Practise problems on Time complexity of an algorithm

1. Analyse the number of instructions executed in the following recursive algorithm for computing \( n \)th Fibonacci numbers as a function of \( n \)

```java
public static int fib(int n)
{
    if(n==0) return 1;
    else if(n==1) return 1;
    else return(fib(n-1) + fib(n-2));
}
```

```java
public static void main(String args[])
{
    int n = Integer.parseInt(args[0]);
    System.out.println(fib(n));
}
```

**Answer:** We proceed similar to the analysis of merge sort. We consider the recursion tree for \( \text{fib}(n) \). We can observe that for \( n > 1 \), the number of instructions executed during \( \text{fib}(n) \) is equal to the number of instructions executed during \( \text{fib}(n-1) \) plus the number of instructions executed during \( \text{fib}(n-2) \) and two or three instructions in addition. Hence, for each node in the recursion tree, the number of instructions executed (excluding those which are executed during its children) is just a constant. So the total number of instructions executed is \( c \) times the number of nodes in the recursion tree of \( \text{fib}(n) \). We shall now try to estimate \( T(n) \) up to some constant multiplicative factor. Let \( T(n) \) be the number of nodes in the recursion tree for \( \text{fib}(n) \). \( T(n) \) can be expressed by the following equation.

\[
T(n) = \begin{cases} 
1 & \text{for } n = 0,1 \\
T(n-1) + T(n-2) + 1 & \text{for } n > 1 
\end{cases}
\]

Let us define function \( G(n) \) as \( T(n) + 1 \). It is easy to observe that \( G(0) = 2, G(1) = 2, \) and \( G(n) = G(n-1) + G(n-2) \) for \( n > 1 \). This equation looks familiar. It is the same as that of fibonacci number except that it differs at the base cases - \( n = \{0,1\} \). It is easy to prove by induction on \( n \) that for all \( n \geq 1 \),

\[
\text{Fibonacci}(n) < G(n) < 4\text{Fibonacci}(n) \quad (1)
\]

Using Equation 1 and the relation between \( T(n) \) and \( G(n) \), it follows that \( \text{Fibonacci}(n) - 1 < T(n) < 4\text{Fibonacci}(n) - 1 \). Hence the number of instructions executed during \( \text{fib}(n) \) is within some constant multiple of \( n \)th Fibonacci number.

The motivated students may try to analyse how big \( n \)th Fibonacci numbers might be. In fact, it grows exponentially with \( n \). Here is a sketch of the
proof (this is optional and won’t be required for end semester exam). Show that \( F(n) \geq H(n) \) for all \( n \geq 2 \) where \( H(1) = H(2) = 1 \), and for \( n > 2 \), \( H(n) \) is defined as follows.

\[
H(n) = 2H(n-2)
\]

By unfolding the above recurrence, it follows easily that \( H(n) \) grows exponentially with \( n \).

2. Analyse the number of instructions executed in the following iterative algorithm for computing \( n \)th Fibonacci numbers as a function of \( n \)

```java
public static void main(String args[])
{
    int n = Integer.parseInt(args[0]);
    if(n==0) System.out.println(0);
    else if(n==1) System.out.println(1);
    else
    {
        int fib1 = 0;
        int fib2 =1;
        for(int i=2; i<=n;i=i+1)
        {
            int temp = fib1+fib2;
            fib1 = fib2;
            fib2 = temp;
        }
        System.out.println(fib2);
    }
}
```

**Answer**: The algorithm takes \( cn \) instructions for some positive constant \( c \).

3. Design an algorithm which computes \( 3^n \) using only \( c \log n \) instructions for some positive constant \( c \).

**Hint**: Write a method based on the following recursive formulation of \( 3^n \) carefully.

\[
3^n = \begin{cases} 
1 & \text{if } n = 0 \\
3^{n/2} * 3^{n/2} & \text{if } n\%2 == 0 \\
3^{n/2} * 3^{n/2} * 3 & \text{if } n\%2 == 1 
\end{cases}
\]

4. Given an array \( A \) which stores 0 and 1, such that each entry containing 0 appears before all those entries containing 1. In other words, it is like \( \{0,0,0,...,0,0,1,1,...,111\} \). Design an algorithm to find out the small index \( i \) in the array \( A \) such that \( A[i] = 1 \) using \( c \log n \) instructions in the worst case for some positive constant \( c \).

**Hint**: exploit the idea used in binary search. The method is described below.
public static void First_index_with_one(int[] A)
{
    if(A[0]==1) System.out.println(‘The first index storing one is 0’);
    else if(A[(A.length-1)]==0) System.out.println(‘All entries of A are 0’);
    else //There is a unique i such that A[i-1]==0 and A[i]==1
    {  int left = 0;  int right=A.length-1;
        boolean Isfound=false;
        while(Isfound==false)
        {  mid = (left+right)/2;
            if(A[mid]==0) left = mid+1;
            else
            {  if(A[mid-1]==0) is_found=true;
                else right=mid-1;
            }
        }
        System.out.println(‘The first index containing one is ’+mid);
    }
}

5. How many instructions are executed when we multiply $n \times m$ matrix $A$ with $m \times r$ matrix $B$?

Answer: The number of instructions executed is $cmr$ for some positive constant $c$.

Explanation: We need to analyse the algorithm for multiplying two matrices as discussed in one of the lecture. The final matrix will be $n \times r$. For each entry of the final matrix we perform vector product of one row of matrix $A$ with one column of $B$, hence $m$ multiplications and $m-1$ additions. Thus $cm$ instructions are executed for computing one entry of the final matrix. Hence the total number of instructions for computing product of $A$ and $B$ is $cmr$. 

3