Oğuz Kaan İmamoğlu

21702233

Section 1

Homework 1

Question 1:

a) Functions was sorted with increasing asymptotic complexity

[$f_4(n) = \log n$] **,** [$f_5(n) = n^{1/\log n}$] **,** [$f_{10}(n) = \sqrt{n}$] **,** [$f_2(n) = n^{\log(\log n)}$] **,** [$f_9(n) = \log(n!)$] **,** [$f_6(n) = n\log n$] **,**

[$f_8(n) = n^3$] **,** [$f_7(n) = e^n$] **,** [$f_3(n) = n!$] **,** [$f_1(n) = 10^n$]

b)
Here is the algorithm:

```
int test (int n) {
if (n <=0)
return 0;
else {
int i = random (n);
return ( test (i) + test (n -1-i));

}

}
```

Random function can generate n as a output but also can give 0. So, we can say that avarage output will be (n+0) / 2 = n/2.

The test funcion will continue until (n/2)-1 = 0.

Let us express these as recurrence relation.

$T(0) = \Theta(1)$

$T(n) = T(n/2) + T(n/2 - 1) + \Theta(1)$

$= T(n/4) + T(n/4 - 1) + \Theta(1) + T(n/4 - ½) + T(n/4 – ½ - 1) + \Theta(1)$

To make it easy to calculate we can write this as:

$T(n) = T(n/2) + T(n/2) + \Theta(1)$

$= T(n/4) + T(n/4) + \Theta(1) + T(n/4) + T(n/4) + \Theta(1)$

$= 2*[2 * T(n/4) + \Theta(1)] + \Theta(1)$

$= 2^k T(n/2^k) + \sum 2^i * \Theta(1)$ (i = 0 to k-1)

⇨ We can say that after k iteration n reaches to 0 and recursion ends. So let us find k.

If n becomes 2^k, $n/2^k-1$ becomes 0 and execution will be completed

$2^k = n$, $k = \log_2(n)$, So:

$$\boxed{T(n) = \log(n)}$$

c)

Bubble Sort

Here is our array [607, 1896, 1165, 2217, 675, 2492, 2706, 894, 743, 568]

And here is our algorithm (which is given in the slides):

```
void bubbleSort( DataType theArray[], int n) {
  bool sorted = false;

    for (int pass = 1; (pass < n) && !sorted; ++pass) {
    sorted = true;
    for (int index = 0; index < n-pass; ++index) {
      int nextIndex = index + 1;
      if (theArray[index] > theArray[nextIndex]) {
        swap(theArray[index], theArray[nextIndex]);
        sorted = false; // signal exchange
      }
    }
  }
}
```

The variable n will be 10 in this case.
The inner loop was designed to transport biggest value to the end of the array, in other words the sorted zone. For this reason, index increases to n-pass. Pass increases after iteration completed once and n-pass represents the border of sorted zone.
The outer loop was designed to re-execute the process. After there is no element to swap, sorted remains true and both loops will be terminated, it means sorting is completed.
Let us trace the algorithm.

Here is our initial array:
607, 1896, 1165, 2217, 675, 2492, 2706, 894, 743, 568
pass = 1,
index = 0,
nextIndex = 1
Since 607 is not bigger than 1896 we will continue. Index and respectively nextIndex is incremented. 1896 is bigger than 1165 so swap operation is executed. Here is the new version.
607, 1165, 1896, 2217, 675, 2492, 2706, 894, 743, 568

After that index pass to 2217 and compares it with 675. New array will be:
607, 1165, 1896, 675, 2217, 2492, 2706, 894, 743, 568. And it continues like this:
607, 1165, 1896, 675, 2217, 2492, 894, 2706, 743, 568
607, 1165, 1896, 675, 2217, 2492, 894, 743, 2706, 568
607, 1165, 1896, 675, 2217, 2492, 894, 743, 568, 2706 With this step the inner loop is completed once. The biggest element is in the right place. And pass is incremented once. So, the end of the array is sorted. There is a bracket like this:

607, 1165, 1896, 675, 2217, 2492, 894, 743, 568 |2706

Now we will re-execute the process from the beginning.
607, 1165, 675, 1896, 2217, 2492, 894, 743, 568 |2706
607, 1165, 675, 1896, 2217, 894, 2492,743, 568 |2706
607, 1165, 675, 1896, 2217, 894, 743, 2492, 568 |2706
607, 1165, 675, 1896, 2217, 894, 743, 568, 2492 |2706 With this step the inner loop is completed twice. The biggest two elements are in the right place. And pass is incremented once. So, the end of the array is sorted. There will be a bracket like this:

607, 1165, 675, 1896, 2217, 894, 743, 568 |2492, 2706

These processes will be re-executed until the all list is sorted.
607, 1165, 675, 1896, 2217, 894, 743, 568 |2492, 2706
607, 675, 1165, 1896, 2217, 894, 743, 568 |2492, 2706
607, 675, 1165, 1896, 894, 2217, 743, 568 |2492, 2706
607, 675, 1165, 1896, 894, 743, 2217, 568 |2492, 2706
607, 675, 1165, 1896, 894, 743, 568, 2217 |2492, 2706

607, 675, 1165, 1896, 894, 743, 568 |2217, 2492, 2706
...
….

After the process completed array will be:
|568, 607, 675, 743, 894, 1165, 1896, 2217, 2492, 2706

Radix Sort

Here is our array [607, 1896, 1165, 2217, 675, 2492, 2706, 894, 743, 568]

And here is our algorithm (which is given in the slides):

radixSort(int theArray[], in n:integer, in d:integer)
// sort n d-digit integers in the array theArray
    for (j=d down to 1) {
        Initialize 10 groups to empty
        Initialize a counter for each group to 0
        for (i=0 through n-1) {
            k = jth digit of theArray[i]
            Place theArray[i] at the end of group k

Increase kth counter by 1
}
Replace the items in theArray with all the items in
group 0, followed by all the items in group 1, and so on.
}

Now let us evaluate the array.

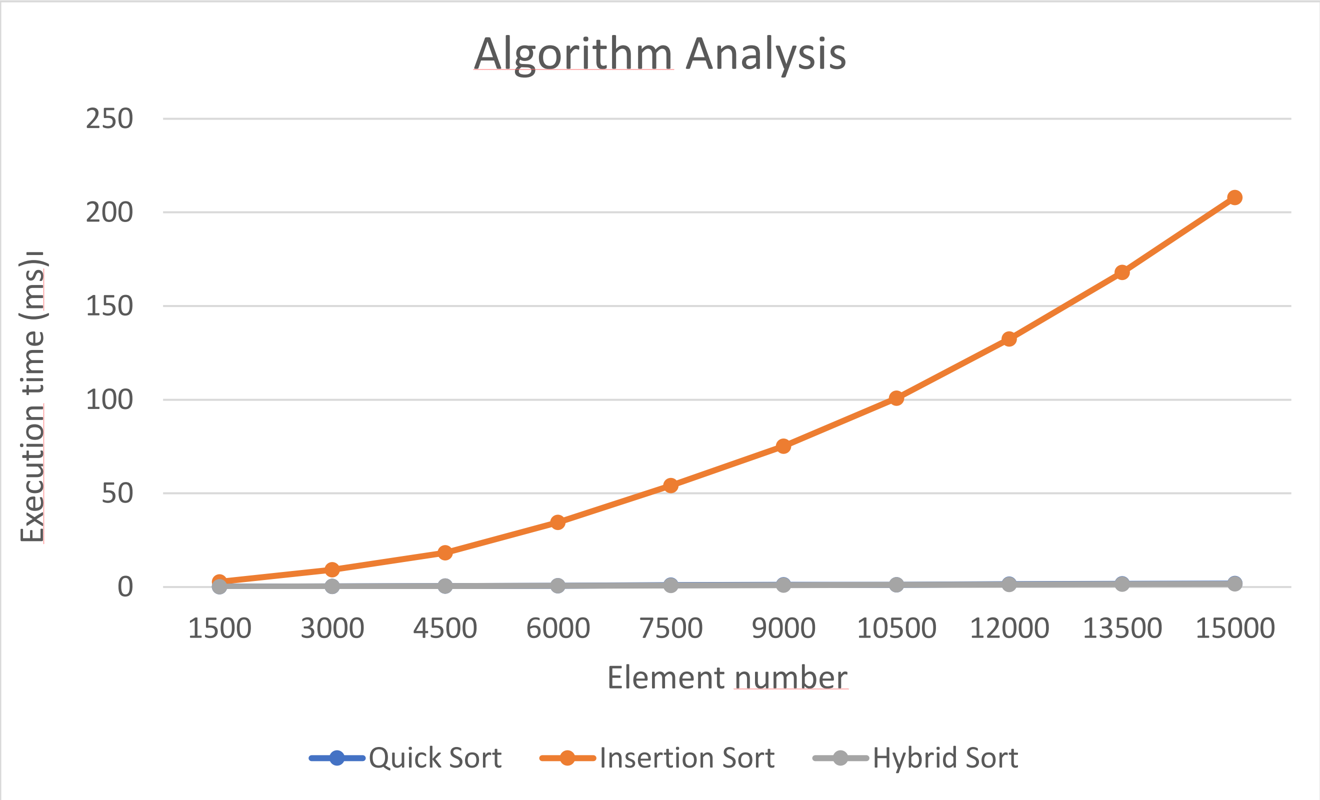
607, 1896, 1165, 2217, 675, 2492, 2706, 894, 743, 568.	Original integers
2492, 743, 894, (675, 1165), (1896, 2706), (607, 2217), 568	Grouped by fourth digit
2492, 743, 894, 675, 1165, 1896, 2706, 607, 2217, 568	Combined
(607, 2706), 743, 2217, (568, 1165), 675, (894, 1896, 2492)	Grouped by third digit
607, 2706, 743, 2217, 568, 1165, 675, 894, 1896, 2492	Combined
1165, 2217, 2492, 568, (607, 675), (743, 2706), (894, 1896)	Grouped by second digit
1165, 2217, 2492, 568, 607, 675, 743, 2706, 894, 1896	Combined
(568, 607, 675, 743, 894), (1165, 1896), (2217, 2492, 2706)	Grouped by first digit
568, 607, 675, 743, 894, 1165, 1896, 2217, 2492, 2706	Combined (sorted)

Question 2:

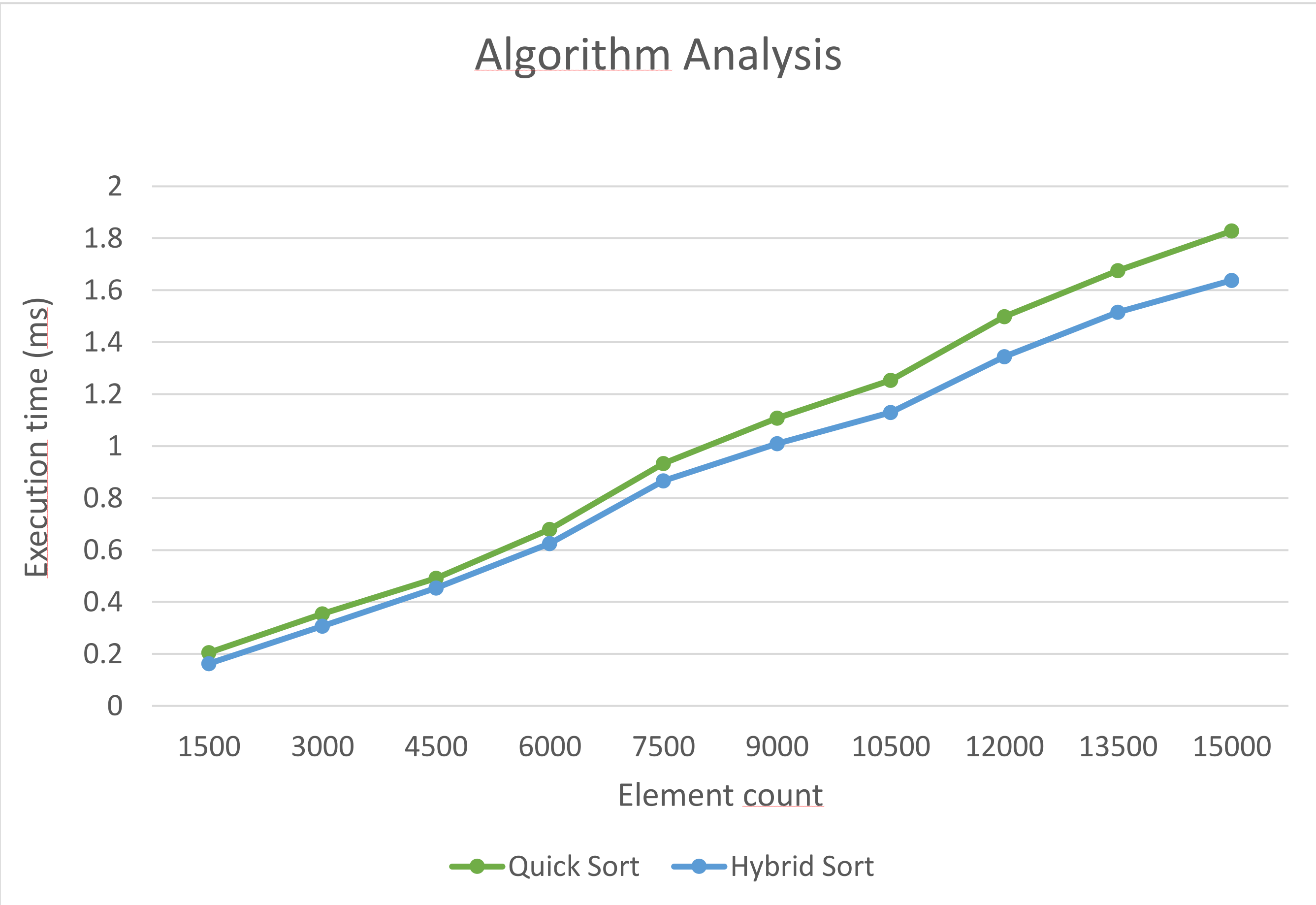
Here is screenshot of the result of my code:

```
-----------------------------------------------------
Part a - Time analysis of Quick Sort
Array Size    Time Elapsed(ms)    compCount    moveCount
1500           0.205               18017        28743
3000           0.355               38749        70459
4500           0.492               60426        88364
6000           0.68                83582        130445
7500           0.933               129073       201712
9000           1.108               150014       230759
10500          1.254               168810       229096
12000          1.498               211022       293960
13500          1.676               241556       328056
15000          1.828               269815       329581
-----------------------------------------------------
Part b - Time analysis of Insertion Sort
Array Size    Time Elapsed(ms)    compCount    moveCount
1500           2.783               558878       560377
3000           9.196               2246523      2249522
4500           18.273              5010113      5014612
6000           34.414              9104239      9110238
7500           54.178              13977338     13984837
9000           75.184              20221281     20230280
10500          100.874             27605983     27616482
12000          132.399             35978231     35990230
13500          167.875             45770826     45784325
15000          207.982             56816343     56831342
-----------------------------------------------------
Part c - Time analysis of Hybrid Sort
Array Size    Time Elapsed(ms)    compCount    moveCount
1500           0.163               17812        26143
3000           0.307               37914        65980
4500           0.454               58225        82132
6000           0.625               79386        122424
7500           0.866               121792       191731
9000           1.01                139161       218685
10500          1.13                153172       215037
12000          1.345               190451       277721
13500          1.516               216503       310145
15000          1.638               240417       310025
```

Question 3:



Algorithm Analysis
250
200
150
100
50
0
Execution time (ms)
1500
3000
4500
6000
7500
9000
10500
12000
13500
15000
Element number
Quick Sort
Insertion Sort
Hybrid Sort



Algorithm Analysis
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Execution time (ms)
1500
3000
4500
6000
7500
9000
10500
12000
13500
15000
Element count
Quick Sort
Hybrid Sort

Since there is huge difference between the running times of insertion sort and the other two, I had to use two different graphs in order to show them all.

From the graphs, we can see that insertion sort has the worst running time by far as expected.

It may be useful for very small arrays like an array has 10 elements. But it is not useful for arrays has 1500 elements or more. Theoretically insertion sort is $O(n^2)$. The graph also supports this.

Both quick sort and hybrid sort are recursive. They use divide and conquer principle. But we see that Hybrid Sort is little bit more efficient than quick sort. Because it uses insertion sort for small arrays and as I already mentioned insertion sort can be efficient for small arrays.

Asymptotic notation of quick sort depends. It depends on order of the array and pivot selection. Quick sort is $O(n^2)$ in worst case and O(nlogn) in best and average case. Worst case has low possibility. In the graph we see average case.

We can say the same things for hybrid sort. Only difference is hybrid sort uses insertion sort for small arrays and that provides an advantage for itself. Because recursion can be inefficient for small inputs.