Exam 3 Notes

Dynamic Programming

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems, solving each of these subproblems just once, and storing their solutions - ideally, using a bottom-up approach. The key idea behind DP is to avoid recalculating answers to the subproblems that are solved multiple times by caching these results. DP can be applied to a wide range of problems, from optimization problems to counting and decision problems.

Key Concepts

1. **Overlapping Subproblems**: DP is used when the problem can be divided into subproblems which are reused several times.

2. **Optimal Substructure**: A problem has an optimal substructure if an optimal solution to the whole problem can be constructed from optimal solutions of its subproblems.

3. **Memoization**: This is a top-down approach where you start solving the problem by breaking it down. If the problem has been solved, you would simply retrieve the stored answer. If not, solve it and store the answer.

4. **Tabulation**: A bottom-up approach where you solve all possible small problems and then combine them to build solutions for bigger problems.

Runtime and Spatial Complexity

The time and space complexity of DP algorithms vary greatly depending on the problem and the approach (memoization vs. tabulation). Generally, the complexity depends on the number of distinct states (subproblems) that need to be solved and stored.

- **Memoization**: The time complexity is often $\mathcal{O}(N *$ complexity of recursion), where N is the number of subproblems uniquely solved. The space complexity includes the storage for memoization and the call stack for recursion.

- **Tabulation**: The space complexity is usually similar to memoization, focusing on storing solutions of subproblems. However, it avoids the additional space required for the recursion call stack, making it more space-efficient for some problems.

Common Applications and Examples

1. **Fibonacci Series**: A classic example where DP can drastically reduce the time complexity from exponential in naive recursion to linear by storing the results of the Fibonacci numbers already calculated.

2. **Knapsack Problem**: Used to determine the maximum value that can be put in a knapsack of a given capacity by using a tabulated DP approach.

3. **Shortest Paths in Graphs (e.g., Floyd-Warshall, Bellman-Ford algorithms)**: DP is used to find shortest paths in a weighted graph with positive or negative edge weights but with no negative cycles.

4. **Coin Change Problem**: Determines the minimum number of coins that make a given value, using a bottom-up DP approach to build up solutions to all values up to the target.

Dynamic Programming is a powerful technique that requires identifying the problem's structure to effectively apply it. By understanding the concept of overlapping subproblems and optimal substructure, you can utilize DP to solve a variety of complex problems more efficiently.

Memoization

Memoization is a technique used in computing to speed up the execution of programs by storing the results of expensive function calls and reusing them when the same inputs occur again. It's a critical concept within Dynamic Programming (DP), enabling it to efficiently solve problems with overlapping subproblems. Here, we delve into memoization, focusing on its mechanism, complexities, and applications.

Mechanism

1. **Storage**: When a function is called, its result is stored in a data structure (e.g., an array or a hash table) with its parameters as the key.
2. **Lookup**: Upon subsequent calls with the same parameters, the function first checks the data structure. If the result is present, it's returned immediately, avoiding the recomputation.
3. **Recursive Calls**: Memoization is commonly applied in recursive algorithms where the same computations are repeated multiple times.

Runtime Complexity

The runtime complexity of a memoized algorithm depends on the number of unique states or subproblems to solve. If a problem has N unique states and the computation for each state is $\mathcal{O}(1)$ (excluding recursive calls), then the total runtime complexity would be $\mathcal{O}(N)$. This significantly reduces the time from what could be exponential without memoization in problems with overlapping subproblems.

Spatial Complexity

The space complexity is primarily determined by the number of unique function calls that need to be stored. This can vary from linear to potentially very high, depending on the problem's scope and the dimensions of the memoization table. The auxiliary space for the call stack should also be considered, especially for recursive solutions.

Applications and Examples

1. **Fibonacci Sequence**: Instead of computing Fibonacci numbers recursively in $\mathcal{O}(2^N)$ time, memoization allows solving it in $\mathcal{O}(N)$ by storing previously computed Fibonacci numbers.
2. **Longest Common Subsequence (LCS)**: Memoization can reduce the time complexity from exponential to $\mathcal{O}(M * N)$, where M and N are the lengths of the two sequences by caching the results of LCS lengths for different pairs of prefixes.
3. **Matrix Chain Multiplication**: Determines the most efficient way to multiply a series of matrices. Memoization stores the results of the minimum number of multiplications needed for a given chain length, converting an exponential problem into a polynomial one.

Advantages of Memoization

- **Efficiency**: Greatly improves the efficiency of algorithms by avoiding redundant calculations.
- **Simplicity**: Easy to implement in a recursive solution with minimal changes.

Considerations

- **Memory Usage**: While memoization accelerates computation, it can increase memory usage significantly, which might be a concern for space-constrained environments.
- **Problem Suitability**: Not all problems benefit from memoization. It's most effective when the problem has overlapping subproblems and a recursive structure.

Memoization is a cornerstone technique in optimizing recursive algorithms, particularly in the realm of Dynamic Programming. By understanding and applying memoization effectively, you can solve a wide range of complex problems more efficiently, both in terms of time and computational resources.

Greedy Algorithms

Greedy Algorithms are a class of algorithms that build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit or that appears to be the best solution at the moment. This approach can lead to globally optimized solutions for some problems and only locally optimized solutions for others, depending on the problem's structure.

Key Concepts

1. **Local Optimum Choices**: Greedy algorithms make decisions from the given solution domain based on some local selection criterion.
2. **No Backtracking**: Once a choice is made, the algorithm never revisits or reverses this decision, which differentiates it from other techniques like backtracking or dynamic programming that might explore multiple possibilities before settling on a solution.

Runtime Complexity

- The time complexity of greedy algorithms varies significantly across different problems but is generally efficient for the problems they are applicable to. For example, the runtime for sorting algorithms can be $\mathcal{O}(n \log(n))$, Greedy Best First Search can operate in polynomial time, and constructing a Minimum Spanning Tree (MST) with algorithms like Prim's or Kruskal's can also be achieved efficiently.
- The efficiency of greedy algorithms often comes from their nature of making a single pass through the problem's data, making decisions that seem best at the moment without considering the future consequences in detail.

Spatial Complexity

Space complexity for greedy algorithms is typically lower than that for dynamic programming solutions because they do not need to store all the subproblem solutions. For example, in the case of algorithmic approaches for finding MSTs, the space complexity primarily depends on the representation of the graph (e.g., adjacency list or adjacency matrix) and the additional structures used (like priority queues in Prim's algorithm), which is often $\mathcal{O}(V)$ or $\mathcal{O}(E)$.

Applications and Examples

1. **Huffman Coding**: A compression algorithm that assigns variable length codes to input characters, with shorter codes for more frequent characters. This is a pure greedy approach.
2. **Activity Selection Problem**: Given a set of activities with their start and end times, the goal is to select the maximum number of activities that don't overlap. Greedy algorithms solve this by always picking the next activity that ends the earliest.
3. **Minimum Spanning Tree (MST)**: Algorithms like Kruskal's and Prim's algorithm are used to find the MST of a graph, which is a subset of the edges that connects all vertices in the graph with the minimum total edge weight.
4. **Dijkstra's Algorithm**: Used for finding the shortest paths from a single source vertex to all other vertices in a graph with non-negative edge weights.

Advantages of Greedy Algorithms

Greedy algorithms are generally more straightforward to conceptualize and can be more efficient in terms of runtime and space for the problems they are suitable for.

Considerations

A major consideration when using a greedy algorithm is whether a greedy choice will lead to an optimal solution. Greedy algorithms work best when every step is a choice that leads to an optimal solution, which is not guaranteed for all problems.

Greedy algorithms are powerful due to their simplicity and the efficiency with which they can solve certain classes of problems. They are particularly useful when a problem has a structure that guarantees that local optimal choices can lead to a global optimum, making them an essential tool in the algorithmic toolbox.

Graph Theory

Graph theory is a fundamental area of computer science and mathematics that involves the study of graphs, which are abstract models used to represent a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by vertices (or nodes), and the links that connect them are called edges. Within graph theory, two important traversal techniques are Breadth-First Search (BFS) and Depth-First Search (DFS). These techniques are pivotal for exploring nodes and edges of a graph and have varied applications, complexities, and characteristics.

Breadth-First Search (BFS)

BFS starts at a selected node (the source) and explores all of its nearest neighbors before moving to their neighbors, spreading outwards from the source in a wave-like manner. BFS is implemented using a queue to keep track of the next location to visit.

- **Functioning**: BFS explores vertices in the order of their distance from the source node, layer by layer. It ensures that for any reachable vertex v, BFS visits v only after visiting all vertices at a distance $d-1$ from the source, where d is the distance from the source to v.
- **Complexity**: The time complexity of BFS is $\mathcal{O}(V+E)$ for a graph represented using an adjacency list, where V is the number of vertices and E is the number of edges. The space complexity is $\mathcal{O}(V)$, as it needs to store a queue of vertices to explore.
- **Applications**: BFS is used in shortest path algorithms, algorithm to find connected components in an undirected graph, and in algorithms for computing the minimum spanning tree.

Depth-First Search (DFS)

DFS explores a graph by going as far into the graph as possible before backtracking to explore other paths. It's implemented using recursion or a stack to keep track of the exploration path.

- **Functioning**: DFS starts at the source node and explores as far as possible along each branch before backtracking. This means it will explore one neighbor of a node, then one neighbor of this neighbor, and so on, until it reaches an unexplored node or an endpoint.
- **Complexity**: The time complexity of DFS is also $\mathcal{O}(V+E)$ for a graph represented using an adjacency list. The space complexity, primarily due to the stack used for recursion, is $\mathcal{O}(V)$ in the worst case.
- **Applications**: DFS is utilized for topological sorting, detecting cycles in a graph, and finding strongly connected components.

Differences Between BFS and DFS

- **Strategy**: BFS explores neighbors first, making it suitable for finding the shortest path, while DFS dives deep into one neighbor before exploring others, useful for tasks that need to explore all possible paths.
- **Implementation**: BFS uses a queue to achieve its level-by-level exploration, whereas DFS uses a stack (either explicitly or implicitly through recursion) to keep track of the vertices to be explored.
- **Space Complexity**: In a sparse graph, both have similar space complexities, but their behavior might differ significantly in dense graphs or in their applications.

Tree, Back, Forward, and Cross Edges (in the context of DFS)

- **Tree Edges**: In a DFS forest, tree edges are those that are part of the original graph and connect vertices to their descendants in the DFS tree.
- **Back Edges**: These are edges that point from a node to one of its ancestors in the DFS tree. Back edges are critical for detecting cycles in directed graphs.
- **Forward Edges**: These edges connect a node to a descendant in the DFS tree, but they are not tree edges. They essentially skip over parts of the tree.
- **Cross Edges**: These are edges that connect nodes across branches of the DFS tree, neither to ancestors nor to descendants.

Rod Cutting Problem

The Rod Cutting Problem is a classic optimization problem that exemplifies the power of Dynamic Programming (DP). The problem statement is as follows:

'Given a rod of length n and a table of prices $p[i]$ for $i = 1, 2, \ldots, n$, determine the maximum revenue $r[n]$ obtainable by cutting up the rod and selling the pieces.'

Without Dynamic Programming

Without DP, a naive approach to solving the Rod Cutting Problem would involve checking all possible ways of cutting the rod and comparing the total revenue from each combination to find the maximum. This can be done using recursion, where the maximum revenue for a rod of length n is the maximum of $p[i] + r[n-i]$ for $i = 1, 2, \ldots, n$. However, this recursive approach recomputes solutions for the same subproblems many times and has an exponential time complexity of $\mathcal{O}(2^n)$, which is highly inefficient for larger values of n.

With Dynamic Programming

DP improves performance by systematically solving the subproblems only once and storing their solutions in a table from which we can reconstruct the final solution. This approach ensures that each subproblem is solved exactly once, eliminating the redundancy of the recursive approach.

The memoization strategy involves writing a recursive function that solves the problem for a rod of length n by dividing it into two parts at each step: a part of length i and a part of length $n - i$, for all i in $1, 2, \ldots, n$. Before making the recursive call to solve the problem for $n - i$, the function checks whether the value has already been computed. If it has, the function uses the stored value; if not, it computes and then stores the value.

Memoized Solution To The Rod Cutting Problem

Improvements with Memoization

By using memoization, the algorithm ensures that the work done for each subproblem of length k is not repeated. As a result, each subproblem is solved exactly once, and the results are reused, leading to significant performance improvements. We can see an implementation of this strategy below.

```python
def cut_rod_memoized(prices, n, revenue):
    if revenue[n] >= 0:
        return revenue[n]

    if n == 0:
        max_revenue = 0
    else:
        max_revenue = -float('inf')
        for i in range(1, n+1):
            max_revenue = max(max_revenue, prices[i] + cut_rod_memoized(prices, n-i, revenue))

    revenue[n] = max_revenue
    return max_revenue

def cut_rod(prices, n):
    revenue = [-float('inf')] * (n+1)
    return cut_rod_memoized(prices, n, revenue)

# Example usage:
prices = [0, 1, 5, 8, 9, 10, 17, 17, 20]  # Assuming 1-indexed prices array
rod_length = 8
print("Maximum Revenue:", cut_rod(prices, rod_length))

```

In this memoized version, `cut_rod_memoized` is the recursive function that computes the maximum revenue, with revenue serving as the memoization array. When `cut_rod` is called, it initializes this array with negative values to indicate that the subproblem solutions are initially uncomputed and then calls `cut_rod_memoized` to get the maximum revenue.

Runtime Complexity

With memoization, the runtime complexity improves to $\mathcal{O}(n^2)$ from the naive recursive approach's $\mathcal{O}(2^n)$. Although the memoized solution involves the same number of subproblems as the bottom-up approach, it differs by solving them in a top-down manner.

Spacial Complexity

The space complexity remains $\mathcal{O}(n)$ because we need to store the maximum revenue for each length up to n. However, because this is a recursive approach, there's also the added space complexity of the call stack, which in the worst case, can grow to $\mathcal{O}(n)$ as well.

Coin Changing Problem

The Coin Change Problem is a classic problem that asks:

'Given an unlimited supply of coins of given denominations, how many different ways can we make change for a particular amount of money?'

The problem can be solved efficiently using Dynamic Programming, specifically with the memoization technique.

Without Dynamic Programming

A naive approach might use brute-force recursion to try every combination of coins, which results in exponential time complexity due to the redundant calculation of subproblems.

With Dynamic Programming

Memoization avoids redundant calculations by storing the results of subproblems in some form of table for quick lookup. We define a recursive function that takes the amount to change and the number of coins as its arguments and returns the number of ways to make change.

Memoized Solution To The Coin Changing Problem

Improvements with Memoization

Memoization reduces the time complexity from exponential to polynomial. Each subproblem is uniquely defined by the remaining amount and the set of coins to be considered. Once computed, the result is stored. The solution to this problem using DP can be seen below.

```python
def count_coin_change_ways(coins, amount, index, memo):
    # Base cases
    if amount == 0:
        return 1  # One way to make change for 0, which is no coins
    if amount < 0 or index == len(coins):
        return 0  # No way to make change

    # Check the memo table to avoid re-computation
    if memo[amount][index] != -1:
        return memo[amount][index]

    # Recursive breakdown: include the coin vs exclude the coin
    # Include the coin: reduce the amount, keep the index same
    count_including_coin = count_coin_change_ways(coins, amount - coins[index], index, memo)

    # Exclude the coin: keep the amount same, go to next index
    count_excluding_coin = count_coin_change_ways(coins, amount, index + 1, memo)

    # Store the computed value in the memo table
    memo[amount][index] = count_including_coin + count_excluding_coin

    return memo[amount][index]

def coin_change(coins, amount):
    # Initialize the memo table with -1
    memo = [[-1 for _ in range(len(coins))] for _ in range(amount + 1)]
    return count_coin_change_ways(coins, amount, 0, memo)

# Example usage:
coins = [1, 2, 5]  # Coin denominations
amount = 5  # Amount to make change for
print("Number of ways to make change:", coin_change(coins, amount))

```

In this example, the function `count_coin_change_ways` uses memoization to store results of subproblems in a two-dimensional list memo. The `coin_change` function initializes this memo structure and starts the recursive process. Each entry `memo[amount][index]` represents the number of ways to make change for amount using coins from `coins[index]` onwards. The use of memoization ensures that each subproblem is calculated only once, thus improving the efficiency of the solution.

Runtime Complexity

With memoization, the time complexity is $\mathcal{O}(m * n)$, where m is the number of coin denominations and n is the amount to make change for. This complexity arises because each unique subproblem is solved only once.

Spatial Complexity

The space complexity for the memoized approach is also $\mathcal{O}(m * n)$ because the algorithm needs to store the solution for each subproblem. Additionally, the recursive implementation introduces a call stack depth which, in the worst case, can be $\mathcal{O}(n)$.

Knapsack Problem

The Knapsack Problem is a classic problem in combinatorial optimization. The problem can be described as follows:

'You are given a set of n items, each with a weight $w[i]$ and a value $v[i]$, and a knapsack that can carry a maximum weight W. The goal is to determine the maximum value that the knapsack can carry.'

Without Dynamic Programming

A brute-force solution would examine all subsets of items to find the maximum value that fits in the knapsack. This approach has a time complexity of $\mathcal{O}(2^n)$, as there are 2^n possible combinations of items.

With Dynamic Programming

We define a recursive function that computes the maximum value for a given capacity and number of items. We store the results of these computations in a two-dimensional array where one dimension represents the remaining capacity and the other represents the number of items considered.

Memoized Solution To The Knapsack Problem

Improvements with Memoization

Memoization brings down the complexity significantly. Instead of solving the exponential number of subproblems, we only solve $\mathcal{O}(n * W)$ subproblems since there are at most n items and W capacity constraints to consider. The improved solution to this problem using DP can be seen below.

```python
def knapsack_memoized(values, weights, capacity, index, memo):
    # Base Case: If no items are left or capacity is 0.
    if index == len(values) or capacity == 0:
        return 0

    # If the result is already in the memo table then return that value.
    if memo[index][capacity] != -1:
        return memo[index][capacity]

    # If the weight of the item is more than the knapsack's capacity, skip this item.
    if weights[index] > capacity:
        memo[index][capacity] = knapsack_memoized(values, weights, capacity, index+1, memo)
    else:
        # Recursive call to choose the item and not to choose the item.
        value_including_item = values[index] + knapsack_memoized(values, weights, capacity - weights[
 index], index+1, memo)
        value_excluding_item = knapsack_memoized(values, weights, capacity, index+1, memo)

        # Store the maximum of including or excluding the current item.
        memo[index][capacity] = max(value_including_item, value_excluding_item)

    return memo[index][capacity]

def knapsack(values, weights, capacity):
    # Initialize the memo table with -1
    memo = [[-1 for _ in range(capacity + 1)] for _ in range(len(values))]
    return knapsack_memoized(values, weights, capacity, 0, memo)

# Example usage:
values = [60, 100, 120]  # The values of the items
weights = [10, 20, 30]   # The weight of the items
capacity = 50            # The maximum capacity of the knapsack
print("Maximum value in Knapsack =", knapsack(values, weights, capacity))
```

In this code, `knapsack_memoized` is the recursive function that computes the maximum value, and memo is a 2D array that serves as the memoization table. When `knapsack` is called, it initializes this table and starts the recursion process. The memoization ensures we do not recompute the value for any state (defined by index and capacity), hence significantly optimizing the brute-force approach.

Runtime Complexity

With memoization, the runtime complexity becomes $\mathcal{O}(n * W)$ where n is the number of items and W is the knapsack capacity. This is because each state of the DP is defined by the current item and the remaining capacity, and we solve each state only once.

Spatial Complexity

The space complexity is $\mathcal{O}(n * W)$ because we store the result for each item-capacity combination in a memoization table. If we're using recursion, we also need to consider the call stack, which can add up to $\mathcal{O}(n)$ in space in the worst case.

Longest Common Subsequence (LCS)

The Longest Common Subsequence (LCS) problem is a classic problem in computer science. The goal is to find the longest subsequence present in two sequences (which could be strings, arrays, etc.). A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous (as opposed to a substring) in both sequences.

Without Dynamic Programming

A brute-force solution to the LCS problem would examine all subsequences of both sequences and check for the longest matching one. This approach would have a time complexity of $\mathcal{O}(2^n * 2^m)$ for two sequences of length n and m, as it involves generating all subsequences for both sequences.

With Dynamic Programming

Memoization improves the solution by avoiding the recalculation of the LCS length for the same subproblem. In the context of LCS, a subproblem is defined as finding the LCS length of prefixes of the sequences.

We define a recursive function that computes the LCS length for two given indices in the sequences, one for each sequence. We then store the results in a two-dimensional array (often called a memoization table) so that each subproblem is computed only once.

Memoized Solution To The Longest Common Subsequence Problem

Improvements with Memoization

Memoization reduces the time complexity from exponential to polynomial. Each subproblem is uniquely defined by the indices within the two sequences and, once computed, will not be recomputed. The improved solution with memoization can be seen below.

```python
def lcs_memoized(seq1, seq2, i, j, memo):
    if i == 0 or j == 0:
        return 0  # Base case: If either sequence is empty

    if memo[i][j] != -1:
        return memo[i][j]  # Return memoized result if already computed

    if seq1[i-1] == seq2[j-1]:
        # If characters match, 1 + LCS of remaining sequences
        memo[i][j] = 1 + lcs_memoized(seq1, seq2, i-1, j-1, memo)
    else:
        # If no match, max of LCS of seq1 minus current char and seq2 minus current char
        memo[i][j] = max(lcs_memoized(seq1, seq2, i, j-1, memo), lcs_memoized(seq1, seq2, i-1, j,
 memo))
    return memo[i][j]

def lcs(seq1, seq2):
    memo = [[-1 for _ in range(len(seq2) + 1)] for _ in range(len(seq1) + 1)]
    return lcs_memoized(seq1, seq2, len(seq1), len(seq2), memo)

# Example usage:
seq1 = "AGGTAB"
seq2 = "GXTXAYB"
print("Length of LCS is", lcs(seq1, seq2))
```

In this code, `lcs_memoized` is the recursive function that computes the length of the LCS, and memo is a 2D array that serves as the memoization table. The function `lcs` initializes the memo table and kicks off the recursion. Each entry `memo[i][j]` represents the length of the LCS for the first i characters of `seq1` and the first j characters of `seq2`. Using memoization ensures that each subproblem is only solved once, greatly optimizing the naive recursive approach.

Runtime Complexity

With memoization, the runtime complexity is $\mathcal{O}(n * m)$, where n is the length of the first sequence and m is the length of the second sequence. This is because there are $n * m$ possible states and each state is solved only once.

Spatial Complexity

The space complexity is $\mathcal{O}(n*m)$ due to the memoization table that stores the solution for each subproblem. If recursion is used, the call stack space should also be taken into account, which could add up to $\mathcal{O}(n+m)$ in space in the worst case.

The Fibonacci Sequence

The Fibonacci sequence is a famous series in mathematics where each number is the sum of the two preceding ones, typically starting with 0 and 1. That is, $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$ for $n > 1$.

Without Dynamic Programming

A naive approach to computing the nth Fibonacci number might involve a simple recursive function. This function would call itself to obtain the two preceding Fibonacci numbers until reaching the base cases of $F(0)$ and $F(1)$. This approach does repeated work and has an exponential time complexity of approximately $\mathcal{O}(2^n)$, due to the growth of the recursion tree exponentially at each level.

With Dynamic Programming

Using memoization drastically reduces the number of function calls, thus improving the performance from exponential to linear time complexity.

Memoized Solution To The Fibonacci Sequence

Improvements with Memoization

Using memoization drastically reduces the number of function calls, thus improving the performance from exponential to linear time complexity. This improvement can be seen below.

```python
def fib_memoized(n, memo):
    if n in memo:
        return memo[n]  # Return the cached result

    if n <= 1:
        return n  # Base cases

    # Recursive calls and store the results in memo
    memo[n] = fib_memoized(n-1, memo) + fib_memoized(n-2, memo)
    return memo[n]

def fibonacci(n):
    memo = {}
    return fib_memoized(n, memo)

# Example usage:
n = 10
print(f"Fibonacci number at position {n} is {fibonacci(n)}")

```

In this implementation, `fib_memoized` is a helper function that uses a dictionary named `memo` to store the Fibonacci numbers that have already been computed. The fibonacci function initializes this memoization store and calls `fib_memoized`. This approach uses memoization to ensure that each Fibonacci number is calculated once, yielding a significant improvement in performance for large n compared to the naive recursive approach.

Runtime Complexity

The time complexity with memoization is $\mathcal{O}(n)$ because we compute each Fibonacci number once and then retrieve the result from the cache for subsequent calls.

Spatial Complexity

The space complexity is also $\mathcal{O}(n)$ because we need to store the result for each of the n Fibonacci numbers. This storage is typically done in an array or a hash map.