Analysis Assignment 1 Report
Hassan ElMasry 52-2481 13000828

I created a stopwatch class that has a Big Decimal as a attribute that contains start time to log when the code started and has a initializer that sets the start time to when the stopwatch is initialized and has a method elapsed time that calculates the difference between the start and end time and divides it by 1000 to convert it into milliseconds, this will be helpful in the assignment to calculate the execution time and plot the graphs.

Question 1:

a.

```
i. public static BigInteger powerStandard(int a, int n){
    BigInteger result = BigInteger.valueOf(1);
    BigInteger bigA = BigInteger.valueOf(a);
    while(n > 0){
        result = result.multiply(bigA);
        n--;
    }
    return result;
}
```

This is the naïve way of calculating the power of a to the n, I used big Integers instead of int to be able to handle extremely big numbers without returning to -ve numbers once the limit of int is reached, the output is calculated by looping on n and decrementing it every time and in every iterartion we multiply the result which was initialed by 1 with a until n is 0.

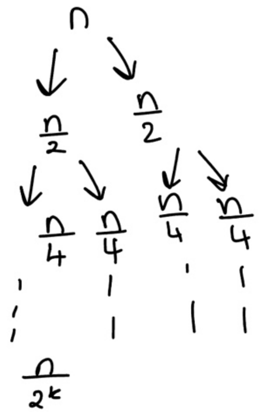
```
ii. public static BigInteger powerDivNConq(int a, int n){
    BigInteger result = BigInteger.valueOf(1);
    BigInteger bigA = BigInteger.valueOf(a);
    if(n == 0){
        return result;
    }
    else if(n == 1){
        return bigA;
    }
    else{
        if(n % 2 == 0){
            result = powerDivNConq(a, n/2);
            return result.multiply(result);
        }
        else{
            result = powerDivNConq(a, (n-1)/2);
            return result.multiply(result).multiply(bigA);
        }
    }
}
```

this is the divide and conquer approach, we start by initializing the result to 1 and checking the base case where n is 0 and it will return a answer of 1, if it is not 0 then we will split the the power, if it is even we directly split it , If its odd we split by (n-1)/2 then do a multiplication with the same function until it reaches the result of 1, then multiplies all the branches of the recursion.

1.b) The naïve method is using a single loop which iterates n times to find the result hence the time complexity is O(n) where n is the power applied to the base a.

The Divide and Conquer Method is using recursion to calculate the result. Hence the recurrence relation is

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



Cost of leave $= n^{\log_2 1} = n^0 = 1$

depth $= \log_2 n$ since we divide each time by 2

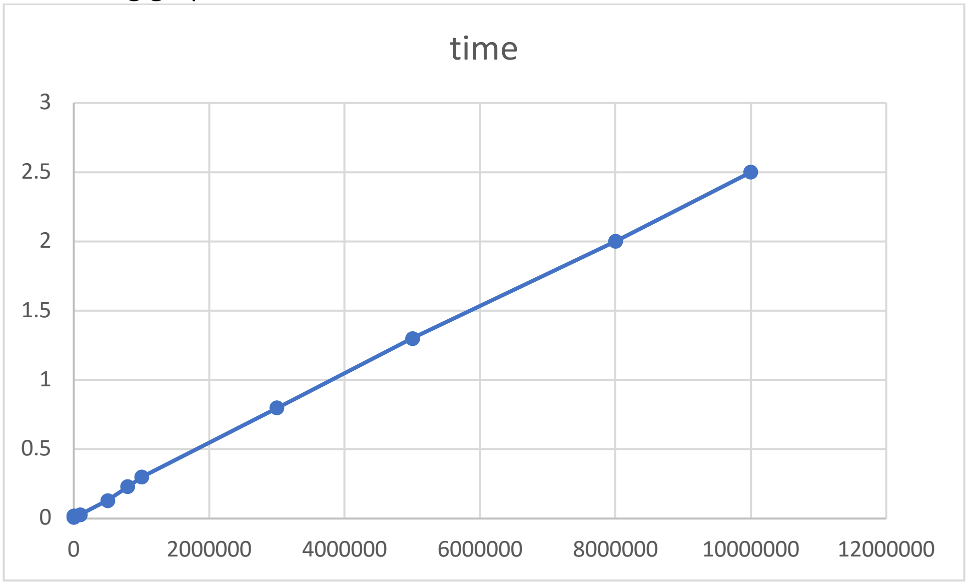
hence cost $= 1 \times \log_2 n = \log_2 n$

$$T(n) = O(1) \times \log_2 n = O(\log_2 n)$$

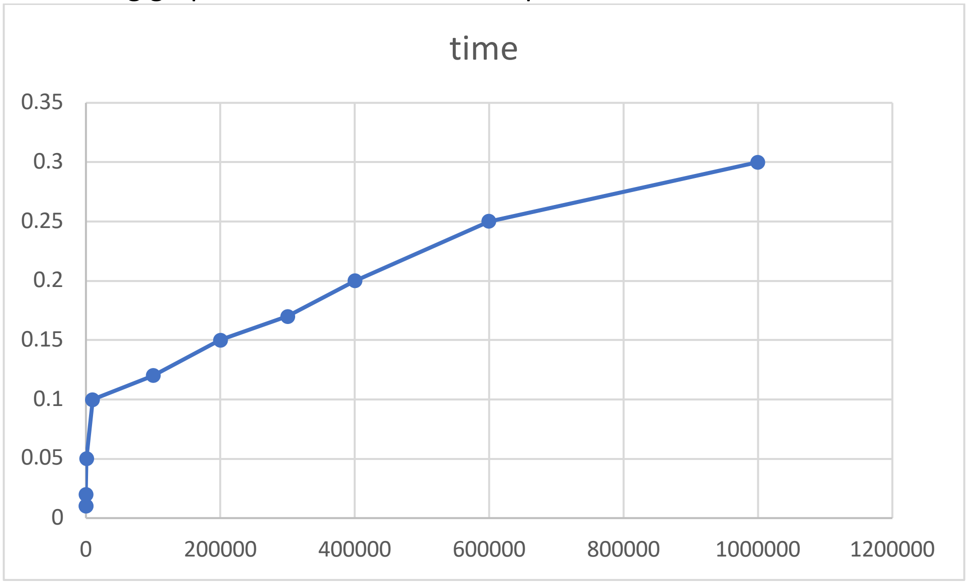
Hence the cost is O(log n) where the log is to the 2nd base

1.c)

The timing graph for the naïve method



The timing graph for the divide and conquer method



d. the experimental results are quite similar to the expected as the graph of the naïve method shows a linear relation similar to o(n) and the divide and conquer method shows a logarithmic function similar to o(log n) where the base of the log is 2.

Question 2: a.

```java
public static int[] mergeSortArray(int[] array, int f, int l ){
    // if the array is only 1 element
    if (f == l){
        int [] r = new int[1];
        r[0] = array[f];
        return r;
    }
    // split the array into 2 parts
    int mid = (f + l)/2;
    // sort the left side with recursion
    int[] left = mergeSortArray(array, f, mid);
    // sort the right side with recursion
    int[] right = mergeSortArray(array, mid + 1, l);
    // merge the 2 arrays ma3 ba3d when its done sorting
    int [] r = mergeSorted(left, right);
    return r;
}
```

this is the method that will merge Sort the array it starts by checking if the first index is equal to the last index and it will return the result with only 1 element which is that element, else it splits the array in 2 using the first and last index it calculates the middle of it then calls itself with the left side of the array and the right side of the array then it calls a method called merge sorted and returns the result of it.

```java
public static int[] mergeSorted(int [] a, int [] b){
    int[] result = new int[a.length + b.length];
    int rstart = 0;
    int acount = 0;
    int bcount = 0;
    while(acount < a.length && bcount < b.length){
        if(a[acount] <= b[bcount]){
            result[rstart++] = a[acount++];
        }else{
            result[rstart++] = b[bcount++];
        }
    }
    while(acount < a.length){
        result[rstart++] = a[acount++];
    }
    while(bcount < b.length){
        result[rstart++] = b[bcount++];
    }
    return result;
}
```

this is the method that merges 2 sorted arrays into another array, we started by creating a new array with the length of both arrays combined then initializing 3 counters once for the 1st array and the 2nd array and the result array then start a loop witch stops when one of the 2 input arrays reaches its end and we compare the elements In hand and insert the smaller element 1st then checks which of the arrays is still not in the end and inserts the rest of the element.

```
public static boolean binarySearch(int[] array , int key, int low, int
high){
    int mid = low + ((high - low) / 2);
    if(low > high){
        return false;
    }
    if(array[mid] == key){
        return true;
    }else if(array[mid] > key){
        return binarySearch(array, key, low , mid - 1);
    }else{
        return binarySearch(array, key, mid + 1, high);
    }

}
```

this is the code that preforms the binary search using an array and a int key which we will search for and a low and high index which we use to identify the bounds for the search, we start by checking if the low is bigger than high which will mean the there is no result found then check if the middle is the key and returns true if it is, else it updates the middle depending on the key and calls the method again.

```
public static List<int[]> findPairs(int[] array, int sum){
    array = mergeSortArray(array,0, array.length - 1);
    List<int[]> pairs = new ArrayList<>();
    for(int i = 0; i < array.length; i++){
        int remainder = sum - array[i];
        if(binarySearch(array,remainder,i + 1, array.length - 1)){
            pairs.add(new int[]{array[i], remainder});
        }
    }
    return pairs;
}
```

here we preform the last step to find the pairs we get the 1st element in the array after we sort it then calculate the remaining of the sum we need by subtraction and preform a binary search to find the element if we find it we add it to the list of results if not we move to the next element until we reach the end of the array.

b.

Merge Sort Array:

$T(n) = 2T(\frac{n}{2}) + O(n)$

$a = 2$
$b = 2$

Cost of leaves $= n^{\log_2 2} = n$

Cost of root $= n$

Distribution is constant

Case 2 from Master theorm

then $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^{\log_2 2} \log n)$

$= \Theta(n \log n)$

Merge Sorted: Same as above Method

$T(n) = 2T(\frac{n}{2}) + O(n)$

$a = 2$

$b = 2$

Cost of leaves $= n^{\log_2 2} = n$

Cost of root $= n$

Distribution is constant

Case 2 from Master theorm

then $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^{\log_2 2} \log n)$

$= \Theta(n \log n)$

Binary Search:

$$T(n) = T(\tfrac{n}{2}) + \Theta(1)$$

$a=1$

$b=2$

Cost of leaves $= n^{\log_2 1} = n^0 = 1$

Cost of root $= f(n) = 1$

Constant distribution

hence Case 2

then $T(n) = \Theta(n^{\log_2 1} \log n) = \Theta(\log n)$

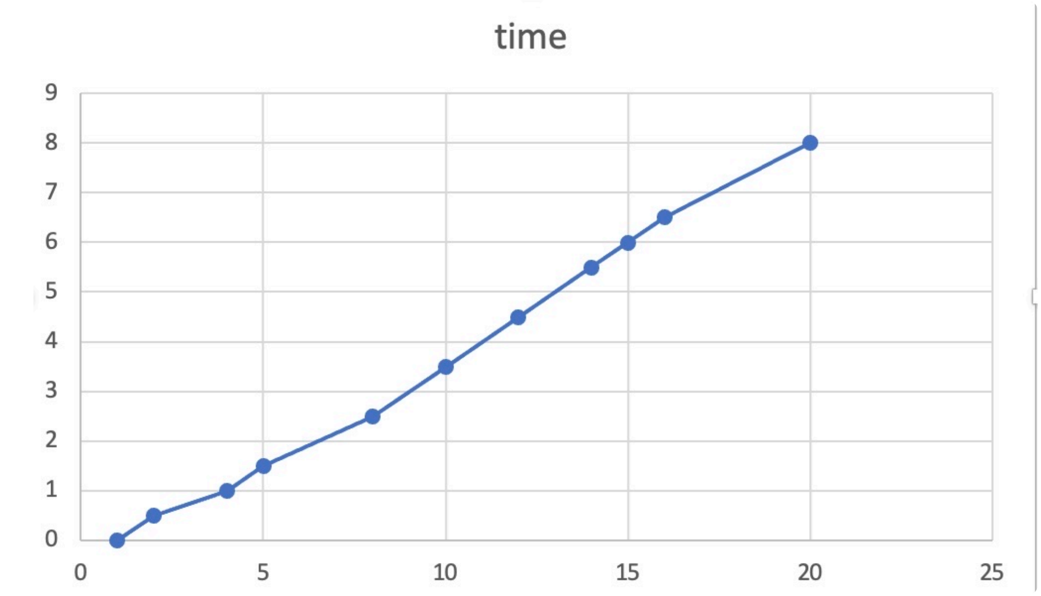
Find Pairs:

$$T(n) = \Theta(n\log n) + n\,\Theta(\log n) + \Theta(n) + \Theta(1)$$

Sumation of all methods used inside

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

c.



The results from the graph follow the
Patern of an nlogn graph hence follows
the prediction in 2.6