Recursion Week 2.1

Mayank Pratap Singh

July 2024

Check out other DSA Notes

Recursion

Recursion

Recursion is when a function calls itself to solve a problem. It has two main parts:

1. **Base Case**: The condition under which the recursion ends.
2. **Recursive Case**: The part of the function where it calls itself.

Factorial of a Number

The factorial of a number n (denoted $n!$) is the product of all positive integers less than or equal to n.

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

In recursion:

$$\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times 1! \\
1! &= 1 \\
0! &= 1
\end{aligned}$$

Pseudocode

```
function factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Explanation

- **Base Case**: If n is less than or equal to 1, return 1.
- **Recursive Case**: Return $n\times$ factorial of $(n - 1)$.

Actual Code

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Stack Data Structure

Recursion uses a stack data structure (LIFO - Last In First Out) to store the function calls.

$$\begin{tikzcd}
\text{fact}(5) \arrow[d] \\
\text{fact}(4) \arrow[d] \\
\text{fact}(3) \arrow[d] \\
\text{fact}(2) \arrow[d] \\
\text{fact}(1)
\end{tikzcd}$$

Time and Space Complexity

- **Time Complexity**: $O(n)$
- **Space Complexity**: $O(n)$

Recurrence Relation

Recurrence relations are equations that define sequences recursively. They express each term of the sequence as a function of the preceding terms.

Example: Factorial Function

Consider the factorial function, which we denote as $n!$. The factorial of a number n is the product of all positive integers less than or equal to n. For example:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

When we use recursion to calculate the factorial, we break down the problem into smaller subproblems.

Base Case and Recursive Case

- **Base Case**: The simplest version of the problem, which we know the answer to without any further calculations.
  - For factorial, the base case is when n is 0 or 1, because $0! = 1$ and $1! = 1$.
- **Recursive Case**: This is where the function calls itself with a smaller version of the problem.
  - For factorial, $n! = n \times (n-1)!$. This means to find $n!$, we multiply n by the factorial of $n-1$.

Recurrence Relation for Factorial

The recurrence relation for the factorial function can be written as:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1) + c & \text{if } n > 1 \end{cases}$$

Here, $T(n)$ represents the time it takes to compute $n!$.

- When $n \leq 1$, the time taken is a constant (we can call it c).
- When $n > 1$, the time taken is the time to compute $(n-1)!$ plus some constant time c for the multiplication.

Solving the Recurrence Relation

To understand the time complexity, we solve the recurrence relation step by step.

1. **Start with the relation**:
$$T(n) = T(n-1) + c$$
2. **Substitute $T(n-1)$ with its own recurrence**:
$$T(n) = (T(n-2) + c) + c$$
$$T(n) = T(n-2) + 2c$$
3. **Continue substituting**:
$$T(n) = (T(n-3) + c) + 2c$$
$$T(n) = T(n-3) + 3c$$
4. **Notice the pattern**:
$$T(n) = T(n-k) + kc$$
5. **Determine when to stop**:
   - When $k = n - 1$:
$$T(n) = T(1) + (n-1)c$$
6. **Substitute the base case $T(1)$**:
$$T(n) = c + (n-1)c$$
$$T(n) = nc$$

Since c is a constant, the time complexity $T(n)$ is $O(n)$.

Visualization

To help visualize the process, imagine you have a stack of blocks:

- Each block represents a recursive call.
- The base case is the bottom block.
- Each block above the base case represents adding a constant amount of time.

If you have n blocks, the total time is proportional to n.

Summary

- Recurrence relations help us understand how the time complexity of a recursive function grows.
- By breaking down the problem and observing patterns, we can solve these relations.
- For the factorial function, the recurrence relation shows that the time complexity is $O(n)$.

Fibonacci Series

The Fibonacci series is a series where each number is the sum of the two preceding ones.

$$1, 1, 2, 3, 5, 8, \dots$$

$$fib(5) = 8$$

Pseudocode

```
function fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Explanation

- **Base Case**: If $n \leq 1$, return 1.
- **Recursive Case**: Return the sum of fib(n-1) and fib(n-2).

Actual Code

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Time and Space Complexity

- **Time Complexity**: $O(2^n)$
- **Space Complexity**: $O(n)$

Recurrence Relation

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + c & \text{if } n > 1 \end{cases}$$

Tree Structure of Fibonacci Series

$$\begin{tikzcd}
 & & & & \text{fib}(5) \arrow[dl] \arrow[dr] & & \\
 & & & \text{fib}(4) \arrow[dl] \arrow[dr] & & \text{fib}(3) \arrow[dl] \arrow[dr] & \\
 & & \text{fib}(3) \arrow[dl] \arrow[dr] & & \text{fib}(2) \arrow[dl] \arrow[dr] & & \text{fib}(1) \\
 & \text{fib}(2) \arrow[dl] \arrow[dr] & & \text{fib}(1) & & \text{fib}(0) & \\
\text{fib}(1) & & \text{fib}(0) & & & &
\end{tikzcd}$$

Overlapping Subproblems in Recursive Solutions

In recursive algorithms, overlapping subproblems occur when the same subproblems are solved multiple times. This leads to inefficiencies as the same computation is repeated. Dynamic programming solves this by storing the results of subproblems and reusing them.

Example: Fibonacci Sequence

The Fibonacci sequence is defined as:

$$F(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

A naive recursive implementation in Python:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

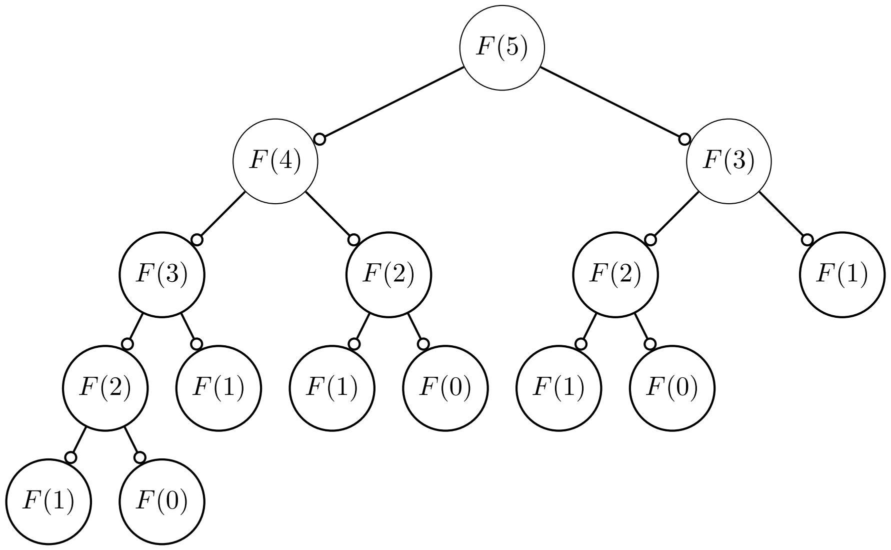

Figure 1: Recursive Call Tree for $F(5)$

We observe that $F(2)$, $F(1)$, and $F(0)$ are computed multiple times, indicating overlapping subproblems.

Recursive Tree Method

To analyze the time complexity of recursive algorithms, we use the recursive tree method. This method visualizes recursive calls as a tree and helps in summing up the total work done.

Recurrence Relation

Consider the recurrence relation for the Fibonacci sequence:

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

where c is a constant representing the base case or the work done at each node.

Recursive Tree for Fibonacci Sequence

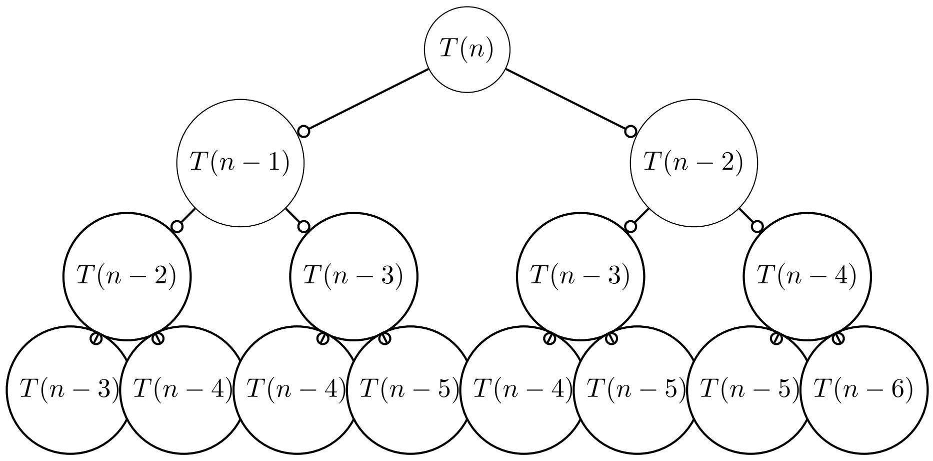

Figure 2: Recursive Tree for $T(n)$

Time Complexity Analysis

To analyze the time complexity, we sum up the work done at each level of the tree.

$$T(n) = c\left(2^0 + 2^1 + 2^2 + \ldots + 2^k\right)$$

This forms a geometric series with a common ratio $r = 2$. The sum of a geometric series is given by:

$$a(r^{n+1} - 1)/(r - 1)$$

Substituting $a = 1$, $r = 2$, and $n + 1$:

$$T(n) = c\left(\frac{2^{n+1} - 1}{2 - 1}\right) = c\left(2^{n+1} - 1\right)$$

Simplifying further:

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

Thus, the time complexity of the naive recursive Fibonacci algorithm is exponential, which is highly inefficient.

Conclusion

Understanding overlapping subproblems and recursive tree methods is crucial for optimizing recursive algorithms. By visualizing the problem as a recursive tree, we can better understand the time complexity and work done, leading to more efficient solutions such as dynamic programming.