Name: Yogendra Pokhrel
Student ID 017585

Lab 2 Continued

1. Answer

Proof by Induction : We need to prove that for all $n > 4$, the Fibonacci sequence satisfies :

$$F_n > (4/3)^n$$

Base case ($n = 5, 6$)

From the given table :

$F_5 = 5$ and $(4/3)^5 \approx 4.21 \rightarrow 5 > 4.21$ (Holds)

$F_6 = 8$ and $(4/3)^6 \approx 5.62 \rightarrow 8 > 5.62$ (Holds)

Since the inequality holds for $n = 5, 6$ we proceed with induction

Hypothesis

Assume the inequality holds for some $k \geq 6$:

$$F_k > (4/3)^k, \quad F_{k-1} > (4/3)^{k-1}$$

Inductive step

We need to prove

$$F_{k+1} = F_k + F_{k-1} > (4/3)^{k+1}$$

By the induction hypothesis :

$$F_k + F_{k-1} > (4/3)^k + (4/3)^{k-1}$$

Factor out $(4/3)^{k-1}$:

$$F_{k+1} > (4/3)^{k-1} \times (4/3 + 1)$$

since $4/3 + 1 = 7/3$ we get:

$$F_{k+1} > (4/3)^{k-1} \times (7/3)$$

Compare this with $(4/3)^{k+1}$:

$$(4/3)^{k+1} = (4/3)^k \times (4/3)$$

since $7/3 > 4/3$ it follows that:

$$F_{k+1} > (4/3)^{k+1}$$

Thus by induction, the statement holds for all $n > 4$.

→ The recursive Fibonacci algorithm follows the relation:

$$F(n) = F(n-1) + F(n-2)$$

which means that each call spawns two new recursive calls.

The total number of recursive calls grows roughly as $O(F_n)$

Since we just proved that

$$F_n > (4/3)^n$$

We conclude that the recursive Fibonacci function has exponential time complexity, roughly $O(2^n)$.

Reason why Recursive Algorithm is slow

<u>Exponential Growth</u>: The number of function calls doubles at each step, making the computation infeasible for large n.

<u>Redundant computation</u>: Many Fibonacci numbers get recomputed multiple times, leading to wasted effort.

<u>Alternatives</u> : Best practices

<u>Memoization (Top Down DP)</u> : Store previously computed values to avoid redundant work, reducing time complexity to $O(n)$.

Bottom up DP: compute Fibonacci iteratively, storing only the last two values also achieving $O(n)$ time complexity.

Lets analyze and

2) ~~Analyzing~~ Consider each statement using asymptotic notation and limits

a) True or False : 4^n is $O(2^n)$? $\Rightarrow$ False

We need to check if :

$$4^n = O(2^n)$$

Using the formal definition of Big-O :

$$\exists\, c>0,\ n_0 \text{ such that } 4^n \leq c \cdot 2^n \text{ for all } n \geq n_0 .$$

Yogendra Pokhrel
C17585

Rewriting 4^n in terms of base 2:

$$4^n = (2^2)^n = 2^{2n}$$

Thus we need to check if:

$$2^{2n} \leq c \cdot 2^n$$

Dividing both sides by 2^n:

$$2^n \leq c$$

Since 2^n grows unbounded as $n \to \infty$ this inequality is false for any constant c. Therefore 4^n is not $O(2^n)$.

b) True or False : $\log n$ is $\Theta(\log_3 n)$? $\Rightarrow$ True

We need to check if:

$$\log n = \Theta(\log_3 n).$$

Using the definition of Θ we need to show:

$$c_1 \cdot \log_3 n \leq \log n \leq c_2 \cdot \log_3 n$$

for some positive constants c_1, c_2 and sufficiently large n.

Using the logarithm change of base formula:

$$\log_3 n = \frac{\log n}{\log 3}$$

So,

$$\log n = (\log 3) \cdot \log_3 n$$

This shows: $\frac{1}{\log 3} \cdot \log n \leq \log_3 n \leq \log 3 \cdot \log n$

since $\log 3$ is a positive constant, we can take $(c_1 = 1/\log 3)$ and $c_2 = \log 3$, proving the asymptotic equivalence.

c) True or ~~false~~ $n/2 \log n/2$ is $\Theta(n \log n)$? $\Rightarrow$ True

We need to check if :

$$n/2 \log n/2 = \Theta(n \log n)$$

Expanding : $\quad n/2 \log n/2 = n/2 (\log n - \log 2)$

$$= n/2 \log n - n/2 \log 2$$

For Big-Theta, lower and upper bounds should be proportional to $n \log n$.

The dominant term is $n/2 \log n$, which is clearly $O(n \log n)$.

Since $1/2$ is a constant, we also have $\Omega(n \log n)$.

Thus,

$$n/2 \log n/2 = \Theta(n \log n).$$

Yogendra Pokhrel
617585

3) Algorithm

```
public class Factorial {
    public static long recursiveFactorial (int n) {
        if(n==0 || n==1) { return 1; }
        return n * recursiveFactorial (n-1);
    }
}
```

4) Worst case Asymptotic Running Time (Using Guessing Method)

Recurrence Relation :

Let $T(n)$ be the time complexity of recursiveFactorial(n)

$$T(n) = T(n-1) + O(1)$$

Expanding recursively :

$$T(n) = T(n-2) + O(1) + O(1)$$

$$T(n) = T(n-3) + O(1) + O(1) + O(1)$$

$$\vdots$$

$$T(n) = T(1) + O(n)$$

Since $T(1)$ is constant $O(1)$ we get :

$$T(n) = O(n)$$

Thus the worst case time complexity is $O(n)$.

e

B) Proof of correctness (mathematical induction)

Base case

For $n=0$ or $n=1$, the algorithm returns 1, which is correct

because $0! = 1! = 1$

Induction step:

Assume the algorithm correctly computes $k!$ ie

1 to

$$\text{recursiveFactorial}(k) = k!$$

For $n = k+1$:

n).

$$\text{recursiveFactorial}(k+1) = (k+1) \times \text{recursiveFactorial}(k)$$

By the induction hypothesis

$$\text{recursiveFactorial}(k) = k!$$

Thus,

$$\text{recursiveFactorial}(k+1) = (k+1) \times k! = (k+1)!$$

which matches the definition of factorial.

Since, both the base case and inductive step hold, the algorithm is correct by induction.

Yogendra Pokhrel
617585

4) Algorithm

```
public class Fibonacci {
    public static long iterativeFibonacci (int n) {
        if (n <= 1) { return n; }

        long prev = 0, curr = 1;
        for ( int i = 2; i <= n; i++) {
            long next = prev+curr;
            prev = curr;
            curr = next;
        }
        return curr;
    }
}
```

→ Time complexity Analysis

The function iterates once from 2 to n, performing constant time operations inside the loop.

loop iterations = $O(n)$

Operations per iteration $O(1)$

overall Time complexity $O(n)$ linear time

unlike the recursive approach $\Theta(2^n)$ this method is efficient and avoids redundant computations.

proof of correctness (Mathematical Induction)

Base cases:

For $n = 0$; the function returns 0, which is correct.
For $n = 1$; the function returns 1, which is also correct.

Inductive step:
Assume for some k, the function correctly computes F_k and
F_{k-1}.

For $k+1$:
$$F_{k+1} = F_k + F_{k-1}$$

Since the loop maintains two variables tracking F_k and F_{k-1}
at each step the computed value for F_{k+1} is correct.

Thus by induction the algorithm correctly computes Fibonacci
numbers.

5) lets use Master Theorem to solve the recurrence:

$$T(n) = T(n/2) + n$$

step 1: Identify Parameters in the Masters Theorem

The recurrence follows the form.

$$T(n) = aT(n/b) + f(n)$$

where:

$a = 1$ (number of recursive calls)

$b = 2$ (sub problem size reduction factor)

$f(n) = O(n)$ additional work done outside recursion

⇒ step 2: compare ($f(n)$) with $O(n^{\log_b a})$

The master theorem cases depend on the comparison between $f(n)$ and $O(n^{\log_b a})$:

$$\log_2(1) = 0 \Rightarrow n^{\log_2 1} = O(n^0) = O(1)$$

comparing $f(n) = O(n)$ with $O(n^0)$:

$O(n)$ grows faster than $O(1)$

⇒ step 3: Apply master Theorem (case 3)

since $f(n) = O(n)$ dominates $O(n^{\log_b a})$ and satisfies the regularity condition $af(n/b) \leq cf(n)$ for some $c \leq 1$, we apply Case 3 of the Master Theorem

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

Thus

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

the asymptotic running time is linear, $O(n)$.

Yogendra Pokhrel
617585

6) Binary search

Since the array is sorted, we can find the first occurance of 1 using binary search in $O(\log n)$, instead of scanning the entire array.

```
public class CountZerosOnes {
    public static int [] countZerosOnes (int [] A) {
        int n = A.length, left = 0, right = n-1;
        while (left <= right) {
            int mid = left + (right - left)/2;
            if (A[mid] == 1) right = mid - 1;
            else left = mid + 1;
        }
        return new int [] {left, n-left};
    }
```

Complexity

Time : $O(\log n)$ binary search

Space : $O(1)$ constant extra space

Since, the array is sorted the only necessary operation is locating the first 1. Binary search is the optimal way to do this in $O(\log n)$