Background: Introduction, algorithms specification, time and space complexity, performance analysis.

Introduction

The name algorithm is given after name of Abu Jafar Muhammad Ibn Musa Ai Khwarizml,9th century is defined as follows:

- An algorithm is set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operation performed on data that have to be organization in data structures
- An algorithm is a step by step procedure to solve a problem. In natural language, the algorithm is defined as a sequence of statement which is used to perform a task.
- An algorithm is a sequence of unambiguous instructions for solving a computational problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time. This definition can be illustrated by a simple diagram Figure.

The notion of algorithm.

- An algorithm is a finite sequence of logically related instructions to solve a computational problem. The following problems are well defined and are considered to be computational problems. - Given an integer x, test whether x is prime or not. - Given a program P, check whether P runs into an infinite loop.
- Any algorithm must have an input provided by the user and must produce an output.
- Computational instructions in the algorithm must involve only basic algebraic (arithmetic) operations and it should terminate after a finite number of steps.
- Finally, any algorithm must involve unambiguous instructions and produce the desired output. Mathematically, an algorithm can be represented as a function, $F : I \rightarrow O$, where I is the set of inputs and O is the set of outputs generated by the algorithm.
- The word algorithm comes from the name of Persian author "Abu Jafar Mohammad ibn Musa al Khawarizmi"..

Every algorithm must satisfy the following criteria: or characteristics of algorithm

1. **Input:** An algorithm must have 0 or well defined.

2. **Output:** An algorithm must have 1 or well defined output and should match with the desired output.
3. **Unambiguous Definiteness**: Each instruction must be clear and unambiguous.
4. **Finiteness**: If we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps.
5. **Effectiveness:** Every instruction must be sufficiently basic that it can in principle be carried out by a person. It is not enough that each operation be definite, but it must also be feasible.
6. **Feasible:** It must be feasible enough to produce each instruction.
7. **Flexibility:** It must be flexible enough to carry out desired changes with no efforts.
8. **Efficient:** The term efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space meeting the acceptable limit of development time.
9. **Independent:** An algorithm must be language independent, which means that it should mainly focus on the input and the procedure required to derive the output instead of depending upon the language.

Algorithm Specification:

1. **Name of algorithm:** Every algorithm is given an identifying name written in capital letters.
2. **Introductory comment:** The algorithm name is followed by a brief description of the tasks the algorithm performs and any assumptions that have been made. The description gives the name and types of the variables used in the algorithm. Comment begins with // and continues until end of line. Comments specify no action and are included only for clarity.
3. **Steps:** Each algorithm is made of a sequence of numbered steps and each step has an ordered sequence of statements, which describe the tasks to be performed. The statements in each step are executed in a left-to-right order.
4. **Data type:** Data types are assumed simple such as integer, real, char, boolean and other data structures such as array, pointer, structure are used. Array i^{th} element can be described as A[i] and (i, j) element can be described as A[i, j]. Structure data type can be formed as

```
node =record
{
        data type-1      identifier 1;
        ...
        data type-n      identifier n;
        node     *link;
}
```

link is a pointer to the record type node.

5. **Assignment statement:** The assignment statement is indicated by placing equal (=) between the variable (in left hand side) and expression or value(s) (in right hand side). For example, addition of a and b is assigned in variable a.

   a = a + b

6. **Expression:** There are three expressions: arithmetic, relational and logical. The arithmetic expression used arithmetic operator such as /, *, +, –, relational expression used relational operator such as <, <=, >, >=, <>, = and logical expression used logical operator such as not, or, and. There are two boolean values, true or false.

7. **If statement:** If statement has one of the following two forms:

   (*a*) if condition
   then
   statement(s)

   (*b*) if condition
   then
   statement(s)
   else
   statement(s)

   if the condition is true, statement(s) followed then are to be executed, otherwise if condition is false, statement(s) followed else are to be executed. Here we can use multiple statements, if required.

8. **Case statement:** In general case statement has the form:

   Select case (expression)

   case value 1: statement (s)
   case value 2: statement (s)
   .
   .
   .
   case value n: statement (s)
   default:

   the expression is evaluated then a branch is made to the appropriate case. If the value of the expression does not match that of any case, then a branch is made to the default case.

9. **Looping statements:** These statements are used when there is a need of some statements to be executed number of times. These statements also called iterative control statements.

(*a*)
```
while (condition) do
{
    statement(s)
}
```

As long the condition is true, the statement(s) within while loop are executed. When the condition tests to false then while loop is exited.

(*b*)
```
for variable = start value to final value step increment value do
{
    statement(s)
}
```

This for loop is executed from start value to final value with addition of increment value in start value. In the absence of step clause, assume increment value 1.

(*c*)
```
repeat
{
statement(s)
} until (condition)
```

This repeat loop is executed until the condition is false, and exited when condition becomes true.

10. **Input and output:** For input of data it used read(variable name) statement and for output of data it used write(variable name) statement. If more than one input or output data then we can use comma as separator among the variable names.
11. **Goto statement:** The goto statement causes unconditional transfer of control to the step referenced. Thus statement goto step N will cause transfer of control to step N.
12. **End statement:** The end statement is used to terminate an algorithm. It is usually the last step and algorithm name is written after the end.
13. **Functions:** A function is used to return single value to the calling function. Transfer of control and returning of the value are accomplished by return(value) statement. A function begins as follows: function function_name (parameters list).

Algorithm vs program

Algorithm	Program
It is step by step process for solving computational problem.	A Program is nothing but set of instruction
An algorithm is designed by domain knowledge(designer)	The program can be implemented by specific programming
An algorithm is done at design time	Program is done at implement time
Any language example mathematical notation, general English language	Any programming language e.g.: c,c++,java
After completion of your algorithm you have to analyze based on some criteria such as time and space	After completion of writing your program you have to do test your program

Algorithm Design Techniques:

For a given problem, there are many ways to design algorithm for it. The following is a list of several popular design approaches.

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

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.

2. **Greedy method:**

An optimization problem is one in which we want to find not just a solution but the best solution. A greedy algorithm sometimes works well for optimization problem a greedy algorithm works in phases:

- We take the best we can get right now, without regard for future consequence

- We hope that by choosing a local optimum at each step, we will end up at a global optimum.

The result is a good solution but not necessarily the best one. The greedy algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal.

Examples: The specific computer algorithms are based on the Greedy approach: Kruskal's algorithm and Prim's algorithm for finding minimum spanning tree.

3. Dynamic programming:

Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

Overlapping Sub-Problems

- Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly.
- The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.
- **For example**, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

Optimal Sub-Structure

- A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.
- **For example**, the Shortest Path problem has the following optimal substructure property −If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

4. Backtracking:

- Backtracking is one of the techniques that can be used to solve the problem. It uses the **Brute force search** to solve the problem, and the brute force search says that for

the given problem, try to make all the possible solutions and pick out the best solution from all the desired solutions.

- This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems.
- In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.
- Backtracking name itself suggests that we are **going back and coming forward**; if it satisfies the condition, then return success, else we go back again.
- It uses **DFS** to generate **state space tree solution**.

The terms related to the backtracking are:

- **Live node:** The nodes that can be further generated are known as live nodes.
- **E node:** The nodes whose children are being generated and become a success node.
- **Success node:** The node is said to be a success node if it provides a feasible solution.
- **Dead node:** The node which cannot be further generated and also does not provide a feasible solution is known as a dead node.

Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:

- **Implicit constraint:** It is a rule in which how each element in a tuple is related.
- **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

Applications of Backtracking

- N-queen problem
- Sum of subset problem
- Graph coloring
- Hamiliton cycle

5. Branch and Bound:

- Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems.
- if we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.
- **Applications** : 0/1 knapsack problem, traveling salesman problem

<u>Performance analysis of an algorithm</u>:

- An algorithm is said to be efficient and fast, if it take less time to execute and consumes less member space.
- Analysis of algorithms is a prediction of how fast the algorithm works based on problem size.
- When an algorithm is executed it uses the computer's primary memory (RAM) and Central Processing Unit(CPU).
- ·Analyzing the amount of resources (time and space) needed helps to choose a good algorithm among several options to solve the same problem.
- Analysis based on time taken to execute the algorithm is called **Tine Complexity** of the Algorithm.
- Analysis based on the memory required to execute the algorithm is called **Space Complexity** of the algorithm.

- The analysis is done at **two** stages: Performance evaluation can be loosely divided into two major phases: a priori estimates and a posteriori testing.
- **Priori Analysis (before developing the program):** This involves analyzing the given algorithm, for various instances of the problem, mathematically to find its time and space complexities.
- **Posteriori Analysis (after developing the program):** This involves running the program for the given algorithm, for various instances of the problem and tracking down the time and space required by them.
- **Differences between Priori and Posteriori analysis:**

Priori	Posteriori
Program for the algorithm is not needed	Program for the algorithm is needed
It is machine independent	It is machine dependent
It is independent of language and skill of the programmer	It depends on the skill of the person who codes the algorithm and also the language in which the algorithm is coded.
This allows analyzing for any input instance sizes.	This may permit testing for moderate input instance sizes only.

- **Performance or Complexity of an algorithm** is measured by estimation the following two important factors: time complexity and space complexity.

Space Complexity:

The space complexity of a computer program is the amount of memory required by it. It varies with the size of the problem being solved. Or

The count of space for memory require of an algorithm to run is known as **space complexity**

Reasons to study:

- In a multiuser system, we need to have an estimate of the amount of memory required by the programs so that, memory allocation can be done efficiently.
- It is a good programming practice to know in advance, whether the available memory is sufficient for the program to be executed.

Calculation of Space Complexity:

- The space S(p) needed to execute a program depends on several factors (sizes of inputs and outputs, variables and other objects etc.)
- It can be approximated as follows **S(p) = C + SP (I)**

Where,

- 'C' is the fixed space requirement, which is independent of the characteristics of the inputs and outputs. It includes instruction space, space for variable, constants etc.
- SP is instance variable is always size may to go up and down.
- Space complexity **S(p)>=C+SP**
- **Example:** Space Complexity

```
Algorithm Sum(number,size)\\ procedure will produce sum of all numbers provided in 'number'
list
{
    result=0.0;
    for count = 1 to size do            \\will repeat from 1,2,3,4,....size times
        result= result + number[count];
```

return result;
}

- In above example, when calculating the space complexity we will be looking for both fixed and variable components. here we have
- **Fixed components** as 'result','count' and 'size' variable there for total space required is three**(3) words**.
- **Variable components** is characterized as the value stored in 'size' variable (suppose value store in variable 'size 'is 'n'). because this will decide the size of 'number' list and will also drive the for loop. therefore if the space used by size is one word then the total space required by 'number' variable will be 'n'(value stored in variable 'size').
- therefore the space complexity can be written as **Space(Sum) = 3 + n;**

Time Complexity:

- Time complexity is usually the amount of time it takes to run a program as a function of the size of the input. Or the amount of time required an algorithm to run to completion is known as time complexity.
- **Calculation of Time complexity:**Time complexity of an algorithm has two components, the compilation time and the execution time. The compilation time is independent of the nature and size of the input. The execution time however, depends on the size and nature of the input data..
- The time taken 'T' by a program 'P'. **T(P) = C + TP(I)**
  Where, 'C' is the compilation time, which is independent of the instance characteristics (e.g. input size)
  TP(I) is execution time i.e. the sum of time required to perform various operation like addition, subtractions and multiplications present in the program. It is typically expressed as a function of 'I'
- Step count:
  For algorithm heading----------1
  For braces----------1
  For expression----------1
  For any loop statement----------no.of time of loop is repeating.
- **Example**: Time Complexity

Statement	**Steps per execution**	**Frequency**	**Total Steps**
Algorithm Sum(number,size)	0	-	0
{	0	-	0
result=0.0;	1	1	1
for count = 1 to size do	1	n+1	n + 1
result= result + number[count];	1	N	n
return result;	1	1	1
}	0	-	0
Total			2n + 3

```
Algorithm sum(A,n)
{
    s=0;                          1
    for(i=0;i<n;i++)              n+1
    {
     s=s+A[i];                    n
    }
    return s;                     1
}
```

Time Complexity: 2n+3

Space Complexity:
A —— n
n —— 1
s —— 1
i —— 1
n+3

Multiplication of two matrices

```
Algorithm multiply(A, B, n)
{
    for(i=0;i<n;i++)                          n+1
    {
      for(j=0;j<n;j++)                        n (n+1)
      {
        c[i][j]=0;                            n*n
        for(k=0;k<n;k++)                      n*n*(n+1)
        {
            c[i][j]=c[i][j]+a[i][j]*b[i][j];  n*n*n
        }
      }
    }
}
```

Time Complexity: $2n^3 + 3n^2 + 2n + 1$

$O(n^3)$

Space Complexity:
A ---------- n^2
B ---------- n^2
C ---------- n^2
i ---------- 1
j ---------- 1
k ---------- 1
n ---------- 1

$3n^2 + 4$

$O(n^2)$

```
for(i=0;i<n;i++)          n+1
    {
      for(j=0;j<n;j++)    n (n+1)
      {
          stmt;           n
      }
    }
```

Time Complexity: $n^2 + 3n + 1$

$Tc = O(n^2)$

i	j	no.of items
0	0	0
1	0	1
	1	-
2	0	1
	1	2
	2	-
n	0	1
	1	2
	.	.
	.	.
	n	n

```
P = 0;
For (i=1;p<=n;i++)
    {
      p=p+i;
     }
```

i	p
1	0+1
2	0+1+2
3	0+1+2+3
.	
.	
.	
K	0+1+2+3+.....+k

$$1+2+3+....+n=\frac{n(n+1)}{2}$$

Assume $p > n$

then $p = \frac{k(k+1)}{2}$

$\frac{k(k+1)}{2} > n$

$k^2 > n$

$k > \sqrt{n} = O(\sqrt{n})$

```
for(i=1;i<n;i=i*2)
    {
      stmt;
     }
```

i	
1	1
1*2	2
2*2	4
4*2	8
.	
.	
.	
K*2	2^k

Assume i>=n

Then $i=2^k$

$2^k >= n$

$2^k = n$

$k = \log_2 n$

$O(\log_2 n)$

```
P=0;
for(i=1;i<n;i=i*2)
    {
      p++;
     }
    for(j=1; j<p;j=j*2)
     {
      stmt;
     }
```

Time Complexity

p++; ———— log n

stmt; ———— log p

$O(\log\log n)$

for (i = 0; i < n; i++) ———— O(n)

for (i = 0; i < n; i=i+2) ———— O(n/2)=O(n)

for (i = n; i >1; i--) ———— O(n)

for (i = 1; i < n; i=i*2) ———— $O(\log_2 n)$

for (i = 1; i < n; i=i*3) ———— $O(\log_3 n)$

for (i = 1; i >1; i=i/2) ———— $O(\log_2 n)$

$$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < 2^n < 3^n < \ldots < n^n$$

Time complexity is mainly classified into 5 types based on frequency count

- Constant
- Linear
- Quadratic
- Cubic
- logarithms

Classification of Algorithms

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < 3^n < \ldots < n^n$$

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc. 1- Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

- **Log n** - When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction. When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to n2 .
- **n** - When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.
- **n log n** - This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving then independently, and then combining the solutions. When n doubles, the running time more than doubles.
- **n^2** - When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.
- **n^3** - Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.
- **2^n** - Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute–force" solutions to problems. Whenever n doubles, the running time squares.

Time complexity represent in 3 ways:

1. **Best Case:** If an algorithm takes or require minimum time to complete an algorithm for set of specific input is known as best case.
   E.g.: In linear search we have to find out 1st element in given list
2. **Worst case:** If an algorithm takes or require maximum time to complete an algorithm for set of specific input is known as best case
   E.g: In linear search we have to find out last element in given list
3. **Average case:** If an algorithm takes or require average time to complete an algorithm for set of specific input is known as best case
   E.g: In linear search we have to find out middle element in given list.

Asymptotic notation – big oh (O), omega (Ω), theta (θ) and little oh (o) notation

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size "n" of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size "n" as we discussed above in the examples.

Complexity shall refer to the b running time of the algorithm. The function f(n), gives the running time of an algorithm, depends not only on the size „n" of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. **Best Case** : The minimum possible value of f(n) is called the best case.
2. **Average Case** : The expected value of f(n).
3. **Worst Case** : The maximum value of f(n) for any key possible input.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

Asymptotic notation

- It is a way to describe the characteristics of a function in the limit.
- It describes the rate of growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions:

Rate of Growth

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm

1. Big–Oh (O) O ≈ ≤
2. Big–Omega (Ω), Ω ≈ ≥
3. Big–Theta (Θ) Θ ≈ =
4. Little–Oh (o) o ≈ <

1. **Big–Oh (O) - Upper Bound**

The function **f (n) = O(g(n))** if and only if there exist positive constants c, n0 such that f (n) ≤ cg(n), ∀n ≥ n0. Big-oh can be used to denote all upper bounds on the time complexity of an algorithm. Big-oh also captures the worst case analysis of an algorithm.

$O(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.

$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Examples

1. 3n + 2

3n + 2 ≤ 4n, c = 4 ∀n ≥ 2

3n + 2 = O(n)

2. 100n + 6 = O(n)
100n + 6 ≤ 101.n, where c = 101 ∀n ≥ 6.
We may also write as 100n + 6 = O(n3).
Analysis of simple for loop
Now let's consider a simple for loop:
for (i = 1; i<=n; i++)
v[i] = v[i] + 1;

This loop will run exactly n times, and because the inside of the loop takes constant time, the total running time is proportional to n. We write it as O(n).

Analysis for nested for loop
Now let's look at a nested for loop:
for (i = 1; i<=n; i++)
for (j = 1; j<=n; j++)
a[i,j] = b[i,j] * x;

The outer for loop executes N times, while the inner loop executes n times for every execution of the outer loop. That is, the inner loop executes n x n = n2 times. The assignment statement in the inner loop takes constant time, so the running time of thecode is O(n2) steps. This piece of code is said to have quadratic running time.

2. Big–Omega Ω - Lower Bound

The function **f (n) = Ω(g(n))** if and only if there exist positive constants c, n0 such that f (n) ≥ c.g(n), ∀n ≥ n0. Omega can be used to denote all lower bounds of an algorithm. Omega notation also denotes the best case analysis of an algorithm.

$\Omega(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Examples
1. 3n + 2
3n + 2 ≥ n, ∀n ≥ 1
3n + 2 = Ω(n)
2. 10n^2+ 4n + 2 = Ω(n^2)
10n2+ 4n + 2 ≥ n2, c = 1 ∀n ≥ 1
3. n^3+ n + 5 = Ω(n^3)
n^3+ n + 5 ≥ n3, c = 1, ∀n ≥ 0

3.Big–Theta Θ Average bound

The function **f (n) = Θ(g(n))** if and only if there exist positive constants c1, c2, n0 such that c1g(n) ≤ f (n) ≤ c2g(n), ∀n ≥ n0. Theta can be used to denote tight bounds of an algorithm. i.e., g(n) is a lower bound as well as an upper bound for f (n).

Note that f (n) = Θ(g(n)) if and only if f (n) =Ω(g(n)) and f (n) = O(g(n)).

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Examples

1. 3n + 10^10

3n ≤ 3n + 10^10 ≤ 4n, ∀n ≥ 1010

3n + 10 = Θ(n)

Note that the first inequality captures 3n + 1010 = Ω(n) and the later one captures 3n + 10^10 = O(n).

2. 10n^2+ 4n + 2 = Θ(n^2)

10n^2≤ 10n^2+ 4n + 2 ≤ 20n^2, ∀n ≥ 1, c1 = 10, c2 = 20

4.Little oh (o) notation :

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound 2n2=O(n2) is asymptotically tight, but the bound 2n=O(n2) is not. We use o-notation (Little oh) to denote an upper bound that is not asymptotically tight**.**

We formally define as f (n) = o(g(n)) if for any positive constant c > 0, there exists a positive constant n0 > 0 such that 0 ≤ f (n) < c.g(n) for all n ≥ n0.

5.Little omega ( ) notation :

We use ω-notation to denote a lower bound that is not asymptotically tight. We define **ω(g(n)) (little-omega) as f (n) = ω(g(n))** if for any positive constant c > 0, there exists a positive constant n0 > 0 such that 0 ≤ c.g(n) < f (n) for all n ≥ n0.

Examples 1. n^2 = ω(n) but n^2 ≠ ω(n^2)

Properties of Asymptotic notation

1. Reflexivity:f (n) = O(f (n)) f (n) = Ω(f (n)) f (n) = θ(f (n))

2. Symmetry:f (n) = θ(g(n)) if and only if g(n) = θ(f (n))

3. Transitivity:f (n) = O(g(n)) and g(n) = O(h(n)) ⇒ f (n) = O(h(n))

4. Transpose Symmetry:f (n) = O(g(n)) if and only if g(n) = Ω(f (n))

Difference Between Big oh, Big Omega and Big Theta :

S. No	Big Oh	Big Omega	Big Theta
1.	It is like <= rate of growth of an algorithm is less than or equal to a specific value	It is like >= rate of growth is greater than or equal to a specified value	It is like == meaning the rate of growth is equal to a specified value
2.	The upper bound of algorithm is represented by Big O notation. Only the above function is bounded by Big O. asymptotic upper bond is it given by Big O notation.	The algorithm's lower bound is represented by Omega notation. The asymptotic lower bond is given by Omega notation	The bounding of function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation.
3.	Big oh (O) – Worst case	Big Omega (Ω) – Best case	Big Theta (Θ) – Average case
4.	Big-O is a measure of the longest amount of time it could possibly take for the algorithm to complete.	Big- Ω is take a small amount of time as compare to Big-O it could possibly take for the algorithm to complete.	Big- Θ is take very short amount of time as compare to Big-O and Big-? it could possibly take for the algorithm to complete.
5.	Mathematically – Big Oh is 0 <=f(n) <= c g(n) for all n>=n0	Mathematically – Big Omega is O<= C g(n) <= f(n) for all n>=n 0	Mathematically – Big Theta is O<=C 2 g(n)<=f(n)<=C 1 g(n) for n>=n 0

Divide and Conquer: Binary search, merge sort, bubble sort, quick sort, Strassen matrix multiplication, maximum and minimum problem.

Binary search:

Searching

- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.

Searching Algorithms

The searching of an element in the given array may be carried out in the following two ways

1. Linear Search
2. Binary Search

Binary Search

Binary Search is one of the fastest searching algorithms.

- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquers technique.

Binary Search Algorithm can be applied only on Sorted arrays. So, the elements must be arranged in

- Either ascending order if the elements are numbers. Or dictionary order if the elements are strings.

To apply binary search on an unsorted array,

- First, sort the array using some sorting technique. Then, use binary search algorithm.

Binary Search working principal

Binary Search Algorithm searches an element by comparing it with the middle most element of the array. Then, following **three** cases are possible

Case-1: If the element being searched is found to be the middle most element, its index is returned.

Case-2: If the element being searched is found to be greater than the middle most element, then its search is further continued in the right sub array of the middle most element.

Case-3: If the element being searched is found to be smaller than the middle most element, then its search is further continued in the left sub array of the middle most element. This iteration keeps on repeating on the sub arrays until the desired element is found or size of the sub array reduces to zero.

Consider, if we have ‘n’ records which have been ordered by keys so that x1 < x2 < …< xn . When we are given a element ‘x’, binary search is used to find the corresponding element from the list. In case ‘x’ is present, we have to determine a value ‘j’ such that a[j] = x (successful search). If ‘x’ is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare ‘x’ with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then ‘x’ must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of ‘x’ with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between ‘x’ and a[mid], and since an array of length ‘n’ can be halved only about log2n times before reaching a trivial length, the worst case complexity of Binary search is about log2n.

Algorithm

```
Algorithm BINSRCH (a, n, x) // Iterative method
// array a(1 : n) of elements in increasing order, n ≥ 0,
// determine whether ‘x’ is present, and if so, set j such that x = a(j)
// else return j
{
low :=1 ; high :=n ;
while (low < high) do
{
mid :=(low + high)/2
if (x < a [mid]) then
 high:=mid – 1;
else if (x > a [mid]) then
low:= mid + 1
else return mid;
}
return 0;
}
```

Example:

Let us illustrate binary search on the following 9 elements:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

Number of comparisons = 4

2. Searching for x = 82

low	high	mid
1	9	5
6	9	7
8	9	8
		found

Number of comparisons = 3

3. Searching for x = 42

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	found

Number of comparisons = 4

4. Searching for x = -14

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	3	2	3	4	1	3	2	3	4

```
Algorithm BINSRCH (low, high, x)    // Recursive method
 {
        if(low = = high)
           if ( low = = a[mid])
            return low;
           else
            return 0;                                   1
       else
         { mid :=(low + high)/2                         1
              if (x < a [mid])                          1
                 return BINSRCH(low, mid, key)          n/2
              else
                 return BINSRCH(mid+1, high, key)       n/2
         }
 }
```

$T(n) = T(n/2) + 1$

Time complexity analysis:

Binary Search time complexity analysis is done below-

- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for n elements in the array, there are log2n iterations or recursive calls.
- Time Complexity of Binary Search Algorithm is O(log2n).
- Here, n is the number of elements in the sorted linear array.
- The time complexity for a successful search is O(log n) and for an unsuccessful search is Θ(logn).

T (n) = $T\left(\frac{n}{2}\right)+1$ ..... (Equation 1)

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + 1$, putting $\frac{n}{2}$ in place of n.

Then we get: $T(n) = \left(T\left(\frac{n}{2^2}\right) + 1\right) + 1$.....By putting $T\frac{n}{2}$ in (1) equation

$T(n) = T\left(\frac{n}{2^2}\right) + 2$..... (Equation 2)

$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1$..... Putting $\frac{n}{2}$ in place of n in eq 1.

$T(n) = T\left(\frac{n}{2^3}\right) + 1 + 2$

$T(n) = T\left(\frac{n}{2^3}\right) + 3$..... (Equation 3)

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$..... Putting $\frac{n}{3}$ in place of n in eq1

Put $T\left(\frac{n}{2^3}\right)$ in eq (3)

$T(n) = T\left(\frac{n}{2^4}\right) + 4$

Repeat the same process ith times

$T(n) = T\left(\frac{n}{2^i}\right) + i$ ... ..

Stopping Condition: T (1) =1

$\frac{n}{2^i} = 1$

$\frac{n}{2^i}$ Is the last term of the equation and it will be equal to 1

$n = 2^i$

Applying log both sides

$\log n = \log_2 i$

Log n = i log 2

$\frac{\log n}{\log 2} = 1$

$\log_2 n = i$

$T(n) = T\left(\frac{n}{2^i}\right) + i$

$\frac{n}{2^i}$ = 1 as in eq 5

$= T(1) + i$

$= 1 + i$..... T (1) =1 by stopping condition

$= 1 + \log_2 n$

$= \log_2 n$ ..... (1 is a constant that's why ignore it)

Therefore, binary search is of order o ($\log_2 n$)

Recursive tree

1
|
n/2
|
$n/2^2$
⋮
$n/2^k$ Complexity = Θ (log n)

Master method:

a = 1, b = 2, f(n) = 1

$n^{\log_2 1} = 1$. Here $f(n) = n^{\log_b a}$ it satisfies case ii. Thus the complexity is Θ (log n)

Best case –Θ(1) In the best case, the key is the middle in the array. A constant number of comparisons (actually just 1) are required.

Worst case - Θ(log2 n) In the worst case, the key does not exist in the array at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving

can be done ceiling(log2n) times. Thus, [log2n] comparisons are required. Sometimes, in case of the successful search, it may take maximum number of comparisons. So worst case complexity of successful binary search is Θ (log2 n).

Average case - Θ (log2n) To find the average case, take the sum of the product of number of comparisons required to find each element and the probability of searching for that element. To simplify the analysis, assume that no item which is not in array will be searched for, and that the probabilities of searching for each element are uniform.

	Best case	**Average case**	**Worst case**
Successful	O(1)	O(log n)	O(log n)
Unsuccessful	O(log n)	O(log n)	O(log n)

Advantages of binary search algorithm

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search

Disadvantages of binary search algorithm

- Programming binary search algorithm is error prone and difficult.
- It employs recursive approach which requires more stack space.
- The interaction of binary search with memory hierarchy i.e. caching is poor.

QUICK SORT

- Quick Sort is one of the fastest sorting algorithm on average.
- It sorts the given data items in ascending order.
- It uses the idea of divide and conquer approach.
- It follows a recursive algorithm.
- It does not need additional memory, so this is called in-place sorting**.**

Quick Sort Algorithm

- Quick Sort follows a recursive algorithm.
- It divides the given array into two sections using a partitioning element called as pivot.

The division performed is such that-

- All the elements to the left side of pivot are smaller than pivot.
- All the elements to the right side of pivot are greater than pivot.

After dividing the array into two sections, the pivot is set at its correct position. Then,sub arrays are sorted separately by applying quick sort algorithm recursively.

Algorithm

```
Algorithm QUICKSORT(a,low, high)
{
        if low < high then
        {
                j := PARTITION(a, low, high+1); // j is the position of the partitioning element
                QUICKSORT(a,low, j – 1);
                QUICKSORT(a,j + 1 , high);
```

```
        }
}
Algorithm PARTITION(a,low ,high)
{
        Pivot=a[low];
        Start=low;
        End=high;
        While(start < end)
        {
                While(a[start]<=pivot)
                        Start++;
                While(a[end]>pivot)
                        End--;
                If(Start<end)
                        Swap(a[start],a[end]);
        }
        Swap(a[end],a[pivot]);
}
```

Example Select last element as the pivot element. Initially ‘i’ pointer is placed in position of ‘p1’ where p is the starting element in an array. Move the ‘i’ pointer from left to right in search of an element smaller than pivot. Move the ‘j’ pointer from left to right in search of an element larger than pivot. If such elements are found, the elements in ‘i’ and ‘j’ position are swapped. This process continues till the ‘i’ pointer crosses the ‘j’ pointer. If ‘j’ pointer reaches the last position and found no other elements to compare, then keep the pivot element in ‘i+1’ position.

Let us consider the following example with 13 elements to analyze quick sort:

1	2	3	4	5	6	7	8	9	10	11	12	13	Remarks
38	08	16	06	79	57	24	56	02	58	04	70	45	
P,j												Pivot	38<45, increment i
38	08	16	06	79	57	24	56	02	58	04	70	45	8<45, increment i
	i ,j – in place sorting												
38	08	16	06	79	57	24	56	02	58	04	70	45	16<45, increment i
		i ,j – in place sorting											
38	08	16	06	79	57	24	56	02	58	04	70	45	6<45, increment i
			i ,j – in place sorting										
38	08	16	06	79	57	24	56	02	58	04	70	45	79>45, increment j
			i	j									
38	08	16	06	79	57	24	56	02	58	04	70	45	57>45, increment j
			i		j								
38	08	16	06	79	57	24	56	02	58	04	70	45	24<45, increment i
			i			j – swap i and j elements							
38	08	16	06	79	57	24	56	02	58	04	70	45	
				i		j							
38	08	16	06	24	57	79	56	02	58	04	70	45	79>45, increment j
				i		j							
38	08	16	06	24	57	79	56	02	58	04	70	45	02<45, increment i
				i				j – swap i and j elements					
38	08	16	06	24	57	79	56	02	58	04	70	45	
					i			j					
38	08	16	06	24	02	79	56	57	58	04	70	45	57, 58 > 45, increment j
					i					j			
38	08	16	06	24	02	79	56	57	58	04	70	45	4<45, increment i
						i				j – swap i and j elements			
38	08	16	06	24	02	04	56	57	58	79	70	45	79, 70>45, increment j
						i					j		
38	08	16	06	24	02	04	56	57	58	79	70	45	J has no moves
							i , pivot - insert pivot in i+1 position						
38	08	16	06	24	02	04	45	56	57	58	79	70	Repeat the same process
						Now array is divided into two sub arrays after placing pivot in position							
(38	08	16	06	24	02	04)	45	(56	57	58	79	70)	Repeat the same procedure

Activate Window

Quick Sort Analysis

To find the location of an element that splits the array into two parts, O(n) operations are required.

- This is because every element in the array is compared to the partitioning element.
- After the division, each section is examined separately.
- If the array is split approximately in half (which is not usually), then there will be log2n splits.
- Therefore, total comparisons required are f(n) = n x log2n = O(nlog2n).

- In the case, if the pivot reaches in the middle then to simply the math, we assume that the two sub-files are each exactly half the size of the original and although this gives a slight over estimate, this is acceptable because we are only interested in a Big – oh answer.
  T (n) = 2 T (n/2) + C n
  (refer substitution / recursive tree method for analysis)
  Applying master's theorem, the complexity is O(n log n)

A=2, b=2, f(n) = n

n^ log2 = n; n = f(n) satisfies case ii. Hence the complexity is O(n log n)

Worst Case

- Quick Sort is sensitive to the order of input data.
- It gives the worst performance when elements are already in the ascending order.
- It then divides the array into sections of 1 and (n-1) elements in each call.
- Then, there are (n-1) divisions in all.
- Therefore, here total comparisons required are f(n) = n x (n-1) = O(n2).

Equation:

1. T (n) =T(1)+T(n-1)+n

T (1) is time taken by pivot element.

T (n-1) is time taken by remaining element except for pivot element.

N: the number of comparisons required to identify the exact position of itself (every element)

T (n) =T (1) + T (n-1) + n…… (1)

T (n-1) =T (1) +T (n-1-1) + (n-1)

By putting (n-1) in place of n in equation1

Put T (n-1) in equation1

T(n)=T(1)+T(1)+(T(n-2)+(n-1)+n……(ii)

T (n) =2T (1) +T (n-2) + (n-1) +n

T (n-2) =T (1) +T (n-3) + (n-2)

By putting (n-2) in place of n in equation1

Put T (n-2) in equation (ii)

T(n)=2T(1)+T(1)+T(n-3)+(n-2)+(n-1)+n

T(n)=3T(1)+T(n-3)+)+(n-2)+(n-1)+n

T (n-3) =T (1) +T (n-4) +n-3

By putting (n-3) in place of n in equation1

T(n)=3T(1)+T(1)+T(n-4)+(n-3)+(n-2)+(n-1)+n

=4T(1)+T(n-4))+(n-3)+(n-2)+(n-1)+n……(iii)

T(n)=(n-1) T(1) + T(n-(n-1))+(n-(n-2))+(n-(n-3))+(n-(n-4))+n
T (n) = (n-1) T (1) + T (1) + 2 + 3 + 4+.....n
T (n) = (n-1) T (1) +T (1) +2+3+4+.....+n+1-1
[Adding 1 and subtracting 1 for making AP series]
T (n) = (n-1) T (1) +T (1) +1+2+3+4+..... + n-1

T (n) = (n-1) T (1) +T (1) + $\frac{n(n+1)}{2}$ -1

Stopping Condition: T (1) =0
Because at last there is only one element left and no comparison is required.

T (n) = (n-1) (0) +0+ $\frac{n(n+1)}{2}$ -1

T (n) =$\frac{n^2+n-2}{2}$

Avoid all the terms expect higher terms n^2

T (n) =O (n^2)

Worst Case Time Complexity : O(n2)
Best Case Time Complexity : O(n*log n)
Average Time Complexity : O(n*log n)
Space Complexity: O(n*log n)

Advantages of Quick Sort

The advantages of quick sort algorithm are-

- Quick Sort is an in-place sort, so it requires no temporary memory.
- Quick Sort is typically faster than other algorithms. (because its inner loop can be efficiently implemented on most architectures)
- Quick Sort tends to make excellent usage of the memory hierarchy like virtual memory or caches.
- Quick Sort can be easily parallelized due to its divide and conquer nature.

Disadvantages of Quick Sort: The disadvantages of quick sort algorithm are-

- The worst case complexity of quick sort is O(n2).
- This complexity is worse than O(nlogn) worst case complexity of algorithms like merge sort, heap sort etc.
- It is not a stable sort i.e. the order of equal elements may not be preserved.

Merge sort:

Merge sort is yet another sorting algorithm that falls under the category of Divide and Conquer technique. It is one of the best sorting techniques that successfully build a recursive algorithm.

Divide and Conquer Strategy

In this technique, we segment a problem into two halves and solve them individually. After finding the solution of each half, we merge them back to represent the solution of the main problem.

Suppose we have an array A, such that our main concern will be to sort the subsection, which starts at index p and ends at index r, represented by A[p..r].

Divide

If assumed q to be the central point somewhere in between p and r, then we will fragment the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

After splitting the arrays into two halves, the next step is to conquer. In this step, we individually sort both of the subarrays A[p..q] and A[q+1, r]. In case if we did not reach the base situation, then we again follow the same procedure, i.e., we further segment these subarrays followed by sorting them separately.

Combine

As when the base step is acquired by the conquer step, we successfully get our sorted subarrays A[p..q] and A[q+1, r], after which we merge them back to form a new sorted array [p..r].

Algorithm

```
Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
 {
        if (low < high)
        {
                mid := (low + high)/2 //finds where to split the set
                MERGESORT(low, mid) //sort one subset
                MERGESORT(mid+1, high) //sort the other subset
```

```
                MERGE(low, mid, high) // combine the results
        }
 }
Algorithm MERGE (low, mid, high)
{
        h :=low;
        i := low;
         j:= mid + 1;
        while ((h < mid) and (J < high)) do
        {
                if (a[h] < a[j]) then
                {
                        b[i] := a[h]; h := h + 1;
                }
                else
                {
                        b[i] :=a[j]; j := j + 1;
                }
                i := i + 1;
        }
         if (h > mid) then
         for k := j to high do
         {
                b[i] :=a[k]; i := i + 1;
         }
         else
         for k := h to mid do
         {
                b[i] := a[k]; i := i + 1;
         }
         for k := low to high do
                a[k] := b[k];
}
```

Eaxmple:

Consider the following example of an unsorted array, which we are going to sort with the help of the Merge Sort algorithm.

A= (36,25,40,2,7,80,15)

Step1: The merge sort algorithm iteratively divides an array into equal halves until we achieve an atomic value. In case if there are an odd number of elements in an array, then one of the halves will have more elements than the other half.

Step2: After dividing an array into two subarrays, we will notice that it did not hamper the order of elements as they were in the original array. After now, we will further divide these two arrays into other halves.

Step3: Again, we will divide these arrays until we achieve an atomic value, i.e., a value that cannot be further divided.

Step4: Next, we will merge them back in the same way as they were broken down.
Step5: For each list, we will first compare the element and then combine them to form a new sorted list.
Step6: In the next iteration, we will compare the lists of two data values and merge them back into a list of found data values, all placed in a sorted manner.

Analysis of Merge Sort:
Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most $2T\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$

But we ignore '-1' because the element will take some time to be copied in merge lists.

So $T(n) = 2T\left(\frac{n}{2}\right) + n$...equation 1

$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$

$= 2^2T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n$

$T(n) = 2^2\,T\left(\frac{n}{2^2}\right) + 2n$.....equation 3

Putting $n = \frac{n}{2^2}$ in equation 1

$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$.....equation4

From Stopping Condition:

$\frac{n}{2^i} = 1$ And $T\left(\frac{n}{2^i}\right) = 0$

$n = 2^i$

apply log both sides:

$\log n = \log_2 i$

$\log n = i \log 2$

$\frac{\log n}{\log 2} = i$

$\log_2 n = i$

From 6 equation

$T(n) = 2^i\,T\left(\frac{n}{2^i}\right) + in$

$= 2^i \times 0 + \log_2 n \,.\, n$

$= T(n) = n.\log n$

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.
Space Complexity: The space complexity of merge sort is **O(n)**

Bubble sort

- Bubble sort is a simple method to sort list, I this sorting techniques we compare adjacent elements and swap if they are in wrong order. This process repeats until no more swaps are needed.

Steps:

- This sorting techniques is also known as exchange sort, which arranges values by iteration over the list several time and in each iteration the larger value gets bubble up to the end of the list.
- This algorithm uses multiple passes and in each pass the 1st and 2nd data items are compared.
- If the 1st element is bigger than the second then the two elements are swapped.
- Next the element in 2nd and 3rd element are compared and if the 1st one is larger than the 2nd then they are swapped, otherwise no change in their order.
- This process continues for each successive pair of elements until all elements are sorted.
- Given data 54,26.93,17,77,31,44,55,20

Pass 1:

54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in its right position

Pass 2:

26	54	17	77	31	44	55	20	93	No Exchange
26	54	17	77	31	44	55	20	93	Exchange
26	17	54	77	31	44	55	20	93	No Exchange
26	17	54	77	31	44	55	20	93	Exchange
26	54	17	31	77	44	55	20	93	Exchange
26	54	17	31	44	77	55	20	93	Exchange
26	54	17	31	44	55	77	20	93	Exchange
26	54	17	31	44	55	20	77	93	No Exchange
26	54	17	31	44	55	20	77	93	77 in its right position

Pass 3:

26	54	17	31	44	55	20	77	93	No Exchange
26	54	17	31	44	55	20	77	93	Exchange
26	17	54	31	44	55	20	77	93	Exchange
26	17	31	54	44	55	20	77	93	Exchange
26	17	31	44	54	55	20	77	93	No Exchange
26	17	31	44	54	55	20	77	93	Exchange
26	17	31	44	54	20	55	77	93	No Exchange
26	17	31	44	54	20	55	77	93	No Exchange
26	17	31	44	54	20	55	77	93	55 in its right position

Pass 4:

26	17	31	44	54	20	55	77	93	Exchange
17	26	31	44	54	20	55	77	93	No Exchange
17	26	31	44	54	20	55	77	93	No Exchange
17	26	31	44	54	20	55	77	93	No Exchange
17	26	31	44	54	20	55	77	93	Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	54 in its right position

Pass 5:

17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	No Exchange
17	26	31	44	20	54	55	77	93	Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	44 in its right position

Pass 6:

17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	No Exchange
17	26	31	20	44	54	55	77	93	Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	31 in its right position

Pass 7:

17	26	20	31	44	54	55	77	93	No Exchange
17	26	20	31	44	54	55	77	93	Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	No Exchange
17	20	26	31	44	54	55	77	93	26 in its right position

Algorithm

```
Algorithm bubblesort(array,size)
{
        int n=arr.length;
        for i=n-1 to 1
                for j=0 to i
                        if arr[j]>arr[j+1]
                                Swap arr[j] and arr[j+1]
}
```

Time Complexities Analysis of Bubble Sort

If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3 and so on. Thus, the total number of comparisons can be found by;

Output;

(n-1) + (n-2) + (n-3) + (n-4) + …… + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

Therefore, the bubble sort algorithm encompasses a time complexity of **$O(n^2)$** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

- **Best Case Complexity**: The bubble sort algorithm has a best-case time complexity of **O(n)** for the already sorted array.
- **Average Case Complexity**: The average-case time complexity for the bubble sort algorithm is **$O(n^2)$**, which happens when 2 or more elements are in jumbled, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity**: The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the descending order of an array into the ascending order.

Strassen's Matrix Multiplication

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969). Suppose we want to multiply two n x n matrices, A and B. Their product, C=AB, will be an n by n matrix and will therefore have n2 elements. The number of multiplications involved in producing the product in this way is Θ(n3)

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j)$$

```
for i := 1 to n do
    for j :=1 to n do
        C[i, j] := 0;
        for K: = 1 to n do
            C[i, j] := C[i, j] + A[i, k] * B[k, j];
```

This algorithm requires n3 scalar multiplication's (i.e. multiplication of single numbers) and n3 scalar additions. So we naturally cannot improve upon. We apply divide and conquer to this problem. For example let us considers three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{21} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{21} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{21} \\ C_{21} & C_{22} \end{pmatrix}$$

Then cij can be found by the usual matrix multiplication algorithm,

C11 = A11 . B11 + A12 . B21
C12 = A11 . B12 + A12 . B22
C21 = A21 . B11 + A22 . B21
C22 = A21 . B12 + A22 . B22

This leads to a divide and conquer algorithm, which performs n x n matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrix additions.

T(1) = 1
T(n) = 8 T(n/2) + O(n2)

Which leads to T (n) = O (n3), where n is the power of 2.

Divide and Conquer through Strassen's Method:

By using divide-and-conquer approach proposed by Strassen in 1969, we can reduce the number of multiplications. Strassen's insight was to find an alternative method for calculating the Cij, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

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

$$= \begin{pmatrix} P+S-T+V & R+T \\ Q+S & P+R-Q+U \end{pmatrix}$$

where,

P = (A11 + A22) (B11 + B22)
Q = (A21 + A22) B11
R = A11 (B12 – B22)
S = A22 (B21 - B11)
T = (A11 + A12) B22
U = (A21 – A11) (B11 + B12)
V = (A12 – A22) (B21 + B22)
C11 = P + S – T + V
C12 = R + T
C21 = Q + S
C22 = P + R - Q + U.

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$$T(1) = 1$$
$$T(n) = 7\,T(n/2) \quad \text{for } n > 1$$

Solving this for the case of $n = 2^k$ is easy:

$$\begin{aligned} T(2^k) &= 7\,T(2^{k-1}) \\ &= 7[\,7\,T(2^{k-2})] \\ &= 7^2\,T(2^{k-2}) \\ &= \text{- - - - - -} \\ &= 7^i\,T(2^{k-i}) \end{aligned}$$

Put i = k

$$\begin{aligned} &= 7^k\,T(1) \\ &= 7^k \\ \text{That is, } T(n) &= 7^{\log_2 n} \\ &= n^{\log_2 7} \\ &= O(n^{\log_2 7}) \\ &= O(n^{2.81}) \end{aligned}$$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

Algorithm

```
Algorithm mm(A,B,n)
{
        if(n<=2)
        {
                C=4 formula's
        }
        else
        {
                Mid…….n/2;
                mm(A11,B11,n/2)+mm(A12+B12,n/2);
                mm(A11,B12,n/2)+mm(A12+B22,n/2);
                mm(A21,B11,n/2)+mm(A22+B21,n/2);
                mm(A21,B12,n/2)+mm(A22+B22,n/2);
        }
}
```

<u>Max - Min Problem</u>

Max-Min problem is to find a maximum and minimum element from the given array. We can effectively solve it using divide and conquer approach.

- Identify low index and high index
- Calculate mid value based on low and high indexes that is mid=l+h/2
- Based on the mid element the given list can be divided into 2 sub lists
  1. Left sub list
  2. Right sub list
- Consider leftsublist ,identify low,high indexes and calculate mid element until the left sub list contain one or two element in the list.
- Consider right sub list ,identify low, high indexes and calculate mid element until the right sub list contain one or two element in the list.
- Compare left and right sub lists identify max and minimum element.

Identify max,min element from following given list: 25,30,50,70,17,23,5,90,16

25 30 50 70 17 23 5 90 16
0 1 2 3 4 5 6 7 8
Low=0,high=8
Mid=0+8/2=4
25 30 50 70 17
1 2 3 4
23 5 90 16
5 6 7 8
Low=0,high=4
Mid=0+4/2=2
Low=5,high=8
Mid=5+8/2=6
25 30 50
70 17
23 5
90 16
1 2 3 4
Low=0,high=2
Mid=0+2/2=1
0 1 2
25 30
50
Max=90
Max=50
Min=25
Max=70
Min=17
Max=90
Min=5

Analyze the algorithm to find the maximum and minimum element from an array.
Algorithm:

```
Algorithm   max&min (i,j, max, min)
{   if (i=j) then
      max=min=a[i]; //list contain only one element
    elseif ( low=high-1)
    {
      if (a[low]<a[high])
      {
        max= a[high]
        min=a[low];
      }
      else
      {
        max =a[low];
        min = a[high]; // list contain 2 elements
      }
    }
    else
    {
        mid = ⌊(low+high)/2⌋;
        max&min ( low, mid, max, min);
        max&min (mid+1, high, max1, min1);
      if (max < max1) //more than one element
          max = max1
      else (min > min1)
          min = min1;
    }
}
```

$$T(n) = \begin{cases} 2T(\frac{n}{2})+1 & n>2 \\ 1 & n\le 2 \end{cases}$$

$O(n)$

* The time complexity for finding max & min number is $O(n)$
The best, avg, worst case of " " $O(n)$

Greedy Methods: General methods, optimal merge patterns, optimal storage on tapes, Knapsack problem, job scheduling problem, single source shortest path problem.

---

1. Greedy method:

An optimization problem is one in which we want to find not just a solution but the best solution. A greedy algorithm sometimes works well for optimization problem a greedy algorithm works in phases:

- We take the best we can get right now, without regard for future consequence
- We hope that by choosing a local optimum at each step, we will end up at a global optimum.

The result is a good solution but not necessarily the best one. The greedy algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal.

Examples: The specific computer algorithms are based on the Greedy approach: Kruskal's algorithm and Prim's algorithm for finding minimum spanning tree.

Characteristics of Greedy method

The following are the characteristics of a greedy method:

- construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.
- A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Applications of Greedy Algorithm

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in a job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.

GENERAL METHOD:

```
Algorithm Greedy (a, n)
{
  Solution : = 0;
 for i = 0 to n do
 {
    x: = select(a);
   if feasible(solution, x)
   {
      Solution: = union(solution , x)
   }
     return solution;
 }
}
```

The above is the greedy algorithm. Initially, the solution is assigned with zero value. We pass the array and number of elements in the greedy algorithm. Inside the for loop, we select the element

one by one and checks whether the solution is feasible or not. If the solution is feasible, then we perform the union.

Let's understand through an example.

Suppose there is a problem 'P'. I want to travel from A to B shown as below:

P : A → B

The problem is that we have to travel this journey from A to B. There are various solutions to go from A to B. We can go from A to B by **walk, car, bike, train, aeroplane**, etc. There is a constraint in the journey that we have to travel this journey within 12 hrs. If I go by train or aeroplane then only, I can cover this distance within 12 hrs. There are many solutions to this problem but there are only two solutions that satisfy the constraint.

If we say that we have to cover the journey at the minimum cost. This means that we have to travel this distance as minimum as possible, so this problem is known as a minimization problem. Till now, we have two feasible solutions, i.e., one by train and another one by air. Since travelling by train will lead to the minimum cost so it is an optimal solution. An optimal solution is also the feasible solution, but providing the best result so that solution is the optimal solution with the minimum cost. There would be only one optimal solution.

The problem that requires either minimum or maximum result then that problem is known as an optimization problem. Greedy method is one of the strategies used for solving the optimization problems.

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.

Optimal merge patterns

- **Optimal merge pattern** is a pattern that relates to the merging of two or more sorted files in a single sorted file. This type of merging can be done by the two-way merging method.
- If we have two sorted files containing n and m records respectively then they could be merged together, to obtain one sorted file in time **O (n+m)**.
- There are many ways in which pairwise merge can be done to get a single sorted file. Different pairings require a different amount of computing time.The main thing is to pairwise merge the n sorted files so that the number of comparisons will be less.
- Greedy method we should follows ia always we should merge a pair of small size lists to get the best result.

- Always to select 2 small size lists and combine, then total merging time will be reduced.
- Total merging cost=$\sum_{i=1}^{size} \boldsymbol{di} * \boldsymbol{xi}$ where di is distance of each node and xi is size of each node

Example: Find out the optimal merge pattern in the following

File : A B C D

Size: 6 5 2 3

Solution:

Total merging cost=11+13+16=40

Other way

Total merging cost=5+10+16=31

To selecting to smaller file for merging first so we get minimum merging cost= 31 no.of comparison

Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.

If merge operations are performed according to the provided sequence, then

M_1 = merge f_1 and f_2 => 20 + 30 = 50

M_2 = merge M_1 and f_3 => 50 + 10 = 60

M_3 = merge M_2 and f_4 => 60 + 5 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

Hence, the total number of operations is 50 + 60 + 65 + 95 = 270

Now, the question arises is there any better solution?

Sorting the numbers according to their size in an ascending order, we get the following sequence –

f_4, f_3, f_1, f_2, f_5

Hence, merge operations can be performed on this sequence

M_1 = merge f_4 and f_3 => 5 + 10 = 15

M_2 = merge M_1 and f_1 => 15 + 20 = 35

M_3 = merge M_2 and f_2 => 35 + 30 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

Therefore, the total number of operations is

15 + 35 + 65 + 95 = 210

Obviously, this is better than the previous one.

In this context, we are now going to solve the problem using this algorithm.

Initial Set

Step-1

Step-2

Step-3

Step-4

Hence, the solution takes 15 + 35 + 60 + 95 = 205 number of comparisons.

Example: find out thr optimal merge pattern the given data files : f1,f2,f3,f4,f5,f6,f7,f8,f9,f10 with Size: 28,32,12,5,84,53,91,35,3,11 (ans:1004)

Example: find out the optimal merge pattern the given data files : f1……f13 with Size:12,34,56,73,24,11,34,56,78,91,34,91,45 (ans:2237)

Algorithm Of Optimal Merge Pattern

Algorithm Greedy procedure for constructing optimal merge Tree

```
struct node
{
 node *lchild;
  int weight;
 node *rchild;
} pt;
Algorithm optimal merge tree(n)
{
  for i=1 to n-1 do        O(n-1)
  {
    pt=new node;
    pt->lchild= least(list);
    pt->rchild= least(list);                O(1)
    pt->weight=((pt->lchild)->weight)+
                ((pt->rchild)->weight)
    insert(list,pt);  (n)
  }
} return least(list);
```

Time complexity is $O(n^2)$

30 →

$n(n-1)$

n^2

Optimal storage on tapes

Optimal Storage on Tapes Problem: Given n programs P_1, P_2, …, P_n of length L_1, L_2, …, L_n respectively, store them on a tap of length L such that Mean Retrieval Time (MRT) is a minimum. The retrieval time of the jth program is a summation of the length of first j programs on tap. Let Tj be the time to retrieve program Pj. The retrieval time of P_j is computed as,

$$T_j = \sum_{k=1}^{j} L_k$$

Length of kth program

Minimum retrieval Time of n programs is the average time required to retrieve any program. It is required to store programs in an order such that their Mean Retrieval Time is minimum. MRT is computed as,

Average retrieval time over n programs

$$MRT = \frac{1}{n}\sum_{i=1}^{N} T_j = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{i} L_k$$

Time to retrieve j^{th} program P_j

Length of kth program

Optimal storage on tape is minimization problem which,

$$\text{Minimize} \sum_{i=1}^{n}\sum_{k=1}^{i} L_k$$

$$\text{Subjected to} \sum_{i=1}^{n} L_i \le L$$

Length of tape

Length of ith program

Example: let n=3 and (l1, l2, l3) = (5, 10, 3). Find out all feasible solution and optimal retrieval order and time

Order	Total RT	mRT
5 10 3 / 1, 2, 3	5+(5+10)+(5+10+3)	38/3
1, 3, 2	5+(3+3)+(5+3+10)	36/3
2, 1, 3	10+(10+5)+(10+5+3)	43/3
2, 3, 1	10+(10+3)+(10+3+5)	41/3
3, 1, 2	3+(3+5)+(3+5+10)	29/3
3, 2, 1	3+(3+10)+(3+10+5)	34/3

Optimal Order (3, 1, 2) with R.T 29/3

The amount this 6 possible way which order has the minimum retrival time & that order can be treat as optimizal

Eg: Let $n=3$ and $(l_1, l_2, l_3) = (4, 7, 6)$

Order	Total RT	
1, 2, 3	4+(4+7)+(4+7+6)	32/3
1, 3, 2	4+(4+6)+(4+6+7)	31/3
2, 1, 3	7+(7+4)+(7+4+6)	35/3
2, 3, 1	7+(7+6)+(7+6+4)	37/3
3, 1, 2	6+(6+4)+(6+7+4)	36/3
3, 2, 1	6+(6+4)+(6+4+7)	33/3

Optimal Order (1, 3, 2) with R.T 31/3

Eg: Find the optimal retrieval time for the following programs & its corresponding lengths

12 5 8 32 7 5 8 13

1. First arrange the prgm lengths in asc ending order

5 5 7 8 8 12 13 32

Total retrieval time time require for access the 1st prgm

$= 5 + (5+5) + (5+5+7) + (5+5+7+8)$
$(5+5+7+8+8) + (5+5+7+8+8+12)$
$+ (5+5+7+8+8+12+13) +$
$(5+5+7+8+8+12+13+32)$

$= 230$ minimum Retrieval time $= \frac{230}{8} = 28.75$

for 3 tapes

Find the total ~~let~~ retrieval Time for the following programs with time Tapes: T_0, T_1, T_2

12 5 8 32 7 5 18 13

Arrange the program l in ASC order

5 5 7 8 12 13 18 32

T_0: 5 8 18

T_1: 5 12 32

T_2: 7 13

Total R.T for Tape T_3

$= 7+(7+13)$

$= 27$

$\therefore$ Total R.T for Tape T_0

$= 5+(5+8)+(5+8+18)$

$= 49$

Total R.T for Tape T_1

$= 5+(5+12)+(5+12+32)$

$= 71$

$\therefore$ Total R.T for 3 Tapes $= 49+71+27$

$= 147$

$m.R.T = \frac{147}{3}$

→ Algorithm GreedyOptimalStorage(n,k)

// Arranage the program length in ASC order

// n is the number of program

// k is the Tapes

5 2 3 4 7 6

n=6 J=0 i=1 Assign 1 to 0

K=3 j = 1 mod 3 =

{

J=0; // Tape Number

for i:=1 to n do

$O(n \log n)$

{ print ("Assign program", i, "for Tape", J);

J = (J+1) mod k;

} }

Eg:
→ Find an optimal placement for 13 program on three tapes T_0, T_1 & T_2 where the program are of length 12 3 8 32 7 5 18 26 4 3 11 10 & 6
Find out the total retrieval time

of storge tapes. these can be Greedy alg
↳ this method is solve the optimization
max result, min result, maybe require
it depend on problem

$n = 13$ → No. of program

managing the program length in ↑ order of lengthen

3 4 5 5 6 7 8 10 11 12 18 26 32

T_0 : 3 5 8 12 32
T_1 : 4 6 10 18
T_2 : 5 7 11 26

Total retrieval time for T_0 Tape

$= 3 + (3+5) + (3+5+8) + (3+5+8+12) + (3+5+8+12+32) = 115$

Total retrieval time for T_1 Tape

$= 4 + (4+6) + (4+6+10) + (4+6+10+18)$

$= 72$

Total retrieval time for T_2 Tape

$= 5 + (5+7) + (5+7+11) + (5+7+11+26) = 89$

Final total retrieval time for three taps i.e., T_0 T_1 & T_2 $= 115 + 72 + 89$

$= 276$

Total retrieval for 13 program using 3 tapes $= 276$

0/1 knapsack

Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The object 'i' has a weight w_i and the knapsack has a capacity 'm'. If a fraction x_i, $0 < x_i < 1$ of object i is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to fill the knapsack that maximizes the total profit earned.

Since the knapsack capacity is 'm', we require the total weight of all chosen objects to be at most 'm'. The problem is stated as:

$$\text{maximize} \sum_{i=1}^{n} p_i \, x_i$$

$$\text{subject to} \sum_{i=1}^{n} a_i \quad x_i \leq M \qquad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n$$

The profits and weights are positive numbers.

Algorithm

If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so that p[i] / w[i] > p[i + 1] / w[i + 1].
// m is the knapsack size and x[1: n] is the solution vector.
{
        for i := 1 to n do x[i]  := 0.0                  // initialize x
        U := m;
        for i := 1 to n do
        {
                if  (w(i) > U) then break;
                x [i] := 1.0; U := U – w[i];
        }
        if (i ≤ n) then x[i] := U / w[i];
}
```

Running time:

The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only O(n) time.

↳ the knapsack problem can be started as follows.

Suppose there are 'n' objects for i=1,...
Each object 'i' has some +ve weight 'w_i' & profit value 'P_i' consider one knapsack or bag with maximum capacity 'm'. Our main object is to fill the knapsack with the objects that has max profit

when an object 'x_i' $0 \leq x \leq 1$, is placed into the knapsack, then profit of $P_i x_i$ can be earned

(i) $x_i = 1$ means an object is completely placed into the knapsack

(ii) $x_i = 0$ mean an object is not placed into the knapsack

(iii) If a part of a object is placed into the knapsack, then

$$x_i = \frac{\text{Remaining Capacity}}{\text{weight of the object}}$$

object : 1, 2, 3 ... n
profit : $P_1 \ldots P_n$
weight : $w_1, w_2 \ldots w_n$
$x_1, x_2 \ldots x_n$ are element of the solution vector
our main objective is maximum $\sum_{i=1}^{n} P_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \le m$ & $0 \le x_i \le 1$, $1 \le i \le n$

[i.e., sum of the profit of object in knapsack should be maximum with respect to the sum of weight of object is $\le$ size of object here x_i is solu better it contains value b/w 0 to 1 where i b/w 1 to n.]

This knapsack & condition

1. $w_1 + w_2 + \ldots + w_n \ge m$
2. $w_1 + w_2 + \ldots + w_n \le m$

Sum of wt of object alway less than size of knapsack

Sam ~~All~~ object placed in knapsack not some object

Algorithm GreedyKnapsack (P, w, m, x, n)
{ // P[1..n], w[1..n] are profit & weight of 'n' objects
// M is the capacity of the knapsack
// x[1..n] is the sol ~~capacity~~ vector
// find P/w ratio for n objects, ordered such that

$$\frac{P[i]}{w[i]} \ge \frac{P[i+1]}{w[i+1]}$$

{
for i = 1 to n do
{ if (w[i] > M) then

break; // ~~pb~~ not taken

else

{ $x[i] := 1$;

$m = m - w[i]$; // full

}

}

if $(i <= n)$ then $O(n)$

$x[i] = \frac{M}{w[i]}$; // fraction

}

$n = 3$ $m = 20$

$(P_1, P_2, P_3) = (25, 24, 15)$

$(w_1, w_2, w_3) = (18, 15, 10)$

$(x_1, x_2, x_3) = (1, \quad)$

Eg: consider the following knapsack Instance of k.p

$n = 3$, $M = 20$ $(P_1, P_2, P_3) = (25, 24, 15)$

$(w_1, w_2, w_3) = (18, 15, 10)$

3 strategy: P/wt

max profit / min weight

1. strategy: select object with max profit maximum profit

Consider $(P1, P2, P3) = (25, 24, 15)$

Among 3 object profit, object 'i' has max profit, 1st we have the place the object 'i' into the knapsack, It's weight is 18

$x_1 = 1$

Remaining capacity $= M - w1 = 20 - 18 = 2$

Next, highest weight profit object is '2', its Profit is 24, weight is 15

$= \frac{\text{Re. cap}}{\text{weight of object}} = \frac{2}{15}$ $x_2 = 2/15$

Further, we are not placing any object into the knapsack because it reaches the max capacity

$x_3 = 0$

max profit $= \sum_{i=1}^{3} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$

$= 25*1 + 24*2/15 + 15 \times 0 = 28.2$

max weight $= \sum_{i=1}^{3} w_i x_i = w_1 x_1 + w_2 x_2 + w_3 x_3 = 18*1 + 15*\frac{2}{15} + 10*0 = 20$

Strategy 2: minimum weight

Among $(w_1, w_2, w_3) = (18, 15, 10)$ object 3 has known as '10', It is completely placed into knapsack $x_3 = 1$

Remaining capacity $= m - w_3 = 20 - 10 = 10$

we have to place object '2' into the knapsack, but its weight is 15, Rem capacity of knapsack is 10 only. so we are placing a part object into knapsack 'x_2' can be calculated by

$$x_2 = \frac{\text{Rem. capacity}}{\text{weight of object 2}} = \frac{10}{15}$$

$$x_2 = 10/15$$

∴ Further, we are not placing any object into the knapsack because it reaches the max capacity

$$x_1 = 0$$

$$\max P = \sum_{i=1}^{3} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$$
$$= 25 * 0 + 24 * \frac{10}{15} + 15 * 1 = 31$$

$$\max \text{weight} = \sum_{i=1}^{3} w_i x_i = w_1 x_1 + w_2 x_2 + w_3 x_3$$
$$= 18 * 0 + 15 * \frac{10}{15} + 10 * 1 = 20$$

Strategy 3: $\frac{P}{w}$ ratio

1st calculate the P/w ratio of 3 object

Object	1	2	3
Profit	25	24	15
weight	18	15	10
P/w	$\frac{25}{18}$	24/15	15/10
	1.3	1.6	1.5
x_i			

Arranging them in Asc Order

$\left(\frac{P_2}{w_2}, \frac{P_3}{w_3}, \frac{P_1}{w_1}\right) = (1.6, 1.5, 1.3)$

1st, we have to place object '2' into the knapsack, it's weight is '15'

$x_2 = 1$

Remaining capacity $= M - w_2 = 20 - 15 = 5$

Next, object '3' is placed into the knapsack, its weight is '10'

$$x_3 = \frac{\text{Remaining capacity of knapsack}}{\text{weight of the object}} = \frac{5}{10} = \frac{1}{2}$$

Next, we can not placing any object into the knapsack because it reaches the maximum capacity

$x_1 = 0$

$$\text{maximum } P = \sum_{i=1}^{n} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$$
$$= 25*0 + 24*1 + 15*1/2$$
$$= 31.5$$

$$\text{maximum } wt = \sum_{i=1}^{3} w_i x_i = w_1 x_1 + w_2 x_2 + w_3 x_3$$
$$= 18*0 + 15*1 + 10*1/2$$
$$= 15 + 5 = 20$$

1: $m.p = 28.2$

$m.p = 31$

$m.p = 31.5$ ✓

Optimal solution $= 31.5$

Solution vector

$(x_1, x_2, x_3) = (0, 1, 1/2$

Eg2: Find the optimal solution for the following knapsack $n = 7, m = 15$

$(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (10, 5, 15, 7, 6, 18, 3)$

$(w_1, w_2, \ldots) = (2, 3, 5, 7, 1, 4, 1)$

Sol: Our objectiile max profit $\sum_{i=1}^{n} P_i x_i$

Subject to $\sum_{i=1}^{n} w_i x_i \le M$

x is solution vector $x = \{x_1, x_2, x_3 \ldots x_n$

Conside P/w ratio strategy: P_i/w_i

$P_i/w_i = 1 \quad 7$

$\frac{P_1}{w_1} = \frac{10}{2} = 5 \quad \frac{P_2}{w_2} = \frac{5}{3} = 1.6 \quad \frac{P_3}{w_3} = \frac{15}{5} = 3 \quad \frac{P_4}{w_4} = \frac{7}{7} = 1 \quad \frac{P_5}{w_5} = \frac{6}{1} = 6$

$\frac{P_6}{w_6} = \frac{18}{4} = 4.5 \quad \frac{P_7}{w_7} = \frac{3}{1} = 3$

Object O_i :	1	2	3	4	5	6	7
Profit P_i :	10	5	15	7	6	18	3
weight w_i :	2	3	5	7	1	4	1
$\frac{\text{Profit}}{\text{weight}}$ $\frac{P_i}{w_i}$:	5	1.6	3	1	6	4.5	3

$w_2 = 2/3$	$P_2 = 3.34$
$w_7 = 1$	$P_7 = 3$
$w_3 = 5$	$P_3 = 15$
$w_5 = 4$	$P_6 = 18$
$w_2 = 2$	$P_1 = 10$
$w_5 = 1$	$P_5 = 6$

Now solution are $x_1, x_2, x_3, x_4, x_5, x_6, x_7$

x_i is in b/w 0 to 1 i.e., $0 \le x_i \le 1$

The knapsack capacity $m = 15$

→ According to P/wt ratio method we consider max.

Arrange the P/w ratio value in dec order

6 5 4.5 3 3 1.6 1

1st, we have

Add object 5 is placed into the knapsack its weight is 1

$x_5 = 1$

Remaining capacity $= M - w5 = 15 - 1 = 14$

Add object 1 is placed into the knapsack its weight is 2 $x_1 = 1$

Remaining capacity $= m - w1 = 14 - 2 = 12$

Add object 6 is placed into the knapsack its weight is 4 $x_6 = 1$

Remaining capacity $= m - w6 = 12 - 4 = 8$

Add object 3 is placed into the knapsack its wt is 3

Remaining capacity $= m - w3 = 8 - 5 = 3$ $x_3 = 1$

Add object 7 is placed into the knapsack its wt is 1

Remaining capacity $= m - w7 = 3 - 1 = 2$ $x_7 = 1$

Add object 2 is placed into the knapsack it wt is

$x_2 = 2/3$

i.e., $\frac{R.C}{Wt\ of\ O} = \frac{2}{3}$

optimal sol
$(1, 2/3, 1, 0, 1, 1, 1)$

$$\text{maximum profit} = \sum_{i=1}^{n} P_i x_i$$

$$= P_1x_1 + P_2x_2 + \cdots + P_7x_7$$

$$= 10*1 + \frac{2}{3}*5 + 1*15 + 0*7 + 1*6 + 1*18 + 1*3 = 55.34$$

$$\text{max wt} = \sum_{i=1}^{n} w_i x_i$$

$$= 2*1 + 3*\frac{2}{3} + 1*5 + 0*7 + 1*1 + 4*1 + 1*1 = 15$$

for knapsack prob

<u>Job Scheduling With Deadlines</u>

When we are given a set of 'n' jobs. Associated with each Job i, deadline di > 0 and profit Pi > 0. For any job 'i' the profit pi is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in 'j' ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], $1 \le r \le k$ are the jobs in 'j' and d (j [1]) ≤ d (j[2]) ≤ . . . ≤ d (j[k]). To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that d [J[r]] ≤ r, $1 \le r \le k+1$.

A feasible sol for this problem is a subset of j & job such that each job in this subset can be completed by this deadline.

Problem stmt:

In JS problem, the objective is to find a sequence of Jobs, which is completed with their deadlines & given max profit.

Sol: Let us consider a set of 'n' given jobs which associated with deadlines & profit is earned, if a jobs is completed by its deadline. These Job need to be ordered in such a way that there is max profit

It may happen a that all of given jobs may not be completed within their deadlines.

Assume, the deadlines of ith job j_i & d_i & the profit received from the job is P_i. Hence the optimal solution of this alg is a feasible sol with max profit. Thus $D(i) > 0$ for $1 \le i \le n$

Intially, these job are ordered according to profit i.e., $P_1 \ge P_2 \ge P_3 \ge \cdots \ge P_n$

Eg:

Deadline x, z, y → max profit → optimization

Our task execution with in deadline that cable to gain profit

Eg: Find out optimal solution of following Job scheduling $n=4$, $(P_1 \cdots P_4) = (25, 45, 16, 60)$
$(D_1, D_2, D_3, D_4) = (3, 2, 3, 2)$

Arranage the profit in Dec order

$(P_4, P_2, P_1, P_3) = (60, 45, 25, 16)$

$(D_4, D_2, D_1, D_3) = (2, 2, 3, 3)$

maximum Deadline

J_2	J_4	J_1
0	1	2	3

→ Each & Every slot contain 1 unit

Job	Assigned slot	Selected Job	Action	Profit
{ }	None	J_4	Assigned to [1,2] slot	60
$\{J_4\}$	[1,2]	J_2	Assigned to [0,1] slot	60+45= 105
$\{J_4, J_2\}$	[0,1] [1,2]	J_1	Assigned to [2,3] slot	105+25 = 130
$\{J_4, J_2, J_1\}$	[0,1] [1,2] [2,3]	J_3	Rejected the J_3 because slot is not empty	130

optimal solution = $\{J_4, J_2, J_1\}$ with processing sequence $J_2 \rightarrow J_4 \rightarrow J_1$ with maximum = 130

Eg 2:

Algorithm Job scheduling(d, J, n)

// d is the set of deadline of n $d_i > 0$

// J is the solution vector

// n is the no. of Jobs

{

J = {1}; // Job sol vector is empty i.e, use max profit 1

for i := 2 to n do

{ if (all jobs in J∪{i} completed within their deadline) then

J = J∪{i};

}

}

Time complexity: O(n^2)

Eg 2: $n=5$ $(P_1, P_2, \ldots P_5) = \{20, 15, 10, 1, 6\}$

$(D_1 \ldots D_5) = \{2, 2, 1, 3, 3\}$

Arranage the profit in ↓ order

$(P_1, P_2, P_3, P_5, P_4) = \{20, 15, 10, 6, 1\}$

$(D_1, D_2, D_3, D_5, D_4) = \{2, 2, 1, 3, 3\}$

maximum Deadline $= 3$

0	1	2	3
J2	J1	J5	

Job	Assigned Slot	Selected Job	Action	Profit
{ }	None	J1	Assigned to [1,2] slot	20
{J1}	[1,2]	J2	Assigned to [0,1] slot	20+15
{J1, J2}	[0,1] [1,2]	J3	Rejected J3	35
{J1, J2}	[0,1] [1,2]	J5	Assigned to [2,3] slot	35+6 = 41
{J1, J2, J5}	[0,1] [1,2] [2,3]	J4	Rejected J4	41

optimal solution = {J1, J2, J5} with maximum profit is 41 with processing scheduling or sequence J2 → J1 → J5

3: $n=6$

$(P_1 \ldots P_6) = \{200, 180, 190, 300, 120, 100\}$

$(D_1 \ldots D_6) = \{5, 3, 3, 2, 4, 2\}$

0	1	2	3	4	5
J4	J4	J3	J5	J1	X

J_6 → rejected discarded

~~300~~ 180 300 190 120 200 = 990

Single Source Shortest Path Problem: Dijkstra's Algorithms

In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between then (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.

Dijkstra's algorithm does not work for negative edges at all.

The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.

Graph

Shortest Paths

Algorithm:

```
Algorithm Shortest-Paths (v, cost, dist, n)
// dist [j], 1 ≤ j ≤ n, is set to the length of the shortest path
// from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{
        for  i :=1 to n do
        {
                S  [i] :=  false;                       // Initialize S.
                dist [i] :=cost [v, i];
        }
        S[v] := true; dist[v]  := 0.0;                  // Put v in S.
        for num := 2 to n – 1 do
        {
                Determine n - 1 paths from v.
                Choose u from among those vertices not in S such that dist[u] is minimum;
                S[u]  :=  true;                                  // Put u is S.
                for (each w adjacent to u with S [w] = false) do
                        if (dist [w] > (dist [u] + cost [u,  w]) then      // Update distances
                                dist [w] := dist [u] + cost [u, w];
        }
}
```

n(n-1)=O(n^2)

Eg:

1: we are starting at source vertex i.e., vertex 1

$S = \{1\}$

From source vertex we can find 2 possible distance

2: $dist[2] = C[1,2] = 1 \to 2 = 2$

$dist[3] = \infty$

$dist[4] = C[1,4] = 1 \to 4 = 6$

$dist[5] = C[1,5] = 1 \to 5 = 20$

$dist[6] = C[1,6] = 1 \to 6 = 10$

2 Among dist[2] . . . dist[6]

dist[2] is min so we have to select vertex '2' as the next current vertex so vertex 2 is added to S

$\therefore S = \{1, 2\}$

2: $S = \{1, 2\}$ Next current vertex 2

$dist[3] = 1 \xrightarrow{2} 3$ (u v)

u = intermediate vertex = 2

v = Destination vertex = 3

$= \min\{dist[v], dist[u] + cost[u,v]\}$

$= \min\{\infty, 2+10\} = 12$

$dist[4] = 6$

$dist[5] = 20$

$dist[6] = 10$

Among d[3], dist[4], dist[3], dist[6];

dist[4] is minimum so we have to select vertex 4 as the next current vertex so vertex 4 is added to S

$S = \{1, 2, 4\}$

Step 3: $S = \{1, 2, 4\}$

$dist[3] = 1 \xrightarrow{4} 3$

$= \min\{dist[v], dist[u] + cost[u,v]\}$

$= \{12, 6+4\} = 10$

$dist[5] = 1 \xrightarrow{4} 5 = min\{dist[v], dist[4] + cost[4,v]\}$

$= min\{22, 6+12\} = 18$

$dist[6] = 10$

Among $dist[3]$, $dist[5]$, $dist[6]$;

$dist[3]$ is minimum $dist[6]$ is also min.

So we have to select vertex 3 as the current vertex. So vertex 3 is added to S

$S = \{1, 2, 4, 3\}$.

4: $S = \{1, 2, 4, 3\}$

$dist[5] = 1 \xrightarrow{3} 5$

$= min\{dist[v], dist[u] + cost[u,v]\}$

$\{18, 10+2\} = 12$

$dist[6] = 10$

$dist[6]$

5: $S = \{1, 2, 4, 3, 6\}$ $1 \xrightarrow{6} 5$

$dist[5] = min\{dist[5], dist[6] + cost[u,v]\}$

$= min\{12, 10+9\} = 12$

$S = \{1, 2, 4, 3, 6, 5\}$

Hence shortest path is $\{1 \xrightarrow{2} 2 \rightarrow 4 \rightarrow 3 \rightarrow 6 \rightarrow 5\}$ & distance is

Hence single source path is & distance is

Single source path from each vertex is summarized below

$1 \rightarrow 2 = 2$

$1 \rightarrow 4 = 6$

$1 \rightarrow 5 = 20$

$1 \rightarrow 6 = 10$

$1 \rightarrow 2 \rightarrow 3 = 12$ $2+10$

$1 \rightarrow 4 \rightarrow 3 = 10$ $6+4$

$1 \rightarrow 4 \rightarrow 5 = 18$ $6+12$

$1 \rightarrow 2 \rightarrow 3$

Example 2:

$A \rightarrow 0$
$B \rightarrow 2$
$C \rightarrow 9$
$D \rightarrow 3$
$E \rightarrow 6$
$F \rightarrow 11$

Step 1: Select source vertex 'A'

$S = \{A\}$

we can find distance

dist[B] = 2 dist[E] = ∞
dist[C] = ∞ dist[F] = ∞
dist[D] = 4

select vertex which optimal path is $\{A \to B, 1 \to 2\}$
& ~~minimum distance is 2~~

2. $S = \{A, B, 1, 2\}$ $A \xrightarrow{B} C$

$dist[C] = min\{dist[C], cost[u,v] + dist[u]\}$
$= min\{\infty, 7 + 2\} = 9$

$dist[D] = min\{dist[D], dist[u] + cost[u,v]\}$ B B,D
$A \xrightarrow{B} D$
$= min\{4, 2+1\} = 3$

dist[E] = ∞
dist[F] = ∞

• dist[D] is minimum we have to select vertex 'D' to next current vertex so vertex 'D' is added to S

3. $S = \{A, B, D\}$ $A \xrightarrow{D} E$

$dist[E] = min\{dist[E], cost[u,v] + dist[D]\}$
$= min\{\infty, 3 + 3\} = 6$

dist[F] = ∞

• dist[E] is select

4. $S = \{A, B, D, E\}$ $A \xrightarrow{E} F$

$dist[F] = min\{dist[F], cost[E,F] + dist[E]\}$
$= min\{\infty, 5 + 6\} = 11$

Hence shortest path is $\{A \xrightarrow{2} B \xrightarrow{3} D \xrightarrow{3} E \xrightarrow{6} F\}$

~~Dist~~ { dist[2] = 10

1. dist[3] = ∞

4 = ∞

5 = ∞

6 = 30

7 = ∞

$1 \xrightarrow{2} 3$

2. 2 dist[3] = min{ dist[v3], cost[u,v3] + dist[u3] } = min{ ∞, 20 + 10 } = 30

dist[4] = ~~10~~ ∞

[5] = ∞

[6] = ∞

[7] = ∞

3. S = { $1 \to 2 \to 3$ } $1 \xrightarrow{3} 4$

d[4] = min{ dist[v4], cost[u,v] + dist[u] }

= min{ ∞, 15 + 30 } = 45

d[5] = min{ ∞, 5 + 30 } = 35

d[6] = ∞

d[7] = ∞

4. S = { $1 \to 2 \to 3 \to 5$ }

$1 \xrightarrow{5} 7$

d[7] = min{ ∞, 7 + 35 } = 42

d[6] = ∞

S = { $1 \xrightarrow{10} 2 \xrightarrow{20} 3 \xrightarrow{5} 5 \xrightarrow{7} 7$ } = 42

Hence shortest path is ↓

Single source path from each vertex

$1 \to 2 = 10$ $\quad$ $1 \to 2 \to 3 \to 4 = 45$

$1 \to 6 = 30$ $\quad$ $1 \to 2 \to 3 \to 5 = 35$

$1 \to 2 \to 3 = 30$ $\quad$ $1 \to 2 \to 3 \to 5 \to 7 = 57$

$4 \to 7$ 65

42

Running time:

Depends on implementation of data structures for dist.

- Build a structure with n elements A
- at most m = |E| times decrease the value of an item mB
- 'n' times select the smallest value nC
- For <u>array</u> A = O (n); B = O (1); C = O (n) which gives O (n^2) total.
- For <u>heap</u> A = O (n); B = O (log n); C = O (log n) which gives O (n + m log n) total.

Examples:

What is a Spanning Tree?

Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

What is a Minimum Spanning Tree?

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation

Prim's and Kruskal's Algorithm for MST

Both Prim's and Kruskal's algorithms are developed for discovering the minimum spanning tree of a graph. Both the algorithms are popular and follow different steps to solve the same kind of problem.

The prim's algorithm selects the root vertex in the beginning and then traverses from vertex to vertex adjacently. On the other hand, Krushal's algorithm helps in generating the minimum spanning tree, initiating from the smallest weighted edge.

What is Prim's Algorithm?

Prim's algorithm is popular as a greedy algorithm that helps in discovering the smallest spanning tree for a weighted undirected graph. That means, this algorithm tends to search the subgroup of the edges that can construct a tree, and the complete poundage of all the edges in the tree should be minimal.

What is the Kruskal Algorithm?

In the world of computer science, Kruskal's Algorithm is used to discover the smallest spanning tree of a connected graph and the spanning forest of an undirected edge-weighted graph. Basically, it accepts a graph as input and discovers the subgroup of the edges of that graph.

Prim's Algorithm-

- Prim's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Prim's algorithm, the given graph must be weighted, connected and undirected.

Prim's algorithm implement:

The implementation of Prim's Algorithm is explained in the following steps-

Step-01:

- Randomly choose any vertex.
- The vertex connecting to the edge having least weight is usually selected.

Step-02:

- Find all the edges that connect the tree to new vertices.
- Find the least weight edge among those edges and include it in the existing tree.
- If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

Step-03:

- Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.

The time complexity of Prim's algorithm is O(V2).

Example

Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-

The above discussed steps are followed to find the minimum cost spanning tree using Prim's Algorithm-

Step1: consider minimum weight of edge 1-6 is 10

Step2: next we can consider minimum weight of edge 6-5 is 25 of all adjacent edges

Step3: next we can consider minimum weight of edge 5-4 is 22 of all adjacent edges

Step4: next we can consider minimum weight of edge 4-3 is 12 of all adjacent edges

Step5: next we can consider minimum weight of edge 3-2 is 16 of all adjacent edges

Step6: next we can consider minimum weight of edge 2-7 is 14 of all adjacent edges

Since all the vertices have been included in the MST, so we stop.

Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

Example 2:

Using Prim's Algorithm, find the cost of minimum spanning tree (MST) of the given graph-

Solution

The minimum spanning tree obtained by the application of Prim's Algorithm on the given graph is as shown below-

Now, Cost of Minimum Spanning Tree

= Sum of all edge weights

= 1 + 4 + 2 + 6 + 3 + 10

= 26 units

<u>Kruskal's Algorithm-</u>

- Kruskal's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Kruskal's algorithm, the given graph must be weighted, connected and undirected.

Kruskal's Algorithm implement:

The implementation of Kruskal's Algorithm is explained in the following steps-

Step-01:

- Sort all the edges from low weight to high weight.

Step-02:

- Take the edge with the lowest weight and use it to connect the vertices of graph.
- If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.

Step-03:

- Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

time complexity of Kruskal's Algorithm = **O(E + V)**

Example:

Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm-

To construct MST using Kruskal's Algorithm,

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

Step 1:

Step2: consider minimum weight of edge 1-6 is 10

Step3: next we can consider minimum weight of edge 3-4 is 12 of all adjacent edges

Step 4: next we can consider minimum weight of edge 2-7 is 14 of all adjacent edges

Step5: next we can consider minimum weight of edge 2-7 is 16 of all adjacent edges

Step6: next we can consider minimum weight of edge 5-4 is 22 of all adjacent edges

Step 7: next we can consider minimum weight of edge 5-6 is 25 of all adjacent edges

Since all the vertices have been connected / included in the MST, so we stop.

Weight of the MST

= Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99 units

Difference between Prims and Kruskal Algorithm

S.No.	Prim's Algorithm	Kruskal's Algorithm
1	This algorithm begins to construct the shortest spanning tree from any vertex in the graph.	This algorithm begins to construct the shortest spanning tree from the vertex having the lowest weight in the graph.
2	To obtain the minimum distance, it traverses one node more than one time.	It crosses one node only one time.
3	The time complexity of Prim's algorithm is O(V2).	The time complexity of Kruskal's algorithm is O(E log V).
4	In Prim's algorithm, all the graph elements must be connected.	Kruskal's algorithm may have disconnected graphs.
5	When it comes to dense graphs, the Prim's algorithm runs faster.	When it comes to sparse graphs, Kruskal's algorithm runs faster.
6	It prefers list data structure.	It prefers the heap data structure.

Algorithm of kruskal's

Algorithm 40 Algorithm Kruskal(G). Input: A weighted, undirected, connected graph $G = (V, E)$, $n = |V|$, $m = |E|$; Output: A MST T of G (T is the set of MST edges).

```
T = ∅.
for Each vertex v ∈ V do
    Make-Set(v) // create a singleton set for each vertex in G
Sort edge set E in increasing order of weight. Let the sorted order be
e_1, e_2, ..., e_m.
i = 1.
while |T| < n do
    Let e_i = (u_i, v_i)
    ℓ_1 = find(u_i)
    ℓ_2 = find(v_i)
    if ℓ_1 ≠ ℓ_2 then
        Since the representatives of u_i and v_i are different, they belong to
        different trees and does not close a cycle with edges already in T
        T = T ∪ e_i
        union(ℓ_1, ℓ_2) // combine the two trees
    i = i + 1.
Output T.
```

Prim's algorithm

```
algorithm prim(E, cost, n, t)
{
let (k,l) be an edge of minimum cost is E;
mincost := cos[k,l];
t[1,1] := k;  t[1,2] := l;
for i=1 to n do // intialize near
if (cost [i,l] < cost [i,k]) then near [i] := l;
else near [i] := k;
near[k] := near[l] := 0;
for i := 2 to n-1 do
{
// find n-2 additional
  edges for t.
let j be an indem such that
near [j] ≠ 0 and
cost [j, near [j]) is minimum
t[i,1] := j;  t[i,2] := near[j];
min cost := mincost + cost [j, near[j]];
near [j] := 0;
for k := 1 to n// update near[ ]
if ((near[k] ≠ 0) and (cost [k, near [k]] > cost
then near[k] := j;                          [k,j]))
}
return mincost;
}
```

Dynamic Programming: General method, multistage graphs, 0/1 Knapsack problem, longest Common subsequence, string editing, matrix chain multiplication, travelling salesmen problem, optimal binary search tree

---

GENERAL METHOD

- The Dynamic Programming (DP) is the most powerful design technique for solving optimization problems.
- Dynamic programming is an algorithm design method that solves a given complex problem by breaking it into sub problem solving each of the subprogram just once and storing their result in an array.
- It follows a principle called "**principle of optimality**"
- Principle of optimality states that an optimal sequence of decision has the property that whatever the initial state and decision are but remaining decision must constitute an optimal decision sequence regard to state resulting from the 1st decision.
- It was invented by mathematician named Richard Bellman inn 1950s.
- It is closely related to divide and conquer techniques, where the problem is divided into smaller sub-problems and each sub-problem is solved recursively. But the only difference from divide and conquer is that, instead of solving sub-problems recursively, it solves each of the sub-problems only once and stores the solution to the sub-problems in a **table.**
- The solution to the main problem is obtained by the solutions of these sub problems.

The 2 properties of a problem that suggest that given problem can be solved using dynamic programming

Optimal substructure: A given problem has optimal substructure property with optimal solution of the given problem can be obtained by using optimal solution of its subproblem.

Eg: the shortest path problem has the following Optimal substructure problem node 'x' lies in the shortest path from source node 'u' is a destination 'v' the shortest path from u to x and x to v path.

Overlapping Subprogram:

- Dynamic Programming used when the solution of solve subprogram are needed

again and again. The computed solutions of subprograms are stored in a table so that subprograms don't have to be recomputed.

- If we take the example of following recursive program for Fibonacci numbers, there are many subprograms which are solved again and again.
- we can see the fib(3) is being called 2 times. If we would have stored the value of fib(3), then inserted of computed it again, we could have reused the old stored value.

```
int fib(int n)
{
if(n<=1)
        return n;
 else
        return fib(n-1) + fib(n-2);
}
```

-

The steps of Dynamic Programming technique are:

- **Dividing the problem into sub-problems:** The main problem is divided into smaller sub-problems. The solution of the main problem is expressed in terms of the solution for the smaller sub-problems.
- **Storing the sub solutions in a table:** The solution for each sub-problem is stored in a table so that it can be referred many times whenever required.
- **Bottom-up computation:** The Dynamic Programming technique starts with the smallest problem instance and develops the solution to sub instances of longer size and finally obtains the solution of the original problem instance.

The strategy can be used when the process of obtaining a solution of a problem can be viewed as **a sequence of decisions**. The problems of this type can be solved by taking **an optimal sequence of decisions**.

- A fundamental difference between the greedy strategy and dynamic programming is that in the **greedy strategy, only one decision sequence is generated**

- In the **dynamic programming, a number of them may be generated**. Dynamic programming technique guarantees the optimal solution for a problem whereas greedy method never gives such guarantee.

Two difficulties may arise in any application of dynamic programming:

1. It may not always be possible to combine the solutions of smaller problems to form the solution of a larger one.
2. The number of small problems to solve may be un-acceptably large.

There is no characterized precisely which problems can be effectively solved with dynamic programming; there are many hard problems for which it does not seen to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

Basic Features of Dynamic programming :

- Get all the possible solution and pick up best and optimal solution.
- Work on principal of optimality.
- Define sub-parts and solve them using recursively.
- Less space complexity But more Time complexity.
- It saves us from having to recompute previously calculated sub-solutions.
- Difficult to understanding.

All the dynamic programming approaches are solved by using

1. **Tabulation method and,**

For a given problem, it method builds a table in **Bottom Up** fashion and return the last entry from the table

2. Memorization method.

- This method follows recursive nature.
- This method follows top-down approaches
- This method follows ,we initialize a look up table with initial value as -1,whenever we need the solution to a subproblem,we first look with the lookup table, if the precomputed value in their then we return that value otherwise we calculate value and put the result in the look-up table so that it can be reused later.

Memorization method :

Consider an array to store the result

Step 1 Initially all values in an array be −1.

	0	1	2	3	4	5
F	0	1	1	2	3	5
	−1	−1	−1	−1	−1	−1

Step 2 Call fib(5), the value we dont know, so call fib(3) that also dont know so finally call fib(1). value = 1

Step 3 Call fib(2), value we dont know, call fib(0) whose value is 0

Step 4 fib(1) already known so for fib(2) the value is 0+1 = 1

(Right side) → Call fib(4), value dont know so call fib(2) whose value is 1 so dont call fib(0) & (1)

Step 5 Call fib(3) value dont know, so call fib(1). Under fib(3) all values already updated in the array so no need to call any function ie, fib(1), fib(2), fib(0) & fib(1)

Tabulation method :

	0	1	2	3	4	5
F	0	1	1	2	3	5
			i	i	i	i

→ follows bottom up approach.

Iterative Algm

```
int fib (int n)
{
    if (n <= 1)
        return n;
    f[0] = 0; F[1] = 1
    for (int i = 2; i <= n; i++)
    {
        F[i] = F[i-2] + F[i-1];
    }
    return F[n];
}
```

Divide and Conquer Method vs Dynamic Programming

Divide and Conquer Method	Dynamic Programming
1.It deals (involves) three steps at each level	1.It involves the sequence of four steps:

of recursion: **Divide** the problem into a number of sub problems. **Conquer** the sub problems by solving them recursively. **Combine** the solution to the sub problems into the solution for original sub problems.	○ Characterize the structure of optimal solutions. ○ Recursively defines the values of optimal solutions. ○ Compute the value of optimal solutions in a Bottom-up minimum. ○ Construct an Optimal Solution from computed information.
2. It is Recursive.	2. It is non Recursive.
3. It does more work on subproblems and hence has more time consumption.	3. It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	4. It is a Bottom-up approach.
5. In this subproblems are independent of each other.	5. In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	6. For example: Matrix Multiplication.

APPLICATIONS

- ✓ Optimal binary search tree
- ✓ Matrix chain multiplication
- ✓ 0/1 knapsack problem
- ✓ All pair shortest path problem
- ✓ Travelling sales person problem

Multistage Graph

A multistage graph G = (V, E) is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. In addition, if <u, v> is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1 \leq i < k$.

Let the vertex 's' is the source, and 't' the sink. Let c (i, j) be the cost of edge <i, j>. The cost of a path from 's' to 't' is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from 's' to 't'. Each set V_i defines a stage in the graph. Because of the constraints on E, every path from 's' to 't' starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k.

A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k – 2 decisions. The ith decision involves determining which vertex in v_{i+1}, $1 \leq i \leq k - 2$, is to be on the path. Let c (i, j) be the cost of the path from source to destination. Then using the forward approach, we obtain:

$$\text{cost}(i, j) = \min_{\substack{l \in V_{i+1} \\ \langle j, l \rangle \in E}} \{c(j, l) + \text{cost}(i + 1, l)\}$$

Multistage Graph in Dynamic programming:

* Multistage Graph G = (V, E) is a directed wt Graph
* All the vertices in G are partitioned into k- stage where k >= 2
* Edge are connected through vertices from one stage to next stage
* The starting stage contain a single vertex called "Source vertex"
* The Ending stage contain a single vertex called "Sink vertex"
* There are so many path from source(S) to Sink (T)
* The objective of multistage graph is we have findout the shortest path from source(s) to sink(T)
* The cost of path is calculated by using weight given along the Edges. The cost of the path from source 'S' to sink 'T' is the sum of the edge of the path
* There is a set of vertices in each stage Except 1st & last stage

The cost of a path is calculated by using Two approaches

- Forward
- Backward

$$\text{cost}(i, J) = \min_{\substack{l \in V_{i+1} \\ \langle J, l \rangle \in E}} \{c(J, l) + \text{cost}(i+1, l)\}$$

stage

$D(i, J) = l$

UNIT-3

→ Multistage Graph : Forward approaches

stages:

Eg:

Formula

$$cost(i,J) = \min\{c(J,l) + cost(i+1,l)\}$$

(i → stage, l → sink)

$\langle J,l\rangle \in E,\ D(i,J) = l$

Step 1: $cost(5,12) = 0$

Step 2: $cost(4,9) = \min\{c(9,12) + cost(5,12)\}$

$= \min\{4+0\}$

$= 4$

$D(4,9) = 12$

$cost(4,10) = \min\{c(10,12) + cost(5,12)\}$

$= \min\{2+0\}$

$= 2$

$D(4,10) = 12$

$cost(4,11) = \min\{c(11,12) + cost(5,12)\}$

$= \min\{5+0\} = 5$

$D(4,11) = 12$

Step 3:

$cost(3,6) = \min\{c(6,9) + cost(4,9),$

$c(6,10) + cost(4,10)\}$

$= \min\{6+4,\ 5+2\} = 7$

$D(3,6) = 10$

Step 4:

$$cost(2,2) = \min \begin{Bmatrix} c(2,6) + cost(3,6), \\ c(2,7) + cost(3,7), \\ c(2,8) + cost(3,8) \end{Bmatrix}$$

$D(2,2) = 7$ $\quad = \min\{4+7, 2+5, 7+7\}$

$= 7$

$$cost(2,3) = \min \begin{Bmatrix} c(3,6) + cost(3,6), \\ c(3,7) + cost(3,7) \end{Bmatrix}$$

$= \min\{2+7, 7+5\}$

$= 9$

$D(2,3) = 6$

$$cost(2,4) = \min\{c(4,8) + cost(3,8)\}$$

$= \min\{11+7\} = 18$

$D(2,4) = 8$

$$cost(2,5) = \min \begin{Bmatrix} c(5,7) + cost(3,7), \\ c(5,8) + cost(3,8) \end{Bmatrix}$$

$= \min\{11+5, 8+7\} = 15$

$D(2,5) = 8$

Step 5: $$cost(1,1) = \min \begin{Bmatrix} c(1,2) + cost(2,2) \\ c(1,3) + cost(2,3) \\ c(1,4) + cost(2,4) \\ c(1,5) + cost(2,5) \end{Bmatrix}$$

$$= \min \begin{Bmatrix} 9+7, 7+9 \\ 3+18, 2+15 \end{Bmatrix} = 16$$

$D(1,1) = 2$ or 3

$D(1,1)=2$	$D(1,1)=3$
$D(2,2)=7$	$D(2,3)=6$
$D(3,7)=10$	$D(3,6)=10$
$D(4,10)=12$	$D(4,10)=12$

Shortest path $1 \rightarrow 2 \rightarrow 7 \rightarrow 10 \rightarrow 12 = 16$

$1 \rightarrow 3 \rightarrow 6 \rightarrow 10 \rightarrow 12 = 16$

```
Algorithm  ForwardGraph (G,k,n,p)
{
  cost[n]:=0;
  for J=n-1 to 1 do
  {
    compute cost[J];
    Let 'l' be a vertex, such that <J,l> is
    an edge of G & C[J,l]+cost[l+1,l] is
    minimum
    cost[J]:=cost[J,l]+cost[l];
    d[J]=l;
  }
//find a minimum cost shortest path
  P[1]:=1;
  P[k]:=n;
  for j:=2 to k-1 do
     P[J]=d[P[J-1]];
}
```

Backward Approach

$$cost[i, J] = \min\{cost(i-1, \overset{\text{source}}{l}) + c(l, \overset{\text{sink}}{J})\}$$

$$\langle l, J \rangle \in E \quad l \in V_{i-1} \quad D(i, J) = l$$

Step 1: $cost(1,1) = 0$

Step 2: $cost(2,2) = \min\{cost(1,1) + c(1,2)\}$

$= \min\{0+9\}$

$= 9$

$D(2,2) = 1$

$cost(2,3) = \min\{cost(1,1) + c(1,3)\}$

$= \min\{0+7\}$

$= 7$

$D(2,3) = 1$

$cost(2,4) = \min\{cost(1,1) + c(1,4)\}$

$= \min\{0+3\} = 3$

$D(2,4) = 3$

$cost(2,5) = \min\{cost(1,1) + c(1,5)\}$

$= \min\{0+2\} = 2$

$D(2,5) = 2$

Step 3: $cost(3,6) = \min\{cost(2,2) + c(2,6), cost(2,3) + c(3,6)\}$

$= \min\{9+4, 7+2\} = 9$

$D(3,6) = 3$

$$cost(3,7) = \min\left\{\begin{array}{l} cost(2,2)+c(2,7), \\ cost(2,3)+c(3,7), \\ cost(2,5)+c(5,7) \end{array}\right\}$$

$$= \min\{9+2,\ 7+7,\ 2+11\} = 11$$

$$D(3,7) = 2$$

$$cost(3,8) = \min\left\{\begin{array}{l} cost(2,2)+c(2,8), \\ cost(2,4)+c(4,8), \\ cost(2,5)+c(5,8) \end{array}\right\}$$

$$= \min\{9+1,\ 3+11,\ 2+8\} = 10$$

$$D(3,8) = 2 \text{ or } 5$$

Step 4:

$$cost(4,9) = \min\left\{\begin{array}{l} cost(3,6)+c(6,9), \\ cost(3,7)+c(7,9) \end{array}\right\}$$

$$= \min\{9+6,\ 11+4\}$$

$$= 15$$

$$D(4,9) = 6 \text{ or } 7$$

$$cost(4,10) = \min\left\{\begin{array}{l} cost(3,6)+c(6,10), \\ cost(3,7)+c(7,10), \\ cost(3,8)+c(8,10) \end{array}\right\}$$

$$= \min\{14, 14, 15\} = 14$$

$$D(4,10) = 6 \text{ or } 7$$

$$cost(4,11) = \min\{cost(3,8)+c(8,11)\}$$

$$= \min\{7+6\} = 13$$

$$D(4,11) = 8$$

Step 5:

$$cost(5,12) = \min\left\{\begin{array}{l} cost(4,9)+c(9,12), \\ cost(4,10)+c(10,12), \\ cost(4,11)+c(11,12) \end{array}\right\}$$

$$= \min\{15+4,\ 14+2,\ 8+5\} = 16$$

$$D(5,12) = 10$$

Shortest path

D(5,12)=10	D(5,12)=10
D(4,10)=6	D(4,10)=7
D(3,6)=3	D(3,7)=2
D(2,3)=1	D(2,2)=1

$1 \to 3 \to 6 \to 10 \to 12$
$= 16$

$1 \to 2 \to 7 \to 10 \to 12$
$= 16$

```
Algorithm BackwardGraph(G,k,n,P)
{
  cost[i]:=0
  for J=2 to n do
  {
    //Compute cost[J]
    Let 'l' be vertex, such that <l,J> is
an edge of G & bcost[i-1,l]+c[l,J] is
minimum
    cost[J] = bcost[i-1,l]+c[l,J];
    d[J]=l;
  }//find a min cost shortest path
    p[i]=1
    p[k]=n
    for J=k-1 to 2 do
        p[J] = d[p[J+1]];
}
```

EXAMPLE 2:

Find the minimum cost path from s to t in the multistage graph of five stages shown below. Do this first using forward approach and then using backward approach.

SOLUTION:

FORWARD APPROACH:

$$\text{cost}(i, J) = \min_{\substack{l \in V_{i+1} \\ \langle J, l\rangle \in E}} \{c(j, l) + \text{cost}(i + 1, l)\}$$

cost (1, 1) = min {c (1, 2) + cost (2, 2), c (1, 3) + cost (2, 3)}
= min {5 + cost (2, 2), 2 + cost (2, 3)}

cost (2, 2) = min {c (2, 4) + cost (3, 4), c (2, 6) + cost (3, 6)}
= min {3+ cost (3, 4), 3 + cost (3, 6)}

cost (3, 4) = min {c (4, 7) + cost (4, 7), c (4, 8) + cost (4, 8)}
= min {(1 + cost (4, 7), 4 + cost (4, 8)}

cost (4, 7) = min {c (7, 9) + cost (5, 9)} = min {7 + 0) = 7

cost (4, 8) = min {c (8, 9) + cost (5, 9)} = 3

Therefore, cost (3, 4) = min {8, 7} = 7

cost (3, 6) = min {c (6, 7) + cost (4, 7), c (6, 8) + cost (4, 8)}
= min {6 + cost (4, 7), 2 + cost (4, 8)} = min {6 + 7, 2 + 3} = 5

Therefore, cost (2, 2) = min {10, 8} = 8

cost (2, 3) = min {c (3, 4) + cost (3, 4), c (3, 5) + cost (3, 5), c (3, 6) + cost (3,6)}

cost (3, 5) = min {c (5, 7) + cost (4, 7), c (5, 8) + cost (4, 8)}= min {6 + 7, 2 +3}= 5

Therefore, cost (2, 3) = min {13, 10, 13} = 10

cost (1, 1) = min {5 + 8, 2 + 10} = min {13, 12} = 12

shortest path 1----→3----→5---→8-----→9

BACKWARD APPROACH:

$$\text{cost}(i, J) = \min_{\substack{l \in v_{i-1} \\ <l,j> \in E}} \{\text{cost}(i-1, l) = c(l, J)\}$$

cost (5, 9) = min {cost (4, 7) + c (7, 9), cost (4, 8) + c (8, 9)}
= min {cost (4, 7) + 7, cost (4, 8) + 3}

cost (4, 7) = min {cost (3, 4) + c (4, 7), cost (3, 5) + c (5, 7),cost (3, 6) +c(6,7)}
= min {cost (3, 4) + 1, cost (3, 5) + 6, cost (3, 6) + 6}

cost (3, 4) = min {cost (2, 2) + c (2, 4), cost (2, 3) + c (3, 4)}
= min {cost (2, 2) + 3, cost (2, 3) + 6}

cost (2, 2) = min {cost (1, 1) + c (1, 2)} = min {0 + 5} = 5

cost (2, 3) = min (cost (1, 1) + c (1, 3)} = min {0 + 2} = 2

Therefore, cost (3, 4) = min {5 + 3, 2 + 6} = min {8, 8} = 8

cost (3, 5) = min {cost (2, 3) + c (3, 5)} = min {2 + 5} = 7

cost (3, 6) = min {cost (2, 2) + c (2, 6), cost (2, 3) + c (3, 6)}
= min {5 + 5, 2 + 8} = 10

Therefore, cost (4, 7) = min {8 + 1, 7 + 6, 10 + 6} = 9

cost (4, 8) = min {cost (3, 4) + c (4, 8), cost (3, 5) + c (5, 8),
cost (3, 6) + c (6, 8)}
= min {8 + 4, 7 + 2, 10 + 2} = 9

Therefore, cost (5, 9) = min {9 + 7, 9 + 3} = 12
Shortest path 1----→3----→5---→8-----→9

0/1 Knapsack problem

We are given n objects and a knapsack. Each object i has a positive weight w_i and a positive value V_i. The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables x1, x2, ... , xn. A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it.

Let us assume that, decisions on the x_i are made in the order x_n, x_{n-1}, ...1. Following a decision on xn, we may be in one of two possible states: the capacity

remaining in m – w_n and a profit of p_n has accrued. It is clear that the remaining decisions x_{n-1}, , x_1 must be optimal with respect to the problem state resulting from the decision on xn. Otherwise, x_n, , x_1 will not be optimal. Hence, the principal of optimality holds.

The problem states-

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.

Knapsack Problem

Other method:

$$F_n(m) = \max\{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \quad -- \quad 1$$

For arbitrary $f_i(y)$, i > 0, this equation generalizes to:

$$F_i(y) = \max\{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \quad -- \quad 2$$

Equation-2 can be solved for $f_n(m)$ by beginning with the knowledge $f_o(y) = 0$ for all y and $f_i(y) = -\propto$, y < 0. Then $f_1, f_2, \ldots f_n$ can be successively computed using equation–2.

When the w_i's are integer, we need to compute $f_i(y)$ for integer y, $0 \le y \le m$. Since $f_i(y) = -\propto$ for y < 0, these function values need not be computed explicitly. Since each f_i can be computed from $f_i - 1$ in Θ (m) time, it takes Θ (m n) time to compute f_n. When the wi's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \le y \le m$. So, f_i cannot be explicitly computed for all y in this range. Even when the w_i's are integer, the explicit Θ (m n) computation of f_n may not be the most efficient computation. So, we explore **an alternative method for both cases.**

The $f_i(y)$ is an ascending step function; i.e., there are a finite number of y's, $0 = y_1 < y_2 < \ldots < y_k$, such that $f_i(y_1) < f_i(y_2) < \ldots < f_i(y_k)$; $f_i(y) = -\propto$, $y < y_1$; f_i

$(y) = f(y_k)$, $y \geq y_k$; and $f_i(y) = f_i(y_j)$, $y_j \leq y \leq y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f(y_j), y_j) \mid 1 \leq j \leq k\}$ to represent $f_i(y)$. Each number of S^i is a pair (P, W), where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 = \{(0, 0)\}$. We can compute S^{i+1} from S_i by first computing:

$$_1S^i = \{(P, W) \mid (P - p_i, W - w_i) \in S^i\}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S^i_1 together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of equation-2. Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j).

Example 1:

Consider the knapsack instance n = 3, $(w_1, w_2, w_3) = (2, 3, 4)$, $(P_1, P_2, P_3) = (1, 2, 5)$ and M = 6.

Solution:

Initially, $f_o(x) = 0$, for all x and $f_i(x) = -\infty$ if $x < 0$.

$F_n(M) = \max\{f_{n-1}(M), f_{n-1}(M - w_n) + p_n\}$

$F_3(6) = \max(f_2(6), f_2(6 - 4) + 5\} = \max\{f_2(6), f_2(2) + 5\}$

$F_2(6) = \max(f_1(6), f_1(6 - 3) + 2\} = \max\{f_1(6), f_1(3) + 2\}$

$F_1(6) = \max(f_0(6), f_0(6 - 2) + 1\} = \max\{0, 0 + 1\} = 1$

$F_1(3) = \max(f_0(3), f_0(3 - 2) + 1\} = \max\{0, 0 + 1\} = 1$

Therefore, $F_2(6) = \max(1, 1 + 2\} = 3$

$F_2(2) = \max(f_1(2), f_1(2 - 3) + 2\} = \max\{f_1(2), -\infty + 2\}$

$F_1(2) = \max(f_0(2), f_0(2 - 2) + 1\} = \max\{0, 0 + 1\} = 1$

$F_2(2) = \max\{1, -\infty + 2\} = 1$

Finally, $f_3(6) = \max\{3, 1 + 5\} = 6$

Other Solution:

For the given data we have:

$S^0 = \{(0, 0)\}$; $S^0 = \{(1, 2)\}$

$S^1 = (S^0 \cup S^0) = \{(0, 0), (1, 2)\}$

X - 2 = 0 => x = 2. y – 3 = 0 => y = 3
X - 2 = 1 => x = 3. y – 3 = 2 => y = 5

$S^{1}{}_{1}$ = {(2, 3), (3, 5)}

S^2 = (S^1 U $S^1{}_1$) = {(0, 0), (1, 2), (2, 3), (3, 5)}

X – 5 = 0 => x = 5.	y – 4 = 0 => y = 4
X – 5 = 1 => x = 6.	y – 4 = 2 => y = 6
X – 5 = 2 => x = 7.	y – 4 = 3 => y = 7
X – 5 = 3 => x = 8.	y – 4 = 5 => y = 9

$S^{2}{}_{1}$ = {(5, 4), (6, 6), (7, 7), (8, 9)}

S^3 = (S^2 U S^2) = {(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8, 9)}

By applying Dominance rule,

S^3 = (S^2 U S^2) = {(0, 0), (1, 2), (2, 3), (5, 4), (6, 6)}

From (6, 6) we can infer that the maximum Profit $\sum p_i x_i = 6$ and weight $\sum x_i w_i = 6$

0/1 knapsack problem

* we are given 'n' objects & a knapsack or a bag in which the object 'i' has weight 'w_i' is to be placed
* The knapsack has capacity 'M'
* The profit $p_i x_i$ can be earned, If an object 'i' has to be placed into the knapsack
* the objective of the knapsack problem is to fill the knapsack with max profitable object & the sum of wt of the object is lessthan or equal to the capacity of the knapsack i.e.,

maximize $\sum_{i=1}^{n} p_i x_i$ subject to be constraint $\sum_{i=1}^{m} w_i x_i \le M$ where $1 \le i \le n$, n is total no. of object $x_i = 0$ or 1.

Purging rule or Dominance Rule:

If S_{i+1} contain (P_J, w_J) & (P_K, w_K), these 2 pairs $\in$ $(P_J \le P_K)$ & $(w_J \ge w_K)$ then (P_J, w_J) pair can be Eliminated. The purging rule is also called Dominance Rule.

In purging basically the Dominance tuple get Purged i.e., remove the pair with less p & more wt.

Eg: (3,5) & (5,4)
(P_J, w_J) & (P_K, w_K)

$P_J \le P_K$ && $(w_J \ge w_K)$
$3 \le 5$ && $5 \ge 4$ ✓

Now we remove the pair (P_J, w_J) i.e., (3,5) pair.

Eg: solve knapsack problem $m = 6$ & $n = 3$ Let P_i, w_i are shown below

i	1	2	3
P_i	1	2	5
w_i	2	3	4

Sol:
we have to build the sequence of decision S^0, S^1, S^2, S^3

$S^0 = \{(0,0)\}$. Initially

$S^1 = \{$ select next (P, w) pair $\}$ & add it with S^0 $= \{(0,0),(1,2)\}$ $= \{(1,2)\}$

{merge S^0 & S_1^0}

$S^1 = S^0 \cup S_1^0 = \{(0,0)\} \cup \{(1,2)\} = \{(0,0),(1,2)\}$

$S_1^1 = \{$select next (P,w) pair & added with $S^1\}$

$= \{$ ~~(2,3)~~ $(2+0, 3+0), (2+1, 3+2)\}$.

$= \{(2,3),(3,5)\}$

merge S^0 & S_1^1

$S^2 = S^1 \cup S_1^1 = \{(0,0),(1,2)\} \cup \{(2,3),(3,5)\}$

$= \{(0,0),(1,2),(2,3),(3,5)\}$

$S_1^2 = \{$next, we have to select the ith pair & add it to be $S^2\}$

$= \{(5+0, 4+0), (5+1, 4+2), (5+2, 4+3), (5+3, 4+5)\}$

$= \{(5,4),(6,6),(7,7),(8,9)\}$

$S^3 = S^2 \cup S_1^2 = \{(0,0),(1,2),(2,3),(3,5)\} \cup \{(5,4),(6,6),(7,7)(8,9)\}$

$= \{(0,0),(1,2),(2,3),(3,5),(5,4),(6,6),(7,7)(8,9)\}$

~~Spec~~ consider The pair $(7,7)$ & $(8,9)$ i the weight are 7 & 8 are ~~not~~ greater then the capacity of Knapsack $m=6$, so we can remove these 2 pair from S^3

$S^3 = \{(0,0),(1,2),(2,3),(3,5),(5,4),(6,6)\}$

By applying the purging rule, the pair (3,5) can be removed

Purging rule (P_J, w_J) && (P_K, w_K) then $(P_J \le P_K)$ && $(w_J \ge w_K)$ then elimatie (P_J, w_J) pair

Consider (3,5) & (5,4) pairs $(P_J, w_J), (P_K, w_K)$

$(P_J \le P_K)$ && $(w_J \ge w_K)$

$(3 \le 5)$ && $(5 \ge 4)$ ✓

So we have eliminate (3,5) pair from S^3

$S^3 = \{(0,0), (1,2), (2,3), (5,4), (6,6)\}$

As $m=6$, we will find tuple

$(6,6) \in S^3$ Select 3rd object pair (5,4)

$(6,6) \in (S^{i-1}) S^2$ Now Set $x_3 = 1$

$(6,6) \notin S^1$ ✓ $x_3 = 1$

$(6-P_3, 6-w_3) \in S^1$ ✓

Now $(1,2) \in S^1$ ✓

$(1,2) \in S^1$ $x_2 = 0$

$(1,2) \notin S^0$ $x_1 = 1$

$\therefore$ optimal solution $= (1,0,1) = (x_1, x_2, x_3)$

max profit $= \sum_{i=1}^{3} P_i x_i$

$= 1 \times 1 + 2 \times 0 + 5 \times 1 = 6$

weight $= \sum_{i=1}^{3} w_i x_i$

$= 2 \times 1 + 3 \times 0 + 4 \times 1 = \underline{\underline{6}}$

<u>0/1 knapsack Alg</u>

$S^0 \to 32$
$S^1 \to 2$
$S^2 \to 2$

```
Algorithm knapsack (P, w, n, m)
{
  S^0 = {0,0};
  for i=1 to n-1 do  → O(n-1)
  {                    ↓
                       O(n)
    S_1^{i-1} = { (P, w) | (P-P_i, w-w_i) ∈ S^{i-1} & w ≤ m};
    S^i := mergepurge (S^{i-1}, S_1^{i-1});
  }
  (Px, wx) = last pair in S^{n-1};
  (Py, wy) = (P^1 + P_n, w^1 + w_n) where w^1 is the
largest w in any pair in S^{n-1} such that
  w + w_n ≤ m;
  if (Px > Py) then x_n := 0;
  else x_n := 1;
  Trace BackFor(x_{n-1} ... x_1);
}
```

<u>Other method to slove 0/1 knapsack problem</u>

Consider-

- Knapsack weight capacity = w
- Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

<u>Step-1:</u>

- Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0th row and 0th column with zeroes as shown-

	0	1	2	3		W
0	0	0	0	0	.....	0
1	0					
2	0					
	.....					
n	0					

T-Table

Step-2:

- Start filling the table row wise top to bottom from left to right.

Use the following formula-

$$V(i, w) = \max \{ V(i-1, w), V(i-1, w - weight_i) + P(i) \}$$

Here, V(i , w) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of w. This step leads to completely filling the table. Then, value of the last box represents the maximum possible value that can be put into the knapsack.

Step-3:

To identify the items that must be put into the knapsack to obtain that maximum profit,

- Consider the last column of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

Algorithm

Dynamic-0-1-knapsack (v, w, n, W)

```
for w = 0 to W do
   c[0, w] = 0

for i = 1 to n do
   c[i, 0] = 0
   for w = 1 to W do
      if wi ≤ w then
         if vi + c[i-1, w-wi] then
                  c[i, w] = vi + c[i-1, w-wi]
```

```
    else c[i, w] = c[i-1, w]
else
    c[i, w] = c[i-1, w]
```

Time Complexity

- Each entry of the table requires constant time θ(1) for its computation.
- It takes θ(nw) time to fill (n+1)(w+1) table entries.
- It takes θ(n) time for tracing the solution since tracing process traces the n rows.
- Thus, overall θ(nw) time is taken to solve 0/1 knapsack problem using dynamic programming.

Consider the problem

$m = 8 \qquad P = \{1, 2, 5, 6\}$

$n = 4 \qquad w = \{2, 3, 4, 5\}$

$x_i = 0/1 \qquad x = \{1, 0, 0, 1\}$

$2^4 \Rightarrow 2^n$

Complexity $= O(2^n)$

0 0 0 0
1 1 1 1
0 0 0 1
1 0 0 0
1 1 0 0

Tabulation method

Weight →

P_i	w_i	V	0	1	2	3	4	5	6	7	8
1	2	0	0	0	0	0	0	0	0	0	0
2	3	1	0	0	1	1	1	1	1	1	1
5	4	2	0	0	1	2	2	3	3	3	3
6	5	3	0	0	1	2	5	5	6	7	7
		4	0	0	1	2	5	6	6	7	8

$V[i, w] = \max\{V[i-1, w], V[i-1, w - w[i]] + P[i]\}$

$V[4, 1] = \max\{\underbrace{V[3, 1]}_{0}, \underbrace{V[3, 1-5]}_{3, -4 \text{ undefined}} + 6\}$

$V[4, 5] = \max[\underbrace{V[3, 5]}_{5}, \underbrace{V[3, 5-5]}_{3, 0} + 6$

$0 + 6$

$$V[4,6] = \max\{V[3,6],\ V[3,\overset{1}{\cancel{6-5}}] + 6\}$$
$$= \max\{6,\ 0+6\} = 6$$

$$V[4,7] = \max\{\underset{7}{V[3,7]},\ \underset{1+6}{V[3,\overset{2}{\cancel{7-5}}] + 6}\}$$
$$= 7$$

$$V[4,8] = \max\{V[3,8],\ V[3,\overset{3}{\cancel{8-5}}] + 6\}$$
$$= \max\{7,\ 2+6\}$$
$$= 8$$

Ans $\{x_1, x_2, x_3, x_4\}$ $\{0\ \ 1\ \ 0\ \ 1\}$

$P_4 \quad P_4$
$8 - 6 = 2$
$2 - 2 = 0$

2) Consider the instance $M = 15$, $n = 4$, $(P_1, P_2, P_3, P_4) = (10, 10, 12, 18)$ and $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$. Solve 0/1 knapsack problem by using dynamic programming approach.

P_i	w_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	2	1	0	0	10	10	10	10	10	10	10	10	10	10	10	10	10
10	4	2	0	0	10	10	10	10	20	20	20	20	20	20	20	20	20
12	6	3	0	0	10	10	10	10	12	12	22	22	22	22	22	22	22
18	9	4	0	0	10	10	10	10	12	12	12	18	18	28	28	28	28	30

Ans $\{x_1, x_2, x_3, x_4\}$ $\{0, 0, 1, 1\}$

$30 - 18 = 12$
$12 - 12 = 0$

Example:

Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach. Consider-

n = 4, w = 5 kg, (w1, w2, w3, w4) = (2, 3, 4, 5), (b1, b2, b3, b4) = (3, 4, 5, 6)

Longest Common Sequence (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 subsequences 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)

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.

LCS (Longest common subsequence)

* LSC problem have 2 sequences given, we have to find the longest possible subsequence which is common in both sequences

* Subsequence: It is a sequence that appear, in the same relative order, but not necessarily continguous. So we can skip some character in the sequence inorder to make the common subsequence

Eg: S1: abcdefghij
S2: cdei

cdei
dei
ei
} here longest common subsequence
cdei

Eg: 1. bcefgk
S2. cfgk

cfgk ✓
fgk
gk

Eg: S1: dabcdef
S2 cade → path across is not allowed in LCS

To solve LCS by using Recursive tree

```
Algorithm LCS(i, j)
{
    if (A[i]==='\0' || B[j]==='\0')
        return 0;
    else if (A[i] == B[j])
        return 1+LCS(i+1, j+1);
    else
        return max(LCS(i+1, j), LCS(i, j+1));
}
```

Time complexity $O(2^{m+n})$

Eg: S1: bd S2: a b c d

∴ LCS is 2
i.e., bd

* **NOTE:** Time taken by a dynamic approach is time taken to fill the table $O(mn)$. where as Recursion method. time taken is $O(2^{m+n})$

Dynamic approach

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i]==y[j] \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

Point an arrow to max(LCS[i-1, j], LCS[i, j-1])

Point an arrow to LCS[i][j]

Eg: S1: Stone S2: Longest

		o	h	o	n	g	e	s	t
	0	0	0	0	0	0	0	0	0
o	0	0	0	0	0	0	0	1	1
s	0	0	0	0	0	0	0	0	2
t	0	0	0	0	0	0	0	0	2
o	0	0	1	1	1	1	1	1	2
n	0	0	1	1	2	2	2	2	2
e	0	0	1	2	2	3	3	3	3

o n g

Step1: In dynamic approach to check 1st character of sequence if it is equal the LCS[i-1, j-1]+1 & point to arrow to LCS[i][j] otherwise to find max {LCS[i-1, j], LCS[i, j-1]} point out an arrow to max of LCS

Step2: 1st & 2nd columns are filled with ze

3. S1[0] == S2[0] then find out max

max { LCS [pervious col], LCS [pervious row] } = 0

Here max value from pervious col & pervious row element is filling the cell. point out an arrow to cell with max value

4. ⋮

8. If → Character corresponding to current row & columns are matching then S1[0] == S2[5] (S == S) is matching, then fill the current cell by adding one of diagonal element. point out an arrow to diagonal cell.

⋮

∴ Finally, Longest common subsequence is 'Ong'

Procedure LCS-LENGTH takes two sequences X and Y as inputs. It stores the $c[i, j]$ values in a table $c[0...m, 0...n]$ whose entries are computed in row-major order, that is first row of c is filled from left to right, then the second row and so on. It also maintains the table $b[1..m, 1..n]$ to simplify construction of an optimal solution. The procedure returns the b and c tables. $c[m, n]$ contains the length of LCS of X and Y.

```
LCS-LENGTH (X, Y)
m ← lenght[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to m
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
        do if x_i = y_j
            then c[i, j] ← c[i − 1, j − 1] + 1
                b[i, j] ← "↖"
            else if c[i − 1, j] ≥ c[i, j − 1]
                then c[i, j] ← c[i − 1, j]
                    b[i, j] ← "↑"
                else c[i, j] ← c[i, j − 1]
                    b[i, j] ← "←"
return c and b.
```

Example. *Given two sequences $X[1..m]$ and $Y[1..n]$. Find the longest subsequence common to both. Note : not substring, subsequence.*

So if x : A B C B D A B

y : B D C A B A

the longest subsequence turns out to be B C B A. There could be other subsequences of the longest length, the algorithm will get one of them.

Time complexity o(mn)

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

i \ j		0	1	2	3	4	5	6
		y_j	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

String edition:

String Editing

Problem statement:

the given & string $x = x_1 x_2 \cdots x_n$ & $y = y_1, y_2 \ldots y_m$ where x_i is $1 \le i \le m$ y_j is $1 \le j \le n$ are members of finite set of symbols.

* To Transform x, to y using a sequence of edit operation of x.
* Editing operations are Insert (I), Delete (D), Replace (R)
* Cost of sequence of operation is sum of cost of individual operation in sequence
* The problem of string editing is to identify minimum cost sequence of Editing operation that will transform x to y
* $D(x_i)$ be cost of deleting the symbol x_i from x
* $I(y_j)$ be cost of Inserting the symbol y_j to x
* $C[x_i, y_j)$ be cost of Replacing symbol x_i to x into y_j to y.

* To slove string edition by using following formula.

Recurrence

$$cost(i,j) = \min \begin{cases} cost(i-1,j) + D(x_i), \\ cost(i-1,j-1) + C(x_i, y_j), \\ cost(i,j-1) + I(y_j) \end{cases}$$

Eg: consider $X = aabab$ $Y = babb$

Sol:

i \ j		0	1	2	3	4
		0	b	a	b	b
0	0	0	1	2	3	4
1	a	1	2	1	2	3
2	a	2	3	2	3	4
3	b	3	2	3	2	4
4	a	4	3	2	3	4
5	b	5	4	3	2	3

→ Insert

↓ Delete

↘ change

$$cost(i,j) = \begin{cases} 0 & i=j=0 \\ cost(i-1,0) + D(x_i) & i>0, j=0 \\ cost(0,j-1) + I(y_j) & i=0, j>0 \\ cost(i,j) & i>0, j>0 \end{cases}$$

$$cost(i,j) = \min \begin{cases} cost(i-1,j) + D(x_i) \\ cost(i-1,j-1) + C(x_i, y_j) \\ cost(i,j-1) + I(y_j) \end{cases}$$

$D(x_i)=1$ $C(x_i, y_j)=2$ $I(y_j)=1$

Delete operation Replace operation Insert operation

Step 1: compute for case of $i=0, j>1$, $i>1, j=0$

Initial $cost(0,0)=0$

$cost(0,1) = cost(i, j-1) + I(y_j)$

$= cost(0, 1-1) + I(y_1) = cost(0,0) + I(y_1)$

$cost(i, j-1) + I(y_j) = 0+1=1$

$cost(0,2) = cost(0, 2-1) + I(y_2)$

$= cost(0,1) + I(y_2) = 1+1=2$

oing

$$cost(3,0) = cost(0, 3-1) + I(y_3)$$
$$= cost(0,2) + 1 = 2+1 = 3$$
$$cost(4,0) = cost(0, 4-1) + I(y_4)$$
$$= cost(0,3) + I(y_4) = 3+1 = 4$$

Now, compute $i>1, j=0$

$$cost(1,0) = cost(i-1,j) + D(x_i)$$
$$= cost(0,0) + D(x_1) = 0+1 = 1$$
$$cost(2,0) = cost(i-1,j) + D(x_i)$$
$$= cost(1,0) + 1 = 1+1 = 2$$

nge

$$cost(i,j) = \begin{cases} 0 & i=j=0 \\ cost(i-1,0) + D(x_i) & i>0, j=0 \\ cost(0,j-1) + I(y_j) & i=0, j>0 \\ cost(i,j) & i>0, j>0 \end{cases}$$

$$cost(i,j) = \min \begin{cases} cost(i-1,j) + D(x_i) \\ cost(i-1,j-1) + C(x_i, y_j) \\ cost(i,j-1) + I(y_j) \end{cases}$$

$$cost(3,0) = cost(i-1,j) + D(x_i)$$
$$= cost(2,0) + D(x_3)$$
$$= 2+1 = 3$$
$$cost(4,0) = cost(i-1,j) + D(x_i)$$
$$= cost(3,0) + D(x_4) = 3+1 = 4$$
$$cost(5,0) = cost(i-1,j) + D(x_i)$$
$$= cost(4,0) + D(x_5) = 4+1 = 5$$

Step 2: Compute row major order

Row 1:

$$cost(1,1) = cost(1,1) \quad i>0, j>0$$

$$= \min \begin{cases} cost(i-1,j) + D(x_i) \\ cost(i-1,j-1) + C(x_i, y_i) \\ cost(i,j-1) + I(y_j) \end{cases}$$

$$= \min \begin{cases} cost(1-1,1) + D(x_1) \\ cost(1-1,1-1) + C(x_1, y_1) \\ cost(1,1-1) + I(y_1) \end{cases}$$

$$= \min\{1+1, 0+2, 1+1\} = 2$$

$$cost(1,2) = \quad i>0, j>0$$

$$\min \begin{cases} cost(i-1,j) + D(x_i) \\ cost(i-1,j-1) + C(x_i, y_j) \\ cost(i,j-1) + I(y_j \end{cases}$$

$$= \min \begin{cases} cost(1-1,2) + D(x_1) \\ cost(1-1,2-1) + C(x_1, y_2) \\ cost(1,2-1) + I(y_2) \end{cases}$$

$$= \min\{2+1, 1+0, 2+1\} = 1$$

a == a
same so replace till to '0'

$$cost(1,3) = \min \begin{cases} cost(0,3) + D(x_1) \\ cost(0,2) + C(x_1, y_3) \\ cost(1,2) + I(y_3) \end{cases}$$

$$= \min\{3+1, 2+2, 1+1\} = 2$$

$$cost(1,4) = \min \begin{cases} cost(0,4) + D(x_1) \\ cost(0,3) + C(x_1, y_4) \\ cost(1,3) + I(y_4) \end{cases}$$

$$= \min\{4+1, 3+2, 2+1\} = 3$$

ROW 2:

$$cost(2,1) = \min\begin{Bmatrix} cost(1,1)+D(x_2) \\ cost(1,1)+C(x_2,y_1) \\ cost(2,0)+I(y_1) \end{Bmatrix}$$

$$= \min\{2+1, 1+2, 2+1\} = 3$$

$$cost(2,2) = \min\begin{Bmatrix} cost(1,2)+D(x_2) \\ cost(1,1)+C(x_2,y_2) \\ cost(2,1)+I(y_2) \end{Bmatrix} \rightarrow$$

$x_2 == y_2$
$a == a$
So fill replace value is 0

$$= \min\{1+1, 2+0, 3+1\} = 2$$

$$cost(2,3) = \min\begin{Bmatrix} cost(1,3)+D(x_2) \\ cost(1,2)+C(x_2,y_3) \\ cost(2,2)+I(y_3) \end{Bmatrix}$$

$$= \min\{2+1, 1+2, 2+1\} = 3$$

$$cost(2,4) = \min\begin{Bmatrix} cost(1,4)+D(x_2) \\ cost(1,3)+C(x_2,y_4) \\ cost(2,3)+I(y_4) \end{Bmatrix}$$

$$= \min\{3+1, 2+2, 3+1\} = 4$$

ROW 3:

$$cost(3,1) = \min\begin{Bmatrix} cost(2,1)+D(x_3) \\ cost(2,0)+C(x_3,y_1) \\ cost(3,0)+I(y_1) \end{Bmatrix}$$

$$= \min\{3+1, 2+0, 3+1\} = 2$$

$$cost(3,2) = \min\begin{Bmatrix} cost(2,2)+D(x_3) \\ cost(2,1)+C(x_3,y_2) \\ cost(3,1)+I(y_2) \end{Bmatrix}$$

$$= \min\{2+1, 2+2, 2+1\} = 3$$

$$cost(3,3) = \min \begin{cases} cost(2,3) + D(x_3) \\ cost(2,2) + C(x_3, y_3) \\ cost(3,2) + I(y_3) \end{cases}$$

$$= \min\{3+1,\ 2+0,\ 3+1\} = 2$$

$$cost(3,4) = \min \begin{cases} cost(2,4) + D(x_3) \\ cost(2,3) + C(x_3, y_4) \\ cost(2,3) + I(y_4 \end{cases}$$

$$= \min\{4+1,\ 3+2,\ 3+1\} = 4$$

<u>ROW 4</u>:

$$cost(4,1) = \min \begin{cases} cost(3,1) + D(x_4) \\ cost(3,0) + C(x_4, y_1) \\ cost(4,0) + I(y_1) \end{cases}$$

$$= \min\{2+1,\ 3+2,\ 4+1\} = 3$$

$$cost(4,2) = \min \begin{cases} cost(3,2) + D(x_4) \\ cost(3,1) + C(x_4, y_2) \\ cost(4,1) + I(y_2) \end{cases}$$

$$= \min\{3+1,\ 2+0,\ 4+1\} = 2$$

$$cost(4,3) = \min \begin{cases} cost(3,3) + D(x_4) \\ cost(3,2) + C(x_4, y_3) \\ cost(4,2) + I(y_3) \end{cases}$$

$$= \min\{2+1,\ 3+2,\ 2+1\} = 3$$

$$cost(4,4) = \min \begin{cases} cost(3,4) + D(x_4) \\ cost(3,3) + C(x_4, y_4) \\ cost(4,3) + I(y_4) \end{cases}$$

$$= \min\{4+1,\ 2+2,\ 3+1\} = 4$$

Row 5:

$$cost(5,1) = \min \begin{Bmatrix} cost(4,1) + D(x_5) \\ cost(4,0) + C(x_5, y_1) \\ cost(5,0) + I(y_1) \end{Bmatrix}$$

$$= \min\{3+1, 4+0, 5+1\} = 4$$

$$cost(5,2) = \min \begin{Bmatrix} cost(4,2) + D(x_5) \\ cost(4,1) + D(x_5, y_2) \\ cost(5,1) + I(y_2) \end{Bmatrix}$$

$$= \min\{2+1, 3+2, 4+1\} = 3$$

$$cost(5,3) = \min \begin{Bmatrix} cost(4,3) + D(x_5) \\ cost(4,2) + C(x_5, y_4) \\ cost(5,2) + I(y_4) \end{Bmatrix}$$

$$= \min\{3+1, 2+0, 3+1\} = 2$$

$$cost(5,4) = \min \begin{Bmatrix} cost(4,4) + D(x_5) \\ cost(4,3) + C(x_5, y_4) \\ cost(5,3) + I(y_4) \end{Bmatrix}$$

$$= \min\{4+1, 3+0, 2+1\} = 3$$

$cost(n,m) \rightarrow$ minimum cost of sequence for ~~to~~ transforming x into y

$cost(5,4) = 3 \rightarrow$ This is optimal sol for given string

3 operations are 2 Delete operation &
1 Insert "

D

a a b a b I

b a b b

```
Algorithm StringEdition(S1, S2)
{
  int [][] disttable = new int (S1.length()+1
                                S2.length+1)
  int r = ~~str~~S1.length()+1;
  int c = S2.length()+1;
  for m:=0 to r
  {
    for n:=0 to c
    {
      if (m==0)
        disttable[m][n]=m;
      else if (n==0)
        disttable[m][n]=n;
      elseif (S1.charAt(m-1)==str2.charAt[n-1])
        disttable[m][n]=disttable[m-1][n-1];
      else
        disttable[m][n]=
          1+ min {disttable[m-1][n-1],
                  disttable[m][n-1],
                  disttable[m-1],[n-1]}.
    }
  }
  return disttable[R-1][c-1];
}  Time complexity is O(m.n)
```

Matrix Chain Multiplication

- It is a Method under Dynamic Programming in which previous output is taken as input for next. Here, Chain means one matrix's column is equal to the second matrix's row [always].

In general:

If A = $\lfloor a_{ij} \rfloor$ is a p x q matrix

B = $\lfloor b_{ij} \rfloor$ is a q x r matrix

$C = \lfloor c_{ij} \rfloor$ is a p x r matrix then

$AB = C$ if $c_{ij} = \sum_{k=1}^{q} a_{ik}\ b_{kj}$

- Given following matrices $\{A_1, A_2, A_3, \ldots A_n\}$ and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications A_1 xA_2 x,A_3 x.....x A_n.In general, for $1 \leq i \leq p$ and $1 \leq j \leq r$

$C[i, j] = \sum_{k=1}^{q} A[i, k]B[k, j]$

Example1: Let us have 3 matrices, A_1, A_2, A_3 of order (10 x 100), (100 x 5) and (5 x 50) respectively.

Three Matrices can be multiplied in two ways:

1. **A_1,(A_2,A_3):** First multiplying(A_2 and A_3) then multiplying and resultant withA_1.
2. **(A_1,A_2),A_3:** First multiplying(A_1 and A_2) then multiplying and resultant withA_3.

No of Scalar multiplication in Case 1 will be:

1. (100 x 5 x 50) + (10 x 100 x 50) = 25000 + 50000 = 75000

No of Scalar multiplication in Case 2 will be:

1. (100 x 10 x 5) + (10 x 5 x 50) = 5000 + 2500 = 7500

To find the best possible way to calculate the product, we could simply parenthesis the expression in every possible fashion and count each time how many scalar multiplication are required.

- Matrix Chain Multiplication Problem can be stated as "**find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplication is minimized".**

Number of ways for parenthesizing the matrices:

There are very large numbers of ways of parenthesizing these matrices. If there are n items, there are (n-1) ways in which the outer most pair of parenthesis can place.

(A_1) (A_2,A_3,A_4,.....A_n)

Or (A_1,A_2) (A_3,A_4 .....A_n)

Or (A_1,A_2,A_3) (A_4 .....A_n)

.....

Or(A_1,A_2,A_3.....A_{n-1}) (A_n)

It can be observed that after splitting the kth matrices, we are left with two parenthesized sequence of matrices: one consist 'k' matrices and another consist 'n-k' matrices.

Now there are 'L' ways of parenthesizing the left sublist and 'R' ways of parenthesizing the right sublist then the Total will be L.R:

$$p(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} p(k)p(n-k) & \text{if } n \geq 2 \end{cases}$$

$$c(n) = \frac{1}{n+1}\binom{2n}{n}$$

On applying Stirling's formula we have

$$c(n) = \Omega\left(\frac{4^n}{n^{1.5}}\right)$$

Which shows that 4^n grows faster, as it is an exponential function, then $n^{1.5}$.

Dynamic Programming Approach

- Let $A_{i,j}$ be the result of multiplying matrices i through j. It can be seen that the dimension of $A_{i,j}$ is p_{i-1} x p_j matrix.
- Dynamic Programming solution involves breaking up the problems into subproblems whose solution can be combined to solve the global problem.
- At the greatest level of parenthesization, we multiply two matrices

$A_{1.....n}$=$A_{1....k}$ x $A_{k+1....n}$)

Thus we are left with two questions:

- How to split the sequence of matrices?
- How to parenthesize the subsequence $A_{1.....k}$ and $A_{k+1.....n}$?

One possible answer to the first question for finding the best value of 'k' is to check all possible choices of 'k' and consider the best among them. But that it can be observed that checking all possibilities will lead to an exponential number of total possibilities. It can also be noticed that there exists only O (n^2) different sequence of matrices, in this way do not reach the exponential growth.

Step1: Structure of an optimal parenthesization:

- Our first step in the dynamic paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from an optimal solution to subproblems.

Let $A_{i....j}$ where $i \leq j$ denotes the matrix that results from evaluating the product $A_i\ A_{i+1}....A_j$.

- If $i < j$ then any parenthesization of the product $A_i\ A_{i+1}$ .....A_j must split that the product between A_k and A_{k+1} for some integer k in the range $i \leq k \leq j$. That is for some value of k, we first compute the matrices $A_{i.....k}$ & $A_{k+1....j}$ and then multiply them together to produce the final product $A_{i....j}$. The cost of computing $A_{i....k}$ plus the cost of computing $A_{k+1....j}$ plus the cost of multiplying them together is the cost of parenthesization.

Step 2: A Recursive Solution:

- Let m [i, j] be the minimum number of scalar multiplication needed to compute the matrix$A_{i....j}$.

If **i=j** the chain consist of just one matrix $A_{i....i}=A_i$ so no scalar multiplication are necessary to compute the product. Thus m [i, j] = 0 for i= 1, 2, 3....n.

If **i<j** we assume that to optimally parenthesize the product we split it between A_k and A_{k+1} where $i \leq k \leq j$. Then m [i,j] equals the minimum cost for computing the subproducts $A_{i....k}$ and $A_{k+1....j}$+ cost of multiplying them together. We know A_i has dimension p_{i-1} x p_i, so computing the product $A_{i....k}$ and $A_{k+1....j}$takes $p_{i-1}\ p_k\ p_j$ scalar multiplication, we obtain

$$\mathbf{m\ [i,j] = m\ [i, k] + m\ [k + 1, j] + p_{i-1}\ p_k\ p_j}$$

- There are only (j-1) possible values for 'k' namely k = i, i+1.....j-1. Since the optimal parenthesization must use one of these values for 'k' we need only check them all to find the best.So the minimum cost of parenthesizing the product $A_i\ A_{i+1}$.....A_j becomes

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

To construct an optimal solution, let us define s [i,j] to be the value of 'k' at which we can split the product $A_i\ A_{i+1}$ .....A_j To obtain an optimal parenthesization i.e. s [i, j] = k such that

$$\mathbf{m\ [i,j] = m\ [i, k] + m\ [k + 1, j] + p_{i-1}\ p_k\ p_j}$$

<u>Example of Matrix Chain Multiplication</u>

eg: A1 A2 A3 A4
3x2 2x4 4x2 2x5

Step1: find out optimal order sequence & optimal cost of multiply. S[i,j] be value of k at which we split the matrix mul

$$m[i,j] = \begin{cases} 0 & i==j \\ \min\limits_{i \le k < j} c[i,k] + c[k+1,j] + d_{i-1}d_k d_j & i<j \end{cases}$$

	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

m

	1	2	3	4
1	0	1	1	3
2		0	2	3
3			0	3
4				0

S

$$m[i,j] = \begin{cases} 0 & i==j \\ \min\limits_{i \le k \le j} m[i,k] + m[k+1,j] + P_{i-1}P_kP_j & \end{cases}$$

Initial $m[1,1]=0$ $m[3,3]=0$
$m[2,2]=0$ $m[4,4]=0$

calculate product of 2 matrix & find m_{ij} where different between $i-j=1$

$$M[1,2] = \min_{1 \le k < 2} \{m[1,1] + m[2,2] + P_0P_1P_2\}$$

$k=1$

$$= \min\{0+0+3*2*4\} = 24$$

$$M[2,3] = \min_{2 \le k < 3} \{m[2,2] + m[3,3] + d_1d_2d_3\}$$

$k=2$

$$= \min\{0+0+2*4*2\} = 16$$

$$m[3,4] = \min_{\substack{3 \le k < 4 \\ k=3}} \{ m[3,3] + m[4,4] + P_2 P_3 P_4 \}$$

$$= \min \{ 0 + 0 + 4 \times 2 \times 5 \} = 40$$

Step2 2. Compute product of 3 matrix

$$m[1,3] = \min_{\substack{1 \le k < 3 \\ k=1,2}} \begin{cases} m[1,1] + m[2,3] + P_0 P_1 P_3 \\ m[1,2] + m[3,3] + P_0 P_2 P_3 \end{cases}$$

$$= \min \{ 0 + 16 + 3 \times 2 \times 2 , 24 + 0 + 3 \times 4 \times 2 \}$$

$$= \min \{ 28, 48 \} = 28$$

$$m[2,4] = \min_{\substack{2 \le k < 4 \\ k=2,3}} \begin{cases} m[2,2] + m[3,4] + P_1 P_2 P_4 \\ m[2,3] + m[4,4] + P_1 P_3 P_4 \end{cases}$$

$$= 36$$

Step3: Compute product of 4 matrix

$$m[1,4] = \min_{\substack{1 \le k < 4 \\ k=1,2,3}} \begin{cases} m[1,1] + m[2,4] + P_0 P_1 P_4 \\ m[1,2] + m[3,4] + P_0 P_2 P_4 \\ m[1,3] + m[4,4] + P_0 P_3 P_4 \end{cases}$$

$$= 58$$

Optimal cost 58

m(1,4) — m(1,3) — m(1,1) A1
m(1,3) — m(2,3) A2, A3
m(1,4) — m(4,4) A4

(A1 A2 A3 A4)
(A1 A2 A3) A4
(A1 (A2 A3) A4) → optimal order sequence

Example. *We are given the sequence {4, 10, 3, 12, 20, 7}. The matrices have sizes 4×10, 10×3, 3×12, 12×20, 20×7. We need to compute M[i, j], 0 ≤ i, j ≤ 5. We know M[i, i] = 0 for all i.*

	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

We proceed, working away from the diagonal. We compute the optimal solutions for products of 2 matrices.

	1	2	3	4	5
1	0	120			
2		0	360		
3			0	720	
4				0	1680
5					0

{4, 10, 3, 12, 20, 7}

Now products of 3 matrices

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 840 \end{cases} = 264$$

Here P_0 to P_5 are Position and M_1 to M_5 are matrix of size (p_i to p_{i-1})
On the basis of sequence, we make a formula

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10 \cdot 12 \cdot 20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10 \cdot 3 \cdot 20 = 1320 \end{cases} = \mathbf{1320}$$

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3 \cdot 20 \cdot 7 = 1140 \\ M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3 \cdot 12 \cdot 7 = 1932 \end{cases} = \mathbf{1140}$$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 4 matrices

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0 p_3 p_4 = 264 + 0 + 4 \cdot 12 \cdot 20 = 1224 \\ M[1,2] + M[3,4] + p_0 p_2 p_4 = 120 + 720 + 4 \cdot 3 \cdot 20 = 1080 \\ M[1,1] + M[2,4] + p_0 p_1 p_4 = 0 + 1320 + 4 \cdot 10 \cdot 20 = 2120 \end{cases} = \mathbf{1080}$$

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10 \cdot 20 \cdot 7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10 \cdot 12 \cdot 7 = 2880 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10 \cdot 3 \cdot 7 = 1350 \end{cases} = \mathbf{1350}$$

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

Now products of 5 matrices

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4 \cdot 20 \cdot 7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4 \cdot 12 \cdot 7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4 \cdot 3 \cdot 7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4 \cdot 10 \cdot 7 = 1630 \end{cases} = \mathbf{1344}$$

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

$\Rightarrow$

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

To print the optimal parenthesization, we use the PRINT-OPTIMAL-PARENS procedure.

```
PRINT-OPTIMAL-PARENS (s, i, j)
if i = j
    then print "A"
    else print "("
        PRINT-OPTIMAL-PARENS (s, i, s[i, j])
        PRINT-OPTIMAL-PARENS (s, s[i, j] + 1, j)
        print ")"
```

Now for optimal parenthesization , Each time we find the optimal value for M[i,j] we also store the value of k that we used. If we did this for the example, we would get

1	2	3	4	5	
0	120/1	264/2	1080/2	1344/2	1
	0	360/2	1320/2	1350/2	2
		0	720/3	1140/4	3
			0	1680/4	4
				0	5

The k value for the solution is 2, so we have $((A_1\ A_2)(A_3\ A_4\ A_5))$. The first half is done. The optimal solution for the second half comes from entry $M[3,5]$.The value of k here is 4, so now we have $((A_1\ A_2)((A_3\ A_4)\ A_5))$.Thus the optimal solution is to parenthesize $((A_1\ A_2)((A_3\ A_4)\ A_5))$.

Algorithm

The algorithm first computes $m[i,j] \leftarrow 0$ for $i = 1,2,3..n$, the minimum costs for chains of length 1.

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
    do m[i, i] ← 0
for l ← 2 to n          ▷ l is the chain length
    do for i ← 1 to n − l + 1
        do j ← i + l − 1
            m[i, j] ← ∞
            for k ← i to j − 1
                do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                    if q < m[i, j]
                        then m[i, j] ← q
                            s[i, j] ← k
return m and s
```

It then uses recurrence to compute $m[i,i+1]$ for $i = 1,2,\ldots n-1$ (the minimum cost of length 2) during the first execution of the loop and in the second time, it computes $m[i,i+2]\ i = 1,2,\ldots n-2$ (the minimum cost of length 3) and so on.

Since we have defined $m[i,j]$ only for $i < j$, only the portion of the table m strictly above the main diagonal is used. MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row. An entry $m[i,j]$ is computed using the products $p_{i-1}\ p_k\ p_j$ for $k = i, i+1, \ldots j-1$.

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm.

The following recursive procedure computes the matrix-chain product $A_{i..j}$ given the matrices $A = \langle A_1\ A_2 \ldots A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. The initial call is MATRIX-CHAIN-MULTIPLY $(A, s, 1, n)$.

```
MATRIX-CHAIN-MULTIPLY (A, s, i, j)
  if j > i
  then X ← MATRIX-CHAIN-MULTIPLY (A, s, i, s[i, j])
       Y ← MATRIX-CHAIN-MULTIPLY (A, s, s[i, j] + 1, j)
       return MATRIX-MULTIPLY (X, Y)
  else return A_i
```

Example of Finding the Multiplication Sequence

Consider $n = 6$. Assume that the array $s[1..6, 1..6]$ has been computed. The multiplication sequence is recovered as follows.

$s[1,6] = 3 \quad (A_1\ A_2\ A_3)(A_4\ A_5\ A_6)$

$s[1,3] = 1 \quad (A_1(A_2\ A_3))$

$s[4,6] = 5 \quad ((A_4\ A_5)A_6)$

Hence the final multiplication sequence is

$$(A_1(A_2 A_3))((A_4 A_5)A_6)$$

TRAVELLING SALES PERSON PROBLEM

A traveler needs to visit all the cities from a list, where distances between allthe 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?

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 canbe 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 thiswill 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 k Є S, let g(S,k) 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 g(S, 1) = ∝ since the path cannot start and end at 1. Now, let express g(S, k) in terms of smaller sub-problems. We need to start at 1 and end at k. We should select the next city by applying the general formula,

$$g(i,S) = \min_{k \in s} \{C_{ik} + g(k, S-\{k\})\}$$

Algorithm:

Traveling-Salesman-Problemg ({1}, 1) = 0
    for s = 2 to n do
        for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
        g (S, 1) = ∞
        for all k Є S and k ≠ 1
            g (i,S) = min { c(i, k) + g (k, S – {k}) for i Є S and i ≠ k}
Return min g ({1, 2, 3, …, n}, k) + c(i, k)

Analysis

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

Example:

For the following graph find minimum cost tour for the traveling salespersonproblem:

Solution:

$$g(i,S) = \min_{k \in S} \{ c_{ik} + g(k, S - \{k\}) \}$$

$$g(1, \{2,3,4\}) = \min_{k \in \{2,3,4\}} \{ c_{ik} + g(k, \{2,3,4\} - \{k\}) \}$$

a- cost function

Here start vertex 1 & we have to visit rem vertices those 2, 3, 4 & min of any start v i can go to k v that belong to S.

it means starting vertex 1 I can go to 2, 3, 4 here

Goal: we findout the min shortest path from source vertex to visiting all vertices once & final reach to starting vertices

$g(4, \phi) = 8$

$g(3, \phi) = 6$

$g(2, \phi) = 5$

$g(1, \phi) = 0$

$$g(1,\{2,3,4\}) = \min_{k\in\{2,3,4\}} \begin{cases} c_{12}+g(2,\{3,4\}) \\ c_{13}+g(3,\{2,4\}) \\ c_{14}+g(4,\{2,3\}) \end{cases}$$

$$g(2,\{3,4\}) = \min\{c_{23}+g(3,\{4\}),\ c_{24}+g(4,\{3\})\}$$

$$g(3,\{4\}) = \min\{c_{34}+g(4,\phi)\} = 12+8 = 20$$

$$g(4,\{3\}) = \min\{c_{43}+g(3,\phi)\} = 9+6 = 15$$

$$\therefore g(2,\{3,4\}) = \min\{9+20,\ 10+25\} = 25$$

$$g(3,\{2,4\}) = \min\begin{cases} c_{32}+g(2,\{4\}), \\ c_{34}+g(4,\{2\}) \end{cases}$$

$$g(2,\{4\}) = \min\{c_{24}+g(4,\phi)\} = 10+8 = 18$$

$$g(4,\{2\}) = \min\{c_{42}+g(2,\phi)\} = 8+5 = 13$$

$$\therefore g(3,\{2,4\}) = \min\{13+18,\ 12+13\} = \{31,\ 25\} = 2$$

$$g(4,\{2,3\}) = \min\{c_{42}+g(2,\{3\}),\ c_{43}+g(3,\{2\}$$

$$g(2,\{3\}) = \min\{c_{23}+g(3,\phi)\} = 9+6 = 15$$

$$g(3,\{2\}) = \min\{c_{32}+g(2,\phi)\} = 13+5 = 18$$

$$\therefore g(4,\{2,3\}) = \min\{8+15,\ 9+18\} = 23$$

$$g(1,\{2,3,4\}) = \min\begin{Bmatrix} 10+35, \\ 15+25, \\ 20+23 \end{Bmatrix} = 35$$

The optimal tour for the graph has length = 35

The optimal tour is: 1-----→2----→ 4----→ 3--- >1.

Optimal Binary Search Tree

A dynamic programming solution to the problem of obtaining an optimal binary search tree can be viewed by constructing a tree as a result of sequence of decisions by holding the principle of optimality.

- A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes. The external nodes are null nodes.
- **The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keysin the right sub-tree are greater.**
- When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree. An optimal binary search tree is a BST, which has minimal expected cost of locating each node.
- Search time of an element in a **BST is O(n)**, whereas in a **Balanced-BST search time is O(log n)**.
- Again the search time can be improved in Optimal Cost Binary Search Tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.
- Here, the Optimal Binary Search Tree Algorithm is presented. First, we build a BST from a set of provided n number of distinct keys < k1, k2, k3, ... kn >.
- Here we assume, the probability of accessing a key K_i is p_i. Some dummy keys (d0, d1, d2, ... dn) are added as some searches may be performed for the values which are not present in the Key set K. We assume, for each dummy key di probability of access is qi.

OPTIMAL BINARY SEARCHTREE

Let us assume that the given set of identifiers is {a1, . . . , an} with a1 < a2 < . . . . < an. Let p (i) be the probability with which we search for ai. Let q (i) be the probability that the identifier x being searched for is such that ai < x < ai+1, 0 ≤i ≤n (assume a0 = -□and an+1 = +□). We have to arrange the identifiers in a binary search tree in a way that minimizes the expected total access time.

In a binary search tree, the number of comparisons needed to access an element at depth 'd' is d + 1, so if 'ai' is placed at depth 'di', then we want to minimize:

$$\begin{aligned}\text{Expected Cost of tree} &= \sum_{i=1}^{n} \text{cost}(k_i)p_i \\ &= \sum_{i=1}^{n} (\text{depth}(k_i) + 1)p_i \\ &= \sum_{i=1}^{n} \text{depth}(k_i)p_i + \sum_{i=1}^{n} p_i \\ &= \left(\sum_{i=1}^{n} \text{depth}(k_i)p_i\right) + 1\end{aligned}$$

Let P (i) be the probability with which we shall be searching for 'ai'. Let Q (i) be the probability of an un-successful search. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

The expected cost contribution for the internal node for 'ai'is:

$P(i)*level(a_i)$.

Unsuccessful search terminate with I = 0 (i.e at an external node). Hence the cost contribution for this node is:

Q (i) * level ((Ei) -1)

The expected cost of binary search tree is:

Given a fixed set of identifiers, we wish to create a binary search tree organization. We may expect different binary search trees for the same identifier set to have different performance characteristics.

The computation of each of these c(i, j)'s requires us to find the minimum of m quantities. Hence, each such c(i, j) can be computed in time O(m). The total time for all c(i, j)'s with j – i = m is therefore O(nm –m^2).

Example 1: The possible binary search trees for the identifier set (a1, a2, a3) = (do, if, stop) are as follows. Given the equal probabilities p (i) = Q (i) = 1/7 for all i, we have:

The C (i, J) can be computedas:

$$C(i, J) = \min_{i<k\leq J} \{C(i, k-1) + C(k, J) + P(K) + w(i, K-1) + w(K,J)\}$$

$$= \min_{i<k\leq J} \{C(i, K-1) + C(K, J)\} + w(i,J) \quad -- \quad (1)$$

Where $W(i, J) = P(J) + Q(J) + w(i,J-1)$ -- (2)

Initially $C(i, i) = 0$ and $w(i, i) = Q(i)$ for $0 \leq i \leq n$.

Equation (1) may be solved for C (0, n) by first computing all C (i, J) such that J - i = 1 Next, we can compute all C (i, J) such that J - i = 2, Then all C (i, J) with J - i = 3 and soon.

C (i, J) is the cost of the optimal binary search tree 'Tij' during computation we record the root R (i, J) of each tree 'Tij'. Then an optimal binary search tree may be constructed from these R (i, J). R (i, J) is the value of 'K' that minimizes equation(1).

We solve the problem by knowing W (i, i+1), C (i, i+1) and R (i, i+1), $0 \leq i < 4$; Knowing W (i, i+2), C (i, i+2) and R (i, i+2), $0 \leq i < 3$ and repeating until W (0, n), C (0, n) and R (0, n) areobtained.

The results are tabulated to recover the actualtree.

Example1:

Let n = 4, and (a1, a2, a3, a4) = (do, if, need, while) Let P (1: 4) = (3, 3, 1, 1) and Q (0: 4) = (2, 3, 1, 1,1)

Solution:
Table for recording W (i, j), C (i, j) and R (i,j):

Column Row	**0**	**1**	**2**	**3**	**4**

0	2, 0,0	3, 0,0	1, 0,0	1, 0,0,	1, 0,0
1	8, 8,1	7, 7,2	3, 3,3	3, 3,4	
2	12, 19,1	9, 12,2	5, 8,3		
3	14, 25,2	11, 19,2			
4	16, 32,2				

Thiscomputationiscarriedoutrow-wisefromrow0torow4.Initially,W(i,i)=Q (i) and C (i, i) = 0 and R (i, i) = 0, 0 $\leq$i <4.

Solving for C (0,n):

First, computing all C (i, j) such that j - i = 1; j = i + 1 and as 0 $\leq$i < 4; i = 0, 1, 2 and 3; i < k $\leq$ J. Start with i = 0; so j = 1; as i < k $\leq$ j, so the possible value for k =1

W (0, 1) = P (1) + Q (1) + W (0, 0) = 3 + 3 + 2 =8
C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} =8
R (0, 1) = 1 (value of 'K' that is minimum in the above equation). Next

with i = 1; so j = 2; as i < k $\leq$ j, so the possible value for k =2

W (1, 2) = P (2) + Q (2) + W (1, 1) = 3 + 1 + 3 =7
C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} =7
R (1, 2) =2

Next with i = 2; so j = 3; as i < k $\leq$ j, so the possible value for k =3

W (2, 3) = P (3) + Q (3) + W (2, 2) = 1 + 1 + 1 =3
C (2, 3) = W (2, 3) + min {C (2, 2) + C (3, 3)} = 3 + [(0 + 0)] =3
R (2, 3) =3

Next with i = 3; so j = 4; as i < k $\leq$ j, so the possible value for k =4 W (3,

4) = P (4) + Q (4) + W (3, 3) = 1 + 1 + 1 =3
C (3, 4) = W (3, 4) + min {[C (3, 3) + C (4, 4)]} = 3 + [(0 + 0)] =3
R (3, 4) =4

Second, Computing all C (i, j) such that j - i = 2; j = i + 2 and as 0 $\leq$i < 3; i = 0, 1, 2; i < k $\leq$ J. Start with i = 0; so j = 2; as i < k $\leq$ J, so the possible values for k = 1 and2.

W (0, 2) = P (2) + Q (2) + W (0, 1) = 3 + 1 + 8 =12
C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2,2))}
= 12 + min {(0 + 7, 8 + 0)} =19

R (0, 2) =1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and3.

W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 7 =9
C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 2) + C (3,3)]}
= W (1, 3) + min {(0 + 3), (7 + 0)} = 9 + 3 =12
R (1, 3) =2

Next, with i = 2; so j = 4; as i < k ≤ j, so the possible value for k = 3 and 4. W (2,

4) = P (4) + Q (4) + W (2, 3) = 1 + 1 + 3 =5
C (2, 4) = W (2, 4) + min {[C (2, 2) + C (3, 4)], [C (2, 3) + C (4,4)]
= 5 + min {(0 + 3), (3 + 0)} = 5 + 3 =8
R (2, 4) =3

Third, Computing all C (i, j) such that J - i = 3; j = i + 3 and as 0 ≤i < 2; i = 0,1;
i < k ≤ J. Start with i = 0; so j = 3; as i < k ≤ j, so the possible values for k = 1, 2 and3.

W (0, 3) = P (3) + Q (3) + W (0, 2) = 1 + 1 + 12 =14
C (0, 3) = W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2,3)],
[C (0, 2) + C (3,3)]}
= 14 + min {(0 + 12), (8 + 3), (19 + 0)} = 14 + 11 =25
R (0, 3) =2

Start with i = 1; so j = 4; as i < k ≤ j, so the possible values for k = 2, 3 and4. W (1, 4)

= P (4) + Q (4) + W (1, 3) = 1 + 1 + 9 =11
C (1, 4) = W (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 2) + C (3,4)],
[C (1, 3) + C (4,4)]}
= 11 + min {(0 + 8), (7 + 3), (12 + 0)} = 11 + 8 =19
R (1, 4) =2

Fourth, Computing all C (i, j) such that j - i = 4; j = i + 4 and as 0 ≤i < 1; i = 0; i < k ≤J.

Start with i = 0; so j = 4; as i < k ≤ j, so the possible values for k = 1, 2, 3 and4.

W (0, 4) = P (4) + Q (4) + W (0, 3) = 1 + 1 + 14 =16
C (0, 4) = W (0, 4) + min {[C (0, 0) + C (1, 4)], [C (0, 1) + C (2,4)],
[C (0, 2) + C (3, 4)], [C (0, 3) + C (4,4)]}
= 16 + min [0 + 19, 8 + 8, 19+3, 25+0] = 16 + 16 =32
R (0, 4) =2

From the table we see that C (0, 4) = 32 is the minimum cost of a binary search tree for (a1, a2, a3, a4). The root of the tree 'T04' is'a2'.

Hence the left sub tree is 'T01' and right sub tree is T24. The root of 'T01' is 'a1' and the root of 'T24' isa3.

The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'

The left and right sub trees for T24 are T22 and T34 respectively.

The root of T24 is'a3'.

The root of T22 is null

The root of T34 isa4.

Example2:

Consider four elements a_1, a_2, a_3 and a_4 with $Q_0 = 1/8$, $Q_1 = 3/16$, $Q_2 = Q_3 = Q_4 = 1/16$ and $p_1 = 1/4$, $p_2 = 1/8$, $p_3 = p_4 = 1/16$. Construct an optimal binary search tree. Solving for C (0,n):

First, computing all C (i, j) such that j - i = 1; j = i + 1 and as $0 \le i < 4$; i = 0, 1, 2 and 3; $i < k \le J$. Start with i = 0; so j = 1; as $i < k \le j$, so the possible value for k =1

W (0, 1) = P (1) + Q (1) + W (0, 0) = 4 + 3 + 2 =9
C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} = 9 + [(0 + 0)] =9
R (0, 1) = 1 (value of 'K' that is minimum in the above equation). Next

with i = 1; so j = 2; as $i < k \le j$, so the possible value for k =2

W (1, 2) = P (2) + Q (2) + W (1, 1) = 2 + 1 + 3 =6
C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} = 6 + [(0 + 0)] =6
R (1, 2) =2

Next with i = 2; so j = 3; as $i < k \le j$, so the possible value for k =3 W (2,

3) = P (3) + Q (3) + W (2, 2) = 1 + 1 + 1 =3
C (2, 3) = W (2, 3) + min {C (2, 2) + C (3, 3)} = 3 + [(0 + 0)] =3

R (2, 3) =3

Next with i = 3; so j = 4; as i < k ≤ j, so the possible value for k =4 W (3,

4) = P (4) + Q (4) + W (3, 3) = 1 + 1 + 1 =3
C (3, 4) = W (3, 4) + min {[C (3, 3) + C (4, 4)]} = 3 + [(0 + 0)] =3
R (3, 4) =4

Second, Computing all C (i, j) such that j - i = 2; j = i + 2 and as 0 ≤i < 3; i = 0, 1, 2; i < k ≤J

Start with i = 0; so j = 2; as i < k ≤ j, so the possible values for k = 1 and 2. W (0,

2) = P (2) + Q (2) + W (0, 1) = 2 + 1 + 9 =12
C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2,2))}
= 12 + min {(0 + 6, 9 + 0)} = 12 + 6 =18
R (0, 2) =1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and3.

W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 6 =8
C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 2) + C (3,3)]}
= W (1, 3) + min {(0 + 3), (6 + 0)} = 8 + 3 =11
R (1, 3) =2

Next, with i = 2; so j = 4; as i < k ≤ j, so the possible value for k = 3 and 4. W (2,

4) = P (4) + Q (4) + W (2, 3) = 1 + 1 + 3 =5
C (2, 4) = W (2, 4) + min {[C (2, 2) + C (3, 4)], [C (2, 3) + C (4,4)]
= 5 + min {(0 + 3), (3 + 0)} = 5 + 3 =8
R (2, 4) =3

Third, Computing all C (i, j) such that J - i = 3; j = i + 3 and as 0 ≤i < 2; i = 0,1;
i < k ≤ J. Start with i = 0; so j = 3; as i < k ≤ j, so the possible values for k = 1, 2 and3.

W (0, 3) = P (3) + Q (3) + W (0, 2) = 1 + 1 + 12 =14
C (0, 3) = W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2,3)],
[C (0, 2) + C (3,3)]}
= 14 + min {(0 + 11), (9 + 3), (18 + 0)} = 14 + 11 =25
R (0, 3) =1

Start with i = 1; so j = 4; as i < k ≤ j, so the possible values for k = 2, 3 and4. W (1, 4)

= P (4) + Q (4) + W (1, 3) = 1 + 1 + 8 =10
C (1, 4) = W (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 2) + C (3,4)],
[C (1, 3) + C (4,4)]}
= 10 + min {(0 + 8), (6 + 3), (11 + 0)} = 10 + 8 =18
R (1, 4) =2

Fourth, Computing all C (i, j) such that J - i = 4; j = i + 4 and as 0 ≤i < 1; i =0;

$i < k \leq J$. Start with i = 0; so j = 4; as $i < k \leq j$, so the possible values for k = 1, 2, 3 and4.

W (0, 4) = P (4) + Q (4) + W (0, 3) = 1 + 1 + 14 =16
C (0, 4) = W (0, 4) + min {[C (0, 0) + C (1, 4)], [C (0, 1) + C (2,4)],
[C (0, 2) + C (3, 4)], [C (0, 3) + C (4,4)]}
= 16 + min [0 + 18, 9 + 8, 18 + 3, 25 + 0] = 16 + 17 =33
R (0, 4) =2

Table for recording W (i, j), C (i, j) and R (i,j)

Column Row	**0**	**1**	**2**	**3**	**4**
0	2, 0,0	1, 0,0	1, 0,0	1, 0,0,	1, 0,0
1	9, 9,1	6, 6,2	3, 3,3	3, 3,4	
2	12, 18,1	8, 11,2	5, 8,3		
3	14, 25,2	11, 18,2			
4	16, 33,2				

From the table we see that C (0, 4) = 33 is the minimum cost of a binary search tree for (a1, a2, a3,a4)

The root of the tree 'T04' is'a2'.

From the table we see that C (0, 4) = 33 is the minimum cost of a binary search tree for (a1, a2, a3,a4)

The root of the tree 'T04' is'a2'.

Hence the left sub tree is 'T01' and right sub tree is T24. The root of 'T01' is 'a1' and the root of 'T24' isa3.

The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'

The left and right sub trees for T24 are T22 and T34 respectively.

The root of T24 is'a3'.

The root of T22 is null.

The root of T34 isa4.

Algorithm:

Algorithm OBST(P, q, n)
// Given n distinct identifier $a_1 \ldots a_n$ where
$a_1 < a_2, a_3 < \ldots a_n$
// probabilities $p[i], 1 \le i \le n$ & $q[i], 0 \le i \le n$
// $c[i,j]$ is the cost of optimal Binary Search tree
// $r[i,j]$ is root of k_{ij}
// $w[i,j]$ is wt of k_{ij}
{
for i := 0 to n-1 do
{ // initialize
$w[i,i] = q_i$
$c[i,i] = 0$
$r[i,j] = 0$

```
// optimal tree with one node
w[i,i+1] = q[i]+q[i+1]+p[i+1];
r[i,i+1]=i+1;
c[i,i+1] = q[i]+q[i+1]+p[i+1];
}
w[n,n] = q[n]
r[n,n]=0
c[n,n]=0
for m=2 to n-m do
{
  w[i,j] := w[i,j-1]+p[j]+q[j];
  k := find(c,r,i,j)
  c[i,j]=w[i,j]+c[i,k-1]+c[k,j];
}r[i,j]=k;
  write( c[0,n], w[0,n], r[0,n]);
}
Algorithm find(c,r,i,j)
{
min=∞;
for m=r[i,j-1] to r[i,j+1] do
  if (c[i,m-1]+c[m,j] < min) then
    { min=c[i,m-1]+c[m,j];
    } i=m
}} return i;
```

$w[i, i+1-1]$

q_i

$p[i+1]$

All pair shortest path ~~algorithm~~ Problem

→ Given Graph G(E,V) is weighted & connected graph

→ The objective of all pairs shortest path problem is to find shortest path between each & every pair of nodes present in given graph

→ By using Dynamic programming to find out

$A^k[i,j]$ where $k=0$ to n

$A^k[i,j]$ represent path b/w i to j of k.

Let $k=0$, we can take A^0 from given graph

i.e, $A^0 = w[i,j]$

if $k \geq 1$ compute the following formula

$$A^k[i,j] = \min\{A^{k-1}[i,j],\ A^{k-1}[i,k] + A^{k-1}[k,j]\}$$

eg:

1. Now convert the graph to matrix form

$$A^0 = \begin{bmatrix} 0 & 4 & 15 \\ 8 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

2. compute A^1 matrix

$A^1_{11} = 0$

$A^1_{12} = \min\{A^0[1,2],\ A^0[1,1] + A^0[1,2]\}$

$= \min\{4,\ 0+4\} = 4$

$A^1_{13} = \min\{A^0[1,3],\ A^0[1,1] + A^0[1,3]\}$

$= \min\{15,\ 0+15\} = 15$

$A^1_{21} = \min\{A^0[2,1],\ A^0[2,1] + A^0[1,1]\}$

$= \min\{8,\ 8+0\} = 8$

$A^1_{22} = \min\{A^0[2,2],\ A^0[2,1] + A^0[1,2]\}$

$= \min\{0,\ 8+4\} = 0$

1. Different between Dynamic and DAC

2. " " Dynamic and Greedy method

3. use the Floyd-warshall algorithm & find shortest path between all pairs of vertices of the following graph. write algorithm for All pair shortest path & its Time complexity?

4. write algorithm for matrix chain multiplication & solve optimal sequence for completion of matrix multiplication

A[1*30] B[30*40] C[40*10] D[10*5]

4. write an algorithm for Travelling salesman. Solve TSP using Dynamic programming with Example

$$\begin{bmatrix} \infty & 12 & 5 & 7 \\ 11 & \infty & 13 & 6 \\ 4 & 9 & \infty & 18 \\ 10 & 3 & 8 & \infty \end{bmatrix}$$

5. write an algorithm of longest common subsequence. Solve the LCS X = {A B C D B C D C D D} Y = {B C D C D} find the longest subsequence of common both

6. Solve the 0/1 knapsack problem in dynamic method for following m = 8, n = 4, Profit = {1, 2, 5, 6} weight = {2, 3, 4, 5}

or

n = 7, m = 15 P = {10, 5, 15, 7, 6, 18, 3}
w = {2, 3, 5, 7, 1, 4, 1}

$A^1_{23} = \min\{A^0[2,3], A^0[2,1] + A^0[1,1]\}$

$= \min\{3, 3+0\} = 3$

$A^1_{31} = \min\{A^0[3,1], A^0[3,1] + A^0[1,1]\}$

$= \min\{3, 3+0\} = 3$

$A^1_{32} = \min\{A^0[3,2], A^0[3,1] + A^0[1,2]\}$

$= \min\{\infty, 7\} = 7$

$A^1_{33} = 0$

$$\therefore A^1 = \begin{bmatrix} 0 & 4 & 15 \\ 8 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

3. $k=2$, Compute A^2

$A^2_{11} = \min\{A^1[1,1], A^1[1,2] + A^1[2,1]\} = \min\{0, 4+8\} = 0$

$A^2_{12} = \min\{A^1[1,2], A^1[1,2] + A^1[2,2]\} = \min\{4, 4+0\} = 4$

$A^2_{13} = \min\{A^1[1,3], A^1[1,2] + A^1[2,3]\} = \min\{15, 4+2\} = 6$

$A^2_{21} = \min\{A^{2-1}[2,1], A^1[2,2] + A^1[2,1]\} = \min\{8, 0+8\} = 8$

$A^2_{22} = 0$

$A^2_{23} = \min\{A^1[2,3], A^1[2,2] + A^2[2,3]\} = \min\{2, 0+2\} = 2$

$A^2_{31} = \min\{A^1[3,1], A^1[3,2] + A^1[2,1]\} = \min\{3, 7+8\} = 3$

$A^2_{32} = \min\{A^1[3,2], A^1[3,2] + A^1[2,2]\} = \min\{7, 7+0\} = 7$

$A^2_{33} = 0$

$$\therefore A^2 = \begin{bmatrix} 0 & 4 & 6 \\ 8 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

4. $k=3$, compute A^3 matrix

$A^3_{11} = 0$

$A^3_{12} = \min\{A^2[1,2], A^2[1,3] + A^2[3,2]\} = \min\{4, 6+7\} = 4$

$A^3_{13} = \min\{A^2[1,3], A^2[1,3] + A^2[3,3]\} = \min\{6, 6+0\} = 6$

$A^3_{21} = \min\{A^2[2,1], A^2[2,3] + A^2[3,1]\} = \min\{8, 2+3\} = 5$

$A^3_{22} = 0$

$A^3_{23} = \min\{A^2[2,3], A^2[2,3] + A^2[3,3]\} = \min\{2, 2+0\} = 2$

$A^3_{31} = \min\{A^2[3,1], A^2[3,3] + A^2[3,1]\} = \min\{3, 0+3\} = 3$

$A^3_{32} = \min\{A^2[3,2], A^2[3,3] + A^2[3,2]\} = \min\{7, 0+7\} = 7$

$A^3_{33} = 0$

$$A^3 = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

<u>Eg 2</u>: Find out all pair shortest path for following Graph

```
algorithm Allpair (cost, A, n)
{ for i:=1 to n do
    for j:=1 to n do
        A[i,j]=cost[i,j];
  for k:=1 to n do
      for i:=1 to n do
          for j:=1 to n do
              A[i,j]=min{A[i,j], A[i,k]+A[k,j]}
}
```

$O(n^3)$

UNIT – IV

BACKTRACKING : General method, Applications – n-queens problem, Sum of subsets problem, Graph coloring, Hamiltonian cycles.

GENENRAL METHOD

- Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion.
- The desired solution is expressed as an n-tuple ($x_1 . . . x_n$) where each $x_i \in S$, S being a finite set.
- The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P ($x_1 . . . x_n$).
- Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories:
    1. Explicit constraints and
    2. Implicit constraints.

Definition 1:

Explicit constraints are rules that restrict each x_i to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Example : xi>=0 or s={set of all non regular real number} xi=0 or xi=1 or s={0,1}

Definition 2:

Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus, implicit constraints describe the way in which the x_i's must relate to each other.

Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the

given problem instance. This search is facilitated by using a tree organization for the solution space.

Backtracking is the procedure where by, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

A backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

Terminology:

1. **Problem state** is each node in the depth first search tree.
2. **Solution states** are the problem states 'S' for which the path from the root node to 'S' defines a tuple in the solution space.
3. **Answer states** are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.
4. **State space** is the set of paths from root node to other nodes. State space tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.
5. **Live node** is a node that has been generated but whose children have not yet been generated.
6. **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
7. **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Algorithm:

```
Algorithm Backtracking (n)
// Describes the backtracking procedure .All solutions are generated in X[1:n]
{
   k=1;
   While (k ≠ 0) do
   {
     if (there remains all untried
     X[k]  ϵ T (X[1],[2],…..X[k-1]) and Bk (X[1],…..X[k])) is true ) then
    {
      if(X[1],……X[k] )is the path to the answer node)
     Then write(X[1:k]);
     k=k+1;              //consider the next step.
    }
  else k=k-1;                  //consider backtracking to the previous set.
 }
}
```

- All solutions are generated in X[1:n]
- T(X[1]…..X[k-1]) is all possible values of X[k] gives that X[1],.....X[k-1] have already been chosen.
- B_k(X[1]……X[k]) is a bounding function which determines the elements of X[k] which satisfies the implicit constraint.

APPLICATIONS

Applications includes,

- ✓ **N- Queens Problem**
- ✓ **Sum of Subsets Problem**
- ✓ **Graph Coloring**
- ✓ **Hamiltonian Cycle**

N- QUEEN PROBLEMS

One of the most common examples of the backtracking is to arrange N queens on an N x N chessboard such that no queen can strike down any other queen. A queen can attack horizontally, vertically, or diagonally. The solution to this problem is also attempted in a similar way. We first place the first queen anywhere arbitrarily and then place the next queen in any of the safe places. We continue this process until the number of unplaced queens becomes zero (a solution is found) or no safe place is left. If no safe place is left, then we change the position of the previously placed queen.

- Let us consider, N = 4. Then 4-Queens Problem is to place eight queens on an 4 x 4 chessboard so that **no two 'attack'**, that is, **no two of them are on the same row, column, or diagonal.**
- All solutions to the 4-queens problem can be represented as 4-tuples ($x_1 . . . x_4$), where x_i is the column of the ith row where the ith queen is placed.
- The **explicit constraints** using this formulation is $\mathbf{S_i = \{1, 2, 3, 4\}}$, $1 < i < 4$. Therefore the solution space consists of 4^4 4-tuples.
- The **implicit constraints** for this problem are that **no two x_i's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.**

- This realization reduces the size of the solution space from 4^4 tuples to 4! tuples. The promising function must check whether two queens are in the same column or diagonal.

4- Queens problem

condition / bounding function

1. No 2 queens in same row
2. No 2 queens in same column
3. No 2 queen's in same diagonal.

1. Initial 4x4 chessboard
2. Q1 is placed in 1x1 position. The possible position of Q1 is 1x2, 1x3, 1x4.

3. Q2 is placed in 2x3 position of board the possible position of Q2 is 2x4.

3. Q3 isn't placed in any position of board. So, to apply backtracking on Q2. Now Q2 is placed on 2x4. change Q2 to 4th location & placed Q3 in right location.
4. Q3 is placed on 3x2 position of chessboard.

5. Q4 is n't placed in any position. So we can apply backtracking on Q3, Q2, Q1. Final Q1 is placed on 1x2 position. Q2 is 2x4. Q3 is 3x1 position.

Complete state space tree

Portion of state space tree giving solution

8 Queens problem

(5)

1. Initially 8x8 chessboard
2. Q1 is placed in 1x1 position in the board. Remaining possible positions are 1x2, 1x3, 1x4, 1x5, 1x6, 1x7, 1x8
3. Q2 is placed in 2x3 position in the board Remaining possible positions are 2x4, 2x5, 2x6, 2x7, 2x8
8. Q3 is placed in 3x5 position in the board Remaining possible position are 3x6, 3x7, 3x8
4. Q4 is placed in 4x7 position. Remaining possible positions of Q4 is 4x8
5. Q5 is placed in 5x4 position. Remaining possible positions of Q8

6. Q6 is not placed in ~~5x8~~ any ~~possibl~~ position of chessboard.

Now, apply backtracking of Q5, Q4, Q3, Q2, Q1

∴ Q1 is placed in 1x5, Q2 is 2x1 position
Q3 is " in 3x4,
Q4 is " " 4x6
Q5 " " " 5x8, & Q6 is placed 6x2

(6)

	1	2	3	4	5	6	7	8
1					Q1			
2	Q2							
3				Q3				
4						Q4		
5								Q5
6		Q6						
7								
8								

7. Q7 is placed in 7x7 position of 8x8 chessboard

8. Q8 is placed in 8x3 position of 8x8 chessboard

	1	2	3	4	5	6	7	8
1					Q1			
2	Q2							
3				Q3				
4						Q4		
5								Q5
6		Q6						
7							Q7	
8			Q8					

∴ possible solution states ~~are~~ is

(5, 1, 4, 6, 8, 2, 7, 3)

The above picture shows an NxN chessboard and we have to place N queens on it. So, we will start by placing the first queen.

Now, the second step is to place the second queen in a safe position and then the third queen.

Now, you can see that there is no safe place where we can put the last queen. So, we will just change the position of the previous queen. And this is backtracking.

Also, there is no other position where we can place the third queen so we will go back one more step and change the position of the second queen.

And now we will place the third queen again in a safe position until we find a solution.

We will continue this process and finally, we will get the solution as shown below.

As now you have understood backtracking, let us now code the above problem of placing N queens on an NxN chessboard using the backtracking method.

State space tree of the four-queens problem

- Built the state space tree
  - Root represents an initial state
  - Bodes reflect the specific choices made for the components of a solution
  - Promising and non-promising nodes are present in the tree
- Explore the state space tree using depth-first search
- Prune non-promising nodes
  - DFS stops exploring subtree rooted at nodes leading to no solution and
  - backtracks to its parent node.

N-Queen Algorithm:

```
Algorithm N - Queens (k, n)
{
  For i ← 1 to n
      do if Place (k, i) then
  {
    x [k] ← i;
    if (k ==n) then
      write (x [1 ...n));
    else
    N - Queens (k + 1, n);
  }
}

    Place (k, i)
      {
        For j ← 1 to k - 1
         do if (x [j] = i)
          or (Abs x [j]) - i) = (Abs (j - k))
        then return false;
```

```
        return true;
} }
```

Complexity analysis

$$1 + n + n^2 + n^3 + \ldots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

SUM OF SUBSETS PROBLEM

Sum of Subsets Problem is finding a subset of a given set S = {s_1,s_2….s_n} of n positive integers whos**e** sum is equal to a given positive integer d or m.

All solutions are having k-tuples, $1 \leq k \leq n$..

Explicit constrains:

- xi **Є** {j | j is an integer and $1 \leq j \leq n$}

Implicit constrains:

- No two xi can be the same
- The sum of the corresponding w_i be d
- $x_i < x_{i+1}$, $1 \leq i < k$ to avoid generating multiple instances of the same subset. (for example (1, 2, 4) and (1, 4, 2) represent the same subset).

For example, for S = {1, 2, 5, 6, 8) and d = 9, there are two solutions: {1, 2, 6} and {1,8}. Of course, some instances of this problem may have no solutions. It is convenient to sort the set‘s elements in increasing order. So we will assume that $\mathbf{s_1 \leq s_2 \leq \ldots \leq s_n}$

- The state-space tree can be constructed as a binary tree as that in the following figure for the instances S = (3, 5, 6, 7) and d = 15.
- The root of the tree represents the starting point, with no decisions about the given elements made as yet.
- Its left and right children represent, respectively, inclusion and exclusion of s1 in a set being sought.
- Similarly, going to the left from a node of the first level corresponds to inclusion of s2, while going to the right corresponds to its exclusion, and soon.
- Thus, a path from the root to a node on the ith level of the tree indicates which of

the first i numbers have been included in the subsets represented by that node.

- We record the value of s' the sum of these numbers, in the node, If s is equal to d, we have a solution to the problem.
- We can either, report this result and stop or, if all the solutions need to he found, continue by backtracking to the node‘s parent.
- If s' is not equal to d, we can terminate the node as non-promising if either of the two inequalities holds:

$$s' + s_{i+1} > d \quad \text{(the sum s' is too large)}$$

$$s' + \sum_{j=i+1}^{n} s_j < d \quad \text{(the sum s' is too small)}$$

State-space tree of sum of subsets

Sum of subset problem

(7)

Suppose we have given 'n' distinct positive Integer numbers are called weight. Now we find combination of number whose sum is 'm' is called sum of subset

* The solution vector is either 1 or 0 depending on whether the weight w_i include or not.

$x_i = 0$ it weight is not include solution

$x_i = 1$ it weight is include solution.

* For node at level 'i' be the left child corresponding to $x_i = 1$ & the right child corresponding to $x_i = 0$

* Bounding function we use $B_k(x_1, x_2 \ldots x_k)$ to kill node $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq M$ & $\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq M$ to generate node

State space tree can be ~~given~~ drawn using the following

Eg: Find possible no. of sum of subset using sum of subset algorithm

$m=9,\ n=3\ (w_1, w_2, w_3) = (2, 4, 5)$

Kill node because satisfy Bounding fun

$\sum w_i x_i + w_k \geq M$

Solution

Subset $= \{4, 5\}$

Solution vector $= \{0, 1, 1\}$

$\therefore \sum w_i x_i \leq m$

$2*0 + 4*1 + 5*1 \leq 9$

$9 \leq 9$

Eg2: m = 81, w[1:4] = {7, 11, 13, 24}

Solution vector = {1, 1, 1, 0} & {1, 0, 0, 1}

Sum of subset = {7, 11, 13} or {7, 24}

$\sum_{i=1}^{n} w_i x_i \le m$

$7\times1+11\times1+13\times1+24\times0 \le 31$ ✓ $31 \le 31$

$7\times1+11\times0+13\times0+24\times1 \le 31$ $31 \le 31$

9

Eg 3: Construct state space tree for following
m = 30, n = 6, (w1 : w6) = (5, 10, 12, 13, 15, 18)

Solution vector = {1, 1, 0, 0, 1, 0}
or
{1, 0, 1, 1, 0, 0}

Subset = {5, 10, 15} or {5, 12, 13}

10

Hamiltonian cycle

Eg2:

Eg3:

Consider example, for S = {1, 2, 3, 4) and d = 6, there are two solutions: {1, 2, 3} and {2,4}. The solution is presented as fixed state and the solution state is {1,1,1,0} and {0,1,0,1}. The state space tree is given below

Sum of Subset Algorithm

```
Algorithm sumofsubset(s,k,r)
// find all subsets of w[1:n] that sum to m.
{
// Generate left child node. Note: s + w[k] ≤ m, since B_{k-1} is true
x[k]=1;
if (s+w[k]=m) then write(x[1:k]); // Subset found
else if (s+w[k]+w[k+1] ≤ m)
then sumofsubset(s+w[k], k+1,r- w[k]);
// Generate right child and evaluate B_k
if ((s + r – w[k] ≥ m ) and (s + w[k+1] ≤ m)) then          // exceeds m
{
x[k]=0;
sumofsubset(s, k+1, r-w[k]);
}}
```

Complexity

Time complexity = θ (2^n)

Space complexity = θ (1)

What is Graph Coloring Problem?

Let G be a graph, we have to color all vertices with the 'M' number of given colors, in such a way that no two adjacent vertices should have the same color.

- It is possible to color all the vertices with the given colors then we have to output the colored result, otherwise output 'no solution possible'.

The least possible value of 'm' required to color the graph successfully is known as the ***chromatic number of the given graph***. For example, the following graph can be colored minimum 3 colors.

Working principle of graph coloring using backtracking

1. In backtracking approach, we color a single vertex and then move to its adjacent (connected) vertex to color it with different color.
2. After coloring, we again move to another adjacent vertex that is uncolored and repeat the process until all vertices of the given graph are colored.
3. In case, we find a vertex that has all adjacent vertices colored and no color is left to make it color different, we backtrack and change the color of the last colored vertices and again proceed further.
4. By backtracking, we come back to the same vertex from where we started and all colors were tried on it, then it means the given number of colors (i.e. 'm') is insufficient to color the given graph and we require more colors (i.e. a bigger chromatic number).

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero. The colors are represented by the integers 1, 2, . . . , m and the solutions are given by the n-tuple (x_1, x_2, . . ., x_n), where x_i is the color of node i.

A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

Algorithm mcoloring (k)

```
// This algorithm was formed using the recursive backtracking schema.
//The graph is represented by its Boolean adjacency matrix G [1: n, 1: n].
//All assignments of 1,2, ...m to the vertices of the graph such that adjacent vertices are
//assigned. k is the index of the next vertex to color.
{
repeat
{ // Generate all legal assignments for x[k].
 NextValue (k); // Assign to x [k] a legal color.
 If (x [k] = 0) then return; // No new color possible
 If (k = n) then // at most m colors have been used to color the n vertices.
    write (x [1: n]);
       else mcoloring (k+1);
  } until (false);
}
```

Algorithm NextValue (k)

```
// x[1],..... x[k-1]have been assigned integer values in the range[1,m]such that
// adjacent vertices have distinct integers. A value for x [k] is determined in the range
// [0, m].x[k] is assigned the next highest numbered color while maintaining distinctness
// from the adjacent vertices of vertex k. If no such color exists, then x [k] is 0.
{
     repeat
      {
```

```
        x [k]: = (x [k] +1) mod(m+1)          // Next highest color.
        If (x [k] = 0) then return;      // All colors have been used
        for j := 1 to n do
        { // check if this color is distinct from adjacent colors
        if ((G [k, j] ≠ 0) and (x [k] = x[j]))
        // If (k, j) is and edge and if adj. vertices have the same color.
        then break;
        }
        if (j = n+1) then return;     // New colorfound
      }until(false);                  // Otherwise try to find anothercolor.
    }
```

Example:

Color the graph given below with minimum number of colors by backtracking using state space tree

Eg: 2
m=3, & n=3
State space tree
Eg: m=4, n=4
m-coloring=?
m=5
State space tree for 3 vertices of graph with 3 colors. m-coloring optimization is 6.
Note: How many color required color for all vertices of Graph G is d+1
degree of G.

HAMILTONIAN CYCLES

A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once.

Example of Hamiltonian Graph

Here,

- This graph contains a closed walk ABCDEFA.
- It visits every vertex of the graph exactly once except starting vertex.
- The edges are not repeated during the walk.
- Therefore, it is a Hamiltonian graph.

Hamiltonian Path

- If there exists a walk in the connected graph that visits every vertex of the graph exactly once without repeating the edges, then such a walk is called as a Hamiltonian path.

Hamiltonian Cycle

- .If there exists a walk in the connected graph that visits every vertex of the graph exactly once (except starting vertex) without repeating the edges and returns to the starting vertex, then such a walk is called as a Hamiltonian cycle.

Examples of Hamiltonian path and Hamiltonian cycle are as follows

Hamiltonian Path Examples
A
B
C
D
E
Hamiltonian Path = ABCDE
E
A
B
D
C
Hamiltonian Path = EABCD
F
A
B
D
C
E
Hamiltonian Path Does Not Exist
Hamiltonian Circuit Examples
A
B
D
C
E
Hamiltonian Circuit = ABCEDA
E
A
B
D
C
Hamiltonian Circuit Does Not Exist
F
A
B
D
C
E
Hamiltonian Circuit Does Not Exist

Important Notes

- Any Hamiltonian cycle can be converted to a Hamiltonian path by removing one of its edges.
- Every graph that contains a Hamiltonian cycle also contains a Hamiltonian path but vice versa is not true.
- There may exist more than one Hamiltonian paths and Hamiltonian cycles in a graph.

Given a graph G = (V, E) we have to find the Hamiltonian Cycle using Backtracking approach.

- We start our search from any arbitrary vertex say 'a.' This vertex 'a' becomes the root of our implicit tree.
- The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed.
- The next adjacent vertex is selected by alphabetical order.
- If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that dead end is reached. In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed.
- The search using backtracking is successful if a Hamiltonian Cycle is obtained.

Example: Consider a graph G = (V, E) shown in figure below. We have to find a Hamiltonian cycle using Backtracking method.

Solution: Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.

Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b,c,d).

Next, we select 'c' adjacent to 'b.'

Next, we select 'd' adjacent to 'c.'

Next, we select 'e' adjacent to 'd.'

Next, we select vertex 'f' adjacent to 'e.' The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.

From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.

Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once. (a - b - c - e - f -d - a).

Again Backtrack

Here we have generated one Hamiltonian circuit, but another Hamiltonian circuit can also be obtained by considering another vertex.

The backtracking solution vector (x_1, ... x_n) is defined so that x_i represents the ith visited vertex of the proposed cycle. If k = 1, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, ..., x_{k-1}$ and v is connected by an edge to k_{x-1}. The vertex x_n can only be one remaining vertex and it must be connected to both x_{n-1} and x_1.

Using NextValue algorithm we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix G[1: n, 1: n], then setting x[2: n] to zero and x[1] to 1, and then executing Hamiltonian(2).

The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

Algorithm NextValue (k)

```
// x [1: k-1] is a path of k – 1 distinct vertices . If x[k] = 0, then no vertex has as yet been
//assignedtox[k].Afterexecution,x[k]isassignedtothenexthighestnumberedvertex
//which does not already appear in x[1:k–1] and is connected by an edge to x[k–1].
// Otherwise x [k] = 0. If k = n, then in addition x [k] is connected to x [1].
{
repeat
x { [k] := (x [k] +1) mod(n+1); // Next vertex.
If(x[k] = 0) then return;
If (G [x [k – 1], x [k]] ≠ 0) then
{ // Is there an edge?
for j := 1 to k – 1 do
if (x [j] = x [k]) then break;
// check for distinctness.
If (j = k)then // If true, then the vertex is distinct.
If ((k < n) or ((k = n) and G [x [n], x [1]] ≠ 0))
then return;
}
} until (false);
}
```

Algorithm Hamiltonian (k)

```
// This algorithm uses the recursive formulation of backtracking to find all the
// Hamiltonian cycles of a graph. The graph is stored as an adjacency
// matrix G [1: n, 1: n]. All cycles begin at node 1.
{
```

```
repeat
{
// Generate values for x[k].
NextValue (k);      //Assign a legal Next value to x [k].
if (x [k] = 0) then return;
if (k = n) then
    write (x [1: n]);
else
   Hamiltonian (k + 1)
} until (false)
```

0/1 knapsack using Backtracking

$n=5, m=12, (P1 \ldots P5) = (10, 15, 6, 8, 4)$

$(w1 \ldots w5) = (4, 6, 3, 4, 2)$

$P/w = \left(\frac{10}{4}, \frac{15}{6}, \frac{6}{3}, \frac{8}{4}, \frac{4}{2}\right) = (2.5, 2.5, 2, 2, 2)$

$(P_1, P_2, P_3, P_4) = (10, 15, 6, 8, 4)$

$w\ (4, 6, 3, 4, 2)$

P,w
0,0
x1=1
x1=0
10,4
0,0
x2=1
x2=0
x2=1
x2=0
25,10
10,4
15,6
0,0
x3=1
x3=0
x3=1
x3=0
x3=1
x3=0
x3=1
x3=0
31,13
13>12
kill the node
25,10
16,7
20,4
21,9
15,6
6,3
0,0
x4=1
x4=0
x4=1
x4=0
x4=1
x4=

UNIT – V

BRANCH AND BOUND PROBLEMS: General method, Applications – travelling sales person problem, 0/1 Knapsack problem

P, NP, NP – HARD and NP – COMPLETE: Non deterministic algorithms and Deterministic algorithms, NP-Hard and NP-complete classes

---

BRANCH AND BOUND PROBLEMS: General method

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating sub trees that do not contain an answer node. However branch and Bound differs from backtracking in **two important manners**:

1. It has a branching function, which can be a depth first search, breadth firstsearch or based on bounding function.
2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node.

Branch and Bound is the generalization of both graph search strategies, BFS and D- search.

- A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).
- A D search like state space search is called as LIFO (Last in first out) search asthe list of live nodes in a last in first out (or stack).

Definition 1: Live node is a node that has been generated but whose children have notyet been generated.

Definition 2: E-node is a live node whose children are currently being explored. Inother words, an E-node is a node currently being expanded.

Definition 3: Dead node is a generated node that is not to be expanded or explored anyfurther. All children of a dead node have already been expanded.

Definition 4: Branch-an-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Definition 5: The adjective "heuristic", means" related to improving problem solving performance". As a noun it is also used in regard to "any method or trick used to improve the efficiency of a problem solving problem". But imperfect methods are not necessarily heuristic or vice versa. "A heuristic (heuristic rule, heuristic method) is a rule of thumb, strategy, trick simplification or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions, they do not guarantee any solution at all. A useful heuristic offers solutions which are good enough most of the time.

Least Cost (LC) search:

In both LIFO and FIFO Branch and Bound the selection rule for the next E-node in rigid and blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can be speeded by using an "intelligent" ranking function c'(.) for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:

$$c'(x) = f(h(x)) + g'(x)$$

where, c'(x) is the cost of x, h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function. g'(x) is an estimate of additional effort needed to reach an answer node from x.

A search strategy that uses a cost function c'(x) = f(h(x) + g'(x) to select the next E-node would always choose for its next E-node a live node with least c'(.) is called a LC–search (Least Cost search).

BFS and D-search are special cases of LC-search. If g'(x) = 0 and f(h(x)) = level of node x, then an LC search generates nodes by levels. This is eventually the same as a BFS. If f(h(x)) = 0 and g'(x) > g'(y) whenever y is a child of x, then the search is essentially a D-search.

An LC-search coupled with bounding functions is called an LC-branch and bound search. We associate a cost c(x) with each node x in the state space tree. It is not possible to easily compute the function c(x). So we compute an estimate c'(x) of c(x).

Control Abstraction for LC-Search:

Let t be a state space tree and c() a cost function for the nodes in t. If x is a node in t, then c(x) is the minimum cost of any answer node in the sub tree with root x. Thus, c(t) is the cost of a minimum-cost answer node in t.

A heuristic c'(.) is used to estimate c(). This heuristic should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then c(x) = c'(x). LC-search uses c' to find an answer node.

The algorithm uses two functions Least() and Add() to delete and add a live nodefrom or to the list of live nodes, respectively.

- Least() finds a live node with least c(). This node is deleted from the list of livenodes and returned.
- Add(x) adds the new live node x to the list of live nodes. The list of live nodesbe implemented as a min-heap.

Algorithm LC Search outputs the path from the answer node it finds to the root node t.This is easy to do if with each node x that becomes live, we associate a field parent which gives the parent of node x. When the answer node g is found, the path from g tot can be determined by following a sequence of parent values starting from the currentE-node (which is the parent of g) and ending at node t.

```
Listnode = record
{
        Listnode * next, *parent; float cost;
}
Algorithm LCSearch(t)
{ //Search t for an answer node

        if *t is an answer node then
        output *t and return;
        E:=t; //E-node.
        initialize the list of live nodes to be empty;
        repeat
        {
                for each child x of E do
                {
                        if x is an answer node then

                        output the path from x to t and return;
                        Add (x); //x is a new live node.
                        (x  parent):=E; // pointer for path to root
                }
                if there are no more live nodes then
                {
                        write (“No answer node”);
                        return;
                }
                E := Least();
        } until (false);
}
```

Branch-and-bound algorithms are used for optimization problems where, we deal directly only with minimization problems. **A maximization problem is easily converted to a minimization problem by changing the sign of the objective function.**

- To formulate the search for an optimal solution for a least-cost answer node in astate space tree, it is necessary to define the cost function c(.), such that c(x) is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for c(.).
  - For nodes representing feasible solutions, c(x) is the value of the objective function for that feasible solution.
  - For nodes representing infeasible solutions, $c(x) = \infty$.
  - For nodes representing partial solutions, c(x) is the cost of the minimum-cost node in the sub tree with root x.

Since, c(x) is generally hard to compute, the branch-and-bound algorithm will use an estimate c(x) such that $c(x) \leq c(x)$ for all x.

Consider job sequencing with deadlines problem for explaining the concept of how FIFO and LC branch and bound works.

Job sequencing with deadlines:

We are given n jobs and one processor. Each job i is associated with it a three tuple (p_i,d_i, t_i). Job i requires t_i units of processing time. If its processing is not completed by thedeadline d_i, then a penalty p_i is incurred. The objective is to select a subset J of the n jobs such that all jobs in J can be completed by their deadlines. Hence, a penalty can beincurred only on those jobs not in J. The subset J should be such that the penalty incurred is minimum among all possible subsets J. Such a J is optimal.

Example:

Let's understand through an example. Jobs = {j1, j2, j3, j4}
P = {10, 5, 8, 3}
d = {1, 2, 1, 2}

The above are jobs, problems and problems given. We can write the solutions in two ways which are given below:

Suppose we want to perform the jobs j1 and j2 then the solution can be represented in two ways:
The first way of representing the solutions is the subsets of jobs.
S1 = {j1, j4}
The second way of representing the solution is that first job is done, second and third jobs are not done, and fourth job is done.
S2 = {1, 0, 0, 1}
The solution s1 is the variable-size solution while the solution s2 is the fixed-size solution.

First, we will see the subset method where we will see the variable size.
First method:

In this case, we first consider the first job, then second job, then third job and finally we consider the last job.
As we can observe in the above figure that the breadth first search is performed but not the depth first search. Here we move breadth wise for exploring the solutions. In backtracking, we go depth-wise whereas in branch and bound, we go breadth wise.
Now one level is completed. Once I take first job, then we can consider either j2, j3 or j4. If we follow the route then it says that we are doing jobs j1 and j4 so we will not consider jobs j2 and j3.

Now we will consider the node 3. In this case, we are doing job j2 so we can consider either job j3 or j4. Here, we have discarded the job j1.

Now we will expand the node 4. Since here we are doing job j3 so we will consider only job j4.

Now we will expand node 6, and here we will consider the jobs j3 and j4.

Now we will expand node 7 and here we will consider job j4.

Now we will expand node 9, and here we will consider job j4.

The last node, i.e., node 12 which is left to be expanded. Here, we consider job j4.
The above is the state space tree for the solution s1 = {j1, j4}

Second method:

We will see another way to solve the problem to achieve the solution s1.
First, we consider the node 1 shown as below:
Now, we will expand the node 1. After expansion, the state space tree would be appeared as:
On each expansion, the node will be pushed into the stack shown as below:

Now the expansion would be based on the node that appears on the top of the stack. Since the node 5 appears on the top of the stack, so we will expand the node 5. We will pop out the node 5 from the stack. Since the node 5 is in the last job, i.e., j4 so there is no further scope of expansion.

The next node that appears on the top of the stack is node 4. Pop out the node 4 and expand. On expansion, job j4 will be considered and node 6 will be added into the stack shown as below:

The next node is 6 which is to be expanded. Pop out the node 6 and expand. Since the node 6 is in the last job, i.e., j4 so there is no further scope of expansion.

The next node to be expanded is node 3. Since the node 3 works on the job j2 so node 3 will be expanded to two nodes, i.e., 7 and 8 working on jobs 3 and 4 respectively. The nodes 7 and 8 will be pushed into the stack shown as below:

The next node that appears on the top of the stack is node 8. Pop out the node 8 and expand. Since the node 8 works on the job j4 so there is no further scope for the expansion.

The next node that appears on the top of the stack is node 7. Pop out the node 7 and expand. Since the node 7 works on the job j3 so node 7 will be further expanded to node 9 that works on the job j4 as shown as below and the node 9 will be pushed into the stack.

The next node that appears on the top of the stack is node 9. Since the node 9 works on the job 4 so

there is no further scope for the expansion.

The next node that appears on the top of the stack is node 2. Since the node 2 works on the job j1 so it means that the node 2 can be further expanded. It can be expanded upto three nodes named as 10, 11, 12 working on jobs j2, j3, and j4 respectively. There newly nodes will be pushed into the stack shown as below:

TRAVELLING SALES PERSON PROBLEM

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. This can be solved by branch and bound technique using efficient bounding function.

The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.

We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

Procedure for solving traveling sale person problem:

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:
   a) Row reduction: Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the sameprocedure for all rows.
   b) Find the sum of elements, which were subtracted from rows.
   c) Apply column reductions for the matrix obtained after row reduction.

Column reduction: Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.

   d) Find the sum of elements, which were subtracted from columns.
   e) Obtain the cumulative sum of row wise reduction and column wise reduction.

Cumulative reduced sum = Row wise reduction sum + column wise reductionsum

1. Associate the cumulative reduced sum to the starting state as lower bound and

   ∞ as upper bound.

2. Calculate the reduced cost matrix for every node R. Let A is the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge <i, j> in the tour. If S is not a leaf node, then the reduced costmatrix for S may be obtained as follows:
   a) Change all entries in row i and column j of A to ∞.
   b) Set A (j, 1) to ∞.
   c) Reduce all rows and columns in the resulting matrix except for rows and column containing only ∞. Let r is the total amount subtracted to reducethe matrix.
   d) Find c(S) = c(R) + A(i,j) + r, where r is the total amount subtracted to reduce the matrix, c(R) indicates the lower bound of the ith node in (i, j)path and c(S) is called the cost function.

3. Repeat step 2 until all nodes are visited.

Example:

Find the LC branch and bound solution for the traveling sale person problem whosecost matrix is as follows:

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & 20 & 30 & 10 & 11 \\ 2 & 15 & \infty & 16 & 4 & 2 \\ 3 & 3 & 5 & \infty & 2 & 4 \\ 4 & 19 & 6 & 18 & \infty & 3 \\ 5 & 16 & 4 & 7 & 16 & \infty \end{array}$$

Step : 1 Find the reduced cost matrixApply row reduction method.

- Deduct 10 (which is the minimum) from all values in the 1st row.
- Deduct 2 (which is the minimum) from all values in the 2nd row.
- Deduct 2 (which is the minimum) from all values in the 3rd row.
- Deduct 3 (which is the minimum) from all values in the 4th row.
- Deduct 4 (which is the minimum) from all values in the 5th row.

The resultant row wise reduced cost matrix is

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & 20 & 30 & 10 & 11 & 10 \\ 2 & 15 & \infty & 16 & 4 & 2 & 2 \\ 3 & 3 & 5 & \infty & 2 & 4 & 2 \\ 4 & 19 & 6 & 18 & \infty & 3 & 3 \\ 5 & 16 & 4 & 7 & 16 & \infty & 4 \\ \hline & & & & & & 21 \end{array} \Longrightarrow \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & 10 & 20 & 0 & 1 \\ 2 & 13 & \infty & 14 & 2 & 0 \\ 3 & 1 & 3 & \infty & 0 & \infty \\ 4 & 16 & 3 & 15 & \infty & 0 \\ 5 & 12 & 0 & 3 & 12 & \infty \end{array}$$

Row wise reduction sum = 10 + 2 + 2 + 3 + 4 = 21

Now apply column wise reduction for the above matrix.

- Deduct 1 (which is the minimum) from all values in the 1st column.
- Deduct 3 (which is the minimum) from all values in the 3rd column.

Colum wise reduction sum = 1 + 0 + 3 + 0 + 0 = 4

Cumulative reduced cost = row wise reduction + column wise reduction sum

= 21 + 4 = 25

Colum wise reduction sum = 1 + 0 + 3 + 0 + 0 = 4

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & 10 & 20 & 0 & 1 \\ 2 & 13 & \infty & 14 & 2 & 0 \\ 3 & 1 & 3 & \infty & 0 & \infty \\ 4 & 16 & 3 & 15 & \infty & 0 \\ 5 & 12 & 0 & 3 & 12 & \infty \\ \hline & 1 & 0 & 3 & 0 & 0 \end{array} \; 4 \quad \Longrightarrow \quad \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & 10 & 17 & 0 & 1 \\ 2 & 12 & \infty & 11 & 2 & 0 \\ 3 & 0 & 3 & \infty & 0 & \infty \\ 4 & 15 & 3 & 12 & \infty & 0 \\ 5 & 11 & 0 & 0 & 12 & \infty \end{array}$$

Cumulative reduced cost = row wise reduction + column wise reduction sum

= 21 + 4 = 25

So the resultant cost matrix is =

$$1 \quad \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & 10 & 17 & 0 & 1 \\ 2 & 12 & \infty & 11 & 2 & 0 \\ 3 & 0 & 3 & \infty & 0 & \infty \\ 4 & 15 & 3 & 12 & \infty & 0 \\ 5 & 11 & 0 & 0 & 12 & \infty \end{array}$$

Reduced Cost = 25

The value 25 is the cost of the root node, ie., node 1, because this is the initially reducedcost matrix. The lower bound for node is 25 and upper bound is ∞.

Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1, 3), (1, 4) and (1,5). The tree organization up to this point is as follows:

Step 2:

Consider the path (1,2). Change all entries of row 1 and column 2 of A to ∞ and also set A(2, 1) to ∞. Apply row and column reduction for the rows and columns. Then theresultant matrix is shown in no. 2. The cost C(s) for C(1,2) is 35

2

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & \infty & \infty & \infty & \infty & 0 \\ 2 & \infty & \infty & 11 & 2 & 0 & 0 \\ 3 & 0 & \infty & \infty & 0 & 2 & 0 \\ 4 & 15 & \infty & 12 & \infty & 0 & 0 \\ 5 & 11 & \infty & 0 & 12 & \infty & 0 \\ \hline & 0 & 0 & 0 & 0 & 0 & \end{array}$$

$C(1,2) + r + \hat{r}$ = 10+25+0
= 35

Consider the path (1, 3). Change all entries of row 1 and column 3 of A to ∞ and also set A(3, 1) to ∞. Apply row and column reduction for rows and columns and the resultant matrix is shown as no. 3. The cost C(s) for C(1,3) is 53

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & \infty & \infty & \infty & \infty & 0 \\ 2 & 12 & \infty & \infty & 2 & 0 & 0 \\ 3 & \infty & 3 & \infty & 0 & 2 & 0 \\ 4 & 15 & 3 & \infty & \infty & 0 & 0 \\ 5 & 11 & 0 & \infty & 12 & \infty & 0 \\ \hline & 11 & 0 & 0 & 0 & 0 & 11 \end{array} \longrightarrow$$

3

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & 1 & \infty & \infty & 2 & 0 \\ 3 & \infty & 3 & \infty & 0 & 2 \\ 4 & 4 & 3 & \infty & \infty & 0 \\ 5 & 0 & 0 & \infty & 12 & \infty \end{array}$$

$C(1,3) + r + \hat{r}$ = 17+25+11
= 53

Consider the path (1, 4). Change all entries of row 1 and column 4 of A to ∞ and also set A(4, 1) to ∞. Apply row and column reduction for rows and columns and the resultant matrix is shown as no. 3. The cost C(s) for C(1,4) is 25.

4

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	12	∞	11	∞	0	0
3	0	3	∞	∞	2	0
4	∞	3	12	∞	0	0
5	11	0	0	∞	∞	0
	0	0	0	0	0	

$C(1,4) + r + \hat{r} = 0+25+0$
$= 25$

Consider the path (1, 5). Change all entries of row 1 and column 5 of A to ∞ and also set A(5, 1) to ∞. Apply row and column reduction for rows and columns and the resultant matrix is shown as no. 3. The cost C(s) for C(1,5) is 31.

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	12	∞	11	2	∞	2
3	0	3	∞	∞	∞	0
4	15	3	12	∞	∞	3
5	∞	0	0	∞	∞	0
	0	0	0	0	0	5

→

5

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	10	∞	9	0	∞
3	0	3	∞	0	∞
4	12	0	9	∞	∞
5	∞	0	0	12	∞

$C(1,5) + r + \hat{r} = 1+25+5$
$= 31$

The tree organization up to this point is as follows:

The cost of the paths between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25 and (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix. So consider the matrix which is numbered as 4.

4

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

Consider the path (4,2):

Change all entries of row 4 and column 2 of A to ∞ and also set A(2, 1) to ∞. Apply row and column reduction for the rows and columns. Then the resultant matrix is,

6

	1	2	3	4	5	
1	∞	∞	∞	∞	∞	0
2	∞	∞	11	∞	0	0 =31
3	0	∞	∞	∞	2	0 5 5
4	∞	∞	∞	∞	∞	0
5	11	∞	0	∞	∞	0
	0	0	0	0	0	

= ∞

Therefore the cost of A(4,2) is 28

$$C(4,2) + c(4) + \hat{r} = 3+25+0$$
$$= 28$$

Consider the path (4,3):

Change all entries of row 4 and column 3 of A to ∞ and also set A(3, 1) to ∞. Applyrow and column reduction for the rows and columns. Then the resultant matrix is,

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & \infty & \infty & \infty & \infty & 0 \\ 2 & 12 & \infty & \infty & \infty & 0 & 0 \\ 3 & \infty & 3 & \infty & \infty & 2 & 2 \\ 4 & \infty & \infty & \infty & \infty & \infty & 0 \\ 5 & 11 & \infty & \infty & \infty & \infty & 11 \\ \hline & 0 & 0 & 0 & 0 & 0 & 13 \end{array} \xrightarrow{7} \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & 1 & \infty & \infty & \infty & 0 \\ 3 & \infty & 1 & \infty & \infty & 0 \\ 4 & \infty & \infty & \infty & \infty & \infty \\ 5 & 0 & 0 & \infty & \infty & \infty \end{array}$$

Therefore the cost of A(4,3) is 50

$$C(4,3) + C(4) + \hat{r} = 12+25+13$$
$$= 50$$

Consider the path (4,5):

Change all entries of row 4 and column 5 of A to ∞ and also set A(5, 1) to ∞. Applyrow and column reduction for the rows and columns. Then the resultant matrix is,

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & \infty & \infty & \infty & \infty & 0 \\ 2 & 12 & \infty & 11 & \infty & \infty & 11 \\ 3 & 0 & 3 & \infty & \infty & \infty & 0 \\ 4 & \infty & \infty & \infty & \infty & \infty & 0 \\ 5 & \infty & 0 & \infty & \infty & \infty & 0 \\ \hline & 0 & 0 & 0 & 0 & 0 & 11 \end{array} \xrightarrow{8} \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & 1 & \infty & \infty & \infty & 0 \\ 3 & \infty & 1 & \infty & \infty & 0 \\ 4 & \infty & \infty & \infty & \infty & \infty \\ 5 & 0 & 0 & \infty & \infty & \infty \end{array}$$

Therefore, the cost of A(4,5) 36

$$C(4,5) + C(4) + \hat{r} = 0+25+11$$
$$= 36$$

The cost of the paths between (4, 2) = 28, (4, 3) = 50 and (4, 5) = 36. The cost of the path between (4, 2) is minimum. Hence the matrix obtained for path (4, 2) is consideredas reduced cost matrix.

So consider the matrix no.4 as base matrix for further processing.

$$
\begin{array}{c|ccccc}
6 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & \infty & \infty & \infty & \infty & \infty \\
2 & \infty & \infty & 11 & \infty & 0 \\
3 & 0 & \infty & \infty & \infty & 2 \\
4 & \infty & \infty & \infty & \infty & \infty \\
5 & 11 & \infty & 0 & \infty & \infty
\end{array}
$$

The new possible paths are (2, 3) and (2, 5).

Consider the path (2, 3):

Change all entries of row 2 and column 3 of A to ∞ and also set A(3, 1) to ∞. Applyrow

and column reduction for the rows and columns.

$$\begin{array}{c|ccccc|c} & 1 & 2 & 3 & 4 & 5 & \\ \hline 1 & \infty & \infty & \infty & \infty & \infty & 0 \\ 2 & \infty & \infty & \infty & \infty & \infty & 0 \\ 3 & \infty & \infty & \infty & \infty & 2 & 2 \\ 4 & \infty & \infty & \infty & \infty & \infty & 0 \\ 5 & 11 & \infty & \infty & \infty & \infty & 11 \\ \hline & & & & & & 13 \end{array} \Rightarrow \begin{array}{c|ccccc} 9 & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & \infty & \infty & \infty \\ 3 & \infty & \infty & \infty & \infty & 0 \\ 4 & \infty & \infty & \infty & \infty & \infty \\ 5 & 0 & \infty & \infty & \infty & \infty \end{array}$$

Therefore the cost of the path A(2,3) is 52

$$C(2,3) + C(6) + \hat{r} = 11+28+13$$
$$= 52$$

Consider the path (2, 5):
Change all entries of row 2 and column 5 of A to ∞ and also set A(5, 1) to ∞. Applyrow and column reduction for the rows and columns.

$$\begin{array}{c|ccccc} 10 & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & \infty & \infty & \infty \\ 3 & 0 & \infty & \infty & \infty & \infty \\ 4 & \infty & \infty & \infty & \infty & \infty \\ 5 & \infty & \infty & 0 & \infty & \infty \end{array}$$

Therefore, the cost of A(2,5) is 28.

$$C(2,5) + C(6) + \hat{r} = 0+28+0$$
$$= 28$$

The tree organization up to this point is as follows:

The cost of the paths between (2, 3) = 52 and (2, 5) = 28. The cost of the path between(2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix. So consider the matrix no.10 for further processing.

10

$$\begin{array}{c|c} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{pmatrix} \end{array}$$

The new possible path is (5,3). Consider the path (5, 3). Change all entries of row 5 and column 3 of A to ∞ and also set A(3, 1) to ∞. Apply row and column reduction for therows and columns. Then the resultant matrix is

$$\begin{array}{c|c} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{pmatrix} \end{array}$$

The cost of A(5,3) is 28

$$C(5,3) + C(10) + \hat{r} = 0+28+0$$
$$= 28$$

The overall tree organization is as follows:

The path of the travelling sales person problem is 1— 4 – 2 – 5 – 3 – 1. Theminimum cost of the path is : 10 + 6 + 2 + 7 + 3 = 28

0/1 Knapsack problem using FIFO:

0/1 KNAPSACK PROBLEM

Consider the instance: M = 15, n = 4, (p1, p2, p3, p4) = (10, 10, 12, 18) and (w1, w2,w3, w4) = (2, 4, 6, 9).0/1 knapsack problem can be solved by using branch and bound technique. In thisproblem we will calculate lower bound and upper bound for each node.

Formula for finding upper and lower bound are,

$$\text{Upper bound} = \sum P_i * X_i \quad \text{and} \quad \text{Lower bound} = \sum P_i * X_i \text{ with fraction part}$$

Place first item in knapsack. Remaining weight of knapsack is 15 – 2 = 13. Place next item w2 in knapsack and the remaining weight of knapsack is 13 – 4 = 9. Place next item w3 in knapsack then the remaining weight of knapsack is 9 – 6 = 3. No fractions are allowed in calculation of upper bound so w4 cannot be placed in knapsack.

Profit = P1 + P2 + P3 = 10 + 10 +12

So, Upper bound = 32

To calculate lower bound we can place w4 in knapsack since fractions are allowed in calculation of lower bound.

Lower bound = 10 + 10 + 12 + ((18/9)*3) = 32 + 6 = 38

Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

Therefore, Upper bound (U) = -32

Lower bound (L) = -38

We choose the path, which has minimum difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.

Now we will calculate upper bound and lower bound for nodes 2, 3. For node 2, x1= 1, means we should place first item in the knapsack.

U = 10 + 10 + 12 = 32, make it as -32

L = 10 + 10 + 12 + ((18/9)*3) = 32 + 6 = 38, make it as -38

For node 3, x1 = 0, means we should not place first item in the knapsack.

U = 10 + 12 = 22, make it as -22

L = 10 + 12 +((18/9)*5) = 10 + 12 + 10 = 32, make it as -32

Next, we will calculate difference of upper bound and lower bound for nodes 2, 3

For node 2, U – L = -32 + 38 = 6

For node 3, U – L = -22 + 32 = 10

Choose node 2 to grow the tree, since it has minimum difference value of 6.

Now we will calculate upper bound and lower bound for nodes 4, 5. For node 4, x2= 1, means we should place first item in the knapsack.

U = 10 + 10 + 12 = 32, make it as -32

L = 10 + 10+ 12 + ((18/9)*3) = 32 + 6 = 38, make it as -38

For node 5, x2 = 0, means we should not place first item in the knapsack.

U = 10 + 12 = 22, make it as -22

L = 10 + 12 +((18/9)*7) = 10 + 12 + 14 = 36, make it as -36

Now Calculate difference of lower and upper bound of nodes 4 and 5.

For node 4, U – L = -32 + 38 = 6

For node 5, U – L = -22 + 36 = 14

Choose node 4 to grow the tree, since it has minimum difference value of 6.

Calculate difference of lower and upper bound of nodes 6 and 7.

For node 6, U – L = -32 + 38 = 6

For node 7, U – L = -38 + 38 = 0

Choose node 7 to grow the tree, since it has minimum difference value of 0.

Calculate difference of lower and upper bound of nodes 8 and 9.

For node 8, U – L = -38 + 38 =0

For node 9, U – L = -20 + 20 =0

Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. So discard node 9 since, it has maximum upper bound.

Consider the path from 1 → 2 → 4 → 7 → 8

X1 =1 X2 =1 X3 =0 X4 = 1

The solution for 0/1 Knapsack problem is (x1, x2, x3, x4) = (1, 1, 0, 1)

Maximum profit is: $\sum P_i * X_i$ = (10 x 1) + (10 x 1) + (12 x 0)+1 (18 x 1)

maximum Profit = 10 + 10 + 0 + 18 = 38

M

Portion of state space tree using FIFO Branch and Bound for above problem: As follows:

Algorithm :

```
Algorithm UBound(cp, cw, k ,m)
// cp, cw,  m are the current profit, current weight, maximum capacity of the
// bag respectively.
{
b = cp; c = cw;
for i = k + 1 to n do
{
   if ( c + w[i] ≤ m) then
   {
      c = c + w[i]; b = b + p[i];
  }
}
return b;
}
```

0/1 knapsack using LIFO (Stack) ①

→ Solve the 0/1 knapsack using LIFO BB for following $n=4, (P1 \ldots P4) = (10, 10, 12, 18)$ $(w1 \ldots w4) = (2, 4, 6, 9)$, $M=15$

Sol: Given profit into negative

$(P1 \ldots P4) = (-10, -10, -12, -18)$

Calculate upper & lower Bound for node 1

$\hat{U} = -10 - 10 - 12 = -32$

-12 6
-10 4
-10 2

$\hat{C} = -10 - 10 - 12 - (3/9 * 18) = -38$

∴ If $\hat{C} > u$ is true then that node is killed otherwise expanded node

① $\hat{C} = -38$
$u = -32$

∴ $-38 > -32$ ✗ to expanded the node 1

∴ if $\hat{u} < u$ is true then updated u

$-32 < -32$ ✗ not updated to u

By using stack

3
2
1

∴ Here top of Stack is node 3

For node 3: $x_1 = 0$

②

$-18 \times 5/9$	$5/9 \times 9$
-12	6
-10	4

15

-12	6
-10	4

$\hat{c} = -10 - 12 - 10 = -32$ $\quad$ $\hat{u} = -10 - 12 = -22$

∴ $\hat{c} > u = -32 > -32$ ✗ expanded node 3

∴ $\hat{u} < u = -22 < -32$ ✗ Not updated to u

Next top of node stack is Node 2

For node 2 : $x_1 = 1$

$-18 \times 3/9$	$3/9 \times 9$
-12	6
-10	4
-10	2

-12	6
-10	4
-10	2

7
6
5
4
~~3~~
~~2~~

$\hat{c} = -10 - 10 - 12 - 6$ $\quad$ $\hat{u} = -10 - 10 - 12$

$= -38$ $\quad$ $= -32$

∴ $\hat{c} > u = -38 > -32$ ✗ expanded the node 2

∴ $\hat{u} < u = -32 < -32$ ✗ Not updated u

Next top of stack is node 7

For node 7 $x_1 = 0, x_2 = 0$

(3)

-18	9
-12	6

-18	9
-12	6

$\hat{c} = -12 - 18 = -30$ $\hat{u} = -12 - 18 = -30$

$\therefore \hat{c} > u = -30 > -32$ ✓ kill the node 7

$\therefore \hat{u} < u = -30 < -32$ ✗ Not updated

Next top of stack is 6

For node 6 : $x_1 = 0, x_2 = 1$

$-5/9 \times 18$	$5/9 \times 9$
-12	6
-10	4

-12	6
-10	4

9
8
~~7~~
~~6~~
5
4

stack

$\hat{c} = -10 - 12 - 10 = -32$ $\hat{u} = -10 - 12 = -22$

$\therefore \hat{c} > u = -32 > -32$ ✗ expanded node 6

$\therefore \hat{u} < u = -22 < -32$ ✗ u is not updated.

Next top of stack is 9

For Node 9 : $x_1 = 0, x_2 = 1, x_3 = 0$

-18	9
-10	4

-18	9
-10	4

$\hat{c} = -10 - 18 = -28$ $\hat{u} = -10 - 18 = -28$

$\therefore \hat{c} > u = -28 > -32$ ✓ kill the node 9

$\hat{u} < u = -28 < -32$ ✗ Not updated u

Next top of node in stack is node 8

For node 8: $x_1=0, x_2=1, x_3=1$

$-18\times5/9$	$5/9\times9$
-12	6
-10	4

-12	6
-10	4

$\hat{c}=-10-12-10=-32$ $\quad \hat{u}=-22$

$\therefore \hat{c}>u=-32>-32$ ✗ expanded the node 8

$\therefore \hat{u}<u=-22<-32$ ✗ u is not updated

~~11~~
~~10~~
~~9~~
~~8~~
~~7~~
~~6~~
5
4

stack

Next top of stack is node 11

For node 11: $x_1=0, x_2=1, x_3=1, x_4=0$

-12	6
-10	4

-12	6
-10	4

$\hat{c}=-22$ $\quad \hat{u}=-22$

$\therefore \hat{c}>u=-22>-32$ ✓ kill the node 11

$\therefore \hat{u}<u=-22<-32$ ✗ u is not updated

Next top of node of stack is node 10

For node 10 : $x_1=0, x_2=1, x_3=1, x_4=1$

(5)

$-18\times 5/9$	$5/9\times 9$
-12	6
-10	4

-12	6
-10	4

$\hat{u}=-10-12=-32$

$\hat{c}=-10-12-10=-32$

$\therefore \hat{c}>u=-32>-32$ expanded node 10.

but all object include into bag so kill

$\therefore$ It is solution ~~inside~~ the node

Next top of node of stack is node 5

For Node 5 : $x_1=1, x_2=0$

$-18\times 7/9$	$7/9\times 9$
-12	6
-10	2

-12	6
-10	2

$\hat{u}=-22$

13
12
8
4

Stack

$\hat{c}=-10-12-14=-36$

$\therefore \hat{c}>u=-36>-32$ ✗ expanded node 5

$\therefore \hat{u}<u=-22<-32$ ✗ Not updated

Next top of stack is node 13

For node 13 $x_1=1, x_2=0, x_3=0$

-18	9
-10	2

-18	9
-10	2

$\hat{u}=-28$

$\hat{c}=-28$

$\therefore \hat{c}>u=-28>-32$ ✓ Kill the node

$\hat{u}<u=-28<-32$ ✗ u is not update

For node 12 $x_1=1, x_2=0, x_3=1$

$-18\times 7/9$	$7/9\times 9$
-12	6
-10	2

$\hat{c}=-10-12-14=36$ $\hat{u}=-10-12=-22$

$\therefore \hat{c}>u=-36>-32\times$ expanded

$\hat{u}<u=-22<-32\times$ Not updated

Top of stack is node 15

For node 15 : $x_1=1, x_2=0, x_3=1, x_4=0$

-12	6
-10	2

$\hat{c}=-22$ $\hat{u}=-22$

$\therefore \hat{c}>\hat{u}=-22>-32$ ✓ Kill the node 15

$\hat{u}<u=-22<-32\times$ Not updated

Top not stack is node 14

For node 14 $x_1=1, x_2=0, x_3=1, x_4=1$

$-18\times 7/9$	$7/9\times 9$
-12	6
-10	2

$\hat{c}=-10-12-14=-36$ $\hat{u}=-10-12=-22$

$\therefore \hat{c}>u=-36>-32\times$ expanded

all object are included

$\hat{u}<u=-22<-32\times$ Kill the node

Top of Stack is node 4.

For node 4: $x_1=1, x_2=1,$

$-3/9\times18$	$3/9\times9$
-12	6
-10	4
-10	2

-12	6
-10	4
-10	2

7

~~19~~
~~18~~
~~17~~
16
~~14~~

$\hat{c} = -10-10-12-6$ $\quad \hat{u} = -10-10-12 = -32$ Stack

$= -38$

$\therefore \hat{c} > u = -38 > -32$ ✗ expanded

$\hat{u} < u = -32 < -32$ ✗ Not updated

For node 17: $x_1=1, x_2=1, x_3=0$

-18	9
-10	4
-10	2

-18	9
-10	4
-10	2

$\hat{c} = -38$ $\quad \hat{u} = -38$

$\hat{c} > u = -38 > -32$ ✗ expanded

$\hat{u} < u = -38 < -32$ ✓ updated $u = -38$

Top of Stack is node 9

For node 19: $x_1=1, x_2=1, x_3=0, x_4=0$

-10	4
-10	2

$\hat{c} = -20$ $\quad \hat{u} = -20$

$\hat{c} > u = -20 > -38$ ✓ killed

$\hat{u} < u = -20 < -38$ ✗ Not updated

Top of stack is node 18

For node 18 $x_1=1, x_2=1, x_3=0, x_4=1$

$\hat{c}=-38$ $\hat{u}=-38$

$\therefore \hat{c} > u = -38 > -38$ ✗ expanded all object included it is solution path because $m=15$ & $w=15$

$\therefore \hat{u} < u = -38 < -38$ ✗ Not updated

Top of stack is node 16

For node 16 $x_1=1, x_2=1, x_3=1$

$\hat{c}=-10-10-12-6=-36$ $\hat{u}=-10-10-12=-32$

$\therefore \hat{c} > u = -36 > -38$ ✓ Killed

$\therefore \hat{u} < u = -32 < -38$ ✗ Not updated

$(x_1, x_2, x_3, x_4) = (1, 1, 0, 1)$

Max profit $= \sum p_i x_i$
$= 1\times10 + 1\times10 + 0 + 1\times18 = 38$

weight $= \sum w_i x_i$
$= 1\times2 + 1\times4 + 0 + 1\times9 = 15$

→ Slove the 0/1 knapsack by using FIFO

$M = 15,\ n = 4,\ (P1 \ldots P4) = (10, 10, 12, 18)$

$(w1 \ldots w4) = (2, 4, 6, 9)$

0/1 Knapsack using FIFO

→ solve the 0/1 knapsack using FIFO BB for following $n=5$, $m=12$ $(P1-P5)=(10,15,6,8,4)$ $(w1-w5)=(4,6,3,4,2)$. Find out optimal Solution

Sol: $(P1-P5)=(-10,-15,-6,-8,-4)$ $m=12$

$(w1-w5)=$ ~~(4,2,3,4,1)~~ $(4,6,3,4,2)$

C = lower Bound with $F.P = \Sigma P_i x_i + FP$

u = upper Bound $= \Sigma P_i x_i$

For node 1:

① $\hat{u} = -29$
$\hat{C} = -25$

$\hat{C} = -10-15-(2/3 * 6) = -29$

$\hat{u} = -10-15 = -25$ $\quad -29 > -25$ ✗ so node 1 is extended

If $\hat{C} > u$ is true, node 1 is killed otherwise node is extended.

For node 2:

$\hat{C} = -10-15-4 = -29$ $\quad \hat{u} = -25$

$\therefore -29 > -25$ ✗ node 2 is Extended

$\hat{u} < u \quad -25 < -25$ ✗ ∴ Not Updated

Global Upper Bound (u)

(14)

For node 3:

$3/4 * 8 = 6$	$3/4 * 4 = 3$	
-6	3	12
-15	6	

-16	3	12
-15	6	

$\hat{c} = -15 - 6 - 6 = -27$ $\quad \hat{u} = -15 - 6 = -21$

If $\hat{c} > u = -27 > -25$ ✗ node 3 can be extended

∴ $\hat{u} < u = -27 < -25$ Not updated.

Queue

For node 4: $x_1 = 1, x_2 = 1$

$-6 \times 2/3$	$3 \times 2/3$	
-15	6	12
-10	4	

-15	6	
-10	4	12

$\hat{c} = -10 - 15 - 4 = -29$ $\quad \hat{u} = -10 - 15 = -25$

∴ $\hat{c} > u \; (-29 > -25)$ ✗ expanded node 4

$\hat{u} < u \; (-25 < -25)$ ✗ Not updated.

For node 5: $x_1 = 1, x_2 = 0$

-9	4
-6	3
-10	4

$-4\times 1/2$	$1/2\times 2$	
-8	4	
-6	3	$\}12$
-10	4	

-8	4	
-6	3	$\}12$
-10	4	

$\hat{c} = -26$ $\quad$ $u = -24$

$\hat{u} < u$ $\quad -24 < -25$ F $\therefore$ Not updated

$\hat{c} > u$ $\quad -26 > -25$ X $\quad$ node 5 expanded

For node 6: $x_1 = 0, x_2 = 1$

$3/4\times -8$	$3/4\times 4$	
-6	3	$\}12$
-15	6	

-6	3	
-15	6	$\}12$

$\hat{c} = -27$ $\quad$ $\hat{u} = -21$

$\hat{c} > u = -27 > -25$ X node 6 is extended

$\hat{u}$ LOU $= -21 < -25$ X Not updated u.

For node 7: $x_1 = 0, x_2 = 0$

-4	2	
-8	4	
-6	3	

-4	2	
-8	4	
-6	3	

$\hat{c} = -13$ $\quad$ $u = -18$

$\hat{c} > u = -18 > -25$ ✓ killed the node 7

$\hat{u} < u = -18 < -25$ ✗ Not updated u.

(16)

In Queue, enqueue is node 8

For node 8 $x_1 = 1, x_2 = 1, x_3 = 1$

$w \rightarrow 4 + 6 + 3 = 13 > 12 (m)$

Sum of object weight is greater than the capacity of bag so node is kill

next enqueue is (9)

For node 9 $x_1 = 1, x_2 = 1, x_3 = 0$

2/4×8	2/4×4
−15	6
−10	4

} 12

−15	9
−10	4

} 12

$\hat{c} = -10 - 15 - 4 = -29$ $\hat{u} = -25$

$\therefore \hat{c} > u = -29 > -25$ ✗

node 9 is extended

$\hat{u} < u = -25 < -25$ ✗ Not updated

Next enqueue is 10

For node 10 $x_1 = 1, x_2 = 0, x_3 = 1$

1/2×−4	1/2×2
−8	4
−6	3
−10	4

12

$\hat{c} = -10 - 6 - 8 - 2$
$= -26$

−8	4
−6	3
−10	4

} 12

$\hat{u} = -24$

$\therefore \hat{c} > u = -26 > -25$ ✗ expanded

$\therefore \hat{u} < u = -24 < -25$ Not updated

Next enqueue is node 11

$m = 12$

For node 11

−4	2
−8	4
−10	4

$\hat{c} = -22$

−4	2
−8	4
−10	4

$\hat{u} = -10 - 8 - 4 = -22$

$\therefore$ $\hat{c} > u = (-22 > -25)$ ✓ kill node 11

$\hat{u} < u = -22 < -25$ ✗ Not updated

Next node 12 in queue $m = 12$

3/4×−8	3/4×4
−6	3
−15	6

$\hat{c} = -15 - 6 - 6$
$= -27$

−6	3
−15	6

$\hat{u} = -15 - 6$
$= -21$

$\therefore \hat{c} > u = -27 > -25$ ✗ node 12 is expanded

$\hat{u} < u = -21 < -25$ ✗ Not updated

23
22
21
20
19
18
17
16
15
~~14~~
~~13~~
~~12~~
~~11~~

Next node 13: $x_1=0, x_2=1, x_3=0$

-4	2
-8	4
-15	6

-4	2
-8	4
-15	6

$\hat{c} = -27$ $\hat{u} = -27$

$\hat{c} > \hat{u}$

$\therefore -27 > -25$

$\therefore \hat{c} > u = -27 > -25$ ✗ ~~not~~ updated

$u = -\cancel{25}\ -27$

Next node in Queue is

For node 14 $x_1=1, x_2=1, x_3=0, x_4=1$

$4+6+4 > 12$

$14 > 12$ overall weight

$\therefore$ killed the node 14

For node 15 $x_1=1, x_2=1, x_3=0, x_4=0$ $m=12$

-4	2
-15	6
-10	4

-4	2
-15	6
-10	4

$\hat{c} = -10-15-4$ $\hat{u} = -29$

$= -29$

$\therefore \hat{c} > u = -29 > -27$ ✗ node 15 is expanded

$\therefore \hat{u} < u = -29 < -27$ ✓ updated

$u = -\cancel{27}\ -29$

Next in Queue is Node 16 : $x_1=1, x_2=0,$
$x_3=1, x_4=1$

-8	4
-6	3
-10	4

-8	4
-6	3
-10	4

$\hat{c} = -10-6-8-2$
$= -26$

$\hat{u} = -10-6-8$
$= -24$

$\therefore \hat{c} > u = -26 > -29$ ✓ kill

$\therefore \hat{u} < u = -24 < -29$ ✗ $\hat{u}$ is not updated

Next node in Queue is Node 17 :

$x_1=1, x_2=0, x_3=1, x_4=0$

-4	2
-6	3
-10	4

-4	2
-6	3
-10	4

23
22
21
20
19
~~18~~
~~17~~
~~16~~
~~18~~

Queue

$\hat{c} = -10-6-4$
$= -20$

$\hat{u} = -10-6-4 = -20$

$\therefore \hat{c} > u = -20 > -29$ ✓ kill node 17

$\hat{u} < u = -20 < -29$ ✗ Not updated of u

Next node in Queue is Node 18

$x_1=0 \; x_2=1, x_3=1 \; x_4=1$

-5	4
-6	3
-15	6

$\rightarrow w = 6+3+4 = 13 >$ capacity of bag 12

infeasible solution so kill the node 18.

Next Node 19 in Queue

for node 19:

$x_1=0,\ x_2=1,\ x_3=1,\ x_4=0$

$\hat{c} = -15-6-4$ $\quad$ $\hat{u} = -15-6-4 = -25$

$= -25$

$\therefore \hat{c} > u = -25 > -29$ ✓ Kill node 19

$\therefore \hat{u} < u = -25 < -29$ ✗ Not updated of u

For node 20 Next Node 20 in Queue

$x_1=0\ \ x_2=1\ \ x_3=0\ \ x_4=1$

$\hat{c} = -27$ $\quad$ $\hat{u} = -15-8-4 = -27$

$\hat{c} > u = -27 > -29$ ✓ Kill the node 20

$\hat{u} < u = -27 < -29$ ✗ Not updated.

For node 21 $x_1=0, x_2=1, x_3=0, x_4=0$

$\hat{c} = -15-4 = -19$ $\quad$ $\hat{u} = -15-4 = -19$

$\therefore \hat{c} > u = -19 > -29$ ✓ Kill the node 21

$\hat{u} < u = -19 < -29$ ✗ Not updated

Next ~~queue~~ node 22 in queue:

For node 22 $x_1=1, x_2=1, x_3=0, x_4=0, x_5=1$

-4	2
-15	6
-10	4

$\}12$

-4	2
-15	6
-10	4

$\hat{c} = -10-15-4 = -29$ $\hat{u} = -29$

$\therefore \hat{c} > u = -29 > -29$ ✗ expanded Node 22

but all objects are include i.e., Active node

$\therefore \hat{u} < u = -29 < -29$ ✗ Not updated

For node 23

-15	6
-10	4

-15	6
-10	4

$\hat{c} = -10-15 = -25$ $\hat{u} = -10-15 = -25$

$\hat{c} > u = -25 > -29$ ✓ till node 25

$\hat{u} < u = -25 < -29$ ✗ Not updated.

~~23~~
~~22~~
~~21~~
~~20~~
~~18~~
~~19~~

$u = -29$

$(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 0, 1)$

maximum ~~profit~~ weight $= \sum w_i x_i$

$= 4\times1 + 6\times1 + 3\times0 + 4\times0 + 2\times1$

$= 12$

To convert minimum into max

max profit $= \sum P_i x_i$

$= 10\times1 + 15\times1 + 0\times0 + 0 + 4\times1 = 29$

→ Slove the 0/1 knapsack using LIFO for following

$n=5,\ m=12,\ (P1 \cdots P5) = (10, 15, 6, 8, 4)$

$(W1 \cdots W5) = (4, 6, 3, 4, 2)$. Find out optimal solution.

0/1 knapsack problem using LIFO Branch & Bound

→ Slove the 0/1 knapsack using LIFO BB for following $n=5$ $m=12$ $(P1-P5)=(10,15,6,8,4)$ $(w1,-w5)=(4,6,3,4,2)$. Find out optimal solution

Sol: $(P1-P5)=(-10,-15,-6,-8,-4)$

$(w1-w5)=(4,6,3,4,2)$

$c=$ lower Bound $=\sum P_i x_i$ with FP

$u=$ upper Bound $=\sum P_i x_i$

For node 1: $m=12$

$c=\underset{w(4+6+2/3)}{-10-15-(6\times 2/3)}=-10-15-4=-29$

$u=\underset{w(4+6)}{-10-15}=-25$

① $\hat{u}=-25$ $\hat{c}=-29$

$u=$ Global U.B $=-25$

node 1 is placed into stack

Stack

To check the condition $(\hat{c} > u)$ is True then that node is kill otherwise that node can be extended ②

$\hat{c} > u = -29 > -25$ X

node 1 can be extended.

∴ Global upper Bound updated or not

i.e., $\hat{u} < u$ $(-25 < -25)$ X

$\hat{u}$ is not updated

or stack

node 1 is pop out. node 2, & 3 placed in stack

Top node is node 3: $x_1 = 0$

$\hat{c} = \cancel{-15-6} = -\cancel{10} - 15 - 6 - \left(\frac{3}{4} \times \cancel{8}^{2}\right)$

$= -15 - 6 - 6 = -27$

$\hat{u} = -\cancel{10} - 15 - 6 = -21$

∴ $\hat{c} > u = -27 > -25$ (false)

node 3 can be extended

∴ $\hat{u} < u = -21 < -25$ (false)

Global UB not updated

node 3 is popout. node 4 & 5 placed in stack

$u = -25$ (3)

∴ Now Top node is node 5

node 5: $x_1 = 0, x_2 = 0$

$(P_3)\ (P_4)\ (P_5)$

$\hat{c} = -6 - 8 - 4 = -18$

$\hat{u} = -6 - 8 - 4 = -18$

∴ $\hat{c} > u = -18 > -25$ ✓ node 5 is kill

; $\hat{u} < u = -18 < -25$ X u is not updated

node 5 is popout from stack

Now Top of Stack is node 4

ode 4: $x_1 = 0, x_2 = 1$

$\hat{c} = -\cancel{10} - 15 - 6 - (\frac{3}{4} \times 8) = -27$

$\hat{u} = -\cancel{10} - 15 - 6 = -21$

$\hat{c} > u = -27 > -25$ X node 4 can be extended

$\hat{u} < u = -21 < -25$ X Global UB not updated.

Now Top of Stack is node 7

node 7: $x_3=0,\ x_1=0,\ x_2=1$

$\hat{c} = -\cancel{10} -15-6-\left(\cancel{\frac{2}{4}\times -8}\right) \cancel{=-27}\ -4 = -27$

$\hat{u} = -\cancel{10} -15-6-4 = -27$

$\therefore\ \hat{c} > u\ (-27 > -25)$ false

$\therefore$ node 7 can be extended

$\therefore\ \hat{u} < u\ (-27 < -25)$ True

$\therefore$ u is updated $u = -27$

Top of stack is node 9

node 9: $x_4=0,\ x_1=0,\ x_2=1,\ x_3=0$

$\hat{c} = -\cancel{10} - 15 - \cancel{6} - \cancel{8} - 4 = -19$

$\hat{u} = -15 - 4 = -19$

$\hat{c} > u = -19 > -27$ ✓

node 9 is killed

$\hat{u} < u = -19 < -27$ X $\hat{u}$ is not updated

popout node 9 from stack. next Top of stack is node 8

for node 8: $x_1=0,\ x_2=1,\ x_3=0,\ x_4=1$

$\hat{c} = -\cancel{10} - 15 - \cancel{6} - 8 - 4 = -\ -27$

$\hat{u} = -15 - 8 - 4 = -27$

$\therefore\ \hat{c} > u = -27 > -27$ X node 8 is extended

$\therefore\ \hat{u} < u = -27 < -27$ X Not updated

Global UB
$u=-27$

Top node of Stack is node 11

node 11: $x_1=0, x_2=1, x_3=0, x_4=1, x_5=0$

$\hat{c} = -\cancel{10} - 15 - \cancel{6} - 8 - \cancel{3} = -23$

$\hat{u} = -15-8 = -23$

$\therefore \hat{c} > u = -23 > -27$ (True)

node 11 is killed

$\therefore \hat{u} < u = -23 < -27$ X Not updated

Stack

Now Top of Stack is node 10.

For node 10: $x_1=0, x_2=1, x_3=0, x_4=0, x_5=1$

$\hat{c} = -15-8-4 = -27$ (7)

$\hat{u} = -15-8-4 = -27$

$\therefore \hat{c} > u = -27 > -27$ ✗ node 10 is extended

$\therefore \hat{u} < u = -27 < -27$ ✗ Not updated

node 10 is extended but given objects are 5. so it is solution node.

Next TopofStack is 6

~~10~~
~~6~~
2

Stack

for node 6

$x_1=0 \quad x_2=1 \quad x_3=1$

$\hat{c} = -\cancel{10}, -15, -6 = \left(\frac{3}{4} \times \frac{-2}{-8}\right) = -27$

$\hat{u} = -15-6 = -21$

$\therefore \hat{c} > u = -27 > -27$ node 6 is extended

$\therefore \hat{u} < u = -21 < -27$ Not updated

13
12
~~6~~
2

Stack

TopofStack is node 13

For node 13: $x_1=0, x_2=1, x_3=1, x_4=0$

$\hat{c} = -15-6-4 = -25$

$\hat{u} = -15-6-4 = -25$

$\therefore \hat{c} > u = -25 > -27$ ✓ killed

$\therefore \hat{u} < u = -25 < -27$ Not updated

~~13~~
12
~~6~~
2

Stack

Next Top of ~~node~~ stack is 12 node

For node 12: $x_1 = 0, x_1 = 1, x_2 = 1, x_4 = 1$

$\hat{c} =$ $\quad$ $c0 \rightarrow 6 + 3 + 4 = 13 > 12$

infeasible sol

kill node 12

Next Top of stack is node 2

For node 2: $x_1 = 1$

$\hat{c} = -10 - 15 - (2/3 \times -6) = -29$

$\hat{u} = -10 - 15 = -25$

$\therefore \hat{c} > u = -29 > -27$ ✗ node 2 can extended

$\hat{u} < u = -25 < -27$ Not updated

Top of Stack is node 15

15
12
2

For node 15: $x_1 = 1, x_2 = 0$

$\hat{c} = -10 - 6 - 8 - (1/2 \times -4) = -26$ (9)

$\hat{u} = -10 - 6 - 8 = -24$

$\therefore \hat{c} > u = -26 > -27$ ✓ killed

$\therefore \hat{u} < u = -24 < -27$ ✗ Not updated

Top of stack is node 14

node 14 $x_1 = 1, x_2 = 1$

$\hat{c} = -10 - 15 - (2/3 \times 6) = -29$

$\hat{u} = -10 - 15 = -25$

$\therefore \hat{u} > u = -29 > -27$ ✗ node 14 is extended

$\therefore \hat{u} < u = -25 < -27$ ✗ Not updated

For node 17: Next Top of ~~node~~ stack is node 17

$x_1=1, x_2=1, x_3=0$

$\hat{c} = -10-15-(2/3 \times -6) = -29$

$\hat{u} = -10-15 = -25$

$\therefore \hat{c} > u = -29 > -25$ ✗ node 17 is extended

$\therefore \hat{u} \not< u = -25 < -27$ ✗ Not updated

For node 19: Top of stack

$x_1=1, x_2=1, x_3=0, x_4=0$

$\hat{c} = -10-15-4 = -29$

$\hat{u} = -10-15-4 = -29$

19
18
~~17~~
16

Stack

$\therefore \hat{c} > u \ (-29 > -27)$ ✗ Node 19 is extended

$\therefore \hat{u} < u \ (-29 < -27)$ ✓ update $u = -\cancel{24}\ -29$

For node 21 $x_1=1, x_2=1, x_3=0$

$x_4=0, x_5=0$

$\hat{c} = -10-15 = -25$

$\hat{u} = -10-15 = -25$

21
20
~~19~~
18
16

Stack

$\therefore \hat{c} > u = -25 > -29$ ✓ Node 21 is killed

$\therefore \hat{u} < u = -25 < -29$ ✗ Not updated

Next Top of Stack is Node 20

$x_1=1, x_2=1, x_3=0, x_4=0, x_5=1$

$\hat{c} = -10-15-4 = -29$

$\hat{u} = -10-15-4 = -29$

~~21~~
20
18
16

Stack

$\therefore \hat{c} > u = -29 > -29$ ✗ extended

$\hat{u} < u = -29 < -29$ ✗ Not updated

Node 20 is extended but subjects is considered i.e., solution node

(11)

Top of stack is node 18

$x_1=1, x_2=1, x_3=0, x_4=1$

$w \rightarrow 4+6+4 = 14 > 12$

$\therefore$ Here at node 18, to get infeasible sol kill node 18

Top of stack is node 16

$x_1=1, x_2=1, x_3=1$

$4+6+3 = 13 > 12$

$\therefore$ infeasible sol so killed node 16

Top of stack is node 16

$\hat{c}_{(20)} = -29$, $\hat{c}_{(10)} = -27$ is minimum i.e., $\hat{c}(20)$

$(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 0, 1)$

max profit $= \Sigma P_i x_i = -29 \rightarrow$ To convert from min to max i.e., 29 is max profit

max weight $= 12$

$\hat{c} = -27$ $\hat{c} = -29$

minimum is $\hat{c} = -29$ ∴

solution $= (x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 0, 1)$

profit $= \sum P_i x_i$

$= 1*10 + 1*15 + 0*6 + 0*8 + 1*4$

$= 10 + 15 + 0 + 0 + 4 = 29$

ght $= \sum w_i x_i$

$= 4*1 + 6*1 + 3*0 + 4*0 + 1*2$

$= 4 + 6 + 0 + 0 + 2 = 12$

P, NP, NP –HARD AND NP – COMPLETE PROBLEMS

Basic Concepts:

P – Polynomial time and NP Nondeterministic Polynomial time.

The problems has best algorithms for their solutions have "Computing times",that cluster into two groups.

Group-1

Problems with solution time bound by a polynomial of a small degree. They are also called "Tractable Algorithms". Most Searching & Sorting algorithms are polynomial time algorithms. Ex: Ordered Search (O (log n)), Polynomial evaluation O(n), Sorting O(n.log n)

Group-II

Problems with solution times not bound by polynomial (simply non polynomial). These are hard or intractable problems. None of the problems in this group has been solved by any polynomial time algorithm. Ex: Traveling Sales Person O(n2 2n), Knapsack O(2n/2)

No one has been able to develop a polynomial time algorithm for any problem in the group –II. So, it is compulsory and finding algorithms whose computing times are greater than polynomial very quickly because such vast amounts of time to execute that even moderate size problems cannot be solved.

Theory of NP-Completeness

Show that may of the problems with no polynomial time algorithms are computational time algorithms are computationally related.

There are two classes of non-polynomial time problems

1. **NP-Hard**
2. **NP-Complete**

NP Complete Problem: A problem that is NP-Complete can solved in polynomial time if and only if (iff) all other NP-Complete problems can also be solved in polynomial time.

NP-Hard: Problem can be solved in polynomial time then all NP-Complete problems can be solved in polynomial time.

All NP-Complete problems are NP-Hard but some NP-Hard problems are not know to be NP-Complete.

Nondeterministic Algorithms

Algorithms with the property that the result of every operation is uniquely defined are termed as deterministic algorithms. Such algorithms agree with the way programs are executed on a computer.

Algorithms which contain operations whose outcomes are not uniquely defined but are limited to specified set of possibilities. Such algorithms are called nondeterministic algorithms.

The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition to be defined later.To specify nondeterministic algorithms, there are 3 new functions.

- Choice(S) arbitrarily chooses one of the elements of sets S
- Failure () Signals an Unsuccessful completion
- Success () Signals a successful completion. Example for Non Deterministic algorithms:

```
Algorithm Search(x){
//Problem is to search an element x
//output J, such that A[J]=x; or J=0 if x is not in A J:=Choice(1,n);
if( A[J]:=x) then
{
Write(J);
Success();
else
{
write(0);
 failure();
}
}
```

Whenever there is a set of choices that leads to a successful completion then one such set of choices is always made and the algorithm terminates. A Nondeterministic algorithm terminates unsuccessfully if and only if (iff) there exists no set of choices leading to a successful signal.

Nondeterministic Knapsack algorithm

```
Algorithm DKP(p, w, n, m, r, x)
{ W:=0;
P:=0;
for i:=1 to n do
{
x[i]:=choice(0, 1);
W:=W + x[i]*w[i];
P:=P + x[i]*p[i];
```

}

if((W>m) or (P<r)) then Failure();

else

success()

}

p given Profits w given Weights

n Number of elements (number of p or w) m Weight of bag limit P Final Profit W Final weight

The Classes NP-Hard & NP-Complete

For measuring the complexity of an algorithm, we use the input length as the parameter. For example, An algorithm A is of polynomial complexity p() such that the computing time of A is O(p(n)) for every input of size n.

Decision problem/ Decision algorithm: Any problem for which the answer is either zero or one is decision problem. Any algorithm for a decision problem is termed a decision algorithm.

Optimization problem/ Optimization algorithm: Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

- **P** is the set of all decision problems solvable by deterministic algorithms in polynomial time.
- **NP** is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

Since deterministic algorithms are just a special case of nondeterministic, by this we concluded that P ⊆NP

Relationship between P & NP

The most famous unsolvable problems in Computer Science is Whether P=NP or P≠NP In considering this problem, Cook's formulated the following question.

If there any single problem in NP, such that if we showed it to be in 'P' then that would imply that P=NP.

Cook answered this question with

Theorem: Satisfiability is in P if and only if (iff) P=NP

Notation of Reducibility

Let L1 and L2 be problems, Problem L1 reduces to L2 (written L1 α L2) iff there is a way to solve L1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L2 in polynomial time

This implies that, if we have a polynomial time algorithm for L2, Then we can solve L1 in polynomial time. Here α is a transitive relation i.e., L1 α L2 and L2 α L3 then L1 α L3

A problem L is NP-Hard if and only if (iff) satisfiability reduces to L ie., Statisfiability α L

A problem L is NP-Complete if and only if (iff) L is NP-Hard and L Є NP

Commonly believed relationship among P, NP, NP-Complete and NP-Hard Most natural problems in NP are either in P or NP-complete.

Examples of NP-complete problems:

Packing problems: SET-PACKING, INDEPENDENT-SET. Covering problems: SET-COVER, VERTEX-COVER. Sequencing problems: HAMILTONIAN-CYCLE, TSP. Partitioning problems: 3-COLOR, CLIQUE.

Constraint satisfaction problems: SAT, 3-SAT.

Numerical problems: SUBSET-SUM, PARTITION, KNAPSACK.