**Lab 2 Continued Solutions**

**Problem1: Solution**

Text, letter

Description automatically generated

**Problem2: Solution**

1. False

A picture containing text, whiteboard

Description automatically generated

1. True

A picture containing text, whiteboard

Description automatically generated

1. True

A close-up of a paper

Description automatically generated with low confidence

**Problem3: Solution**

**Step1**: Formulate the recurrence relation as

T(n) = { 4 if n = 1

T(n − 1) + 4 if n > 1

}

**Step2**: Guess the solution as

T(1) = 4 = 1\*4

T(2) = T(1) + 4 = 4 + 4 = 2\*4

T(3) = T(2) + 4 = 4 + 4 + 4 = 3\*4

……

T(n) = T(n − 1) + 4 = n\*4 = 4n

**Step 3**: Prove the formula from Step 2 is a solution

Let f(n) = 4n.

We show f(1) = 4 and f(n) = f(n − 1) + 4.

The first part is obviously true.

We also have f(n) = 4n = 4(n − 1) + 4 = f(n − 1) + 4 which we have to prove.

**Step 4**: Prove correctness

(a) Verify valid recursion. The recursion is valid because “n == 0 || n == 1” is the base case and repeated self-calls in the algorithm lead to the base case since each self-call reduces the input size by 1.

(b) Verify base case outputs are correct. This follows since 0! = 1 and 1! = 1.

(c) Verify inductively that outputs are correct for all n. Assume recursiveFactorial(j) outputs j! for all j < n, where n > 1. Then, recursiveFactorial on input n returns recursiveFactorial(n-1) \* n, which, by inductive hypothesis, is (n-1)! \* n = n!

**Problem4: Solution**

Below is an iterative Java method that computes fibonacci number Fn on input n. It executes a single loop that depends on n, so running time is O(n).

int[] store;

//precondition: n is non-negative integer

public int fib(int n) {

store = new int[n+1];

store[0] = 0; store[1] = 1;

//Loop Invariant: I(i): store[i] = F\_i

for(int i = 2; i <= n; i++) {

store[i] = store[i-1] + store[i-2];

}

//postcondition: store[n] = F\_n

return store[n];

}

//postcondition: F\_n is returned

We verify correctness. We first establish a loop invariant I and show that I(k) holds at the end of the i = k pass, for 2 ≤ k ≤ n. Our loop invariant is:

I(i) : store[i] = Fi.

For the Base Case, we establish I(2) at the end of the i = 2 pass. But this is clear since I[0] = F0 and I[1] = F1 and store[2] = store[0] + store[1].

For the Induction Step, assume I(j) holds at the end of the i = j pass for each j < k, where 2 ≤ k ≤ n; we show I(k) holds at the end of the i = k pass. By the Induction Hypothesis, store[k−1] = Fk−1 and store[k−2] = Fk−2. By inspection of the algorithm, it is clear therefore that

store[k] = store[k − 1] + store[k − 2] = Fk−1 + Fk−2 = Fk

This completes the induction and shows that the loop invariant holds for 2 ≤ k ≤ n. Therefore, at the end of the i = n pass, we have that store[n] stores Fn, and this is the value returned by the algorithm. So, our algorithm is correct.

**Problem5: Solution**

T(n) = T(n/2) + n; T(1) = 1

Using master formula,

Here, a = 1, b = 2, c = 1, d = 1, k = 1.

Here, a < bk

According to master formula, T(n) = Θ(n)

**Problem6: Solution**

Java Code:

public class ZeroesAndOnes {

public static int[] findNum0s1s(int[] A) {

int[] retval = new int[2];

if(A.length == 0) {

retval[0] = 0;

retval[1] = 0;

return retval;

}

else if(A[0] == 1) {

retval[0] = 0;

retval[1] = A.length;

return retval;

}

else if(A[A.length-1] == 0) {

retval[1] = 0;

retval[0] = A.length;

return retval;

}

return recurse(A, 0, A.length-1);

}

private static int[] recurse(int[] A,int lower, int upper) {

int[] retval = new int[2];

int mid = (upper + lower)/2;

if(A[mid] == 0 && A[mid+1] == 1) {

retval[0] = mid +1;

retval[1] = A.length - (mid+1);

return retval;

}

if(A[mid] == 0 && A[mid+1] == 0) {

return recurse(A, mid+1, upper);

}

if (A[mid] == 1) {

return recurse(A, lower, mid -1);

}

return null;

}

public static void main(String[] args) {

int[] testArray = {0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};

System.out.println(Arrays.toString(findNum0s1s(testArray)));

}

}

Since, BinarySearch is used to locate a 0 followed by a 1, this part of the algorithm takes O(log n), and the rest takes constant time. We can also easily show that f/g reaches 0 as n tends to infinity. Hence, it is o(n).