CS202 – HW1 REPORT

QUESTIONS 1 AND 3

Olcay Akman

21702671

Question 1

a) The given functions sorted in increasing order of their asymptotic complexity is as follows:

$(f_5(n) = n^{1/\log n}) < f_4(n) = \log n < f_{10}(n) = n^{1/2} < (f_9(n) = \log(n!)) = (f_6(n) = n\log n) < f_8(n) = n^3 < f_2(n) = n^{\log(\log n)} < f_7(n) = e^n < f_1(n) = 10^n < f_3(n) = n!$

Because:

$f_1(n) = 10^n = O(10^n)$

$f_2(n) = n^{\log(\log n)} = O(n^{\log(\log n)})$

$f_3(n) = n! = O(n!)$

$f_4(n) = \log n = O(\log n)$

$f_5(n) = n^{1/\log n} = O(1)$

$f_6(n) = n\log n = O(n\log n)$

$f_7(n) = e^n = O(e^n)$

$f_8(n) = n^3 = O(n^3)$

$f_9(n) = \log(n!) = O(n\log n)$

$f_{10}(n) = n^{1/2} = O(n^{1/2})$

b) The average processing time T(n) of the following algorithm where random(n) takes Θ(1) time should be as follows:

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

}
```

It takes constant time to find a random number and assign it to i. It also takes constant time to do the comparison in the if statement. It takes constant time to “return 0;”. Thus, all these can be combined to be represented as $\Theta(1)$ in our final product for $T(n)$. Now only the return statement of the “else” part remains unresolved, in which the function itself is called again, resulting in recursion.

In order to calculate the average processing time, we need to deduce the value of i on each iteration of the function. On average, i should be $i = \frac{(n+0)}{2}$.

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2} - 1\right) + \Theta(1)$$

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{4} - 1\right) + T\left(\frac{n}{4} - \frac{1}{2}\right) + T\left(\frac{n}{4} - \frac{3}{2}\right) + 2\Theta(1)$$

$$T(n) = T\left(\frac{n}{8}\right) + T\left(\frac{n}{8} - \frac{1}{4}\right) + 2T\left(\frac{n}{8} - \frac{3}{4}\right) + 2T\left(\frac{n}{8} - 1\right) + T\left(\frac{n}{8} - \frac{5}{4}\right) + T\left(\frac{n}{8} - \frac{7}{4}\right) + 6\Theta(1)$$

.

.

.

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

c) The array to be sorted is:

[607, 1896, 1165, 2217, 675, 2492, 2706, 894, 743, 568]

Let's first apply Bubble sort to it:

Pass 1:

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

Pass 2:

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

607	1165	675	1896	2217	894	743	568	2492	2706

Pass 3:

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

Pass 4:

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

607	675	1165	894	743	568	1896	2217	2492	2706

Pass 5:

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

607	675	894	743	568	1165	1896	2217	2492	2706

Pass 6:

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

607	675	743	568	894	1165	1896	2217	2492	2706

Pass 7:

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

607	675	568	743	894	1165	1896	2217	2492	2706

Pass 8:

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

607	568	675	743	894	1165	1896	2217	2492	2706

Pass 9:

568	607	675	743	894	1165	1896	2217	2492	2706

568	607	675	743	894	1165	1896	2217	2492	2706

The sorted array:

568	607	675	743	894	1165	1896	2217	2492	2706

Similarly, the initial array will follow the following steps if we apply radix sort on it:

Sorting by 1st digits:

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

0607, 1896, 1165, 2217, 0675, 2492, 2706, 0894, 0743, 0568

(249**2**) (074**3**) (089**4**) (116**5**, 067**5**) (189**6**, 270**6**) (060**7**, 221**7**) (056**8**)

2492, 0743, 0894, 1165, 0675, 1896, 2706, 0607, 2217, 0568

Sorting by 2nd digits:

2492, 0743, 0894, 1165, 0675, 1896, 2706, 0607, 2217, 0568

(27**0**6, 06**0**7) (22**1**7) (07**4**3) (11**6**5, 05**6**8) (06**7**5) (24**9**2, 08**9**4, 18**9**6)

2706, 0607, 2217, 0743, 1165, 0568, 0675, 2492, 0894, 1896

Sorting by 3rd digits:

2706, 0607, 2217, 0743, 1165, 0568, 0675, 2492, 0894, 1896

(1**1**65) (2**2**17) (2**4**92) (0**5**68) (0**6**07, 0**6**75) (2**7**06, 0**7**43) (0**8**94, 1**8**96)

1165, 2217, 2492, 0568, 0607, 0675, 2706, 0743, 0894, 1896

Sorting by 4th digits:

1165, 2217, 2492, 0568, 0607, 0675, 2706, 0743, 0894, 1896

(**0**568, **0**607, **0**675, **0**743, **0**894) (**1**165, **1**896) (**2**217, **2**492, **2**706)

0568, 0607, 0675, 0743, 0894, 1165, 1896, 2217, 2492, 2706

The sorted array after radix sort:

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

Question 3

The elapsed time of each sorting algorithm varies from one run to another and it varies from device to device. In order to calculate the elapsed times of each sorting algorithm, in the performanceAnalysis method of question 2, a clock function from the C++ standard library was used, so that the elapsed times could be calculated with regards to the time of the computer, thus the variations from device to device. The variations from run to run in the same device depend on the processor of the computer being used at that moment.

After having run my code in my Windows machine and on the Dijkstra server, I have obtained fairly different results. I decided to rely on the physical machine at hand, that is my computer, to plot my graph. Below are the screenshots of what I have obtained from both the Dijkstra compiler and the compiler on my Windows machine.

Windows machine:

Part a - Time analysis of Quick Sort

Array Size	Time Elapsed	compCount	moveCount
1500	2 ms	18314	31351
3000	4 ms	40642	68620
4500	4 ms	67047	95953
6000	5 ms	95560	145690
7500	4 ms	124938	168246
9000	4 ms	159286	219351
10500	5 ms	189738	271625
12000	5 ms	229521	308302
13500	5 ms	260832	314667
15000	6 ms	315396	388247

Part b - Time analysis of Insertion Sort

Array Size	Time Elapsed	compCount	moveCount
1500	6 ms	564759	567757
3000	22 ms	2186570	2192568
4500	44 ms	4957027	4966025
6000	39 ms	8901970	8913968
7500	51 ms	13914843	13929841
9000	79 ms	20206572	20224570
10500	103 ms	27508861	27529859
12000	139 ms	36009265	36033263
13500	179 ms	45448401	45475399
15000	212 ms	56124693	56154691

Part c - Time analysis of Hybrid Sort

Array Size	Time Elapsed	compCount	moveCount
1500	0 ms	18314	31351
3000	2 ms	40642	68620
4500	2 ms	67047	95953
6000	1 ms	95560	145690
7500	2 ms	124938	168246
9000	2 ms	159286	219351
10500	2 ms	189738	271625
12000	3 ms	229521	308302
13500	3 ms	260832	314667
15000	3 ms	315396	388247

Dijkstra:

Part a - Time analysis of Quick Sort

Array Size	Time Elapsed	compCount	moveCount
1500	0 ms	17705	25676
3000	0 ms	39083	65402
4500	0 ms	65518	90636
6000	0 ms	94293	141613
7500	0 ms	118871	161316
9000	0 ms	155242	211888
10500	0 ms	184965	206978
12000	0 ms	217200	260081
13500	0 ms	265542	342701
15000	10 ms	291957	322835

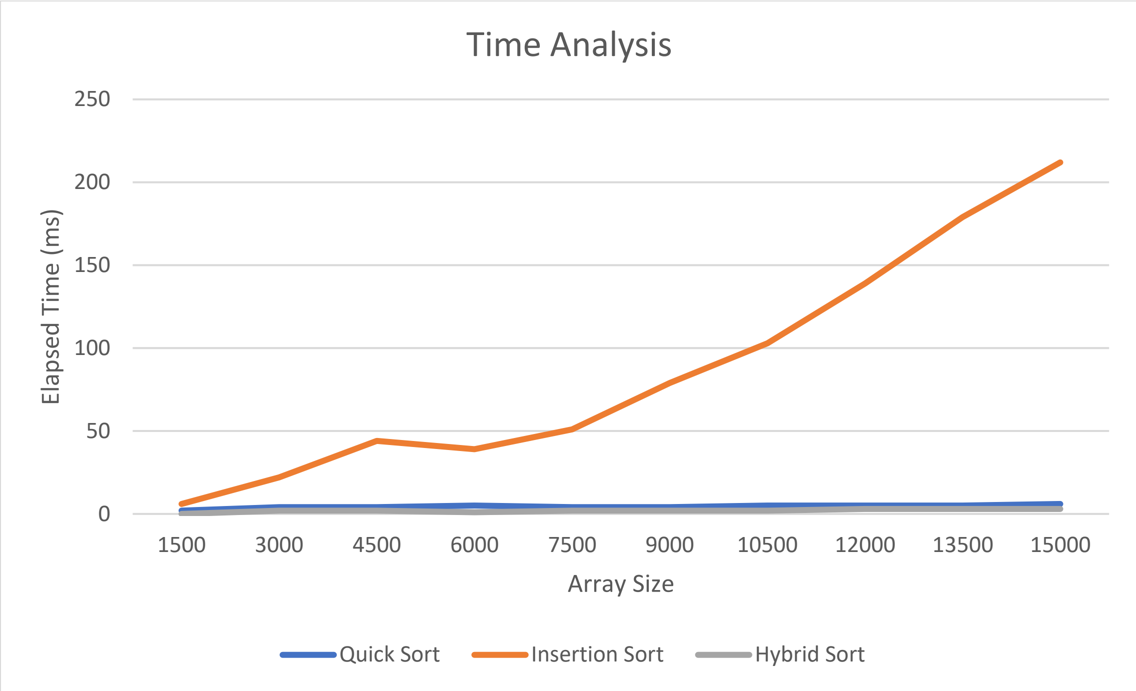
Part b - Time analysis of Insertion Sort

Array Size	Time Elapsed	compCount	moveCount
1500	0 ms	555406	558404
3000	20 ms	2248470	2254468
4500	30 ms	5021720	5030718
6000	60 ms	8938152	8950150
7500	90 ms	13995218	14010216
9000	130 ms	20326194	20344192
10500	180 ms	27571428	27592426
12000	230 ms	36044252	36068250
13500	290 ms	45353815	45380813
15000	350 ms	56069281	56099279

Part c - Time analysis of Hybrid Sort

Array Size	Time Elapsed	compCount	moveCount
1500	0 ms	17705	25676
3000	0 ms	39083	65402
4500	0 ms	65518	90636
6000	0 ms	94293	141613
7500	0 ms	118871	161316
9000	0 ms	155242	211888
10500	0 ms	184965	206978
12000	10 ms	217200	260081
13500	0 ms	265542	342701
15000	10 ms	291957	322835

And below is my graph containing the data from the Windows machine:



Although rather invisible here, the quick sort and the hybrid sort functions take relatively similar times to be executed, whereas the insertion sort takes much longer to do so. Hybrid sort is the most efficient of the three for it utilizes both the quick sort and the insertion sort wherever they are swifter. That is, quick sort is fast on larger amounts of data whereas insertion sort is faster on smaller amounts of data. Since hybrid sort applies quick sort when array size is large, and uses insertion sort when array size is small, it is the most efficient of the three.