Rizvi College Of Arts Science and Commerce

Fundamental Of Algorithm

Unit-Third

Class: - SY B.Sc.(CS)

Dr. Ruchi Gupta

Assistant Professor

CS\IT Department

B. Sc. (COMPUTER SCIENCE) Semester – IV

Course: USCS401 **TOPICS (Credits: 02 Lectures/Week: 03)**

Fundamentals of Algorithms

Unit III

Algorithms Design Techniques: Introduction, Classification, Classification by Implementation Method, Classification by Design Method

Greedy Algorithms: Introduction, Greedy Strategy, Elements of Greedy Algorithms, Advantages and Disadvantages of Greedy Method, Greedy Applications, Understanding Greedy Technique

Divide and Conquer Algorithms: Introduction, What is Divide and Conquer Strategy? Divide and Conquer Visualization, Understanding Divide and Conquer, Advantages of Divide and Conquer, Disadvantages of Divide and Conquer, Master Theorem, Divide and Conquer Applications

Dynamic Programming: Introduction, What is Dynamic Programming Strategy? Properties of Dynamic Programming Strategy, Problems which can be solved using Dynamic Programming, Dynamic Programming Approaches, Examples of Dynamic Programming Algorithms, Understanding Dynamic Programming, Longest Common Subsequence

Algorithms Design Techniques

Classification by purpose
Each algorithm has a goal, for example, the purpose of the Quick Sort algorithm is to sort data in ascending or descending order. But the number of goals is infinite, and we have to group them by kind of purposes.

Classification by implementation
An algorithm may be implemented according to different basically principles.

Recursive or iterative
A recursive algorithm is one that calls itself repeatedly until a certain condition matches. It is a method common to functional programming. Iterative algorithms use repetitive constructs like loops. Some problems are better suited for one implementation or the other. For example, the towers of hanoi problem is well understood in recursive implementation. Every recursive version has an iterative equivalent iterative, and vice versa.

Logical or procedural
An algorithm may be viewed as controlled logical deduction.

A logic component expresses the axioms which may be used in the computation and a control component determines the way in which deduction is applied to the axioms.

This is the basis of the logic programming. In pure logic programming languages the control component is fixed and algorithms are specified by supplying only the logic component.

Serial or parallel
Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. This is a serial algorithm, as opposed to parallel algorithms, which take advantage of computer architectures to process several instructions at once. They divide the problem into sub-problems and pass them to several processors. Iterative algorithms are generally parallelizable. Sorting algorithms can be parallelized efficiently.

Deterministic or non-deterministic
Deterministic algorithms solve the problem with a predefined process whereas non-deterministic algorithm must perform guesses of best solution at each step through the use of heuristics.

Classification by design paradigm
A design paradigm is a domain in research or class of problems that requires a dedicated kind of algorithm:

Divide and conquer
A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively), until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in conquer phase by merging them.

The binary search algorithm is an example of a variant of divide and conquer called decrease and conquer algorithm, that solves an identical subproblem and uses the solution of this subproblem to solve the bigger problem.

Dynamic programming
The shortest path in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices.

When the optimal solution to a problem can be constructed from optimal solutions to subproblems, using dynamic programming avoids recomputing solutions that have already been computed.

- The main difference with the "divide and conquer" approach is, subproblems are independent in divide and conquer, where as the overlap of subproblems occur in dynamic programming.

- Dynamic programming and memoization go together. The difference with straightforward recursion is in caching or memoization of recursive calls. Where subproblems are independent, this is useless. By using memoization or maintaining a table of subproblems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.

Greedy method

A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the subproblems do not have to be known at each stage. Instead a "greedy" choice can be made of what looks the best solution for the moment.

The most popular greedy algorithm is finding the minimal spanning tree as given by Kruskal.

Linear programming

The problem is expressed as a set of linear inequalities and then an attempt is made to maximize or minimize the inputs. This can solve many problems such as the maximum flow for directed graphs, notably by using the simplex algorithm. A complex variant of linear programming is called integer programming, where the solution space is restricted to all integers.

Reduction also called transform and conquer

Solve a problem by transforming it into another problem. A simple example: finding the median in an unsorted list is first translating this problem into sorting problem and finding the middle element in sorted list. The main goal of reduction is finding the simplest transformation possible.

Backtracking

Many problems, such as playing chess, can be **modelled** as problems on graphs. A graph exploration algorithms are used. This category also includes the search algorithms and backtracking.

Probabilistic

Those that make some choices randomly.

Genetic

Attempt to find solutions to problems by mimicking biological evolutionary processes, with a cycle of random mutations yielding successive generations of "solutions". Thus, they emulate reproduction and "survival of the fittest".

Heuristic

Whose general purpose is not to find an optimal solution, but an approximate solution where the time or resources to find a perfect solution are not practical?

Greedy Algorithm

Introduction

"Greedy Method finds out of many options, but you have to choose the best option." In this method, we have to find out the best method/option out of many present ways. In this approach/method we focus on the first stage and decide the output, don't think about the future. This method may or may not give the best output.
Greedy Algorithm solves problems by making the best choice that seems best at the particular moment. Many optimization problems can be determined using a greedy algorithm. Some issues have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal.
A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution. However, generally greedy algorithms do not provide globally optimized solutions.

Formal Definition

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

In general, greedy algorithms have five components:

- A candidate set, from which a solution is created
- A selection function, which chooses the best candidate to be added to the solution
- A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
- An objective function, which assigns a value to a solution, or a partial solution, and
- A solution function, which will indicate when we have discovered a complete solution

A greedy algorithm works if a problem exhibits the following two properties:

1. **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.
2. **Optimal substructure:** Optimal solutions contain optimal sub solutions. In other words, answers to subproblems of an optimal solution are optimal.

Example:

- machine scheduling
- Fractional Knapsack Problem
- Minimum Spanning Tree
- Huffman Code
- Job Sequencing
- Activity Selection Problem

Steps for achieving a Greedy Algorithm are:

- **Feasible:** Here we check whether it satisfies all possible constraints or not, to obtain at least one solution to our problems.
- **Local Optimal Choice:** In this, the choice should be the optimum which is selected from the currently available
- **Unalterable:** Once the decision is made, at any subsequence step that option is not altered.

History of Greedy Algorithms

Here is an important landmark of greedy algorithms:

- Greedy algorithms were conceptualized for many graph walk algorithms in the 1950s.
- Esdger Djikstra conceptualized the algorithm to generate minimal spanning trees. He aimed to shorten the span of routes within the Dutch capital, Amsterdam.
- In the same decade, Prim and Kruskal achieved optimization strategies that were based on minimizing path costs along weighed routes.

- In the '70s, American researchers, Cormen, Rivest, and Stein proposed a recursive substructuring of greedy solutions in their classical introduction to algorithms book.
- The greedy paradigm was registered as a different type of optimization strategy in the NIST records in 2005.
- Till date, protocols that run the web, such as the open-shortest-path-first (OSPF) and many other network packet switching protocols use the greedy strategy to minimize time spent on a network.

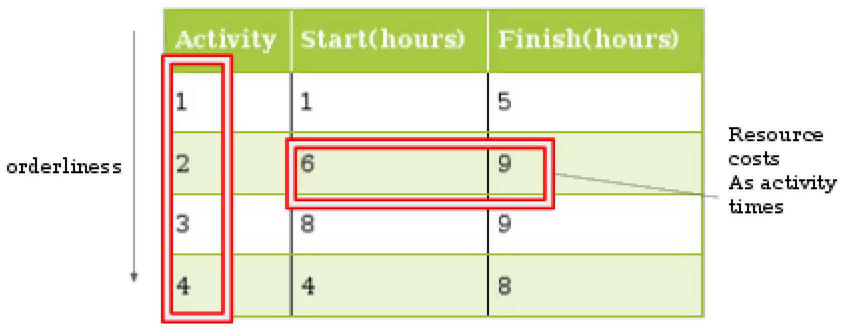
Characteristics of the Greedy Approach

The important characteristics of a greedy method are:

There is an ordered list of resources, with costs or value attributions. These quantify constraints on a system.

You will take the maximum quantity of resources in the time a constraint applies.

For example, in an activity scheduling problem, the resource costs are in hours, and the activities need to be performed in serial order.


orderliness
Resource costs As activity times

Activity	Start(hours)	Finish(hours)
1	1	5
2	6	9
3	8	9
4	4	8

Why use the Greedy Approach?

Here are the reasons for using the greedy approach:

- The greedy approach has a few trade-offs, which may make it suitable for optimization.
- One prominent reason is to achieve the most feasible solution immediately. In the activity selection problem (Explained below), if more activities can be done before finishing the current activity, these activities can be performed within the same time.
- Another reason is to divide a problem recursively based on a condition, with no need to combine all the solutions.
- In the activity selection problem, the "recursive division" step is achieved by scanning a list of items **only once** and considering certain activities.

How to solve the activity selection problem

In the activity scheduling example, there is a "start" and "finish" time for every activity. Each Activity is indexed by a number for reference. There are two activity categories.

- **Considered activity**: is the Activity, which is the reference from which the ability to do more than one remaining Activity is analyzed.
- **Remaining activities:** activities at one or more indexes ahead of the considered activity.

The total duration gives the cost of performing the activity. That is (finish - start) gives us the durational as the **cost** of an activity.

You will learn that the **greedy extent** is the number of remaining activities you can perform in the time of a considered activity.

Defining the Core Concept

Till now, we know what a greedy algorithm is and why is it named so. The below pointers will make you understand the greedy algorithm in a better way. By now, it has been very clear that the greedy algorithm only works when there is a problem; nevertheless, this approach is only applicable if we have a condition or constraint to that problem.

Types of Problems

Minimization Problem: Getting a solution to a problem is easy given that all the conditions are met. However, when this problem demands a minimum result, it is then called a Minimization Problem.

Maximization Problem: A problem that demands the maximum result is known as the maximization problem.

Optimization Problem: A problem is called optimization problem when it requires either minimum or maximum results.

Types of Solutions

Feasible Solution: Now when a problem arises, we have many plausible solutions to this problem. Yet, taking into consideration the condition set on that problem, we choose solutions that satisfy the given condition. Such solutions that help us get results meeting the given condition is called a Feasible Solution**.**

Optimal Solution: A solution is called optimal when it is already feasible and achieves the objective of the problem; the best result. This objective could either be the minimum or maximum result. The point here to be noticed is that any problem will only have one optimal solution.

Core Components of a Greedy Algorithm

Now when we have a better understanding of this mechanism, let’s explore the core components of a greedy algorithm that sets it apart from other processes:

- **Candidate set:** An answer is created from this set.
- **Selection function:** It selects the best candidate to be included in the solution.
- **Feasibility function:** This section calculates if a candidate can be used to contribute to the solution.
- **An objective function:** It assigns a value to a complete or a partial solution.
- **A solution function:** This is used to indicate if a proper solution has been met.

Where does the Greedy Algorithm work the best?

Greedy Algorithm can be applied to the below-mentioned problems.

- The Greedy approach can be used to find the minimal spanning tree graph using Prim’s or Kruskal’s algorithm
- Finding the shortest path between two vertices is yet another problem that can be solved using a greedy algorithm. Applying the Dijkstra’s algorithm along with the greedy algorithm will give you an optimal solution.
- Activity Selection Problem
- Dijkstra’s Algorithm for Adjacency List Representation
- Dijkstra’s Shortest Path Algorithm
- Huffman Coding Algorithm
- Efficient Huffman Coding for Sorted Input
- Job Sequencing Problem with Deadlines
- Kruskal’s Minimum Spanning Tree Algorithm
- Minimum Coin Change Problem
- Minimum Number of Platforms Problem
- Prim’s Minimum Spanning Tree Algorithm
- Prim’s MST for Adjacency List Representation
- Fractional Knapsack Problem

Advantages and disadvantages**:**

Advantages

1. It is quite easy to **come up with a greedy algorithm** (or even multiple greedy algorithms) for a problem.
2. **Analysing the run time for greedy algorithms will generally be much easier** than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.
3. They are easier to implement.
4. They require much less computing resources.

5. They are much faster to execute.
6. Greedy algorithms are used to solve optimization problems

Disadvantages:

1. Their only disadvantage being that they not always reach the global optimum solution.
2. On the other hand, even when the global optimum solution is not reached, most of the times the reached sub-optimal solution is a very good solution.
3. The difficult part is that for greedy algorithms **you have to work much harder to understand correctness issues**. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

Huffman Codes

- Huffman Algorithm was developed by David Huffman in 1951.
- This is a technique which is used in a data compression or it can be said that it is a coding technique which is used for encoding data.
- This technique is a mother of all data compression scheme.
- This idea is basically dependent upon the frequency, i.e. the frequency of the corresponding character which needs to be compressed, and by that frequency, only Huffman code will be generated.
- In case of Huffman coding, the most generated character will get the small code and least generated character will get the large code.
- Huffman tree is a specific method of representing each symbol.
- This technique produces a code in such a manner that no codeword is a prefix of some other code word. These codes are called as prefix code.

Algorithm for Huffman code

```
1.  Input:-Number of message with frequency count.
2.  Output: - Huffman merge tree.
3.  Begin
4.  Let Q be the priority queue,
5.  Q= {initialize priority queue with frequencies of all symbol or message}
6.  Repeat n-1 times
7.  Create a new node Z
8.  X=extract_min(Q)
9.  Y=extract_min(Q)
10. Frequency(Z) =Frequency(X) +Frequency(y);
11. Insert (Z, Q)
12. End repeat
13. Return (extract_min(Q))
14. End.
```

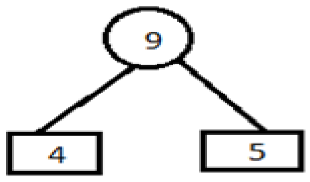
Example

Let obtain a set of Huffman code for the message **(m1.....m7)** with relative frequencies **(q1.....q7) = (4,5,7,8,10,12,20)**. Let us draw the Huffman tree for the given set of codes.

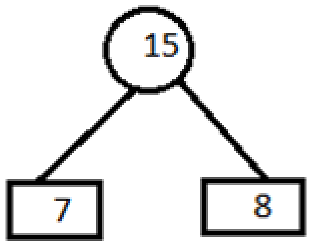
Step 1) Arrange the data in ascending order in a table.

4,5,7,8,10,12,20

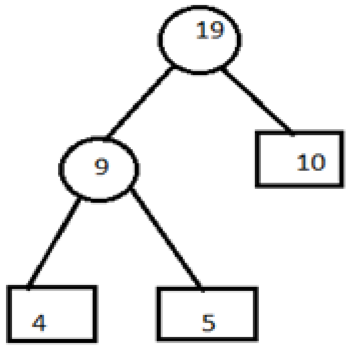
Step 2) Combine first two entries of a table and by this create a parent node.



Step 3)



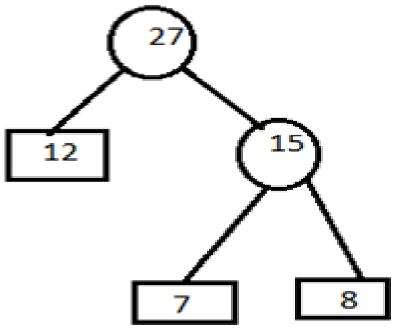
A) Remove the entries 4 and 5 from the table and inert 9 at its appropriate position. **7,8,9,10,12,20.** Combine minimum value of table and create a parent node.

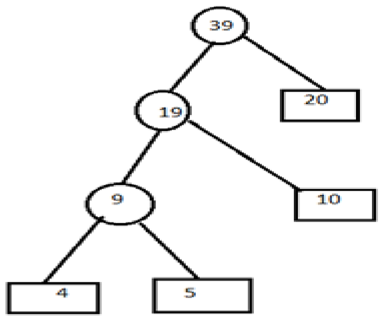


B) Now remove the entries 7 and 8 from the table and insert 15 at its appropriate position. **9,10,12,15,20.** Combine minimum value of two blocks and create a parent node.

C) Remove the entries 9 and 10 from the table and insert 19 at its proper position. **12,15,19,20.**

Combine minimum value of two blocks and create parent node.



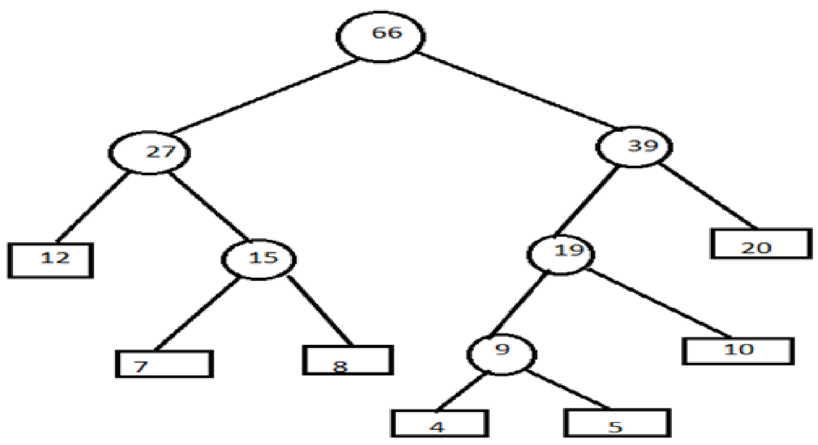


D) Remove the entries 15 and 12 from the table and insert 27 at its appropriate position. **19,20,27**

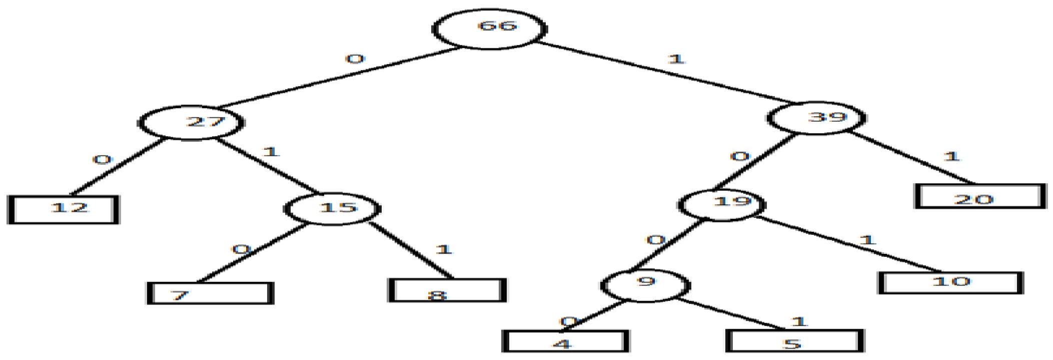
Combine minimum value of two blocks and create parent node.

E) Remove the entries 19 and 20 from the table and insert 39 in the table. **27,39**

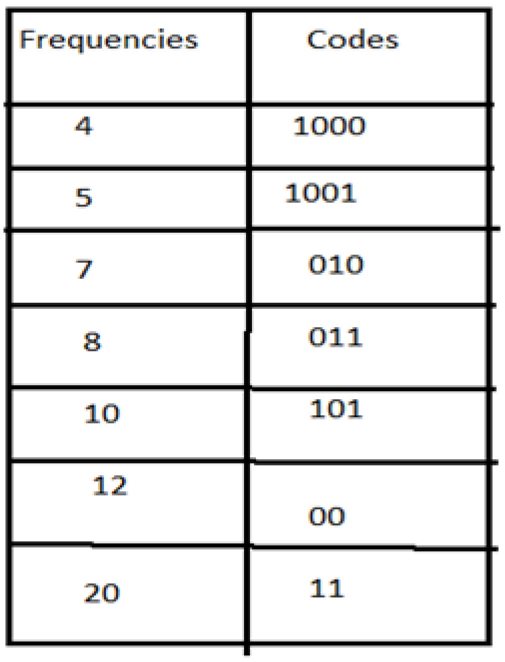
Combine minimum value of two blocks and create parent node.



Step 4) Now assign left child as 0 and right child as 1 to encode the frequencies.



Now, codes for the given frequencies are given below:



Frequencies	Codes
4	1000
5	1001
7	010
8	011
10	101
12	00
20	11

Time complexity:

O(nlogn) is the overall time complexity. Where n is the number of characters.

Advantages of Huffman Encoding-

- This encoding scheme results in saving lot of storage space, since the binary codes generated are variable in length
- It generates shorter binary codes for encoding symbols/characters that appear more frequently in the input string
- The binary codes generated are prefix-free

Disadvantages of Huffman Encoding-

- Lossless data encoding schemes, like Huffman encoding, achieve a lower compression ratio compared to lossy encoding techniques. Thus, lossless techniques like Huffman encoding are suitable only for encoding text and program files and are unsuitable for encoding digital images.
- Huffman encoding is a relatively slower process since it uses two passes- one for building the statistical model and another for encoding. Thus, the lossless techniques that use Huffman encoding are considerably slower than others.
- Since length of all the binary codes is different, it becomes difficult for the decoding software to detect whether the encoded data is corrupt. This can result in an incorrect decoding and subsequently, a wrong output.

Time Complexity Analysis-

Since Huffman coding uses min Heap data structure for implementing priority queue, the complexity is O(nlogn). This can be explained as follows-

- Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).
- Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).

Since building a min heap and sorting it are executed in sequence, the algorithmic complexity of entire process computes to O(nlogn)

We can have a linear time algorithm as well, if the characters are already sorted according to their frequencies.

Travelling salesman problem

Problem Statement

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are **(*n* - 1)!** number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph ***G* = (*V*, *E*)**, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e*(*u*, *v*)** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d*(*u*, *v*)**, which should be non-negative.

Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities ***S* Є {*1*, *2*, *3*, ... , *n*}** that includes **1**, and ***j* Є *S***, let ***C*(*S*, *j*)** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.

When |***S***| > 1, we define ***C*(*S*, *1*)** = ∝ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

C(S,j)=minC(S−{j},i)+d(i,j)wherei∈Sandi≠jc(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi≠jC(S,j)=minC(S−{j},i)+d(i,j)wherei∈Sandi≠j
c(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi≠j

Naive Solution:

1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.
Time Complexity: Θ(n!)

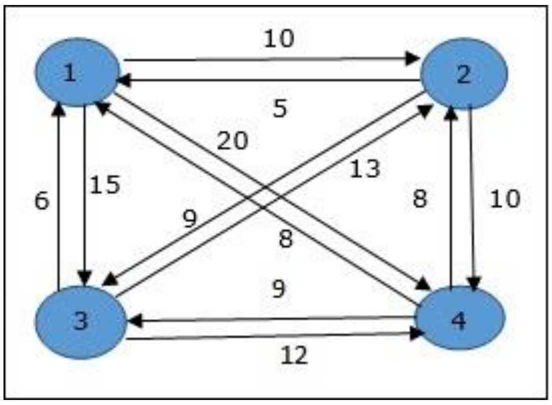
Algorithm: Traveling-Salesman-Problem

```
C ({1}, 1) = 0

for s = 2 to n do

  for all subsets S Є {1, 2, 3, … , n} of size s and containing 1

    C (S, 1) = ∞

  for all j Є S and j ≠ 1

    C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Analysis

There are at the most 2n.n2n.n sub-problems and each one takes linear time to solve. Therefore, the total running time is O(2n.n2)O(2n.n2).

We will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5

Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6

Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8

S = 1

Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}Cost(i,s)=min{Cost(j,s)−(j))+d[i,j]}Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}Cost(i,s)=min{Cost(j,s)−(j))+d[i,j]}

Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15cost(2,{3},1)=d[2,3]+cost(3,Φ,1)=9+6=15Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15cost(2,{3},1)=d[2,3]+cost(3,Φ,1)=9+6=15

Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18cost(2,{4},1)=d[2,4]+cost(4,Φ,1)=10+8=18Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18cost(2,{4},1)=d[2,4]+cost(4,Φ,1)=10+8=18

Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18cost(3,{2},1)=d[3,2]+cost(2,Φ,1)=13+5=18Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18cost(3,{2},1)=d[3,2]+cost(2,Φ,1)=13+5=18

Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20cost(3,{4},1)=d[3,4]+cost(4,Φ,1)=12+8=20Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20cost(3,{4},1)=d[3,4]+cost(4,Φ,1)=12+8=20

Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15cost(4,{3},1)=d[4,3]+cost(3,Φ,1)=9+6=15Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15cost(4,{3},1)=d[4,3]+cost(3,Φ,1)=9+6=15

Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13cost(4,{2},1)=d[4,2]+cost(2,Φ,1)=8+5=13Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13cost(4,{2},1)=d[4,2]+cost(2,Φ,1)=8+5=13

S = 2

Cost(2,{3,4},1)=⎧⎩⎨ d[2,3]+Cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25=25Cost(2,{3,4},1){d[2,3]+cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25=25Cost(2,{3,4},1)={d[2,3]+Cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25=25Cost(2,{3,4},1){d[2,3]+cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25=25

Cost(3,{2,4},1)=⎧⎩⎨ d[3,2]+Cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12+13=25=25Cost(3,{2,4},1){d[3,2]+cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12+13=25=25Cost(3,{2,4},1)={d[3,2]+Cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12+13=25=25Cost(3,{2,4},1){d[3,2]+cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12+13=25=25

Cost(4,{2,3},1)=⎧⎩⎨ d[4,2]+Cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23Cost(4,{2,3},1){d[4,2]+cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23Cost(4,{2,3},1)={d[4,2]+Cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23Cost(4,{2,3},1){d[4,2]+cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23

S = 3

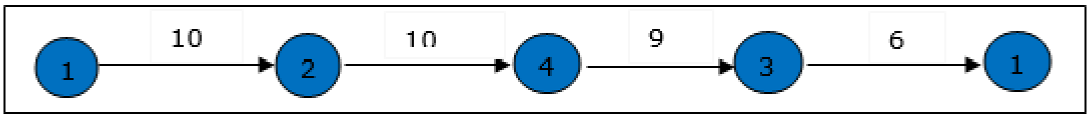
Cost(1,{2,3,4},1)=⎧⎩⎨⎪⎪⎪⎪⎪⎪⎪⎪⎪⎪⎪⎪⎪⎪⎪⎪⎪⎪ d[1,2]+Cost(2,{3,4},1)=10+25=35d[1,3]+Cost(3,{2,4},1)=15+25=40d[1,4]+Cost(4,{2,3},1)=20+23=43=35cost(1,{2,3,4}),1)d[1,2]+cost(2,{3,4},1)=10+25=35d[1,3]+cost(3,{2,4},1)=15+25=40d[1,4]+cost(4,{2,3},1)=20+23=43=35Cost(1,{2,3,4},1)={d[1,2]+Cost(2,{3,4},1)=10+25=35d[1,3]+Cost(3,{2,4},1)=15+25=40d[1,4]+

Cost(4,{2,3},1)=20+23=43=35cost(1,{2,3,4}),1)d[1,2]+cost(2,{3,4},1)=10+25=35d[1,3]+cost(3,{2,4},1)=15+25=40d[1,4]+cost(4,{2,3},1)=20+23=43=35

The minimum cost path is 35.

Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.

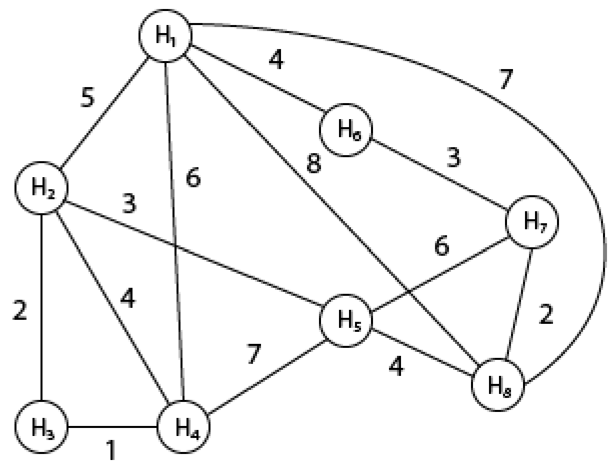
When **s = 1**, we get the minimum value for **d [4, 3]**. Selecting path 4 to 3 (cost is 9), then we shall go to then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).



Suppose the cities are x_1 x_2..... x_n where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesperson problem is to find a route starting and ending at x_1 that will take in all cities with the minimum cost.

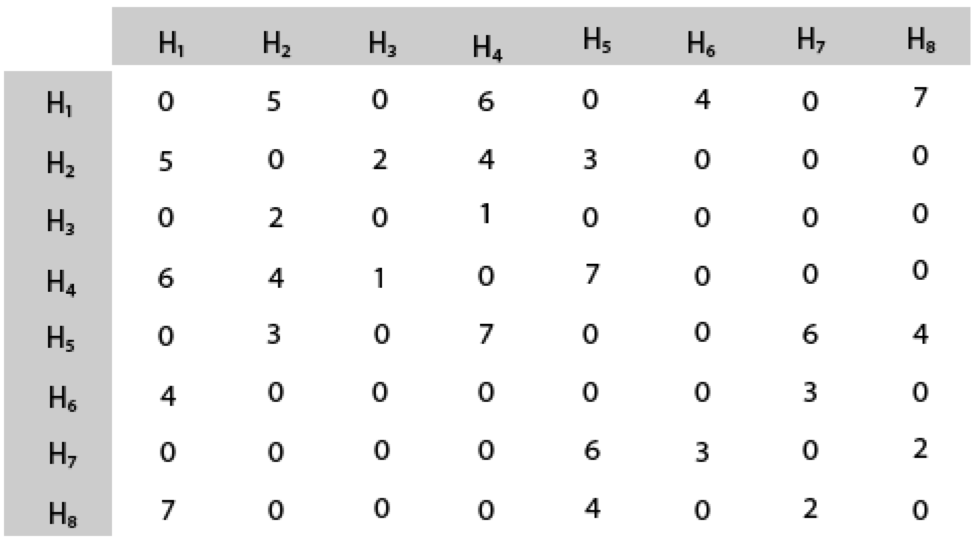
Example: A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.

The area assigned to the agent where he has to drop the newspaper is shown in fig:



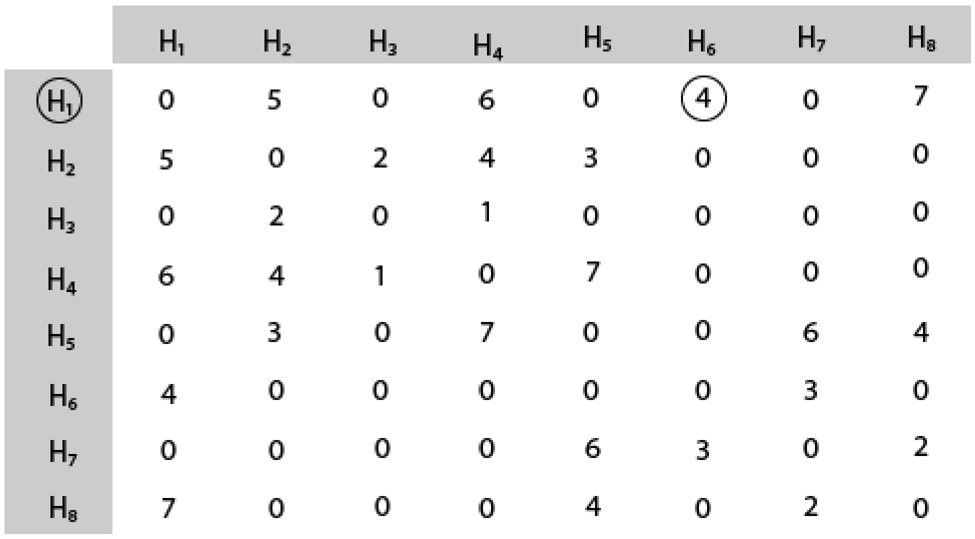
Solution: The cost- adjacency matrix of graph G is as follows:

$cost_{ij}$ =



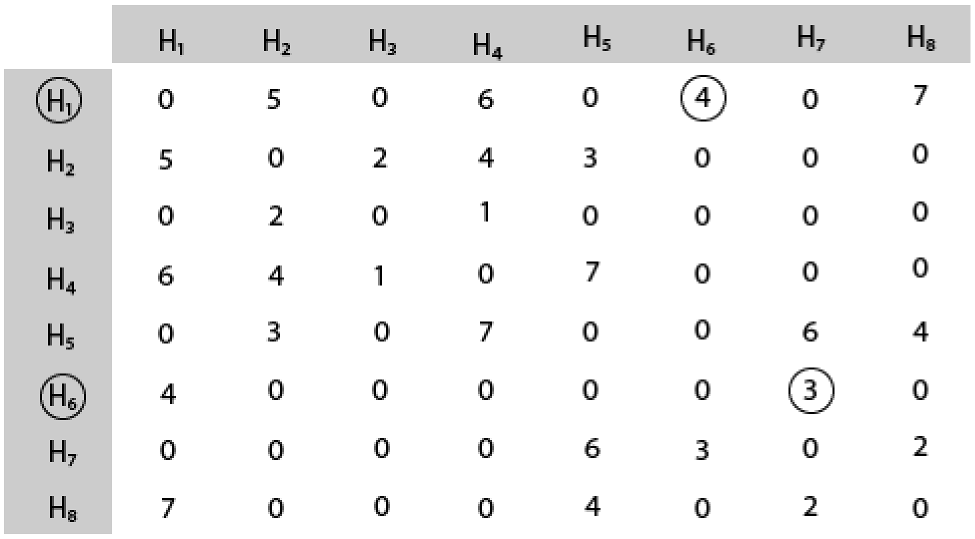
	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

The tour starts from area H_1 and then select the minimum cost area reachable from H_1.



	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_6 because it is the minimum cost area reachable from H_1 and then select minimum cost area reachable from H_6.



	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_7 because it is the minimum cost area reachable from H_6 and then select minimum cost area reachable from H_7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
H_8	7	0	0	0	4	0	2	0

Mark area H_8 because it is the minimum cost area reachable from H_8.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
$\textcircled{H_8}$	7	0	0	0	$\textcircled{4}$	0	2	0

Mark area H_5 because it is the minimum cost area reachable from H_5.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
$\textcircled{H_5}$	0	$\textcircled{3}$	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
$\textcircled{H_8}$	7	0	0	0	$\textcircled{4}$	0	2	0

Mark area H_2 because it is the minimum cost area reachable from H_2.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
$\textcircled{H_2}$	5	0	$\textcircled{2}$	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
$\textcircled{H_5}$	0	$\textcircled{3}$	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
$\textcircled{H_8}$	7	0	0	0	$\textcircled{4}$	0	2	0

Mark area H_3 because it is the minimum cost area reachable from H_3.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
$\textcircled{H_1}$	0	5	0	6	0	$\textcircled{4}$	0	7
$\textcircled{H_2}$	5	0	$\textcircled{2}$	4	3	0	0	0
$\textcircled{H_3}$	0	2	0	$\textcircled{1}$	0	0	0	0
H_4	6	4	1	0	7	0	0	0
$\textcircled{H_5}$	0	$\textcircled{3}$	0	7	0	0	6	4
$\textcircled{H_6}$	4	0	0	0	0	0	$\textcircled{3}$	0
$\textcircled{H_7}$	0	0	0	0	6	3	0	$\textcircled{2}$
$\textcircled{H_8}$	7	0	0	0	$\textcircled{4}$	0	2	0

Mark area H_4 and then select the minimum cost area reachable from H_4 it is H_1.So, using the greedy strategy, we get the following.

4 3 2 4 3 2 1 6

$H_1 \to H_6 \to H_7 \to H_8 \to H_5 \to H_2 \to H_3 \to H_4 \to H_1$.

Thus the minimum travel cost = 4 + 3 + 2 + 4 + 3 + 2 + 1 + 6 = 25

Fractional Knapsack

Fractions of items can be taken rather than having to make binary (0-1) choices for each item. Fractional Knapsack Problem can be solvable by greedy strategy whereas 0 - 1 problem is not.

Steps to solve the Fractional Problem:

1. Compute the value per pound v_i/w_i for each item.
2. Obeying a Greedy Strategy, we take as possible of the item with the highest value per pound.
3. If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound.
4. Sorting, the items by value per pound, the greedy algorithm run in O (n log n) time.

Algorithm

```
Fractional Knapsack (Array v, Array w, int W)
    • for i= 1 to size (v)
    • do p [i] = v [i] / w [i]
    • Sort-Descending (p)
    • i ← 1
    • while (W>0)
    • do amount = min (W, w [i])
    • solution [i] = amount
    • W= W-amount
    • i ← i+1
    • return solution
```

Example: Consider 5 items along their respective weights and values: -

$I = (I_1,I_2,I_3,I_4,I_5)$

w = (5, 10, 20, 30, 40)

v = (30, 20, 100, 90,160)

The capacity of knapsack W = 60

Now fill the knapsack according to the decreasing value of p_i.

First, we choose the item I_i whose weight is 5.

Then choose item I_3 whose weight is 20. Now,the total weight of knapsack is 20 + 5 = 25

Now the next item is I_5, and its weight is 40, but we want only 35, so we chose the fractional part of it, i.e., $5 \times \frac{5}{5} + 20 \times \frac{20}{20} + 40 \times \frac{35}{40}$

Weight = 5 + 20 + 35 = 60

Maximum Value:-

$30 \times \frac{5}{5} + 100 \times \frac{20}{20} + 160 \times \frac{35}{40}$

= 30 + 100 + 140 = 270 (Minimum Cost)

Solution:

ITEM	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio i.e. $p_i = \frac{v_i}{w_i}$

ITEM	wi	vi	$Pi = \frac{v_i}{w_i}$
I1	**5**	**30**	**6.0**
I2	**10**	**20**	**2.0**
I3	**20**	**100**	**5.0**
I4	**30**	**90**	**3.0**
I5	**40**	**160**	**4.0**

Now, arrange the value of p_i in decreasing order.

ITEM	w_i	v_i	$p_i = \frac{v_i}{w_i}$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Differentiate between Dynamic Programming and Greedy Method

Dynamic Programming	Greedy Method
1. Dynamic Programming is used to obtain the optimal solution.	1. Greedy Method is also used to get the optimal solution.
2. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems.	2. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made.
3. Less efficient as compared to a greedy approach	3. More efficient as compared to a greedy approach
4. Example: 0/1 Knapsack	4. Example: Fractional Knapsack
5. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality.	5. In Greedy Method, there is no such guarantee of getting Optimal Solution.

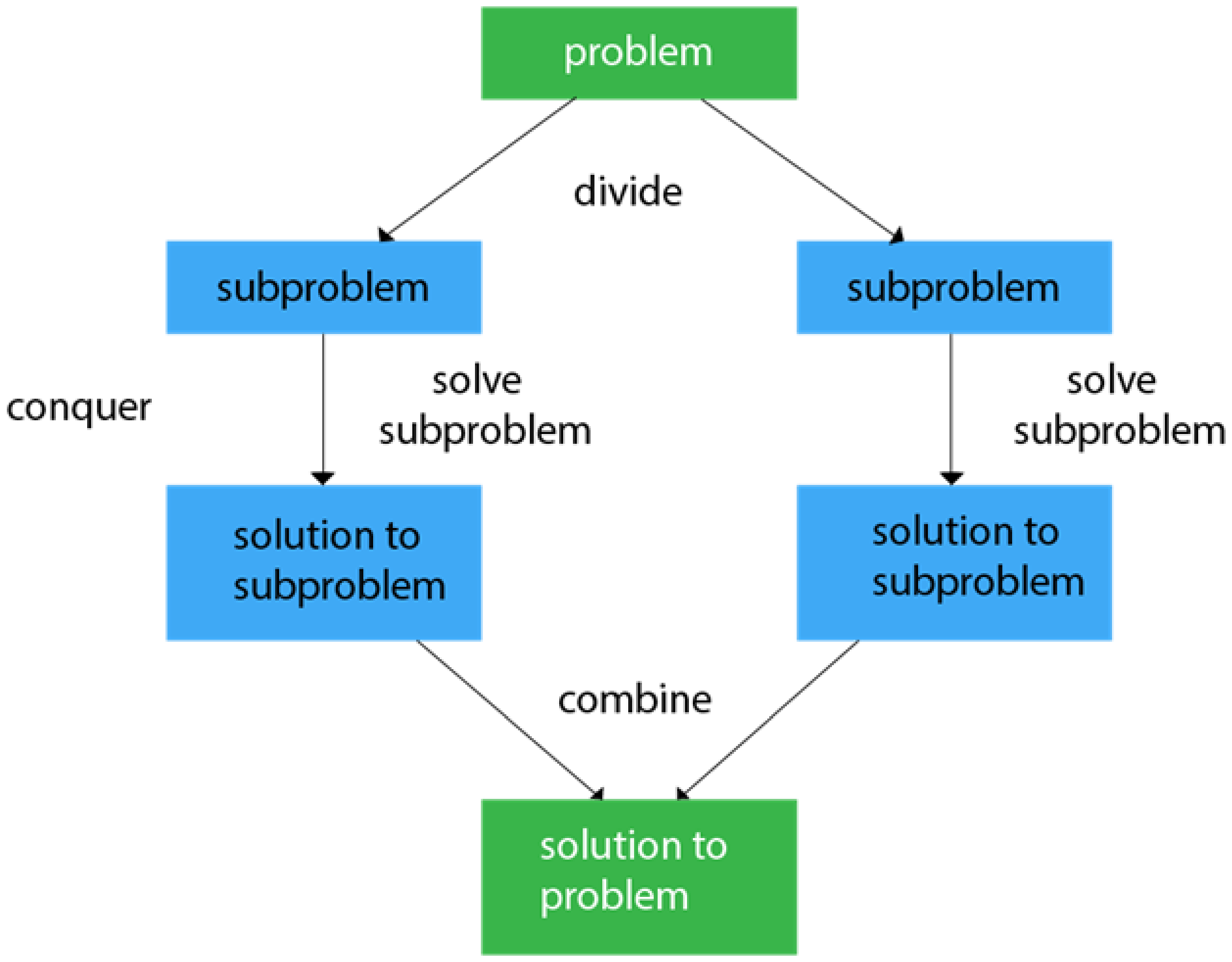
Divide and Conquer Algorithm

Introduction

Divide and Conquer Approach: It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps: Divide and Conquer is an algorithmic paradigm. Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)

4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

Divide and Conquer Approach is divided into three processes and these three processes form the basis of divide and conquer paradigm for problem-solving:

1) Divide

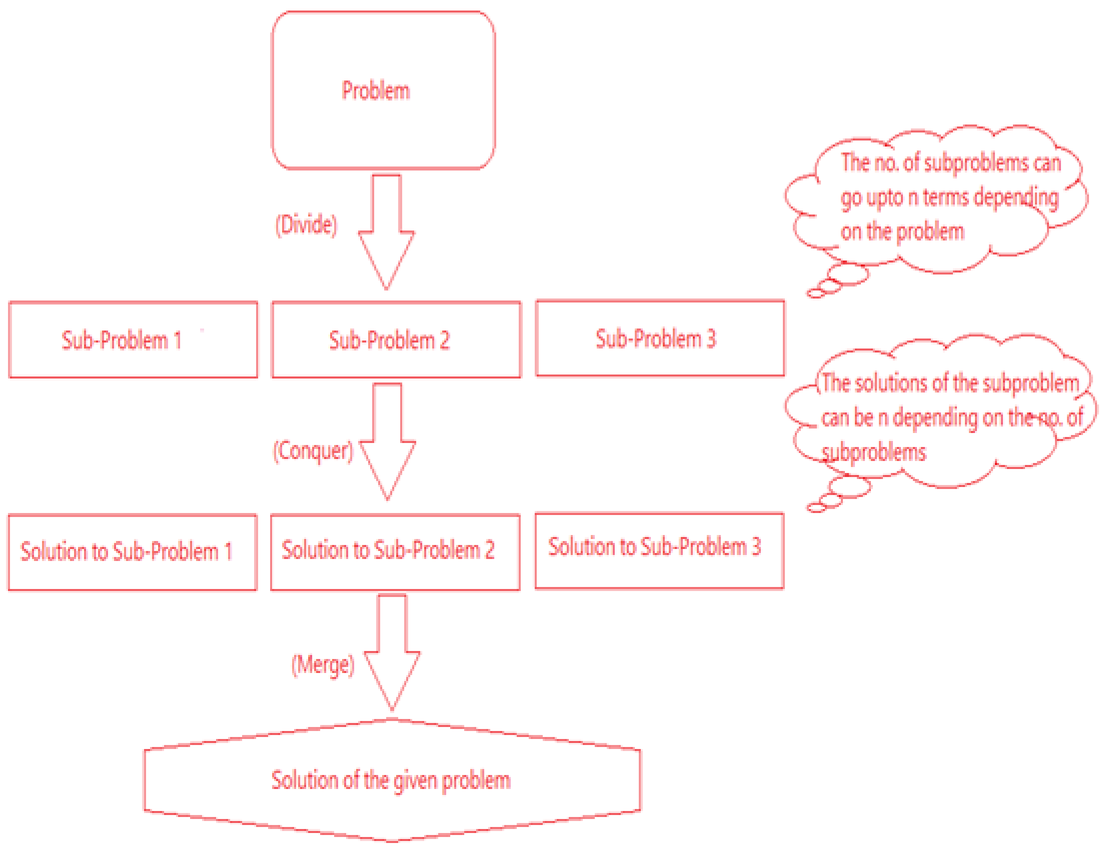
The first and foremost process for solving any problem using Divide and Conquer paradigm. As suggested by the name it’s function is just to divide the problem into sub-problem which in turn if are more complex then are again divided into more sub-parts. Basically , if we consider for example binary search (an example of Divide and Conquer approach) the given list is broken (under some specified consider defined by its algorithm and user input) into single elements among which then the element to search is compared and the user gets a prompt whether the element is in the list.

2) Conquer

This process is referred to as ‘ Conquer ’ because this process is what which performs the basic operation of a defined algorithm like sort in cases of various sorts, finds the element to be searched in case of binary search, multiplying of the numbers in Karatsuba Algorithm and etc. But almost among algorithms at least the most basic ones that we study mostly are considered to be solved in this part since the divide part breaks them into single elements which can be simply solved.

3) Merge

This is the last process of the **'Divide'** and **'Conquer'** approach whose function is to recursively rebuild the original problem but what we get now is the solution to that problem. In merge procedure, the solved elements of the ‘ Conquer ‘ are recursively merged together like the way they divided in the first step i.e. ‘ Divide’.


Problem
(Divide)
Sub-Problem 1
Sub-Problem 2
Sub-Problem 3
The no. of subproblems can go upto n terms depending on the problem
(Conquer)
The solutions of the subproblem can be n depending on the no. of subproblems
Solution to Sub-Problem 1
Solution to Sub-Problem 2
Solution to Sub-Problem 3
(Merge)
Solution of the given problem

The steps **'Conquer'** and **'Merge'** work so close that sometimes they are treated as a single step. The Divide and Conquer can be implemented in two ways:

1. **Naturally** i.e. by using recursion
2. **Explicitly** i.e. by using data structures like stack and queues etc

One of the major **characteristics of Divide and Conquer** is that the time complexity of its algorithms can be easily calculated by solving recurrence relations and its correctness can be evaluated via Mathematical Induction.

Applications of Divide and Conquer algorithm

Following are some standard algorithms that are Divide and Conquer algorithms.

1) Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.

2) Quicksort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3) Merge Sort is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4) Closest Pair of Points The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(nLogn) time.

5) Strassen's Algorithm is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3). Strassen's algorithm multiplies two matrices in O(n^2.8974) time.

6) Cooley–Tukey Fast Fourier Transform (FFT) algorithm is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(nlogn) time.

7) Karatsuba algorithm for fast multiplication it does multiplication of two *n*-digit numbers in at most $3n^{\log_2 3} \approx 3n^{1.585}$ single-digit multiplications in general (and exactly $n^{\log_2 3}$ when *n* is a power of 2). It is therefore faster than the classical algorithm, which requires n^2 single-digit products. If $n = 2^{10} = 1024$, in particular, the exact counts are $3^{10} = 59{,}049$ and $(2^{10})^2 = 1{,}048{,}576$, respectively.

Advantages:

- In a perfect world, where the problem is easy to divide, and the sub-problem at some level is easy to solve, divide and conquer can be optimal for a general case solution, like merge sort.
- Parallel availability, divide and conquer by it's very nature lends itself well to parallel processing.
- The most recognizable benefit of the divide and conquer paradigm is that it allows us to solve difficult problem, such as the Tower of Hanoi, which is a mathematical game or puzzle. Being given a difficult problem can often be discouraging if there is no idea how to go about solving it. However, with the divide and conquer method, it reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable, and usually runs faster than other algorithms would.
- It also uses memory caches effectively. When the sub problems become simple enough, they can be solved within a cache, without having to access the slower main memory.
- Solves difficult problems with ease.
- Algorithms designed with Divide and Conquer strategies are efficient when compared to its counterpart Brute-Force approach for e.g. if we compare time complexity of simple matrix multiplication i.e. O(n3) and that of Strassen's matrix multiplication i.e. O(n2.81).
- Algorithms designed under Divide and Conquer strategy does not require any modification as they inhibit parallelism and can easily be processed by parallel processing systems.
- It makes efficient use of memory cache this happens so because that problems when divided gets so small that they can be easily solved in the cache itself.
- If we use floating numbers then the results may improve since round off control is very efficient in such algorithms.

Disadvantages:

- Problem decomposition may be very complex and thus not really suitable to divide and conquer.
- Recursive nature of the solution may end up duplicating sub-problems, dynamic/memoized solutions may be better in some of these cases, like Fibonacci.
- Recursion into small/tiny base cases may lead to huge recursive stacks, and efficiency can be lost by not applying solutions earlier for larger base cases.
- One of the most common issues with this sort of algorithm is the fact that the recursion is slow, which in some cases outweighs any advantages of this divide and conquer process.
- Sometimes it can become more complicated than a basic iterative approach, especially in cases with a large n.
- Divide and Conquer strategy uses recursion that makes it a little slower and if a little error occurs in the code the program may enter into an infinite loop.
- Usage of explicit stacks may make use of extra space.
- If performing a recursion for no. times greater than the stack in the CPU than the system may crash.

- Choosing base cases is also a limitation as picking small cases may work if the cases are taken much higher than the capacity of the system than problems may occur.
- Sometimes a case where the problem when broken down results in same subproblems which may needlessly increase the solving time and extra space may be consumed.

Master Theorem

Masters Theorem for divide and conquer is an analysis theorem that can be used to determine a big-0 value for recursive relation algorithms. It is used to find the time required by the algorithm and represent it in asymptotic notation form. Master's Theorem is a popular method for solving the recurrence relations.

Example of runtime value of the problem in the above example −

```
T(n) = f(n) + m.T(n/p)
```

For most of the recursive algorithm, you will be able to find the Time complexity For the algorithm using the master's theorem, but there are some cases master's theorem may not be applicable. These are the cases in which the master's theorem is not applicable. When the problem T(n) is not monotone, for example, T(n) = sin n. Problem function f(n) is not a polynomial.

As the master theorem to find time complexity is not hot efficient in these cases, and advanced master theorem for recursive recurrence was designed. It is design to handle recurrence problem of the form −

```
T(n) = aT(n/b) + ø((n^k)logpn)
```

Where n is the size of the problem.

a = number of subproblems in recursion, a > 0

n/b = size of each subproblem defined, n, b > 0

For solving this type of problem, we will use the following solutions,

- If $a>b_k$, then T(n) = Ø (nlogba)
- If $a = b_k$, then
  - If p> -1, then T(n) = Ø(nlogba $\log^{p+1}$n)
  - If p = -1, then T(n) = Ø(nlog$_{ba}$ loglogn)
  - If p < -1, then T(n) = Ø(nlog$_{ba}$)
- If $a<b_k$, then
  - If p> = 0, then T(n)= Ø($n_k$log$_p$n)
  - If p< 0, then T(n) = Ø(nk)

Introduction of Dynamic Programming

- Dynamic Programming is the most powerful design technique for solving optimization problems.
- Divide & Conquer algorithm partition the problem into disjoint subproblems solve the subproblems recursively and then combine their solution to solve the original problems.
- Dynamic Programming is used when the subproblems are not independent, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.
- Dynamic Programming solves each subproblems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.
- Dynamic Programming is a **Bottom-up approach-** we solve all possible small problems and then combine to obtain solutions for bigger problems.
- Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "**principle of optimality**".

Characteristics of Dynamic Programming:

Dynamic Programming works when a problem has the following features:-

- **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
- **Overlapping subproblems:** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has overlapping subproblems.

If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping subproblems, then we can improve on a recursive implementation by computing each subproblem only once.

If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.

If the space of subproblems is enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Elements of Dynamic Programming

There are basically three elements that characterize a dynamic programming algorithm:-

1. **Substructure:** Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.
2. **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.
3. **Bottom-up Computation:** Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrives at a solution to complete problem.

Note: Bottom-up means:-

i. Start with smallest subproblems.
ii. Combining their solutions obtain the solution to sub-problems of increasing size.

iii. Until solving at the solution of the original problem.

Components of Dynamic programming

1. **Stages:** The problem can be divided into several subproblems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
2. **States:** Each stage has several states associated with it. The states for the shortest path problem was the node reached.
3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
4. **Optimal policy:** It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.
5. Given the current state, the optimal choices for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.
6. There exist a recursive relationship that identify the optimal decisions for stage j, given that stage j+1, has already been solved.
7. The final stage must be solved by itself.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterize the structure of an optimal solution.
2. Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest subproblems)
4. Construct the optimal solution for the entire problem form the computed values of smaller subproblems.

Applications of dynamic programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair Shortest path problem
4. Reliability design problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximize CPU usage

0/1 knapsack problem

Knapsack Problem:

Knapsack is basically means bag. A bag of given capacity.

We want to pack n items in your luggage.

- The ith item is worth v_i dollars and weight w_i pounds.
- Take as valuable a load as possible, but cannot exceed W pounds.

- v_i w_i W are integers.

  - W ≤ capacity
  - Value ← Max

Input:

- Knapsack of capacity
- List (Array) of weight and their corresponding value.

Output: To maximize profit and minimize weight in capacity.
The knapsack problem where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.
Knapsack problem can be further divided into two parts:
1. Fractional Knapsack: Fractional knapsack problem can be solved by **Greedy Strategy** where as 0 /1 problem is not.
It cannot be solved by **Dynamic Programming Approach.**

0/1 Knapsack Problem:

In this item cannot be broken which means thief should take the item as a whole or should leave it. That's why it is called **0/1 knapsack Problem**.

- Each item is taken or not taken.
- Cannot take a fractional amount of an item taken or take an item more than once.
- It cannot be solved by the Greedy Approach because it is enable to fill the knapsack to capacity.
- **Greedy Approach** doesn't ensure an Optimal Solution.

Example of 0/1 Knapsack Problem:

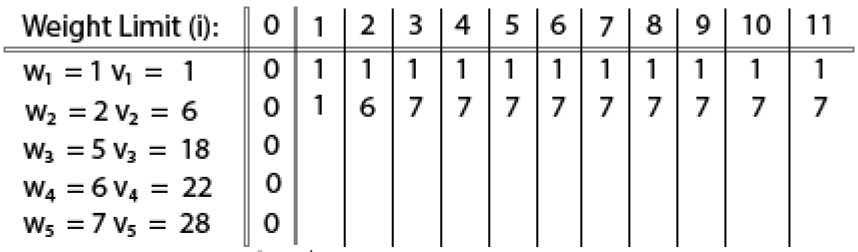
Example: The maximum weight the knapsack can hold is W is 11. There are five items to choose from. Their weights and values are presented in the following table:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$												
$w_2 = 2$ $v_2 = 6$												
$w_3 = 5$ $v_3 = 18$												
$w_4 = 6$ $v_4 = 22$												
$w_5 = 7$ $v_5 = 28$												

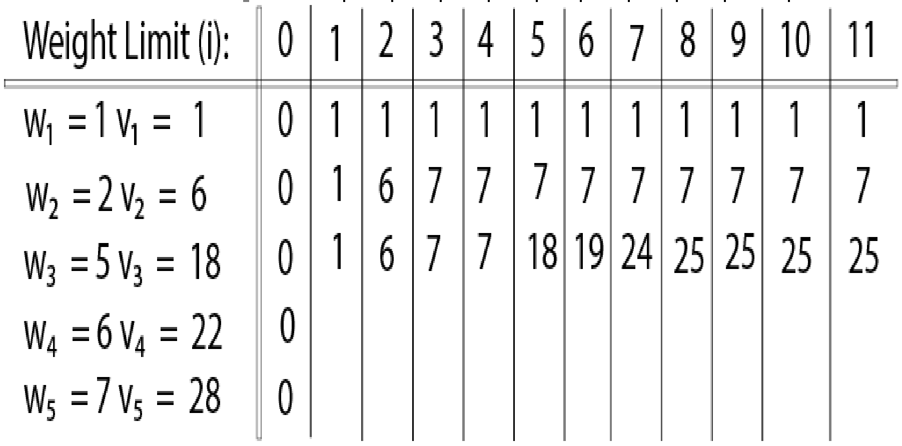
The [i, j] entry here will be V [i, j], the best value obtainable using the first "i" rows of items if the maximum capacity were j. We begin by initialization and first row.

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0											
$w_3 = 5$ $v_3 = 18$	0											
$w_4 = 6$ $v_4 = 22$	0											
$w_5 = 7$ $v_5 = 28$	0											

$V[i, j] = \max \{V[i - 1, j], v_i + V[i - 1, j - w_i]$



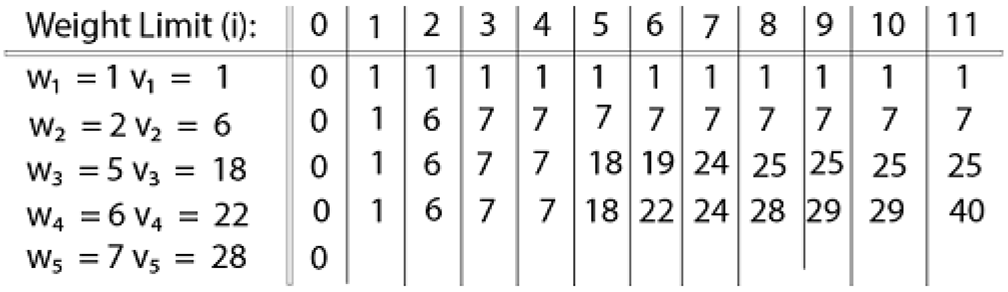
Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											



Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

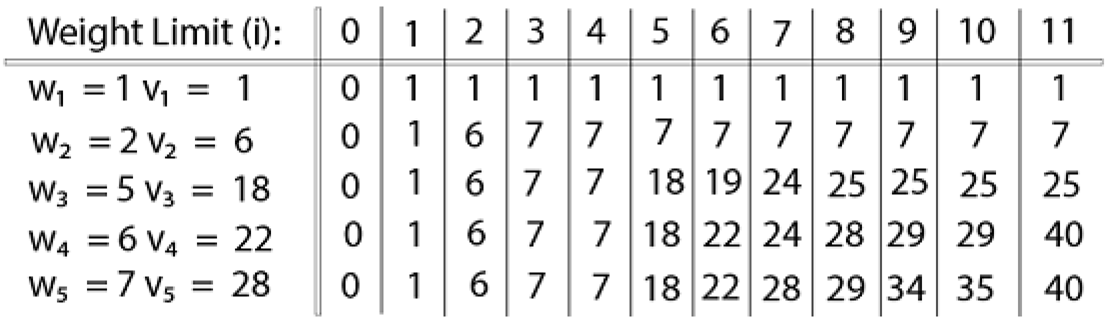
The value of V [3, 7] was computed as follows:

V [3, 7] = max {V [3 - 1, 7], v_3 + V [3 - 1, 7 - w_3]
= max {V [2, 7], 18 + V [2, 7 - 5]}
= max {7, 18 + 6}
= 24



Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0											

Finally, the output is:



Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

The maximum value of items in the knapsack is 40, the bottom-right entry). The dynamic programming approach can now be coded as the following algorithm:

Algorithm of Knapsack Problem

KNAPSACK (n, W)

1. for w = 0, W

```
2. do V [0,w] ← 0
3. for i=0, n
4. do V [i, 0] ← 0
5. for w = 0, W
6. do if (wi≤ w & vi + V [i-1, w -  wi]> V [i -1,W])
7. then V [i, W] ← vi + V [i - 1, w - wi]
8. else V [i, W] ← V [i - 1, w]
```

Longest common subsequence (LCS)

A subsequence of a given sequence is just the given sequence with some elements left out. Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

In the longest common subsequence problem, we are given two sequences X = (x_1 x_2....x_m) and Y = (y_1 y_2 y_n) and wish to find a maximum length common subsequence of X and Y. LCS Problem can be solved using dynamic programming.

Characteristics of Longest Common Sequence

A brute-force approach we find all the subsequence's of X and check each subsequence to see if it is also a subsequence of Y, this approach requires exponential time making it impractical for the long sequence.

Given a sequence X = (x_1 x_2.....x_m) we define the ith prefix of X for i=0, 1, and 2...m as X_i= (x_1 x_2.....x_i). For example: if X = (A, B, C, B, C, A, B, C) then X_4= (A, B, C, B)

Step1: Optimal Substructure of an LCS: Let X = (x_1 x_2....x_m) and Y = (y_1 y_2.....) y_n) be the sequences and let Z = (z_1 z_2......z_k) be any LCS of X and Y.

- If $x_m = y_n$, then z_k=x_m=y_n and Z_{k-1} is an LCS of $X_{m-1}$and Y_{n-1}
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of $X_{m-1}$and Y.
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Step 2: Recursive Solution: LCS has overlapping subproblems property because to find LCS of X and Y, we may need to find the LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we must solve two subproblems finding an LCS of X and Y_{n-1}.Whenever of these LCS's longer is an LCS of x and y. But each of these subproblems has the subproblems of finding the LCS of X_{m-1} and Y_{n-1}.

Let c [i,j] be the length of LCS of the sequence $X_i$and Y_j.If either i=0 and j =0, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem given the recurrence formula

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

Step3: Computing the length of an LCS: let two sequences X = (x_1 x_2.....x_m) and Y = (y_1 y_2..... y_n) as inputs. It stores the c [i,j] values in the table c [0......m,0..........n].Table b [1..........m, 1..........n] is maintained which help us to construct an optimal solution. c [m, n] contains the length of an LCS of X,Y.

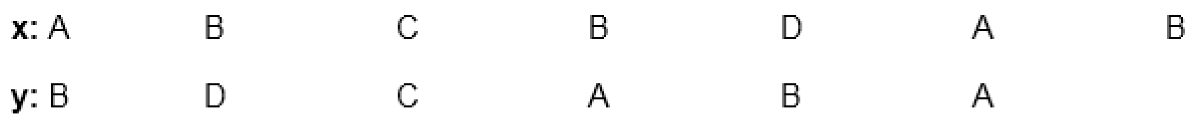
Algorithm of Longest Common Sequence

LCS-LENGTH (X, Y)

```
1. m ← length[X]
2. n ← length [Y]
3. for i ← 1 to m
4. do c [i,0] ← 0
5. for j ← 0 to m
6. do c [0,j] ← 0
7. for i ← 1 to m
8. do for j ← 1 to n
9. do if $x_i = y_j$
10. then c [i,j] ← c [i-1,j-1] + 1
11. b [i,j] ← "↖"
12. else if c[i-1,j] ≥ c[i,j-1]
13. then c [i,j] ← c [i-1,j]
14. b [i,j] ← "↑"
15. else c [i,j] ← c [i,j-1]
16. b [i,j] ← "← "
17. return c and b.
```

Example of Longest Common Sequence

Example: Given two sequences X [1...m] and Y [1.....n]. Find the longest common subsequences to both.



x: A	B	C	B	D	A	B
y: B	D	C	A	B	A	

here X = (A,B,C,B,D,A,B) and Y = (B,D,C,A,B,A)

m = length [X] and n = length [Y]

m = 7 and n = 6

Here x_1= x [1] = A y_1= y [1] = B

x_2= B y_2= D

x_3= C y_3= C

x_4= B y_4= A

x_5= D y_5= B

x_6= A y_6= A

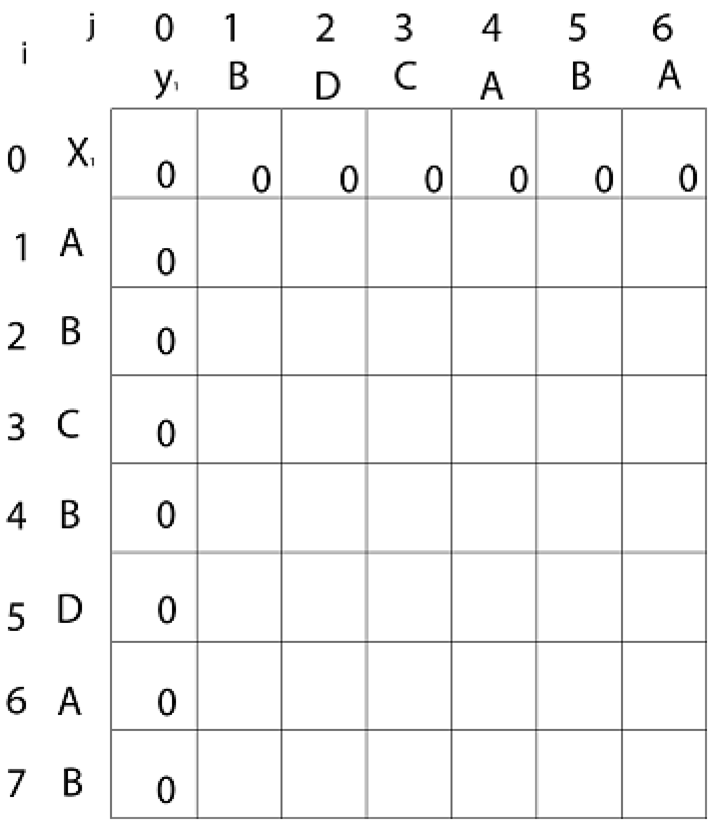
x_7= B

Now fill the values of c [i, j] in m x n table

Initially, for i=1 to 7 c [i, 0] = 0

For j = 0 to 6 c [0, j] = 0

That is:



i	j	0	1	2	3	4	5	6
		y_i	B	D	C	A	B	A
0	X_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Now for i=1 and j = 1

x_1 and y_1 we get $x_1 \neq y_1$ i.e. $A \neq B$

And c [i-1,j] = c [0, 1] = 0

c [i, j-1] = c [1,0] = 0

That is, c [i-1,j]= c [i, j-1] so c [1, 1] = 0 and b [1, 1] = ' ↑ '

Now for i=1 and j = 2

x_1 and y_2 we get $x_1 \neq y_2$ i.e. $A \neq D$

c [i-1,j] = c [0, 2] = 0

c [i, j-1] = c [1,1] = 0

That is, c [i-1,j]= c [i, j-1] and c [1, 2] = 0 b [1, 2] = ' ↑ '

Now for i=1 and j = 3

x_1 and y_3 we get $x_1 \neq y_3$ i.e. $A \neq C$

c [i-1,j] = c [0, 3] = 0

c [i, j-1] = c [1,2] = 0

so c [1,3] = 0 b [1,3] = ' ↑ '

Now for i=1 and j = 4

x_1 and y_4 we get. $x_1=y_4$ i.e A = A

c [1,4] = c [1-1,4-1] + 1

= c [0, 3] + 1

= 0 + 1 = 1

c [1,4] = 1

b [1,4] = ' ↖ '

Now for i=1 and j = 5

x_1 and y_5 we get $x_1 \neq y_5$

c [i-1,j] = c [0, 5] = 0

c [i, j-1] = c [1,4] = 1

Thus c [i, j-1] > c [i-1,j] i.e. c [1, 5] = c [i, j-1] = 1. So b [1, 5] = '⟵'

Now for i=1 and j = 6

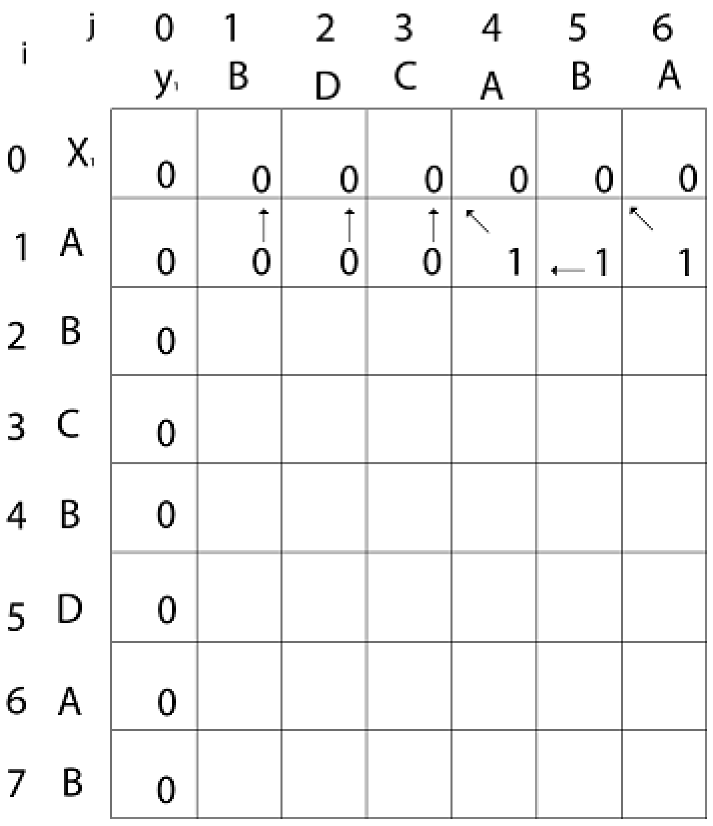
x_1 and y_6 we get $x_1 = y_6$

c [1, 6] = c [1-1,6-1] + 1

= c [0, 5] + 1 = 0 + 1 = 1

c [1,6] = 1

b [1,6] = ' ↖ '



i	j	0	1	2	3	4	5	6
		y_i	B	D	C	A	B	A
0	X_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	⟵ 1	↖ 1
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Now for i=2 and j = 1

We get x_2 and y_1 B = B i.e. $x_2 = y_1$

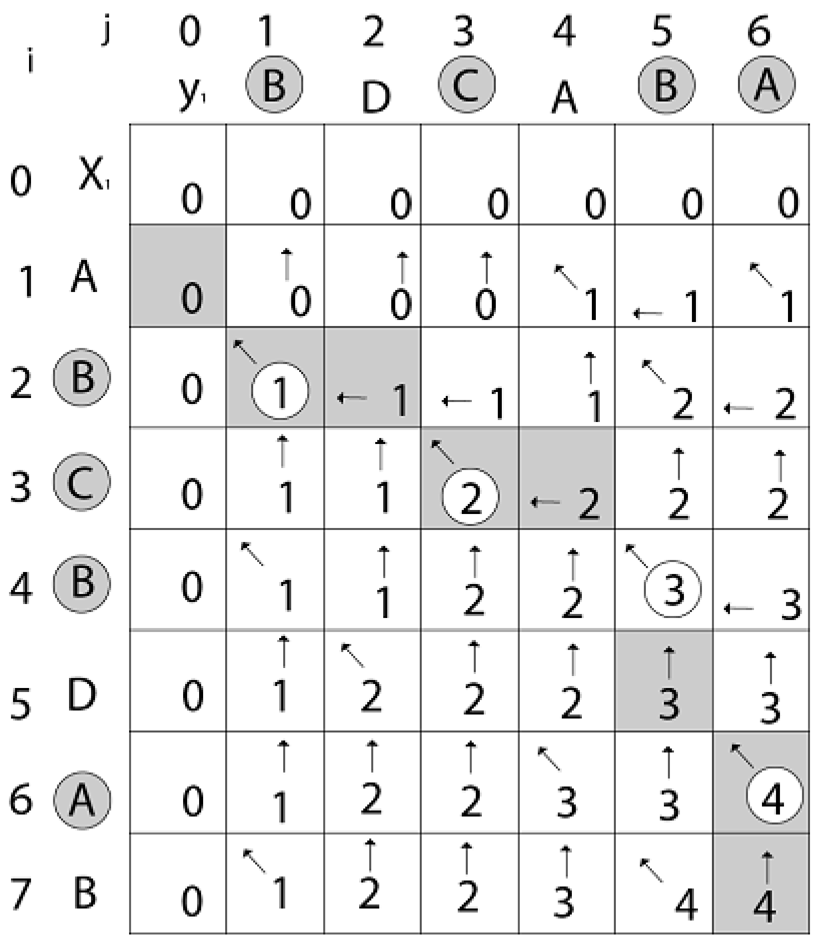
c [2,1] = c [2-1,1-1] + 1

= c [1, 0] + 1

= 0 + 1 = 1

c [2, 1] = 1 and b [2, 1] = ' ↖ '

Similarly, we fill the all values of c [i, j] and we get



i	j	0	1	2	3	4	5	6
		y_i	B	D	C	A	B	A
0	X_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Step 4: Constructing an LCS: The initial call is PRINT-LCS (b, X, X.length, Y.length)

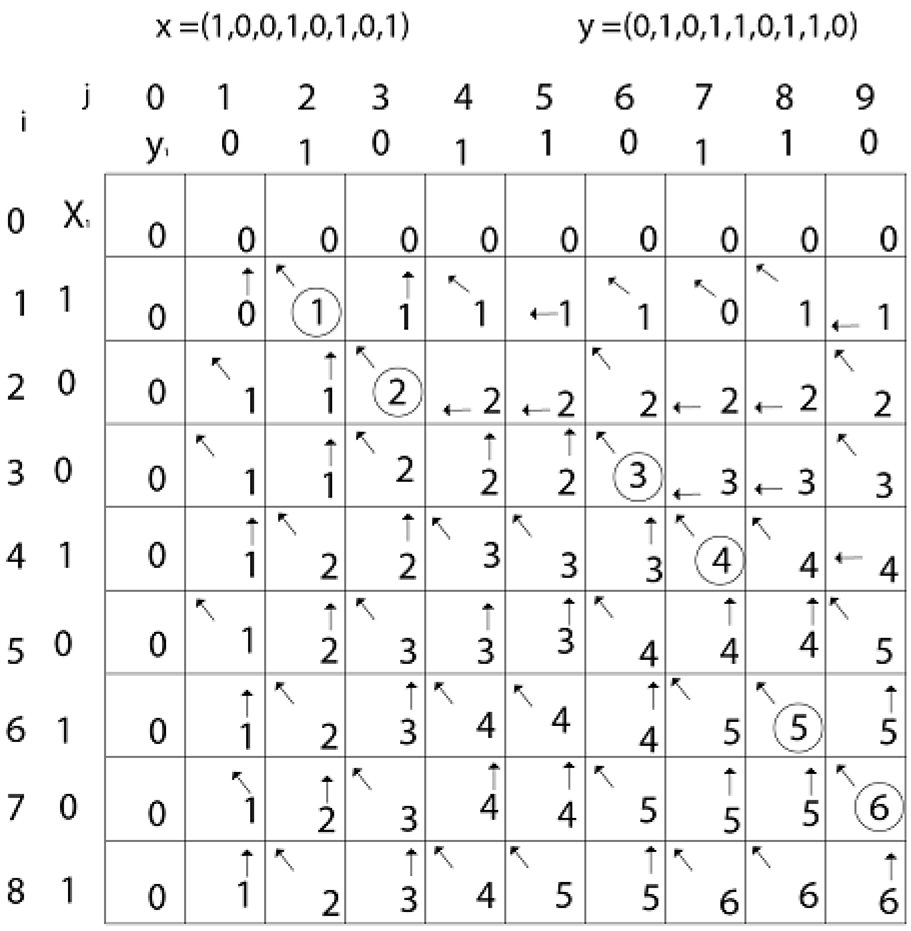
```
PRINT-LCS (b, x, i, j)
1. if i=0 or j=0
2. then return
3. if b [i,j] = ' ↖ '
4. then PRINT-LCS (b,x,i-1,j-1)
5. print x_i
6. else if b [i,j] = ' ↑ '
7. then PRINT-LCS (b,X,i-1,j)
8. else PRINT-LCS (b,X,i,j-1)
```

Example: Determine the LCS of (1,0,0,1,0,1,0,1) and (0,1,0,1,1,0,1,1,0).

Solution: let X = (1,0,0,1,0,1,0,1) and Y = (0,1,0,1,1,0,1,1,0).

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

We are looking for c [8, 9]. The following table is built.



x =(1,0,0,1,0,1,0,1) y =(0,1,0,1,1,0,1,1,0)

i	j	0	1	2	3	4	5	6	7	8	9
	y_i		0	1	0	1	1	0	1	1	0
0	X_i	0	0	0	0	0	0	0	0	0	0
1	1	0	↑ 0	↖ ①	↑ 1	↖ 1	← 1	↖ 1	↖ 0	↖ 1	← 1
2	0	0	↖ 1	↑ 1	↖ ②	← 2	← 2	↖ 2	← 2	← 2	↖ 2
3	0	0	↖ 1	↑ 1	↖ 2	↑ 2	↑ 2	↖ ③	← 3	← 3	↖ 3
4	1	0	↑ 1	↖ 2	↑ 2	↖ 3	↖ 3	↑ 3	↖ ④	↖ 4	← 4
5	0	0	↖ 1	↑ 2	↖ 3	↑ 3	↑ 3	↖ 4	↑ 4	↑ 4	↖ 5
6	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 4	↑ 4	↖ 5	↖ ⑤	↑ 5
7	0	0	↖ 1	↑ 2	↖ 3	↑ 4	↑ 4	↖ 5	↑ 5	↑ 5	↖ ⑥
8	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 5	↑ 5	↖ 6	↖ 6	↑ 6

From the table we can deduct that LCS = 6. There are several such sequences, for instance (1,0,0,1,1,0) (0,1,0,1,0,1) and (0,0,1,1,0,1)