Sangyun Lee

HW2

Q1

1-a. $T(n) = b \cdot T(n-1) + 1$ where b is a fixed positive integer greater than 1. Let's assume b = 2

$= T(n) = 2T(n-1) + 1,$

= then with muster method, a=2, b=1, d = 1 because f(n) = c, since a > 1

$= T(n) = \theta(n^0 * 2^{n/1}) = \theta(2^n)$

= Therefore, $T(n) = \theta(b^n)$

1-b. $T(n) = 3 \cdot T(n/9) + n \cdot \log n$; using master theorem, a=3, b=9, f(n)=nlogn

$= \log_9 3 = ½$, therefore, $n^{\log b(a)} = n^{1/2}$

$= f(n) = \Omega(n^{1/2 + \varepsilon})$, This case 3, so we need to verify regularity cond.

= a f(n/b) <= c f(n)

= 3 (n/9)log(n/9) <= 1/3 n log n = c*f(n), c= 1/3

$= T(n) = \theta(n \log n)$

Q2

Karatsuba_mulitification(x, y)

If (size of x or y == 1)

return x*y

Else

XL = Left half of the value X , XR = right half of the value x

YL = Left half of the value Y, YR = Left half of the value Y

P1 = Karatsuba_mulitification(XL, YL)

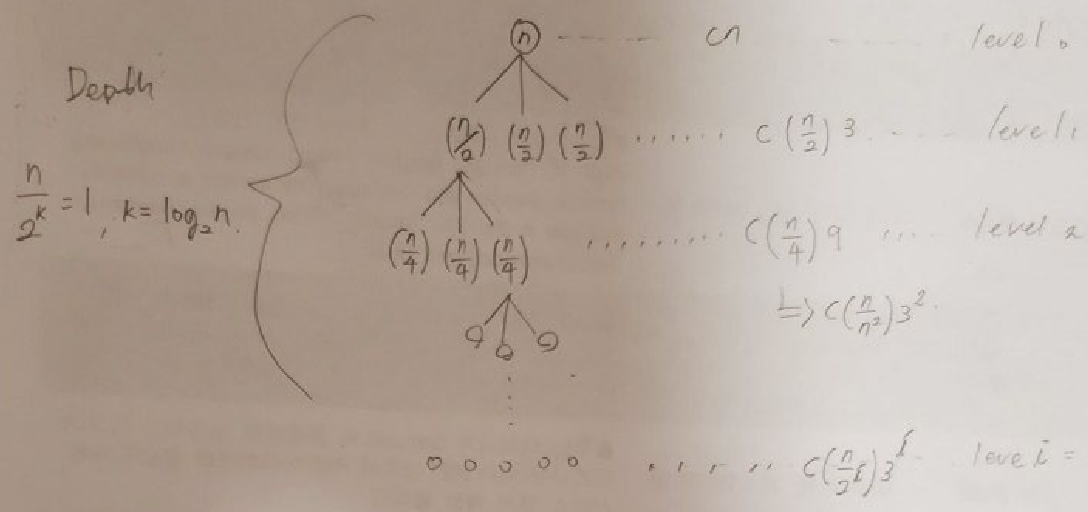
P2 = Karatsuba_mulitification(XL + XR, YL + YR)

P3 = Karatsuba_mulitification(XR, YR)

Return P1 * 2^n + (P2 – P1 – P3) * 2^n/2 + P3

Q3

3-a



there for $\sum_{i=0}^{\log_2 n} c\left(\frac{n}{2^i}\right)3^i$

$$\Rightarrow cn \cdot \sum_{i=0}^{\log_2 n}\left(\frac{3}{2}\right)^i = \Theta\left(n\left(\frac{3}{2}\right)^{\log_2 n}\right)$$

$$= \Theta\left(n\,\frac{3^{\log_2 n}}{2^{\log_2 n}}\right) = \Theta\left(3^{\log_2 n}\right)$$

$$= \Theta\left(\left(\underbrace{3^{\log_3 n}}_{n}\right)^{\log_2 3}\right) = \Theta\left(n^{\log_2 3}\right)$$

$$\frac{n\left(\left(\frac{3}{2}\right)^{\log_2 n} - 1\right)}{\frac{3}{2} - 1} = \frac{n\left(\left(\frac{3}{2}\right)^{\log_2 n} - 1\right)}{\frac{1}{2}} = 2 \cdot n \cdot \left(\left(\frac{3}{2}\right)^{\log_2 n} - 1\right)$$

$$= 2 \cdot n \cdot \left(n^{\log_2 \frac{3}{2}} - 1\right)$$

$$= 2 \cdot \left(n \cdot n^{\log_2 3 - \log_2 2} - n\right)$$

$$= 2 \cdot \left(n^{\log_2 3 - 1 + 1} - n\right)$$

$$= 2 \cdot n^{\log_2 3} - 2n$$

$$= \Theta\left(n^{\log_2 3}\right)$$

- It looks a little bit blurry, so I write here again. $= \Theta(n^{\log_2(3)})$

3-b

$$T(2^{i+1}) = \left(3T\left(\frac{2^i}{2}\right) + 2^i\right) \times 2$$

$$= 3T\left(\frac{2^{i+\cancel{1}}}{\cancel{2}}\right) + 2^{i+1}$$

$$= 3\underline{T(2^i)} + 2^{i+1}$$

$$= 3 \cdot (2^i)^{\log_2 3} + 2^{i+1}$$

$$= (2^i \times 2)^{\log_2 3} + 2^{i+1}$$

$$= (2^{i+1})^{\log_2 3}$$

Q4

4-a.

Let say

badSort(A[0 ⋯ m − 1]) = r_function_a

badSort(A[n – m ⋯ n − 1]) = r_function_b

badSort(A[0 ⋯ m − 1]) = r_function_c

This sorting algorithms compares the first elements with the last elements of the array. So, if the array is divided by 2, the last r_function_c conducts the sorting process for the sorted array which is already sorted by r_function_a. So, it jus simply repeat the process which is done already. Therefore, array wasn't sorted even after the r_function_a, then r_function_c would not sort the array because there is no overlapped elements.

4-b.

If I keep the ceiling of alpha value as ¾, then it falls in infinity loop when array size == 3. Because 3 * ¾ = 3 if we round up the value. So, if change it to floor. Then it works.

4.c

a = 3 since it has 3 recursive calls

b = 2/3 since it is divided array size of 2/3

There for the recurrence is T(n) = 3 * T($\frac{2}{3}n$) + Θ(1);

4.d

a = 3, b = 3/2 n^log_1.5(3) = n^2.7

f(n) = cn^0.

This is case 1, therefore, t(n) = Θ(n^2.7);

Q5

5.a

Code is submitted on TEACH

5.b

For the 5.a, I used vector since vector is easy to implement for 2d array since output should be save in single text file. Here I used single dynamic array. And this code will be submitted on TEACH as well.

```
int main()

{

   srand(time(NULL));


   double twoThree = 2.0 / 3.0;

   double threeFour = 3.0 / 4.0;


   //Sort and get time duration with alpha value of 2/3

   int count = 7;

   int arrSize = 10;

   std::cout << "alpha value: 2/3" << std::endl;


   while (count >= 1) {


      //repeatedly define single dynamic array

      int* arrSort = new int[arrSize];

      for (int i = 0; i < arrSize; i++)

      {

         arrSort[i] = rand() % arrSize;

      }


      //sort and measure the execution time

      auto start = high_resolution_clock::now();              //set time start

      badsort(arrSort, 0, arrSize - 1, twoThree);

      auto stop = high_resolution_clock::now();               //set time stop

      auto duration = duration_cast<microseconds>(stop - start);
```

```
        //display result

        std::cout << "Array size: " << arrSize << " / time taken by function: " << duration.count() << "
microseconds" << std::endl;

        arrSize = arrSize * 2;

        count--;

    }


    //Sort and get time duration with alpha value of 3/4

    count = 7;

    arrSize = 10;

    std::cout << "alpha value: 3/4" << std::endl;


    while (count >= 1) {


        //repeatedly define single dynamic array

        int* arrSort = new int[arrSize];


        for (int i = 0; i < arrSize; i++)

        {

            arrSort[i] = rand() % arrSize;                        //since it doesn't need to save to file, simply use single
dynamic array

        }


        //sort and measure the execution time

        auto start = high_resolution_clock::now();                //set time start

        badsort(arrSort, 0, arrSize - 1, threeFour);

        auto stop = high_resolution_clock::now();                 //set time stop
```

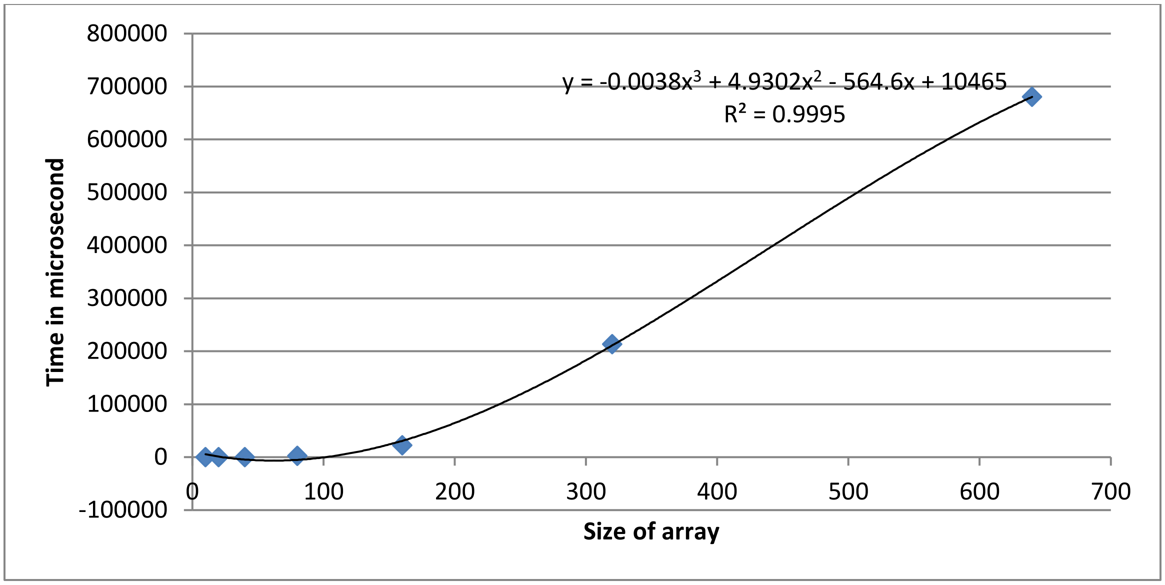
```
        auto duration = duration_cast<microseconds>(stop - start);


        //display result

        std::cout << "Array size: " << arrSize << " / time taken by function: " << duration.count() << "
microseconds" << std::endl;

        arrSize = arrSize * 2;

        count--;

    }


    return 0;

}
```

5.c

*Time was measured on local computer not school server for this answer.

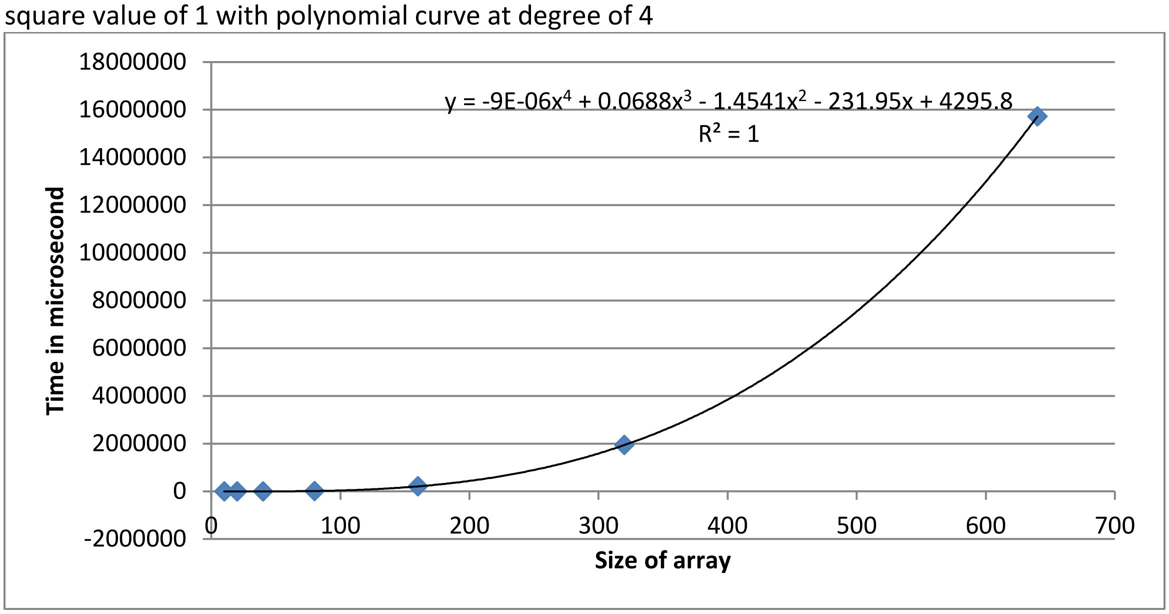
alpha value	0.666667	Try1	Try2	Try3	average	
Array size	10	10	29	15	18	microsecond
Array size	20	102	103	109	104.6667	microsecond
Array size	40	362	314	323	333	microsecond
Array size	80	3211	3005	2091	2769	microsecond
Array size	160	24608	18918	24137	22554.33	microsecond
Array size	320	209571	216245	214055	213290.3	microsecond
Array size	640	646275	646891	747883	680349.7	microsecond

For the alpha value of 2/3, it fit better with polynomial curve at degree of 3. I guess the theoretical t(n) = n^2.71 is close to n^3. That is why it has better r square value of 0.9995 with polynomial curve at degree of 3



alpha value	0.75	Try1	Try2	Try3	average	
Array size	10	10	7	10	9	microsecond
Array size	20	103	101	103	102.3333	microsecond
Array size	40	905	1073	886	954.6667	microsecond
Array size	80	8333	7993	7694	8006.667	microsecond
Array size	160	201933	209715	208237	206628.3	microsecond
Array size	320	1916390	1916719	1979688	1937599	microsecond
Array size	640	17051672	17127682	12978408	15719254	microsecond

For the alpha value of 3/4, it fit better with polynomial curve at degree of 4. The experimental f(n) is well fitted with theoretical f(n) = n^4.18, and it is close That is why it has better r square value of 1 with polynomial curve at degree of 4



5.d

I plotted the chart with array size from 10, and it grows by 2times. As I expected, badsort algorithm with alpha value of 2/3 had shown better performance than alpha value of 3/4.