Merge	Quick
R1k	1	0	0	254383	8715	10342	255382	19952	16390
R5k	32	2	2	6378081	55182	71388	6383080	123616	121496
R10k	127	3	2	24905286	120407	152158	24915285	267232	256352
R15k	276	4	3	56192368	189323	249542	56207367	417232	423851
R20k	492	6	4	99957680	260845	334561	99977679	574464	543054
A1k	0	0	0	999	5044	499500	1998	19952	1503495
A5k	1	0	6	4999	32004	12497500	9998	123616	37517495
A10k	1	2	147	9999	69008	49995000	19998	267232	150034995
A15k	0	3	571	14999	106364	112492500	29998	417232	337552495
A20k	0	5	1268	19999	148016	199990000	39998	574464	600069995
D1k	3	0	0	500499	4932	499500	501498	19952	753495
D5k	62	1	6	12502499	29804	12497500	12507498	123616	18767495
D10k	247	2	97	50004999	64608	49995000	50014998	267232	75034995
D15k	557	3	383	112507499	102252	112492500	112522498	417232	168802495
D20k	982	4	854	20009999	139216	199990000	200029998	574664	300069995




Ascending Integer Arrays
Time (in ms)
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
1k
5k
10k
15k
20k
Size of Array
Insertion Sort
Merge Sort
Quick Sort


Descending Integer Arrays
Time (in ms)
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
1k
5k
10k
15k
20k
Size of Array
Insertion Sort
Merge Sort
Quick Sort

Merge sort algorithm theoretically has an average, best and worst running time of O(logN). This is supported from the results as it is nearly the same case independent of the array type. It is to be noticed that merge sort's empirically calculated values are very similar to those of its theoretical, and it is seen to take the least time out of the 3 sorting algorithms as it also has the least data moves and comparisons in general.

Quick sort algorithm theoretically has an average and best case of O(logN) but a worst case of O(N^2). In the three different array types for the quick sort we can clearly see that for ascending and descending array types the quick sort algorithm takes really long since it needs to compare the entire array independent of if the pivot initial selection point is from the beginning or end. So in these cases it has a running time of O(N^2) in its worst case. A average case would be where the array is random, in this case we can see that quicksort didn't take much time in comparison to the other arrays and ran with a time of O(logN). So the empirical results are accurately compared with the theoretical.

For insertion sort in worst case it runs at O(N^2). For the descending integer array quicksort run at O(N^2) because it had to compare every element and sort it entirely in the opposite way. We can see for random arrays and for ascending arrays that quicksort ran at its average and best cases: O(N). Although the difference between these two is not displayed accurately, because of the chance that the random array is almost as good as its best case.

Since there is a high variance in the graphs, the scales are not good enough to accurately show the comparisons for graphs in certain cases. Example: In random integer arrays quick sort and merge sort are nearly on the same trend.

• When should insertion sort algorithm be preferred over merge sort and quick sort algorithms?

In the case of its best case where we want to sort an already sorted array in an ascending manner.

• When should merge sort algorithm be preferred over quick sort algorithm?

For the worst cases of the three: sort a descending integer array into an ascending integer array.