APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS

KTU ASSIST

a complete app for ktu students

Get it on Google Play

www.ktuassist.in

Module 4

Divide and Conquer Method & Dynamic Programming

DIVIDE AND CONQUER

Ques 1) Discuss the divide and conquer approach in detail.

Ans: Divide and Conquer Approach

Divide-and-Conquer Approach is a **top-down technique** for designing algorithms that consists of dividing the problem into smaller sub-problems hoping that the solutions of the sub-problems are easier to find and then composing the partial solutions into the solution of the original problem.

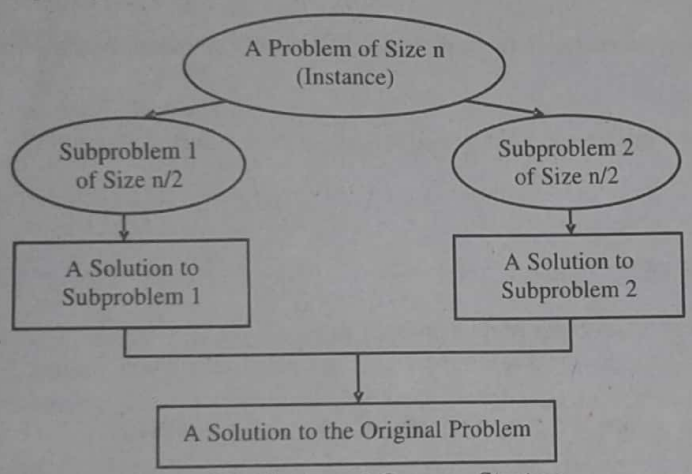


Figure 4.1: Divide-and-Conquer Strategy

Phases of Divide and Conquer

Divide-and-conquer algorithm consists of following major phases:

1) **Divide Phase:** In this phase of the algorithm, the input is partitioned into $p \geq 1$ parts, each of size strictly less than n, the size of the original instance. The most common value of p is 2, although other small constants greater than 2 are not uncommon.

2) **Conquer Phase:** This phase consists of performing p recursive call(s) if the problem size is greater than some pre-defined threshold n_0. This threshold is derived by mathematical analysis of the algorithm. Once it is found, it can be increased by any constant amount without affecting the time complexity of the algorithm.

3) **Combine Phase:** In this phase, the solutions to the p recursive call(s) are combined to obtain the desired output. The combine step in a divide-and-conquer algorithm may consist of merging, sorting, searching, finding the maximum or minimum, matrix addition, etc.

Examples of Divide and Conquer

1) Merge Sort
2) Quick Sort,
3) Binary Search,
4) Binary Tree Traversals,
5) Multiplication of Large Integers,
6) **Matrix Multiplication:** Strassen's Algorithm

Ques 2) Write and explain the control abstraction for divide- and conquer method.

Or

Give the control abstraction of divide-and-conquer.

Ans: Control Abstraction for Divide and Conquer

The general direction in which a particular algorithm proceeds is specified by the parent abstract algorithm. The control it specifies is called **control abstraction**.

A control abstraction is a procedure that reflects the way an actual program based on divide and conquers. A control abstraction shows clearly the flow of control but the primary operations are specified by other procedures. The control abstraction can be written either iteratively or recursively.

If we are given a problem with 'n' inputs and if it is possible for splitting the 'n' inputs into 'k' subsets where each subset represents a sub problem similar to the main problem then it can be achieved by using divide and conquer strategy.

If the sub problems are relatively large then divide and conquer strategy is reapplied. The sub problem resulting from divide and conquer design are of the same type as the original problem. Generally divide and conquer problem is expressed using recursive formulas and functions.

Algorithm: A general divide and conquer design strategy (control abstraction) is illustrated below:

```
DAndC (P)
{
    if small(P) then
        return S(P)  //termination condition
```

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

else
{Divide P into smaller instances $P_1, P_2, P_3 \ldots P_k$ $k \geq 1$; or $1 \leq k \leq n$
Apply DAndC to each of these sub problems.
Return Combine (DAndC(P_1), DAndC (P_2), DAndC (P_3)...
DAndC (P_k) }}

The above blocks of code represents a control abstraction for divide and conquer strategy. Small (P) is a Boolean valued function that determines whether the input size is small enough that the answer can be computed without splitting. If small (P) is true then function 'S' is invoked. Otherwise the problem 'P' is divided into sub problems. These sub problems are solved by recursive application of Divide-and-conquer. Finally the solution from k sub problems is combined to obtain the solution of the given problem.

If the size of 'P' is 'n' and if the size of 'k' sub problems is $n_1, n_2, \ldots n_k$ respectively then the computing time of DAndC is described by the recurrence relation.

$T(n) = g(n)$ when n is small
$= T(n_1) + T(n_1) + T(n_2) + \ldots\ldots + T(n_k) + f(n)$ Otherwise.

Here,
1) T(n) denotes the time for DAndC on any input of size 'n'.
2) g(n) is the time to compute the answer directly for small inputs.
3) g(n) is the time for dividing 'P' and combining the solutions of sub problems.

Ques 3) Give the analysis of the complexity of the divide and conquer strategy.

Ans: To analyse the running time of a divide and conquer algorithms we make use of a recurrence relation. A function T(n) denotes the running time of the algorithm on an input of size 'n' and characterize T(n) using an equation that relates T(n) to values of the function T for problem sizes smaller than n.

The recurrence relations are of the form:

$T(n) = T(1)$ $n=1$
$= aT(n/b)+f(n)$ $n>1$

Where a, b are known constants. We assume that T(1) is known and n is a power of b (i.e., $n = b^k$).

For example, let consider the case in which a = 2 and b = 2. Let T (1) = 2 and f(n) = n. then we have

$T(n) = 2T(n/2) + n$
$= 2[2T(n/4) + n/2] + n$
$= 4T(n/4) + 2n$
$= 4[2T(n/8) + n/4] + 2n$
$= 8T(n/8) + 3n$
$= \ldots$
$= 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$.
$= 2^{\log_2 n} T(n/\log_2 n) + n\log_2 n$,

corresponding to the choice of $i = \log_2 n$. Thus

$T(n) = n\,T(1) + n\log_2 n = n\log_2 n + 2n$

Ques 4) What is two-way merge sort? Write the algorithm for merge sort.

Ans: Merge Sort

Two way merging deals with the merging of two ordered lists.

The operation of sorting is closely related to the process of merging. This sorting method uses merging of two ordered arrays, which can be combined to produce a single sorted array. This process can be accomplished easily by successively selecting the record with the smallest key occurring in either of the arrays and placing this record in a new table, thereby creating an ordered array. Merge sort is one of the divide and conquer class of algorithm.

The basic idea is:
1) Divide the array to a number of sub-arrays.
2) Sort each of these sub-arrays.
3) Merge them to get a single sorted array.

Two-way merge sort divides the list into two sorted sub-arrays and then merges them to get the sorted array, also called concatenate sort.

Multiple merging can also be accomplished by performing a simple merge repeatedly.

For example, if we have 16 arrays to merge, we can first merge them in pairs. The result of this first step yields eight tables, which are again merged in pairs to give four tables. This process is repeated until a single table is obtained. In this example, four separate passes are required to yield a single list. In general, k separate passes are required to merge 2k separate lists into a single list.

Algorithm: Merge Sort

Sorts the array using Merge sort.

MERGE_SORT (n, list _a, m, list_b, result)

Where,

n → Represents number of elements in first list(list_a)

list_a → Represents the list of elements (first list)

m → Represents number of elements in the second list(list_b)

list_b → Represents the list of elements(second list)

Step 1: [initialize]
i = 0
j = 0
k = 0

Step 2: Repeat step 3 while((i < n) and (j < m))

Step3: if(list_a[i]<list_b[j])
i) result_list[k] = list_a[i]
ii) i = i + 1
iii) k = k+1
else if(list_a[i]>list_b[j])
iv) result_list[k] = list_b[j]
v) j = j+1
vi) k = k + 1
else[Means elements of both the list are equal and insert only one of them into the result_list]

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

vii) result_list[k] = list_a[i]
viii) i = i + 1
ix) j = j + 1
x) k = k + 1

Step4: if(i<n) [size of list_a is larger than the list_b]

Step 5: Repeat through Step 5 for 1 = i, i + 1, i + 2 ... n–1
i) result_list[k] = list_a[i]
ii) i = i + 1
iii) k = k+l

Step 6: [Size of list_b is larger than list)a]
else if(j<m)

Step 7: Repeat through Step 7 for l = j, j + l, j + 2 ... m – 1
i) result_list[k] = list[j]
ii) j = j + 1
iii) k = k+l

Step 8: Return

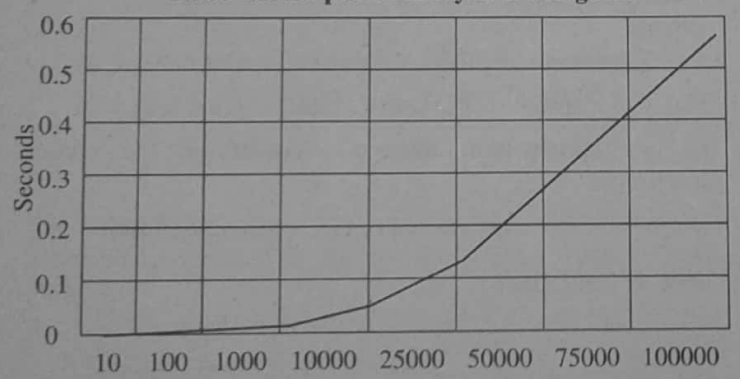
Ques 5) What is the efficiency of merge sort?

Ans: Efficiency/Complexity of Merge Sort
The while loop is iterated at the most (N – L + 1) times. The if statement moves at most one record per iteration (considering sorting of records), the total time is therefore has O(N – L + 1). If records are of length M then this time is O(M(N – L + 1)).

Analysis of routine that uses procedure **merges**.

For an array of size N, on the ith pass the files being merged are of size 2^{i-1}. Consequently, a total of $\log_2(N)$ passes are made over the data. Since two files can be merged in linear time (algorithm **merge**), each pass of merge sort takes O(N) time. As there are $[\log_2(N)]$ passes, the total computing time is $O(N * \log_2 N)$.

Table 4.1: Empirical Analysis of Merge Sort



This additional memory requirement makes it unattractive for most purposes.

Ques 6) Consider the following array with 7 elements as:

33, 26, 35, 29, 18, 10, 24

Sort the above elements using two-way merge sort.

Ans: The first step of merge sort is to divide the array into two sub-arrays. Thus we divide the array into following way:

33, 26, 35, 29 and 18, 10, 24

and first consider the left sub-array.

It is again divided into two sub-arrays as:

33, 26 and 35, 29

For each of these sub-arrays, again apply the same method, dividing each of these into sub arrays of one element each. Sub-arrays of size one, of course, require no sorting.

Finally, start merging the sub-arrays to obtain a sorted array. The sub-arrays 33 and 26 merge to give the sorted array 26, 33 and the sub-arrays 35 and 29 merge to give sorted array 29, 35.

At the next step, merge these two-sorted sub-arrays of size two to obtain a sorted array of size four as:

26, 29, 33, 35

Now the left half of the given array is sorted do the same steps on the right half. First, divide it into two sub-arrays as:

18, 10 and 24

The first of these is divided into two sub-arrays of size one, which are merged to give 10, 18. The second sub-array, 24, has size one, so needs no sorting. Now these are merged to give the sorted array:

10, 18, 24

Finally, the sorted sub-arrays of size four and three are merged to give:

10, 18, 24, 26, 29, 33, 35

Ques 7) Discuss about the Strassen's algorithm for matrix multiplication. Also analyse the algorithm.

Ans: Strassen's Algorithm for Matrix Multiplication
Strassen's algorithm is based on a recursive divide and conquer scheme. Given n × n matrices A and B, calculate C=AB.

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{12} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

It is assumed that each block is square. This can be achieved by padding the matrices. Strassen showed how C can be computed using only 7 block multiplications and 18 blocks adds:

$P_1 = (A_{11} + A_{22}) (B_{11} + B_{22})$
$P_2 = (A_{21} + A_{22}) * B_{11}$
$P_3 = A_{11} * (B_{12} - B_{22})$
$P_4 = A_{22} * (B_{21} - B_{11})$
$P_5 = (A_{11} + A_{12}) * B_{22}$
$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$
$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$
$C_{11} = P_1 + P_4 - P_5 + P_7$
$C_{12} = P_3 + P_5$
$C_{21} = P_2 + P_4$
$C_{22} = P_1 + P_3 - P_2 + P_6$

DOWNLOADED FROM KTUASSIST.IN
Scanned by CamScanner

Algorithm of Strassen Multiplication

Suppose $n = 2^q$ and that $A \in IR^{n \times n}$ and $B \in IR^{n \times n}$. If $n_{min} = 2^d$ with $d \le q$, then this algorithm computes $C = AB$ by applying Strassen procedure recursively $q - d$ times.

Function: C = strass (A, B, n, n_{min})

Step 1: if $n \le n_{min}$

Step 2: C = AB

Step 3: else

Step 4: m = n/2; u = 1: m; v = m + 1: n;

Step 5: P_1 = strass (A(u, u) + A(v, v), B(u, u) + B(v, v), m, n_{min})

Step 6: P_2 = strass (A(v, u) + A(v, v), B(u, u), m, n_{min})

Step 7: P_3 = strass (A(u, u,), B(u, v) – B(v, v), m, n_{min})

Step 8: P_4 = strass (A(v, v), B(v, u) – B(u, u), m, n_{min})

Step 9: P_5 = strass (A(u, u) + A(u, v), B(v, v), m, n_{min})

Step 10: P_6 = strass (A(v, u) – A(u, u), B(u, u) + B(u, v), m, n_{min})

Step 11: P_7 = strass (A(u, v) – A(v, v), B(v, u) + B(v, v), m, n_{min}

Step 12: $C(u, u) = P_1 + P_4 - P_5 + P_7$

Step 13: $C(u, v) = P_3 + P_5$

Step 14: $C(v, u) = P_2 + P_4$

Step 15: $C(v, v) = P_1 + P_3 - P_2 + P_6$

Analysis of Strassen's Algorithm for Matrix Multiplication

The amount of arithmetic associated with strass function is a complicated function of n and n_{min}. If $n_{min} >> 1$, then it suffices to count multiplications as the number of additions is roughly the same. If one just count the multiplications, then it suffices to examine the deepest level of the recursion as that is where all the multiplications occur.

In strass function, there are $q - d$ subdivisions and thus, 7^{q-d} conventional matrix-matrix multiplications to perform. These multiplications have size n_{min} and thus strass function involves about $s = (2^d)^3 7^{q-d}$ multiplications compared to $c = (2^q)^3$, the number of multiplications in the conventional approach.

$$\frac{s}{c} = \left(\frac{2^d}{2^q}\right)^3 7^{q-d} = \left(\frac{7}{8}\right)^{q-d}$$

If d = 0, i.e., we recur on down to the 1-by-1 level, then

$$s = \left(\frac{7}{8}\right)^q c = 7^q = n^{\log_2 7} \approx n^{2.807}$$

Thus, asymptotically, the number of multiplications in the Strassen procedure is $O(n^{2.807})$. However, the number of additions (relative to the number of multiplications) becomes significant as n_{min} gets small.

Ques 8) Let consider $A = \begin{bmatrix} 1 & 2 \\ 0 & 2 \end{bmatrix}$ and $B = \begin{bmatrix} 0 & 1 \\ 3 & 2 \end{bmatrix}$. Then, **Find AB using Strassen's Algorithm for Matrix Multiplication.**

Ans: Here,

$A_{11} = 1, A_{12} = 2, A_{21} = 0, A_{22} = 2$

$B_{11} = 0, B_{12} = 1, B_{21} = 3, B_{22} = 2$

Hence,

$P_1 := (A_{11} + A_{22})(B_{11} + B_{22}) = (1 + 2)(0 + 2) = 6$

$P_2 := (A_{21} + A_{22}) B_{11} = (0 + 2) \times 0 = 0$

$P_3 := A_{11}(B_{12} - B_{22}) = 1 \times (1 - 2) = -1$

$P_4 := A_{22}(B_{21} - B_{11}) = 2 \times (3 - 0) = 6$

$P_5 := (A_{11} + A_{12}) B_{22} = (1 + 2) \times 2 = 6$

$P_6 := (A_{21} - A_{11})(B_{11} + B_{12}) = (0 - 1)(0 + 1) = -1$

$P_7 := (A_{12} - A_{22})(B_{21} + B_{22}) = (2 - 2)(3 + 2) = 0$

$C_{11} := P_1 + P_4 - P_5 + P_7 = 6 + 6 - 6 + 0 = 6$

$C_{12} = P_3 + P_5 = -1 + 6 = 5$

$C_{21} = P_2 + P_4 = 0 + 6 = 6$

$C_{22} = P_1 - P_2 + P_3 + P_6 = 6 - 0 - 1 - 1 = 4$

Hence $AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 6 & 5 \\ 6 & 4 \end{bmatrix}$

DYNAMIC PROGRAMMING

Ques 9) What is dynamic programming? What are the different approaches of dynamic programming?

Or

What is the principle of optimality?

Ans: Dynamic Programming

Dynamic programming is a problem solving technique that like Divide and Conquer solves problems by dividing them into sub-problems.

Dynamic programming is used when the sub-problems are not independent e.g. when they share the same sub-problems.

Dynamic programming is a method of solving complex problems by breaking them down into simpler steps. It is applicable to problems that exhibit the properties of **overlapping subproblems** and **optimal substructure.** When applicable, the method takes much less time than naive methods.

Dynamic Programming is a technique for computing recurrence relations efficiently by sorting partial results. It is typically applied to optimization problems: In such problems there can be many possible solutions.

Each solution has a value, and wish to find a solution with the optimal (minimum or maximum) value. Such a solution is called an **optimal solution** to the problem, since there may be several solutions that achieve the optimal value.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

Principle of Optimality

To use dynamic programming, the problem must observe the principle of optimality, that whatever the initial state is, remaining decisions must be optimal with regard the state following from the first decision. Combinatorial problems may have this property but may use too much memory/time to be efficient

Suppose that in solving a problem, we have to make a sequence of decisions $D_1, D_2, \ldots, D_n$. If this sequence is optimal, then the last k decisions, $1< k< n$ must be optimal.

For example, let consider the shortest path problem if $i, i_1, i_2, \ldots, j$ is a shortest path from i to j, then $i_1, i_2, \ldots, j$ must be a shortest path from i 1 to j

Approaches in Dynamic Programming

There are two approaches of solving dynamic programming problems:

1) **Bottom-up Approach:** Bottom-up approach simply means storing the results of certain calculations, which are then re-used later because the same calculation is a sub-problem in a larger calculation.

2) **Top-down Approach:** Top-down approach involves formulating a complex calculation as a recursive series of simpler calculations.

Ques 10) Write the steps of dynamic programming approach.

Ans: Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps:

Step 1) Characterise the structure of an optimal solution.

Step 2) Recursively define the value of an optimal solution.

Step 3) Compute the value of an optimal solution, typically in a bottom-up fashion.

Step 4) Construct an optimal solution from the computed information.

Ques 11) Explain the control abstraction of dynamic programming?

Ans: Control Abstraction of Dynamic Programming

A control abstraction we mean a procedure whose flow of control is clear but whose primary operations are by other procedures whose precise meanings are left undefined.

For example, the control abstraction of Fibonacci series is shown below:

```
FiboDP(n):
    F[0]← 0
    F[1] ←1
    for i = 2 to n do
        F[i]←F[i-1] + F[i-2]
    endfor
    return(F[n])
```

Space-Saving Dynamic Programming

```
FiboDP(n):
    prev←0
    curr ← 1
    for i=2 to n do
        temp ←curr
        curr ←curr + prev
        prev ←temp
    endfor
    return(curr)
```

Dynamic Programming is useful when same recursive sub-problems occur repeatedly.

Ques 12) What is difference between divide & conquer and dynamic programming?

Ans: Divide & Conquer *versus* Dynamic Programming

Table 4.2 describes the difference between Divide and Conquer and Dynamic Programming.

Table 4.2: Divide and Conquer *versus* Dynamic Programming

Divide and Conquer	Dynamic Programming
The problem is divided into small sub-problems. These sub-problems are solved independently. Finally, all the solutions of sub-problems are collected together to get the solution to the given problem.	In dynamic programming, many decision sequences are generated and all the overlapping sub-instances are considered.
In this method, duplications in sub-solutions are neglected, i.e., duplicate sub-solutions may be obtained.	In dynamic computing, duplications in solutions are avoided totally.
Divide and conquer is less efficient because of re-work on solutions.	Dynamic programming is efficient than divide and conquer strategy.
The divide and conquer uses top-down approach of problem solving (recursive methods).	Dynamic programming uses bottom-up approach of problem solving (iterative method).
Divide and conquer splits its input at specific deterministic points usually in the middle.	Dynamic programming splits its input at every possible split points rather than at a particular point. After trying all split points it determines which split point is optimal.

Ques 13) Discuss how to develop a dynamic programming algorithm.

Ans: Developing a Dynamic Programming Algorithm

The essence of dynamic programming algorithms is that they trade space for speed by storing solutions to sub-problems rather than re-computing them.

Some general points about 'how to develop a dynamic programming solution to a problem' are given as follows:

1) It is usually useful to tackle the problem "top-down" as if we were going to develop a recursive algorithm.

2) If it appears that saving results from smaller problems can avoid repeated computation, define an appropriate dictionary for saving results and make a clear statement to characterize the entries in the dictionary.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

3) Based on the number of sub-problems (maximum size of the dictionary is an upper bound) and the number of edges in the sub-problem graph, the complexity of the dynamic programming procedure can be analyzed by its relationship to depth-first search on the sub-problem graph.

4) Decide on an appropriate data structure for the dictionary. Often it is a one- or two-dimensional array.

5) If possible, analyze the structure of the sub-problem graph and figure out a simpler order in which the dictionary entries can be computed.

6) Determine how to get the solution to the problem from the data in the dictionary.

Steps for Development of Dynamic Programming Problem

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

Step 1: Characterize structure of an optimal solution.

Step 2: Recursively define value of an optimal solution.

Step 3: Compute the value of an optimal solution in a bottom-up fashion.

Step 4: Construct an optimal solution from computed information.

Steps 1-3 form the basis of dynamic programming solution to a problem. Step 4 can be omitted if only the value of an optimal solution is required. When step 4 is performed additional information during the computation in step 3 is maintained to ease the construction of an optimal solution.

Ques 14) What are the different elements of dynamic programming?

Ans: Elements of Dynamic Programming

Dynamic programming is mainly used to tackle problems that are solvable in polynomial time. There are two key attributes that a problem must have in order for dynamic programming to be applicable:

1) **Optimal Substructure:** It means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub problems. Consequently, the first step towards devising a dynamic programming solution is to check whether the problem exhibits such optimal substructure. Such optimal substructures are usually described by means of recursion.

   A common pattern in discovering optimal substructure is:

   i) The user shows that a solution to the problem consists of making a choice, such as choosing a preceding assembly line station or choosing an index at which to split the matrix chain.

   ii) The user supposes that for a given problem, he is given the choice that leads to an optimal solution.

   iii) Given this choice, user determines which sub-problems ensue and how to best characterize the resulting space of sub-problems.

   iv) The user shows that the solutions to the sub-problems used within the optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique.

   Optimal substructure varies across problem domains in two ways:

   i) How many sub-problems are used in an optimal solution to the original problem, and

   ii) How many choices we have in determining which sub-problem(s) to use in an optimal solution.

   For example, given a graph G = (V, E), the shortest path p from a vertex u to a vertex v exhibits optimal substructure: take any intermediate vertex w on this shortest path p. If p is truly the shortest path, then the path p_1 from u to w and p_2 from w to v are indeed the shortest paths between the corresponding vertices. Hence, one can easily formulate the solution for finding shortest paths in a recursive manner which is what the Bellman-Ford algorithm does.

2) **Overlapping Sub-Problems:** It means that the space of sub-problems must be small, i.e., any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new subproblems. **For example,** consider the recursive formulation for generating the Fibonacci series: $F_i = F_{i-1} + F_{i-2}$, with base case $F_1 = F_2 = 1$. Then $F_{43} = F_{42} + F_{41}$ and $F_{42} = F_{41} + F_{40}$. Now F_{41} is being solved in the recursive subtrees of both F_{43} as well as F_{42}.

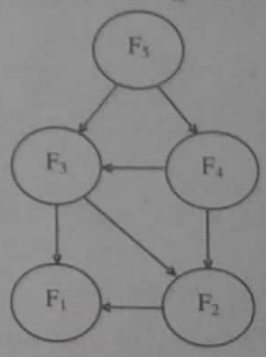


Figure 4.2: Sub-problem Graph for Fibonacci Sequence

Even though the total number of subproblems is actually small (only 43 of them), we end-up solving the same problems over and over, if we adopt a naive recursive solution such as this. Dynamic programming takes account of this fact and solves each sub-problem only once.It is not a tree indicating overlapping subproblems.

Ques 15) What are the applications of dynamic programming?

Ans: Applications of Dynamic Programming

Dynamic programming approach is quite applicable in real life scenario. Some of the most important of applications of dynamic programming problems are:

1) Matrix-Chain Multiplication (Order-chain Multiplication)

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

2) Multistage Graphs
3) All Pairs Shortest Path Problem
4) Optimal Binary Search Trees
5) 0/1 Knapsack Problem
6) Traveling Salesman Problem
7) Flow Shop Scheduling

Ques 16) Explain the optimal matrix multiplication with example.

Or

Write the algorithm for optimal matrix multiplication and analyse them.

Ans: Matrix-Chain Multiplication (Order-Chain Multiplication)

The common application of dynamic programming is an algorithm that solves the problem of matrix chain multiplication. Chain matrix multiplication problem is to find the product of a given collection of matrices in the indicated manner. Let us suppose that one have to find the product:

$$M = M_1 \times M_2 \times \ldots \times M_n$$

Where, M_1 has $r_i - 1$ rows and r_1 columns.

The order in which matrices are multiplied together can have significant effect on the total number of operations required to evaluate M, no matter what matrix multiplication method is used.

For example,

M =	M_1	×	M_2	×	M_3	×	M_4
	[20 × 50]		[50 × 3]		[3 × 50]		[50 × 2]

Evaluating M in the order:

1) $M_1 \times (M_2 \times (M_3 \times M_4))$ requires 2,600 operations as detailed below:
2) $M_3 \times M_4$ requires 300 operations
3) $M_2 \times (M_3 \times M_4)$ requires an additional 300 operations
4) $M_1 \times (M_2 \times (M_3 \times M_4))$ requires an additional 2,000 operations, i.e., a total of 2,600 operations are required.

While evaluating M in the order:

5) **(M1 × (M2 × M3)) × M4** requires 59,500 operations as detailed below:
6) **M2 × M3** requires 7,500 operations.
7) **(M1 × (M2 × M3))** requires an additional 50,000 operations.
8) **(M1 × (M2 × M3)) × M4** requires an additional 2,000 operations so that the total operations count is 59,500.

Algorithm: Chain-matrix ($M_1, M_2, \ldots, M_n, r_0, r_1, \ldots, r_n$)

Step 1: Array c(n, n), M_1 (r_0, r_1), M_2(r_1, r_2), … M_n (r_{n-1}, r_n)

Step 2: For i = 1 to n Do

Step 3: c (i, i) = 0;

Step 4: For p = 1 to n – 1 Do

Step 5: For i = 1 to n – p Do

Step 6: j = i + p;

Step 7: c (i, j) = ∞;

Step 8: For ik = i to j – 1 Do

Step 9: S = c(i, ik) + c(ik + 1, j) + $r_{i-1} r_{ik} r_j$;

Step 10: If s < c(i, j) Then

Step 11: C(i, j) = s;

Step 12: k = ik;

Step 13: Write c(1, n);

Step 14: End Chain-matrix

Steps to find the Solution

Input

A chain of n matrices ($A_1, A_2, \ldots, A_n$) such that:
Cols[A_i] = Rows[A_{i+1}], for i = 1, 2,…, n

Output

An optimal, fully parenthesized product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ (i.e., in a way that minimises the total number of scalar multiplications).

Step 1: Characterise the Structure of an Optimal Solution

Assume that the optimal way to multiply $A_1A_2\ldots A_n$ is $(A_1A_2\ldots A_k)(A_{k+1}\ldots A_n)$, for some k.

Its associated cost is:
cost($A_1\ldots A_n$) = cost($A_1\ldots A_k$) + cost($A_{k+1}\ldots A_n$) + rows[A_1] · col[A_k] · col[A_n] (1)

If cost($A_1\ldots A_n$) is minimal, then both cost ($A_1\ldots A_k$) and cost($A_{k+1}\ldots A_n$) are minimal.

Step 2: Recursively Define the Value of an Optimal Solution

Consider the general case of multiplying $A_i, \ldots A_j$.

Use an array p[] to record the dimensions of matrices. Thus, each matrix A_i is $p_{i-1} \times p_i$, where p_{i-1} = rows[A_i], p_i = col[A_i]. Define m[i, j] to be the minimum cost of multiplying $A_i, \ldots A_j$.

Then:

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

Directly compute m[i, j] based on the recursive solution:

Algorithm

RECURSIVE-MATRIX-CHAIN (p, i, j)
if i = j return 0;
m[i, j] := ∞;

for k := i to j – 1

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

q:= RECURSIVE-MATRIX-CHAIN (p, i, k) + RECURSIVE-MATRIX-CHAIN (p, k + 1, j) + $p_k \cdot p_j$

if q < m[i, j]

m[i, j] := q;

return m[i, j]

The following recurrence relation describes the running time of RECURSIVE-MATRIX-CHAIN( ):

$$T(1) \geq 1$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$$

$$= 2\sum_{i=1}^{n-1} T(i) + n$$

Minimum Operation Count

The algorithm is used to find the optimal way of multiplying the above sequence of matrices for minimum operation count. The computational result is given in **table 4.3**.

Table 4.3: Chain Matrix Multiplication

c (1, 1) = 0	c 2, 2) = 0	c (3, 3) = 0	c (4, 4) = 0
c (1, 2) = 3,000	c (2, 3) = 7,500	c (3, 4) = 300	
c (1, 3) = 6,000	c (2, 4) = 600		
k = 2	k = 2		
c (1, 4) = 2,600			
k = 1			

Thus, the minimum cost of finding the product of the example chain matrix multiplication $M = M_1 \times M_2 \times M_3 \times M_4$ is 2,600. This minimum is based on multiplying $M_2 \times M_3 \times M_4$ which, in turn, is based on multiplying $M_3 \times M_4$. The sequence of multiplication for minimum cost is as follows:

First, multiply M_3 and M_4 producing matrix A at a cost of 300. We then find $M_2 \times A$ producing matrix B at a cost of 300. Finally, multiply M_1 and B producing M at an additional cost of 2,000.

Dynamic Programming Approach

The dynamic programming approach calculates the c(i, j)s in the order of increasing difference in the subscripts. One begin by calculating c(i, i) for all i, then c(i, i + 1) for all i, next c(i, i + 2) and so on. In this way, the terms c(i, i_k) and c(i_k + 1, j) will be available when calculating c(i, j). This follows since j – i must be strictly greater than either of $i_k - i$ and $j - (i_k + 1)$ if i_k is in the range $i \leq i_k < j$.

Table 4.3 gives the optimal count of operations, but it is a simple extension of the above algorithm to find the sequence of multiplication. If at each stage of computation of c(i, i), one keep record of i_k that produced the minimum value of c(i, j), we will be able to decide the optimal sequence of the chain matrix multiplication.

The dynamic programming approach calculates the c(i, j)s in the order of increasing difference in the subscripts. One begin by calculating c(i, i) for all i, then c(i, i + 1) for all i, next c(i, i + 2) and so on. In this way, the terms c(i, i_k) and c(i_k + 1, j) will be available when calculating c(i, j). This follows since j – i must be strictly greater than either of $i_k - i$ and $j - (i_k + 1)$ if i_k is in the range $i \leq i_k < j$.

Table 4.3 gives the optimal count of operations, but it is a simple extension of the above algorithm to find the sequence of multiplication. If at each stage of computation of c(i, i), one keep record of i_k that produced the minimum value of c(i, j), we will be able to decide the optimal sequence of the chain matrix multiplication.

Ques 17) Consider the following matrix and its dimensions.

Matrix	Dimension
A_1	30 × 35
A_2	35 × 15
A_3	15 × 5
A_4	5 × 10
A_5	10 × 20
A_6	20 × 25

Find optimal cost of multiplying A2×5.

Ans: Consider the different cost of multiplying matrix A2A3A4A5 which can be parenthesized in different manner.

Therefore, cost is given by,

m[2,5] = m[2,2] + m[3,5] + P1P2P5. (i = 2, k = 2, j = 5) ..... (1)

If it is (A2A3) (A4A5) then i = 2, k = 3

Therefore, cost is given by,

m[2,5] = m[2,3] + m[4,5] + P1P3P5. (i = 2, k = 3, j = 5) ..... (2)

If it is (A2A3A4)(A5) then i = 2, k = 4 therefore its cost is given by

m[2,5] = m[2,4] + m[5, 5] + P1P4P5 ..... (3)

Now to compute equation (1) we have to compute m[3,5] first as m[2,2] is 0.

$$m[3,5] = \min \begin{cases} A_3(A_4A_5) = \underset{(15\times5)}{m[3,3]} + \underset{(5\times20)}{m[4,5]} + \underset{(15\times5\times20)}{P_2P_3P_5} \\ (A_3A_4)A_5 = \underset{(15\times10)}{m[3,4]} + \underset{(10\times20)}{m[5,5]} + \underset{(15\times10\times20)}{P_2P_4P_5} \end{cases}$$

$$= \underset{(1000)}{0+5\times10\times20} + \underset{(1500)}{15\times5\times20} = 2500$$

$$= \underset{(750)}{10\times5\times10+0} + \underset{(3000)}{15\times10\times20} = 3750$$

m[3, 5] = 2500 is the minimum, so it is chosen.

Thus at each stage of parenthesization, one calculates the minimum cost and is added to obtain the optimal cost. Similarly one can calculate m[2, 4] in equation (3) thus final optimal cost is given by,

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

$$m[2,5] = \min \begin{bmatrix} m[2,2]+m[3,5]+P_1P_2P_5 \\ m[2,3]+m[4,5]+P_1P_3P_5 \\ m[2,4]+m[5,5]+P_1P_4P_5 \end{bmatrix}$$

$= 0 + 2500 + 35.15.20 = 1300$
$= 2625 + 1000 + 35.5.20$
$= 7125$ is the optimal cost.

Ques 18) Describe the Bellman-Ford Algorithms and analyse them.

Ans: Bellman-Ford Algorithms

Shortest-paths algorithms typically exploit the property that a shortest path between two vertices contains other shortest paths within it. This optimal sub-structure property admits both **dynamic programming** and the **greedy method**. Dijkstra's algorithm might fail in delivering a shortest-paths tree when the graph has some negative-weight edges.

The Bellman-Ford algorithm works in a more-general case than Dijkstra's algorithm. It solves the shortest-paths tree problem even when the graph has negative-weight edges. If there is a negative-weight cycle, the algorithm indicates that no solution exists.

Dijkstra's algorithm relaxes only those edges incident to the chosen vertex with the minimum shortest-path estimate. The failure of Dijkstra's algorithm for graphs with negative-weight edges is due to the fact that it does not calculate the domino effect caused by negative edges.

Initially, the shortest-path estimates of all vertices other than the source are set to be ∞. Then the algorithm makes $n - 1$ passes over all the edges of the graph. In each pass, if $\delta[v] > \delta[u] + w(u,v)$, then set the value of $\delta[v]$ to be $\delta[u] + w(u, v)$, and modify the predecessor of vertex v.

A notable feature of the Bellman-Ford algorithm is that in the kth iteration, the shortest-path estimate for vertex v, i.e., $\delta[v]$, equals the length of the shortest path from the source to v with at most k edges. If, after $n - 1$ passes, there exists an edge (u,v) such that $\delta[v] > \delta[u] + w(u,v)$, then a negative cycle has been detected. Otherwise, for all vertices other than the source, we build the shortest-paths tree by adding the edges from their predecessors to them.

Algorithm: Bellman-Ford

Input: A weighted, directed graph G = (V,E, w); a source vertex s.

Step 1: for each vertex $v \in V$ do
Step 2: $\delta[v] \leftarrow \infty$
Step 3: $\pi[v] \leftarrow$ NIL
Step 4: $\delta[s] \leftarrow 0$
Step 5: for $i \leftarrow 0$ to $n - 1$ do
Step 6: for each $(u, v) \in E$ do
Step 7: if $\delta[v] > \delta[u] + w(u, v)$ then
Step 8: $\delta[v] \leftarrow \delta[u] + w(u, v)$
Step 9: $\pi[v] \leftarrow u$
Step 10: for each $(u, v) \in E$ do
Step 11: if $\delta[v] > \delta[u] + w(u, v)$ then
Step 12: Output "A negative cycle exists."
Step 13: Exit
Step 14: $T \leftarrow \varphi$
Step 15: For $v \in V - s$ do
Step 16: $T \leftarrow T \cup \{(\pi[v], v)\}$

Execution of Bellman-Ford Algorithm

Consider following figures show how the execution of the Bellman-Ford works on the graph .The dark dotted edges record those that do cause some effect in the relaxation. Initially, $\delta[a] = 0$. In **figure 4.3**, the relaxation of edge (a, b) changes $\delta[b]$ from ∞ to 2; and the relaxation of edge (a, g) changes $\delta[g]$ from ∞ to 5.

Shortest-path estimates $\delta[a]$, $\delta[b]$, and $\delta[g]$ are finalized. Shortest-path estimates $\delta[c]$, $\delta[d]$, $\delta[e]$, $\delta[f]$, and $\delta[h]$ are still ∞ since their corresponding vertices cannot be reached from vertex a by a path with only one edge.

In **figure 4.4**, the relaxation of edge (b, c) changes $\delta[c]$ from ∞ to 4; the relaxation of edge (b, d) changes $\delta[d]$ from ∞ to 3; the relaxation of edge (g, e) changes $\delta(e)$ from ∞ to 2; and the relaxation of edge (g, h) changes $\delta[h]$ from ∞ to 6.

Up to this stage, shortest-path estimates $\delta[a]$, $\delta[b]$, $\delta[c]$, $\delta[d]$, $\delta[e]$, and $\delta[g]$ are finalized. Shortest-path estimate $\delta[f]$ is still ∞ since vertex f cannot be reached from vertex a by a path with at most two edges, and shortest-path estimate $\delta[h]$ will be modified later.

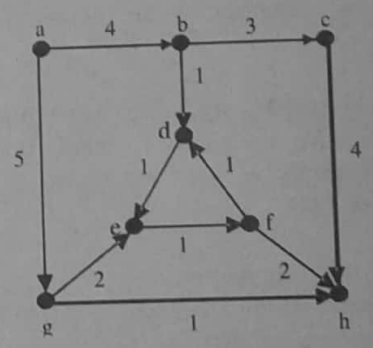


Figure 4.3: Weighted, Directed Graph

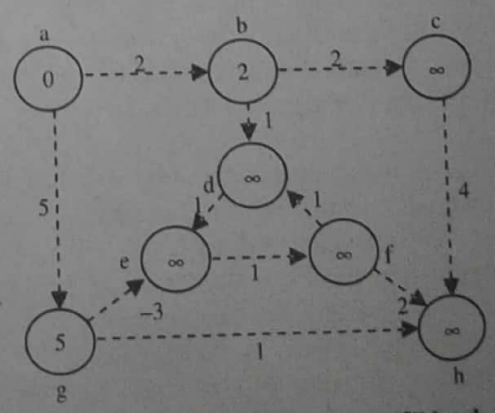


Figure 4.4: Shortest-Path estimates δ[b] and δ[g] are modified.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner

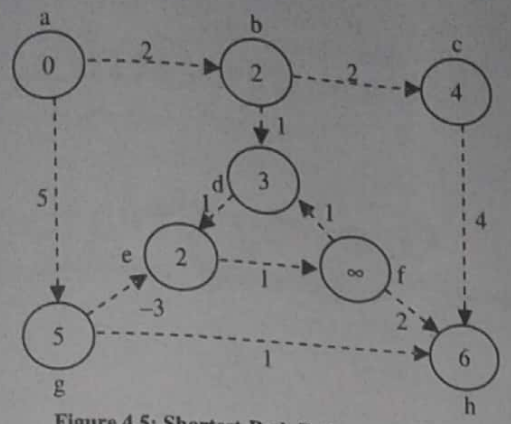


Figure 4.5: Shortest-Path Estimates δ[c], δ[d], δ[e] and δ[h] are modified.

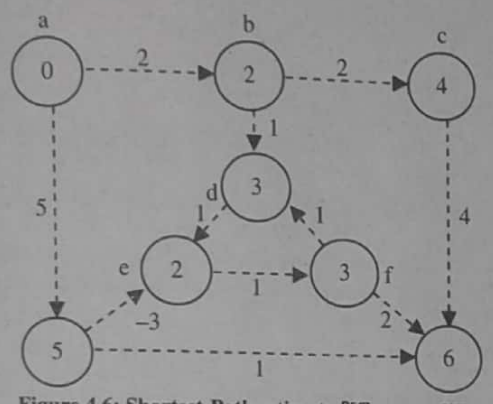


Figure 4.6: Shortest-Path estimate δ[f] are modified.

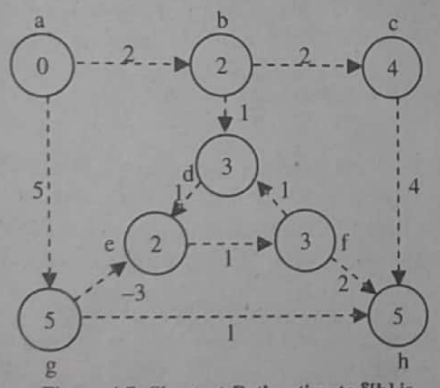


Figure 4.7: Shortest-Path estimate δ[h] is modified.

In **figure 4.6**, the relaxation of edge (e, f) changes δ[f] from ∞ to 3. In **figure 4.7**, the relaxation of edge (f, h) changes δ[h] from 6 to 5. In **figure 4.8**, gives the final shortest-paths tree constructed by the Bellman-Ford algorithm.

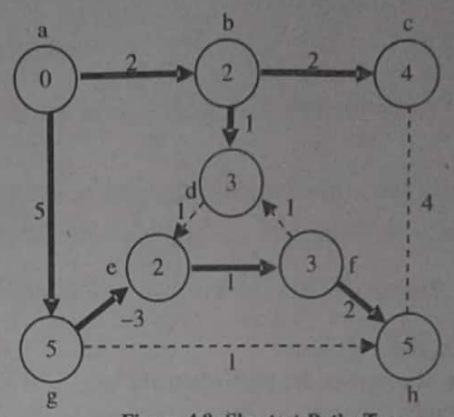


Figure 4.8: Shortest-Paths Tree Constructed by the Bellman-Ford Algorithm.

Analysis of Bellman-Ford Algorithm

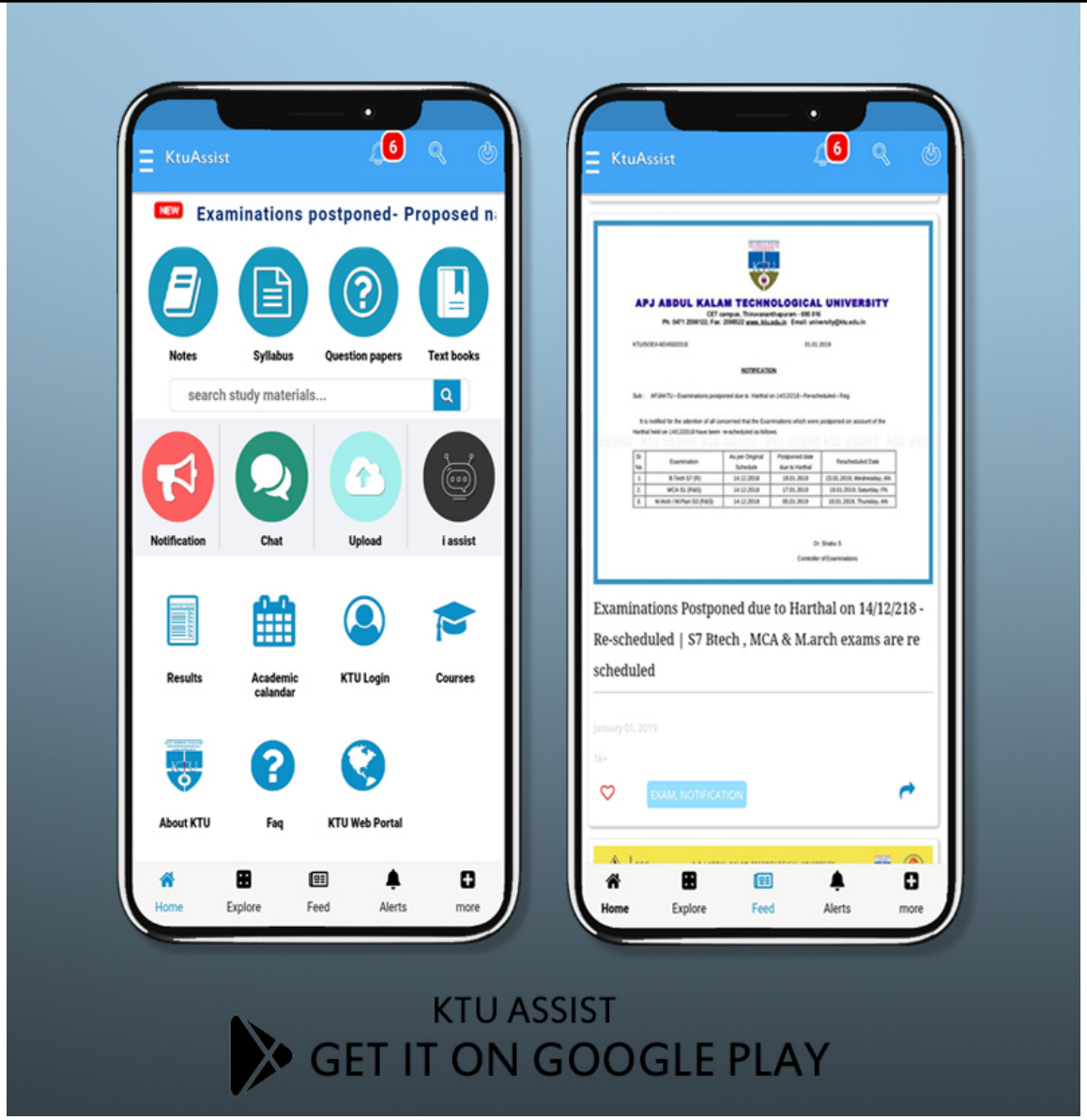
1) The Bellman-Ford algorithm has the running time of O(mn) since there are O(n) iterations, and each iteration takes O(m) time.

2) The correctness follows from the fact that in the kth iteration, the shortest-path estimate for each vertex equals the length of the shortest path from the source to that vertex with at most k edges.

3) Since a simple path in G contains at most (n – 1) edges, the shortest-path estimates stabilize after (n – 1) iterations unless there exists a negative cycle in the graph.

DOWNLOADED FROM KTUASSIST.IN

Scanned by CamScanner



END

facebook.com/ktuassist

instagram.com/ktu_assist