Lecture Notes
Data Structures and Algorithms

Dynamic Programming

1. Introduction to dynamic programming
    a. Define dynamic programming
    b. Explain the difference between dynamic programming and other techniques such as divide-and-conquer and greedy algorithms
    c. Mention common applications of dynamic programming
2. The role of overlapping subproblems in dynamic programming
    a. Explain how dynamic programming solves problems by breaking them down into smaller subproblems
    b. Discuss the importance of overlapping subproblems in dynamic programming
3. The principle of optimality in dynamic programming
    a. Introduce the principle of optimality and how it is used in dynamic programming
    b. Explain how the principle of optimality helps to determine the order in which subproblems should be solved
4. Memoization and tabulation in dynamic programming
    a. Define memoization and tabulation
    b. Explain the difference between the two techniques
    c. Discuss the pros and cons of each technique
5. Example dynamic programming problems
6. Conclusion

Introduction

Dynamic programming is a method for solving problems by breaking them down into smaller subproblems and storing the solutions to these subproblems in a table or array. This technique is called "dynamic programming" because it involves making a sequence of decisions, each of which is based on the decisions that came before it.

To use dynamic programming to solve a problem, we must first identify the subproblems that make up the larger problem. These subproblems must be overlapping, meaning that each subproblem must be used to solve one or more of the other subproblems. Once we have identified the subproblems, we can then solve the larger problem by solving the subproblems in a specific order and storing the solutions to each subproblem as we go. This allows us to avoid solving the same subproblem multiple times, which can save a significant amount of time and resources.

Dynamic programming is typically used to solve optimization problems, where we are looking to find the optimal solution (i.e. the best solution) to a problem. This can be contrasted with decision problems, where we are simply trying to determine whether a solution exists, rather than finding the best solution.

There are several key differences between dynamic programming and divide and conquer:

1. Subproblem overlap: In dynamic programming, the subproblems are required to overlap, meaning that each subproblem must be used to solve one or more of the other subproblems. In contrast, the subproblems in divide and conquer do not need to overlap.
2. Solution storage: In dynamic programming, the solutions to the subproblems are stored in a table or array and reused as needed, while in divide and conquer, the solutions to the subproblems are discarded once they have been used to solve the larger problem.
3. Order of subproblem solving: In dynamic programming, the order in which the subproblems are solved is important, as it determines which solutions get stored and how they are used to solve the larger problem. In contrast, the order in which the subproblems are solved is not important in divide and conquer.
4. Use cases: Dynamic programming is typically used to solve optimization problems, while divide and conquer is more general and can be used to solve a wide range of problems.

Overlapping Subproblems

In dynamic programming, the key to solving a problem is to identify the smaller subproblems that make up the larger problem and then solve these subproblems in the correct order. To do this effectively, the subproblems must be overlapping, meaning that each subproblem must be used to solve one or more of the other subproblems.

For example, consider the problem of computing the nth Fibonacci number. We can use dynamic programming to solve this problem by breaking it down into the following subproblems:

- $f(0) = 0$
- $f(1) = 1$
- $f(n) = f(n-1) + f(n-2)$ for $n > 1$

These subproblems overlap because the solution to f(n) depends on the solutions to f(n-1) and f(n-2). This allows us to solve the larger problem by solving the smaller subproblems and storing their solutions in an array or table.

The importance of overlapping subproblems in dynamic programming lies in the fact that they allow us to avoid solving the same subproblem multiple times. By solving each subproblem only once and then storing the solution, we can save a significant amount of time and resources. This is known as the "principle of optimality", which states that once we have solved a subproblem, the optimal solution to that subproblem will not change, no matter how it is used to solve the larger problem.

Optimality

The principle of optimality is a fundamental concept in dynamic programming that states that once we have solved a subproblem, the optimal solution to that subproblem will not change, no matter how it is used to solve the larger problem. This principle allows us to determine the order in which subproblems should be solved and to avoid solving the same subproblem multiple times.

To apply the principle of optimality to a dynamic programming problem, we must first identify the subproblems that make up the larger problem and the order in which they should be solved. This is typically done by working backwards from the final solution to the initial state of the problem. For example, consider the problem of computing the nth Fibonacci number. We can use the principle of optimality to determine that the subproblems should be solved in the following order:

1. f(0) = 0
2. f(1) = 1
3. f(2) = f(1) + f(0) = 1 + 0 = 1
4. f(3) = f(2) + f(1) = 1 + 1 = 2
5. ...
6. f(n) = f(n-1) + f(n-2)

This order ensures that each subproblem is solved only once and that the solutions to the subproblems are stored and reused as needed.

Memoization and Tabulation

Memoization and tabulation are two techniques that can be used to store and reuse the solutions to subproblems in dynamic programming. Both techniques involve storing the solutions in a table or array, but they differ in the way the solutions are stored and accessed.

Memoization is a technique in which the solutions to the subproblems are stored in a table or array, but only the solutions to subproblems that have already been solved are stored. To solve a subproblem, we first check the table to see if the solution has already been computed. If it has, we can use the stored solution; if it has not, we compute the solution and store it in the table before returning it. This technique is called "memoization" because it involves creating a "memory" of the solutions to the subproblems.

Tabulation is a technique in which the solutions to all of the subproblems are stored in a table or array, regardless of whether they have been solved or not. To solve a subproblem, we simply look up the solution in the table. This technique is called "tabulation" because it involves creating a "table" of the solutions to the subproblems.

Both memoization and tabulation can be effective techniques for storing and reusing the solutions to subproblems in dynamic programming, but they have different trade-offs in terms of space and time complexity. Memoization requires less space, as it only stores the solutions to the subproblems that have been solved, but it may require more time, as it involves checking the table to see if a solution has

already been computed. Tabulation requires more space, as it stores the solutions to all of the subproblems, but it may require less time, as it does not involve checking the table to see if a solution has already been computed.

Example Problems

To solve a dynamic programming problem, we must follow these steps:

1. Identify the subproblems that make up the larger problem. These subproblems must be overlapping, meaning that each subproblem must be used to solve one or more of the other subproblems.
2. Determine the order in which the subproblems should be solved using the principle of optimality. This typically involves working backwards from the final solution to the initial state of the problem.
3. Use memoization or tabulation to store and reuse the solutions to the subproblems as they are solved.
4. Combine the solutions to the subproblems to obtain a solution to the larger problem.

Example 1: Recursive Fibonacci

To illustrate this process, let's consider the problem of computing the nth Fibonacci number using dynamic programming.

1. Identify the subproblems:
    - f(0) = 0
    - f(1) = 1
    - f(n) = f(n-1) + f(n-2) for n > 1
2. Determine the order in which the subproblems should be solved:
    - f(0) = 0
    - f(1) = 1
    - f(2) = f(1) + f(0)
    - f(3) = f(2) + f(1)
    - ...
    - f(n) = f(n-1) + f(n-2)
3. Use memoization or tabulation to store and reuse the solutions: We can use either memoization or tabulation to store and reuse the solutions to the subproblems. Here is an example of how to do this using memoization:

```java
public class Fibonacci {
    // Table to store the solutions to the subproblems
    private static int[] table;

    // Recursive function to compute the nth Fibonacci number
    private static int fibonacci(int n) {
        // If the solution has already been computed, return it
        if (table[n] != -1) {
            return table[n];
        }
        // Otherwise, compute the solution and store it in the table
        table[n] = fibonacci(n-1) + fibonacci(n-2);
        return table[n];
    }

    public static void main(String[] args) {
        // Initialize the table with the base cases
        table = new int[n+1];
        Arrays.fill(table, -1);
        table[0] = 0;
        table[1] = 1;

        // Compute and print the nth Fibonacci number
        int n = 10;
        int result = fibonacci(n);
        System.out.println(result);
    }
}
```

This code uses a recursive function to compute the nth Fibonacci number, with the base cases (f(0) = 0 and f(1) = 1) hardcoded into the function. The solutions to the subproblems are stored in a table and reused as needed using memoization. To compute the nth Fibonacci number, we simply call the **fibonacci()** function with the desired value of n.

4. Combine the solutions to the subproblems to obtain a solution to the larger problem: To obtain the solution to the larger problem (i.e. the nth Fibonacci number), we simply call the helper function with the desired value of n. The helper function will then recursively solve the subproblems and return the solution to the nth Fibonacci number.

Example 2. The Knapsack Problem

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

1. Subproblems: The subproblems in this example are the maximum value that can be obtained by including items in the knapsack, given a specific capacity and a set of items. These subproblems overlap because the solution to each subproblem depends on the solutions to one or more other subproblems. For example, the solution to the subproblem with capacity 50 and four items depends on the solutions to the subproblems with capacity 50 and three items, capacity 30 and four items, and so on.

2. Order of subproblem solving: The order in which the subproblems are solved in this example is determined by the capacity of the knapsack and the number of items. The subproblems are solved in a bottom-up fashion, starting with the base cases (capacity 0 or no items) and working up to the final solution (capacity 50 and all items).

3. Storing and reusing solutions: The solutions to the subproblems are stored and reused using memoization. The knapsack() function checks the table to see if the solution to a subproblem has already been computed before computing it and storing it in the table.

4. Combining subproblem solutions: The knapsack() function combines the solutions to the subproblems by considering two options: including the current item in the knapsack or excluding it. The maximum of these two options is returned as the solution to the subproblem.

Complexity

The knapsack problem involves filling in a two-dimensional array with the solutions to the subproblems. The size of the array is determined by the maximum capacity of the knapsack and the number of items, which we will denote as C and n, respectively.

To fill in the array, we use a double loop that iterates over all the capacities and items. The time complexity of each iteration is O(1), since it involves a constant number of operations. Therefore, the time complexity of the double loop is $O(Cn)$.

Overall, the time complexity of the knapsack problem is **$O(Cn)$**, since it involves filling in a two-dimensional array of size C x n.

```java
import java.util.Arrays;

public class Knapsack {
    // Table to store the solutions to the subproblems
    private static int[][] table;
    // Recursive function to compute the maximum value that can be obtained
    // by including items in the knapsack
    private static int knapsack(int[] values, int[] weights, int capacity, int n) {
        // If the solution has already been computed, return it
        if (table[capacity][n] != -1)
            return table[capacity][n];
        // If the capacity is 0 or there are no more items, return 0
        if (capacity == 0 || n == 0)
            return 0;
        // If the weight of the nth item is greater than the capacity, skip it
        if (weights[n - 1] > capacity)
            return knapsack(values, weights, capacity, n - 1);
        // Otherwise, compute the maximum value by including or excluding the item
        int include = values[n - 1] + knapsack(values, weights, capacity - weights[n - 1], n - 1);
        int exclude = knapsack(values, weights, capacity, n - 1);
        int max = Math.max(include, exclude);
        // Store the solution in the table and return it
        table[capacity][n] = max;
        return max;
    }
    public static void main(String[] args) {
        // Initialize the table with -1
        int capacity = 50;
        int n = 5;
        table = new int[capacity + 1][n + 1];
        for (int[] row : table)
            Arrays.fill(row, -1);
        // Define the values and weights of the items
        int[] values = { 60, 100, 120, 180, 200 };
        int[] weights = { 10, 20, 30, 40, 50 };
        // Compute and print the maximum value that can be obtained
        int result = knapsack(values, weights, capacity, n);
        System.out.println(result);
    }
}
```

Example 3. Coin Collecting

Consider a game in which a player can collect coins by landing on squares of a grid. Some squares contain coins, while others are empty. The player starts at the top left square and can move either right or down to the adjacent squares. The goal is to collect as many coins as possible while reaching the bottom right square.

Here is a sample grid with coin values:

```
5  3  1  1
2  2  1  5
1  3  4  4
5  5  2  3
```

To solve this problem using dynamic programming, we can define a two-dimensional array table to store the solutions to the subproblems. The subproblems in this case are the maximum number of coins that can be collected from the top left square to each square in the grid. We can solve these subproblems in a bottom-up fashion, starting with the base cases (top left square) and working our way to the final solution (bottom right square).

Here is the pseudocode for the dynamic programming solution:

```
for i from 0 to m-1:
    for j from 0 to n-1:
        if i == 0 and j == 0:
            table[i][j] = grid[i][j]
        else if i == 0:
            table[i][j] = table[i][j-1] + grid[i][j]
        else if j == 0:
            table[i][j] = table[i-1][j] + grid[i][j]
        else:
            table[i][j] = max(table[i-1][j], table[i][j-1]) + grid[i][j]
return table[m-1][n-1]
```

This code uses the solutions to the subproblems to fill in the **table** array. The base cases (top left square) are hardcoded, while the other squares are filled in using the solutions to the subproblems. The **max()** function is used to choose the maximum of the two possible options (moving right or moving down) at each step.

Once the **table** array is filled in, the solution to the problem (i.e. the maximum number of coins that can be collected) is simply the value in the bottom right square (**table[m-1][n-1]**).

Relating this algorithm to the concepts that we introduced in Dynamic Programming:

1. Subproblems: The subproblems in this problem are the maximum number of coins that can be collected from the top left square to each square in the grid. These subproblems overlap because the solution to each subproblem depends on the solutions to one or more other subproblems. For example, the solution to the subproblem for the third square in the second row depends on the solutions to the subproblems for the second square in the second row and the third square in the first row.

2. Order of subproblem solving: The order in which the subproblems are solved in this problem is determined by the position of the squares in the grid. The subproblems are solved in a bottom-up fashion, starting with the base cases (top left square) and working our way to the final solution (bottom right square).

3. Storing and reusing solutions: The solutions to the subproblems are stored and reused using a two-dimensional array (**table**). The solutions are computed and stored in the **table** array as the subproblems are solved, and they are reused as needed to solve the larger problem.

4. Combining subproblem solutions: The solutions to the subproblems are combined by considering two options (moving right or moving down) at each step and choosing the one that leads to the maximum number of coins collected. The **max()** function is used to choose the maximum of these two options.

Complexity

The coin collecting problem involves filling in a two-dimensional array with the solutions to the subproblems. The size of the array is determined by the number of rows and columns in the grid, which we will denote as **m** and **n**, respectively.

To fill in the array, we use a double loop that iterates over all the rows and columns. The time complexity of each iteration is O(1), since it involves a constant number of operations. Therefore, the time complexity of the double loop is O(mn).

Overall, the time complexity of the coin collecting problem is O(mn), since it involves filling in a two-dimensional array of size mxn.