Module 3 Dynamic Programming

3.1. General method

Dynamic programming solves complex problems by separating them into simple subproblems. Each subproblem only once solved, and its solution is saved for future reference. It is beneficial when the major problem can be divided into overlapping sub-problems, as it avoids redundant calculations by caching the results of already solved subproblems.

Principle of optimality

The principle of optimality is a fundamental concept in dynamic programming. It is stated that an optimal policy has the property that, regardless of the initial state and decisions, the subsequent decisions must form an optimal policy for the resultant state after the initial decision. In simpler terms, if a problem can be solved by breaking it down into smaller subproblems, then the overall problem can be obtained by optimally combining the solutions.

The principle of optimality is often associated with Bellman's Principle of Optimality, named after mathematician Richard Bellman, who contributed significantly to the development of dynamic programming. Bellman's principle applies to problems that exhibit the property of optimal substructure.

This principle is widely used in optimization problems, such as shortest-path and knapsack problems. By exploiting the principle of optimality, dynamic programming algorithms can efficiently solve complex problems by breaking them down into simpler subproblems and reusing solutions to build the optimal solution.

3.2. Matrix-chain multiplication

Matrix-chain multiplication is a classic computer science problem that efficiently multiples a series of matrices together. Dynamic programming is a common technique for solving this problem efficiently.

We want to find an efficient way to multiply given sequence of matrices.

For instance, A, B, and C, are given matrices if we want to compute the product ABC, we have multiple ways to parenthesize the multiplication:

(AB)C

A(BC)

Each parenthesization leads to a different number of scalar multiplications, and we want to find the parenthesization that requires the fewest scalar multiplications.

Dynamic programming solves complex problems by separating them down into smaller subproblems. It works well for problems with overlapping subproblems and optimal substructure.

For the matrix-chain multiplication problem, dynamic programming can be applied to find the optimal parenthesization.

Steps for Dynamic Programming Solution

1. **Define Subproblems**: Identify the subproblems and define a recursive solution.
2. **Find Recurrence Relation**: Determine how to relate the larger subproblems solution to the solutions of smaller sub-problems.
3. **Memoization or Bottom-up Tabulation**: Implement the solution using either memoization (top-down with caching) or tabulation (bottom-up with iteration).

Matrix-Chain Order and Parenthesization:

To find the optimal parenthesization, dynamic programming maintains a table where each entry stores the minimum number of scalar multiplications needed to compute the product of a subchain of matrices. Additionally, another table keeps track of the split points that lead to the optimal solution.

Complexity Analysis:

- The time complexity of using a dynamic programming approach to solve the matrix-chain multiplication problem is $O(n^3)$, where n is the number of matrices.
- The space complexity is $O(n^2)$ to store the tables.

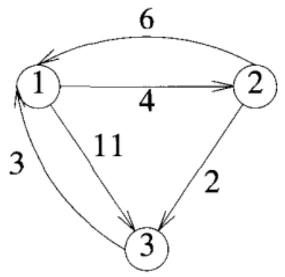
Matrix-chain multiplication is a problem of finding the optimal way to multiply a sequence of matrices, minimizing the number of scalar multiplications. Dynamic programming efficiently solves this problem by breaking it into smaller subproblems and storing the solutions to avoid redundant calculations. The dynamic programming approach ensures an optimal solution through systematic exploration of all possible combinations of parenthesization.

3.3. All pairs shortest path (Floyd-Warshall algorithm)

The All-Pairs Shortest Path problem involves finding the shortest path between every vertex pair in a weighted graph.

- The Floyd-Warshall algorithm is a dynamic programming-based approach to finding the shortest paths between all vertex pairs in a weighted graph.
- It can be applied to directed and undirected graphs with positive or negative edge weights without negative cycles. The algorithm iterates over all pairs of vertices (i, j) and considers each vertex k a potential intermediate vertex in the path from vertex i to vertex j.
- It updates the shortest path between vertices i and j by considering whether going through vertex k would result in a shorter path.
- Floyd-Warshall algorithm's time complexity is $O(V^3)$, with V being the number of vertices in the graph.

```
Algorithm AllPaths($cost, A, n$)
// $cost[1:n, 1:n]$ is the cost adjacency matrix of a graph with
// $n$ vertices; $A[i, j]$ is the cost of a shortest path from vertex
// $i$ to vertex $j$. $cost[i, i] = 0.0$, for $1 \leq i \leq n$.
{
    for $i := 1$ to $n$ do
        for $j := 1$ to $n$ do
            $A[i, j] := cost[i, j]$; // Copy $cost$ into $A$.
    for $k := 1$ to $n$ do
        for $i := 1$ to $n$ do
            for $j := 1$ to $n$ do
                $A[i, j] := \min(A[i, j], A[i, k] + A[k, j])$;
}
```



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

The Floyd-Warshall algorithm's time complexity is $O(V^3)$, where V is the number of vertices in the graph.

Here is a breakdown of how this time complexity is derived:

1. **Initialization**:
    - Initializing the distance matrix with initial values from the graph takes $O(V^2)$ time since it involves visiting every cell in the $V \times V$ matrix.
2. **Main Loop**:
    - The main loop iterates V times for each of the V vertices. Within each iteration, two nested loops are iterating over all pairs of vertices. Hence, the nested loops together contribute to $O(V^2)$ operations. Since the main loop iterates V times, the total complexity of the nested loops becomes $O(V^3)$.

3.4. Optimal binary search trees

The Optimal Binary Search Tree (BST) is an extension of the basic Binary Search Tree data structure to minimize the average search time for a given set of keys.

In a standard Binary Search Tree, the keys are organized in a binary tree structure such that for any node:

1. The left subtree contains keys smaller than the node's key.
2. The right subtree contains keys greater than the node's key.
3. Both subtrees (left and right) are also binary search trees.

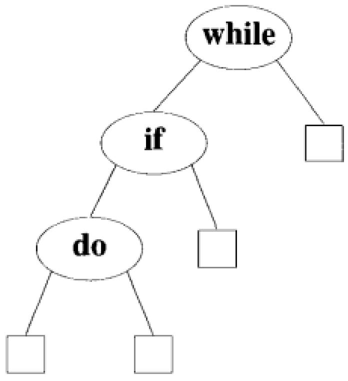
However, in a regular BST, if keys are not inserted in a sorted order, the tree may become unbalanced, leading to worst-case search times of $O(n)$, where n is the number of keys in the tree.

An Optimal Binary Search Tree aims to minimize the average search time by ensuring that frequently searched keys are closer to the root, thus reducing the depth of the tree for those keys.

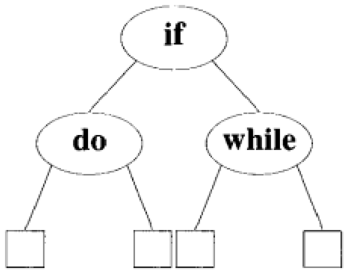
Features of Optimal Binary Search Tree:

1. Key Frequency: The frequency of accessing each key is considered. Keys accessed more frequently are placed closer to the root so that the average search time is minimized.
2. Balanced Structure: Unlike standard BSTs, Optimal BSTs aim to maintain a balanced structure so that the average search time is minimized. This is achieved by strategically placing keys to ensure that the tree is as balanced as possible.

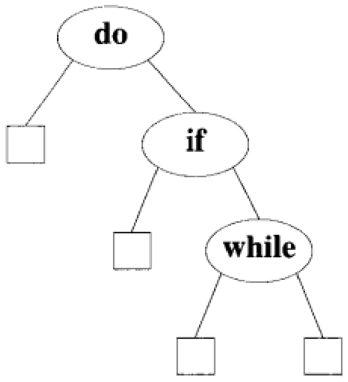
Ex: Possible binary search trees for the identifier ser (a1,a2,a3)= (do,if,while) are



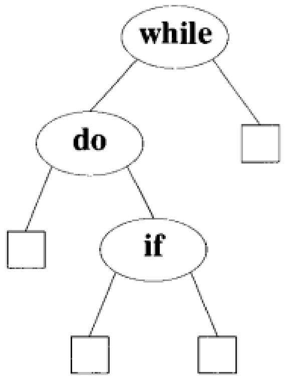
(a)



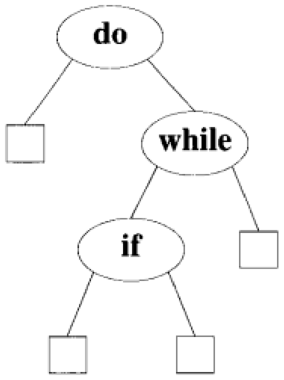
(b)



(c)



(d)



(e)

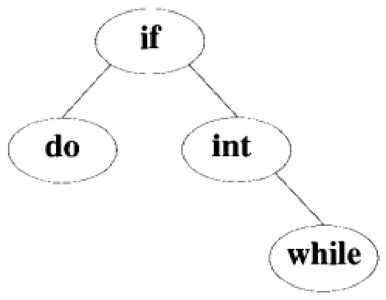
$$c(i,j) = \min_{i<k\leq j}\{c(i,k-1)+c(k,j)\} + w(i,j)$$

Example:

Let $n = 4$ and $(a_1, a_2, a_3, a_4) =$ (**do, if, int, while**). Let $p(1:4) = (3,3,1,1)$ and $q(0:4) = (2,3,1,1,1)$. The p's and q's have been multiplied by 16 for convenience. Initially, we have $w(i,i) = q(i), c(i,i) = 0$ and $r(i,i) = 0$, $0 \leq i \leq 4$. Using Equation and the observation $w(i,j) = p(j) + q(j) + w(i,j-1)$, we get

$$\begin{aligned}
w(0,1) &= p(1)+q(1)+w(0,0)=8\\
c(0,1) &= w(0,1)+\min\{c(0,0)+c(1,1)\} \quad = \quad 8\\
r(0,1) &= 1\\
w(1,2) &= p(2)+q(2)+w(1,1) \quad = \quad 7\\
c(1,2) &= w(1,2)+\min\ \{c(1,1)+c(2,2)\} \quad = \quad 7\\
r(0,2) &= 2\\
w(2,3) &= p(3)+q(3)+w(2,2) \quad = \quad 3\\
c(2,3) &= w(2,3)+\min\ \{c(2,2)+c(3,3)\} \quad = \quad 3\\
r(2,3) &= 3\\
w(3,4) &= p(4)+q(4)+w(3,3) \quad = \quad 3\\
c(3,4) &= w(3,4)+\min\ \{c(3,3)+c(4,4)\} \quad = \quad 3\\
r(3,4) &= 4
\end{aligned}$$

	0	1	2	3	4
0	$w_{00}=2$ $c_{00}=0$ $r_{00}=0$	$w_{11}=3$ $c_{11}=0$ $r_{11}=0$	$w_{22}=1$ $c_{22}=0$ $r_{22}=0$	$w_{33}=1$ $c_{33}=0$ $r_{33}=0$	$w_{44}=1$ $c_{44}=0$ $r_{44}=0$
1	$w_{01}=8$ $c_{01}=8$ $r_{01}=1$	$w_{12}=7$ $c_{12}=7$ $r_{12}=2$	$w_{23}=3$ $c_{23}=3$ $r_{23}=3$	$w_{34}=3$ $c_{34}=3$ $r_{34}=4$	
2	$w_{02}=12$ $c_{02}=19$ $r_{02}=1$	$w_{13}=9$ $c_{13}=12$ $r_{13}=2$	$w_{24}=5$ $c_{24}=8$ $r_{24}=3$		
3	$w_{03}=14$ $c_{03}=25$ $r_{03}=2$	$w_{14}=11$ $c_{14}=19$ $r_{14}=2$			
4	$w_{04}=16$ $c_{04}=32$ $r_{04}=2$				



3.5. 0/1 Knapsack Problem

The 0/1 knapsack problem is a classic problem in computer science and optimization. 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. In the 0/1 knapsack problem, each item can either be included in the knapsack or not (hence the name 0/1).

Here is how you can solve the 0/1 knapsack problem using dynamic programming:

Problem Statement

You are given:

- n items, each with a weight w[i] and a value v[i].
- A knapsack with a maximum weight capacity W.

Goal

Maximize the total value of the items included in the knapsack without exceeding the weight capacity W.

Methodology

1. **Define the DP table:** Let dp[i][w] be the maximum value that can be achieved using the first i items and with a maximum weight limit w.
2. **Base Case:**

   dp[0][w] = 0 for all w (0 items can achieve 0 value).

   dp[i][0] = 0 for all i (maximum weight limit 0 can achieve 0 value).
3. **Recursive Relation:** For each item i ($1 \leq i \leq n$) and each weight w ($1 \leq w \leq W$):

   If the weight of the item i is greater than w, then it cannot be included: $dp[i][w] = dp[i-1][w]$

   If the weight of the item i is less than or equal to w, then you have two choices:

- Exclude the item i: $dp[i][w] = dp[i-1][w]$
- Include the item i: $dp[i][w] = dp[i-1][w-w[i]] + v[i]$

$$dp[i][w] = max(dp[i-1][w], dp[i-1][w-w[i]] + v[i])$$

4. **Final Solution:** The maximum value that can be achieved with the given weight limit W is dp[n][W].

5. **Java Implementation**

```
public class Knapsack {
  public static int knapsack(int[] values, int[] weights, int W) {
    int n = values.length;
    int[][] dp = new int[n + 1][W + 1];

    for (int i = 1; i <= n; i++) {
      for (int w = 0; w <= W; w++) {
        if (weights[i - 1] <= w) {
          dp[i][w] = Math.max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i -
1]);
        } else {
          dp[i][w] = dp[i - 1][w];
        }
      }
    }|
    return dp[n][W];
  }
  public static void main(String[] args) {
    int[] values = {60, 100, 120};
    int[] weights = {10, 20, 30};
    int W = 50;
    System.out.println("Maximum value in Knapsack = " + knapsack(values, weights,
W));
  }
}
```

3.6. Travelling Salesperson Problem

The dynamic programming approach for TSP uses bit masking and memoization. Here's the idea:

1. **State Representation:**

   Use a bitmask to represent the set of cities visited so far.

   Use an integer to represent the current city.

2. **State Transition:**

   From the current city, transition to an unvisited city and update the bitmask.

3. **Base Case:**

   When all cities have been visited, return to the starting city.

4. **Memoization:**

   To avoid redundant calculations, store the results of subproblems.

Algorithm Steps

1. **Initialize memoization table:**

   Use a 2D array dp[mask][i] where mask represents the set of visited cities and i represents the current city. dp[mask][i] stores the minimum cost to visit all cities in mask ending at city i.

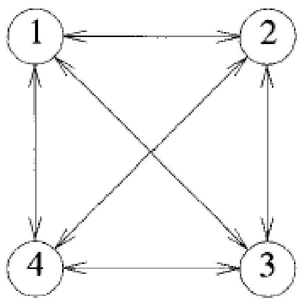
2. **Recursive Transition:**

   For each city i and each set of visited cities mask, calculate the minimum cost to visit an unvisited city j.

3. **Combine Results:**

   Start from the origin city, visit all the remaining cities, and return back to the origin city.

Here is an example to illustrate the process:



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

$$\begin{aligned} g(1,\{2,3,4\}) &= \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\} \\ &= \min\{35, 40, 43\} \\ &= 35 \end{aligned}$$

3.7.Flow Shop Scheduling

The dynamic programming approach to solving the flow shop scheduling problem can be complex, but it involves breaking down the problem into subproblems and solving each subproblem optimally to build the solution for the original problem.

Problem Statement

Given:

- n jobs
- m machines
- Processing time p[i][j] for job i on machine j

Goal

Minimize the makespan (the total completion time).

Approach

1. **State Representation:**

   Let dp[i][j] be the minimum time required to complete the first i jobs on the first j machines.

2. **State Transition:**

   To compute dp[i][j], consider the time to complete the i-th job on the j-th machine as the maximum of:

   - The time when the i-th job completes on the previous machine j-1 (i.e., dp[i][j-1] plus the processing time of the i-th job on machine j).
   - The time when the previous job i-1 completes on the j-th machine (i.e., dp[i-1][j] plus the processing time of the i-th job on machine j).

3. **Base Case:**

   dp[0][j] = 0 for all j (no jobs to process).

   dp[i][0] = sum(p[k][0] for k in range(i+1)) for all i (processing the first job on all machines).

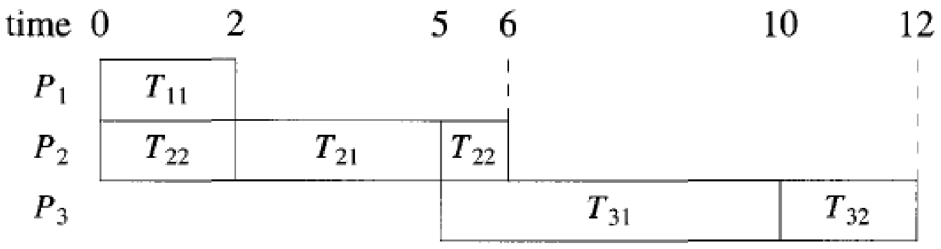
4. **Final Solution:**

   The value of dp[n][m] will be the makespan for all n jobs on m machines.

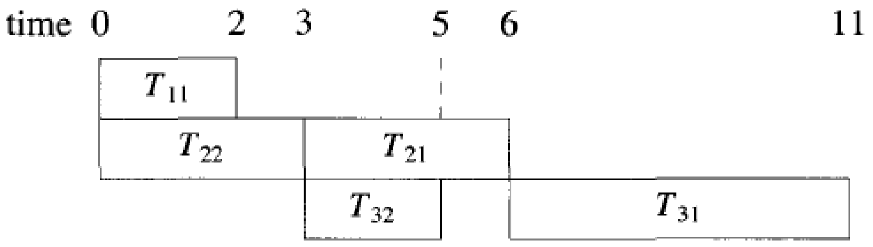
Example: On three processors two jobs have to be scheduled. The matrix J represents the tasks and times.

$$J = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

Below figure illustrates this example.



(a)



(b)