1. Prove by induction that for all n > 4, Fn > (4/3)n. Then use this result to explain  
   the approximate asymptotic running time of the recursive algorithm for computing  
   the Fibonacci numbers. Is the recursive Fibonacci algorithm fast or slow? Why?

A table with numbers and letters

Description automatically generated

**Ans:**

Fibonacci Sequence Growth Rate: Proving Fn > (4/3)^n for n ≥ 5

Base Case

For n = 5:

F5 = 5 (from the Fibonacci sequence: 1, 1, 2, 3, 5, ...)

(4/3)^5 ≈ 3.16

Therefore, F5 > (4/3)^5

Inductive Step

Assume the inequality Fk > (4/3)^k holds for some k ≥ 5.

We need to prove that Fk+1 > (4/3)^(k+1)

Since Fk+1 = Fk + Fk-1, we have:S

By our inductive hypothesis, Fk > (4/3)^k

For k ≥ 5, we can show that Fk-1 ≥ (1/3)(4/3)^k

Therefore: Fk+1 = Fk + Fk-1 > (4/3)^k + (1/3)(4/3)^k = (4/3)^k · (1 + 1/3) = (4/3)^k · (4/3) = (4/3)^(k+1)

Conclusion

By the principle of mathematical induction, Fn > (4/3)^n for all n ≥ 5.

This exponential growth rate explains why the recursive Fibonacci algorithm is inefficient for large values of n. Each recursive call branches into two more calls, creating an exponential number of redundant calculations as the algorithm repeatedly computes the same Fibonacci values.

1. More big-oh: (Work with someone who is familiar with limits)

a. True or false: 4n is O(2n). Prove your answer.

**Ans:**

False.

For 4^n to be O(2^n), there must exist constants c > 0 and n₀ such that 4^n ≤ c·2^n for all n ≥ n₀.

Let's check if this is possible:

* 4^n = (2^2)^n = 2^(2n)
* So we need: 2^(2n) ≤ c·2^n
* Dividing both sides by 2^n: 2^n ≤ c

For any constant c, 2^n will eventually exceed c as n grows (since 2^n grows exponentially). Therefore, no such constant c exists that satisfies the inequality for all n ≥ n₀.

We can also use limits: lim(n→∞) 4^n/2^n = lim(n→∞) 2^n = ∞

Since this limit is not finite, 4^n is not O(2^n).

b. True or false: log n is Θ(log3 n). Prove your answer.

**Ans:**

True.

Using the change of base formula: log₃ n = (log n)/(log 3)

To prove Θ notation, we need to show both O and Ω bounds:

1. To show log n is O(log₃ n):
   * log n = log 3 · log₃ n
   * Since log 3 is a constant (≈ 1.099), there exists c₁ = log 3 such that:
   * log n ≤ c₁ · log₃ n for all n ≥ 2
2. To show log n is Ω(log₃ n):
   * log₃ n = (log n)/(log 3)
   * Since 1/log 3 is a constant (≈ 0.91), there exists c₂ = 1/log 3 such that:
   * log n ≥ c₂ · log₃ n for all n ≥ 2

Since both conditions are satisfied, log n is Θ(log₃ n).

c. True or false: (n/2) log(n/2) is Θ(nlog n). Prove your answer.

**Ans:**

True.

Let's expand (n/2) log(n/2):

* (n/2) log(n/2) = (n/2) · (log n - log 2)
* = (n log n)/2 - (n log 2)/2

To prove Θ notation:

1. To show (n/2) log(n/2) is O(n log n):
   * For large n, n log n dominates n
   * (n log n)/2 - (n log 2)/2 ≤ c₁ · n log n for some constant c₁
   * We can choose c₁ = 1/2 + ε for any small ε > 0
2. To show (n/2) log(n/2) is Ω(n log n):
   * For sufficiently large n, log n >> log 2
   * (n log n)/2 - (n log 2)/2 ≈ (n log n)/2
   * So (n/2) log(n/2) ≥ c₂ · n log n for some constant c₂
   * We can choose c₂ = 1/4 for n sufficiently large

Since both conditions are satisfied, (n/2) log(n/2) is Θ(n log n).

1. Below, pseudo-code is given for the recursive factorial algorithm  
   recursiveFactorial.  
   **Algorithm** recursiveFactorial(n)  
   **Input**: A non-negative integer n  
   **Output**: n!  
   **if** (n = 0 || n = 1) **then**  
   **return** 1  
   **return** n \* recursiveFactorial(n-1)  
   Do the following:

A. Use the Guessing Method to determine the worst-case asymptotic running time of  
this algorithm. Then verify correctness of your formula.

**Ans:**

1. Recurrence relation: T(n) = T(n-1) + c
   * Base case: T(0) = T(1) = d (constant)
   * Each call has constant overhead (c)
2. Guess: T(n) = O(n)
3. Verification: T(n) = T(n-1) + c = T(n-2) + 2c = T(n-3) + 3c ... = T(0) + nc = d + nc = O(n)

Therefore, recursiveFactorial has a worst-case time complexity of O(n).

B. Prove the algorithm is correct.

**Ans:**

Base Cases:

* For n = 0: recursiveFactorial(0) returns 1, which equals 0! = 1 ✓
* For n = 1: recursiveFactorial(1) returns 1, which equals 1! = 1 ✓

Inductive Hypothesis:

Assume recursiveFactorial(k) correctly returns k! for some k ≥ 1.

Inductive Step:

We need to prove that recursiveFactorial(k+1) correctly returns (k+1)!

recursiveFactorial(k+1) = (k+1) × recursiveFactorial(k) = (k+1) × k! (by inductive hypothesis) = (k+1)!

Therefore, by mathematical induction, recursiveFactorial(n) correctly computes n! for all non-negative integers n.

1. Devise an iterative algorithm for computing the Fibonacci numbers and compute its  
   running time. Prove your algorithm is correct.

**Ans:**A screenshot of a computer program

AI-generated content may be incorrect.

Correctness Proof

Base Cases:

For n = 0: The algorithm returns 0, which is F(0) ✓

For n = 1: The algorithm returns 1, which is F(1) ✓

Inductive Step:

Assume the algorithm correctly computes F(k) for all k where 2 ≤ k ≤ j for some j ≥ 1.

Let's prove it correctly computes F(j+1):

After j-1 iterations of the loop:

fibPrev holds F(j-1)

fibCurr holds F(j)

In the jth iteration:

fibNext = fibPrev + fibCurr = F(j-1) + F(j)

fibPrev is updated to F(j)

fibCurr is updated to F(j+1)

By the definition of Fibonacci numbers: F(j+1) = F(j) + F(j-1)

Therefore, after j iterations, fibCurr correctly holds F(j+1).

By induction, the algorithm correctly computes F(n) for all non-negative integers n.

1. Find the asymptotic running time using the Master Formula:  
   T(n) = T(n/2) + n; T(1) = 1

**Ans:**

Apply the Master Theorem: For the recurrence T(n)=T(n/2)+n:

* Here, a=1, b=2, and f(n)=n
* Since nlogba = n0=1 and f(n)=n, this fits Case 2 of the Master Theorem, where T(n)=Θ(n)

1. You are given a length-n array A consisting of 0s and 1s, arranged in  
   sorted order. Give an o(n) algorithm that counts the total number of 0s and 1s in the array. Your algorithm may not make use of auxiliary storage such as arrays or  
   hashtables (more precisely, the only additional space used, beyond the given array, is O(1)). You must give an argument to show that your algorithm runs in o(n) time.

**Ans:**

Proof of o(n) Time Complexity

This algorithm runs in O(log n) time, which is o(n) (strictly less than linear time):

1. The algorithm uses binary search to find the boundary between 0s and 1s
2. In each iteration of the binary search, we divide the search space in half
3. The number of iterations required is log₂(n) in the worst case
4. All operations inside the loop take constant time

Therefore, the overall time complexity is O(log n), which is o(n).

Space Complexity

The algorithm uses only a constant amount of extra space (a few variables), so the space complexity is O(1) as required.

Correctness

* Since the array is sorted, all 0s appear before all 1s
* Binary search correctly finds the index of the first 1
* The count of 0s equals this index
* The count of 1s equals the array length minus this index

This approach takes advantage of the sorted property to avoid examining every element, allowing us to solve the problem in logarithmic rather than linear time.

A screenshot of a computer program

AI-generated content may be incorrect.