Design and Analysis of Algorithm [R1UC407B]

Module-V: Dynamic Programming

Dr. A K Yadav

GALGOTIAS UNIVERSITY

School of Computer Science and Engineering
Plat No 2, Sector 17A, Yamuna Expressway
Greater Noida, Uttar Pradesh - 203201

February 14, 2025

Contents

Dynamic programming 3
- Coin Change Problem 9
- Longest Common Subsequence 14
- Traveling Salesman Problem 26
- Multi stage graph 27
- Floyd Warshall algorithm 28
- 0-1 Knapsack Problem 36
- Matrix Chain Multiplication(MCM) 43
- Optimal binary search trees 53
- Binomial Coefficient 73

Dynamic programming

- Solves problems by combining the solutions to sub-problems
- Sub-problems are overlapping
- Doesn't solve overlapping sub-problems again and again
- Behave like Divide and Conquer if sub-problems are not overlapping
- Used in optimization problems
- Can be used either top-down with memoization or bottom-up method

Four Steps of Dynamic programming

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution
4. Construct an optimal solution from computed information

Elements of dynamic programming

Two key ingredients that an optimization problem must have in order to apply dynamic programming:

1. Optimal substructure
2. Overlapping sub-problems

Memoization

- Memoization is mainly storing the value in the form of Memo or in some tabular method
- Whenever we need some calculation of the sub-problem then we first check the memo
- If solution of the sub-problems is already available in memo then we use that solution and not solve the sub-problem again
- If the solution of the sub-problem is not in memo then we solve the sub-problem and note the result in the form of the memo for next call

Difference between Dynamic Programming and Memoization

- Memoization is a term describing an optimization technique where we cache previously computed results, and return the cached result when the same computation is needed again.
- Dynamic programming is a technique for solving problems of recursive nature, iteratively and is applicable when the computations of the subproblems overlap.
- Dynamic programming is typically implemented using tabulation, but can also be implemented using memoization. So as we can see, neither one is a "subset" of the other.

- When we solve a dynamic programming problem using tabulation we solve the problem bottom-up i.e. by solving all related sub-problems first, typically by filling up an n-dimensional table. Based on the results in the table, the solution to the top / original problem is then computed.
- If we use memoization to solve the problem we do it by maintaining a map of already solved sub problems. We do it top-down in the sense that we solve the top problem first (which typically recurses down to solve the sub-problems).

Coin Change Problem

Given an integer array of *coins* of size N representing different types of denominations and an integer *sum*, the task is to count all combinations of coins to make a given value sum. Assume that you have an infinite supply of each type of coin.

For each coin, there are 2 options:

1. **Include the current coin**: Subtract the current coin's denomination from the target sum and call the count function recursively with the updated sum and the same set of coins i.e., $count(coins, n, sum–coins[n-1])$
2. **Exclude the current coin**: Call the count function recursively with the same sum and the remaining coins. i.e., $count(coins, n-1, sum)$.

The final result will be the sum of both cases.

Base case:

- If the target sum (sum) is 0, there is only one way to make the sum, which is by not selecting any coin. So, $count(0, coins, n) = 1$.
- If the target sum ($sum < 0$) is negative or no coins are left to consider ($n = 0$), then there are no ways to make the sum, so $count(sum, coins, 0) = 0$.

We can use the following steps to implement the dynamic programming(tabulation) approach for Coin Change.

- Create a 2D dp array with rows and columns equal to the number of coin denominations and target sum.
- dp[0][0] will be set to 1 which represents the base case where the target sum is 0, and there is only one way to make the change by not selecting any coin.

- Iterate through the rows of the dp array (i from 1 to n), representing the current coin being considered.
- The inner loop iterates over the target sums (j from 0 to sum).
  - Add the number of ways to make change without using the current coin, i.e., dp[i][j] += dp[i-1][j].
  - Add the number of ways to make change using the current coin, i.e., dp[i][j] += dp[i][j-coins[i-1]].
- dp[n][sum] will contain the total number of ways to make change for the given target sum using the available coin denominations.


```
// Returns total distinct ways to make sum using n coins
//of different denominations
int count(vector<int>& coins, int n, int sum)
{
 // 2d dp array where n is the number of coin
    // denominations and sum is the target sum
    vector<vector<int>> dp(n+1, vector<int>(sum+1,0));
    // Represents the base case where the target sum is (
    // and there is only one way to make change:
    // by not selecting any coin
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++)
    {
     for (int j = 0; j <= sum; j++)
     {
      // Add the number of ways to make change without
      // using the current coin,
```


```
            dp[i][j] += dp[i - 1][j];
            if ((j - coins[i - 1]) >= 0)
            {
             // Add the number of ways to make change
             // using the current coin
             dp[i][j] += dp[i][j - coins[i - 1]];
            }
          }
        }
        return dp[n][sum];
}
// Driver Code
int main()
{
        vector<int> coins{ 1, 2, 3 };
         int n = 3;
        int sum = 5;
```


```
    cout << count(coins, n, sum);
    return 0;
}
```

Longest Common Subsequence

- Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ of length m and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ of length n
- $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a common subsequence of X and Y if Z is a subsequence of both X and Y
- Z is a Longest common subsequence of X and Y if X and Y have no common subsequence of length greater then Z
- ith prefix of X is $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ for $i = 0, 1, \ldots, m$

Step 1: Characterizing a longest common subsequence

- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.
- **Optimal substructure of an LCS**
- If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
- If $x_m \neq y_n$ then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y
- If $x_m \neq y_n$ then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Step 2: A recursive solution

- In step 1 there are two cases:
- Case 1: if $x_m = y_n$ then find the LCS of X_{m-1} and Y_{n-1}
- Appending x_m to LCS of X_{m-1} and Y_{n-1} gives LCS of X and Y
- Case 2: if $x_m \neq y_n$ then solve two sub-problems: find LCS of X_{m-1} and Y and find LCS of X and Y_{n-1}
- Longer of these two LCS will be the LCS of X and Y
- Let $c[i, j]$ is the length of an LCS of the sequences X_i and Y_j
- If either of two sequences is empty i.e. $i = 0$ or $j = 0$ then LCS will be of length zero that is $c[i, j] = 0$

- Recursive solution will be

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Step 3: Computing the length of an LCS

- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.
- Let Table $c[i, j]$ to store the length of an LCS of the sequences X_i and Y_j
- $c[0 \ldots m, 0 \ldots n]$
- Let Table $b[i, j]$ to store the direction of the flow of length of LCS to construct an optimal solution
- $b[1 \ldots m, 1 \ldots n]$

Bottom-up Approach

LCS-LENGTH(X, Y)

1. $m = length(X)$
2. $n = length(Y)$
3. Let $c[0 \ldots m, 0 \ldots n]$ and $b[1 \ldots m, 1 \ldots n]$ two new tables
4. for $i = 1$ to m
5. $\quad c[i, 0] = 0 // Y$ is empty
6. for $j = 1$ to n
7. $\quad c[0, j] = 0 // X$ is empty
8. for $i = 1$ to m
9. $\quad$ for $j = 1$ to n
10. $\quad\quad$ if $x_i = y_j$
11. $\quad\quad\quad c[i, j] = c[i - 1, j - 1] + 1$

12. $b[i,j] = \nwarrow$
13. else if $c[i-1,j] \geq c[i,j-1]$
14. $c[i,j] = c[i-1,j]$
15. $b[i,j] = \uparrow$
16. else
17. $c[i,j] = c[i,j-1]$
18. $b[i,j] = \rightarrow$
19. return c, b

Length of the LCS is **c[m,n]**
Complexity of the algorithm is $\Theta(mn)$

Top-down Approach:

MEMOIZED-LCS(X, Y, m, n)

1. Let $c[0 \ldots m, 0 \ldots n]$
2. for $i = 0$ to m
3. for $j = 0$ to n
4. $c[i, j] = 0$
5. return LCS(X, Y, c, b, m, n)

LCS(X, Y, c, b, i, j)

1. if $c[i, j] > 0$
2. return $c[i, j]$
3. if $x_i = y_j$
4. $c[i, j] = LCS(X, Y, c, b, i - 1, j - 1] + 1$
5. $b[i, j] = \nwarrow$

6. else if $LCS(X, Y, c, b, i-1, j) \geq LCS(X, Y, c, b, i, j-1)$
7. $\quad c[i,j] = LCS(X, Y, c, b, i-1, j)$
8. $\quad b[i,j] = \uparrow$
9. else
10. $\quad c[i,j] = LCS(X, Y, c, b, i, j-1)$
11. $\quad b[i,j] = \rightarrow$
12. return $c[i,j]$

Step 4: Constructing an LCS

- Step 3 calculates $c[i,j]$ and $b[i,j]$
- $c[i,j]$ gives the length of LCS of X_i and Y_j
- $b[i,j]$ tells from where we calculated $c[i,j]$
- $\nwarrow$ shows that $x_i = y_j$ and part of LCS
- $\uparrow$ shows that $c[i-1,j] \geq c[i,j-1]$
- $\rightarrow$ shows that $c[i,j-1] > c[i-1,j]$
- call PRINT-LCS(b, X, m, n)

PRINT-LCS(b, X, i, j)

1. if $i = 0$ or $j = 0$
2. return
3. if $b[i,j] = \nwarrow$
4. PRINT-LCS$(b, X, i-1, j-1)$

5. print x_i
6. else if $b[i,j] = \uparrow$
7. PRINT-LCS$(b, X, i-1, j)$
8. else
9. PRINT-LCS$(b, X, i, j-1)$

Complexity of the algorithm is $\Theta(m+n)$

Traveling Salesman Problem

- ▶

Multi stage graph

Floyd Warshall algorithm

- The algorithm find the shortest route between all nodes/vertices of the non-negative directed weighted graph G.
- Suppose V is set of n nodes/vertices in graph G i.e. $\{1, 2, \ldots, n\} \in V$
- E is the set of edges of the graph G .i.e $(i, j) \in E$ if there is a edge from i to j in the graph
- So we can say $G = (V, E)$
- w_{ij} is the weight of the edge from vertex i to j.
- if p is a path from vertex v_i to v_j through vertices $v_1, v_2, \ldots, v_l$ then the nodes other than source and destination is known as intermediate nodes.
- Sum of weight of the edges of the path p is the distance d_{ij} between vertex v_i to v_j
- if this distance is the shortest among all possible paths from v_i to v_j then it is called the shortest path $\delta(i, j)$

Step 1: The structure of a shortest path

- Let graph G has set of vertices $V = \{1, 2, \ldots, n\}$
- Let any pair of vertices $i, j \in V$, all paths can be drawn from the set of intermediate vertices $\{1, 2, \ldots, k\}$ for some k
- Let p is the shortest/minimum-weight simple path from i to j.
- There are two cases for any vertex k : either k is the part of the minimum-weight path or not the part of the path.
- If k is not the part of the path then it can be removed from the the set of vertices without effecting the path. So the path will be from $\{1, 2, \ldots, k-1\}$ or we can say a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$

- But if it is part of the path then we can divide the whole path p into two paths: p_1 - from i to k and p_2 - from k to j
- Intermediate nodes of p_1 and p_2 will be from $\{1, \ldots, k-1\}$ because now k is a source in one path and destination in other but not the intermediate node.

$$\delta(i,j) = \begin{cases} \delta(i,j) & \text{if } k \text{ is not part of } p \\ \delta(i,k) + \delta(k,j) & \text{if } k \text{ is part of } p \end{cases}$$

with intermediate nodes $\{1, 2, \ldots, k-1\}$

Step 2: A recursive solution

- d_{ij}^k is the shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$
- if $k = 0$ then no intermediate node so $d_{ij}^0 = w_{ij}$ if there is a direct edge from i to j else $d_{ij}^0 = \infty$ if there is no direct edge from i to j and $d_{ij}^0 = 0$ if $i = j$.
- $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$
- π_{ij}^k is the predecessor of vertex j for the shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$
- if $k = 0$ then no intermediate node so $\pi_{ij}^0 = i$ if there is a direct edge from i to j else $\pi_{ij}^0 = NIL$ if there is no direct edge from i to j and $\pi_{ij}^0 = NIL$ if $i = j$.

- $\pi_{ij}^{k} = \pi_{ij}^{k-1}$ if $d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1}$ or
  $\pi_{ij}^{k} = \pi_{kj}^{k-1}$ if $d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1}$

$$d_{ij}^{k} = \begin{cases} 0 & \text{if } i = j, k = 0 \\ \infty & \text{if } (i,j) \notin E, k = 0 \\ w_{ij} & \text{if } (i,j) \in E, k = 0 \\ \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\} & \end{cases}$$

$$\pi_{ij}^{k} = \begin{cases} NIL & \text{if } i = j, k = 0 \\ NIL & \text{if } w_{ij} = \infty, k = 0 \\ i & \text{if } w_{ij} < \infty, k = 0 \\ \pi_{ij}^{k-1} & \text{if } d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1} \\ \pi_{kj}^{k-1} & \text{if } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1} \end{cases}$$

Step 3: Computation of Shortest path

Let W is the weight matrix and A is the adjacency matrix of the graph G of n vertices.

Bottom-up Approach-I

FLOYD-WARSHALL(W,A,n)

1. $D^0 = W$
2. $\Pi^0 = A$
3. for $k = 1$ to n
4. Let $D^k = (d_{ij}^k)$ a $n \times n$ matrix
5. Let $\Pi^k = (\pi_{ij}^k)$ a $n \times n$ matrix
6. for $i = 1$ to n
7. for $j = 1$ to n
8. if $d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1}$

9. $d_{ij}^{k} = d_{ij}^{k-1}$
10. $\pi_{ij}^{k} = \pi_{ij}^{k-1}$
11. else
12. $d_{ij}^{k} = d_{ik}^{k-1} + d_{kj}^{k-1}$
13. $\pi_{ij}^{k} = \pi_{kj}^{k-1}$
14. return D^{n}, Π^{n}

Bottom-up Approach-II

FLOYD-WARSHALL(W,A,n)

1. $D = W$
2. $\Pi = A$
3. for $k = 1$ to n
4. for $i = 1$ to n
5. for $j = 1$ to n
6. if $d_{ij} \leq d_{ik} + d_{kj}$
7. $d_{ij} = d_{ij}$
8. $\pi_{ij} = \pi_{ij}$
9. else
10. $d_{ij} = d_{ik} + d_{kj}$
11. $\pi_{ij} = \pi_{kj}$
12. return D, Π

0-1 Knapsack Problem

- Set of n items from 1 to n are given
- w_i is the weight of the i^{th} item
- v_i is the value of the i^{th} item
- We can pick any item as a whole, fraction of weight is not allowed
- Total weight we pick can not be more than W but value of the sum of picked items must be maximum

- So 0-1 Knapsack Problem is an optimization problem stated as:

  Maximize $\sum_{i=1}^{n} v_i x_i$

  under the constraints $\sum_{i=1}^{n} x_i w_i \leq W$, and $x_i \in \{0, 1\}$

- In bounded Knapsack Problem we can pick upto c_i copies of x_i item:

  Maximize $\sum_{i=1}^{n} v_i x_i$

  under the constraints $\sum_{i=1}^{n} x_i w_i \leq W$, and $x_i \in \{0 \ldots c_i\}$

- In unbounded Knapsack Problem we can pick any number of x_i item including none:

Maximize $\sum_{i=1}^{n} v_i x_i$

under the constraints $\sum_{i=1}^{n} x_i w_i \leq W$, and $x_i \geq 0$

Compute the maximum Bag Value

- Total number of items is n
- $v[i]$ is the value of the i^{th} item whose weight is $w[i]$.
- Let Table $m[0 \ldots n, 0 \ldots W]$ is used to store the value of the bag for n items and W maximum capacity.
- $m[i, j]$ is the value of the bag from i item and j capacity.

Bottom-up Approach:

OPTIMAL-0-1KP(w,v,n,W)

1. Let $m[0 \ldots n, 0 \ldots W]$
2. for $i = 0$ to n
3. for $j = 0$ to W
4. if $i = 0$ or $j = 0$ //no item or nil bag capacity
5. $m[i, j] = 0$
6. else if $w[i] \leq j$// i^{th} item can be accommodated in bag
7. if $(m[i-1, j-w[i]] + v[i]) > m[i-1, j]$
8. $m[i, j] = m[i-1, j-w[i]] + v[i]$//pick the item
9. else
10. $m[i, j] = m[i-1, j]$//don't pick the item
11. return m

Top-down Approach:

Mamoized-0-1KP(w,v,n,W)

1. Let $m[0 \ldots n, 0 \ldots W]$
2. for $i = 0$ to n
3. for $j = 0$ to W
4. $m[i, j] = 0$
5. return $knapsack(m, w, v, n, W)$

$knapsack(m, w, v, i, j)$

1. if $m[i, j] > 0$
2. return $m[i, j]$
3. if $i = 0$ or $j = 0$ //no item or nil bag capacity
4. return 0
5. if $w[i] < j$

6. return $max(knapsack(m, w, v, i-1, j),$
   $knapsack(m, w, v, i-1, j-w[i]) + v[i])$
7. else
8. return $knapsack(m, w, v, i-1, j)$
9. return m

Complexity of the Knapsack is $O(nW)$

Matrix Chain Multiplication(MCM)

- Matrix chain multiplication is not the multiplication of the matrices but it is the way to find the order of matrix multiplication with minimum number of scalar multiplication.
- It is mainly fully parenthesization of the matrices
- ABC can be multiplied by two way : $A(BC)$ or $(AB)C$.
- Which one cost the minimum number of scalar multiplication can be found using MCM
- Find the order of multiplication of the matrices $A_1 A_2 \ldots A_n$
- We need an array p of size $n+1$ to store the dimensions of n compatible matrices

Step 1: The structure of an optimal parenthesization

- Let $A_{i\ldots j} = A_i \ldots A_j$ is the product of matrices from $A_i, A_{i+1}, \ldots A_j$ for $i \leq j$
- if $i = j$ then there is only one matrix and number of scalar multiplication is zero.
- if $i < j$ then we can put parenthesis anywhere after A_i but before A_j
- Suppose we use parenthesis after matrix A_k i.e. $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$ where $i \leq k < j$ for optimal solutions
- Now the total cost of the $A_{i\ldots j}$ will be cost of $A_{i\ldots k}$ plus cost of $A_{k+1\ldots j}$ plus the cost of multiplying them together.
- We supposed k is at optimal position so no other value of k is less costly.

- So $A_{i\ldots k}$ and $A_{k+1\ldots j}$ is also optimal.
- We can find out the optimal solution of the the problem from optimal solution of the the sub-problem
- We have to take the correct value of $k, i \leq k < j$ such that the sub-problems having the optimal solutions.
- We have to examine all possible value of k for the optimal solution.

Step 2: A recursive solution

We have to define cost of an optimal solution recursively in terms of the optimal solutions to sub-problems.

- Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i \ldots j}$ where $1 \leq i \leq j \leq n$
- The lowest cost to multiply $A_{1 \ldots n}$ will be $m[1, n]$.
- If $i = j$ that is there is only one matrix then no need to multiply. The $m[i, i] = 0$ for all $1 \leq i \leq n$.
- If $i < j$ then we split the matrix at position k for optimal solution.
- The total cost will be $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} \times p_k \times p_j$
- For all possible values of $k = i, i + 1, \ldots, j - 1$ we have to find out $m[i, j]$

- We pick that value of k for which $m[i,j]$ is minimum.
- $\therefore$ Recursive definition for the minimum cost of parenthesizing the product $A_{i\ldots j} = A_i A_{i+1} \ldots A_j$ will be:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} \times p_k \times p_j\} & \text{if } i < j \end{cases}$$

- Store the value of k in $s[i,j]$ for which $m[i,j]$ is minimum.

Step 3: Computing the optimal costs

- We can find the solution of the above recurrence by using two methods: Bottom-up or top-down.
- We compute overlapping sub-problems only once to avoid the exponential time cost.
- A_i is on dimensions $p_{i-1} \times p_i$.
- Table $m[1 \ldots n, 1 \ldots n]$ is used for storing $m[i, j]$ and $s[1 \ldots n-1, 2 \ldots n]$ is used to store the value of k for which $m[i, j]$ is minimum for all $i \leq k < j$.

Bottom-up Approach:

MATRIX-CHAIN-ORDER(p,n)

1. Let $m[1 \ldots n, 1 \ldots n]$ and $s[1 \ldots n-1, 2 \ldots n]$
2. for $i = 1$ to n
3. $\quad m[i, i] = 0$
4. for $l = 2$ to n // l is the chain length
5. $\quad$ for $i = 1$ to $n - l + 1$
6. $\quad\quad j = i + l - 1$
7. $\quad\quad m[i, j] = \infty$
8. $\quad\quad$ for $k = i$ to $j - 1$
9. $\quad\quad\quad q = m[i, k] + m[k + 1, j] + p_{i-1} \times p_k \times p_j$
10. $\quad\quad\quad$ if $q < m[i, j]$
11. $\quad\quad\quad\quad m[i, j] = q$
12. $\quad\quad\quad\quad s[i, j] = k$
13. return m, s

Top-down Approach:

MEMOIZED-MATRIX-CHAIN(p,n)

1. Let $m[1 \ldots n, 1 \ldots n]$
2. for $i = 1$ to n
3. for $j = i$ to n
4. $m[i, i] = \infty$
5. return LOOKUP-CHAIN(m,s,p,1,n)

LOOKUP-CHAIN(m,s,p,i,j)

1. if $m[i, j] < \infty$
2. return $m[i, j]$
3. if $i = j$
4. $m[i, j] = 0$
5. else for $k = i$ to $j - 1$

6. q =LOOKUP-CHAIN(m,s,p,i,k)
   +LOOKUP-CHAIN(m,s,p,k+1,j)+$p_{i-1}p_k p_j$
7. if $q < m[i,j]$
8. $m[i,j] = q$
9. $s[i,j] = k$
10. return $m[i,j]$

Step 4: Constructing an optimal solution

- Table $m[i, j]$ gives the number of scalar multiplication to multiply matrices from A_i to A_j but does not show the order of multiplication
- Table $s[i, j]$ store the position of the parenthesis to partition the matrices from A_i to $A_k = A_{s[i,j]}$ and $A_{s[i,j]+1}$ to A_j.
- So the multiplication will be $(A_i \dots A_{s[i,j]})$ and $(A_{s[i,j]+1} \dots A_j)$

PRINT-OPTIMAL-PARENS(s,i,j)

1. if $i = j$
2. print A_i
3. else print "("
4. PRINT-OPTIMAL-PARENS$(s, i, s[i, j])$
5. PRINT-OPTIMAL-PARENS$(s, s[i, j] + 1, j)$
6. print ")"

Complexity of the MCM is $O(n^3)$

Optimal binary search trees

- Suppose we want to store English dictionary
- We can use any balanced binary search try to store the dictionary word
- If every word of dictionary is to be searched same number of times
- we can search any world in $O(\lg n)$
- Total time taken is $O(mn \lg n)$
  where $m =$ number of times each word to be searched
  $n =$ number of words
- This system works perfectly if every word is searched equal number of times.
- But if the some words are searched very frequently and some are very rare then this performance of this system goes slow

- An alternate of this is Optimal binary search trees
- Let a $K = \langle k_1, k_2, \ldots k_n \rangle$ sequence of n distinct key in the sorted order such that $k_1 < k_2 < \cdots < k_n$
- If we search keys not in database that we represent as dummy keys d_i
- Total key and dummy keys will be $d_0 < k_1 < d_1 < k_2 \ldots d_{n-1} < k_n < d_n$
- d_0 represents all keys less than k_1 and not in database, so are dummy keys
- d_n represents all keys greater then k_n and not in database, so are dummy keys
- d_i for $0 < i < n$ represents all keys greater then k_i and less than k_{i+1} and not in database, so are dummy keys.
- So Total no of n Keys are $k_1, k_2, \ldots k_n$

- So Total no of $n+1$ Dummy Keys are $d_0, d_1, d_2, \ldots d_n$
- Let probability of key k_i to be searched is p_i for $1 \leq i \leq n$.
- Let probability of dummy key d_i to be searched is q_i for $0 \leq i \leq n$.
- If we build some binary search tree for these keys and dummy keys then keys will be on internal nodes and represents successful search and dummy keys will be on external nodes and represents unsuccessful search.
- Every search is either successful finding some key k_i or unsuccessful finding some dummy key d_i so:

$$\sum_{k=1}^{n} p_k + \sum_{k=0}^{n} q_k = 1$$

- if k_r is the root of the tree then two subtree will be: one subtree having keys from k_1 to k_{r-1} and dummy keys d_0 to d_{r-1} and other subtree having keys from k_{r+1} to k_n and dummy keys d_r to d_n
- if we search any keys from k_i to k_j then dummy keys will be d_{i-1} to d_j for $1 \leq i \leq j \leq n$
- If $j = i - 1$ then there is no key but only one dummy key d_{i-1}
- Let

$$w[i,j] = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$$

- k_r is the root of the tree having keys from k_i to k_j and dummy keys from d_{i-1} to d_j then two subtree will be: one subtree having keys from k_i to k_{r-1} and dummy keys d_{i-1} to d_{r-1} and other subtree having keys from k_{r+1} to k_j and dummy keys d_r to d_j
- The actual cost of a search equals the number of nodes examined i.e. the depth of the node found by the search in tree plus 1

- Then the expected cost of a search in tree for al keys and dummy keys is

$$E[1,n] = \sum_{k=1}^{n} (depth(k_k) + 1) .p_k + \sum_{i=0}^{n} (depth(d_k) + 1) .q_k$$

$$E[1,n] = \sum_{k=1}^{n} depth(k_k).p_k + \sum_{i=0}^{n} depth(d_k).q_k + \sum_{k=1}^{n} p_k + \sum_{k=0}^{n} q_k$$

$$E[1,n] = \sum_{k=1}^{n} depth(k_k).p_k + \sum_{i=0}^{n} depth(d_k).q_k + 1$$

- The expected cost of a search in tree for keys from k_i to k_j and dummy keys d_{i-1} to d_j is

$$E[i,j] = \sum_{k=i}^{j} (depth(k_k) + 1) \cdot p_k + \sum_{k=i-1}^{j} (depth(d_k) + 1) \cdot q_k$$

$$E[i,j] = \sum_{k=i}^{j} depth(k_k).p_k + \sum_{k=i-1}^{j} depth(d_k).q_k + \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$$

$$E[i,j] = \sum_{k=i}^{j} depth(k_k).p_k + \sum_{k=i-1}^{j} depth(d_k).q_k + w[i,j]$$

Step 1: The structure of an optimal binary search tree

- Let T is a optimal tree having subtree $T^{'}$ with keys $k_i \ldots k_j$ and dummy keys $d_{i-1} \ldots d_j$ for some $1 \leq i \leq j \leq n$
- Then $T^{'}$ will also be a optimal subtree
- We can find the optimal cost of the tree by optimal cost of the subtree
- Take k_r as the root from keys k_i to k_j and dummy keys d_{i-1} to d_j
- Now there are two subtrees one having keys from k_i to k_{r-1} with dummy keys from d_{i-1} to d_{r-1}
- The second tree having keys from k_{r+1} to k_j with dummy keys from d_r to d_j

- Now the total cost of the tree will be sum of cost of left subtree having keys from k_i to k_{r-1} with dummy keys from d_{i-1} to d_{r-1} plus cost of right subtree having keys from k_{r+1} to k_j with dummy keys from d_r to d_j plus cost of key k_r
- d_i is the dummy key between the keys k_i and k_{i+1}
- If we choose k_i as the root then there is no key (keys from k_i to k_{i-1}) on left subtree but only one dummy key d_{i-1}
- If we choose k_j as the root then there is no key (keys from k_{j+1} to k_j) on right subtree but only one dummy key d_j
- By taking all key from k_i to k_j as root one by one, we can find the optimal cost of the tree

Step 2: A recursive solution

- Let a subproblem with keys k_i to k_j and dummy keys d_{i-1} to d_j where $i \geq 1$, $j \leq n$ and $j \geq i-1$
- Let $e[i,j]$ is the expected cost of searching an optimal binary search tree containing the keys k_i to k_j and dummy keys d_{i-1} to d_j
- Finally we have to find out $e[1,n]$
- Let $w[i,j]$ is the sum of the probability of all the keys $k_i \ldots k_j$ and dummy keys $d_{i-1} \ldots d_j$ and $w[1,n] = 1$

$$w[i,j] = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$$

- When $j = i - 1$ then no key but only dummy key so $e[i, i-1] = q_{i-1}$ and $w[i, i-1] = q_{i-1}$
- When $j > i - 1$, then take a key k_r as root such that left subtree having keys from k_i to k_{r-1} and right subtree having keys from k_{r+1} to k_j is optimal
- When a tree becomes subtree of a root node then hight of the each node of the tree increases by one.
- The expected search cost of this subtree increases by the sum of all the probabilities in the subtree i.e.

$$e[i, r-1] = e[i, r-1] + w[i, r-1]$$

$$e[r+1, j] = e[r+1, j] + w[r+1, j]$$

- Taking k_r as the root $e[i,j]$ becomes

$$e[i,j] = e[i,r-1] + w[i,r-1] + p_r + e[r+1,j] + w[r+1,j]$$

$$e[i,j] = e[i,r-1] + e[r+1,j] + w[i,j]$$

$$\because w[i,j] = w[i,r-1] + p_r + w[r+1,j]$$

- The recursive solution will be:

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w[i,j]\} & \text{if } j > i-1 \end{cases}$$

Step 3: Computing the expected search cost of OBST

- Optimal cost of OBST is stored in $e[1 \ldots n+1, 0 \ldots n]$ that is size of upper triangular matrix is $(n+1) \times (n+1)$.
- $e[1,0]$ is used to store the value of d_0 and $e[n+1, n]$ is used to store the value of d_n
- $e[i,j]$ is used to store the value of optimal search cost of the keys from k_i to k_j for $1 \leq i \leq n+1, 0 \leq j \leq n,$ and $j \geq i-1$
- $root[i,j]$ is used to store the root of the tree having keys from k_i to k_j for $1 \leq i \leq j \leq n$
- $e[i, i-1] = w[i, i-1] = q_{i-1}$ for $1 \leq i \leq n+1$
- $w[i,j] = w[i,j-1] + p_j + q_j$ for $1 \leq i \leq n+1, j > i-1$
- $e[i,j] = \min_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w[i,j]\}$ for $j > i-1$

Bottom-up Approach:

OPTIMAL-BST(p,q,n)

1. Let $e[1 \ldots n+1, 0 \ldots n], w[1 \ldots n+1, 0 \ldots n]$ and $root[1 \ldots n, 1 \ldots n]$
2. for $i = 1$ to $n+1$ //when no keys, will be only one dummy key
3. $\quad w[i, i-1] = q_{i-1}$
4. $\quad e[i, i-1] = q_{i-1}$
5. for $l = 1$ to n // l number of keys in the subtree
6. $\quad$ for $i = 1$ to $n - l + 1$
7. $\quad\quad j = i + l - 1$
8. $\quad\quad e[i, j] = \infty$
9. $\quad\quad w[i, j] = w[i, j-1] + p_j + q_j$
10. $\quad\quad$ for $r = i$ to j

11. $q = e[i, r-1] + e[r+1, j] + w[i, j]$
12. if $q < e[i, j]$
13. $e[i, j] = q$
14. $root[i, j] = r$
15. return $e, root$

Top-down Approach:

MEMOIZED-OPTIMAL-BST(p,q,n)

1. Let $e[1 \ldots n+1, 0 \ldots n], w[1 \ldots n+1, 0 \ldots n]$ and $root[1 \ldots n, 1 \ldots n]$
2. for $i = 1$ to $n+1$
3. for $j = i-1$ to n
4. $e[i,j] = \infty$
5. if $j = i-1$
6. $w[i,j] = q_{i-1}$
7. else
8. $w[i,j] = w[i,j-1] + p_j + q_j$
9. return LOOKUP-OBST(e,w,root,p,q,1,n)

LOOKUP-OBST(e,w,root,p,q,i,j)

1. if $e[i,j] < \infty$
2. return $e[i,j]$
3. if $j = i - 1$
4. $e[i,j] = q_{i-1}$
5. else for $r = i$ to j
6. q =LOOKUP-OBST(e,w,root,p,q,i,r-1) +LOOKUP-OBST(e,w,root,p,q,r+1,j)+$w[i,j]$
7. if $q < e[i,j]$
8. $e[i,j] = q$
9. $root[i,j] = r$
10. return $e[i,j]$

Step 4: Constructing an OBST

- Table $e[i, j]$ gives the cost of OBST for the keys k_i to k_j with dummy keys d_{i-1} to d_j
- Table $root[i, j]$ store the root node $k_r = k_{root[i,j]}$
- At $k_r = k_{root[i,j]}$ there will be two subtree: one left subtree with keys k_i to $k_{root[i,j]-1}$ and second right subtree with $k_{root[i,j]+1}$ to k_j
- First call will be:
- if $n \geq 1$ //atleast one key
- $r = root[1, n]$
- print k_r is the root of the OBST
- PRINT-OBST$(root, i, r-1, r, "left")$
- PRINT-OBST$(root, r+1, j, r, "right")$
- else
- print d_0 is the root of the OBST

PRINT-OBST(root,i,j,r,child)

1. if $i \leq j$
2. $\quad c = root[i, j]$
3. $\quad$ print k_c is child of k_r
4. $\quad$ PRINT-OBST$(root, i, r-1, c, "left")$
5. $\quad$ PRINT-OBST$(root, r+1, j, c, "right")$

Complexity of the OBST is $O(n^3)$

Class Test I ADA, IT 5^{th} *Sem*, Dated: 06^{th} *Sep*, 2019

Even numbered students will attempt even numbered questions and odd numbered students will attempt odd numbered questions.

Q1. Prove that $\lg n! = O(n \lg n)$

Q2. Find the solution of $T(n) = 3T(\sqrt[3]{n}) + \lg_3 n$

Q3. Find the cost to multiply the matrices whose dimensions are given as 15,5,10,2,5.

Q4. Find the maximum length subsequence of the two sequences $< 00110101 >$ and $< 101100110 >$

Binomial Coefficient

- $(x+a)^n$ can be expended as ${}^nC_0x^na^0 + {}^nC_1x^{n-1}a^1 + \cdots + {}^nC_kx^{n-k}a^k + \ldots {}^nC_nx^0a^n$
- Coefficient of the term $x^{n-k}a^k$ is know as Binomial Coefficient $C(n,k) = {}^nC_k$
- $C(n,k) = \frac{n!}{n-k!k!}$
- $C(n,k) = C(n-1,k-1) + C(n-1,k)$ if $n > k > 0$
- $\Rightarrow \frac{n-1!}{n-k!k-1!} + \frac{n-1!}{n-k-1!k!}$
- $\Rightarrow \frac{n-1!}{n-k!k-1!}\frac{k}{k} + \frac{n-1!}{n-k-1!k!}\frac{n-k}{n-k}$
- $\Rightarrow \frac{(n-1)!k}{n-k!k!} + \frac{(n-1)!(n-k)}{n-k!k!}$
- $\Rightarrow \frac{(n-1)!(k+n-k)}{n-k!k!}$
- $\Rightarrow \frac{n!}{n-k!k!}$
- $\Rightarrow C(n,k)$
- $C(n,0) = C(n,n) = 1$

Computing a Binomial Coefficient

Bottom-up Approach:

No optimal solution only fixed value

BC(c,n,k)

1. Let $c[0 \ldots n, 0 \ldots k]$
2. for $i = 0$ to n
3. for $j = 0$ to i
4. if $i = j$ or $j = 0$
5. $c[i, j] = 1$
6. else
7. $c[i, j] = c[i - 1, j - 1] + c[i - 1, j]$
8. return c

Top-down Approach:

Mamoized-BC(c,n,k)

1. Let $c[0 \ldots n, 0 \ldots k]$
2. for $i = 0$ to n
3. for $j = 0$ to i
4. $c[i, j] = 0$
5. return $BC(c, n, k)$

$BC(c, i, j)$

1. if $c[i, j] > 0$
2. return $c[i, j]$
3. if $i = j$ or $j = 0$
4. return 1
5. return $c[i, j] = BC(c, i - 1, j - 1) + BC(c, i - 1, j]$

Complexity of the Knapsack is $O(nk)$

Thank you

Please send your feedback or any queries to
ashokyadav@galgotiasuniversity.edu.in