UNIT 1 DYNAMIC PROGRAMMING

Structure **Page Nos.**

1.0 INTRODUCTION

In the earlier units of the course, we have discussed some well-known techniques, including the divide-and-conquer technique, for developing algorithms for algorithmically solvable problems. The divide-and-conquer technique, though quite useful in solving problems from a number of problem domains, yet in some cases, as shown below, may give quite *inefficient* algorithms to solve problems.

Example 1.0.1: Consider the problem of computing binomial coefficient $\binom{\mathbf{n}}{\mathbf{k}}$ (or in linear notation **c (n, k))** where n and k are given non-negative integers with n≥k. One way of defining and calculating the binomial coefficient is by using the following recursive formula

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = n \text{ or } k = 0 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ 0 & \text{otherwise} \end{cases} \qquad (1.0.1)$$

The following recursive algorithm named *Bin (n, k*), implements the above formula for computing the binomial coefficient.

Function Bin (n, k)

If k = n or k = 0 **then return** 1
else return Bin (n–1, k–1) + Bin (n–1, k)

For computing Bin (n, k) for some given values of n and k, a number of terms Bin (i, j), $1 \le i \le$ and $1 \le j \le k$, particularly for smaller values of i and j, are *repeatedly* calculated. For example, to calculate Bin (7, 5), we compute Bin (6, 5) and Bin (6, 4). Now, for computing Bin (6, 5), we compute Bin (5, 4) and Bin (5, 5). But for calculating Bin (6, 4) we have to calculate Bin (5, 4) again. If the above argument is further carried out for still smaller values, the number of repetitions for Bin (i, j) increases as values of i and j decrease.

For given values of n and k, in order to compute Bin (n, k), we need to call Bin (i, j) for $1 \le i \le n - 1$ and $1 \le j \le k-1$ and as the values of i and j decrease, the number of times Bin (i, j) is required to be called and executed generally increases.

The above example follows the *Divide-and-Conquer* technique in the sense that the task of calculating C(n, k) is replaced by the two relatively simpler tasks, viz., calculating C(n–1, k) and C (n–1, k–1). But this technique, in this particular case,

makes large number of *avoidable repetitions* of computations. This is not an isolated instance where the Divide-and-Conquer technique leads to *inefficient* solutions. In such cases, an alternative technique, viz., *Dynamic Programming*, may prove quite useful. This unit is devoted to developing algorithms using Dynamic Programming technique. But before, we discuss the technique in more details, let us briefly discuss *underlying idea* of the technique and the *fundamental difference* between Dynamic Programming and Divide-and-Conquer technique.

Essential idea of Dynamic Programming, *being quite simple, is that we should avoid calculating the same quantity more than once, usually by keeping a table of known results for simpler instances*. These results, in stead of being calculated repeatedly, can be retrieved from the table, as and when required, after first computation.

Comparatively, ***Divide-and-Conquer*** **is conceptually a** ***top-down*** **approach** for solving problems or developing algorithms, in which the problem is attempted *initially* with *complete* instance and gradually replacing the more complex instances by simpler instances.

On the other hand, ***Dynamic Programming*** **is a** ***bottom-up*** **approach** for solving problems, in which we first attempt the *simplest* subinstances of the problem under consideration and then gradually handle more and more complex instances, using the results of earlier computed (sub) instances.

We illustrate the bottom-up nature of Dynamic Programming, by attempting solution of the problem of computing binomial coefficients. In order to calculate C(n, k), for given numbers n and k, we consider an n $\times$ k table or matrix of the form shown below.

The (i, j) th entry of the table contains the value C(i, j). We know,

C (i, 0) = 1 for all i = 0, 1, 2,…., n and
C (0, j) = 0 for j = 1, 2, …., k

Thus, initially the table looks like

	0	1	2	3	……..	k
0	1	0	0	0	……..	0
1	1					
2	1					
3	1					
.	.					
.	.					
.	.					
.	.					
n	1					

Then row C (1, j) for j = 1, 2, …. k may be filled up by using the formula

C (i, j) = C (i – 1, j – 1) + C (i – 1, j).

After filling up the entries of the first row, the table takes the following form:

	0	1	2	3	……..	k
0	1	0	0	0		0
1	1	1	0	0		0
2	1					
.	1					
.	.					
.	.					
.	.					
n	1					

From the already calculated values of a given row i, adding successive pairs of consecutive values, we get the values for (i + 1)th row. After completing the entries for row with index 4, the table may appear as follows, where the blank entries to the right of the main diagonal are all zeros.

	0	1	2	3	4	…..	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
.	.						
.	.						
.	.						
.	.						
n	1						

***We Summarize below the process followed above, for calculating C(i, j)*:**

First of all, the simplest values C(i, 0) = 1 for i = 1, 2, …., n and C(0, j) = 0 for $j \geq 1$, are obtained directly from the given formula. Next, more complex values are calculated from the already available less complex values. Obviously, the above mentioned process is a *bottom-up* one.

Though the purpose in the above discussion was to introduce and explain the Dynamic Programming technique, yet we may also consider the complexity of calculating C (n, k) using the tabular method.

Space Complexity/Requirement: In view of the following two facts, it can be easily seen that the amount of space required is not for all the n × k entries but only for k values of any row C (i, j) for j = 1, 2, ….., k independent of the value of i:

(i) The value in column 0 is always 1 and hence need not be stored.

(ii) Initially 0^{th} row is given by C(0,0) = 1 and C (0, j) = 0 for j = 1, 2, …., k. Once any value of row 1, say C(1, j) is calculated the values C(0, j–1) and C(0, j) are no more required and hence C (1, j) may be written in the space currently occupied by C(0, j) and hence no extra space is required to write C (1, j).

In general, when the value C(i, j) of the ith row is calculated the value C (I – 1, j) is no more required and hence the cell currently occupied by C (i –1, j) can be used to store

the value C(i, j). Thus, at any time, one row worth of space is enough to calculate C(n, k). Therefore, space requirement is θ (k).

Time Complexity: If we notice the process of calculating successive values of the table, we find that any value C (i, j) is obtained by adding 1's. For example, C(4, 2) = 6 is obtained by adding C(3, 1) = 3 and C(3, 2) = 3. But then C(3, 1) is obtained by adding C(2, 0) = 1 and C(2, 1) = 2. Again C(2, 1) is obtained by adding C(1, 0) = 1 and C(1, 1) = 1. Similarly, C(3, 2) and hence C(4, 2) can be shown to have been obtained by adding, directly or indirectly, a sequence of 1's. Also, the number of additions for calculating C(n, k) can not be more than all the entries in the (n–1) rows viz., 1, 2, ….. (n–1), each row containing k elements.
Thus, number of additions $\leq$ n k.

Therefore, the time complexity is θ (n k).

In the next sections, we shall discuss solution of some well-known problems, using Dynamic Programming technique.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the dynamic programming technique for solving optimization problems;
- apply the dynamic programming technique for solving optimization problems. specially you should be able to solve the following well-known problems using this technique:
  - Shortest paths problems
  - Knapsack Problem
  - Chained Matrix Multiplication Problem
  - Problem of making change
- Understand the principle of optimality.

1.2 THE PROBLEM OF MAKING CHANGE

First of all, we state a special case of the problem of making change, and then discuss the problem in its general form.

We, in India, have currency notes or coins of denominations of Rupees 1, 2, 5, 10, 20, 50, 100, 500 and 1000. Suppose a person has to pay an amount of Rs. 5896 after having decided to purchase an article. Then, *the **problem** is about how to pay the amount using **minimum** number of coins/notes.*

A simple, frequently and unconsciously used **algorithm** based on Greedy technique is that after having collected an amount A < 5896, choose a note of denomination D, which is s.t

(i) A + D $\leq$ 5896 and

(ii) D is of maximum denomination for which (i) is satisfied, i.e., if E > D then A+E > 5896.

***In general, the Change Problem may be stated as follows*:**

Let $d_1, d_2, \ldots d_k$, with $d_i > 0$ for i = 1, 2, …, k, be the only coins that are available such that each coin with denomination d_i is available in sufficient quantity for the purpose

of making payments. Further, let A, a positive integer, be the amount to be paid using the above-mentioned coins. The problem is to use the *minimum* number of coins, for the purpose.

The problem with above mentioned algorithm based on greedy technique, is that in some cases, it may either *fail* or may yield *suboptimal* solutions. In order to establish *inadequacy* of greedy technique based algorithms, we consider the following two examples.

Example 1.2.1: Let us assume a hypothetical situation in which we have supply of rupee-notes of denominations 5, 3 and 2 and we are to collect an amount of Rs. 9. Then using greedy technique, first we choose a note of Rupees 5. Next, we choose a 3-Rupee note to make a total amount of Rupees 8. But then according to greedy technique, we can not go ahead in the direction of collecting Rupees 9. The failure of greedy technique is because of the fact that there is a solution otherwise, as *it is possible* to make payment of Rupees 9 using notes of denominations of Rupees 5, Rupees 3 and Rupees 2, viz., 9 = 5+2+2.

Example 1.2.2: Next, we consider another example, in which greedy algorithm may yield a solution, but the solution may not be optimal, but only *suboptimal*. For this purpose, we consider a hypothetical situation, in which currency notes of denominations 1, 4 and 6 are available. And, we have to collect an amount of 8 = 6+1+1. But this solution uses **three currency** notes/coins, whereas another solution using only **two currency** notes/coins, viz., 8 = 4+4, is available.

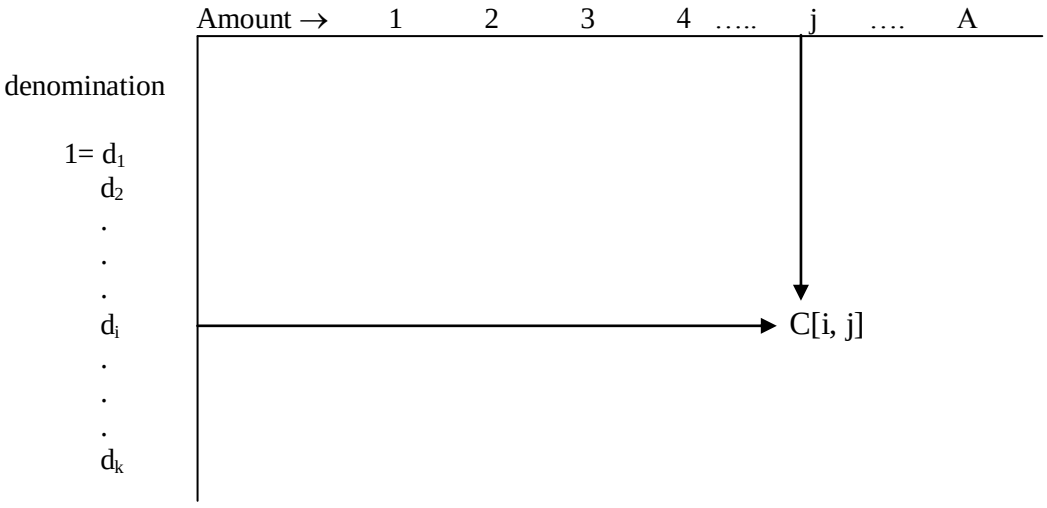
Next, we discuss how the Coin Problem is solved using Dynamic Programming technique.

As mentioned earlier, to solve the coin problem using Dynamic Programming technique, we construct a table, some of the entries in which, corresponding to simple cases of the problem, are filled up *initially* from the definition of the problem. And then recursively entries for the more complex cases, are calculated from the already known ones.

We recall the definition of the coin problem:

Pay an amount A using minimum number of coins of denominations d_i, $1 \le i \le k$. It is assumed coins of each denomination, are available in sufficient quantities.

Each of the denomination d_i, $1 \le i \le k$, is made a row label and each of the value j for $1 \le j \le A$ is made a column label of the proposed table as shown below, where A is the amount to be paid:



In the table given on page no. 9, $0 < d_1 < d_2 < \ldots < d_k$ and **C[i, j]** denotes the minimum number of coins of denominations $d_1, d_2, \ldots, d_i$ (only) that is used to make an amount j, where more than one coin of the same denomination may be used. The value C[i, j] is entered in the table with row label d_i and column label j.

Next, in respect of entries in the table, we make the following two observations:

(i) In order to collect an amount 0, we require zero number of coins, which is true whether we are allowed to choose from, say, one of the successively larger sets of denominations viz., $\{d_1\}, \{d_1, d_2\}, \{d_1, d_2, d_3\}, \ldots, \{d_1, d_2, \ldots, d_k\}$. Thus, entries in the table in the column with 0 as column label, are all 0's.

(ii) If $d_1 \neq 1$, then there may be some amounts j (including j = 1) for which, **even with dynamic programming technique, no solution may exist, i.e., there may not be any number of coins of denominations $d_1, d_2, \ldots d_k$ for which the sum is j.** ***Therefore, we assume $d_1 = 1$.*** **The case $d_1 \neq 1$ may be handled similarly**.

As $d_1 = 1$, therefore, the first row of the table in the jth column, contains j, the number of coins of denomination only $d_1 = 1$ to form value j.

Thus, the table at this stage looks like–

	0	1	2	3	4	…..	j	….	A
d_1	0	1	2	3	4	…	j	…	A
d_2	0								
d_i	0								
.	.								
.	.								
.	.								
d_k	0								

Next for $i \geq 2$ and $j \geq 1$, the value C[i , j], the minimum number of **coins of denominations upto d_i (only)** to sum up to j, can be obtained recursively through either of the following two ways:

(i) **We choose** a coin of denomination d_i the largest, available at this stage; to make up j (provided, of course, $j \geq d_i$). In this case, rest of the amount to be made out of coins of denomination $d_1, d_2, \ldots, d_i$ is $(j - d_i)$. Thus, in this case, we may have–

$$C[i, j] = 1 + C[i, j - d_i], \quad j \geq d_i \text{ if } j \geq d_i \qquad (1.2.1)$$

(ii) **We do not choose** a coin of denomination d_i even when such a coin is available. Then we make up an amount j out of coins of denominations $d_1, d_2, \ldots, d_{i-1}$ (only). Thus, in this case, we may have

$$C[i, j] = C[i-1, j] \quad \text{if } i \geq 1 \qquad (1.2.2)$$

If we choose to fill up the table row-wise, in increasing order of column numbers, we can easily see from (i) and (ii) above that both the values – $C[i, j - d_i]$ and $C[i-1, j]$ are already known for comparison, to find out better alternative for C[i, j].

By definition, C[i, j] = min { 1 + C[i, j – d_i], C [i–1, j]} and can be calculated, as the two involved values viz., C [i, j – d_i] and C [i–1, j] are already known.

Comment 1.2.3

If j < d_i in case (1.1) then the Equation (1.1) is impossible. Mathematically we can say C [i, j – d_i] = ∞ if j < d_i, because, then the case is automatically excluded from consideration for calculating C[i, j].

Similarly we take

C[i–1, j] = ∞ if i < 1

Following the above procedure, C{k, A] gives the desired number.

In order to explain the above method, let us consider the earlier example *for which greedy algorithm gave only suboptimal solution.*

Example 1.2.4: *Using Dynamic Programming technique, find out minimum number of coins required to collect Rupees 8 out of coins of denominations 1, 4, 6.*

From the earlier discussion we already know the following portion of the table to be developed using Dynamic Programming technique.

	0	1	2	3	4	5	6	7	8
d_1 = 1	0	1	2	3	4	5	6	7	8
d_2= 4	0								
d_i= 6	0								

Next, let us **calculate C[2,1],** which by definition, is given by

C[2, 1] = min {1 + c [1, 1– 4], c [1,1]}

By the comment above C [1, – 3] = ∞

∴ C [2, 1] = C [1, 1] = 1

Similarly, we can show that

C[3, 1] = C[2, 1] = 1

Next we consider

C[2, 2] = min [1 + C (2, – 2), C(1, 2)]

Again C[2, – 2] = ∞

Therefore, C[2, 2] = [1, 2] = 2

Similarly, C[3, 2] = 2

On the similar lines, we can show that

C[2, 3] = C[1, 3] = 3

C[3, 3] = C[2, 3] = 3

Next, interesting case is C[2, 4], i.e., to find out minimum number of coins to make an amount 4 out of coins of denominations 1, 4, 6. By definition,

C[2, 4] = min {1 + C (2, 4 – 4), C(1, 4)}
But C[2, 0] = 0 and C [1, 4] = 4, therefore,
= min {1 + 0, 4} = 1 C[2, 4]

By following the method explained through the above example steps, finally, we get the table as

	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	**1**	2	3	4	**2**
$d_i = 6$	0	1	2	3	**1**	2	**1**	2	**2**

Let us formalize the method explained above of computing C[k, A], in the general case, in the form of the following algorithm:

Function Min-Num-Coins (A, D[1..k])

{gives the minimum number of coins to add up to A where the number k of denominations and the value A are supplied by the calling procedure/program}

```
array C [1… k,   0… A]
For i = 1 to k
Read (d [i ] )
{reads the various denominations available with each denomination coin in sufficient
numbers}.
{assuming d1 = 1, initialize the table C {  } as follows}

For i = 1 to k
        C[ i, 0] = 0
For j = 1to A
       C [ 1, j] = j

If i = 1 and j < d [ i ] then
        C [ i, j] = ∞
else
        if
        j < d [ i ] then
        C [ i, j] = C [ i – 1, j]
else

        C[ i, j] = min 1+C {i, j – d  [i] ],    C [i – 1, j]}
return C [ k, A]
```

Comments 1.2.5

Comment 1: The above algorithm *explicitly* gives *only the number* of coins, which are minimum to make a pre-assigned amount A, yet it can also be used to *determine the set of coins* of various denominations that add upto A.

This can be seen through the following argument:

By definition, C [i, j] = either 1 + C [i, j – d_i] or C [i –1, j], which means either we choose a coin of denomination d_i or we do not choose a coin of denomination d_i,

depending upon whether $1 + C [i, j - d_i] \leq C [i - 1, j]$ or not. Applying the above rule recursively for decreasing values of i and j, we know which coins are chosen for making an amount j out of the available coins.

Comment 2: Once an algorithm is designed, it is important to know its computational complexity in order to determine its utility.

In order to determine C[k, A], the above algorithm computes $k \times (A + 1)$ values through additions and comparisons. Therefore, the complexity of the algorithm is $\theta (k . A)$, the order of the size of the matrix C[k, A].

1.3 THE PRINCIPLE OF OPTIMALITY

The Principle of Optimality states that components of a globally optimum solution must themselves be optimal. Or, Optimality Principle states subsolutions of an optimal solution must themselves be optimal.

While using the Dynamic Programming technique in solving the coin problem we tacitly assumed the above mentioned principle in defining C [i, j], as minimum of $1 + C [i, j - d_i]$ and $C [i - 1, j]$. The principle is used so frequently and without our being aware of having used it.

However, we must be aware that there may be situations in which the principle may not be applicable. The principle of optimality may **not** be true, specially, in situations where resources used over the (sub) components may be more than total resources used over the whole, and where total resources are limited and fixed. A simple example may explain the point. For example, suppose a 400-meter race champion takes 42 seconds to complete the race. However, covering 100 meters in 10.5 seconds may not be the best/optimal solution for 100 meters. The champion may take less than 10 seconds to cover 100 meters. The reason being total resources (here the concentration and energy of the athlete) are limited and fixed whether distributed over 100 meters or 400 meters.

Similarly, for a vehicle with best performance over 100 miles, can not be thought of in terms of 10 times best performance over 10 miles. First of all, fuel usage after some lower threshold, increases with speed. Therefore, as the distance to be covered increases (e.g., from 10 to 100 miles) then fuel has to be used more cautiously, restraining the speed, as compared to when distance to be covered is less (e.g., 10 miles). Even if refuelling is allowed, refuelling also takes time. The drivers concentration and energy are other fixed and limited resources; which in the case of shorter distance can be used more liberally as compared to over longer distances, and in the process produce better speed over short distances as compared to over long distances. The above discussion is for the purpose of driving attention to the fact that principle of optimality is ***not*** universal, specially when the resources are limited and fixed. Further, it is to draw attention that Dynamic Programming technique assumes validity of Principle of Optimality for the problem domain. Hence, while applying Dynamic Programming technique for solving optimisation problems, in order for the validity of the solution based on the technique, we need to ensure that the Optimality Principle is valid for the problem domain.

Ex. 1) Using Dynamic Programming, solve the following problem (well known as **Knapsack Problem**). Also write the algorithm that solves the problem.

We are given *n objects* and a knapsack. For i = 1, 2, …, n, object i has a positive *weight* w_i and a positive *value* v_i. The knapsack can carry a *weight not exceeding W*. Our aim is to fill the knapsack in a way that maximizes the value of the objects included in the knapsack. Further, it is assumed that the objects *may **not** be broken*

into pieces[*]. In other words, either a whole object is to be included or it has to be excluded.

1.4 CHAINED MATRIX MULTIPLICATION

In respect of multiplication of matrices, we recall the following facts:

(i) Matrix multiplication is a binary operation, i.e., at one time only two matrices can be multiplied.

(ii) However, not every given pair of matrices may be multiplicable. If we have to find out the product M_1M_2 then the orders of the matrices must be of the from $m \times n$ and $n \times p$ for some positive integers m, n and p. Then matrices M_1 and M_2, in this order, are said to be multiplication – compatible. Number of scalar (number) multiplications required is mnp.

As can be easily seen that matrix multiplication *is not commutative*. Even $M_2 M_1$ may not be defined even if $M_1 M_2$ is defined.

(iii) Matrix multiplication is associative in the sense that if M_1 M_2 and M_3 are three matrices of order $m \times n$, $n \times p$ and $p \times q$ then the matrices

$(M_1 M_2) M_3$ and $M_1 (M_2 M_3)$ are defined,
$(M_1 M_2) M_3 = M_1 (M_2 M_3)$
and the product is an $m \times n$ matrix.

(iv) Though, for three or more matrices, matrix multiplication is associative, yet the number of scalar multiplications may very significantly depending upon how we pair the matrices and their product matrices to get the final product.

For example, If A is 14×6 matrix, B is 6×90 matrix, and C is 90×4 matrix, then the number of scalar multiplications required for (AB)C is

	$14 \times 6 \times 90 = 7560$	(*for (AB) which of order 14×90*)
Plus	$14 \times 90 \times 4 = 5040$	(*for product of (AB) with C*)

equal to 12600 scalar multiplication.

On the other hand, number of scalar multiplications for A(BC) is

	$6 \times 90 \times 4 = 2160$	(*for (BC) which is of order 6×4*)
Plus	$14 \times 6 \times 4 = 336$	(*for product of with BC*)

equal to 2496 scalar multiplication

Summering: The product of matrices A (14×6), B(6×90) and C(90×4) takes **12600** scalar operators when first the product of A and B is computed and then product AB is multiplied with C. On the other hand, if the product BC is calculated first and then product of A with matrix BC is taken then **only 2496** scalar multiplications are required. The later number is around 20% of the former. In case when large number of matrices are to be multiplied for which the product is defined, proper parenthesizing through pairing of matrices, may cause dramatic saving in number of scalar operations.

[*] Another version allows any fraction x_i with $0 \le x_i \le 1$. However, in this problem, we assume either $x_i = 1$ or $x_i = 0$.

This raises the question of how to parenthesize the pairs of matrices within the expression $A_1A_2 \ldots A_n$, a product of n matrices which is defined; so as to optimize the computation of the product $A_1A_2 \ldots A_n$. The product is known as Chained Matrix Multiplication.

Brute-Force Method: One way for finding the optimal method (i.e., the method which uses minimum number of scalar (numerical) operations) is to parenthesize the expression $A_1A_2 \ldots A_n$ in all possible ways and calculate number of scalar multiplications required for each way. Then choose the way which requires minimum number of scalar multiplications.

However, if **T(n)** denotes the number of ways of putting parentheses for pairing the expression $A_1A_2 \ldots A_n$, T(n) is an exponentially increasing function. The rate at which values of T(n) increase may be seen from the following table of values of T(n) for n = 1, 2, …….

n :	1	2	3	4	5	…	10	…	15
T(n):	1	1	2	5	14	…	4862	…	2674440

These values of T(n) are called *Catalan numbers*.

Hence, it is almost impossible to use the method for determining how to optimize the computation of the product $A_1A_2 \ldots A_n$.

Next, we discuss how to optimize the computation using Dynamic Programming Technique.

1.5 MATRIX MULTIPLICATION USING DYNAMIC PROGRAMMING

It can be seen that if one arrangement is optimal for $A_1A_2 \ldots A_n$ then it will be optimal for any pairings of $(A_1 \ldots A_k)$ and $(A_{k+1}A_n)$. Because, if there were a better pairing for say $A_1A_2 \ldots A_k$, then we can replace the better pair $A_1A_2 \ldots A_k$ in $A_1A_2 \ldots A_k\ A_{k+1} \ldots A_n$ to get a pairing better than the initially assumed optimal pairing, leading to a contradiction. Hence the principle of optimality is satisfied.

Thus, the Dynamic Programming technique can be applied to the problem, and is discussed below:

Let us first define the problem. Let A_i, $1 \le i \le n$, be a $d_{i-1} \times d_i$ matrix. Let the vector d [0.. n] stores the dimensions of the matrices, where the dimension of A_i is $d_{i-1} \times d_i$ for i = 1, 2, …, n. By definition, any subsequence $A_j \ldots A_k$ of $A_1A_2 \ldots A_n$ for $1 \le j \le k \le n$ is a well-defined product of matrices. Let us consider a table m [1.. n, 1 .. n] in which the entries m_{ij} for $1 \le i \le j \le n$, represent **optimal (i.e., minimum)** number of operations required to compute the product matrix $(A_i \ldots A_j)$.

We fill up the table diagonal-wise, i.e., in one iteration we fill-up the table one diagonal $m_{i, i+s}$, at a time, for some constant $s \ge 0$. Initially we consider the biggest diagonal m_{ii} for which **s = 0**. Then next the diagonal $m_{i, i+s}$ for **s = 1** and so on.

First, *filling up the entries m_{ii}, i = 1, 2, …, n*.

Now m_{ii} stands for the minimum scalar multiplications required to compute the product of single matrix A_i. But number of scalar multiplications required are zero.

Hence,

$$m_{ii} = 0 \qquad \text{for } i = 1, 2, \ldots n.$$

Filling up entries for $m_{i(i+1)}$ for i = 1, 2, …, (n – 1).

$m_{i(i+1)}$ denotes the minimum number of scalar multiplication required to find the product $A_i A_{i+1}$. As A_i is $d_{i-1} \times d_i$ matrix and A_{i+1} is $d_i \times d_{i+1}$ matrix. Hence, there is a unique number for scalar multiplication for computing $A_i A_{i+1}$ giving

$$m_{i,(i+1)} = d_{i-1} d_i d_{i+1}$$

for i = 1, 2, …, (n – 1).

The above case is also subsumed by the general case

$m_{i(i+s)}$ for $s \geq 1$

For the expression

$A_i A_{i+1} \ldots A_{i+s}$,

let us consider top-level pairing

$(A_i A_{i+1} \ldots A_j)(A_{J+1} \ldots A_{i+s})$

for some k with $i \leq j \leq i + s$.

Assuming optimal number of scalar multiplication viz., $m_{i,j}$ and $m_{i+1,j}$ are already known, we can say that

$m_{i(i+s)} = \min_{i \leq j \leq i+s} (m_{i,j} + m_{j+1,s} + d_{i-1} d_j d_{i+s})$

for i = 1, 2, …, n – s.

When the term $d_{i-1} d_j d_{i+s}$ represents the number of scalar multiplications required to multiply the resultant matrices $(A_i \ldots A_j)$ and $(A_{j+1} \ldots A_{i+s})$

Summing up the discussion, we come the definition $m_{i,i+s}$ for i=1,2, …(n –1) as

for s = 0: $m_{i,i} = 0$ for i = 1, 2, …, n

for s = 1: $m_{i,i+1} = d_{i-1} d_i d_{i+1}$ for i = 1, 2, …, (n–1)

for $1 < s < n$:

$m_{i,i+s} = \min_{i \leq j \leq i+s} (m_{ij} + m_{j+1,i+s} + d_{i-1} d_i d_{i+1})$ for i= 1, 2, …, (n–s)

Then $m_{1,n}$ is the final answer

Let us illustrate the algorithm to compute $m_{j+1,i+s}$ *discussed above through an example*

Let the given matrices be

A1	of order	14×6
A2	of order	6×90
A3	of order	90×4
A4	of order	4×35

Thus the dimension of vector d [0..4] is given by [14, 6, 90, 4, 35]

For s = 0, we know $m_{ii} = 0$. Thus we have the Matrix

Next, consider for s = 1, the entries

$m_{i,i+1} = d_{i-1} d_i d_{i+1}$

$m_{12} = d_0\ d_1\ d_2 = 14 \times 6 \times 90 = 7560$
$m_{23} = d_1\ d_2\ d_3 = 6 \times 90 \times 4 = 3240$
$m_{34} = d_2\ d_3\ d_4 = 9 \times 4 \times 35 = 1260$

	1	2	3	4
1	0	7560		
2		0	3240	
3			0	1260
4				0

Next, consider for s = 2, the entries

m_{13} = min $(m_{11} + m_{23} + 14 \times 6 \times 4,\ m_{12} + m_{33} + 14 \times 90 \times 4)$
= min $(0 + 3240 + 336,\ 7560 + 0 + 5040)$
= 3576

m_{24} = min $(m_{22} + m_{34} + 6 \times 90 \times 35,\ m_{23} + m_{44} + 6 \times 4 \times 35)$
= min $(0 + 1260 + 18900,\ 3240 + 0 + 840)$
= 4080

Finally, for s = 3

m_{14} = min $(m_{11} + m_{24} + 14 \times 6 \times 35,$ {when k = 1}
$M_{12} + m_{34} + 14 \times 90 \times 35,$ {when k = 2}
$M_{13} + m_{44} + 14 \times 4 \times 35)$ {when k = 3}

= min $(0 + 4080 + 2090,\ 7560 + 3240 + 44100,\ 3576 + 0 + 1960)$
= 5536

Hence, the optimal number scalar multiplication, is 5536.

1.6 SUMMARY

(1) The Dynamic Programming is a technique for solving optimization Problems, using bottom-up approach. The underlying idea of dynamic programming is to avoid calculating the same thing twice, usually by keeping a table of known results, that fills up as substances of the problem under consideration, are solved.

(2) In order that Dynamic Programming technique is applicable in solving an optimisation problem, it is necessary that the principle of optimality is applicable to the problem domain.

(3) The principle of optimality states that for an optimal sequence of decisions or choices, each subsequence of decisions/choices must also be optimal.

(4) **The Chain Matrix Multiplication Problem:** The problem of how to parenthesize the pairs of matrices within the expression $A_1A_2 \ldots A_n$, a product of n matrices which is defined; so as to minimize the number of scalar multiplications in the computation of the product $A_1A_2 \ldots A_n$. The product is known as Chained Matrix Multiplication.

(5) **The Knapsack Problem:** We are given *n objects* and a knapsack. For i = 1, 2, …, n, object i has a positive *weight* w_i and a positive *value* v_i. The knapsack can carry a *weight not exceeding W*. The problem requires that the knapsack is filled in a way that maximizes the value of the objects included in the knapsack.

Further, a special case of knapsack problem may be obtained in which the objects *may **not** be broken into pieces*[*]. In other words, either a whole object is to be included or it has to be excluded.

1.7 SOLUTIONS/ANSWERS

Ex. 1)

First of all, it can be easily verified that Principle of Optimality is valid in the sense for an optimal solution of the overall problem each subsolution is also optimal. Because in this case, a non-optimal solution of a subproblem, when replaced by a better solution of the subproblem, would lead to a better than optimal solution of the overall problem. A contradiction.

As usual, for solving an optimization problem using Dynamic Programming technique, we set up a table V[1..n, 0..W].

In order to label the rows we first of all, order the given objects according to increasing relative values R = v/w.

Thus first object O_1 is the one with minimum relative value R_1. The object O_2 is the one with next least relative value R_2 and so on. The last object, in the sequence, is O_n, with maximum relative weight R_n.

The **ith row** of the Table Corresponds to object O_i having ith relative value, when values are arranged in increasing order. The jth column corresponds to weight j for $0 \leq j \leq W$. The entry **Knap [i, j]** denotes the maximum value that can be packed in knapsack when objects $O_1, O_2, \ldots, O_i$ *only* are used and the included objects have weight at most j.

Next, in order to fill up the entries Knap[i, j], $1 \leq i \leq n$ and $0 \leq j \leq W$, of the table, we can check, as was done in the coin problem that,

(i) Knap [i, 0] = 0 for i = 1, 2, …, n

(ii) Knap [1, j] = V, for j = 1, …, W
where V is the value of O_1

[*] Another version allows any fraction x_i with $0 \leq x_i \leq 1$. However, in this problem, we assume either $x_i = 1$ or $x_i = 0$.

	0	1	2	3	4	…	j	…	W
1	0	V	V	V	V		V		V
2	0								
.	.								
.	.								
.	.								
j	0								
.	.								
.	.								
.	.								
n	0								

Where V is the value of the object O_1.

Further, when calculating Knap [i, j], there are two possibilities: either

(i) The ith object O_i is taken into consideration, then
Knap [i, j] = Knap [i–1, j – w_i] + v_i or

(ii) The ith object O_i is not taken into consideration then
Knap [i, j] = V[i–1, j]

Thus we define

Knap [i, j] = max {Knap [i–1, j], Knap [i–1, j - w_i] + v_i}

The above equation is valid for i = 2 and j ≥ w_i. In order that the above equation may be applicable otherwise also, without violating the intended meaning we take,

(i) Knap [O, j] = O for j ≥ O and
(ii) Knap [i, j] = – ∞ for k < 0

We explain the Dynamic Programming based solution suggested above, through the following example.

We are given six objects, whose weights are respectively 1, 2, 5, 6, 7, 10 units and whose values respectively are 1, 6, 18, 22, 28, 43. Thus relative values in increasing order are respectively 1.00, 3.00, 3.60, 3.67, 4.00 and 4.30. **If we can carry a maximum weight of 12, then the table below shows that we can compose a load whose value is 49.**

Weights	0	1	2	3	4	5	6	7	8	9	10	11	12
Relative Values													
w_1= 1, v_1 = 1, R_1 = 1.00	0	1	1	1	1	1	1	1	1	1	1	1	1
w_2 = 2, v_2 = 6, R_2 = 3.00	0	1	6	7	7	7	7	7	7	7	7	7	7
w_3 = 5, v_3 = 18, R_3 = 3.60	0	1	6	7	7	18	19	24	25	25	25	25	25
W_4 = 6, v_4 = 22, R_4 =3.67	0	1	6	7	7	18	22	24	28	29	29	40	41
W_5 = 7, v_5 = 28 R_5 = 4.00	0	1	6	7	7	18	22	28	29	34	35	40	46
W_6 = 10, v_6 = 43, R_6 = 4.30	0	1	6	7	7	18	22	28	29	34	43	44	49

Algorithm for the solution of the solution explained above of the Knapsack Problem.

Function Knapsack (W, Weight [1..n], value [1..n])

{returns the maximum value corresponding to maximum allowed weight W, where we are given n objects O_i, $1 \le i \le n$, with weights Weight [i] and values Value [i]}
array R [1 ..n], Knap [1..n, 0..W]
For i = 1 to n do

begin
read (Weight [i])
read (value [i])
R [i] / weight [i]
end

For j = 1 to n do

{Find k such that R[k] is minimum of
R[j], R[j+1], …R[n]}

Begin
k = j
For t = j + 1 to n do
If R [t] < R [k] then
k = t
Exchange (R [j], R [k]);
Exchange (Weight [j], Weight [k]);
Exchange (Value [j], value [k]);
end

{At this stage R[1.. n] is a sorted array in increasing order and Weight [j] and value [j] are respectively weight and value for jth least relative value}
{Next, we complete the table knap for the problem}

For i = 1 to n do
Knap [i , 0] = 0 for i = 1, …, n
Knap [1, j] = value [1] for j = 1, …, W

{Value [1] is the value of the object with minimum relative value}

{Next values for out of the range of the table Knap}

If $i \le 0$ and $j \ge 0$ then
Knap [i , j] = 0
Else if $j < 0$ then
Knap [i, j] = $-\infty$

For i = 2 to n do
For j = 1 to W do
Knap [i, j] = maximum {Knap [i –1, j], Knap [i – 1, j – Weight [i]]
+ value [i]

Retu rn Knap [n, W]

1.8 FURTHER READINGS

1. *Foundations of Algorithms,* R. Neapolitan & K. Naimipour: (D.C. Health & Company, 1996).
2. *Algoritmics: The Spirit of Computing,* D. Harel, (Addison-Wesley Publishing Company, 1987).
3. *Fundamental Algorithms* (*Second Edition*), D.E. Knuth, (Narosa Publishing House).
4. *Fundamentals of Algorithmics,* G. Brassard & P. Brately, (Prentice-Hall International, 1996).
5. *Fundamentals of Computer Algorithms,* E. Horowitz & S. Sahni, (Galgotia Publications).
6. *The Design and Analysis of Algorithms,* Anany Levitin, (Pearson Education, 2003).
7. *Programming Languages* (*Second Edition*) – *Concepts and Constructs,* Ravi Sethi, (Pearson Education, Asia, 1996).