DATA STRUCTURES AND ALGORITHMS HOMEWORK #2

MERVE HORUZ
18010426451

1)

a $\rightarrow \log_2 n^2 + 1 \Rightarrow 2\log_2 n + 1 \rightarrow \log_2 n \Rightarrow O(\log_2 n)$

- Because of the notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time and n bigger than $\log_2 n$, the notation $O(n)$ is true for this statement.

b $\rightarrow \sqrt{n(n+1)} = \sqrt{n^2+n} \rightarrow n^2 > n$ we evaluate according to n^2.

- since n is obtained from $\sqrt{n^2}$, the expression is simplified to n. The notation $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time.
so $\Omega(n)$ is true for this statement,

c $\rightarrow n^{n-1} = \Theta(n^n)$ $\lim\limits_{n\to\infty} \frac{n^{n-1}}{n^n} = \frac{1}{n} \Rightarrow 0$

- The notation $\Theta(n)$ is the formal way to express both the lower bound and the upper bound of an algorithm's running time, so $\Theta(n^n)$ is false for this statement.

2) $\lim\limits_{n\to\infty} \frac{n^2}{n^3} \rightarrow \frac{1}{n} \rightarrow n^3$ grows faster than n^2 $n^3 > n^2$

$\lim\limits_{n\to\infty} \frac{n^3}{n^2 \log n} \rightarrow \frac{n}{\log n} \rightarrow n$ grows faster than $\log n$ $n^3 > n^2 \log n$

$\lim\limits_{n\to\infty} \frac{n^2 \log n}{\log n} \rightarrow \frac{n^2}{1}$ grows faster than 1 $n^2 \log n > \log n$

$\lim\limits_{n\to\infty} \frac{n^2}{\log n} \rightarrow n^2$ grows faster than $\log n$ $n^2 > \log n$

$\lim\limits_{n\to\infty} \frac{\log n}{n^{0.5}} \rightarrow \lim\limits_{n\to\infty} \frac{1/n}{(0.5)\, n^{0.5-1}} = \lim\limits_{n\to\infty} \frac{1}{0.5 n^{0.5}} = 0$

$\sqrt{n} > \log n$

$\Rightarrow n^3 > n^2 \log n > n^2 > \sqrt{n} > \log n$

• $\lim_{n\to\infty} \frac{10^n}{2^n} \Rightarrow \frac{2^n \cdot 5^n}{2^n} \Rightarrow$ 10^n grows faster than 2^n $10^n > 2^n$

• $\lim_{n\to\infty} \frac{2^n}{8^{\log_2 n}} \Rightarrow \frac{2^n}{2^{3\log_2 n}} \Rightarrow 2^n / 2^{\log_2 n^3}$

$\lim_{n\to\infty} \frac{n}{\log_2 n^3} \Rightarrow$ n grows faster than $\log_2 n$ $2^n > 8^{\log_2 n}$

$\Rightarrow 10^n > 2^n > 8^{\log_2 n}$

• $\lim_{n\to\infty} \frac{10^n}{n^3} \Rightarrow \frac{\log 10^n}{\log n^3} \Rightarrow \frac{n}{3\log n} \Rightarrow 10^n > n^3$

• $\lim_{n\to\infty} \frac{2^n}{n^3} \Rightarrow \frac{\log_2 2^n}{3\log_2 n} \Rightarrow \frac{n}{3\log_2 n} \Rightarrow 2^n > n^3$

• $\lim_{n\to\infty} \frac{8^{\log_2 n}}{n^3} \Rightarrow \frac{2^{\log_2 n^3}}{n^3} \Rightarrow \frac{\log_2 2^{\log_2 n^3}}{\log_2 n^3} \Rightarrow \frac{3\log_2 n}{3\log_2 n} = 1$

$n^3 = 8^{\log_2 n}$

$10^n > 2^n > 8^{\log_2 n} = n^3 > n^2 \log n > n^2 > \sqrt{n} > \log n$

3)

a)
```
for (int i=2; i<=n; i++) {   // O(logn)
        if(i%2 == 0)      // constant time
             count++;     // constant time
        else              // constant time
             i=(i-1)i;    // constant time
}
```
$\Rightarrow O(\log n)$

b) first_element = my_array[0]; //constant time
second_element = my_array[0]; //constant time
for(int i=0; i<sizeOfArray; i++){ //sizeOfArray = n, n time
if(my_array[i] < first_element){ //constant time
second_element = first_element; //constant time
first_element = my_array[i]; //constant time
}else if(my_array[i] < second_element) //constant time
if(my_array[i] != first_element) //constant time
second_element = my_array[i]; //constant time
}
$\Rightarrow$ $\Theta(n)$ (theta n)

c) return array[0] * array[2]; //constant time

$\Theta(1)$.

d) int sum=0; //constant time
for(int i=0; i<n; i=i+5) //worst case $\Rightarrow$ $(n/5)+1$
sum += array[i] * array[i]; //constant time
return sum; //constant time

$O(n)$

e) for(int i=0; i<n; i++) //n time
for(int j=1; j<i; j=j*2) // $\log_2 n$ time
printf("%d", array[i] * array[j]); //constant time

m is the number of characters that will printed,

$\Rightarrow O(n \cdot \log_2 n)$.

f) if (p_4 (array, n) > (200)) // O(n)
P_5 (array, n); // O(n log₂n)
else
printf ("%d", p_3 (array) * p_4 (array, n)); // O(n)

if block execute → O(n·log₂n) → worst case
else block execute → O(1).O(n) → O(n) → best case

g) int i = n; // O(1)
while (i > 0) { // O(log₂n)
for (int j = 0; j < n; j++) // O(n)
System.out.println ("*"); // O(1)
i = i/2; // O(1)
}
⟹ O(n.log₂n)

h) while (n > 0) { // O(log₂n)
for (int j = 0; j < n; j++) // O(n/2^k)
System.out.println ("*"); // O(1)
n = n/2; // O(1)
}
(($k = \log_2 n - 1$))
k is the number of while loop minus 1.

$\Rightarrow \log_2 n \frac{n}{2^{\log_2 n - 1}} \Rightarrow O\left(\log_2 n \frac{n}{2^{\log_2 n - 1}}\right)$

i) if (n == 0) return 1; // O(1)
else return n * p_9 (n-1); // O(n)

⟹ O(n)

j) if (n == 1) // O(1)
return; // O(1)
p_10 (A, n-1); // O(n) iterates n-1 times.
j = n-1; // O(1)
while (j > 0 and A[j] < [j-1]) { // O(n) iterates n-1 times
SWAP(A[j], A[j-1]); // O(1)
j = j-1; // O(1)
}
$\Rightarrow O(n.n) \Rightarrow O(n^2)$

4)

a) "The running time of algorithm A is at least $O(n^2)$" is meaningless, ridiculous. Running time of algorithm A is at least faster than $O(n^2)$, because notation of $O(n)$ is the formal way to express the upper bound so running time might be slower than $O(n^2)$.

b)

I. $2^{n+1} = \Theta(2^n)$ is true because for Θ notation we need to have $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$

$c_1 2^n \le 2^{n+1} \le 2^n c_2 \quad \rightarrow \quad c_1 2^n \le 2^n \le 2^n c_2$

II. $2^{2n} = \Theta(2^n) \Rightarrow c_1 2^n \le 2^{2n} \le 2^n c_2$

$2^{2n} > 2^n$ so it is false., Because Θ is exact complexity.

III. $f(n) = O(n^2) \qquad g(n) = \Theta(n^2)$

Big O notation is upper bound so $f(n)$ can be faster than n^2 that is smaller than n^2 for ex: n
if $f(n) = O(n)$ then $f(n) * g(n) = \Theta(n^3)$.
so it is false.

5)

a) $T(n) = 2T(n/2) + n, \quad T(1) = 1$

$n \rightarrow n$ time
$n/2 \quad n/2 \rightarrow n$ "
$n/4 \quad n/4 \quad n/4 \quad n/4 \rightarrow n$ " $\Big\}$ k time $\rightarrow$ $n.k \rightarrow n \lg n$
$n/2^k \qquad n/2^k$

$\frac{n}{2^k} = 1 \qquad n = 2^k \rightarrow \lg n = k$

$\Rightarrow O(n \lg n)$

b) $T(n) = 2T(n-1) + 1, \quad T(0) = 0$

$T(n) \rightarrow 1$
$1 \quad T(n-1) \quad T(n-1) \rightarrow 2$
$1 \quad T(n-2) \quad T(n-2) \quad 1 \quad T(n-2) \quad T(n-2) \rightarrow 2^2$
$T(0) \quad T(0) \qquad T(0) \quad (T(0) \qquad 2^k$

$1 + 2 + 2^2 + 2^3 + \dots + 2^k = 2^{k+1} - 1$

Assume $n - k = 0$

$\Rightarrow 2^{n+1} - 1 \Rightarrow O(2^n)$

6) for (int i=0; i<numbers.length; i++) { // length time (n)
for (int k=i+1; k<numbers.length; k++) (n-1) time
if ((numbers[i] + numbers[k] == sum) //constant
System.out.printf ("sum is %d\n %d, %d, \n",
sum, numbers[i], numbers[k]); //constant
}

m is the number of characters that will printed.

$\Rightarrow O(n.(n-1)) \rightarrow O(n^2)$

if we increase number of elements in array running time is increase. for ex: 20 element take 0.0024 seconds 10 element take 0.0022 seconds.

for $n=10$; $O(100m)$ ①
for $n=20$; $O(400m)$ ②

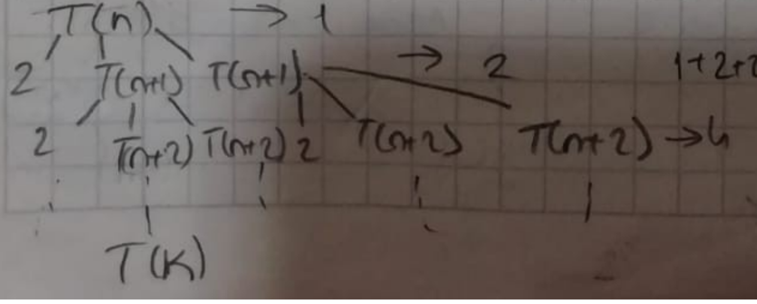
② four time slower than ①

$\frac{0.0024}{0.0022} \Rightarrow \frac{12}{11} = 1.09$ slower according to my theoretical result.

7) if(c_index == numbers.length -1) // O(1)
return; // O(1)
if (numbers[c_index] + numbers[next_index] == sum) // O(1)
print // O(1)
if (next_index == numbers.length-1) // O(1)
O(n)// findPairs(numbers, sum, c_index+1, next_index - numbers.length-2)
else
findPairs(numbers, sum, c_index, next_index+1); // O(n-1)

$\Rightarrow O(n^2-n) \Rightarrow O(n^2)$

recurrence relation is $T(n) = 2T(n+1) + 2$



$1+2+2^2+2^3+\dots+2^x = 2^{x+1}-1$

Algorithm for the 6. question:

makefile × FindPairsRecursive.java × FindPairs.java × Main.java ×

```java
public class FindPairs{

    public void findPairs(int [] numbers, int sum){

        for(int i=0;i<numbers.length;i++){
            for(int k=i+1;k<numbers.length;k++)
                if((numbers[i] + numbers[k]) == sum)
                    System.out.printf("sum is %d\n%d, %d \n", sum, numbers[i], numbers[k]);
        }
    }
}
```

Algorithm for the 7.question:

makefile × FindPairsRecursive.java × FindPairs.java × Main.java ×

```java
public class FindPairsRecursive{

    public void findPairs(int [] numbers, int sum, int c_index, int next_index){

        if(c_index == numbers.length-1)
            return;

        if(numbers[c_index] + numbers[next_index] == sum)
            System.out.printf("sum is %d\n%d, %d \n", sum, numbers[c_index], numbers[next_index]);

        if(next_index == numbers.length-1)
            findPairs(numbers, sum, c_index: c_index+1, next_index: next_index-numbers.length+2);

        else
            findPairs(numbers, sum, c_index, next_index: next_index+1);
    }

}
```

Makefile:

```
JC = javac
JFLAGS = -classpath .
JD = javadoc
JDFLAGS = -protected -splitindex -use -author -version -d ./javadoc
RM = rm
JR = java

CLASSES = \
        FindPairs.java \
        Main.java

all : Main.class

run :
	$(JR) Main

classes : $(CLASSES:.java=.class)

%.class : %.java
	$(JC) $(JFLAGS) $<

doc:
	$(JD) $(JDFLAGS) *.java

clean:
	$(RM) *.class

cleandoc:
	$(RM) -r ./javadoc
```

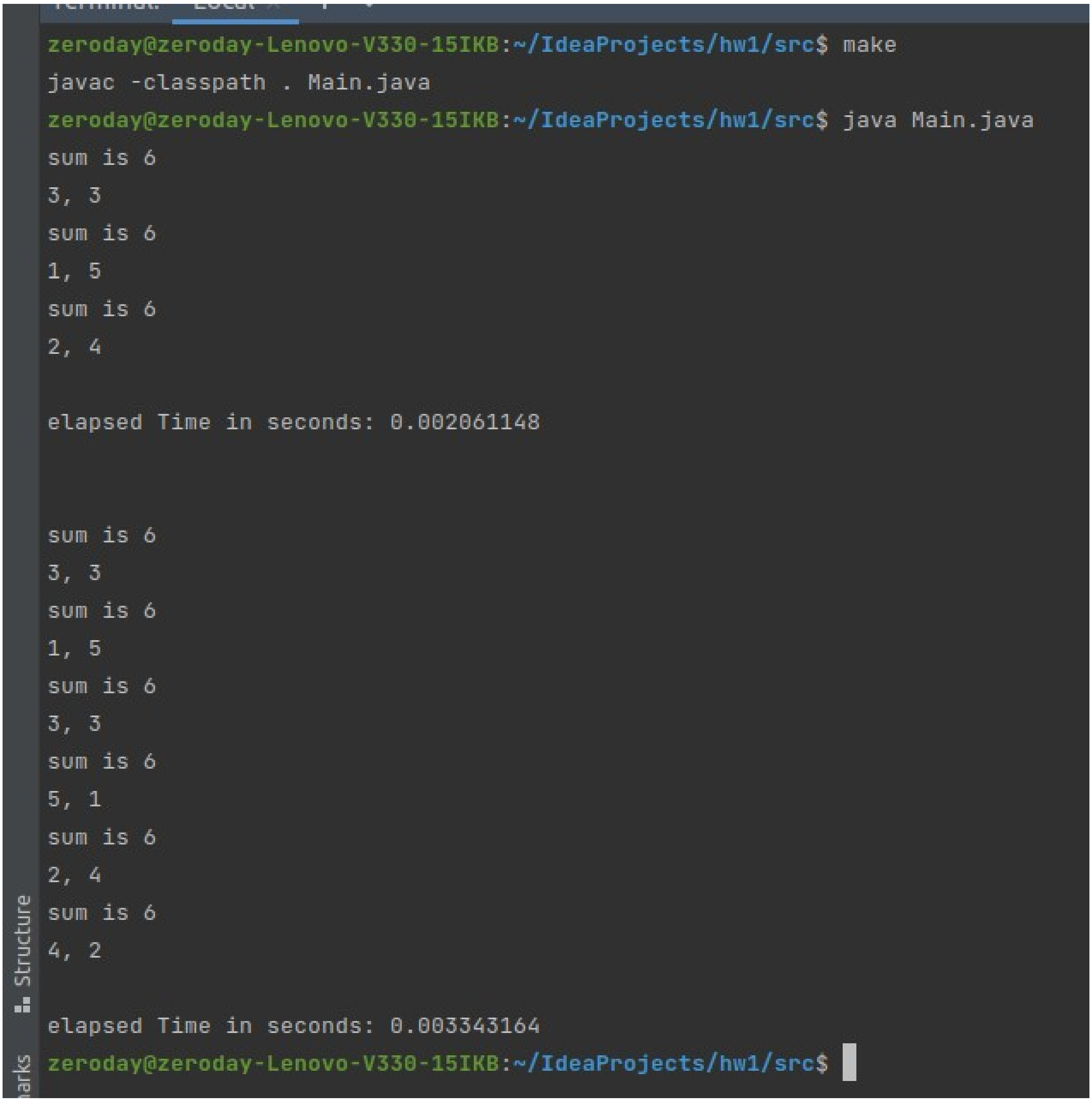
Main:

```java
public class Main{

    public static void main(String [] args){

        FindPairs pairs = new FindPairs();
        FindPairsRecursive pairs2 = new FindPairsRecursive();
        int [] numbers = new int[10];

        numbers[0] = 3;
        numbers[1] = 1;
        numbers[2] = 3;
        numbers[3] = 6;
        numbers[4] = 8;
        numbers[5] = 9;
        numbers[6] = 5;
        numbers[7] = 2;
        numbers[8] = 4;
        numbers[9] = 7;

        /*numbers[0] = 3;
        numbers[1] = 1;
        numbers[2] = 29;
        numbers[3] = 6;
        numbers[4] = 8;
        numbers[5] = 9;
        numbers[6] = 5;
        numbers[7] = 2;
        numbers[8] = 4;
        numbers[9] = 7;
        numbers[10] = 32;
        numbers[11] = 5;
        numbers[12] = 25;
        numbers[13] = 34;
        numbers[14] = 12;
        numbers[15] = 23;
        numbers[16] = 14;
        numbers[17] = 15;
        numbers[18] = 17;
        numbers[19] = 19;*/
```

e × FindPairsRecursive.java × FindPairs.java × Main.java ×

```java
        numbers[1] = 1;
        numbers[2] = 29;
        numbers[3] = 6;
        numbers[4] = 8;
        numbers[5] = 9;
        numbers[6] = 5;
        numbers[7] = 2;
        numbers[8] = 4;
        numbers[9] = 7;
        numbers[10] = 32;
        numbers[11] = 5;
        numbers[12] = 25;
        numbers[13] = 34;
        numbers[14] = 12;
        numbers[15] = 23;
        numbers[16] = 14;
        numbers[17] = 15;
        numbers[18] = 17;
        numbers[19] = 19;*/


        long start = System.nanoTime();
        pairs.findPairs(numbers, sum: 6);
        long end = System.nanoTime();

        System.out.println("\nelapsed Time in seconds: "+ (double)(end-start)/1000000000 + "\n\n");

        long start2 = System.nanoTime();
        pairs2.findPairs(numbers, sum: 6, c_index: 0, next_index: 1);
        long end2 = System.nanoTime();

        System.out.println("\nelapsed Time in seconds: "+ (double)(end2-start2)/1000000000);
    }
```

Program output:



```
zeroday@zeroday-Lenovo-V330-15IKB:~/IdeaProjects/hw1/src$ make
javac -classpath . Main.java
zeroday@zeroday-Lenovo-V330-15IKB:~/IdeaProjects/hw1/src$ java Main.java
sum is 6
3, 3
sum is 6
1, 5
sum is 6
2, 4

elapsed Time in seconds: 0.002061148


sum is 6
3, 3
sum is 6
1, 5
sum is 6
3, 3
sum is 6
5, 1
sum is 6
2, 4
sum is 6
4, 2

elapsed Time in seconds: 0.003343164
zeroday@zeroday-Lenovo-V330-15IKB:~/IdeaProjects/hw1/src$
```