Name: Drishti Maharjan

Algorithm and Data Structure

Assignment 2

Problem 2.1

a) Code is in mergesort_V3F.c

b) &

c)

Graphs:

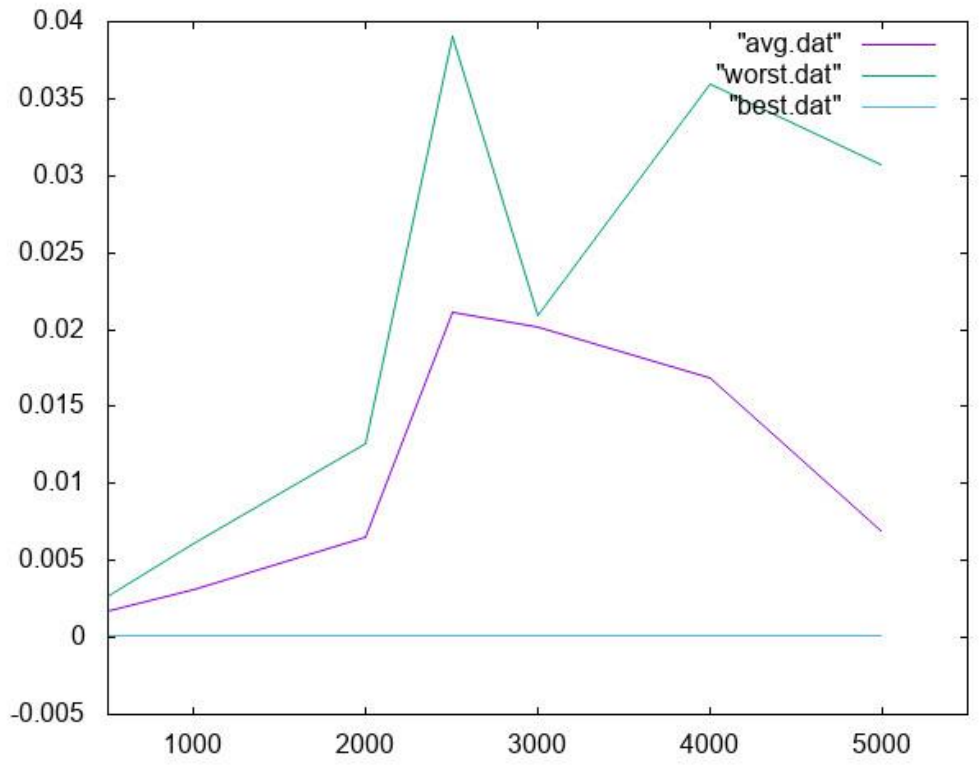


Figure 1 x-axis: k, y-axis: time

In figure 1, we take different values for k for n=5000. The top most line/graph in figure 1 (green) represents worst case, the purple graph represents average case, and the blue graph(lowermost) represents best case. Time for average case and worst case are lesser when k is less, and higher with higher values of k. For best case, the time gradually decreases when k increases from 1 to 5000, and since the range of k(1-5000) is very high in this plot, and I have taken a rough number of data(not for many k's), and joined the points with straight lines, it almost looks like a linear decreasing curve(even though the real case might be slightly different). However, the main idea here is that for average and worst case, when we have smaller k's, the code implementation is almost equivalent to merge-sort. And, for higher k's the program is almost equivalent to insertion sort. From time complexities of merge sort and insertion sort, we know that merge sort performs better for larger n. Here, n=5000 is a pretty large number, and the performance speed of merge sort being better than insertion sort for worst and average case is demonstrated. However, for the best case, the program performs better for higher values of k , which implies insertion sort performs a bit better than merge sort for already sorted numbers(even though the time difference is very minimal in terms of seconds).

{Choice of k-explained further in number d.}

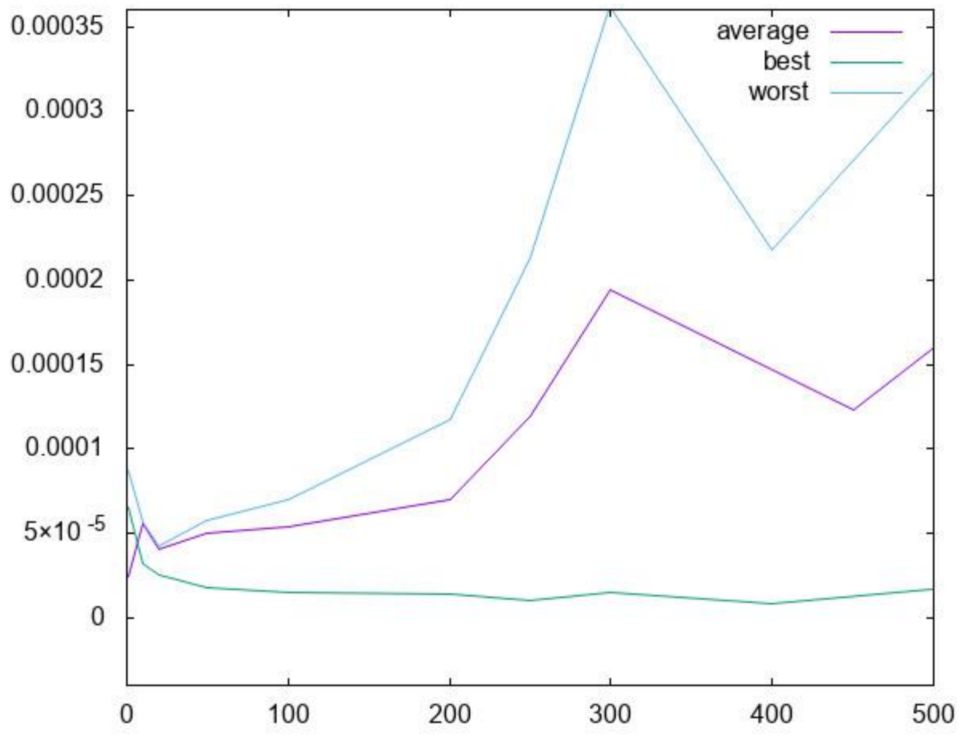


Figure 2 x-axis:k, y-axis: time

In Figure 2, we take different values of k for n=500. Here n is a smaller value than the previous graph, but still not a trivial number. We see similar observations as in the 1st Figure, except for the best case, which doesn't look as linear as the 1st Figure. This implies that, for larger values of n, choice of k doesn't really make a huge difference for best cases.. This only noticeable difference in Figure 1 and 2 concludes that the effectiveness of insertion sort relative to merge sort for best cases is significantly seen for small values of n. The main conclusion being the same that for average and worst cases, choosing smaller values of k makes the program relatively more efficient

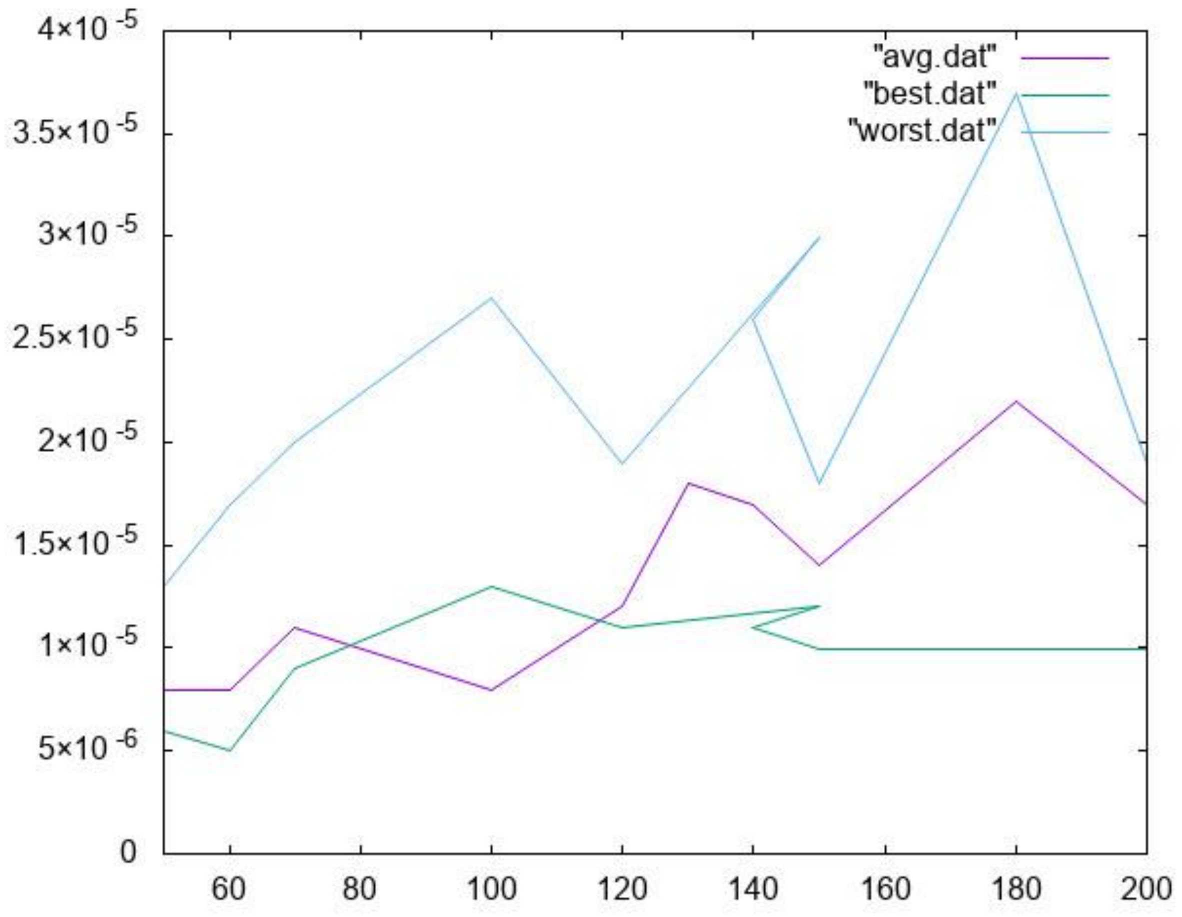


Figure 3 x-axis: n, y-axis: time

In Figure 3, we take different values of n(from 50-200) for k = 40. Here, x axis represents size of array (i.e. n), and y axis represents time computation. For smaller values of n, taking k =40 means the program is almost equivalent to insertion sort, and for larger values of n, taking k=40(a trivial number) means the program is almost equivalent to merge sort. The random peaks and fluctuations show how the same choice of k may perform differently for different values of n. For example, for larger n, taking the most trivial value of k may not be the most efficient idea, but taking a value k at somewhere in the right most range (yet not the most trivial k) results in a faster performance. However, limitations of this figure includes that the randomly generated numbers may not result in the most precise graphs, as we can see at some points, average case graph takes unusual turns.

d) From the analysis of these 3 figures, we can conclude that choice of k can wisely be chosen according to the input size and nature. For example, in Figure 1, values of k between 1 to 1000 (for n = 5000), and in Figure 2, k = 1 to 100 (for n = 500) would be a good choice for unsorted numbers, and for arrays that are almost sorted, taking values of k closer to n would make most sense. Generalizing it, I would say values between 1 and 1/5th of the input array size would give better performance when we want sorted list from a totally unsorted list.

Problem 2.2

a) $T(n) = 36\,T\left(\frac{n}{6}\right) + 2n$

$\Rightarrow$ Master method:

$a = 36 \quad b = 6$

$n^{\log_6 36} = n^2$

Here, $f(n) = 2n$

$= 2n^{\log_6 6}$

$= 2n^{(\log_6 36 - \log_6 6)}$ consider $\epsilon = \log_6 6$

As ϵ is > 0 and $2n \in O(n^{\log_6 36})$, it implies $f(n)$ is polynomially smaller than $n^{\log_6 36}$. Hence, by case 1 $T(n) = \Theta(~~n \log~~ n^{\log_6 36}) = \Theta(n^2)$

b) $T(n) = 5T\left(\frac{n}{3}\right) + 17n^{1.2}$

$\Rightarrow$ Master Method:

$a = 5 \quad b = 3$

$n^{\log_3 5} = n^{1.46}$

$f(n) = 17n^{1.2}$

$= 17n^{1.46 - 1.26}$ where $\epsilon = 1.26$

As $\epsilon > 0$ & $17n^{1.2} \in O(n^{\log_3 5})$, it implies $f(n)$ is polynomially smaller than $n^{\log_3 5}$. Hence, by case 1, $T(n) = \Theta(n^{\log_3 5})$

c> $T(n) = 12\,T\left(\frac{n}{2}\right) + n^2 \lg n$

$\Rightarrow f(n) = n^2 . \lg n$

Here, $a = 12$ $b = 2$

$n^{\log_2 12} = n^{3.58}$

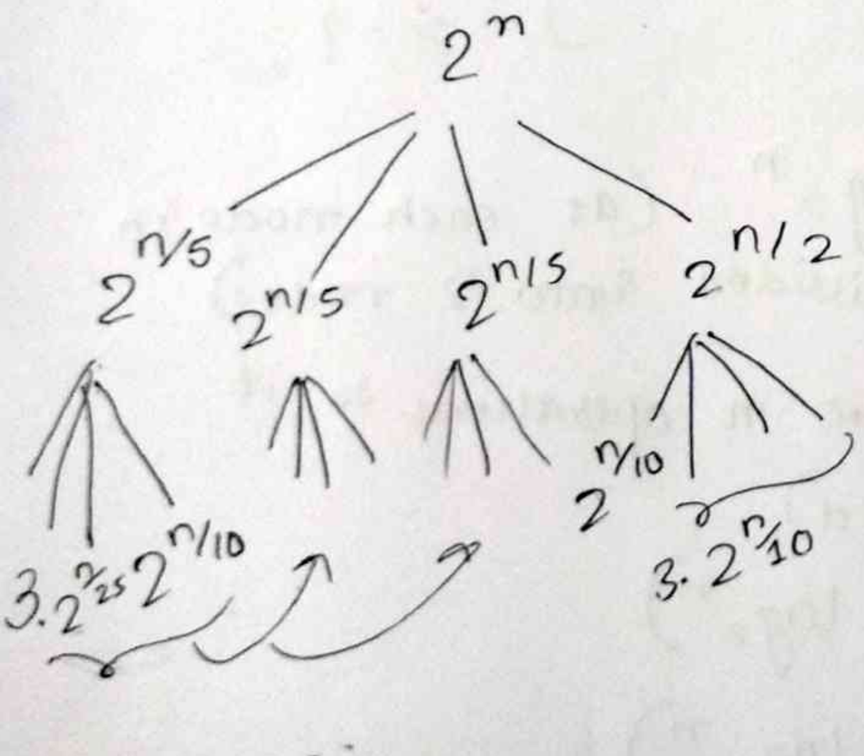
$f(n) = O(n^{3.58 - 1.58})$ where $\epsilon = 1.58$

So, $f(n) \in O(n^{\log_b a - \epsilon})$ in this case.

Hence by case I, $T(n) = \Theta(n^{\log_2 12})$

d> $T(n) = 3T\left(\frac{n}{5}\right) + T\left(\frac{n}{2}\right) + 2^n$

$\Rightarrow$ Recursion Tree



$\Rightarrow 2^n$

$\Rightarrow 3 \cdot 2^{n/5} + 2^{n/2}$

$= 2^n\left(3 \cdot 2^{1/5} + 2^{1/2}\right)$

$\Rightarrow 2^n\left(9 \cdot 2^{1/5^2} + 3 \cdot 2^{\frac{1}{5\cdot}} + 2^{1/4}\right)$

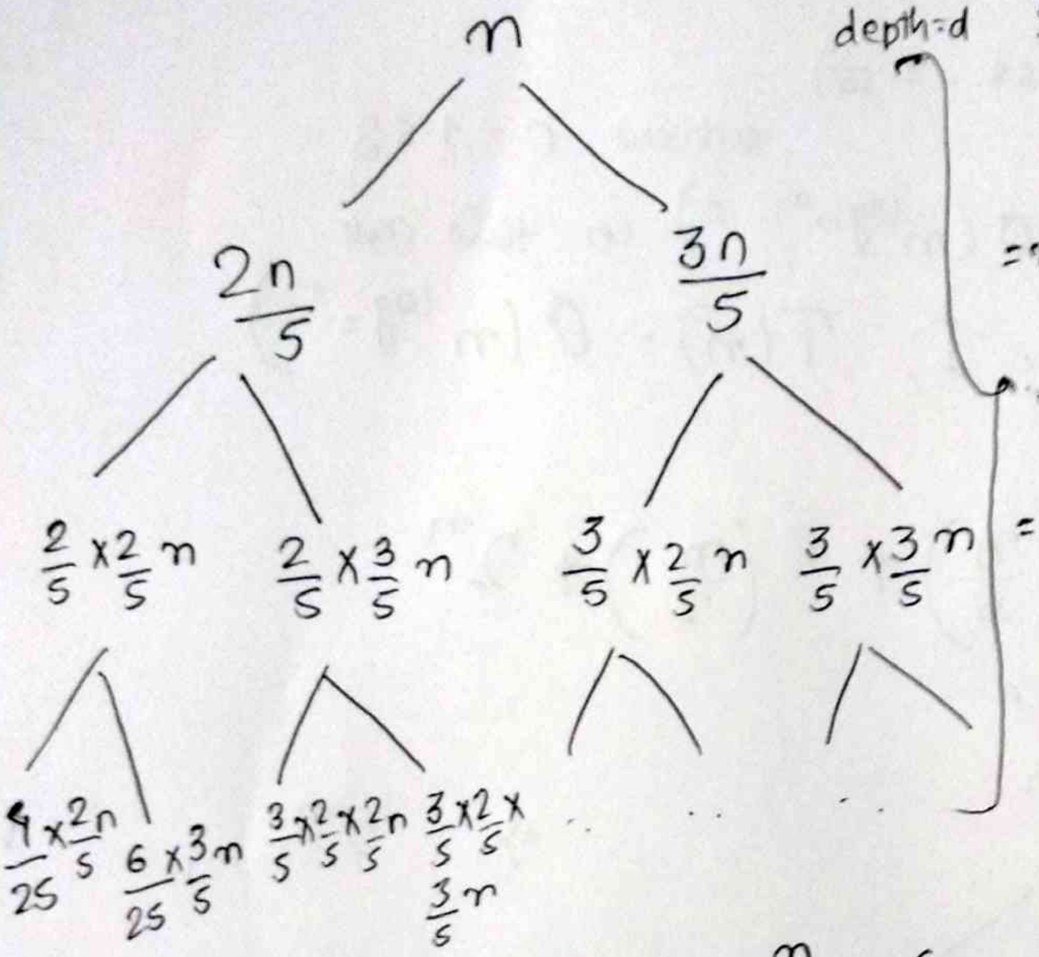
we can see that in each level, 2^n tends to dominate the rest of the sum. Hence, we can safely say that $\Sigma = \Theta(2^n)$.

Therefore, $T = \Theta(2^n)$.

e) ● Recursion Tree

$\Theta(n)$

Let's model tree in form of n. Sum in form of $\Theta(n)$


n
depth=d
$\frac{2n}{5}$
$\frac{3n}{5}$
$\frac{2}{5} \times \frac{2}{5} n$
$\frac{2}{5} \times \frac{3}{5} n$
$\frac{3}{5} \times \frac{2}{5} n$
$\frac{3}{5} \times \frac{3}{5} n$
$\frac{4}{25} \times \frac{2n}{5}$
$\frac{6}{25} \times \frac{3n}{5}$
$\frac{3}{5} \times \frac{2}{5} \times \frac{2n}{5}$
$\frac{3}{5} \times \frac{2}{5} \times \frac{3n}{5}$
...
...

$\Rightarrow n \Rightarrow \Theta(n)$

$= n\left(\frac{2}{5} + \frac{3}{5}\right) \Rightarrow \Theta(n)$

$= n\left(\frac{4}{25} + \frac{6}{25} + \frac{6}{25} + \frac{9}{25}\right)$
$\Rightarrow \Theta(n)$

$\Sigma = ?$

Here, depth $= d = \log_2{n}$ (as each node in each level is ~~a~~ sub-divided into 2 nodes).

At each level there are n operations. So, it implies that $\Sigma = \Theta(n \cdot d)$
$= \Theta(n \cdot \log_2{n})$

Hence, $T = \Theta(n \cdot \log_2{n})$