Introduction to Asymptotic Analysis & Solving Recursions

Fibonacci Numbers

Fibonacci sequence:

```
0, 1, 1, 2, 3, 5, 8, ……
```

Formula:

```
F_n = F_{n-1} + F_{n-2}
```

Recursive implementation in code:

```
def fib(n):
  if n == 0 or n == 1:
    return n
  return fib(n-1) + fib(n-2)
```

- Recursion takes space; it uses stacks to store results. Find the longest recursion path to determine the space complexity for this solution.
- Recursion can cause stack overflow.

Time & Space Complexity

- Evaluate the efficiency of an algorithm based on Time & Space Complexity.
- Common running times for sorting algorithms include: O(n^2), O(nlogn).

Recursion Tree

- Continue calling until reaching the base case.
- Overall runtime equals the sum of time on each tree node.
- $2^{(n/2)} \leq T(n) \leq 2^n \rightarrow (1.4)^n \leq T(n) \leq (2)^n$

How to use less memory?

Iterative solution with array:

```
def fib(n):
  A = [0, 1]
  for i in range(2, n + 1):
    A.append(A[i-1] + A[i-2])
  return A[n]
```

This improves the time efficiency to O(n) with space complexity of O(n).

Optimized memory usage:

```
def fib(n):
  if n <= 1:
    return n
  a, b = 0, 1
  for i in range(2, n + 1):
    a, b = b, a + b
  return b
```

Now the space complexity is O(1).

Matrices and Fibonacci Numbers

The Fibonacci sequence can also be represented using matrices. Refer to this explanation on Stack Exchange for a detailed proof using matrices.

Mathematical Theories and Series

Arithmetic Series

- **Formula:** The sum of the first n natural numbers is given by the expanded series $S_n = 1 + 2 + 3 + \ldots + n$, which simplifies to $S_n = \frac{n \times (n+1)}{2}$.
- **Complexity:** The computational complexity for calculating the sum is $O(n^2)$ due to the nature of the operations involved, considering each addition as a separate operation.

Geometric Series

- **Formula:** The sum of a geometric series with the first term as 1 and a common ratio of 2, expanded up to n terms, is $S_n = 1 + 2 + 2^2 + 2^3 + \ldots + 2^n$, which simplifies to $S_n = 2^{(n+1)} - 1$.
- **Complexity:** The series grows exponentially, leading to a complexity of $O(2^n)$.

Harmonic Series

- **Formula:** The sum of the harmonic series up to n terms is represented by the expanded series $S_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}$, which is approximately $\ln(n) + \gamma$, where γ (Euler's constant) is roughly 0.57721.
- **Characteristics:** The sum of the harmonic series grows logarithmically as n increases, being asymptotically close to $\ln(n)$ but not exactly equal to $\log_e(n)$.

Logarithm Formula

change of Base:

$$\log_b(a) = \frac{\log_c(a)}{\log_c(b)}$$

logarithmic property of a product:

$$\log(a \cdot b) = \log(a) + \log(b)$$

Asymptotic Notations

Asymptotic notations are crucial for comparing the growth rates of functions, particularly in the context of algorithm analysis. Below are two tables summarizing these notations and the typical functions (logarithmic, polynomial, and exponential) used in algorithm analysis.

Table 1: Asymptotic Notations

Comparison	Notation	Description	Use Case
>	Big O (O)	Upper bound, worst-case scenario	Algorithm won't be slower than this rate
<	Big Omega (Ω)	Lower bound, best-case scenario	Algorithm won't be faster than this rate
=	Big Theta (Θ)	Tight bound, exact growth rate	Algorithm runs in this time in all cases
≥	Little o (o)	Non-tight upper bound, asymptotically smaller	Algorithm is faster than this rate, but not by how much

Comparison	Notation	Description	Use Case
≤	Little omega (ω)	Non-tight lower bound, asymptotically larger	Algorithm is slower than this rate, but not by how much

Table 2: Function Types in Algorithm Analysis

Function Type	Description	Complexity	Example in Algorithms
Logarithmic	Divides problem into smaller parts each step	Very efficient	Binary Search, Divide and Conquer algorithms
Polynomial	Input size raised to a constant power	Moderately efficient	Bubble Sort (n^2), Insertion Sort (n^2)
Exponential	Growth doubles with each input addition	Highly inefficient	Brute-force algorithms, NP-complete problems

Examples and Comparisons

- **Big O Example:** If an algorithm's running time increases linearly with the input size, it can be described as $O(n)$.
  - **Given:** Compare $f(n) = (\log_2(n))^2$ and $g(n) = n$ as n approaches infinity.
  - **Conclusion:** $f(n) = O(g(n))$, indicating $f(n)$ grows slower than $g(n)$ and is therefore bounded above by $g(n)$ for large n.
- **Big Omega Example:** An algorithm with a running time that never goes below a linear relationship with the input size can be described as $\Omega(n)$.
  - **Given:** Compare $f(n) = (1.4)^n$ and $g(n) = n^{10000}$.
  - **Conclusion:** $f(n) = \Omega(g(n))$, showing $f(n)$ grows faster than $g(n)$ for large n and is therefore bounded below by $g(n)$.
- **Big Theta Example:** An algorithm that has both upper and lower bounds linearly dependent on the input size can be described as $\Theta(n)$.
  - **Given:** Compare $f(n) = n$ and $g(n) = 2n$.
  - **Conclusion:** $f(n) = \Theta(g(n))$, indicating $f(n)$ and $g(n)$ grow at the same rate, and $f(n)$ is tightly bounded by $g(n)$.
- **Little o Example:** An algorithm that grows slower than n^2 but faster than n can be described as $o(n^2)$.
  - **Given:** Compare $f(n) = \log_2^n$ and $g(n) = (\log_2^n)^2$.

    - **Conclusion:** $f(n) = o(g(n))$, indicating $f(n)$ grows strictly slower than $g(n)$ and is not a tight upper bound.
- **Little omega Example:** An algorithm that grows faster than any polynomial function of n but is not exponential can be described as $\omega(n^k)$ for any constant k.
    - **Given:** Compare $f(n) = 2^n$ and $g(n) = 3^n$.
    - **Conclusion:** $f(n) = \omega(g(n))$, showing $f(n)$ grows strictly faster than $g(n)$ and is not a tight lower bound.

Additional Examples

- **Given $f(n) = O(g(n))$:** There exist constants c and n_0 such that for all $n > n_0$, $f(n) \leq c \cdot g(n)$.
- **Comparing $f(n) = 2^n$ and $g(n) = 3^n$:** $f(n) = O(g(n))$, as 2^n grows slower than 3^n for large n.
- **Comparing $f(n) = \log_2^n$ and $g(n) = \log_3^n$:** $f(n) = \Theta(g(n))$, since logarithmic functions with different bases grow at the same rate.

Analyzing Limits and Growth Rates

- If $\lim\_n \to \infty \frac{f(n)}{g(n)} = 0$, then $f(n) = o(g(n))$, meaning $f(n)$ grows much slower than $g(n)$.
- If $\lim\_n \to \infty \frac{f(n)}{g(n)} = c$ (where c is a positive, non-zero constant), then $f(n) = \Theta(g(n))$, meaning $f(n)$ and $g(n)$ grow at the same rate.
- If $\lim\_n \to \infty \frac{f(n)}{g(n)} = \infty$, then $f(n) = \omega(g(n))$, indicating $f(n)$ grows much faster than $g(n)$.

Solving Recursion

Master's Theorem

The Master's Theorem provides a way to solve recurrence relations of the form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + n^c$$

where a, b, and c are constants. The solution compares the values of $\log_b^a$ and c to determine the time complexity:

- If $\log_b^a < c$, then $T(n) = \Theta(n^c)$.
- If $\log_b^a = c$, then $T(n) = \Theta(n^c \cdot \log n)$.
- If $\log_b^a > c$, then $T(n) = \Theta(n^{\log_b^a})$.

Example:

Consider $T(n) = 2T\left(\frac{n}{2}\right) + n$ where $a = 2$, $b = 2$, and $c = 1$.

- Since $\log_2^2 = 1 = c$, by the Master's Theorem, $T(n) = \Theta(n \cdot \log n)$.

Recursion Tree Method

The Recursion Tree Method offers a more visual approach to solving recurrences, illustrating how the costs accumulate across the recursive calls. It requires drawing a tree where each node represents a recursive call and its associated cost. The total cost is the sum of all node costs.

Example:

Consider $2T(n) = 3T\left(\frac{n}{4}\right) + n^2$ where $a = 3$, $b = 4$, and $c = 2$.

- At the root, we have the cost n^2.
- At the next level, there are 3 nodes, each with a cost of $\left(\frac{n}{4}\right)^2$.

This pattern continues until the size of the problems becomes 1. To find the total cost, sum the costs at each level of the tree, then sum these totals.

Textual Representation of a Recursion Tree:

```
Level 0:          T(n)            -> Cost: n^2
                 /  |  \
Level 1:     T(n/4) T(n/4) T(n/4)  -> Cost: 3 * (n/4)^2
            / | \  / | \  / | \
Level 2:   ...  ...  ...  ...      -> Cost: 3^2 * (n/16)^2
            ..............
            ..............
```

Total Time Complexity Calculation using Recursion Tree Method

Given Recurrence Relation:

$$2T(n) = 3T\left(\frac{n}{4}\right) + n^2$$

Recursion Tree Levels:

- **Level 0:** $T(n)$ with cost n^2
- **Level 1:** 3 nodes, each $T\left(\frac{n}{4}\right)$ with total cost $3 \times \left(\frac{n}{4}\right)^2$
- **Level 2:** 3^2 nodes, each $T\left(\frac{n}{4^2}\right)$ with total cost $3^2 \times \left(\frac{n}{4^2}\right)^2$

- **...**
- **Level k:** 3^k nodes, each $T\left(\frac{n}{4^k}\right)$ with total cost $3^k \times \left(\frac{n}{4^k}\right)^2$

Cost at Each Level:

1. **Level 0:** n^2
2. **Level 1:** $3 \times \left(\frac{n}{4}\right)^2 = \frac{3}{16}n^2$
3. **Level 2:** $3^2 \times \left(\frac{n}{4^2}\right)^2 = \frac{3^2}{4^4}n^2$
4. **...**
5. **Level k:** $3^k \times \left(\frac{n}{4^k}\right)^2 = \frac{3^k}{4^{2k}}n^2$

Total Time Complexity:

The total time complexity is the sum of costs across all levels. Assuming the tree has h levels, the total cost $T(n)$ is:

$$T(n) = n^2 + \frac{3}{16}n^2 + \frac{3^2}{4^4}n^2 + \ldots + \frac{3^h}{4^{2h}}n^2$$

This series continues until the base case is reached, which happens when $\frac{n}{4^h} = 1$, implying $h = \log_4 n$.

Conclusion:

To find the exact total time complexity, we would sum the geometric series formed by the costs at each level. However, this requires determining when the series converges, which depends on the depth of the recursion tree and the specific terms of the series. For most practical purposes, analyzing the dominant term (usually the largest term as n grows) provides a good approximation of the total time complexity.