Unit 11 Dynamic Programming – 2

Structure:

11.1 Introduction

In the previous unit we studied some concepts of dynamic programming. As you know, dynamic programming is defined as an optimization technique that is used for particular classes of backtracking algorithms where the sub problems are repeatedly solved.

The basic steps for dynamic programming are given below:

1) Develop a mathematical notation that is used to find a solution and sub solutions for the problem.
2) Prove the solution using the Principle of Optimality.
3) Derive a recurrence relation that solves the sub solutions using the mathematical notation in step 1.
4) Write an algorithm to compute the recurrence relation.

Sometimes, we have to solve problems optimally. At some other time a non-optimal solution also gives a good solution. We cannot judge a problem by a single criterion. Optimization of a problem is useful in any type of problem we have to solve.

This unit defines the Principle of Optimality and analyzes binary search trees using dynamic programming. It also introduces the Knapsack problem

and solves an instance of it using dynamic programming and memory functions.

Objectives:

After studying this unit you should be able to:

- define 'Principle of Optimality'
- analyze optimal binary search trees with an example
- describe Knapsack problem with an example
- explain memory functions with an example

11.2 Principle of Optimality

Principle of Optimality is defined as a basic dynamic programming principle which helps us to view problems as a sequence of sub problems.

Richard Ernest Bellman, a mathematician, invented the Principle of Optimality. This principle explains that an optimal path has the property that whatever the initial conditions and choices over some initial period, the decision variables chosen over the remaining period must be optimal for the remaining problem. Therefore, we can say that the optimal solution to a problem is a combination of optimal solutions of its sub problems.

We might face a difficulty in converting the Principle of Optimality into an algorithm, as it is not very easy to identify the sub problems that are relevant to the problem under consideration. Bellman developed an equation for the Principle of Optimality. We can study this equation only with the help of dynamic programming concepts.

All optimization problems tend to minimize space and time, and maximize profits. There is a mathematical function defined for this namely, optimization function. Every instance of a dynamic programming problem should be tracked as it involves the use of various different sub problems to solve it. This information about the current situation required to make a correct decision is known as a state. The variables we choose for solving the problem at any point of time are called control variables. At every state we have to choose the control variable with the help of the previous state. The rule that determines the controls as a function of states is known as policy function. The best possible value of the problem objective, written as a function of the state, is called the value function.

Bellman formulated an equation for the Principle of Optimality during the early stages in technology development. The computers used during that stage were not as powerful as we use now. This principle may help in advanced dynamic programming applications which support larger dimensions than that used today.

We can derive the Bellman equation using a step by step recursive process of writing down the relationship between the value function of one stage and the value function of the next stage.

The Bellman equation for Principle of Optimality is given as:

$$V_N(x_0) = \max_a \sum_{x1} p_1(x_1|x_0, a_0)\left[r_1(x_0, a_0, x_1) + V_{N-1}(x_1)\right] \quad \textbf{Eq: 11.1}$$

In the equation Eq: 11.1, $V_N(x_0)$ is the value function where N is the number of decision steps. We are aware that the value function explains the best possible value of the objective, as a function of the state x_0. Here $p_1(x_1|x_0, a_0)[r_1(x_0, a_0, x_1) + V_{N-1}(x_1)]$ are the policies with respect to distribution.

Self Assessment Questions

1. Principle of _____ is defined as a basic dynamic programming principle which helps us to view problems as a sequence of sub problems.
2. _____, a mathematician, invented the Principle of Optimality.
3. All optimization problems tend to minimizing cost, time and maximizing _____.

11.3 Optimal Binary Search Trees

Now that we have discussed the basic Principle of Optimality, let us optimize a binary search tree. First, let us define a binary search tree.

Binary search tree – Binary search tree is defined as a binary tree with the following properties:

- The values of elements in the left sub-tree of a node are lesser than the node's value.
- The values of elements in the right sub-tree of a node are greater than the node's value.
- The right and left sub-trees of a node are also binary search trees.

Binary search trees are node based data structures used in many system programming applications for managing dynamic sets. An example for binary search trees is given in figure 11.1.

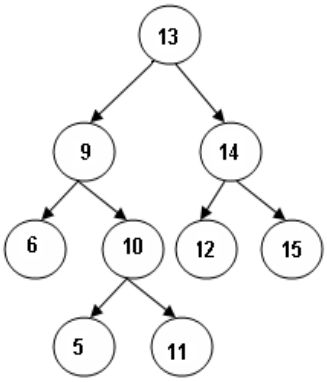

Figure 11.1: Binary Search Tree

We use binary search trees for applications such as sorting, searching and in–order traversal.

The four main operations that we perform on binary trees are:

Searching – Here we match the searching element with the root node first, left sub-tree, right sub-tree until we find the node or if no nodes are left. We can search a binary search tree using the pseudocode given below.

Pseudocode for Searching a Binary Search Tree

```
find(Y, node){
if(node = NULL)
        return NULL
if(Y = node:data)
        return node
else if(Y < node:data)
        return find(Y,node:leftChild)
else if(Y> node:data)
       return find(Y,node:rightChild)
```

Insertion – If the root node of the tree does not have any value, we can insert the new node as the root node. For inserting a new element in an existing binary search tree, first we compare the value of the new node with

the current node value. If the value of the new node is less than the current node value, we insert it as a left sub-node. If the value of the new node is greater than the current node value, then we insert it as a right sub-tree.

Let us now discuss the pseudocode for inserting a new element in a binary search tree.

Pseudocode for Inserting a Value in a Binary Search Tree

```
//Purpose: insert data object Y into the Tree
//Inputs: data object Y (to be inserted), binary-search-tree node
//Effect: do nothing if tree already contains Y;
// otherwise, update binary search tree by adding a new node containing
data object Y
insert(Y, node){
        if(node = NULL){
           node = new binaryNode(Y,NULL,NULL)
           return
}
if(Y = node:data)
        return
else if(Y < node:data)
        insert(Y, node:leftChild)
else // Y > node:data
        insert(Y, node:rightChild)
}
```

Deletion – If the node to be removed has no children, we can just delete it. If the node to be removed has one child, then the node is deleted and the child is connected directly to the parent node.

To remove a node which has two children, we adopt the following procedure:

1) We find the minimum value in the right sub-tree
2) We replace the node to be removed with the minimum value found.
3) We then remove the duplicate value from the right sub-tree.

We can delete an existing element from a binary search tree using the following pseudocode:

Pseudocode for Deleting a Value from a Binary Search Tree

```
//Purpose: delete data object Y from the Tree
//Inputs: data object Y (to be deleted), binary-search-tree node
//Effect: do nothing if tree does not contain Y;
// else, update binary search tree by deleting the node containing data
object Y
delete(Y, node){
if(node = NULL) //nothing to do
        return
if(Y < node:data)
delete(Y, node:leftChild)
else if(Y > node:data)
delete(Y, node:rightChild)
else { // found the node to be deleted! Take action based on number of
node children
        if(node:leftChild = NULL and node:rightChild = NULL){
         delete node
        node = NULL
        return}
else if(node:leftChild = NULL){
 tempNode = node
node = node:rightChild
delete tempNode}
else if(node:rightChild = NULL){
(similar to the case when node:leftChild = NULL)
}
else { //replace node:data with minimum data from right sub-tree
        tempNode = findMin(node.rightChild)
        node:data = tempNode:data
        delete(node:data,node:rightChild)
}
}
}
```

Traversal – Traversing a binary search tree involves visiting all the nodes in the tree. First we visit the root node. Then we visit its left and right sub-trees. We can visit a node only once. We can traverse a binary tree recursively using the following pseudocode:

Pseudocode for Traversing a Binary Search Tree

```
PROCEDURE PreOrder (Binary_Tree_Node N)
BEGIN
ProcessNode(N)
If (N's left child is NOT NULL)
BEGIN
PreOrder(N's left child)
END
If (N's right child is NOT NULL)
BEGIN
PreOrder(N's right child)
END
END
```

The time taken to perform operations on a binary search tree is directly proportional to the height of the tree. The insertion, deletion and search operations has an average case complexity of O(log n), where n is the number of nodes in the binary search tree.

If we know the frequency of operations performed on a binary search tree, then, instead of modifying it, we can optimize it. The basic criterion for optimizing a binary tree is that every binary tree should have a root node and optimal sub-trees under it. We can use the principles of dynamic programming to optimize a binary search tree.

Let us consider an example to search six elements P, Q, R, S, T and U with probabilities 1/8, 1/32, 1/16, 1/32, 1/4 and 1/2. Here we can form 132 different binary search trees. One of them is shown in the figure 11.2.

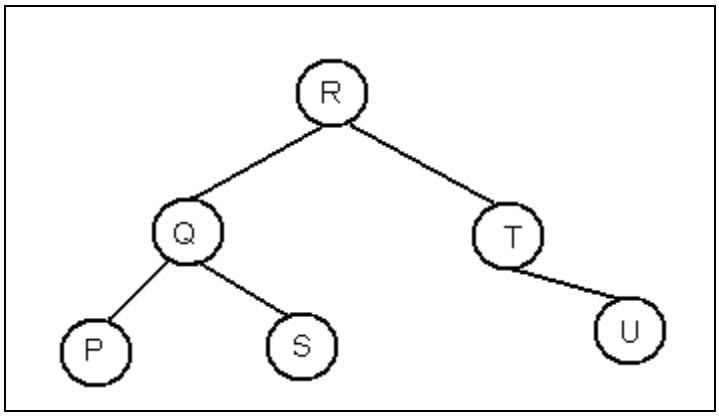

Figure 11.2: Binary Search Trees

For the trees in the figure 11.2, we can find the average number comparisons from the equation Eq: 11.2.

Average number comparisons for a tree = $\sum n * x$ **Eq: 11.2**

Where, n is the value of the node and x is the level of the node in the tree.

For the tree of figure 11.2, the average number of comparisons is given as (1/16 * 1) + (1/32 * 2) + (1/4 * 2) + (1/8 * 3) + (1/32 * 3) + (1/2 * 3) ≈ 2.6. Here we can see that the tree is not optimized. We can generate all 132 binary search trees and analyze them to find the optimal one. If we start this task using the binary search tree algorithms the task becomes exhaustive.

The total number of binary search trees with n elements is equal to the n^{th} Catalan number, c(n), given in Eq 11.3.

$$c(n) = \binom{2n}{n}\frac{1}{n+1} \qquad \text{for } n>0, \quad c(0) \approx 1 \qquad \textbf{Eq: 11.3}$$

The equation Eq: 11.3 reaches infinity as fast as $4^n/n^{1.5}$.

Let us use dynamic programming approach to solve this problem. Let a_1, $a_2 \ldots a_n$ be the distinct elements given in ascending order and let p_1, $p2 \ldots p_n$ be the probabilities of searching the elements. Let c[i,j] be the smallest average number of comparisons made in a binary search tree T_i^j of elements ai….aj, where i,j are some integer indices, $1 \leq i \leq j \leq n$.

Now let us find the values of C[i,j] for its the sub instances. We have to choose a root a_k for keys $a_i \ldots a_j$, so as to derive the recurrence relation using dynamic programming. For such binary search tree, the root consist of the key a_k, the left sub-tree T_i^{k-1} contains keys ai…ak-1 optimally arranged and the right sub-tree T_{k+1}^{j} contains keys $a_{k+1} \ldots a_j$ also optimally arranged. Here we are taking advantage of the Principle of Optimality. If we start counting tree levels at 1 then we can derive the following recurrence relation:

$$C[i,j] = \min_{i \le k \le j} \Big\{ p_k \cdot 1 + \sum_{s=1}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1) + \sum_{s=k+1}^{i} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1) \Big\}$$

$$= \min_{i \le k \le j} \Big\{ p_k + \sum_{s=1}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=i}^{k-1} p_s + \sum_{s=k+1}^{i} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=k+1}^{j} p_s \Big\}$$

$$= \min_{i \le k \le j} \Big\{ \sum_{s=1}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=i}^{i} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=i}^{j} p_s \Big\}$$

$$= \min_{i \le k \le j} \{ C[i, k-1] + C[k+1, j] \} + \sum_{s=i}^{j} p_s$$

Thus, we have the recurrence relation given in Eq 11.3.

$$C[i,j] = \min_{i \le k \le j} \{ C[i, k-1] + C[k+1, j] \} + \sum_{s=i}^{j} p_s \text{ for } 1 \le i \le j \le n. \qquad \textbf{Eq: 11.4}$$

In the recurrence relation given by Eq: 11.4, let us assume that C[i,i-1] ≈ 0 for $1 \le i \le n+1$. This we can interpret as the number of comparisons in the

empty tree. The figure 11.3 shows the values required to compute the C[i,j] formula.

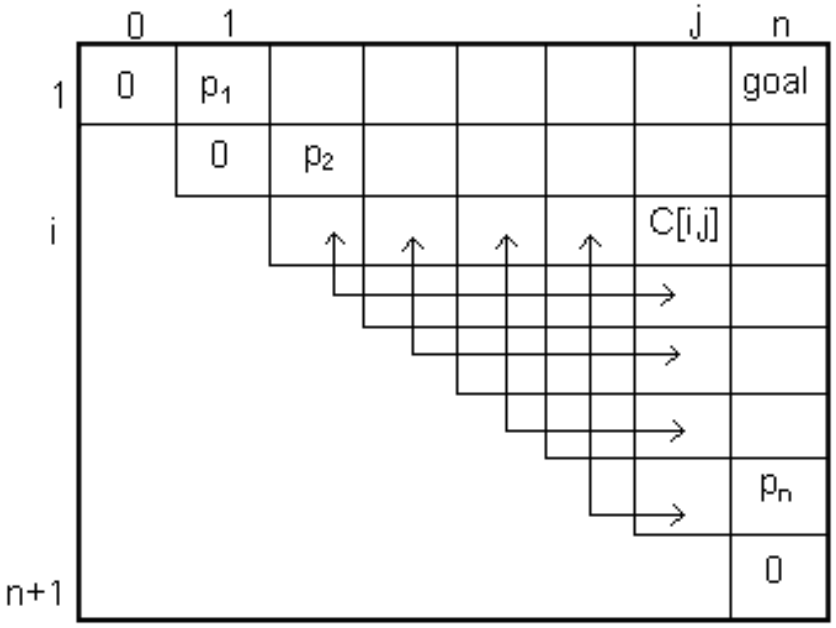

Figure 11.3: Dynamic Programming Algorithm for Optimal Binary Search Tree

In figure 11.3, we can find the values at row i and columns to the left of column j and in column j and the rows below the row i. The arrows shown point to the pairs of entries that are added up and the smallest one is recorded as the value of C[i,j]. We have to fill the table along its diagonal, starting with zeroes on the main diagonal and with probabilities given as p_i, $1 \le i \le n$, and moving toward the upper right corner.

This algorithm helps us to compute C[1,n], the average number of comparisons for the successful searches in the optimal binary search tree. We have to maintain another two dimensional table to record the value of k for which the minimum is achieved. The table will be same as the one in figure 11.3, and will be filled in the same manner. The table entries will start at R[i,i] for $1 \le i \le n$ and is used to find the optimal solution.

Let us next discuss the dynamic programming algorithm for binary search tree optimization.

Dynamic Programming Algorithm for Binary Search Tree Optimization

```
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//optimal binary search tree and table of sub trees' roots in the optimal
//binary search tree
for i ← 1 to n do
        C[i,i-1] ← 0
        C[i,i] ←P[i]
R[i,i] ←i
C[n+1,n] ←0
for d←1 to n-1 do//diagonal count
        for i←1 to n - d do
            j←i + d
    minval ← ∞
for k←I to j do
if C[i, k-1]+C[k+1, j]< minval
minval←C[I,k-1]+ C[k+1,j]; kmin←k
R[i,j] ←kmin
sum ←P[i];for s←i+1 to j do sum←sum + P[s]
C[I,j] ← minval + sum
return c[1,n],R
```

Let us now trace the dynamic programming algorithm for binary search tree optimization.

Algorithm Tracing for Binary Search Tree Optimization

```
P[5]={1,2,3,4,5}, n=5;
C[5,5]=0//array for comparisons in successful search
R[5,5]=0//root array
for i = 1 to 5 do //this loop will occur from i = 1 to i = 5
        C[1,0]=0;
```

```
C[1,1]=P[1]=1// value of first element in the array is assigned to C[1,1]
R[1,1]=1;
C[6,5]=0;
For d=1 to 5-1// this loop will occur from d = 1 to d = 4
        for i=1 to 5-1
            j= 1+1
    minval= infinite value
for k=1 to 2 do // this loop will occur from k = 1 to k = 2
if C[1,0] + C[2,2] < infinite value
minval = C[1,0] + C[2,2]=0+0
kmin=1;R[1,2]=1
Sum=P[1]=1;for s= 1+1 to 2 do
sum=P[1]+P[s]=1+2=3// s is assigned a value 2 in the previous step
C[1,2]=minval + sum = 0 + 3=3
return 3,1
```

The space efficiency of this algorithm is in quadratic terms and the time efficiency is in cubic terms. We can also see that the values of the root table are always non-decreasing along each row and column. This limits values for R[i,j] to the range r[i,j-1],….r[i+1,j] and makes it possible to reduce the running time of the algorithm to Θ (n^2).

11.3.1 Solving binary search trees using dynamic programming

Let us illustrate the above mentioned algorithm using the four keys that we used in the previous section. The keys and the probabilities are given in table 11.1.

Table 11.1: Table of Keys and Probabilities

Key	P	Q	R	S	T	U
Probability	1/8	1/32	1/16	1/32	1/4	1/2

At initial stage the main table - table 11.2 is given as:

Table 11.2: Main Table

	0	1	2	3	4	5	6
1	0	1/8					
2		0	1/32				
3			0	1/32			
4				0	1/16		
5					0	1/4	
6						0	1/2
7							0

Let us compute C[1,2] as shown in equation Eq:11.5:

$$C[1,2] = \min_{\substack{k=1: C[1,0]+C[2,2]+\sum_{s}^{2} Ps = 0+1/32+1/2= \\ k+1: C[1,1]+C[3,2]+\sum_{s=1}^{2} Ps = 1/8+0+1/2=}} = 3/16 \quad \textbf{Eq:11.5}$$

Thus, from the two possible binary trees P and Q, the root of the optimal tree has index 2 and the average number of comparisons in a successful search in the tree is 3/6.

Let us complete the above given table. The completed table 11.3 is the main table.

Table 11.3: Main Table

	0	1	2	3	4	5	6
1	0	1/8	3/16	9/32	15/32	31/32	63/32
2		0	1/32	3/32	7/32	19/32	47/32
3			0	1/32	1/8	15/32	21/16
4				0	1/16	3/8	19/16
5					0	¼	1
6						0	1/2
7							0

Thus we can compute the average numbers of key comparisons in the optimal tree to be 63/32. According to these probabilities, the optimal tree is shown in the figure 11.4.

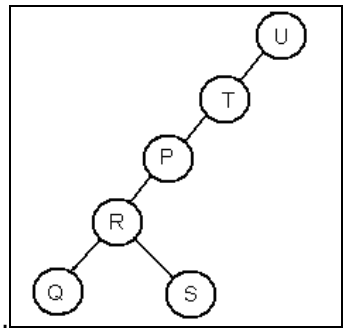

Figure 11.4: Optimal Binary Search Tree

Self Assessment Questions

4. _____are node based data structures used in many system programming applications for managing dynamic sets.
5. The Insertion, deletion and search operations of a binary search tree has an average case complexity of _____.
6. The time taken to perform operations on a binary search tree is directly proportional to the _____ of the tree.

11.4 Knapsack Problem

In this section we will define and analyze the Knapsack problem. Let us first define the Knapsack problem.

If a set of items are given, each with a weight and a value, determine the number of items that minimizes the total weight and maximizes the total value.

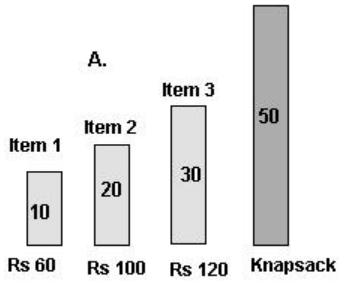

Figure 11.5: Knapsack Example

Consider a situation where a thief breaks into a store and tries to fill his knapsack with as much valuable goods as possible. The figure 11.5 given

above shows the number of goods with its value and weight. There are 3 items given with weights 10 kg, 20 kg and 30 kg with values Rs.60, Rs.100 and Rs. 120 respectively. The capacity of the knapsack is given as 50 kg. We have to fill the knapsack with the items appropriately to get the maximum value, but not exceeding the weight 50Kg

Let us try to fill the knapsack using different items as shown in the figure 11.6.

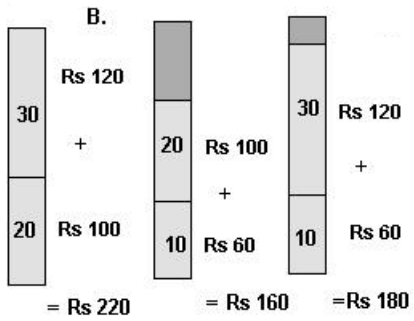

Figure 11.6: Different Ways of Filling the Knapsack

Firstly we try to fill it using item 2 and 3 and the values add up to Rs 220. Secondly we try to fill the knapsack using items 1 and 2 but these weights do not fill the knapsack completely. Thirdly, we try to fill it with items 1 and 2. This also does not fill the knapsack.

Now let us see the best possible solution for this problem from the figure 11.7.

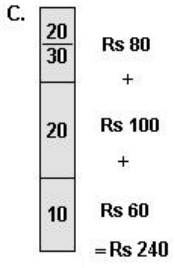

Figure 11.7: Optimal Solution for Knapsack Problem

Here we take items 1 and 2 as such and we take the 20/30 part of item 3. Thus the values add up to Rs 240, which is the maximum value. Now let us formally define the Knapsack problem.

Formal definition: There is a knapsack of capacity c > 0 and N items. Each item has value $v_i > 0$ and weight $w_i > 0$. Find the selection of items (δ_i = 1 if selected, 0 if not) that fit, $\sum_{i=1} N\delta_i w_i \leq c$ and the total value, $\sum_{i=1} N\delta_i v_i$ is maximized. This is known as the 0-1 Knapsack problem or the Binary Knapsack problem.

Let us see the Dynamic programming algorithm for the Knapsack problem:

Dynamic Programming Algorithm for Knapsack Problem

```
Dynamic knapsack (v,w,n,W)
FOR w = 0 TO W
   DO  c[0, w] = 0
FOR i=1 to n
   DO c[i, 0] = 0
      FOR w=1 TO W
         DO IF wi ≤ w
            THEN IF  vi + c[i-1, w-wi]
               THEN c[i, w] = vi + c[i-1, w-wi]
               ELSE c[i, w] = c[i-1, w]
            ELSE
               c[i, w] = c[i-1, w]
```

Let us now trace the dynamic programming algorithm for Knapsack problem.

Algorithm Tracing for Knapsack problem

```
v[3] = {1,2,3}, W= 5,n=3,C[5,5]=0 w[5]=0//wi and vi are arrays for weights
and values
FOR w = 0 TO 5//  this loop will occur from w = 0 to w = 5
   DO  c[0, 0] = 0
FOR i=1 to 3 // this loop will occur from i = 1 to i = 3
   DO c[1, 0] = 0
      FOR w=1 TO 5 // this loop will occur from w = 1 to w = 5
         DO IF 0 ≤ 0
            THEN IF  v1 + c[1-1, w-w1] //this value is calculated as 1+0=1
           THEN c[1, 0] = v1 + c[1-1, w-w1] //this value is calculated as
1+0=1
               ELSE c[1, 0] = c[1-1, 0]
            ELSE
               c[1, 0] = c[1-1, 0]
```

The different types of Knapsack problems are:

Fractional Knapsack problem – If we have materials of different values per unit volume and maximum amounts, the Fractional Knapsack problem finds the most valuable mix of materials which fit in a knapsack of fixed volume. We have to take as much as possible material that is most valuable per unit volume. Continue this process until the knapsack is full.

Bounded Knapsack problem – If we have the types of items of different values and volumes, find the most valuable set of items that fit in a knapsack of fixed volume. Here the number of items of each type is unbounded. This is an NP-hard optimization problem.

Now let us design a dynamic programming algorithm for the Knapsack problem. We have n number of items with weights w_1, w2…w_n and values v_1, v2….v_n. The capacity of knapsack is given as W. We have to find the most valuable subset of items that fit into the knapsack. Here, we assume that the knapsack capacity and the weights given are positive integers and the item values are not necessarily integers.

As we have done for every problem in dynamic programming, we have to form a recurrence relation to solve the Knapsack problem. This recurrence relation expresses the problem using its sub instances.

Let the instance defined by the first i items be $1 \leq I \leq n$, the weights be $w_1 \ldots w_i$ and the values be $v_1 \ldots v_i$. The capacity of knapsack is given as j, where $1 \leq j \leq w$. Let us also assume that V[i,j] be the value of the most valuable subset of the first i items that fit into the knapsack with capacity j. V[i,j] gives the optimal solution to the Knapsack problem. We can split the 'i' number of items that fit into the knapsack with capacity j into two. These are as given below.

- We can have the subsets that do not include the i^{th} item. Here the value of the optimal subset is given as V[i-1,j].
- We can have the subsets that do include the i^{th} item. An optimal subset is made out of this item and another optimal subset from first i-1 items that fit into the knapsack of capacity j-1. Here the value of the optimal subset is given as $v_i + V[i-1,j-w_i]$.

The value of an optimal solution from these two feasible subsets of the first i items is the maximum of these two values. If we cannot fit the i^{th} item

in the knapsack, then the value of an optimal solution from the first i items is the same as the value of an optimal subset selected from the first i-1 items. Thus we can arrive at a recurrence relation as given in equation Eq: 11.6.

$$V[i,j] = \begin{cases} \max\{V[i-1], v_i + V[i-1, w_i]\} \rightarrow if j - w_i \geq 0 \\ V[i-1, j] \rightarrow if j - w_i < 0 \end{cases} \quad \textbf{Eq:11.6}$$

We can define the initial conditions as

V[0,j] = 0 for $j \geq 0$ and v[i,0] = 0 for $i \geq 0$.

Now we have to find V[n,W], the maximum value of a subset of the n given items that fit into the knapsack of capacity W. This should be an optimal subset. Table 11.4 illustrates the values computed from the equations. We can fill the table either row wise or column wise.

To compute the entry in the i^{th} row and the j^{th} column, V[i,j]:

- We compute the maximum of the entry in the previous row and the same column.
- We compute the sum of v_i, the entry in the previous row and w_i columns to the left.

Let us compute the Knapsack problem using the table 11.4.

Table 11.4: Table for Solving the Knapsack Problem

	0	$j-w_i$	J	W
0	0	0	0	0
i-1	0	$V[i-1,j-W_1]$	V[i-1,j]	
I	0		V[i,j]	
N	0			goal

11.4.1 Solving Knapsack problem using dynamic programming

Let us solve an instance of Knapsack problem using dynamic programming.

Consider the following data given in table 11.5:

Table 11.5: Sample Data for Knapsack Problem

Item	1	2	3	4
Weight	5	4	6	3
Value	Rs.10	Rs.40	Rs.30	Rs.50

Knapsack capacity is given as W=10.

If we apply the recurrence formulas to this set of data, then we will get the following table 11.6.

Table 11.6: Example Table for Knapsack Problem

1	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

w_1= 5, v_1=10

w_2= 4, v_2= 40

w_3= 6, v_3= 30

w_4= 3, v_4= 50

We can compute the maximum value of V[4,10] as Rs.90. We can use the table to track the optimal subset. Since V[4,10] $\neq$ V[3,10], item 4 is included in an optimal solution along with an optimal subset for filling 10-3=7 remaining units of the Knapsack capacity. This is represented as V[3,7]. Since V[3,7]=V[2,7], item, 3 is not a part of an optimal subset. Since V[2,7] $\neq$ V[1,7], item 2 is a part of an optimal solution. V[1,7-1] is left behind as the remaining composition. Similarly, V[1,6] $\neq$ V[0, 6] , therefore item 1 is included in the solution.

We can find the time efficiency and space efficiency of the algorithm as Θ (nW).The time required to find the composition of an optimal solution is in Θ (n + W).

Activity 1

Item	1	2	3	4
Weight	3	5	2	4
Value (in Rs.)	10	15	25	45

Knapsack capacity is given as W=10. Analyze the Knapsack problem using dynamic programming with the help of the values given above.

Self Assessment Questions

7. The _____ expresses the problem using its sub-instances.
8. _____ is an NP-hard optimization problem.
9. The Knapsack problem minimizes the total _____ and maximizes the total value.

11.5 Memory Functions

In the previous section we solved the Knapsack problem using dynamic programming. In this section let us solve the Knapsack problem using memory functions.

As you know, dynamic programming deals with problems whose solutions satisfy a recurrence relation with overlapping sub problems. It uses a direct top down approach to find a solution to such recurrence. This is a very inefficient method. In the classic bottom up method, it fills a table with solutions to all smaller sub problems. Sometimes, we do not need solutions to all sub problems. This is where we use memory functions. The goal of using memory functions is to solve only the sub problems which are necessary. Memory functions use a dynamic programming technique called memoization in order to reduce the inefficiency of recursion that might occur.

We use memoization for finding solution to sub problems, so as to reduce recalculation. We use it in algorithms which have lots of recursive calls to the sub problems. Memory functions method solves problems using top

down approach, but maintains a table which is used for the bottom up dynamic programming algorithms. We can initialize the table values to a 'null' symbol. When we have to compute a new value:

- The method checks the corresponding entry in the table
- If this entry is not 'null', it is retrieved
- If this entry is 'null', then the value is computed using recursive calls and the results are entered in the table.

The algorithm for solving Knapsack problem using memory functions is given below.

Algorithm for Solving Knapsack Problem Using Memory Functions

```
//Input: A nonnegative integer i indicating the number of the first items
used //and a non negative integer j indicating the Knapsack's capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: uses as global variables input arrays weights[1..n],
values[1..n],and //table V[0..n,0..W] whose entries are initialized with –I's
except for row 0 //and column 0 which are initialized as 0's.
If V[i,j]<0
        If j< Weights[i]
        value ← MFKnapsack(i-1,j)
        else
        value ← max[MFKnapscak(i-1,j),
                Values[i] + MFKnapsack[i-1,,j-Weigths[i])]
        V[i,j] ←value
return V[i,j]
```

Let us now trace the above algorithm that uses memory functions.

Algorithm tracing for Knapsack Problem Using Memory Functions

```
i=2,j=2,weights[3]={1,2,3},values[3]={3,5,4}, V[5,5]= -1
n=5, W=5,V[0,0]=0
 If V[2,2]<0//value of V[2,2]= -1,which is less than 0
        If 2< Weights[2]//If 2<2 this condition is false , jump to else
        value = MFKnapsack(2-1,2)//this is a recursive loop
        else
value = max[MFKnapscak(2-1,2), Values[2]
```

```
                                    + MFKnapsack[2-1,,2-Weigths[2])]
// value = max[MFKnapsack(1,2), 5+ MFKnapsack[1,0]
        V[22] ←value
return V[2,2]
```

11.5.1 Solving Knapsack problem using memory functions

Now, let us solve the same instance given in the previous section by using memory functions.

The table 11.7 gives the result. We can see that, here only 13 out of 40 non trivial values are computed.

Table 11.7: Example Table for Knapsack Problem by Memory Functions

I	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	-	-	-	40	40	-	50
3	0	0	0	0	-	-	-	-	40	-	70
4	0	0	0	-	-	-	-	-	-	-	90

w_1= 5, v_1=10

w_2= 4, v_2= 40

w_3= 6, v_3= 30

w_4= 3, v_4= 50

We cannot expect more than a constant factor gain in using the memory function methods for the Knapsack problem. The time efficiency class of this method is the same as the bottom up algorithm. The space efficiency is less than that of bottom up algorithm.

Activity 2

Item	1	2	3	4
Weight	2	6	4	8
Value (in Rs.)	12	16	30	40

Knapsack capacity is given as W=12. Analyze the Knapsack problem using memory functions with the help of the values given above.

Self Assessment Questions

10. The goal of using _____ is to solve only the sub problems which are necessary.
11. Memory functions use a dynamic programming technique called _____ in order to reduce the inefficiency of recursion that might occur.
12. Memory functions method solves the problem using _____ approach.

11.6 Summary

Let us summarize what we have discussed in this unit.

In this unit we recollected the dynamic programming principle. Next we defined the Principle of Optimality. Principle of Optimality is defined as a basic dynamic programming principle which helps us to view problems as a sequence of sub problems. Next we defined the binary search tree and explained the various operations performed on the tree. We studied the applicability of the Principle of Optimality on the binary search trees.

In the next section we studied the Knapsack problem. The problem is defined for a set of items where each item has a weight and a value, and it determines the number of items that minimizes the total weight and maximizes the total value. We solved the Knapsack problem using the dynamic programming technique. Next we discussed memory functions. It uses a dynamic programming technique called memoization in order to reduce the inefficiency of recursion that might occur. We also solved the Knapsack problem using the memory functions.

11.7 Glossary

Terms	Description
Recurrence relation	Recurrence relation is an equation that recursively defines a list where each term of the list is defined as a function of the preceding terms.
NP hard problem	NP hard problems are problems that are solved in non deterministic polynomial time.

11.8 Terminal Questions

1. What is the basic Principle of Optimality?
2. What are the properties followed by a binary search tree?
3. Explain the steps for inserting an element in a binary search tree and give its pseudocode.
4. Explain the dynamic programming algorithm for solving a binary search tree.
5. Explain the algorithm to solve the Knapsack problem using the dynamic programming method.

11.9 Answers

Self Assessment Questions

1. Optimality
2. Richard Ernest Bellman
3. Profits
4. Binary search trees
5. O(log n)
6. Height
7. Recurrence relation
8. Bounded Knapsack problem
9. Weight
10. Memory functions
11. Memoization
12. Top down

Terminal Questions

1. Refer section 11.2 – Principle of optimization
2. Refer section 11.3 – Optimal binary search trees
3. Refer section 11. 3 – Optimal binary search trees
4. Refer section 11.4.1 – Solving binary search trees using dynamic programming
5. Refer section 11.5.1 – Solving Knapsack problem using memory functions

References

- Anany Levitin (2009). *Introduction to Design and Analysis of Algorithms.* Dorling Kindersley, India

- Kellerer Hans, Pferschy Ulrich, Pisinger David(2004). *Knapsack problems*. Springer, New York

E-References

- www.cecs.csulb.edu/~ebert/teaching/spring2002/cecs328/.../bst.pdf
- www.cs.ubc.ca/~nando/550-2006/lectures/l3.pdf
- http://lcm.csa.iisc.ernet.in/dsa/node91.html
- http://www.ehow.com/way_5172387_binary-tree-traversal-methods.html
- http://www.itl.nist.gov/div897/sqg/dads/HTML/knapsackProblem.html