DYNAMIC PROGRAMMING

ALGORITHMS IN JAVA

Sercan Külcü | Algorithms In Java | 10.05.2023

Contents

Introduction

Dynamic programming is a problem-solving technique used in computer science and mathematics to solve complex problems by breaking them down into simpler overlapping subproblems and solving each subproblem only once. It is especially useful when the problem exhibits the property of overlapping subproblems and optimal substructure.

The main idea behind dynamic programming is to store the results of solved subproblems in a table or memoization array, so they can be reused when needed instead of recomputing them. This approach avoids redundant computations and significantly improves the efficiency of the algorithm.

Dynamic programming typically involves the following steps:

- Characterize the structure of an optimal solution: Understand the problem and identify the subproblems that need to be solved.
- Define the value of an optimal solution recursively: Express the solution of the problem in terms of solutions to smaller subproblems.
- Define the order in which the subproblems are solved: Determine the dependencies between subproblems and choose an order that ensures all the required subproblems are solved before solving a particular subproblem.
- Determine the size of the table or memoization array: Decide on the dimensions and size of the table or array needed to store the solutions to subproblems.
- Compute the values of the subproblems in a bottom-up fashion or using memoization: Fill in the table or array by solving the subproblems in a systematic way, either starting from the smallest subproblems and building up to the larger ones (bottom-up) or by recursively solving subproblems and storing their results for future use (top-down with memoization).

- Construct an optimal solution from the computed information: Once the table or array is filled, trace back the decisions made during the computation to construct the optimal solution.

Dynamic programming is commonly used to solve optimization problems, such as finding the shortest path in a graph, the maximum sum subarray, or the optimal way to multiply matrices. It is also utilized in various algorithms and applications, including graph algorithms, sequence alignment, resource allocation, and more.

Overall, dynamic programming provides an efficient and systematic approach to solving complex problems by breaking them down into simpler subproblems and reusing the solutions to these subproblems. It is a powerful technique that can lead to significant performance improvements in algorithm design.

Common problem types

Dynamic programming can be applied to solve a wide range of problems. Here are some common problem types that can be effectively solved using dynamic programming:

- Fibonacci series: Computing the nth Fibonacci number efficiently using memoization or bottom-up techniques.
- Shortest path problems: Finding the shortest path between two vertices in a graph, such as Dijkstra's algorithm or Bellman-Ford algorithm.
- Longest common subsequence: Finding the longest subsequence that two sequences have in common.
- Knapsack problem: Determining the most valuable combination of items to include in a knapsack, given a weight constraint.
- Matrix chain multiplication: Optimizing the order of multiplying matrices to minimize the total number of multiplications.
- Coin change problem: Finding the minimum number of coins required to make a certain amount of change.
- Edit distance: Computing the minimum number of operations (insertion, deletion, substitution) required to transform one string into another.
- Subset sum problem: Determining if there is a subset of a given set that adds up to a specified target value.
- Maximum subarray sum: Finding the contiguous subarray with the largest sum within a given array.
- Traveling salesman problem: Determining the shortest possible route that visits a set of cities and returns to the starting point.
- Assembly line scheduling: Optimizing the sequence of tasks in a production system to minimize the overall processing time.
- Optimal binary search trees: Constructing a binary search tree with minimum expected search time for a given set of keys.

These are just a few examples, and dynamic programming can be applied to many other problems with overlapping subproblems and optimal substructure. The key is to identify the underlying structure and relationships between subproblems to leverage dynamic programming techniques effectively.

The principle of optimality and Bellman's equation

The principle of optimality and Bellman's equation are fundamental concepts in dynamic programming that underpin its effectiveness in solving optimization problems.

Principle of Optimality:

The principle of optimality, proposed by Richard Bellman, states that an optimal solution to a problem contains within it optimal solutions to its subproblems. In other words, if we have an optimal solution to a larger problem, then the subproblems within that solution must also be solved optimally. This principle allows dynamic programming to break down complex problems into smaller subproblems and solve them independently, with the assurance that the optimal solutions will eventually lead to an optimal solution for the overall problem.

Bellman's Equation:

Bellman's equation is a recursive formulation that expresses the value of an optimal solution to a problem in terms of the values of its subproblems. It provides a mathematical foundation for solving problems using dynamic programming. Bellman's equation is typically written in the form of a recurrence relation, which relates the optimal value of a problem to the optimal values of its subproblems.

For example, consider a problem where we need to find the maximum sum of a subarray within a given array. Let's define a function V(i) as the maximum sum of a subarray ending at index i. Bellman's equation for this problem can be written as:

$V(i) = \max(arr[i], V(i-1) + arr[i])$

In this equation, arr[i] represents the element at index i in the array, and V(i-1) represents the maximum sum of a subarray ending at the previous

index. The equation states that the maximum sum of a subarray ending at index i is either the element at index i itself or the sum of the element at index i and the maximum sum of a subarray ending at index i-1.

By applying Bellman's equation iteratively for all indices, we can compute the maximum sum of a subarray for the entire array. This is an example of the bottom-up approach in dynamic programming, where we build solutions for smaller subproblems and use them to solve larger subproblems until we reach the desired solution.

Bellman's equation provides a recursive structure that allows dynamic programming algorithms to efficiently solve problems by breaking them down into overlapping subproblems and using the principle of optimality to compute optimal solutions. It forms the basis for many dynamic programming algorithms and is a key tool in designing efficient solutions for optimization problems.

Memoization and tabulation

Memoization and tabulation are two common techniques used in dynamic programming to optimize the computation of subproblems and store their results for future use. Both techniques aim to avoid redundant calculations and improve the efficiency of dynamic programming algorithms. Let's explore each technique:

Memoization:

Memoization involves storing the results of solved subproblems in a memoization table or cache. When a subproblem needs to be solved, the algorithm first checks if the result is already present in the table. If it is, the stored result is directly returned, avoiding redundant computation. If the result is not present, the subproblem is solved, and its result is stored in the table for future use.

Memoization is typically implemented using recursion. The algorithm maintains a memoization table that maps subproblem inputs to their corresponding results. Before solving a subproblem, it first checks the memoization table. If the result is found, it is returned; otherwise, the subproblem is computed, and its result is stored in the table before returning it. This technique ensures that each subproblem is solved only once.

Memoization is particularly useful for problems with overlapping subproblems, as it allows the algorithm to avoid redundant computations. It is often employed in a top-down approach, where the algorithm starts with the main problem and recursively breaks it down into smaller subproblems. Memoization can be seen as a trade-off between time complexity and space complexity, as it can save computation time but requires additional space to store the memoization table.

Tabulation:

Tabulation, also known as bottom-up dynamic programming, involves solving subproblems iteratively and building up the solutions from the bottom to the top. Instead of using recursion and memoization, tabulation relies on a table or array to store the results of subproblems in a systematic manner.

In the tabulation approach, the algorithm starts by solving the smallest subproblems and fills in the table with their results. It then iteratively solves larger subproblems using the results of previously solved subproblems until the desired solution is obtained.

The tabulation table is typically a multi-dimensional array, where each cell represents the solution to a specific subproblem. The table is filled in a specific order based on the dependencies between subproblems. By using the results stored in the table, the algorithm avoids redundant computations and efficiently computes the solution to the main problem.

Tabulation is well-suited for problems with a well-defined order of subproblem computation and optimal substructure. It has a straightforward implementation and often leads to efficient and space-optimized solutions. Unlike memoization, tabulation doesn't rely on recursion, making it easier to implement in some cases.

Both memoization and tabulation are powerful techniques for optimizing dynamic programming algorithms. The choice between them depends on the problem at hand, the structure of the subproblems, and the trade-off between time complexity and space complexity.

Applications of dynamic programming

Dynamic programming is widely used to solve various optimization problems. Some of the key applications of dynamic programming include:

Shortest Path Problems:

Dynamic programming algorithms, such as Dijkstra's algorithm and Bellman-Ford algorithm, are commonly used to find the shortest path between two vertices in a graph. These algorithms build the shortest paths incrementally, considering the optimal substructure of the problem.

Knapsack Problem:

Dynamic programming can be applied to solve the knapsack problem, where a set of items with different weights and values must be selected to maximize the total value while staying within a given weight constraint. The dynamic programming approach considers all possible item selections and their corresponding weights and values to determine the optimal solution.

Sequence Alignment:

Dynamic programming is extensively used in bioinformatics and computational biology for sequence alignment problems. This includes applications such as finding the optimal alignment between DNA or protein sequences using algorithms like the Needleman-Wunsch and Smith-Waterman algorithms.

Optimal Binary Search Trees:

Dynamic programming can be employed to construct optimal binary search trees, where the goal is to minimize the expected search time. This problem involves determining the optimal arrangement of keys in a binary search tree to achieve the minimum average search time.

Matrix Chain Multiplication:

Dynamic programming is used to efficiently multiply a series of matrices in the optimal order, minimizing the total number of multiplications required. This problem has numerous applications in fields like computer graphics, robotics, and numerical computation.

Traveling Salesman Problem (TSP):

Dynamic programming algorithms, such as Held-Karp algorithm, can be used to solve the TSP, where the objective is to find the shortest possible route that visits a set of cities and returns to the starting city. Dynamic programming is used to store the optimal solutions to subproblems and build up to the final solution.

Subset Sum Problem:

Dynamic programming techniques are applied to solve the subset sum problem, where the task is to determine if there exists a subset of a given set that sums up to a specified target value. Dynamic programming allows for efficient exploration of all possible subsets and their sums.

Resource Allocation:

Dynamic programming can be utilized to optimize the allocation of limited resources among multiple activities or projects. By considering the optimal substructure of the problem, dynamic programming algorithms can determine the best allocation strategy to maximize overall efficiency or minimize costs.

These are just a few examples of how dynamic programming is applied to solve optimization problems. The technique can be adapted to various other scenarios where the problem exhibits overlapping subproblems and optimal substructure, allowing for efficient and optimal solutions.

Solutions

FIBONACCI SERIES USING MEMOIZATION:

Here is the Java code for computing the nth Fibonacci number efficiently using memoization:

```
import java.util.HashMap;


public class Fibonacci {


  private static HashMap<Integer, Integer> memo = new HashMap<>();


  public static int fib(int n) {
    if (n == 0 || n == 1) {
      return n;
    }


    if (!memo.containsKey(n)) {
      memo.put(n, fib(n - 1) + fib(n - 2));
    }


    return memo.get(n);
  }
```

```
    public static void main(String[] args) {
        System.out.println(fib(10)); // 55
    }
}
```

This code works by first checking if the nth Fibonacci number is already stored in the memoization table. If it is, then the value is returned. Otherwise, the nth Fibonacci number is computed recursively and the result is stored in the memoization table.

The time complexity of this algorithm is O(n), which means that the time it takes to compute the nth Fibonacci number increases linearly with n.

FIBONACCI SERIES USING BOTTOM-UP TECHNIQUES:

Here is the Java code for computing the nth Fibonacci number efficiently using bottom-up techniques:

```
public class Fibonacci {


    public static int fib(int n) {
        int[] fibs = new int[n + 1];
        fibs[0] = 0;
        fibs[1] = 1;


        for (int i = 2; i <= n; i++) {
            fibs[i] = fibs[i - 1] + fibs[i - 2];
        }
```

```
        return fibs[n];
    }

    public static void main(String[] args) {
        System.out.println(fib(10)); // 55
    }
}
```

This code works by first creating an array to store the first n Fibonacci numbers. The first two Fibonacci numbers are then stored in the array.

For each subsequent Fibonacci number, the value is computed by adding the two previous Fibonacci numbers.

The nth Fibonacci number is then returned from the array.

The time complexity of this algorithm is O(n), which means that the time it takes to compute the nth Fibonacci number increases linearly with n.

DIJKSTRA'S ALGORITHM

Here is the Java code for finding the shortest path between two vertices in a graph using Dijkstra's algorithm:

```
import java.util.*;

public class Dijkstra {

    private static class Node {
```

```
    int id;
    int distance;

    public Node(int id, int distance) {
        this.id = id;
        this.distance = distance;
    }
}

public static List<Integer> shortestPath(List<List<Integer>> adjList, int source, int destination) {
    // Create a set to store the nodes that have already been visited.
    Set<Integer> visited = new HashSet<>();

    // Create a priority queue to store the nodes that are yet to be visited, sorted by their distance from the source node.
    PriorityQueue<Node> pq = new PriorityQueue<>((n1, n2) -> n1.distance - n2.distance);

    // Initialize the distance of the source node to 0 and add it to the priority queue.
    pq.add(new Node(source, 0));

    while (!pq.isEmpty()) {
```

```
        // Get the node with the shortest distance from the priority queue.
        Node currentNode = pq.poll();

        // If the current node is the destination node, then we have found the shortest path.
        if (currentNode.id == destination) {
            List<Integer> path = new ArrayList<>();
            while (currentNode != null) {
                path.add(currentNode.id);
                currentNode = adjList.get(currentNode.id).get(0);
            }
            Collections.reverse(path);
            return path;
        }

        // Mark the current node as visited.
        visited.add(currentNode.id);

        // For each neighbor of the current node, add it to the priority queue if it has not been visited yet.
        for (int neighbor : adjList.get(currentNode.id)) {
            if (!visited.contains(neighbor)) {
```

```
                pq.add(new Node(neighbor, currentNode.distance + 1));
            }
        }
    }

    // If we reach this point, then there is no path from the source node to the destination node.
    return new ArrayList<>();
}

public static void main(String[] args) {
    // Create a graph.
    List<List<Integer>> adjList = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
        adjList.add(new ArrayList<>());
    }

    // Add edges to the graph.
    adjList.get(0).add(1);
    adjList.get(0).add(2);
    adjList.get(1).add(3);
    adjList.get(2).add(4);
    adjList.get(3).add(5);
```

```
        adjList.get(4).add(6);
        adjList.get(5).add(7);
        adjList.get(6).add(8);
        adjList.get(7).add(9);

        // Find the shortest path from node 0 to node 9.
        List<Integer> path = shortestPath(adjList, 0, 9);

        // Print the shortest path.
        for (int node : path) {
            System.out.print(node + " ");
        }
    }
}
```

This code works by first creating a set to store the nodes that have already been visited. A priority queue is then created to store the nodes that are yet to be visited, sorted by their distance from the source node.

The source node is initialized with a distance of 0 and added to the priority queue.

While the priority queue is not empty, the following steps are performed:

- The node with the shortest distance from the priority queue is removed.
- If the removed node is the destination node, then the shortest path has been found.

- The removed node is marked as visited.
- For each neighbor of the removed node, if it has not been visited yet, then it is added to the priority queue with its distance from the source node set to the distance of the removed node plus 1.

If the priority queue is empty, then there is no path from the source node to the destination node.

The shortest path is then printed to the console.

BELLMAN-FORD'S ALGORITHM

Here is the Java code for finding the shortest path between two vertices in a graph using Bellman-Ford's algorithm:

```
import java.util.*;


public class BellmanFord {


   private static class Node {
      int id;
      int distance;


      public Node(int id, int distance) {
         this.id = id;
         this.distance = distance;
      }
   }
```

```
public static List<Integer> shortestPath(List<List<Integer>> adjList, int source, int destination) {
    // Create a set to store the nodes that have already been visited.
    Set<Integer> visited = new HashSet<>();

    // Initialize the distance of the source node to 0 and add it to the priority queue.
    int[] distance = new int[adjList.size()];
    Arrays.fill(distance, Integer.MAX_VALUE);
    distance[source] = 0;

    // Relax all edges |V| - 1 times.
    for (int i = 0; i < adjList.size() - 1; i++) {
        for (int u = 0; u < adjList.size(); u++) {
            for (int v : adjList.get(u)) {
                if (distance[u] != Integer.MAX_VALUE && distance[v] > distance[u] + 1) {
                    distance[v] = distance[u] + 1;
                }
            }
        }
    }
```

```
        // Check for negative weight cycles.
        for (int u = 0; u < adjList.size(); u++) {
            for (int v : adjList.get(u)) {
                if (distance[u] != Integer.MAX_VALUE && distance[v] > distance[u] + 1) {
                    System.out.println("Negative weight cycle detected!");
                    return new ArrayList<>();
                }
            }
        }

        // Find the shortest path from the source node to the destination node.
        List<Integer> path = new ArrayList<>();
        int currentNode = destination;
        while (currentNode != source) {
            path.add(currentNode);
            currentNode = adjList.get(currentNode).get(0);
        }
        Collections.reverse(path);
        return path;
    }
```

```
public static void main(String[] args) {
    // Create a graph.
    List<List<Integer>> adjList = new ArrayList<>();
    for (int i = 0; i < 10; i++) {
        adjList.add(new ArrayList<>());
    }

    // Add edges to the graph.
    adjList.get(0).add(1);
    adjList.get(0).add(2);
    adjList.get(1).add(3);
    adjList.get(2).add(4);
    adjList.get(3).add(5);
    adjList.get(4).add(6);
    adjList.get(5).add(7);
    adjList.get(6).add(8);
    adjList.get(7).add(9);

    // Find the shortest path from node 0 to node 9.
    List<Integer> path = shortestPath(adjList, 0, 9);
```

```
        // Print the shortest path.
        for (int node : path) {
            System.out.print(node + " ");
        }
    }
}
```

This code works by first creating a set to store the nodes that have already been visited. The distance of the source node is initialized to 0 and added to the set.

For each edge in the graph, the distance of the destination node is updated if it is less than the current distance.

This process is repeated |V| - 1 times, where |V| is the number of vertices in the graph.

If the distance of any vertex changes after |V| - 1 iterations, then there is a negative weight cycle in the graph.

Otherwise, the shortest path from the source node to any other node can be found by following the edges in the set.

LONGEST COMMON SUBSEQUENCE

Here is the Java code for finding the longest subsequence that two sequences have in common:

```
import java.util.*;


public class LongestCommonSubsequence {
```

```
public static int longestCommonSubsequence(String s1, String s2) {
    int n = s1.length();
    int m = s2.length();

    // Create a table to store the lengths of the longest common subsequences
    // ending at each position in the strings.
    int[][] dp = new int[n + 1][m + 1];

    // Initialize the table.
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
            } else if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    // Return the length of the longest common subsequence.
```

```
        return dp[n][m];
    }

    public static void main(String[] args) {
        String s1 = "ABCD";
        String s2 = "ABDC";

        System.out.println(longestCommonSubsequence(s1, s2)); // 4
    }
}
```

This code works by first creating a table to store the lengths of the longest common subsequences ending at each position in the strings.

The table is initialized with all values set to 0.

For each position in the strings, the table is updated by taking the maximum of the following:

- The value in the table at the previous position in the first string and the previous position in the second string.
- The value in the table at the previous position in the first string and the current position in the second string, if the characters at these positions are equal.
- The value in the table at the current position in the first string and the previous position in the second string, if the characters at these positions are equal.

The length of the longest common subsequence is then returned by taking the value in the table at the last position in the strings.

KNAPSACK PROBLEM

Here is the Java code for determining the most valuable combination of items to include in a knapsack, given a weight constraint:

```
import java.util.*;


public class Knapsack {


  public static List<Item> mostValuableItems(List<Item> items, int capacity) {

      // Create a table to store the maximum value of items that can be included in the knapsack

      // up to each weight.

      int[][] dp = new int[items.size() + 1][capacity + 1];


      // Initialize the table.

      for (int i = 0; i <= items.size(); i++) {

        for (int j = 0; j <= capacity; j++) {

          if (i == 0 || j == 0) {

            dp[i][j] = 0;

          } else {

            dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - items.get(i - 1).weight] + items.get(i - 1).value);

          }

        }
```

```
    }

    // Find the index of the last row where the maximum value is greater than or equal to the capacity.
    int lastIndex = -1;
    for (int i = items.size(); i >= 0; i--) {
        if (dp[i][capacity] >= capacity) {
            lastIndex = i;
            break;
        }
    }

    // If no such index exists, then there are no items that can be included in the knapsack.
    if (lastIndex == -1) {
        return new ArrayList<>();
    }

    // Create a list to store the items that can be included in the knapsack.
    List<Item> includedItems = new ArrayList<>();
    for (int i = lastIndex; i > 0; i--) {
        if (dp[i][capacity] == dp[i - 1][capacity - items.get(i - 1).weight] + items.get(i - 1).value) {
```

```
            includedItems.add(items.get(i - 1));
            capacity -= items.get(i - 1).weight;
        }
    }

    // Return the list of included items.
    return includedItems;
}

public static void main(String[] args) {
    // Create a list of items.
    List<Item> items = new ArrayList<>();
    items.add(new Item("Item 1", 10, 100));
    items.add(new Item("Item 2", 20, 200));
    items.add(new Item("Item 3", 30, 300));

    // Find the most valuable combination of items to include in a knapsack with a capacity of 50.
    List<Item> includedItems = mostValuableItems(items, 50);

    // Print the list of included items.
    for (Item item : includedItems) {
        System.out.println(item.name);
```

```
        }
    }
}
```

This code works by first creating a table to store the maximum value of items that can be included in the knapsack up to each weight.

The table is initialized with all values set to 0.

For each item, the table is updated by taking the maximum of the following:

- The value in the table at the previous row and the previous column.
- The value in the table at the previous row and the current column, if the weight of the item is less than or equal to the current column.

The list of included items is then created by starting at the last row and working backwards, adding items to the list if the value in the table at the current row and column is greater than or equal to the capacity.

MATRIX CHAIN MULTIPLICATION

Here is the Java code for optimizing the order of multiplying matrices to minimize the total number of multiplications:

```
import java.util.*;


public class MatrixChainMultiplication {


    public static int mcm(int[] p) {

        // Create a table to store the minimum number of multiplications
```

```
        // required to multiply the matrices in each chain.
        int[][] dp = new int[p.length][p.length];

        // Initialize the table.
        for (int i = 0; i < p.length; i++) {
            for (int j = 0; j < p.length; j++) {
                if (i == j) {
                    dp[i][j] = 0;
                } else {
                    dp[i][j] = Integer.MAX_VALUE;
                }
            }
        }

        // Fill the table.
        for (int l = 2; l < p.length; l++) {
            for (int i = 0; i + l < p.length; i++) {
                int j = i + l;
                for (int k = i + 1; k < j; k++) {
                    dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k][j] + p[i] * p[k] *
p[j]);
                }
            }
```

```
        }

        // Return the minimum number of multiplications required to multiply the matrices in the chain.
        return dp[0][p.length - 1];
    }

    public static void main(String[] args) {
        // Create an array to store the dimensions of the matrices.
        int[] p = {3, 4, 5, 6, 7};

        // Find the minimum number of multiplications required to multiply the matrices.
        int mcm = mcm(p);

        // Print the minimum number of multiplications.
        System.out.println(mcm);
    }
}
```

This code works by first creating a table to store the minimum number of multiplications required to multiply the matrices in each chain.

The table is initialized with all values set to Integer.MAX_VALUE.

For each chain, the table is updated by taking the minimum of the following:

- The value in the table at the previous row and the previous column.
- The value in the table at the previous row and the current column, plus the product of the dimensions of the matrices at the current row and column.

The minimum number of multiplications required to multiply the matrices is then returned by taking the value in the table at the last row and the last column.

COIN CHANGE PROBLEM

Here is the Java code for finding the minimum number of coins required to make a certain amount of change in Java:

```
import java.util.*;


public class CoinChange {


   public static int minCoins(int[] coins, int amount) {

      // Create a table to store the minimum number of coins

      // required to make each amount of change.

      int[] dp = new int[amount + 1];


      // Initialize the table.

      for (int i = 0; i <= amount; i++) {

         dp[i] = Integer.MAX_VALUE;

      }
```

```
        // Fill the table.
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int j = 0; j < coins.length; j++) {
                if (i >= coins[j]) {
                    dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
                }
            }
        }

        // Return the minimum number of coins required to make the change.
        return dp[amount];
    }

    public static void main(String[] args) {
        // Create an array to store the coins.
        int[] coins = {1, 2, 5};

        // Find the minimum number of coins required to make a change of 11.
        int minCoins = minCoins(coins, 11);
```

```
        // Print the minimum number of coins.
        System.out.println(minCoins);
    }
}
```

This code works by first creating a table to store the minimum number of coins required to make each amount of change.

The table is initialized with all values set to Integer.MAX_VALUE.

For each amount of change, the table is updated by taking the minimum of the following:

- The value in the table at the previous amount.
- The value in the table at the previous amount minus the coin, plus 1.

The minimum number of coins required to make the change is then returned by taking the value in the table at the last amount.

EDIT DISTANCE

Here is the Java code for computing the minimum number of operations (insertion, deletion, substitution) required to transform one string into another in Java:

```
import java.util.*;


public class LevenshteinDistance {


    public static int levenshteinDistance(String s1, String s2) {
```

```
    int n = s1.length();
    int m = s2.length();

    // Create a table to store the minimum distance between
    // prefixes of the strings.
    int[][] dp = new int[n + 1][m + 1];

    // Initialize the table.
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = i + j;
            } else if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = Math.min(dp[i - 1][j - 1] + 1, Math.min(dp[i - 1][j] +
1, dp[i][j - 1] + 1));
            }
        }
    }

    // Return the minimum distance between the strings.
    return dp[n][m];
```

```
    }

    public static void main(String[] args) {
        // Create two strings.
        String s1 = "kitten";
        String s2 = "sitting";

        // Find the minimum number of operations required to transform one string into another.
        int levenshteinDistance = levenshteinDistance(s1, s2);

        // Print the minimum number of operations.
        System.out.println(levenshteinDistance);
    }
}
```

This code works by first creating a table to store the minimum distance between prefixes of the strings.

The table is initialized with all values set to the sum of the lengths of the strings.

For each character in the strings, the table is updated by taking the minimum of the following:

- The value in the table at the previous character in both strings.
- The value in the table at the previous character in the first string and the current character in the second string, plus 1.

- The value in the table at the previous character in the second string and the current character in the first string, plus 1.

The minimum distance between the strings is then returned by taking the value in the table at the last character in both strings.

SUBSET SUM PROBLEM

Here is the Java code for determining if there is a subset of a given set that adds up to a specified target value in Java:

```
import java.util.*;


public class SubsetSum {


   public static boolean isSubsetSum(int[] set, int target) {
      int n = set.length;


      // Create a table to store the results of
      // all possible subset sums.
      boolean[][] dp = new boolean[n + 1][target + 1];


      // Initialize the table.
      for (int i = 0; i <= n; i++) {
         for (int j = 0; j <= target; j++) {
            if (i == 0) {
```

```
                dp[i][j] = false;
            } else if (j == 0) {
                dp[i][j] = true;
            } else if (set[i - 1] <= j) {
                dp[i][j] = dp[i - 1][j] || dp[i - 1][j - set[i - 1]];
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }

    // Return the result of the target value.
    return dp[n][target];
}

public static void main(String[] args) {
    // Create a set.
    int[] set = {1, 2, 3, 4, 5};

    // Find if there is a subset of the set that adds up to 10.
    boolean isSubsetSum = isSubsetSum(set, 10);
```

```
        // Print the result.
        System.out.println(isSubsetSum);
    }
}
```

This code works by first creating a table to store the results of all possible subset sums.

The table is initialized with all values set to false.

For each element in the set, the table is updated by taking the logical OR of the following:

- The value in the table at the previous element.
- The value in the table at the previous element and the current element, if the current element is less than or equal to the target value.

The result of the target value is then returned by taking the value in the table at the last element.

MAXIMUM SUBARRAY SUM

Here is the Java code for finding the contiguous subarray with the largest sum within a given array in Java:

```
import java.util.*;


public class LargestSumSubarray {


    public static int largestSumSubarray(int[] arr) {
        int n = arr.length;
```

```
    // Create a table to store the maximum sum of subarrays
    // ending at each position in the array.
    int[] dp = new int[n + 1];

    // Initialize the table.
    for (int i = 0; i <= n; i++) {
        dp[i] = 0;
    }

    // Fill the table.
    for (int i = 1; i <= n; i++) {
        for (int j = i; j <= n; j++) {
            dp[j] = Math.max(dp[j], dp[j - i] + arr[j - 1]);
        }
    }

    // Return the maximum sum of subarrays.
    return dp[n];
}

public static void main(String[] args) {
```

```
        // Create an array.
        int[] arr = {-2, -3, 4, -1, -2, 1, 5, -3};

        // Find the contiguous subarray with the largest sum.
        int largestSum = largestSumSubarray(arr);

        // Print the result.
        System.out.println(largestSum);
    }
}
```

This code works by first creating a table to store the maximum sum of subarrays ending at each position in the array.

The table is initialized with all values set to 0.

For each position in the array, the table is updated by taking the maximum of the following:

- The value in the table at the previous position.
- The value in the table at the previous position and the current element, if the current element is positive.

The maximum sum of subarrays is then returned by taking the value in the table at the last position.

TRAVELING SALESMAN PROBLEM

Here is the Java code for determining the shortest possible route that visits a set of cities and returns to the starting point in Java:

```
import java.util.*;
```

```
public class TravellingSalesmanProblem {


   public static List<Integer> shortestRoute(List<Integer> cities, List<Integer[]> distances) {
      int n = cities.size();


      // Create a table to store the minimum distance between
      // all pairs of cities.
      int[][] dp = new int[n][n];


      // Initialize the table.
      for (int i = 0; i < n; i++) {
         for (int j = 0; j < n; j++) {
            if (i == j) {
               dp[i][j] = 0;
            } else {
               dp[i][j] = Integer.MAX_VALUE;
            }
         }
      }


      // Fill the table.
```

```
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                dp[i][j] = Math.min(dp[i][j], dp[i][k] + dp[k][j] + distances.get(i)[k] + distances.get(k)[j]);
            }
        }
    }

    // Find the minimum distance between all pairs of cities.
    int minDistance = Integer.MAX_VALUE;
    List<Integer> minRoute = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j) {
                if (dp[i][j] < minDistance) {
                    minDistance = dp[i][j];
                    minRoute = new ArrayList<>();
                    minRoute.add(i);
                    minRoute.add(j);
                } else if (dp[i][j] == minDistance) {
                    minRoute.add(j);
                }
```

```
            }
        }
    }

    // Return the shortest route.
    return minRoute;
}

public static void main(String[] args) {
    // Create a list of cities.
    List<Integer> cities = Arrays.asList(0, 1, 2, 3, 4);

    // Create a list of distances between cities.
    List<Integer[]> distances = new ArrayList<>();
    distances.add(new Integer[]{0, 10, 20, 30, 40});
    distances.add(new Integer[]{10, 0, 5, 15, 25});
    distances.add(new Integer[]{20, 5, 0, 25, 35});
    distances.add(new Integer[]{30, 15, 25, 0, 15});
    distances.add(new Integer[]{40, 25, 35, 15, 0});

    // Find the shortest route.
    List<Integer> shortestRoute = shortestRoute(cities, distances);
```

```
        // Print the shortest route.
        for (int city : shortestRoute) {
            System.out.print(city + " ");
        }
    }
}
```

This code works by first creating a table to store the minimum distance between all pairs of cities.

The table is initialized with all values set to Integer.MAX_VALUE.

For each pair of cities, the table is updated by taking the minimum of the following:

- The value in the table at the previous pair of cities.
- The value in the table at the previous pair of cities and the distance between the two cities.

The minimum distance between all pairs of cities is then returned by taking the value in the table at the last pair of cities.

ASSEMBLY LINE SCHEDULING

Here is the Java code for optimizing the sequence of tasks in a production system to minimize the overall processing time in Java:

```
import java.util.*;


public class TaskScheduling {
```

```
public static List<Integer> optimizeTaskSequence(List<Integer> tasks, List<Integer[]> dependencies) {
    int n = tasks.size();

    // Create a table to store the minimum processing time for each task.
    int[] dp = new int[n];

    // Initialize the table.
    for (int i = 0; i < n; i++) {
        dp[i] = Integer.MAX_VALUE;
    }

    // Fill the table.
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (dependencies.get(i)[j] == 1) {
                dp[i] = Math.min(dp[i], dp[j] + tasks.get(i));
            }
        }
    }

    // Find the minimum processing time for all tasks.
```

```
    int minProcessingTime = Integer.MAX_VALUE;
    List<Integer> minTaskSequence = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        if (dp[i] < minProcessingTime) {
            minProcessingTime = dp[i];
            minTaskSequence = new ArrayList<>();
            minTaskSequence.add(i);
        } else if (dp[i] == minProcessingTime) {
            minTaskSequence.add(i);
        }
    }

    // Return the minimum task sequence.
    return minTaskSequence;
}

public static void main(String[] args) {
    // Create a list of tasks.
    List<Integer> tasks = Arrays.asList(1, 2, 3, 4, 5);

    // Create a list of dependencies.
    List<Integer[]> dependencies = new ArrayList<>();
```

```
        dependencies.add(new Integer[]{0, 1});
        dependencies.add(new Integer[]{1, 2});
        dependencies.add(new Integer[]{2, 3});
        dependencies.add(new Integer[]{3, 4});

        // Find the minimum task sequence.
        List<Integer> minTaskSequence = optimizeTaskSequence(tasks, dependencies);

        // Print the minimum task sequence.
        for (int task : minTaskSequence) {
            System.out.print(task + " ");
        }
    }
}
```

This code works by first creating a table to store the minimum processing time for each task.

The table is initialized with all values set to Integer.MAX_VALUE.

For each task, the table is updated by taking the minimum of the following:

- The value in the table at the previous task.
- The value in the table at the previous task plus the processing time of the current task.

The minimum processing time for all tasks is then returned by taking the value in the table at the last task.

OPTIMAL BINARY SEARCH TREES

Here is the Java code for constructing a binary search tree with minimum expected search time for a given set of keys in Java:

```
import java.util.*;


public class OptimalBinarySearchTree {


   public static Node constructOptimalBST(int[] keys, int[] frequencies)
{

      int n = keys.length;


      // Create a table to store the minimum expected search time for
each subtree.

      int[][] dp = new int[n + 1][n + 1];


      // Initialize the table.

      for (int i = 0; i <= n; i++) {

         for (int j = 0; j <= n; j++) {

            if (i == 0 || j == 0) {

               dp[i][j] = 0;

            } else {
```

```
                dp[i][j] = Integer.MAX_VALUE;
            }
        }
    }

    // Fill the table.
    for (int i = 1; i <= n; i++) {
        for (int j = i; j <= n; j++) {
            for (int k = i; k <= j; k++) {
                dp[i][j] = Math.min(dp[i][j], dp[i][k - 1] + dp[k + 1][j] +
frequencies[k - 1]);
            }
        }
    }

    // Create a node for the root of the tree.
    Node root = new Node(keys[0], frequencies[0]);

    // Recursively construct the left and right subtrees.
    root.left = constructOptimalBST(keys, frequencies, 0, root.key - 1);
    root.right = constructOptimalBST(keys, frequencies, root.key + 1, n
- 1);
```

```
    return root;
}

public static void main(String[] args) {
    // Create a set of keys.
    int[] keys = {1, 2, 3, 4, 5};

    // Create a set of frequencies.
    int[] frequencies = {1, 2, 3, 4, 5};

    // Construct the optimal binary search tree.
    Node root = constructOptimalBST(keys, frequencies);

    // Print the tree.
    printTree(root);
}

private static void printTree(Node root) {
    if (root == null) {
        return;
    }
```

```
        System.out.print(root.key + " ");
        printTree(root.left);
        printTree(root.right);
    }
}

class Node {

    int key;
    int frequency;
    Node left;
    Node right;

    public Node(int key, int frequency) {
        this.key = key;
        this.frequency = frequency;
        this.left = null;
        this.right = null;
    }
}
```

This code works by first creating a table to store the minimum expected search time for each subtree.

The table is initialized with all values set to Integer.MAX_VALUE.

For each subtree, the table is updated by taking the minimum of the following:

- The value in the table at the previous subtree.
- The value in the table at the previous subtree plus the sum of the frequencies of all nodes in the current subtree.

The optimal binary search tree is then constructed recursively by calling the constructOptimalBST() function on each subtree.

The tree is then printed by calling the printTree() function.