Sangyun Lee

HW2

**Q1**

1-a. *T*(*n*) = *b* · *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) = θ(n0\*2n/1) = θ(2n)

= Therefore, T(n) = θ(bn)

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

=log93 = ½, therefore, nlogb(a) = n1/2

= f(n) = Ω(n1/2 + ε), 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) = θ(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)

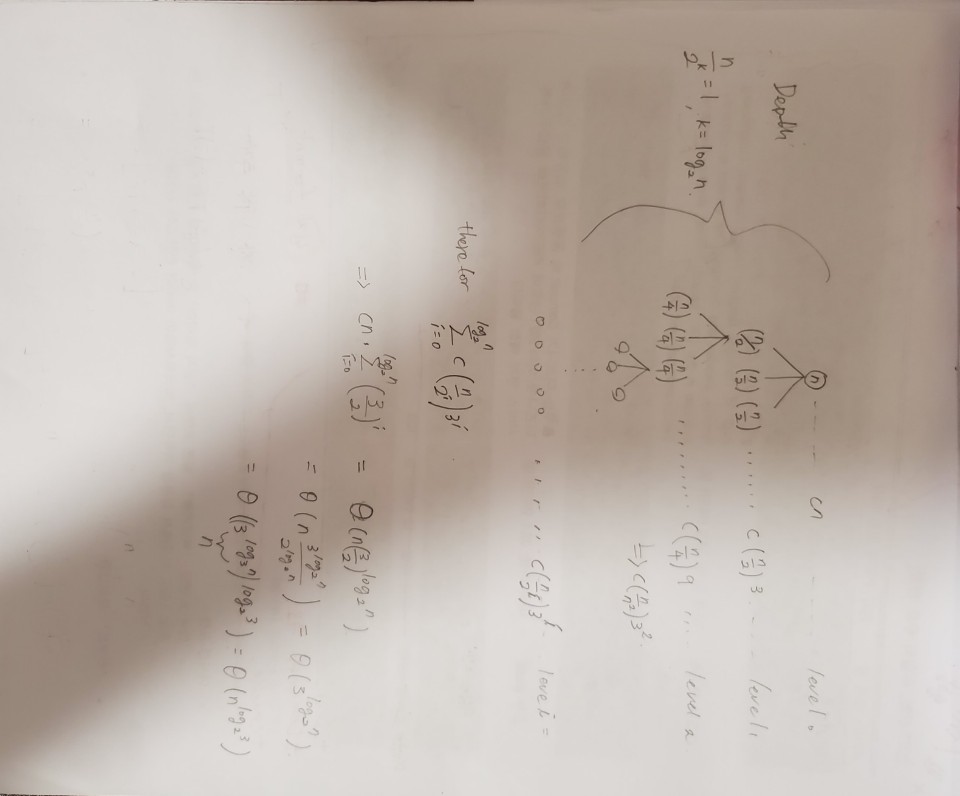
P2 = Karatsuba\_mulitification(XL + XR, YL + YR)

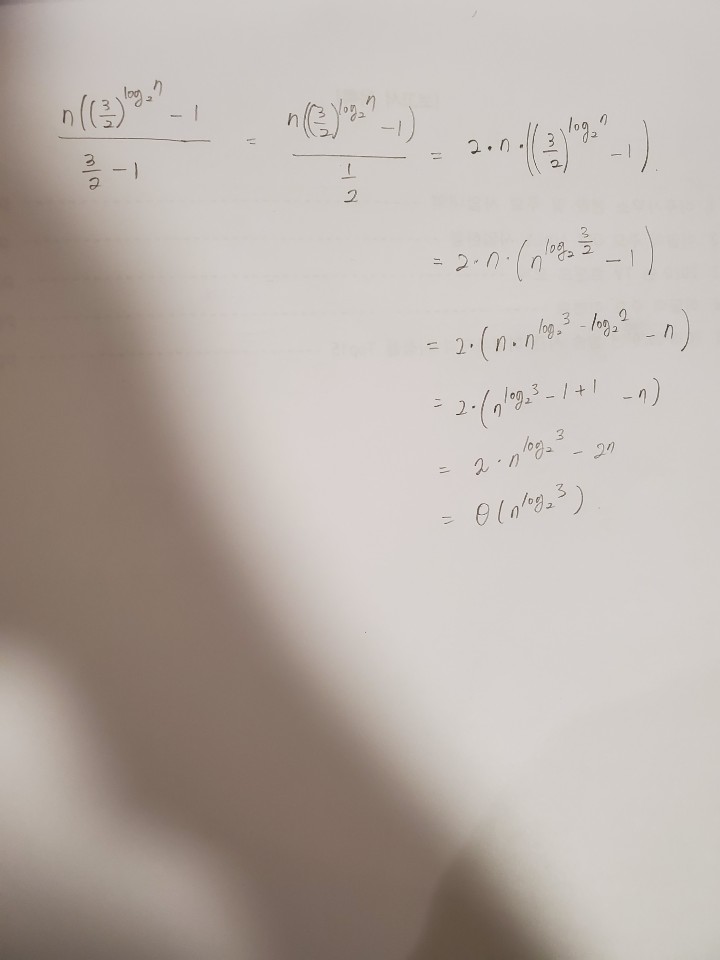
P3 = Karatsuba\_mulitification(XR, YR)

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

**Q3**

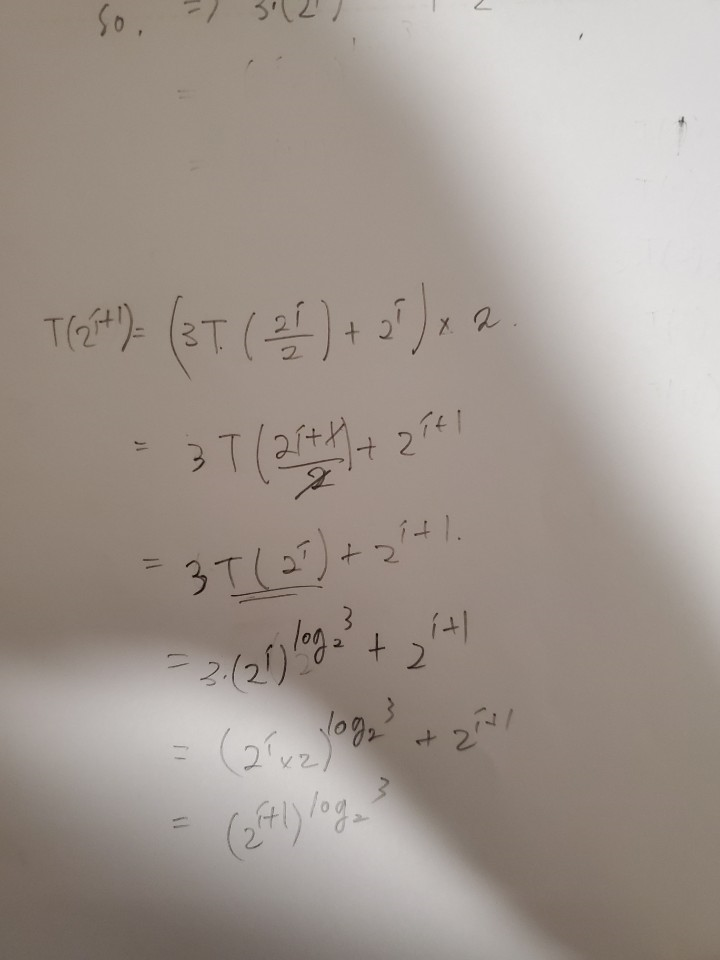
**3-a**

****

****

* It looks a little bit blurry, so I write here again. = Θ(nlog\_2(3))

**3-b**



**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( + 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.



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



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.

