DESIGN AND ANALYSIS OF ALGORITHMS

(DAA)

UNIT – 1

Introduction:

Algorithm

Pseudo Code for Expressing Algorithms

Performance Analysis, Space Complexity

Time Complexity

Asymptotic Notation-

- Big Oh Notation
- Omega Notation
- Theta Notation
- Little Oh Notation

probabilistic analysis

Amortized analysis.

SHRI VISHNU ENGINEERING COLLEGE FOR WOMEN : : BHIMAVARAM

(AUTONOMOUS)

PSUEDOCODE FOR EXPRESSING ALGORITHM

PSEUDOCODE:

Pseudocode *is an informal high level description of the operating principle of a computer program or other algorithm. It uses the structural conventions of a normal programming language, but is intended for human reading rather than machine reading. Pseudocode makes creating programs easier. Programs can be complex and long; preparation is the key.It was difficult to display all parts of a program with a flowchart. It is challenging to find a mistake without understanding the complete flow of a program. That is where pseudocode becomes more appealing.*

ADVANTAGES OF PSUEDOCODE:

Improves the readability of any approach. It's one of the best approaches to start implementation of an algorithm.
Acts as a bridge between the program and the algorithm or flowchart.

The main goal of a pseudo code is to explain what exactly each line of a program should do, hence making the code construction phase easier for the programmer.

Are there alternatives to Pseudocode?

There are some alternatives to Pseudocode. Some of them are flowcharts,drakon charts and unified model language chartds. They will serve the purpose but they comparatively require more resources.

STATEMENTS IN PSEUDOCODE:

A statement is defined as an instruction that directs the computer to perform a specific action. In writing pseudocode, we will refer to singular instructions as statements.

When writing pseudocode, we assume that the order of execution of the statements is from top to bottom. This changes when using control structures, functions and exception handling.

Mathematical operations

Mathematical operations are integral to solution development. They allow us to manipulate the values we have stored. Here are some common mathematical sysmbols:

Assignment: ← or :=
Comparison: =, ≠, <, >, ≤, ≥
Arithmetic: +, −, ×, /, mod

floor/ceiling: ⌊, ⌋, ⌈,⌉*a* ← ⌊*b*⌋+⌈*c*⌉
Logical: and, or
Sums, products: Σ Π
Keywords

A keyword is a word that is reserved by a program because the word has a special meaning. Keywords can be commands or parameters. Every programming language has its own keywords (reserved words). Keywords ***cannot*** *be used as variable names.*

In Pseudocode, they are used to indicate common input-output and processing operations. They are written fully in uppercase.

START: *This is the start of your pseudocode.*
INPUT: *This is data retrieved from the user through typing or through an input device.*
READ / GET: *This is input used when reading data from a data file.*
PRINT, DISPLAY, SHOW: *This will show your output to a screen or the relevant output device.*
COMPUTE, CALCULATE, DETERMINE: *This is used to calculate the result of an expression.*
SET, INIT: *To initialize values.*
INCREMENT, BUMP: *To increase the value of a variable.*
DECREMENT: *To reduce the value of a variable.*

CONDITIONS IN PSEUDOCODE:

During algorithm development, we need statements which evaluate expressions and execute instructions depending on whether the expression evaluated to True or False. Here are some common conditions used in Pseudocode:

IF ➪ ELSE

This is a conditional that is used to provide statements to be executed if a certain condition is met. This also applies to multiple conditions and different variables.

Here is an if statement with one condition

EXAMPLE:

```
IF you are happy
  THEN smile
ENDIF
```

Here is an if statement with an else section. Else allows for some statements to be executed if the "if" condition is not met.

EXAMPLE:

```
IF you are happy THEN
   smile
ELSE
   frown
ENDIF
```

We can add additional conditions to execute different statements if met.

EXAMPLE:

```
IF you are happy THEN
  smile
ELSE IF you are sad
   frown
ELSE
   keep face plain
ENDIF
```

CASE

Case structures are used if we want to compare a single variable against several conditions.

EXAMPLE:

```
INPUT colorCASE color of    red: PRINT "red"
   green: PRINT "green"
   blue: PRINT "blue"OTHERS
   PRINT "Please enter a value color"ENDCASE
```

The OTHERS clause with its statement is optional. Conditions are normally numbers or characters.

ITERATIONS IN PSEUDOCODE:

To iterate is to repeat a set of instructions in order to generate a sequence of outcomes. We iterate so that we can achieve a certain goal.

FOR structure

The FOR loop takes a group of elements and runs the code within the loop for each element.

EXAMPLE:

FOR every month in a year

Compute number of days

ENDFOR

WHILE structure

Similar to the FOR loop, the while loop is a way to repeat a block of code as long as a predefined condition remains true. Unlike the FOR loop, the while loop evaluates based on how long the condition will remain true.

To avoid a scenario where our while loop runs infinitely, we add an operation to manipulate the value within each iteration. This can be through an increment, decrement, et cetera.

EXAMPLE:

PRECONDITION: variable X is equal to 1

WHILE Population < Limit

Compute Population as Population + Births — Deaths

ENDWHILE

FUNCTIONS IN PSEUDOCODE:

When solving advanced tasks it is necessary to break down the concepts in block of statements in different locations. This is especially true when the statements in question serve a particular purpose. To reuse this code, we create functions. We can then call these functions every-time we need them to run.

EXAMPLE:

```
Function clear monitor
  Pass In: nothing
  Direct the operating system to clear the monitor
  Pass Out: nothing
Endfunction
```

*To emulate a function call in pseudocode, we can use the **Call** keyword*

EXAMPLE:

call: clear monitor

PROGRAM WRAPPING IN PSEUDOCODE:

After writing several functions in our pseudocode, we find the need to wrap everything into one container. This is to improve readability and make the execution flow easier to understand.

To do this, we wrap our code as a program. A program can be defined as a set of instructions that performs a specific task when executed.

EXAMPLE:

```
PROGRAM makeacupoftea

END
```

EXCEPTION HANDLING IN PSEUDOCODE:

An exception is an event which occurs during program execution that disrupts the normal flow of the instructions. These are events that are non-desirable.

We need to observe such events and execute code-blocks in response to them. This is called exception handling.

EXAMPLE:

```
BEGIN
   statements
EXCEPTION
   WHEN exception type
      statements to handle exception
   WHEN another exception type
      statements to handle exception
END
```

Pseudo code for expressing an algorithm:

An algorithm is basically a sequence of instruction written in English language and broadly divided into 2 sections.

1. Algorithm heading:

It consists of

(i) Name of algorithm

(ii) Problem description

(iii) Input

(iv) Output

2. Algorithm body: It consists of logical body of algorithm.

Rules for writing an algorithm:

Step.1:

Algorithm is a procedure consisting of heading and body. The heading section consists of keyword algorithm, name of the algorithm and parameters list.

Syntax: Algorithm max(A[] , n)

Step.2:

In the heading section we should write the following things

Problem description

Input

Output

Step.3:

Comments begin with forward slash and continue until the end of the line.

Step.4:

Compound statement should be enclosed with in {and } .

Step.5:

An identifier begin with a letter and an identifier can be a combination of alpha numeric string.

Step.6:

Assigning values to variables is done by using assignment statement.

< Variable >:= < expression >

Step.7:

There are 2 boolean values “True” and “False”. In order to produce these

values we use logical operators such as and, or, not and relational operators such as <, >, <=, >=, ==,!= .

Step.8:

Elements of multi-dimensional arrays are accessed using [and] .

Example:

If 'a' is a 2-dimensional array (i , j)th element of an array is denoted by

a [i , j] . Array indexes start at Zero.

Step.9:

Looping statements are also employed.

Example:

While loop

while< condition > loop

{

< statement 1 >

< statement 2 >

.

.

.

.

< statement n >

}

General form of for loop

for variable := value 1 to n do

{

< statement 1>

(step i := i + 1)

< statement 2 >

.

.

.

.

< statement n >

}

Here, value 1 is initialisation condition. Value n is terminating condition. Sometimes, a keyword 'step' used to denote increment or decrement the value.

A repeat until statement is constructed as

< statement 1 >

< statement 2 >

.

.

.

.

< statement n >

until < condition >

These statements are executed as long as the condition is true.

Step.10:

Input and output are done using the instruction read and write. No format is used to represent the size of input and output quantities.

Examples:-

1) Write an algorithm to find sum of n numbers.

Sol:-

Algorithm sum (n)

//problem description: Sum of given 'n' numbers.

//Input: 1 to n numbers.

//Output: Sum of 'n' numbers.

{

sum := 0;

```
for i := 1 to n do
sum := sum +i;
write (sum);
}
```

2) Write an algorithm to check whether given number is odd or even.

Sol:-

```
Algorithm check (n)
//problem description: To find given no is even or odd.
//Input: One integer value.
//Output: Given no is even or odd.
{
read (n);
if (n%2 == 0)
write ("n is even");
else
write("n is odd");
}
```

3) Algorithm for finding maximum of given n numbers.

Sol:-

```
Algorithm Max (a [ ], n)
//problem description: To find maximum numbers in an array.
//Input: Array elements.
//Output: Maximum value.
{
read (n);
read (a [n]);
max := a[0];
for i:= 1 to n-1 do
  {
```

```
    if (a[i] > max) then
    max := a[i];
    }
write (max);
}
```

Time complexity :

Time complexity of an algorithm quantifies the amount of **time** taken by an algorithm to run as a function of the length of the input. Similarly, Space **complexity** of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

O(n^2): You go and ask the first person of the class, if he has the pen. Also, you ask this person about other 99 people in the classroom if they have that pen and so on, This is what we call O(n^2).

O(n): Going and asking each student individually is O(N).

O(log n): Now I divide the class into two groups, then ask: "Is it on the left side, or the right side of the classroom?" Then I take that group and divide it into two and ask again, and so on. Repeat the process till you are left with one student who has your pen. This is what you mean by O(log n).

Space complexity:-
The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.

Time complexity

Algorithm RSum (A,n)	n<0	n>0
{	0	0
if n<0	1	1
return 0	1	0
else	0	0
return RSum (A,n-1) + A[n]	0	x+1
	2	2+x

x is nthg but which is purely depend on n element

$T(A) = 2$ if $n<0$

$= 2 + x$ if $n>0$

$(2+T(n-1) \Rightarrow 2+2+T(n-1)$

```
Algorithm Rsum (A,n) → 0
    {                          →0
    Sum := 0                   →1
     for i:=1 to n do :        →n+1
          sum := sum+A[i]      →n
       end for
         write sum              →1
         }                      →0
```

$T(A) = 2n+3 \quad n>0$

Space complexity

```
Algorithm sum (A,n)
    {
   declare i, sum → 8 bytes (4+4)
    Sum = 0
  -for i:= 1 to n do
     {
     Sum := Sum + A[i]
      }
   write sum
    }
```

No change

O(1)

```
-Algorithm Sum (A,n)
  {
  create B,n
  -for i:=1 -to n do
    {
    B[i] = A[i]
    }
  }
```

O(n)

Introduction to Asymptotic Notations:

1. What is Asymptotic Notations?

Asymptotic Notations are languages that allow us to analyse an algorithm's run-time performance. Asymptotic Notations identify running time by algorithm behaviour as the input size for the algorithm increases. This is also known as an algorithm's growth rate. Usually, the time required by an algorithm falls under three types –

- Best Case – Minimum time required for program execution
- Average Case – Average time required for program execution.
- Worst Case – Maximum time required for program execution.

2. Types of Asymptotic Notation

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation (Big-O Notation)
- Ω Notation (Big-Omega Notation)
- θ Notation (Theta Notation)

Big – Oh Notation:

The Big-Oh notation defines an upper bound of an algorithm, it bounds a function only from above. The Big-Oh Notation can be used in the following instances:

- For expressing the upper bound or the worst-case complexity of an algorithm.
- For expressing that "time complexity is never more than" or "at most" the given complexity function.

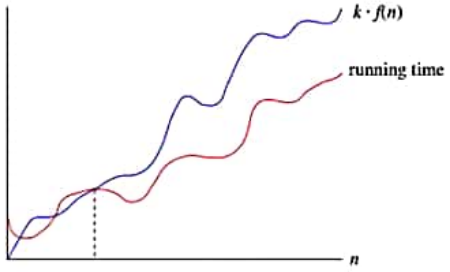
For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.

It would be convenient to have a form of asymptotic notation that means "***the running time grows at most this much, but it could grow more slowly.***" We use "big-Oh" notation for just such occasions.

The Big-O Asymptotic Notation gives us the Upper Bound Idea, mathematically described below:

$f(n) = O(g(n))$, if there exists a positive integer n_0 and a positive constant c, such that $f(n) \leq c * g(n)$ $\forall$ $n \geq n_0$

If a running time is O(f(n)), then for large enough *n*, the running time is at most ***k*f(n)*** for some constant *k*. Here's how to think of a running time that is O(f(n)):



We say that running time is "Big-Oh of f(n)" or just "O of f(n)". We use Big-Oh notation for ***Asymptotic upper bounds,*** since it bounds the growth of the running time from the above for large enough input sizes.

The general step wise procedure for Big-Oh runtime analysis is as follows:

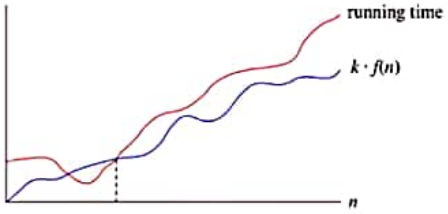
- Figure out what the input is and what 'n' represents.
- Express the maximum number of operations, the algorithm performs in terms of 'n'.
- Eliminate all excluding the highest order terms.
- Remove all the constant factors.

Big Omega Notation:

Big Omega notation is used to define the **lower bound** of any algorithm or we can say **the best case** of any algorithm.
This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.
In simple words, when we represent a time complexity for any algorithm in the form of big-Ω, we mean that the algorithm will take at least this much time to complete its execution. It can definitely take more time than this too.

f(n) = O(g(n)), if there exists a positive integer n_0 and a positive constant c, such that $f(n) \geq c * g(n)$ $\forall$ $n \geq n_0$

If a running time is \Omega(f(n))Ω(*f*(*n*))\Omega, left parenthesis, f, left parenthesis, n, right parenthesis, right parenthesis, then for large enough n*n*n, the running time is at least k \cdot f(n)*k*⋅*f*(*n*)k, dot, f, left parenthesis, n, right parenthesis for some constant k*k*k. Here's how to think of a running time that is \Omega(f(n)) Ω(*f*(*n*))\Omega, left parenthesis, f, left parenthesis, n, right parenthesis, right parenthesis:



We say that the running time is "big-Ω of f(n)*f(n)*f, left parenthesis, n, right parenthesis." We use big-Ω notation for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.

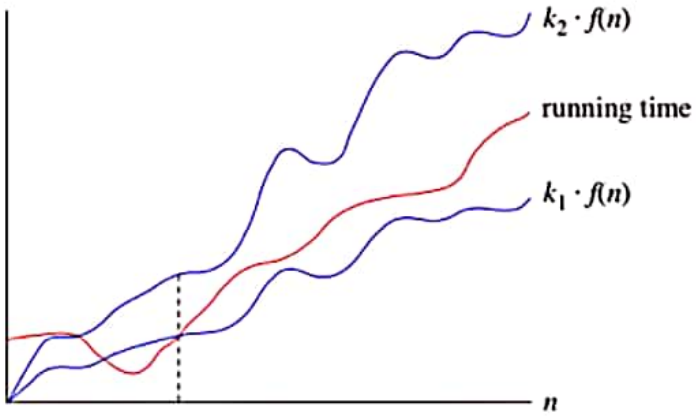
Just as \Theta(f(n))Θ(*f*(*n*))\Theta, left parenthesis, f, left parenthesis, n, right parenthesis, right parenthesis automatically implies O(f(n))*O*(*f*(*n*))O, left parenthesis, f, left parenthesis, n, right parenthesis, right parenthesis, it also automatically implies \Omega(f(n))Ω(*f*(*n*))\Omega, left parenthesis, f, left parenthesis, n, right parenthesis, right parenthesis. So, we can say that the worst-case running time of binary search is \Omega (\log_2 n) Ω(log2*n*) \Omega, left parenthesis, log, start base, 2, end base, n, right parenthesis.
We can also make correct, but imprecise, statements using big-Ω notation. For example, if you really do have a million dollars in your pocket, you can truthfully say "I have an amount of money in my pocket, and it's **at least** 10 dollars." That is correct, but certainly not very precise. Similarly, we can correctly but imprecisely say that the worst-case running time of binary search is \Omega (1)Ω(1) \Omega, left parenthesis, 1, right parenthesis, because we know that it takes **at least** constant time. Of course, typically, when we are talking about algorithms, we try to describe their running time as precisely as possible. We provide the examples of the imprecise statements here to help you better understand big-\OmegaΩ\Omega, big-O*O*O, and big-\ThetaΘ\Theta.

Theta Notation:

When we say tight bounds, we mean that the time complexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.
For example, if for some algorithm the time complexity is represented by the expression $3n^2 + 5n$, and we use the Big-Θ notation to represent this, then the time complexity would be $\Theta(n^2)$, ignoring the constant coefficient and removing the insignificant part, which is 5n.

f(n) = O(g(n)), if there exists a positive integer n_0 and a positive constants c1, c2, such that $c1 * g(n) \leq f(n) \leq c2 * g(n)$ $\forall$ $n \geq n_0$

Here, in the example above, complexity of $\Theta(n^2)$ means, that the average time for any input n will remain in between, $k1 * n^2$ and $k2 * n^2$, where k1, k2 are two constants, thereby tightly binding the expression representing



When we use big-Θ notation, we're saying that we have an **asymptotically tight bound** on the running time. "Asymptotically" because it matters for only large values of n*m*n. "Tight bound" because we've nailed the running time to within a constant factor above and below.

Properties of Asymptotic notations:

1. Transitive

- If f(n) = Θ(g(n)) and g(n) = Θ(h(n)), then f(n) = Θ(h(n))
- If f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n))
- If f(n) = o(g(n)) and g(n) = o(h(n)), then f(n) = o(h(n))
- If f(n) = Ω(g(n)) and g(n) = Ω(h(n)), then f(n) = Ω(h(n))
- If f(n) = ω(g(n)) and g(n) = ω(h(n)), then f(n) = ω(h(n))

2. Reflexivity

- f(n) = Θ(f(n))
- f(n) = O(f(n))
- f(n) = Ω(f(n))

3. Symmetry

- f(n) = Θ(g(n)) if and only if g(n) = Θ(f(n))

4. Transpose Symmetry

- f(n) = O(g(n)) if and only if g(n) = Ω(f(n))
- f(n) = o(g(n)) if and only if g(n) = ω(f(n))

5. Some other properties of asymptotic notations are as follows:

- If f (n) is O(h(n)) and g(n) is O(h(n)), then f (n) + g(n) is O(h(n)).
- The function loga n is O(logb n) for any positive numbers a and b ≠ 1.
- loga n is O(log n) for any positive a ≠ 1, where log n = log2 n.

Eg:- $f(n) = 5n+7 = O(g(n))$

5n

for Big-Oh notation,

$f(n) \leq c * g(n)$

where c is a constant

f(n) is given function

g(n) is Result function.

$\Rightarrow 5n+7 \leq c * g(n)$

c = (Coefficient of greatest degree + 1)

$\therefore c = 5+1 = 6$

$\Rightarrow 5n+7 \leq 6 * g(n)$

Let $g(n) = 1$

$5n+7 \leq 6$ $\Rightarrow$ which is false

$g(n) = 2$

$5n+7 \leq 12$ $\Rightarrow$ which is false

$g(n) = n$

$5n+7 \leq 6n$

Let n=1 :

$5+7 \leq 6$ $\Rightarrow$ which is false

Let n=2 :

$10+7 \leq 12$ $\Rightarrow$ which is false

Let n=3 :

$15+7 \leq 18$ $\rightarrow$ which is false

Let n=4 :

$20+7 \leq 24$ $\Rightarrow$ which is false

Let $n=5$:

$25+7 \leq 30$ => which is false

Let $n=6$:

$30+7 \leq 36$ => which is false

Let $\boxed{n=7}$:

$35+7 \leq 42$ => which is True

Let $n=8$

$40+7 \leq 48$ => which is True

⋮

from $n=7$, equation satisfies. where $g(n)=n$.

$\therefore f(n) = O(g(n))$

$f(n) = O(n)$

Here, $c=11$, $c_0=7$

$\therefore f(n) = O(n)$.

Eg: $f(n) = 3n^3 + 2n + 7$

$3n^3 + 2n + 7 = O(g(n))$

for Big-Oh notation,

$f(n) \leq c * g(n)$

where $f(n)$ is given function

c is constant where

c = (Coefficient of higher degree + 1)

$c = 3 + 1 = 4$

$f(n) \leq c * g(n)$

$\rightarrow 3n^3 + 2n + 7 \leq 4 * g(n)$

Let $g(n) = 1$

$3n^3 + 2n + 7 \leq 4 * 1 \Rightarrow$ false

$g(n) = 2$

$3n^3 + 2n + 7 \leq 8 \Rightarrow$ false

⋮

$g(n) = n$

$3n^3 + 2n + 7 \leq 4n$

Let $n = 1$:

$3 + 2 + 7 \leq 4 \Rightarrow$ false

Let $n = 2$:

$24 + 4 + 7 \leq 8 \Rightarrow$ false

Let $n = 3$:

$81 + 6 + 7 \leq 12 \Rightarrow$ false

Let $n = 4$:

$192 + 8 + 7 \leq 16 \Rightarrow$ false

Let $n = 5$:

$375 + 10 + 7 \leq 20 \Rightarrow$ false

⋮

false

Let $g(n) = n^2$

$3n^3 + 2n + 7 \leq 4n^2$

$n = 1$:

$3 + 2 + 7 \leq 4 \Rightarrow$ false

$n = 2$:

$24 + 4 + 7 \leq 16 \Rightarrow$ false

$n = 3$:

$81 + 6 + 21 \leq 36 \Rightarrow$ false

$n = 4$:

$192 + 8 + 7 \leq 64 \Rightarrow$ false

$n = 5$:

$375 + 10 + 7 \leq 100 \Rightarrow$ false

⋮

$\Rightarrow$ false

Let $g(n) = n^3$

$$3n^3 + 2n + 7 \leq 4n^3$$

$n=1$:

$$3 + 2 + 7 \leq 4 \quad \Rightarrow \text{false}$$

$n=2$:

$$24 + 4 + 7 \leq 32 \quad \Rightarrow \text{false}$$

$\boxed{n=3}$

$$81 + 6 + 21 \leq 108 \quad \Rightarrow \text{True}$$

$n=4$:

$$192 + 8 + 7 \leq 256 \quad \Rightarrow \text{True}$$

$n=5$:

$$375 + 10 + 7 \leq 500 \quad \Rightarrow \text{True}$$

from $n=3$, $\therefore$ Equation satisfies where $g(n) = n^3$.

Here, $c_0 = 3$, $c = 4$

$\therefore f(n) = O(g(n))$

$f(n) = O(n^3)$

$\therefore$ Big-oh notation of $f(n) = O(n^3)$

Omega notation (Ω):

- The notation $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete

$$f(n) = \Omega(g(n)) \text{ iff}$$

$$f(n) \geq c * g(n) \; \forall \; c \; \& \; n_0 \; \forall \; n \geq n_0$$

Ex: $f(n) = 3n + 2$.

Sol: $f(n) \geq c * g(n)$

$3n + 2 \geq 2 * g(n)$ $\qquad c = 3 - 1 = 2$

For $g(n) = 1$, i.e

$3n + 2 \geq 2 * 1$

For $n = 1$: $3(1) + 2 \geq 2 * 1$

$5 \geq 2$

True

$n = 2$: $3(2) + 2 \geq 2 * 1$

$8 \geq 2$ True

$n = 3$: $3(3) + 2 \geq 2 * 1$

$11 \geq 2$ True

$\Omega(1)$ is possible, because whatever values we take for n, it satisfies the condition.

Now, for $g(n) = n$, i.e

$3n + 2 \geq 2 * n$

$n = 1$: $3(1) + 2 \geq 2 * 1$

$5 \geq 2$ True

$n = 2$: $3(2) + 2 \geq 2 * 2$

$8 \geq 4$ True

$n = 3$: $3(3) + 2 \geq 2 * 3$

$11 \geq 6$ True

when we keep on substituting 'n' values, we will get true, i.e condition is satisfied

$\therefore \Omega(n)$ is possible.

Now when $g(n) = n^2$, i.e

$3n + 2 \geq 2 * n^2$

$n = 1$: $3(1) + 2 \geq 2 * (1)^2$

$5 \geq 2$ True

$n = 2$: $3(2) + 2 \geq 2 * (2)^2$

$8 \geq 8$ True

$n = 3$: $3(3) + 2 \geq 2 * (3)^2$

$11 \geq 18$ False

$n = 4$: $3(4) + 2 \geq 2 * (4)^2$

Here

$14 \geq 32$ False

As the conditions are not satisfied, $\Omega(n^2)$ is not possible.

$f(n) = 3n + 2$

$\Omega(1)$, $\Omega(n)$ are the possible omega notations.

Ex 2: $f(n) = 10n^2 + 3n + 3$

$f(n) = \Omega(g(n))$

$f(n) \geq c * g(n)$

$10n^2 + 3n + 3 \geq 9 * g(n)$. $c = 10 - 1 = 9$

For $g(n) = 1$

$10n^2 + 3n + 3 \geq 9 * 1$

$n = 1$: $10(1)^2 + 3(1) + 3 \geq 9 * 1$

$16 \geq 9$ True

$n = 2$: $10(2)^2 + 3(2) + 3 \geq 9 * 1$

$49 \geq 9$ True

$\Omega(1)$ is possible

For $g(n) = n$,

$10n^2 + 3n + 3 \geq 9 * n$

$n = 1$: $10(1)^2 + 3(1) + 3 \geq 9 * 1$

$16 \geq 9$ True

$n = 2$: $10(2)^2 + 3(2) + 3 \geq 9 * 2$

$49 \geq 18$ True

$n = 3$: $10(3^2) + 3(3) + 3 \geq 9 * 3$

$102 \geq 27$ True

For any value of n, the condition gets satisfied

$\therefore \Omega(n)$ is possible

For $g(n) = n^2$, i.e

$$10n^2 + 3n + 3 \geq 9 + n^2$$

$n = 1$: $10(1)^2 + 3(1) + 3 \geq 9 + 1^2$

$16 \geq 9$ True

$n = 2$: $10(2^2) + 3(2) + 3 \geq 9 + 2^2$

$49 \geq 36$ True

$n = 3$: $10(3^2) + 3(3) + 3 \geq 9 + 3^2$

$102 \geq 81$ True

Here also, for any value of n, the condition is satisfied.

$\therefore \Omega(n^2)$ is possible

For $f(n) = 10n^2 + 3n + 3$

$\Omega(1)$, $\Omega(n)$ and $\Omega(n^2)$ are the possible omega notations.

Theta (Θ):

$f(n) = \Theta(g(n))$ iff $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

$\} \underbrace{c_1, c_2 \text{ & } n_0}_{+ve} \quad \forall\ n \geq n_0$

Let us take an example 1:

$f(n) = 4n + 3$

here $c_1 = 3$

$c_2 = 5$

$\Rightarrow 3 * g(n) \leq 4n+3 \leq 5 * g(n)$

let us take

$g(n) = 1$

$3 \leq 4n+3 \leq 5$

if $n = 1$ then

$3 \leq 7 \leq 5 \rightarrow$ false

if $n = 2$ then

$3 \leq 11 \leq 5 \rightarrow$ false

⋮

$\therefore \Theta(1)$ is not possible

let us consider

$g(n) = n$

$3 * n \leq 4n + 3 \leq 5 * n$

if $n = 1$ then

$3 \leq 7 \leq 5 \longrightarrow$ false

if $n = 2$ then

$6 \leq 11 \leq 10 \longrightarrow$ false

if $n = 3$ then

$9 \leq 15 \leq 15 \longrightarrow$ True.

$\therefore \Theta(n)$ is possible

let us consider

$g(n) = n^2$

$3 * n^2 \leq 4n + 3 \leq 5n^2$

if $n = 1$ then

$3 \leq 7 \leq 5 \longrightarrow$ false

if $n = 2$ then

$12 \leq 11 \leq 20 \longrightarrow$ false

if $n = 3$ then

$27 \leq 15 \leq 45 \longrightarrow$ false.

$\therefore \Theta(n^2)$ is not possible.

In theta notation only $\Theta(n)$ is possible in this function.

$\therefore \Theta(n)$ is possible notation.

Let us take another example 2:

$f(n) = 10n^2 + 3n + 3$

$c_1 = 9$

$c_2 = 11$

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

$9 * g(n) \leq 10n^2 + 3n + 3 \leq 11 * g(n)$

$g(n) = 1$

$9 \leq 10n^2 + 3n + 3 \leq 11$

if $n = 1$ then

$9 \leq 16 \leq 11 \rightarrow$ false

if $n = 2$ then

$9 \leq 49 \leq 11 \rightarrow$ false

$\therefore \Theta(1)$ is not possible for $g(n) = n$.

$9 * n \leq 10n^2 + 3n + 3 \leq 11 * n$

if $n = 1$ then

$9 \leq 16 \leq 11 \rightarrow$ false

if $n = 2$ then $18 \leq 49 \leq 22 \rightarrow$ false

⋮

$\therefore \Theta(n)$ is not possible

for $g(n) = n^2$

$9 * n^2 \leq 10n^2 + 3n + 3 \leq 11 * n^2$

$n = 1$ $\quad 9 \leq 16 \leq 11 \rightarrow$ false

$n = 2$ $\quad 36 \leq 49 \leq 44 \rightarrow$ false

$n = 3$ $\quad 81 \leq 102 \leq 99 \rightarrow$ false

$n = 4$ $\quad 144 \leq 175 \leq 176 \rightarrow$ True

⋮

$\Theta(n^3)$ is possible

for $g(n) = n^3$

$9 * n^3 \leq 10n^2 + 3n + 3 \leq 11 * n^3$

if $n = 2$ then

$72 \leq 49 \leq 88 \rightarrow$ false

⋮

$\Theta(n^3)$ is not possible

In this function, $\Theta(n^2)$ is the possible notation.

⇒ Find possible Assymptotic Notations for the time Complexity equation

$$f(n) = 3n+2$$

Big Oh:

$$f(n) = O(g(n)) \Rightarrow$$

$$\Rightarrow f(n) \leq c * g(n) \quad c \nexists n_0 \nexists n \geq n_0$$

$$3n+2 \leq 4 * g(n)$$

for $g(n) = 1$

$$3n+2 \leq 4*1$$

if $n=1$ then

$$5 \leq 4 \rightarrow \text{false}$$

if $n=2$ then

$$8 \leq 4 \rightarrow \text{false}$$

$\therefore O(1)$ is not possible

for $g(n) = n$

$$3n+2 \leq 4*n$$

if $n=1$ then

$$5 \leq 4 \rightarrow \text{false}$$

if $n=2$ then

$$8 \leq 8 \rightarrow \text{True}$$

$\therefore O(n)$ is possible

then

$g(n) = n^2$

$3n+2 \leq 1 * n^2$

if $n = 1$ then

$5 \leq 4 \rightarrow$ false

if $n = 2$ then

$8 \leq 16 \longrightarrow$ true

$O(n^2)$ is possible

$\therefore$ possible Big-Oh notations are

$O(n), O(n^2), O(n^3)$ .....

Omega :

$f(n) = \Omega(g(n))$ iff $f(n) \geq c * g(n)$

$c = 2.$

$3n + 2 \geq 2 * g(n)$

$g(n) = 1$

$3n + 2 \geq 2 * 1$

if $n = 1$ then

$5 \geq 2 \longrightarrow$ true

if $n = 2$ then

$8 \geq 2 \longrightarrow$ true.

$\Omega(1)$ is ~~no~~ possible

for $g(n) = n$

$3n+2 \geq 2 * n$

if $n=1$ then

$3+2 \geq 2 \rightarrow$ True

if $n=2$ then

$8 \geq 4 \rightarrow$ True

if $n=3$ then

$11 \geq 6 \rightarrow$ True

⋮

$\Omega(n)$ is possible

for $g(n) = n^2$

$3n+2 \geq 2 * n^2$

if $n=1$ then

$5 \geq 2 \rightarrow$ True

if $n=2$ then

$8 \geq 8 \rightarrow$ True

if $n=3$ then

$11 \geq 18 \rightarrow$ False

⋮

$\therefore \Omega(n^2)$ is not possible

$\therefore$ possible omega notations are $\Omega(1), \Omega(n)$

Theta :

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

$c_1 = 2$

$c_2 = 4$

$$2 * g(n) \leq 3n+2 \leq 4 * g(n)$$

for $g(n) = 1$

$$2 * 1 \leq 3n+2 \leq 4 * 1$$

if $n=1$ then

$2 \leq 5 \leq 4 \rightarrow$ false

if $n=2$ then

$2 \leq 8 \leq 4 \rightarrow$ false

⋮

$\Theta(1)$ is not possible

for $g(n) = 2$

$$2 * n \leq 3n+2 \leq 4 * n$$

if $n=1$ then

$2 \leq 5 \leq 4 \rightarrow$ false

if $n=2$ then

$4 \leq 8 \leq 8 \rightarrow$ True

⋮

$\therefore \Theta(n)$ is possible.

for $g(n) = n^2$ then

$$2 * n^2 \leq 3n+2 \leq 4n^2$$

if $n = 1$: $2 \leq 5 \leq 4$ → false

if $n = 2$ then

$8 \leq 8 \leq 16$ → True

if $n = 3$ then

$18 \leq 11 \leq 36$ → false

⋮

$\Theta(n^2)$ is not possible

$f(n) = 3n+2$

Possible notations:

- → $O(n)$, $O(n^2)$, $O(n^3)$
- → $\Omega(1)$, $\Omega(n)$
- → $\Theta(n)$

1 Probabilistic analysis and randomized algorithms

Consider the problem of hiring an office assistant. We interview candidates on a rolling basis, and at any given point we want to hire the best candidate we've seen so far. If a better candidate comes along, we immediately fire the old one and hire the new one.

```
HIRE-ASSISTANT(n)
1  best = 0          // candidate 0 is a least-qualified dummy candidate
2  for i = 1 to n
3      interview candidate i
4      if candidate i is better than candidate best
5          best = i
6          hire candidate i
```

In this model, there is a cost ci associated with interviewing people, but a much larger cost ch associated with hiring people. The cost of the algorithm is O(cin + chm), where n is the total number of applicants, and m is the number of times we hire a new person.

Exercise: What is the worst-case cost of this algorithm?

Answer: In the worst case scenario, the candidates come in order of increasing quality, and we hire every person that we interview. Then the hiring cost is O(chn), and the total cost is O((ci + ch)n).

1.1 Average case analysis

So far this class has mainly focused on the worst case cost of algorithms. Worst case analysis is very important, but sometimes the typical case is a lot better than the worst case, so algorithms with poor worst-case performance may be useful in practice. (We will use HireAssistant as an example of this.)

Defining the "average" case is tricky, because it requires certain assumptions on how often different types of inputs come up. One possible assumption is that all permutations (i.e. orderings) of candidates are equally likely. But this assumption might not always be true. For example, candidates may be sourced through a recruiter, who sends the strongest candidates first because those are the candidates who are most likely to get him a referral bonus. Without knowing the distribution of inputs, it's hard to perform a rigorous average-case analysis.

If we know the distribution of inputs, we can compute the average-case cost of an algorithm by finding the cost on each input, and then averaging the costs together in accordance with how likely the input is to come up.

Example: Suppose the algorithm's costs are as follows:

Input	Probability that the input happens	Cost of the algorithm when run on that input
A	0.5	1
B	0.3	2
C	0.2	3

Then the average-case cost of the algorithm is just the expected value of the cost, which is $1 \cdot 0.5 + 2 \cdot 0.3 + 3 \cdot 0.2$.

In HireAssistant, we can kind of enforce a distribution on inputs by changing the model a bit. Suppose that instead of the candidates coming in on a rolling basis, the recruiter sends us the list of n candidates in advance, and we get to decide which ones to interview first. Then our algorithm can include a randomization step where we choose randomly which candidate to interview on each day.

This is important when analyzing the algorithm, because now that each ordering of candidates is equally likely, it is easier to compute the expected cost. It's also important when designing the algorithm, because now an evil user can't feed the algorithm a deliberately bad input. No particular input elicits the worst case behavior of the algorithm.

1.2 Average case analysis of HireAssistant

1.2.1 Indicator random variables

An indicator random variable is a variable that indicates whether an event is happening. If A is an event, then the indicator random variable IA is defined as

$$I_A = \begin{cases} 1 & \text{if A occurs} \\ 0 & \text{if A does not occur} \end{cases}$$

Example: Suppose we are flipping a coin n times. We let Xi be the indicator random variable associated with the coin coming up heads on the ith coin flip. So

$$X_i = \begin{cases} 1 & \text{if } i\text{th coin is heads} \\ 0 & \text{if } i\text{th coin is tails} \end{cases}$$

By summing the values of Xi , we can get the total number of heads across the n coin flips. To find the expected number of heads, we first note that

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

by linearity of expectation. Now the computation reduces to computing E[Xi] for a single coin flip, which is

$$\begin{aligned} E[X_i] &= 1 \cdot P(X_i = 1) + 0 \cdot P(X_i = 0) \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2 \end{aligned}$$

So the expected number of heads $\sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} (1/2) = n/2.$

Example: Let A be an event and IA be the indicator random variable for that event. Then $E[I_A] = P(A).$

Proof: There are two possibilities: either A occurs, in which case IA = 1, or A does not occur, in which case IA = 0. So

$$E[I_A] = 1 \cdot P(A) + 0 \cdot P(\text{not } A) = P(A)$$

1.2.2 Analysis of HireAssistant

We are interested in the number of times we hire a new candidate. Let Xi be the indicator random variable that indicates whether candidate i is hired, i.e. let Xi = 1 if candidate i is hired, and 0 otherwise.

Let X = Pn i=1 Xi be the number of times we hire a new candidate. We want to find E[X], which by linearity of expectation, is just Pn i=1 E[Xi].

By the previous example, E[Xi] = P(candidate i is hired), so we just need to find this probability. To do this, we assume that the candidates are interviewed in a random order. (We can enforce this by including a randomization step at the beginning of our algorithm.)

Candidate i is hired when candidate i is better than all of the candidates 1 through i − 1. Now consider only the first i candidates, which must appear in a random order. Any one of them is equally likely to be the best-qualified thus far. So the probability that candidate i is better than candidates 1 through i − 1 is just 1/i.

Therefore E[Xi] = 1/i, and E[X] = Pn i=1 1/i = ln n+O(1) (by equation A.7 in the textbook)

So the expected number of candidates hired is O(ln n), and the expected hiring cost is O(ch ln n).

Amortized Analysis:

This analysis is used for finding average time of an algorithm. Amortized analysis initially used for specific algorithms particularly those including binary tree and union operations. There are three different ways we can platform the Amortized analysis.

*Aggregate Method

*Accounting Method

*Potential Method

Here the only requirement is that the sum of the amortized complexities of all the operations in any sequence of operations must be greater than or equal to their sum of the actual complexities i.e.,

$(\sum(1 <= i <= n)\ \text{amortized}(i)) >= (\sum(1 <= i <= n)\ \text{actual}(i))$

When amortized(i) and actual(i) respectively denote the amortized and actual complexities of ith operation in the sequence of operations.

Month	1	2	3	4	5	6	7	8	9
Actual	50	50	100	50	50	100	50	50	100
Amortized	75	75	75	75	75	75	75	75	75
P()	25	50	25	50	75	50	75	100	75

10	11	12	13	14	15	16
50	50	200	50	50	100	50
75	75	75	75	75	75	75
100	125	0	25	50	25	50

Relative to the actual and amortized costs of each operation in a sequence of n operations, We define the potential function as

$P(i) = \text{amortized}(i) - \text{actual}(i) + P(i-1)$

Generalized form:

$\sum(1 <= i <= n)\ P(i) = \sum(1<=i<=n)\ (\text{amortized}(i) - \text{actual}(i) + P(i-1))$

Aggregate Method:-

In this method we uniformly assign the amortized to the

each operation.

Accounting Method:-

In this method we assign amortized costs to the operators by guessing what assignments will work.

Potential Method:-

Here we start with a potential function which is obtained by using good guess work.

—

Reference:

https://brilliant.org/wiki/amortized-analysis/

DESIGN AND ANALYSIS OF ALGORITHMS

(DAA)

UNIT – 2

Disjoint Sets:

Disjoint Sets

Disjoint Set Operations

Union and Find Algorithms

Connected Components and Bi-Connected Components.

Divide and Conquer:

General method

Applications :

Binary Search

Quick sort

Merge Sort

Strassen's Matrix Multiplication.

Unit - 2

Disjoint set:

A *disjoint-set data structure* is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A *union-find algorithm* is an algorithm that performs two useful operations on such a data structure.

Operations on disjoint sets:

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union*:* Join two subsets into a single subset.

Find :

Find(x) follows the chain of parent pointers from *x* upwards through the tree until an element is reached whose parent is itself. This element is the root of the tree and is the representative member of the set to which *x* belongs, and may be *x* itself.

Path compression, is a way of flattening the structure of the tree whenever *Find* is used on it. Since each element visited on the way to a root is part of the same set, all of these visited elements can be reattached directly to the root. The resulting tree is much flatter, speeding up future operations not only on these elements, but also on those referencing them.

Pseudo code:

```
function Find(x)
        if x.parent != x
          x.parent := Find(x.parent)
 return x.parent
```

Tarjan and Van Leeuwen also developed one-pass *Find* algorithms that are more efficient in practice while retaining the same worst-case complexity.

Union:

Union (x, y) uses *Find* to determine the roots of the trees *x* and *y* belong to. If the roots are distinct, the trees are combined by attaching the root of one to the root of the other. If this is done naively, such as by always making *x* a child of *y*, the height of the trees can grow as to prevent this *union by rank* is used.

Union by rank always attaches the shorter tree to the root of the taller tree. Thus, the resulting tree is no taller than the originals unless they were of equal height, in which case the resulting tree is taller by one node.

To implement *union by rank*, each element is associated with a rank. Initially a set has one element and a rank of zero. If two sets are unioned and have the same rank, the resulting set's rank is one larger; otherwise, if two sets are unioned and have different

ranks, the resulting set's rank is the larger of the two. Ranks are used instead of height or depth because path compression will change the trees' heights over time.

Pseudo code:

```
function Union (x, y)
        xRoot:= Find(x)
       yRoot:= Find(y)
  // x and y are already in the same set
        if xRoot == yRoot
           return
  // x and y are not in same set, so we merge them
        if xRoot.rank < yRoot.rank
              xRoot.parent := yRoot
       else if xRoot.rank > yRoot.rank
              yRoot.parent := xRoot
       else
             xRoot.parent := yRoot
             yRoot.rank   := yRoot.rank + 1
```

Connected Components:

A connected component, of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super graph. For example, the graph shown in the illustration has three components. A vertex with no incident edges is itself a component. A graph that is itself connected has exactly one component, consisting of the whole graph.

It is straightforward to compute the components of a graph in linear time (in terms of the numbers of the vertices and edges of the graph) using either breadth-first search or depth-first search. In either case, a search that begins at some particular vertex v will find the entire component containing v (and no more) before returning. To find all the components of a graph, loop through its vertices, starting a new breadth first or depth first search whenever the loop reaches a vertex that has not already been included in a previously found component

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes

to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

1. Pick a starting node and push all its adjacent nodes into a stack.
2. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
3. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Algorithm:

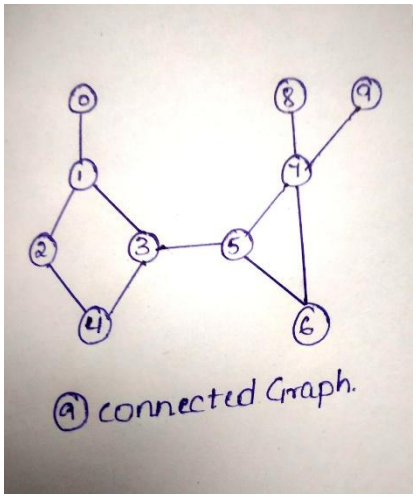
```
DFS-iterative (G, s):                          //Where G is graph and s is source vertex
    let S be stack
    S. push (s)          //Inserting s in stack
    mark s as visited.
    while (S is not empty):
        //Pop a vertex from stack to visit next
        v  =  S.top( )
       S.pop( )
       //Push all the neighbours of v in stack that are not visited
      for all neighbours w of v in Graph G:
          if w is not visited:
                  S.push( w )
                 mark w as visited


  DFS-recursive(G, s):
      mark s as visited
      for all neighbours w of s in Graph G:
          if w is not visited:
              DFS-recursive(G, w)
```

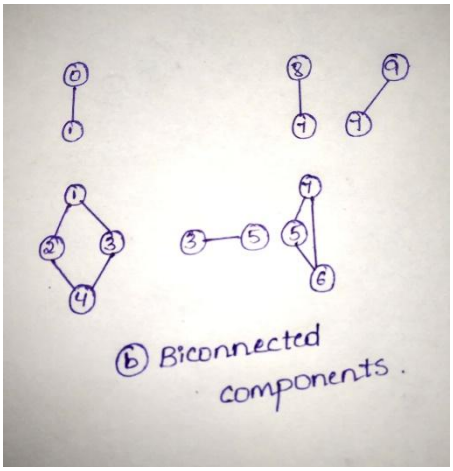
Time complexity is O(V + E), when implemented using the adjacency list

Biconnected Graphs:

Let us assume that G is an undirected connected graph. An articulation point is a vertex v of G such that the deletion of v, together with all edges incident on v, produces a graph, G', that has at least two connected components. For example, the connected graph of Figure(a) has four articulation points, vertices 1,3,5, and 7. A biconnected graph is a connected graph that has no articulation points.

In many graph applications, articulation points are undesirable. For instance, suppose that the graph of Figure (a) represents a communication network. In such graphs, the vertices represent communication stations and the edges represent communication links. Now suppose that one of the stations that is an articulation point fails. The result is a loss of communication not just to and from that single station, but also between certain other pairs of stations. A biconnected component of a connected undirected graph is a maximal biconnected subgraph, H, of G. By maximal, we mean that G contains no other subgraph that is both biconnected and properly contains H. For example, the graph of Figure (a) contains the six biconnected components shown in Figure(b). It is easy to verify that two biconnected components of the same graph have no more than one vertex in common. This means that no edge can be in two or more biconnected components of a graph. Hence, the biconnected components of G partition the edges of G. We can find the biconnected components of a connected undirected graph, G, by using any depth first spanning tree of G.



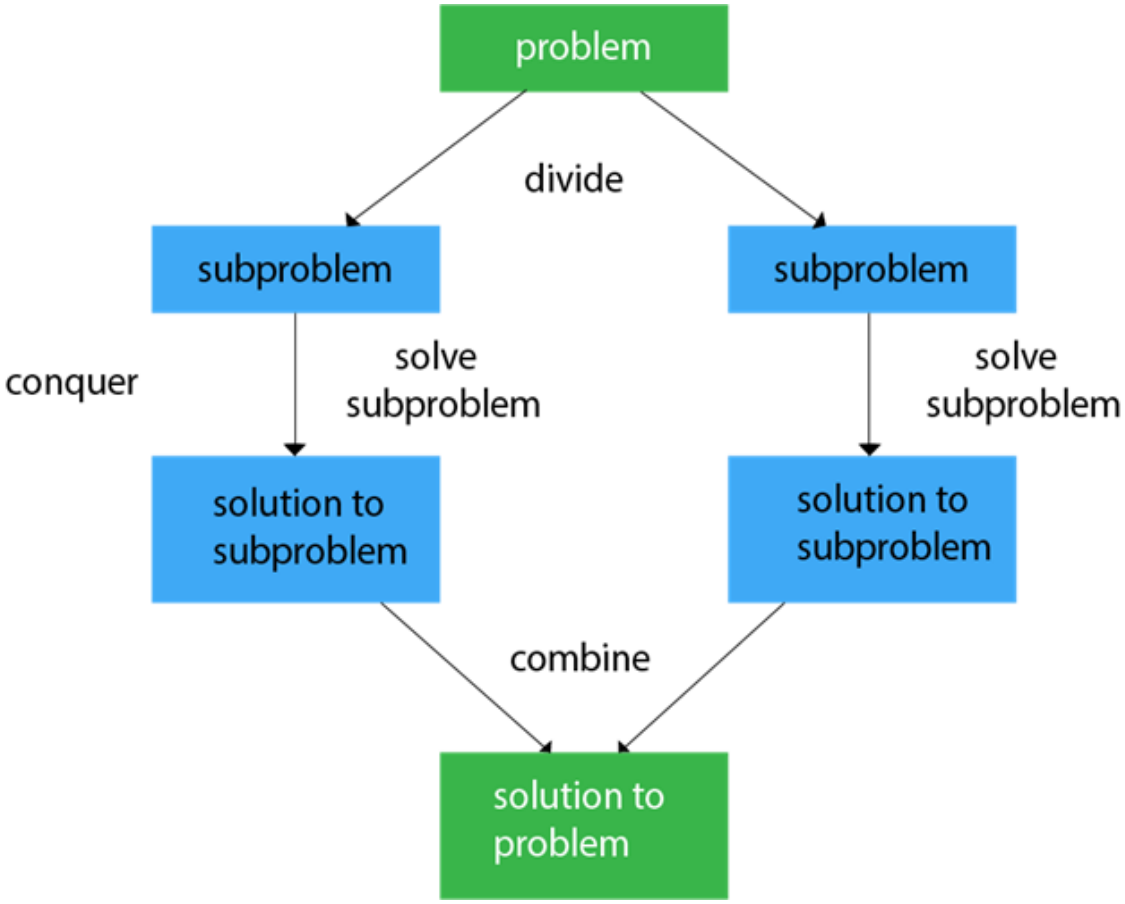


Divide and Conquer Introduction

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

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

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



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

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

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

Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

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

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

BINARY SEARCH:

1) Let a_i , 1<=i<=n be a list of elements that are sorted in non-descending order.

2) Suppose if we have 'n' number of elements then x is element searched.

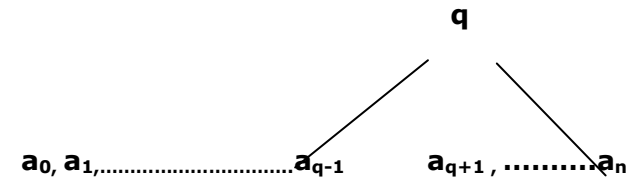
3) If n = 1, small(p) be true then there is no requirement of applying divide and conquer , if n > 1 then it can be divided into new such problems.

4) Pick an index 'q' in the range [i,l] and compare x with a_q then the following 3 possibilities will occur,

a) x = a_q ,then search element is found,

b) x < a_q, then x is searched in left sublist,

c)x > a_q, then x is searched in right sublist,



0	1	2	3	4	5
1	2	3	4	5	6

EXAMPLE:

Consider the list n = 6 where a = { 1,2,3,4,5,6}

X = 4(key element which has to be find)

First find the mid value of the given array

Mid = low + high / 2

Low = first element in the array (*array index*)

High = last element in the array (*array index*)

Mid = 0 + 5 / 2

= 2.5 = 2

Cases:-

1) Compare the key element with a[mid]

X == a[mid] 4 == a[2] 4 == 3(false)

2) Check if the key element is bigger or smaller than mid element

X <= a[mid] X >= a[mid]

4 <= a[2] 4 >= a[2]

4 < 3 4 > 3

As said in the procedure key element in bigger than respective array element so the key element will be in right sub list of the array

Again find the mid for the right sublist

Now low = mid + 1

= 2 + 1 = 3

High will not be changed because we are searching in the right sublist

Mid = (3 + 5) / 2

= 4

4	5	6
3	4	5

a[mid] = a[4]

= 5

Cases:-

1) Compare key element with new mid value

X = a[mid] X <= a[mid]

4 = 5(false) 4 <= 5(true)

The key element is less than mid value so now we have the calculate mid again to find key element

Now low = 3

High value changes now

High = mid - 1

= 4 – 1 = 3

Mid = low + high / 2

= (3 + 3) / 2 = 3

a[mid] = a[3] = 4

cases:-

4 == a[3]

4 == 4(true)

ANALYSIS:-

TIME COMPLEXITY:

T(n) = T(n/2) + 1. It is in the form of T(n) = aT(n/b) + f(n)

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

By using master's theorem

1) f(n) = g(n) => O(f(n)logn)

2) f(n) < g(n) => O(g(n))

3) f(n) > g(n) => O(n)

$g(n) = n^{\log_b a}$

$g(n) = n^{\log_2 1}$

$= n^0$

$= 1$

$f(n) = g(n) => 1 = 1$

Time Complexity T(n) = O(f(n)logn)

= O(logn)

ALGORITHM FOR BINARY SEARCH:

Binary Search (A, x, l, r)

```
Begin:
    If l > r then
        Return -1 (or) not found
    End if
    M = (l + r) / 2
    If A[m] == x then
        Return m;
    Else if x < A[m] then
        Return BinarySearch (A, x, l, m - 1)
    else
        return BinarySearch (A, x, m + 1, r)
    End if
End
```

MERGE SORT:

Merge sort is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort. Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945. It is one of the most popular sorting algorithms and a great way to develop confidence in building recursive algorithms.

Divide and Conquer Strategy:

Using the Divide and Conquer technique, we divide a problem into sub problems. When the solution to each sub problem is ready, we 'combine' the results from the sub problems to solve the main problem.

Suppose we had to sort an array A. A sub problem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

***Divide*:**

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

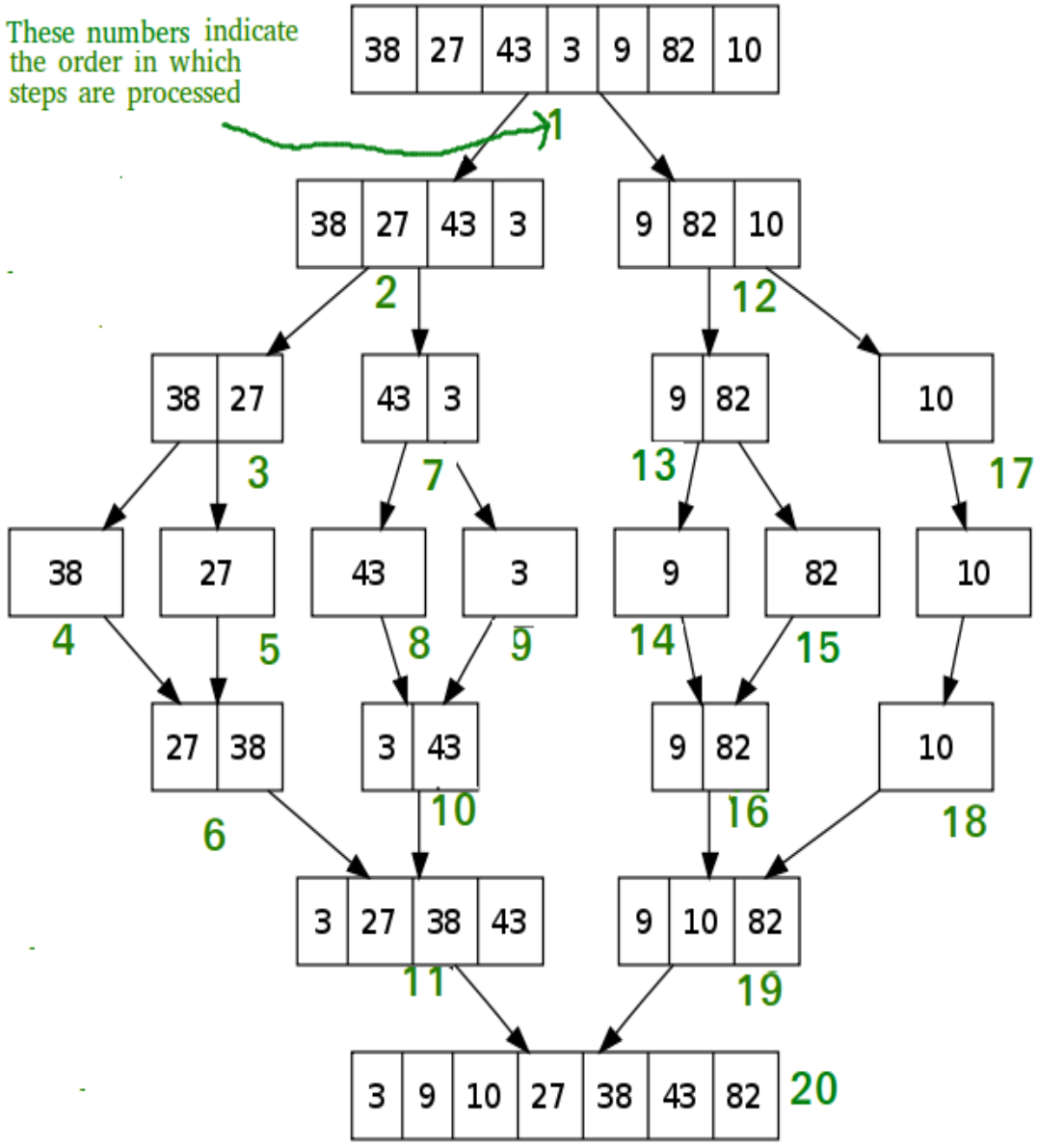
Conquer:

In the conquer step, we try to sort both the sub arrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

***Combine*:**

When the conquer step reaches the base step and we get two sorted sub arrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted sub arrays A[p..q] and A[q+1, r]

EXAMPLE –



The MergeSort Algorithm:

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a sub array of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged

```
MergeSort (A, p, r) {
 If p > r
       return;
   q = (p+r)/2;
   mergeSort (A, p, q)
   mergeSort (A, q+1, r)
   merge (A, p, q, r)
}
void merge (int A[], int p, int q, int r)
{
   /* Create L ← A[p..q] and M ← A[q+1..r] */
   int n1 = q - p + 1;
   int n2 =  r - q;

   int L[n1], M[n2];

   for (i = 0; i < n1; i++)
      L[i] = A[p + i];
   for (j = 0; j < n2; j++)
      M[j] = A[q + 1 + j];

   /* Maintain current index of sub-arrays and main array */
   int i, j, k;
   i = 0;
   j = 0;
   k = p;
   /* Until we reach either end of either L or M, pick larger among elements L and M and place them in the correct position at A[p..r] */
```

```
while (i < n1 && j < n2)
   {
      if (L[i] <= M[j])
      {
         arr[k] = L[i];
         i++;
      }
      else
      {
         arr[k] = M[j];
         j++;
      }
      k++;
   }

    /* When we run out of elements in either L or M, pick up the remaining elements and put in A[p..r] */
   while (i < n1)
   {
      A[k] = L[i];
      i++;
      k++;
   }

   while (j < n2)
   {
      A[k] = M[j];
      j++;
      k++;
   }
}
```

Time Complexity:

Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

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

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is O(nlogn). Time complexity of Merge Sort is O(nlogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

QUICK SORT AND ITS COMPLEXITY:

It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations as it is not difficult to implement. It is a good general-purpose sort and it consumes relatively fewer resources during execution.

Quick Sort is also based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting (major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element (Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as **pivot**

Function: Partition (A, p, r)	***Partition Algorithm:***

x ← A[p]
i ← p-1
j ← r+1
while TRUE do
Repeat j ← j - 1
until A[j] ≤ x
Repeat i← i+1
until A[i] ≥ x
if i < j then
exchange A[i] ↔ A[j]
else
return j
	Partition algorithm rearranges the sub arrays in a place.
	PARTITION (array A, **int** m, **int** n)
	1 x ← A[m]
	2 o ← m
	3 **for** p ← m + 1 to n
	4 **do if** (A[p] < x)
	5 then o ← o + 1
	6 swap A[o] with A[p]
	7 swap A[m] with A[o]
	8 **return** o

Complexity Analysis of Quick Sort:

For an array, in which **partitioning** leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the **pivot**, hence on the right side.

And if keep on getting unbalanced subarrays, then the running time is the worst case, which is $O(n^2)$

Whereas if **partitioning** leads to almost equal subarrays, then the running time is the best, with time complexity as **O (n*log n)**.

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **O (n*log n)**

Average Time Complexity [Big-theta]: **O (n*log n)**

Space Complexity: **O (n*log n)**

As we know now, that if subarrays **partitioning** produced after partitioning are unbalanced, quick sort will take more time to finish. If someone knows that you pick the last index as **pivot** all the time, they can intentionally provide you with array which will result in worst-case running time for quick sort.

To avoid this, you can pick random **pivot** element too. It won't make any difference in the algorithm, as all you need to do is, pick a random element from the array, swap it with element at the last index, make it the **pivot** and carry on with quick sort.

Space required by quick sort is very less, only O(n*log n) additional space is required.

Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

Advantages:

It is in-place since it uses only a small auxiliary stack.

It requires only ***n (log n)*** time to sort **n** items.

It has an extremely short inner loop.

This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

Disadvantages:

It is recursive. Especially, if recursion is not available, the implementation is extremely complicated.

It requires quadratic (i.e., n2) time in the worst-case.

It is fragile, i.e. a simple mistake in the implementation can go unnoticed and cause it to perform badly.

Strassen's Matrix Multiplication:-

Let A and B be two $n \times n$ matrices then the product matrix $C = A \times B$ & $C(i,j) = \sum A(i,k), 1 \le k \le n, B(k,j)$

Ordinary matrix multiplication needs the time complexity $O(n^3)$ strassen tried to reduce the time complexity from that to $O(n^{2.81})$.

Ex:- Suppose if we want to perform multiplication between $n \times n$ matrices we need to divide the $n \times n$ matrices into submatrix until $n=2$.

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

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{22}B_{12} + A_{22}B_{22}$$

To multiply 2x2 matrices, we need 8 multiplications and 4 Additions.

Analysis:-

Recursive relation of above problem is

$$T(n) = \begin{cases} b & n < 2 \\ 8T(n/2) + n^2 & n \ge 2 \end{cases}$$

By using masters Theorem, $T(n) = 8T(n/2) + n^2$

$f(n) = g(n)$

$f(n) < g(n)$

$f(n) > g(n)$

$g(n) = n^{\log_a b} = n^{\log_2 8}$

$= n^{\log_2 2^3}$

$= n^{3\log_2 2}$

$\boxed{g(n) = n^3}$

$n^2 < n^3$

$f(n) < g(n)$

$\Rightarrow O(g(n)) = O(n^3)$

Volkar Strassen discovered a new way to compute matrix multiplication which includes only 7 multiplications.

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = B_{11}(A_{21} + A_{22})$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = (A_{11} + A_{12})B_{22}$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R$

Time Complexity by using this theorem:-

$$T(n) = 7T(n/2) + n^2$$

It is in the form of $T(n) = aT(n/b) + f(n)$

$$a = 7, \quad b = 2, \quad f(n) = n^2$$

$$g(n) = n^{\log_b^a} = n^{\log_2^7}$$

$$= n^{\log_2^{(3+4)}}$$

$$= n^{2.81}$$

$$f(n) < g(n)$$

$$O(g(n)) \Rightarrow O(n^{2.81})$$

Ex:

$$\left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right] \qquad \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{array}\right]$$

A_{11} A_{12} A_{21} A_{22} — B_{11} B_{12} B_{21} B_{22}

$$P = \begin{bmatrix} 12 & 14 \\ 20 & 12 \end{bmatrix}\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 52 & 52 \\ 64 & 64 \end{bmatrix}$$

$$Q = \begin{bmatrix} 20 & 22 \\ 28 & 30 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 42 & 42 \\ 28 & 30 \end{bmatrix}$$

$$R = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}\left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right) = 0$$

$$S = \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix}\left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right) = 0$$

$$T = \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix}\right)\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 10 & 10 \\ 26 & 26 \end{bmatrix}$$

$$U = \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix}\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 32 & 32 \\ 32 & 32 \end{bmatrix}$$

$$V = \left(\begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} - \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix}\right)\left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right) = \begin{bmatrix} -32 & -32 \\ -32 & -32 \end{bmatrix}$$

$$C_{11} = \begin{bmatrix} 10 & 10 \\ 6 & 6 \end{bmatrix}$$

$$C_{12} = \begin{bmatrix} 10 & 10 \\ 26 & 26 \end{bmatrix}$$

$$C_{21} = \begin{bmatrix} 42 & 42 \\ 28 & 30 \end{bmatrix}$$

$$C_{22} = \begin{bmatrix} 42 & 42 \\ 68 & 66 \end{bmatrix}$$

$$= \begin{bmatrix} 10 & 10 & 10 & 10 \\ 6 & 6 & 26 & 26 \\ 42 & 42 & 42 & 42 \\ 28 & 30 & 68 & 66 \end{bmatrix}$$

$$= \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

DESIGN AND ANALYSIS OF ALGORITHMS

(DAA)

UNIT – 3

Greedy Method:

General Method

Applications:

Job Sequencing with Deadlines

Knapsack Problem

Minimum Cost Spanning Trees

Single Source Shortest Path Problem.

SHRI VISHNU ENGINEERING COLLEGE FOR WOMEN : : BHIMAVARAM

(AUTONOMOUS)

UNIT – III

GREEDY METHODOLOGY:-

Every problem have some constraints and objective functions

OBJECTIVE FUNCTION:

It is an attempt to express a business goal (or) a function that is desired to be aximized or minimized.

Any subset that specifies the given constraints is called feasible solution.

FEASIBLE SOLUTION:

If the given problem has 'n' inputs then the subset of inputs satisfies the constraints of particular problem is called feasible solution.

OPTIMAL SOLUTION:

A feasible solution that either maximizes or minimizes the objective function is called optimal solution.

In greedy method, we works in stages. At each stage, we take one input at a time and make a decision, either it gives optimal solution or not.

A decision made in one stage cannot be changed in later stages. i.e, there is no backtracking.

ALGORITHM:

```
Algorithm greedy(a,n) || a[1:n] contains 'n' inputs
{
        Solution:= ¶; || initialize the solution
        For i:=1 to n do
                {
                x :=select(a);
                if feasible(solution,x)then
                solution:=union(solution,x)
                }
return solution
}
```

KNAPSACK PROBLEM:-

We have 'n' objects and a knapsack or bag.Each object has weight 'Wi' and profit 'Pi' and knapsack has capacity 'm'. Objective is filling of Knapsack that maximises the total profit earned.So the problem can be stated as maximise

$$\Sigma\ Pi\ Xi \text{ subject to } \Sigma\ Wi\ Xi <= m$$

1<=i<=n 1<=i<=n and 0<=Xi<=1 , 1 <= i <= n To Compute maximum profit, we take some solution factor i.e Xi.

If object directly placed Xi = 1 (If Enough space is available)

Otherwise Xi=0.

If object does not fit in the knapsack but some amount of space is available, Xi = Remaining Space/Actual Weight of Object.

ALGORITHM :

```
Algorithm knapsack(p,w,n)
        {
        //p[1:n] & w[1:n] are profits and weights of objects such that
        Pi/Wi > P(i+1)/W(i+1) > .....
                {
                for i := 1 to n do
                        x[i] := 0 (u -> capacity of bag)
                        m = u;
                for i := 1 to n do {
                        if(w[i] > m) break;
                                x[i] := 1;
                                m = u - w[i];
                }
        if(i <= n) then
        x[i] = m / w[i] ;
        }
```

****Time Complexity is O(n).***

EXAMPLE:

1. No. of objects n = 3, m = Capacity of knapsack=20

   (p1,p2,p3) = (25,24,15)

   (w1,w2,w3)=(18,15,10)

   Fill the bag with Maximum Profit using Knapsack greedy method

sol: Given, n=3, m=20

Our main aim is to fill the bag with Maximum Profit

Case 1: (Maximum Profit)

First we place Maximum profit object

p1 = 25 w1=18 x1=1

After placing First Object

M = 20 - 18 = 2 p2 = 24 w2 = 15 x2 = 2/15

There is no space available => x3 = 0

Maximize $\Sigma P_i X_i$ ***=> p1x1 + p2x2 + p3x3***

$1<=i<=n$ =>25(1)+24(2/15)+15(0) =>25+3.2 =>**28.2**

Case 2: (Minimum Weight)

Among the above three onjects W3 weight is minimum

w3=10 M=20-10=10 x3=1 [Fitted]

Next object is w2=15

x2 = 10/15 Bag is Full

So,x1=0

Here,Σ Pixi = p1x1 + p2x2 + p3x3

1<=i<=n => 25(0)+24(10/15)+15(1) =>0+16+15 =>**31**

Case 3: (Maximum Profit per unit weight)

p1/w1 = 25/18 = 1.4

p2/w2 = 24/15 = 1.6

p3/w3 = 15/10 = 1.5

We place an item in bag whose profit per weight (p/w) is maximum i.e. p2/w2 is maximum so that x2=1

M= 20-15=5 x3 = 5/10=0.5 x1=0

=>25(0)+24(1)+15(0.5) =>24+7.5 =>**31.5**

Therefore,

The Maximum profitable feasible solution that maximises profit is 31.5

The optimal solution is (0,1,1/2)

Time Complexity is O(n).

MINIMUM COST SPANNING TREE:

It is a connected undirected acyclic graph.let G = (V , E) be an connected graph , then the subgraph t = (V , E') of G is a spanning tree iff 't' is a tree.

WEIGHTED GRAPH: A collection of vertices , edges and also weights on the edges then the graph is said to be weighted graph.

SPANNING TREE: Any tree consisting of all vertices of a graph , then it is called a spanning tree.

MINIMUM COST SPANNING TREE: A spanning tree with minimum cost is called minimum cost spanning tree. In case of complete graph the possible number of spanning trees are n^(n – 2).

To find the minimum cost spaning tree we will use two standard algorithms.

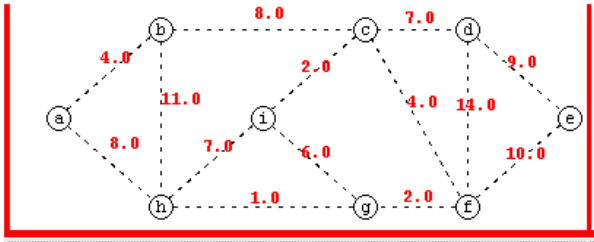
1. prims Algorithm
2. 2. krushkal's Algorithm

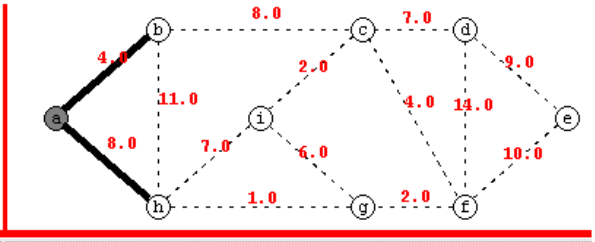
1.PRIMS ALGORITHM:

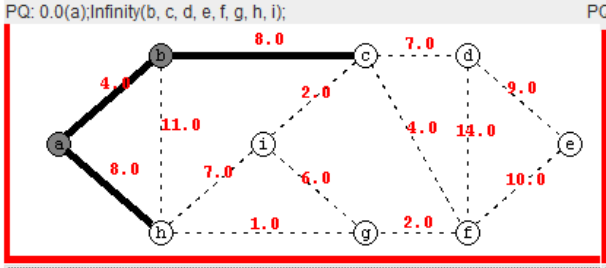
Step 1: Take minimum cost edges from all the edges

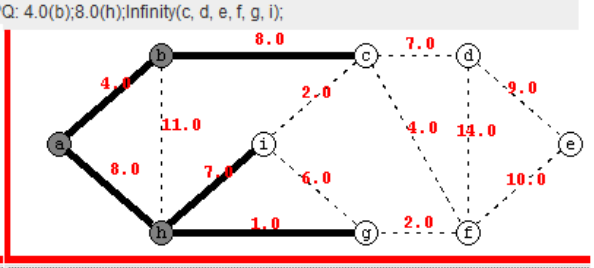
Step 2: Take another edge having minimum weight among those vertices which are adjacent to previous edge.

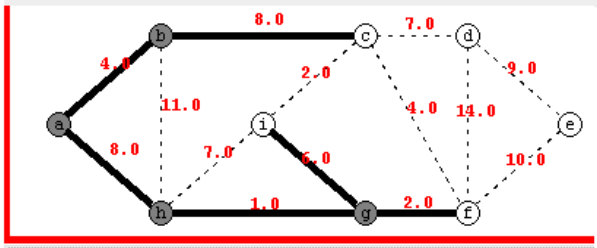
Step 3: Repeat the above process untill spanning tree contains (n – 1) edges.

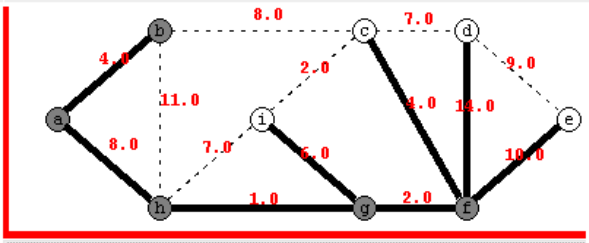


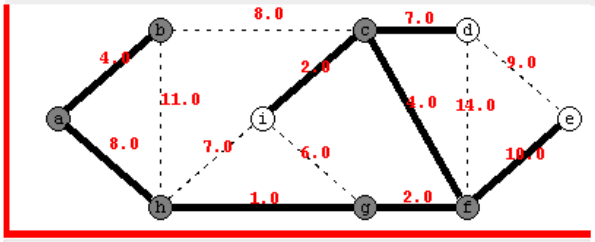


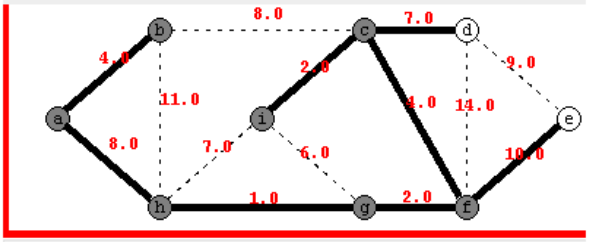


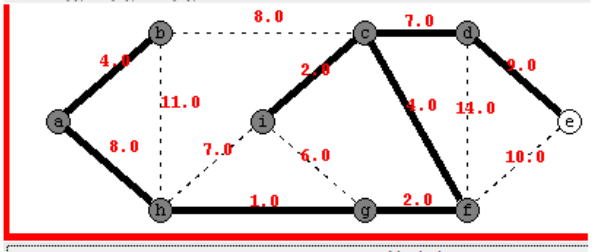


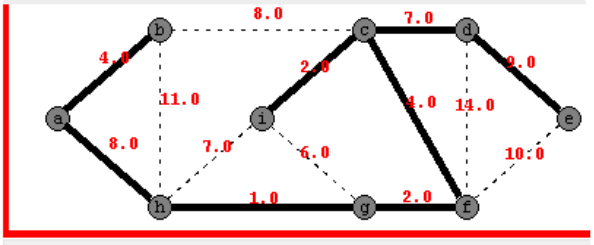












EXAMPLE:

Total cost = 4.0 + 8.0 + 1.0 + 2.0 + 2.0 + 11.0 + 7.0 + 4.0 = 39

ALGORITHM PRIMS (G):

```
//INPUT: A weighted connected graph G = <V , E>
//OUTPUT: T minimum spanning tree of G
{
        V = { V0}
        T = { } // empty graph
        for i = 1 to n – 2
        {
                choose nearest neighbour Vj of V that is adjacent to Vi
                Vi belongs to V and for which edge e i j = (Vi ,Vj ) does not form a
                cycle with members of T.
        V = V union { Vj }
        T = T union { e i j }
        }
return T;
}
```

2.)KRUSHKAL'S ALGORITHM:

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest.

It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest*

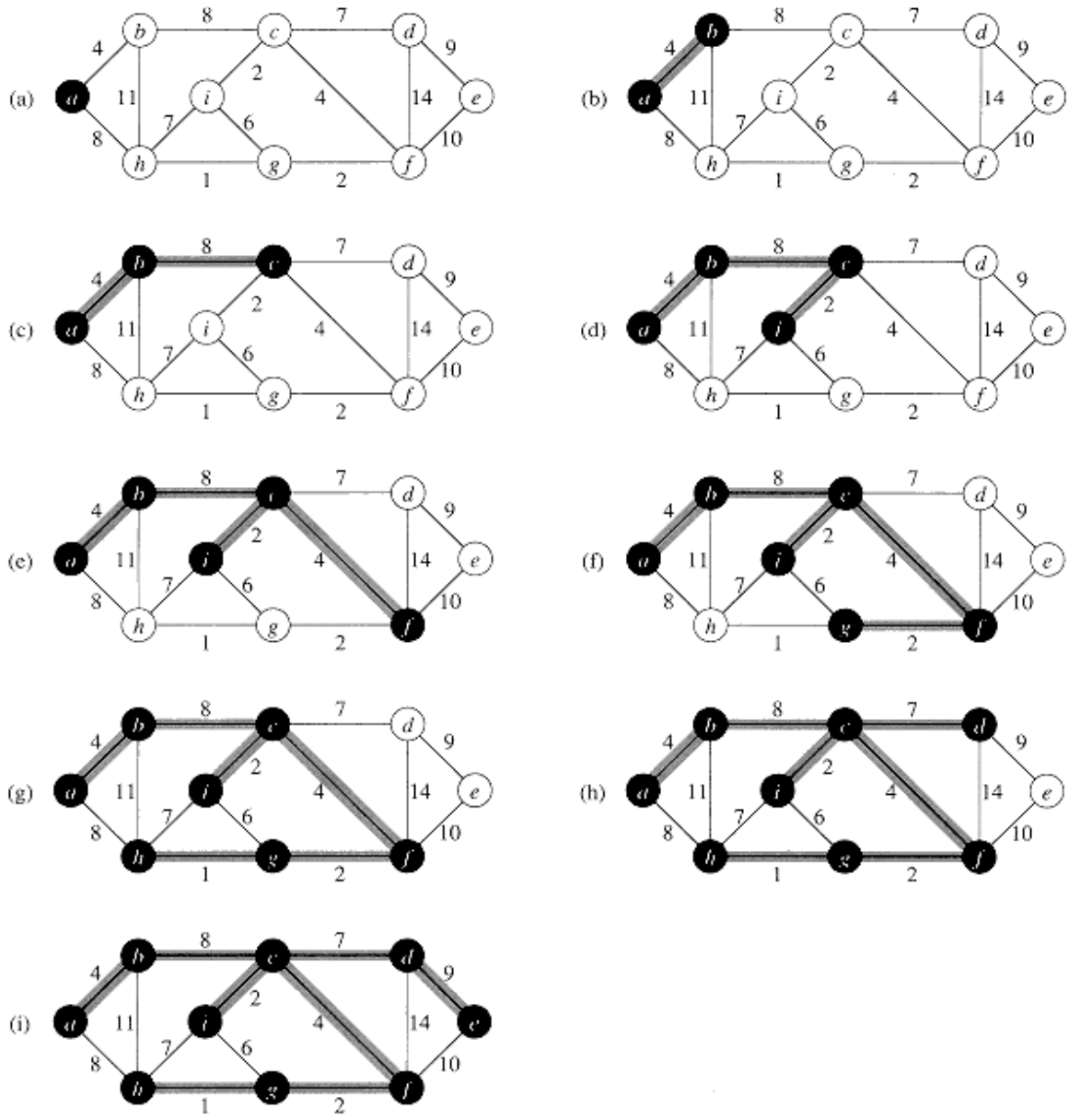
Below are the steps for finding MST using Kruskal’s algorithm:

Step 1: Take minimum cost edges from all the edges

Step 2: Take minimum weight edge among all remaining (n – 1) edges

Step 3: Repeat the above process untill spanning tree contains (n – 1) edges.

EXAMPLE:



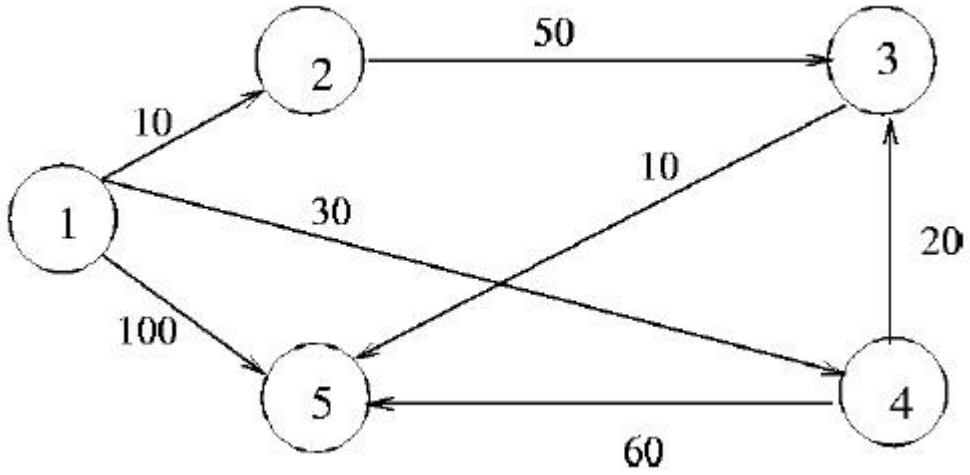
Total cost = 4 + 8 + 7 + 9 + 2 + 4 + 1 + 2 = 37

ALGORITHM:

ALGORITHM PRIMS (G):
//INPUT: A weighted connected graph G = <V , E>
//OUTPUT: T minimum spanning tree of G
{
T = { } // empty graph
for i = 1 to n – 2 do {
e := any edge in G with smallest weight that does not form a cycle when added to T
T := T union { e }
}
return T; }

SINGLE SOURCE SHORTEST PATH PROBLEM:

Graphs can be used to represent distance between states or country with vertices representing cities and edge representing sections of highway. The edges can be assign weights which may be either distances between two cities connected by edge or average time to drive along that section of highway.



Start from source vertex :

->Set s[1] = true;
{1,2} = 10;
{1,3} = ∞;
{1,4} = 30;
{1,5} = 100;
Now shortest distance from vertex 1 is10 i.e{1,2}
	and set[2] = true;
	{1,2,3} = 60;
	{1,2,4} = ∞;
	{1,2,5} =∞;
	Now shortest distance from vertex 2 is 50 i.e{1,2,3}
	and set[3] = true;
		{1,2,3,4} = ∞;
		{1,2,3,5} = 70;
		Now shortest distance from vertex 3 is 10 i.e{1,2,3,5} and selected vertex is 5;
		set s[5] = true;
		destination vertex 5 is achieved through shortest path 1-2-3-5 with path length 70.

ANALYSIS :- single source shortest path O(N^2) times

ALGORITHM:-

Algorithm shortest path(v,cost,dist,n)
{
For i:= 1 to n do
{
s[i] := false;
dist[i] := cost[v,i];
}
s[v] = true;
dist[v] =0.0;
	for num:= 2 to n do{
	// determine n-1 paths from v
	Choose u from among those vertices not in s(not visible)
	Such that dist[u] is minimum
	dist[u] := min{dist[i]}
	s[u]:= true;
	for (each w adjacent to u with s[w] = false) do
	if(dist[w]>dist[u]+cost[u,w]) then
	dist[w] = dist[u]+cost[u,w];
	}
	}

UNIT-III

GREEDY METHOD:-

Every problem has some constraints and objective function If a subset of the problem satisfies the given constraints then it is called feasible solution.

Objective Function:-

It is an attempt to express a business goal in mathematical terms for use in decision analysis & operation research. The objective function which is desired to be maximised or minimized depends on the given problem.

Feasible Solution:-

If the given problem has n inputs then the subset of inputs satisfies the constraints of particular problem, then it is called feasible solution. A problem can have more than one feasible solutions

Optimal Solution:-

A feasible solution that either maximises or minimizes the objective function is called optimal solution.

In greedy method, we works in stages. At each stage, we take one input at a time & make a decision either it gives optimal solution or not a decision made in one stage can't be changed in later stages. i.e, no backtracking is allowed.

Algorithm:-

```
Algorithm Greedy(a,n)
{
    Solution = ϕ;
    for i=1 to n do
    {
        x = select(a);
        if feasible(solution, x) then solution = union(solution, x);
```

```
    }
    return solution;
}
```

APPLICATIONS:

i, Knapsack problem.

We have n objects and a knapsack or bag each object has weight w_i and profit p_i and knapsack has capacity M. The objective is filling of the knapsack that maximises the total profit earned. so that the problem can be state as maximized $\sum_{1 \le i \le n} p_i x_i$, $\sum_{1 \le i \le n} w_i x_i \le M$

To compute maximum profit, we take some solution factor i.e, x_i ranges from $0 \le x_i \le 1$

If object directly placed then $x_i = 1$. otherwise $x_i = 0$.

If object doesn't fit but some amount of space is available then $x_i = \frac{\text{remaining space}}{\text{actual weight of object}}$

Example:

Consider the following instance of Knapsack problem.

$n = 3$, $m = 20$, $(P_1, P_2, P_3) = (25, 24, 15)$, $(w_1, w_2, w_3) = (18, 15, 10)$

Our aim is to fill the bag that maximizes the profit.

CASE-I : Maximum Profit

$P_1 = 25$ $x_1 = 1$

$m = 20 - 18 = 2$

remaining capacity : 2.

$P_2 = 24$.

$x_2 = \frac{2}{15}$

$P_3 = 15$.

$x_3 = 0$ [No space to place the object]

$\sum_{1 \le i \le n} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$

$= 25(1) + 24\left(\frac{2}{15}\right) + 15(0)$

$= 25 + 3.2$

$= 28.2$.

CASE-II : Minimum weight

$m = 20$

$w_3 = 10$.

remaining space = $20 - 10 = 10$.

$x_3 = 1$

$w_2 = 15$

remaining

$x_2 = 10/15 = 2/3$.

$x_1 = 0$ [No available space]

$\sum_{1 \le i \le n} P_i x_i = P_1 x_1 + P_2 x_2 + P_3 x_3$

$= 25(0) + 24(2/3) + 15(1)$

$= 16 + 15$

$= 31$

case-III : Maximum profit per unit weight.

$\frac{P_1}{w_1} = \frac{25}{18} = 1.4$.

$\frac{P_2}{w_2} = \frac{24}{15} = 1.6$.

$\frac{P_3}{w_3} = \frac{15}{10} = 1.5$

$x_2 = 1$

$M = 20 - 15 = 5$

$x_3 = \frac{5}{10} = 0.5$

$x_1 = 0$ [No available space]

$$\sum_{1 \le i \le n} P_i x_i = P_1x_1 + P_2x_2 + P_3x_3.$$

$$= 25(0) + 24(1) + 15(0.5)$$

$$= 24 + 15/2$$

$$= 31.5$$

∴ The maximum profitable & feasible solution that maximizes the profit is. $(0, 1, 0.5)$

The optimal solution = 31.5

```
Algorithm Greedy knapsack (m,n)
{
    for i = 1 to n do
        x[i] = 0.0; // fraction of object
        u = m;
    for i = 1 to n do
    {
        if (w[i] > u) then break;
        x[i] = 1.0;
        u = u - w;
    }
    if (i ≤ n) then x[i] = u/w[i];
}
```

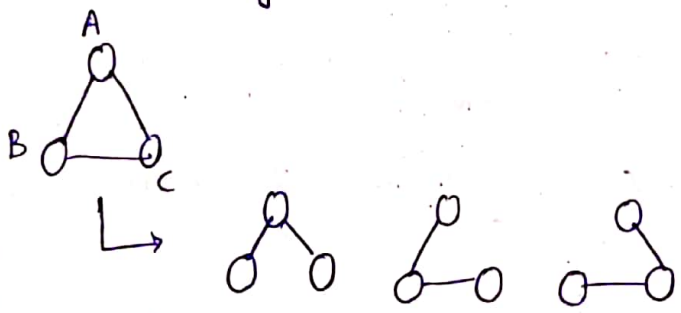
Minimum Cost Spanning Trees:-

Tree:
It is a connected undirected acyclic graph. Let $G = (V, E)$ be an undirected acyclic graph then the sub graph $t = (V, E')$ of G is a spanning tree of G if & only if t is a tree.

Weighted Graph:-
A collection of vertices and edges and weights on those edges then that graph is said to be a weighted graph.

Spanning Tree:-
Any tree consisting of all the vertices of a graph is called a spanning tree. A spanning tree with minimum cost i.e selecting edges that would minimize the total cost of the spanning tree, then it is called minimum cost spanning Tree. In case of a complete graph, the possible no.of spanning trees are n^{n-2}.

To find the Minimum cost spanning tree we will use 2 standard algorithms.

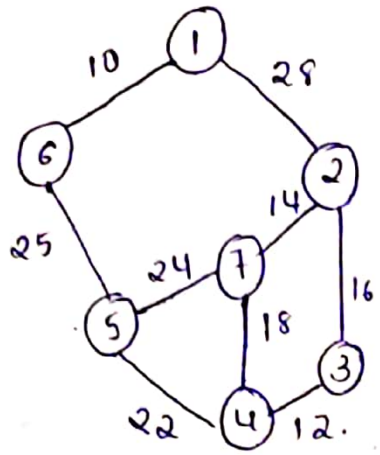


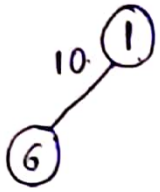
$n^{n-2} = 3^{3-2} = 3^1 = 3$

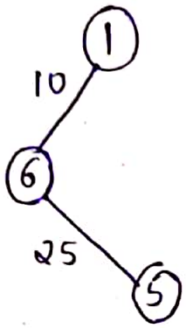
i. Prims
ii. KrushKal's

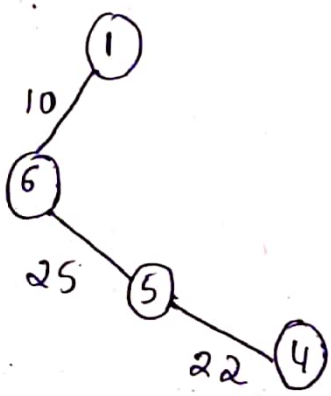
PRIMS ALGORITHM:-
Step 1: Take minimum cost edge from all the edges
Step 2: Take another edge having minimum weight among those vertices which are adjacent to the previous edge.
Step 3: Repeat the above process until spanning tree contains n-1 edges.

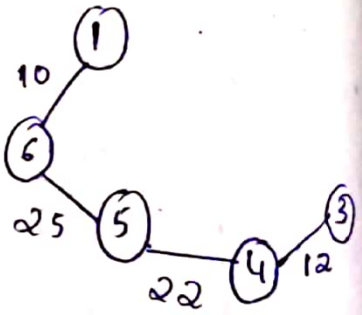
Ex:

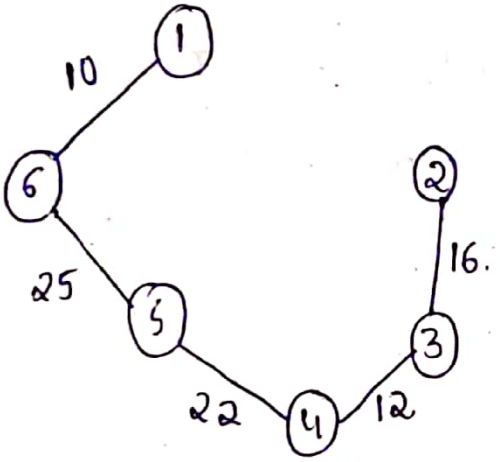


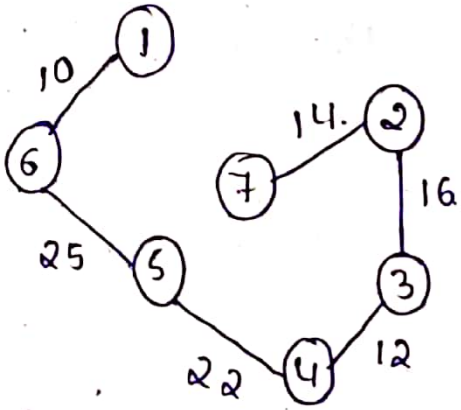












cost = 10 + 25 + 22 + 12 + 16 + 14

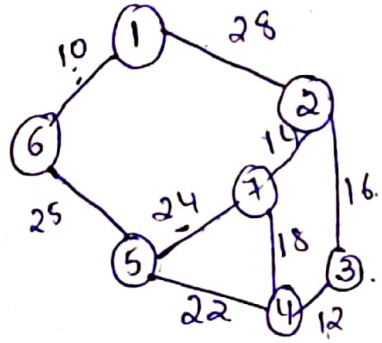
= 99.

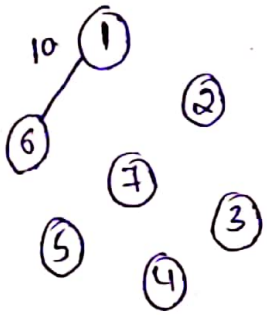
Krushkals Algorithm:

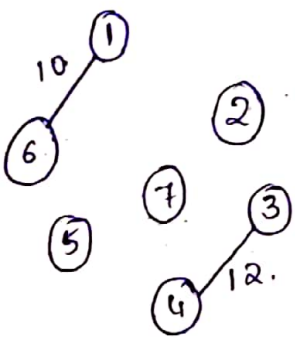
step 1: Take minimum weight edge from all edges.

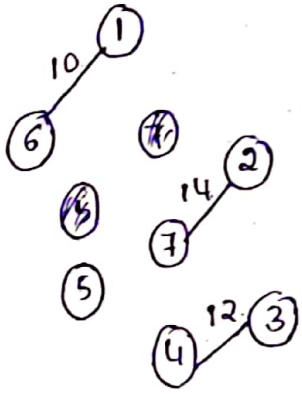
step 2: Take minimum weight edge among all remaining n-1 edges.

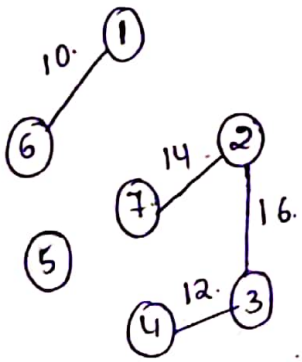
step 3: Repeat the process until spanning tree contains n-1 edges.

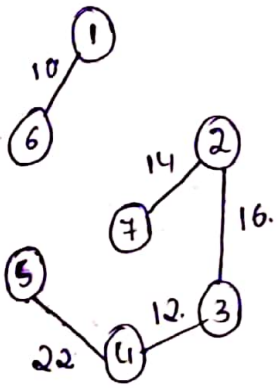


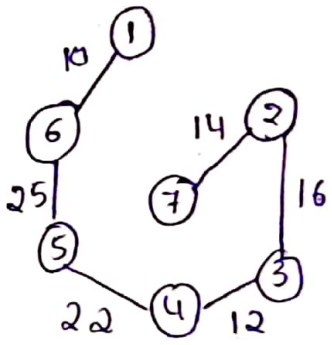












Cost = 10+25+22+12+16+14
= 99.

Job sequencing with deadlines:-

We have a set of n jobs each has an integer dead line $d_i \geq 0$ and profit $P_i > 0$. For any job i the profit P_i is earned if and only if the job is completed by its deadline. To complete a job, one has to process the job on a machine for 1 unit of time. The feasible solution is subset of jobs that are completed in its deadline. The profit is sum of profits of jobs in that subset. An optimal solution is a feasible solution with maximum profit. For finding feasible solution, we have to follow some rules.

i. Each job takes 1 unit of time

ii. If job completed before its deadline, then the profit is earned otherwise no profit.

iii. Maximum time limit for completing the process is

$$t = \max \{d_i\}.$$

Ex:-

Let $n=4$, $(P_1 : P_4) = (100, 10, 15, 27)$

$(d_1 : d_4) = (2, 1, 2, 1)$

$t = \max \{d_i\}$

$t = \max (2,1,2,1)$

$t = 2$

Feasible solutions:

$(1,2) \longrightarrow P_1 = 100, d_1 = 2$

$P_2 = 10, d_2 = 10$

$P_1 \rightarrow P_2$ not possible

$P_2 \rightarrow P_1$ possible profit $= P_1 + P_2 = 100 + 10 = 110$

$(1,3) \longrightarrow P_1 = 100 \quad d_1 = 2$

$P_3 = 15 \quad d_3 = 2$

$P_1 \rightarrow P_3$ possible profit $= P_1 + P_3 = 100 + 15 = 115$

$P_3 \rightarrow P_1$ possible profit $= P_3 + P_1 = 15 + 100 = 115$

(1,4) $P_1 = 100$ $d_1 = 2$

$P_4 = 27$ $d = 1$ (1,4) $P_1 = 100$ $d_1 = 2$

$P_1 \rightarrow P_4$ Not possible

$P_4 \rightarrow P_1$ Possible profit = $P_4 + P_1 = 27 + 100 = 127$

(2,3) $P_2 = 10$ $d_2 = 1$

$P_3 = 15$ $d_3 = 2$

$P_2 \rightarrow P_3$. Possible profit = $P_2 + P_3 = 10 + 15 = 25$

$P_3 \rightarrow P_2$ Not possible

(2,4) $P_2 = 10$ $d_2 = 1$

$P_4 = 27$ $d_4 = 1$

$P_2 \rightarrow P_4$. Not possible

$P_4 \rightarrow P_2$ Not possible

(3,4) $P_3 = 15$ $d_3 = 2$

$P_4 = 27$ $d_4 = 1$

$P_3 \rightarrow P_4$ Not possible

$P_4 \rightarrow P_3$ Possible profit = $P_4 + P_3 = 15 + 27 = 42$

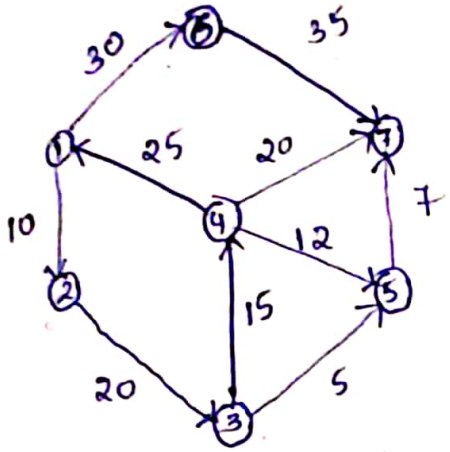
Optimal solution:-

$P_4 \rightarrow P_1$ profit = 127

Note:- shortest job must be done first

SINGLE SOURCE SHORTEST PATH:-

Graphs can be used to represent distance b/w cities or states. Each graph contains vertices & edges where vertices represent cities & edges represent section of Highways The edges can be assigned with weights which may be either distance b/w 2 cities or avg time to drive along that section of highway.



set s[i] = false

s[1] = true
{1,2} = 10
{1,3} = ∞
{1,4} = ∞
{1,5} = ∞
{1,6} = 30
{1,7} = ∞

s[2] = true.
{1,2,3} = 30
{1,2,4} = ∞
{1,2,5} = ∞
{1,2,6} = ∞
{1,2,7} = ∞

s[3] = true
{1,2,3,4} = 45
{1,2,3,5} = 35
{1,2,3,6} = ∞
{1,2,3,7} = ∞

s[5] = true.
{1,2,3,5,4} = ∞
{1,2,3,5,6} = ∞
{1,2,3,5,7} = 42

∴ The shortest path is {1,2,3,5,7} = 42.
Destination vertex 7 is obtained through shortest path {1,2,3,5,7} with path length 42. The time complexity is $O(n^2)$

DESIGN AND ANALYSIS OF ALGORITHMS

(DAA)

UNIT – 4

Dynamic Programming:

General Method

Applications:

Matrix Chain Multiplication

0/1 Knapsack Problem

All Pairs Shortest Path Problem

Travelling Sales Person Problem

Reliability Design

SHRI VISHNU ENGINEERING COLLEGE FOR WOMEN : : BHIMAVARAM

(AUTONOMOUS)

Dynamic programming

Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping sub instances.

Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems .

Main idea:

- set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
- solve smaller instances once
- record solutions in a table
- extract solution to the initial instance from that table

- Applicable when sub problems are **not** independent
  - Sub problems share sub problems

E.g.: Combinations:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$\binom{n}{1} = 1 \qquad \binom{n}{n} = 1$$

A divide and conquer approach would repeatedly solve the common sub problems

Dynamic programming solves every sub problem just once and stores the answer in a table

- Used for **optimization problems**
  - A set of choices must be made to get an optimal solution
  - Find a solution with the optimal value (minimum or maximum)
  - There may be many solutions that lead to an optimal value
  - Our goal: **find an optimal solution**

Dynamic Programming Algorithm

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information (not always necessary)

- An algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decision.
- Dynamic Programming algorithm store results, or solutions, for small subproblems and looks them up, rather than recomputing them, when it needs later to solve larger subproblems
- Typically applied to optimiation problems

Principle of Optimalilty

- An optimal sequence of decisions has the property that whatever the initial state an decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.
- Essentially, this principles states that the optimal solution for a larger subproblem contains an optimal solution for a smaller subproblem.

Dynamic Programming VS. Greedy Method

Greedy Method

- only one decision sequence is ever genertated.

Dynamic Programming

- Many decision sequences may be genertated.

Dynamic Programming VS. Divide-and Conquer

Divide-and-Conquer

- partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem

Dynamic Programming

- applicable when the subproblems are not independent, that is, when subproblems share subsubproblems.
- Solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered

0/1 Knapsack:

- We have n items each with an associated weight and value (profit). The objective is to fill the knapsack with items such that we have a maximum profit without crossing the weight limit of the knapsack.
- Since this is a 0/1 knapsack problem, we can either take an entire item or reject it completely.
- To solve this, we have 2 methods:
  1) Set Method
  2) Table Method

*Table Method (or) Tabulation:

Problem Statement:

The maximum weight the knapsack can hold is m = 11. There are 5 items to choose from. The weights and profits are

$P_i = \{1, 6, 18, 22, 28\}$

$w_i = \{1, 2, 5, 6, 7\}$

Tabulation, the name itself indicates we need to create a table with 5 rows (no. of objects) including 0th object 6 rows and with 12 columns from 0 to 11 because Here 11 is the maximum capacity of the knapsack and 0 is the minimum capacity of the knapsack.

→ initially, 0th object means no weight hence place all zero's in both 0th row & 0th column as from given table

Objects	01	02	03	04	05
weights	1	2	5	6	7
profits	1	6	18	22	28

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2	0											
3	0											
4	0											
5	0											

for 1st object,

	wt	pf
01	1	1

at weight = 1 and 01 = 1 we need to place value 1 at wt = 1

0 1 — Sequentially increasing

remaining values are ≥ 1, Hence we need to place 1 at every cell at object 1

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0											
3	0											
4	0											
5	0											

For object 2

objects	01	02	03	04	05
weights	1	2	5	6	7
profits	1	6	18	22	28

	wt	pf
01	1	1
02	2	6

For that, at 02, wt = 2 we need to place valu 6

and at wt = 1, we need to place previous value

now,

	wt	pf
01	1	1
02	2	6

wt	pf
1+2	1+6
3	7

at wt = 3, we need to place value 7

Hence, we get at 02

0 1 6 7 — 7 = 7

Sequentially increasing manner there we need to place 7.

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0											
4	0											
5	0											

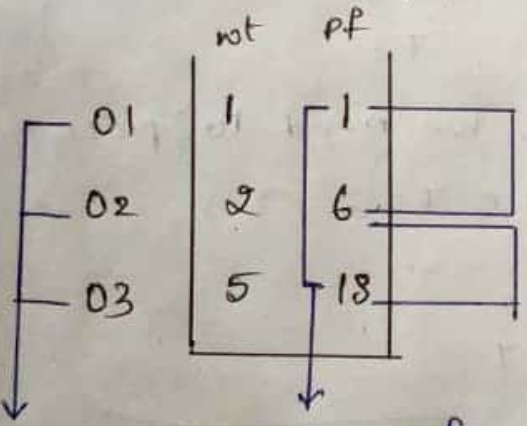
for object 3,

objects :	01	02	03	04	05
weights :	1	2	5	6	7
profits :	1	6	18	22	28

	wt	pf
01	1	1
02	2	6
03	5	18

at wt = 5, we need to place 18

and upto 5th weight, we need to place previous values, now coming to all possible combinations



wt: 1+2 = 3
pf: 1+6 = 7 → already placed

wt: 2+5 = 7
pf: 6+18 = 24 → at wt 7 profit is 24

at wt: 1+2+5
pf: 1+6+18

wt: 1+5 = 6
pf: 1+18 = 19

8 25 at weight 6, profit = 19

at weight 8,
profit 25

place 25

0 1 6 7 7 18 19 24 25

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0											
5	0											

In the same manner, we need to find values of entire table

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

Formula:

We have formula also to calculate profits with respective their weights of objects

$$v[i, w] = \max\{v[i-1, w],\ v[i-1, w - w[i]] + p[i]\}$$

(i: object, w: weight)

eg: $v[5, 1] = \max\{v[4, 1],\ v[4, 1-7] + 28\}$

$= \max\{v[4, 1],\ v[4, -6] + 28\}$

$= 1$

$v[5,11] = \max\{v[4,11], v[4,11-7]+28\}$

$= \max\{v[4,11], v[4,4]+28\}$

$= \max\{40, 7+28\}$

$= 40$

calculating $x1, x2, x3, x4, x5$:

→ Max value = 40 present in 4th row first so include 4th object $x5 = 0, x4 = 1$

→ subtract 40 & corresponding profit i.e $22 = 18$

→ 18 is initially present in 3rd row so include 3rd object $x3 = 1$

→ subtract 18 with corresponding profit i.e $18 = 0$

→ since we can't take any more object because profit is 0, so we don't select 2nd and 1st objects

$x2 = 0, x1 = 1$

→ so, finally objects selected are

$\{x1, x2, x3, x4, x5\}$

$\{0, 0, 1, 1, 0\}$

means

wt		pf
5+6		18+22
11	→	40

2. Set method

Set method is used to find all possibilities and become the best solution.

Problem Statement :

1) Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

Sol:-

$m = 8$

$n = 4$

$P = \{1, 2, 5, 6\}$

$W = \{2, 3, 4, 5\}$

P- profit, W- weight

Let us start with set zero, (S^0)

$S^0 = \{(0, 0)\}$ in this we say no profit & no weight.

Consider 1st object,

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

Now prepare set one, merge the above two S^0 and S_1^0

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

consider 2nd object, i.e, S_1^1 add (2, 3) to these order pairs

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

this is 2nd set for the second object.

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

prepare S_1^2 by considering (5,4) for 3rd object.

$$S_1^2 = \{(5,4)(6,6)(7,7)(8,9)\}$$

here profit is 8 and the weight is 9 it is exceeding the capacity of bag so, exclude it.

merge all these then we get.

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

here profit is increased but weight is decreased

we have to discard any one. so we will discard any one with smaller profit.

i.e, (3,5).

Consider 4th object (6,5),

$$S_1^3 = \{(6,5)(7,7)(8,8)(11,9)(12,11)(13,12)\}$$

These 3 are exceeding the capacity of bag so, exclude it.

merge these order pairs shown above.

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

here the profit is remaining same but weight is reduced.

so, according to dominance rule we should discard any one with smaller profit i.e, (6,6).

here the making order pair is (8,8)

① $(8,8) \in S^4$

but $(8,8) \notin S^3$ $\therefore x_4 = 1$

(8,8) → subract it from 4th object

$(8-6,\ 8-5) = (2,3)$

② check whether $(2, 3)$ is in S^3. so,

$(2, 3) \in S^3$

$(2, 3) \in S^2$ this is not because of third object

therefore $x_3 = 0$.

③ $(2, 3) \in S^2$ $\therefore x_2 = 1$.

but $(2, 3) \in S^2$

Subract, $(2-2, 3-3) = (0, 0)$

④ $(0, 0) \in S^1$

and also, $x_1 = 0$

$(0, 0) \in S^0$

From bottom if we check $(0, 1, 0, 1)$

So, the the answer is $(0, 1, 0, 1)$

$P = \{1, 2, 5, 6\}$

$W = \{2, 3, 4, 5\}$

$\{0, 1, 0, 1\}$ //

Matrix chain Multiplication:

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C and D, we would have

$$(ABC)D = (AB)(CD) = A(BCD) = \ldots$$

Matrix chain multiplication is an optimization problem that to find the most efficient way to multiply given sequence of matrices. The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved. The number of multiplications that needed varies based on the paranthesization.

$$c[i,j] = \min_{i \le k < j} \{ c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j \}$$

If given 4 matrices A_1, A_2, A_3 and A_4 the combination of paranthesization is equal to the catalog number

$$\text{catalog number} = \frac{{}^{2n}C_n}{n+1}$$

Here n = number of matrices − 1

In the above $A_1 \times A_2 \times A_3 \times A_4$ $n = 4-1 = 3$

$$\text{catalog number} = \frac{{}^{2\times3}C_3}{3+1} = \frac{{}^{6}C_3}{4}$$

$$= \frac{\frac{6\times5\times4}{3\times2}}{4} = 5$$

Ex: 1. $A_1((A_2 \times A_3) \times A_4)$

2. $A_1(A_2 \times (A_3 \times A_4))$

3. $(A_1 \times A_2) \times (A_3 \times A_4)$

4. $((A_1 \times A_2) \times A_3) \times A_4$

5. $(A_1 \times (A_2 \times A_3)) \times A_4$.

Ex: Calculate cost of each sub matrix and stored them in a table. and find minimum of $c[i,j]$ for A_1, A_2, A_3 and A_4 of size 4×10, 10×3, 3×12 and 12×20.

Sol:

All the diagnals cost in the table = 0.

So, $c[1,1]$, $c[2,2]$, $c[3,3]$, $c[4,4] = 0$.

c	1	2	3	4
1	0			
2		0		
3			0	
4				0

(i ↓, j →)

A_1	A_2	A_3	A_4
4×10	10×3	3×12	12×20
d_0 d_1	d_1 d_2	d_2 d_3	d_3 d_4

$$c[1,2] = \min_{1\le k<2}\{c[1,1] + c[2,2] + d_0 \times d_1 \times d_2\}$$

$$= 0+0+4\times10\times3$$

$$= 120.$$

We need another table to store the value of k so that the cost is minimum.

cost table

C	1	2	3	4
1	0	120		
2		0		
3			0	
4				0

K	1	2	3	4
1		1		
2				
3				
4				

$c[2,3] = \min_{2 \le k < 3} \{ k=2 \{ c[2,2] + c[3,3] + d_1 \times d_2 \times d_3 \} \}$

$= 0 + 0 + 10 \times 3 \times 12$

$= 360.$

C	1	2	3	4
1	0	120		
2		0	360	
3			0	
4				0

K	1	2	3	4
1		1		
2			2	
3				
4				

$c[3,4] = \min_{3 \le k < 4} \{ k=3 \{ c[3,3] + c[4,4] + d_2 \times d_3 \times d_4 \} \}$

$= 0 + 0 + 3 \times 12 \times 20$

$= 720$

C	1	2	3	4
1	0	120		
2		0	360	
3			0	720
4				0

K	1	2	3	4
1		1		
2			2	
3				3
4				

$$c[1,3] = \min_{1 \le k < 3} \begin{cases} k=1 & c[1,1] + c[2,3] + d_0 \times d_1 \times d_3 \\ k=2 & c[1,2] + c[3,3] + d_0 \times d_2 \times d_3 \end{cases}$$

$$= \min \begin{cases} 0 + 360 + 4 \times 10 \times 12 \\ 120 + 0 + 4 \times 3 \times 12 \end{cases}$$

$$= \min(840, 264)$$

$$= 264.$$

c	1	2	3	4
1	0	120	264	
2		0	360	
3			0	720
4				0

(i ↓, j →)

k	1	2	3	4
1		1	2	
2			2	
3				3
4				

(i ↓, j →)

$$c[2,4] = \min_{2 \le k < 4} \begin{cases} k=2 & c[2,2] + c[3,4] + d_1 \times d_2 \times d_4 \\ k=3 & c[2,3] + c[4,4] + d_1 \times d_3 \times d_4 \end{cases}$$

$$= \min \begin{cases} 0 + 720 + 10 \times 3 \times 20 \\ 360 + 0 + 10 \times 12 \times 20 \end{cases}$$

$$= \min(1320, 2760)$$

$$= 1320.$$

c	1	2	3	4
1	0	120	264	
2		0	360	1320
3			0	720
4				0

	1	2	
		2	2
			3

$$c[1,4] = \min_{1 \le k < 4} \begin{cases} k=1 & c[1,1] + c[2,4] + d_0 \times d_1 \times d_4 \\ k=2 & c[1,2] + c[3,4] + d_0 \times d_2 \times d_4 \\ k=3 & c[1,3] + c[4,4] + d_0 \times d_3 \times d_4 \end{cases}$$

$$= \min \begin{cases} 0 + 1320 + 4 \times 10 \times 20 \\ 1200 + 720 + 4 \times 3 \times 20 \\ 264 + 0 + 4 \times 12 \times 20 \end{cases}$$

$$= \min (2120 , 1080 , 1224)$$

$$= 1080 .$$

c	1	2	3	4
1	0	120	264	1080
2		0	360	1320
3			0	720
4				0

(i ↓, j →)

k	1	2	3	4
1		1	2	2
2			2	2
3				3
4				

(i ↓, j →)

Paranthesization:

$A_1 \quad A_2 \quad A_3 \quad A_4$.

$k[1,4] = 2$, so

$(A_1 \times A_2) \times (A_3 \times A_4)$

$k[1,2] = 1$ and $k[3,4] = 3$.

$$\boxed{((A_1 \times A_2) \times (A_3 \times A_4))}$$

Travelling Salesman Problem

Introduction:- Travelling sales man problem can be solved by using 3 techniques.

1) Brute force Algorithm.
2) Dyanamic Programming
3) Branch and bound.

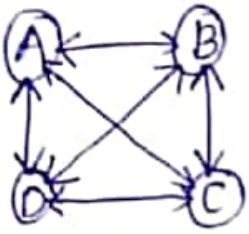
Now, in this we will solve the problem using Dyanamic Programming method.

Travelling Salesman Problem (TSP):- Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Difference b/w Hamiltonean cycle and TSP:-

The Hamiltonean cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamilton tour exists in TSP because it's a complete graph and infact many such tours exists and the problem is to find minimum weight hamilton cycle.

Let us take an example and the given adjacent matrix is



Adjacency matrix

	A	B	C	D
A	0	16	11	6
B	8	0	13	16
C	4	7	0	9
D	5	12	2	0

Let the given set of variables (vertices) be $\{A, B, C, D\}$.

Let us consider 'A' as starting and ending point of output for every other vertex j other than starting vertex, We find the minimum cost path with 'A' as starting point, j as ending point and all vertices appearing exactly once.

Let the cost of this path be cost (j) the cost of corresponding cycle would be cost(j) + dist(j, A) where dist(j, A) is the distance from j to A. Finally we return the minimum of all (cost(j) + dist(j, A)) values.

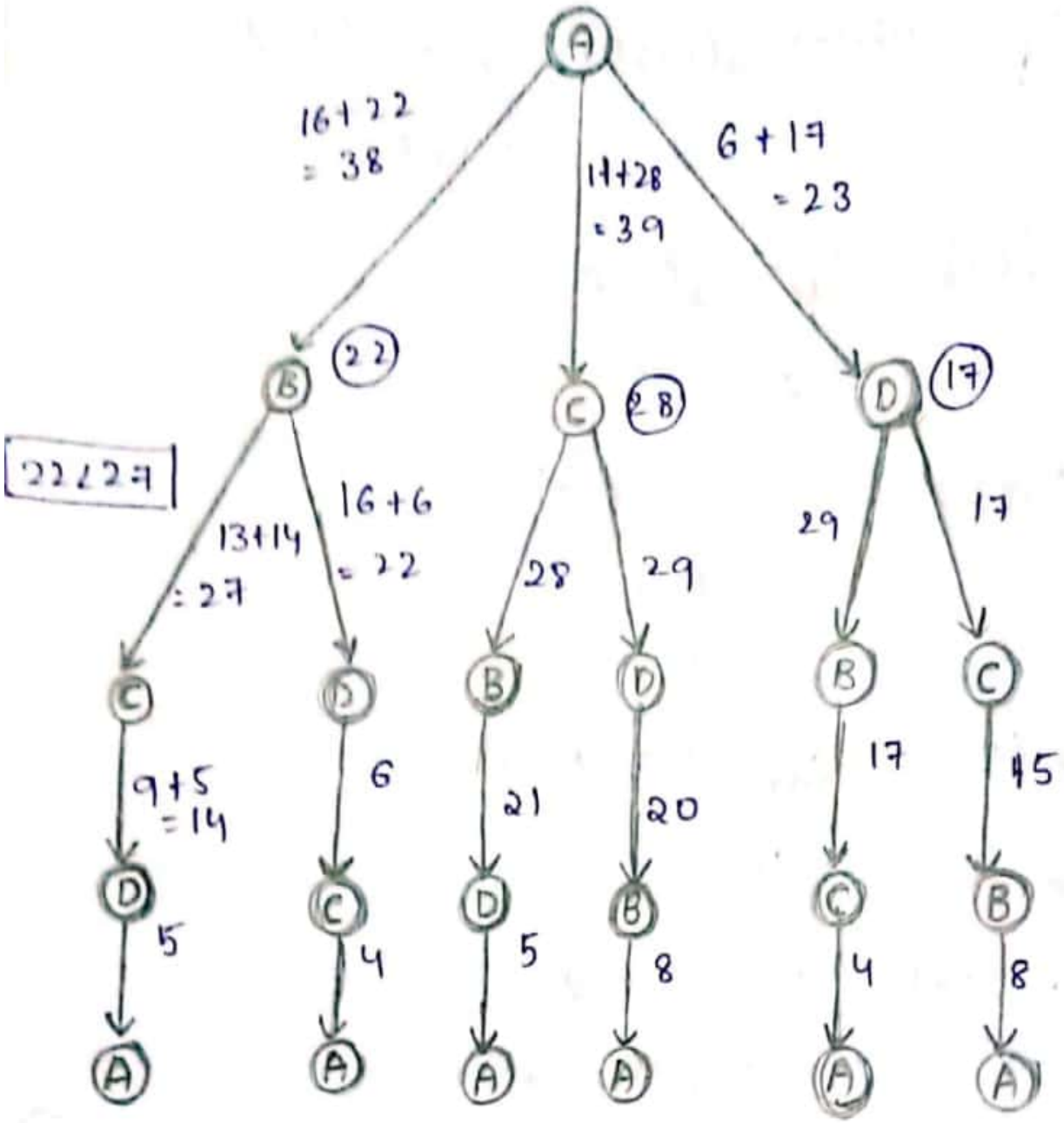
Formulae:-

$$g(i, s) = \min [w(i, j) + g(j, (s - j)]$$

where i is the starting vertex.,

j is another vertex. $\in S$

s is set of vertices. except starting vertex.



Thus from the above diagram, we obtain $g(A, \phi)$ means from D there is no otherway to move to any other vertex except starting vertex. Now, from the adjacency matrix we have

$g(B, \phi) = 8$

$g(C, \phi) = 4$

$g(D, \phi) = 5$

Now, we obtain $g(C,\{D\}) = C_{cd} + g(D,\phi)$

$= 9 + 5 = 14$

$g(D,\{C\}) = C_{dc} + g(C,\phi)$

$= 9 + 4 = 6$

$g(B,\{D\}) = C_{bd} + g(D,\phi)$

$= 16 + 5$

$= 21$

$g(D,\{B\}) = C_{db} + g(B,\phi)$

$= 12 + 8 = 20$

$g(B,\{C\}) = C_{bc} + g(C,\phi)$

$= 13 + 4 = 17$

$g(C,\{B\}) = C_{cb} + g(B,\phi)$

$= 7 + 8 = 15$

Next we compute $g(i,S)$ with $mod S = 2$

$g(B,\{C,D\}) = \min(C_{BC} + g(C,\{D\}), C_{BD} + g(D,\{C\}))$

$= \min(13+14, 16+6)$

$= \min(27, 22)$

$\boxed{g(B,\{C,D\}) = 22}$

$g(C, \{B,D\}) = \min(C_{CB} + g(B,\{D\}), C_{CD} + g(D,\{B\}))$

$= \min(7+21, 9+20)$

$= \min(28, 29)$

$\boxed{g(C,\{B,D\}) = 28}$

$g(D, \{B,C\}) = \min(C_{DB} + g(B,\{C\}), C_{DC} + g(C,\{B\}))$

$= \min(12+17, 2+15)$

$= \min(29, 17)$

$\boxed{g(D,\{B,C\}) = 17}$

Finally from above we obtain g(

$g(A,\{B,C,D\}) = \min(C_{AB} + g(B,\{C,D\}), C_{AC} + g(C,\{B,D\}$

$C_{AD} + g(D,\{B,C\}))$

$= \min(16+22, 11+28, 6+17)$

$= \min(38, 39, 23)$

$\boxed{g(A,\{B,C,D\}) = 23}$

The minimum cost for Hamiltonian cycle is 23.

All pair shortest path

→ In all pair shortest path, when a weighted graph is represented by its weight matrix W then objective is to find the distance between every pair of nodes.

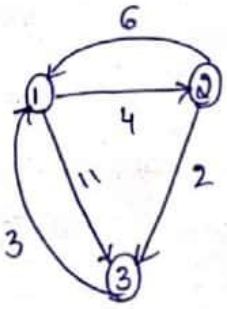
→ We will apply dynamic programming to solve all pairs shortest path.

→ In all pair shortest path algorithm, we first decompose the given problem into sub problems.

→ In this principle of optimally is used for solving the problem.

→ It means any sub path of shortest path is a shortest path between the end nodes.

Example: Compute all pair shortest path for following figure.



Sol: From the above directed graph we will write the Adjacency matrix by considering the vertices and weighted edges.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \end{array}$$

In the matrix A^0 all diagnoal elements are zero because there is no self loops in the given graph.

The i^{th} row and j^{th} column should be the weight of the path from each vertix. Suppose if we take 1^{st} row and 2^{nd} column in the graph there exist path between those vertices and the weight is 4.

If there is no path between the vertices directly we represent that weight as ∞.

Similarly we have to write all the weights.

Now taking 1 as an intermediate vertex. we will write adjacency matrix, using A^0 matrix.

For this we have a formula i.e.,

$$A^k[i,j] = \min \{ A^{k-1}[i,j] , A^{k-1}[i,k] + A^{k-1}[k,j] \}$$

$$A' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

As we are taking 1st vertex as an intermediate vertex we should not keep 1st row and 1st column unchange.

Now $A'[2,3]$

$$A'[2,3] = \min\{ A^{\phi-1}[2,3],\ A^{1-1}[2,1] + A^{1-1}[1,3] \}$$

$$= \min\{ A^{0}[2,3],\ A^{0}[2,1] + A^{0}[1,3] \}$$

$$= \min\{ 2,\ 6+11 \}$$

$$= \min\{ 2,\ 17 \}$$

$$= 2$$

$$A'[3,2] = \min\{ A^{1-1}[3,2],\ A^{1-1}[3,1] + A^{1-1}[1,2] \}$$

$$= \min\{ A^{0}[3,2],\ A^{0}[3,1] + A^{0}[1,2] \}$$

$$= \min\{ \infty,\ 3+4 \}$$

$$= \min\{ \infty,\ 7 \}$$

$$= 7$$

Now taking 2 as an intermediate vertex, we will write adjacency matrix using A' matrix.

Scanned with CamScanner

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

we should consider 2nd row and 2nd column as unchange, because, we are considering 2nd vertex as intermediate vertex.

Now $A^2[1,3]$

$$A^2[1,3] = \min\{A^{2-1}[1,3], A^{2-1}[1,2] + A^{2-1}[2,3]\}$$
$$= \min\{A^1[1,3], A^1[1,2] + A^1[2,3]\}$$
$$= \min\{11, 4+2\}$$
$$= \min\{11, 6\}$$
$$= 6$$

$$A^2[3,1] = \min\{A^{2-1}[3,1], A^{2-1}[3,2] + A^{2-1}[2,1]\}$$
$$= \min\{A^1[3,1], A^1[3,2] + A^1[2,1]\}$$
$$= \min\{3, 7+6\}$$
$$= \min\{3, 13\}$$
$$= 3$$

Similarly to considering 3 as an intermediate vertex write adjacent matrix. using A^2 matrix.

Scanned with CamScanner

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \end{array}$$

3rd row and 3rd column as unchange because we are considering 3rd matrix vertex as intermediate vertex.

$$A^3[1,2] = \min\{ A^{3-1}[1,2],\ A^{3-1}[1,3] + A^{3-1}[3,2] \}$$

$$= \min\{ A^2[1,2],\ A^2[1,3] + A^2[3,2] \}$$

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

$$= \min\{ 4, 13 \}$$

$$= 4$$

$$A^3[2,1] = \min\{ A^{3-1}[2,1],\ A^{3-1}[2,3] + A^{3-1}[3,1] \}$$

$$= \min\{ A^2[2,1],\ A^2[2,3] + A^2[3,1] \}$$

$$= \min\{ 6,\ 2+3 \}$$

$$= \min\{ 6, 5 \}$$

$$= 5$$

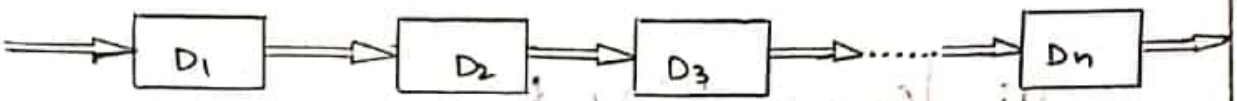
Algorithm:

```
for k=1 to n do
    for i=1 to n do
        for j=1 to n do
            A^k[i,j] = min{A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j]}
```

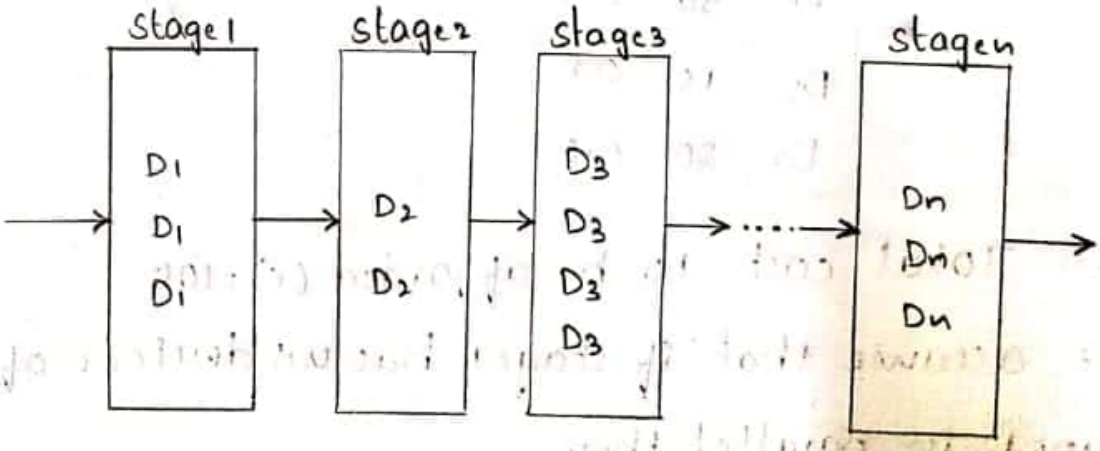

Scanned with CamScanner

Reliability Design:-

In reliability design, the problem is to design a system that is composed of several devices connected in series.



If we duplicate the devices at each stage then the reliability of the system can be increased. Multiple copies of the same device type are connected in parallel through the use of switching circuits. The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.



Multiple devices connected in parallel in each stage.

A dynamic programming solution can be obtained in a manner similar to that used for the knapsack problem. since, we can assume each $c_i > 0$, each m_i must be the range $1 \le m_i \le u_i$, where

$$u_i = \left[(c + c_i - \sum_1^n c_j) / c_i \right]$$

problem:-

Let us design a three stage system with devices D_1, D_2, D_3. the costs are Rs.30, Rs.15 and Rs.20 respectively. the cost constraint to the system is Rs.105. Reliability of devices are 0.9, 0.8 & 0.5.

Tabular form:-

Di	Ci	ri
D_1	30	0.9
D_2	15	0.8
D_3	20	0.5

Total cost to be afforded (c) = 105

we assume that if stage i has m_i devices of type i in parallel then

$$\phi(m_i) = 1 - (1 - r_i)^{m_i}.$$

In a general form:

$$\text{reliability} = 1 - (1-r)^{\text{no. of devices}}$$

we assume $c_i > 0$, each m_i must be range

$$1 \le m_i \le u_i$$

$$u_i = \lceil (c + c_i - \sum c_j) / c_i \rceil$$

using above formula, compute u_1, u_2, u_3 for devices D_1, D_2, D_3

$$u_1 = \left\lceil \frac{105+30-(30+15+20)}{30} \right\rceil = \frac{70}{30} = 2$$

$$u_2 = \left\lceil \frac{105+15-(30+15+20)}{15} \right\rceil = \frac{55}{15} = 3$$

$$u_3 = \left\lceil \frac{105+20-(30+15+20)}{20} \right\rceil = \frac{60}{20} = 3$$

Tabular form:

D_i	c_i	r_i	u_i
D_1	30	0.9	2
D_2	15	0.8	3
D_3	20	0.5	3

Maximum No. of extra copies obtained for each device

$$D_1 = \left| \frac{C - \Sigma C_i}{C_1} \right| = \frac{40}{30} = 1$$

total = 1 original copy + 1 extra copy

= 2 copies.

$$D_2 = \left| \frac{C - \Sigma C_i}{C_i} \right| = \frac{40}{15} = 2$$

total = 1 + 2 = 3

$$D_3 = \left| \frac{C - \Sigma C_i}{C_1} \right| = \frac{40}{20} = 2$$

total = 1 + 2 = 3

$S_{copy}^{device} \Rightarrow (r, c)$

for each copy, reliability is multiplied,

$r_t = r_1 \times r_2 \times r_3$, cost is added, $c_t = c_1 + c_2 + c_3$

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

Device 1:

for devices having 1 copy

reliability: $1 - (1 - r)^1$

$= 1-(1-0.9)^1$

$= 1-0.1$

$= 0.9$

cost = 30

$S_1' = \{(0.9, 30)\}$

for device 1, having 2 copies

reliability $= 1-(1-r)^2$

$= 1-(1-0.9)^2$

$= 1-0.01$

$= 0.99$

cost $= 30+30 = 60$

$S_2' = \{(0.99, 60)\}$

$S' = \{(0.9, 30), (0.99, 60)\}$

Dominance Rule:-

If S' contains two pairs (f_1, x_1) and (f_2, x_2) with the property that $f_1 \geq f_2$ and $x_1 \leq x_2$, then (f_1, x_1) dominates (f_2, x_2) can be discarded. Discarding & pruning rules such as the one above is known as dominance rule. Dominating tuples will be present in S^1 and dominated tuples has to be discarded from S'.

case1 : If $f_1 \leq f_2$ and $x_1 > x_2$ then discard (f_1, x_1)

case2: If $f_1 \geq f_2$ and $x_1 < x_2$ the discard (f_2, x_2)

case3: otherwise simply write (f_1, x_1)

Reliability (1) $= 1 - (1-r_1)^{m_1} = 1 - (1-0.5)^1$

$= 1 - 0.5 = 0.5$

cost = 20

$S_1^3 = \{(0.5\,(0.72), 45+20), (0.5\,(0.864), 60+20), (0.5\,(0.8928), 75+20)\}$

$S_1^3 = \{(0.36, 65), (0.437, 80), (0.4464, 95)\}$

Reliability (2) $= 1 - (1-0.5)^2$

$= 0.75$

cost $= 20+20 = 40$

$S_2^3 = \{(0.75\,(0.72), 45+20+20), (0.75\,(0.864), 60+20+20),$

$(0.75\,(0.8928), 75+20+20)\}$

$= \{(0.54, 85), (0.648, 100), (0.6696, 115)\}$

Reliability (3) $= 1 - (1-0.5)^3 = 0.875$

cost $= 20+20+20$

$S_3^3 = \{(0.875\,(0.72), 45+20+20+20), (0.875\,(0.864), 60+20+20+20)$

$(0.875\,(0.8928), 75+20+20+20)\}$

$S_3^3 = \{(0.63, 105), (1.756, 120), (0.7812, 135)\}$

If cost exceeds 105, remove that tuples.

$S^3 = \{(0.36,65),(0.439,80),(0.54,85),(0.648,100)\}$

Devices	No.of copies
D_1	1
D_2	2
D_3	2

final result: (0.64,100)-from s_3 which is taken from s_3^2

So we take 2 copies of devices 3

$2 \times 20 = 40$

$100 - 40 = 60$

$C = 60$

Cost 60 from s_3 - (0.864, 60) is taken from s_2^2 so we need 2 copies of devices

$2 \times 15 = 30$

$60 - 30 = 30$

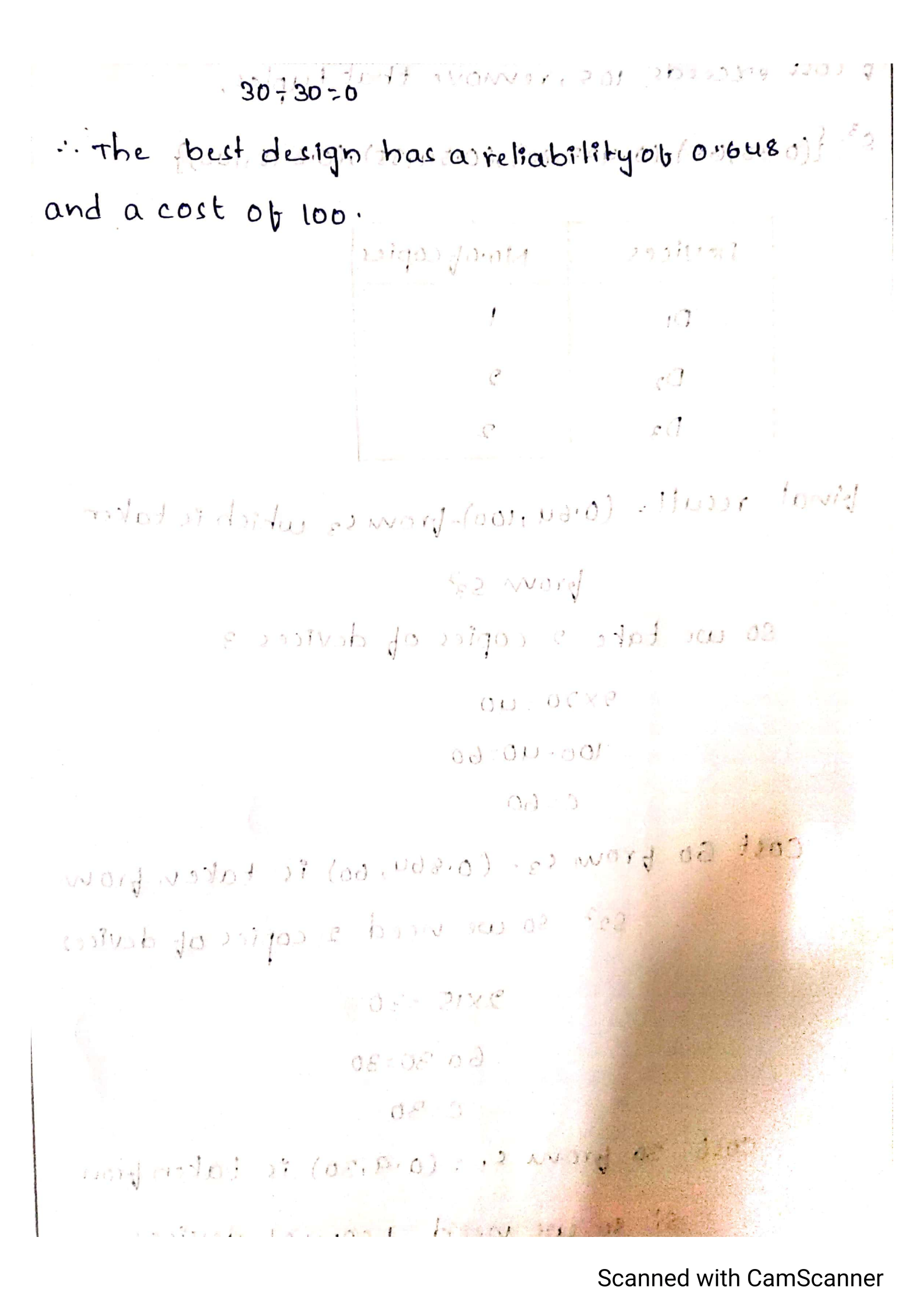
$C = 30$

Cost 30 from $s_1 = (0.9, 30)$ is taken from Si so we need 1 copy of devices

$1 \times 30 = 30$.

$30 - 30 = 0$

∴ The best design has a reliability of 0.648 and a cost of 100.



DESIGN AND ANALYSIS OF ALGORITHMS

(DAA)

UNIT – 5

Backtracking:

- General Method
- Applications:
  - N-Queen Problem
  - Sum of Subsets Problem
  - Graph Coloring
  - Hamiltonian Cycles.

Branch and Bound:

- General Method
- Applications:
  - Travelling Sales Person Problem
  - 0/1 Knapsack Problem
  - LC Branch and Bound Solution
  - FIFO Branch and Bound Solution.

0/1 Knapsack Problem Using FIFO Branch & Bound

Definition:.

Branch & Bound discovers branches within the complete search space by using estimated bounds to limit the number of possible solutions. The different types

1. FIFO
2. LIFO
3. LC

define different strategies to explore the search space and generate branches.

FIFO:-

FIFO (first in, first out) always the oldest node in the queue is used to extend the branch. This leads to a breadth-first search, where all nodes at depth d are visited first, before any nodes at depth d+1 are visited.

In FIFO branch and bound, as is visible by the name, the child nodes are explored in first in first out manner. We start exploring from starting child node.

Procedure:-

1. Draw a state space tree and set upper bound = ∞
2. compute $c^{\wedge}(x)$, $u(x)$ for each node.
3. $u(x) = -\sum p_i$

$$c^{\wedge}(x) = u(x) - [m - \text{current total weight}] * \frac{[\text{actual profit of remaining object}]}{[\text{actual weight of remaining object}]}$$

4. If $u(x)$ is minimum than upper will set to $u(x)$
5. If $c^{\wedge}(x) >$ upper, kill node x
6. Next live node becomes E-node and Generate children for E-node.
7. Repeat 2 to 6 until all nodes get covered
8. The minimum cost $c^{\wedge}(x)$ becomes the answer node. Trace the path in backward direction from x to root for solution subset.

Steps:

for node - 1:

We take the table

O_i	P_i	w_i
1	10	2
2	10	4
3	12	6
4	18	9

the bag capacity is $m = 15$.

i) node - 1 :

Let us consider $m = 15$ and upper bound $= \infty$

for node 1 $\hat{c}(1) = -38$

$u(1) = -32$

6	
~~18~~	$3/9 \times 18^2$
12	6
10	4
10	2

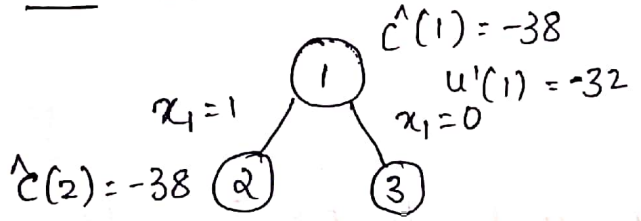
$\}u(1)\}\hat{c}(1)$

So now upper bound $= -32$

(1) $\hat{c}(1) = -38$

$u(1) = -32$

2) node - 2 :



The $x_1 = 1$ the node 1 is included so.

6	$3/9 \times 18$
12	6
10	4
10	2

$\}u(2)\}\hat{c}(2)$

3) node - 3 :

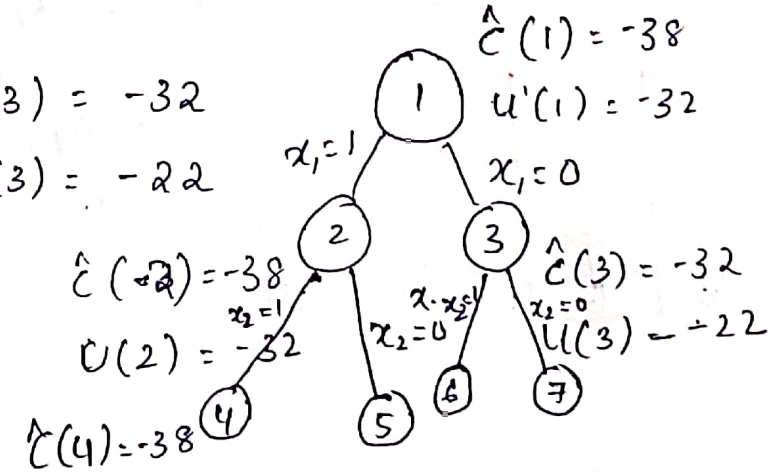
here for node - 3 $x_1 = 0$ x_1 is not included so

10	$5/9 \times 18$
12	6
10	4

$\}-22\}-32$

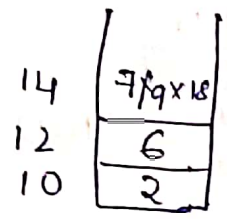
$\hat{c}(3) = -32$

$u(3) = -22$



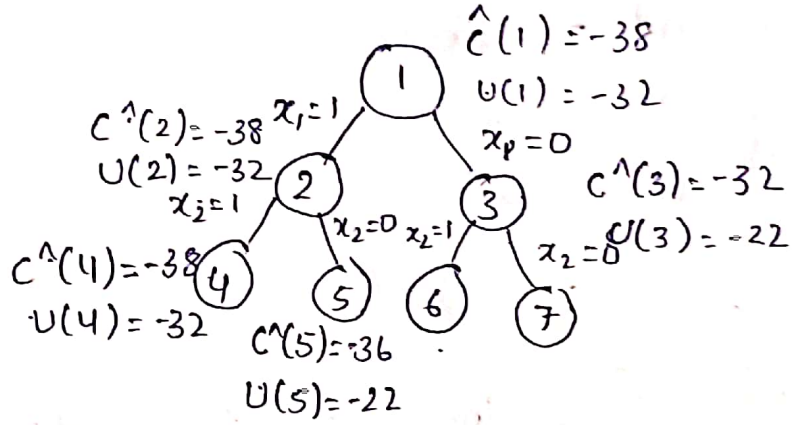
node - 5 :-

Here x_2 is not included



$\hat{c}(5) = -36$

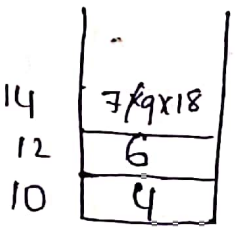
$u(5) = -22$



node - 6 :-

we have include x_2 and doesn't include x_1

so

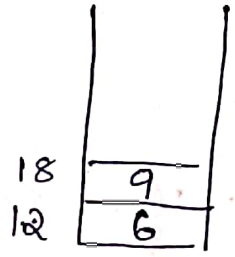


$\hat{c}(6) = -32$

$u(6) = -22$

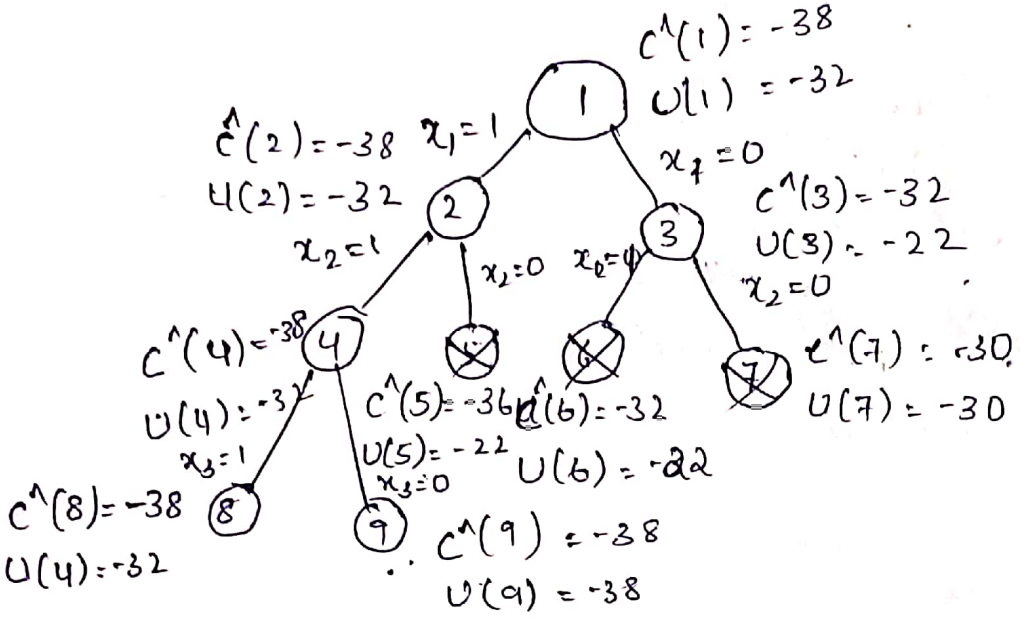
node - 7 :-

we have to exclude $x_1 = 0$ and $x_2 = 0$



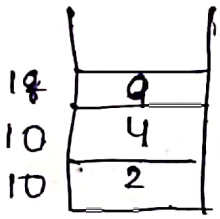
$\hat{c}(7) = -30$

$u(7) = -30$



Node - 9 :

x_3 must not be included



$c^(9) = -38$

$u^(9) = -38$

The upper bound node is replaced because -32 is larger than -38

upper bound = -38

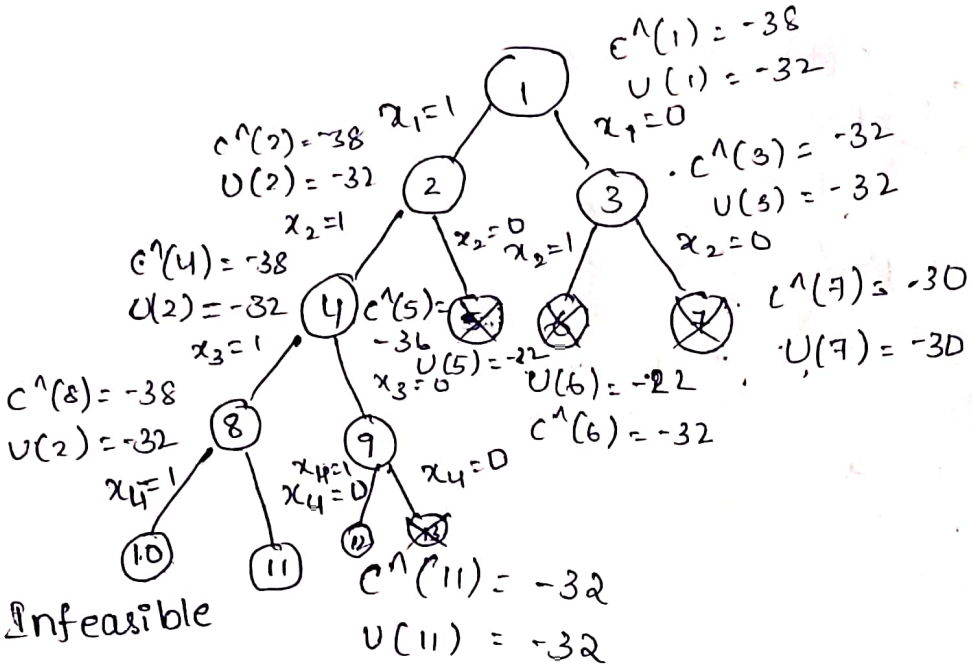
- $c^(x) >$ upper, Kill node x

5 - -36 > -38 so, Kill the node 5.

6 - -32 > -38, so Kill the node 6.

7 - -30 > -38, so Kill the node 7

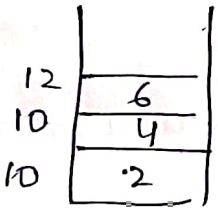
So the tree will be



For node-10 the bag exceeds the capacity so it is infeasible.

node-11 :-

we have to exceed x_4.



then

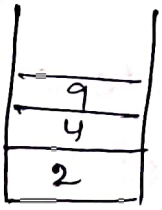
$c^(11) = -32$

$U(11) = -32$

Here $-32 > -38$; kill the node-11

node-12 :-

$x_3 = 0$ is excluded

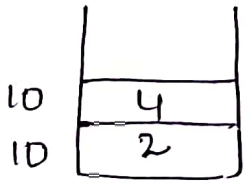


$c^(12) = -38$

$U(12) = -38$

node-13 :-

x_3, x_4 is excluded



$c^(13) = -20$

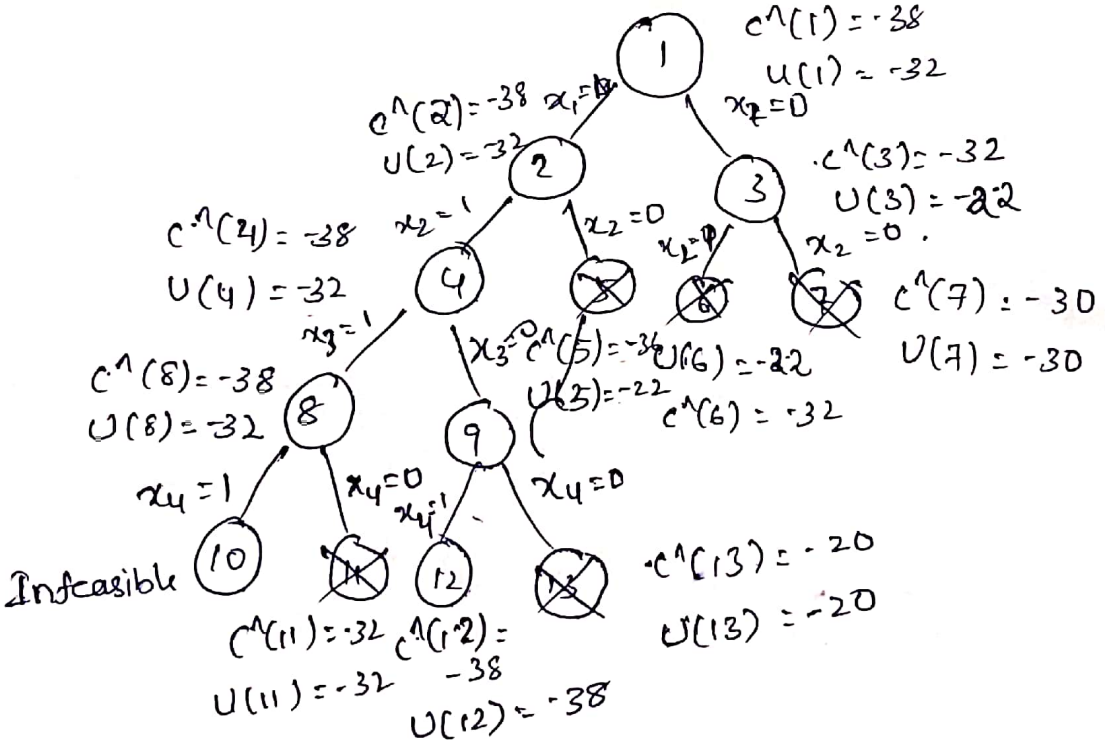
$U(13) = -20$

$-20 > -38$, node 13 is killed

Node - 12 is the answer

	P_i	w_i
$x_1 = 1$	10	2
$x_2 = 1$	10	4
$x_3 = 0$	-	
$x_4 = 1$	18	9
	+38	15

The profit is 38 and weight 15.



SUM OF SUBSETS

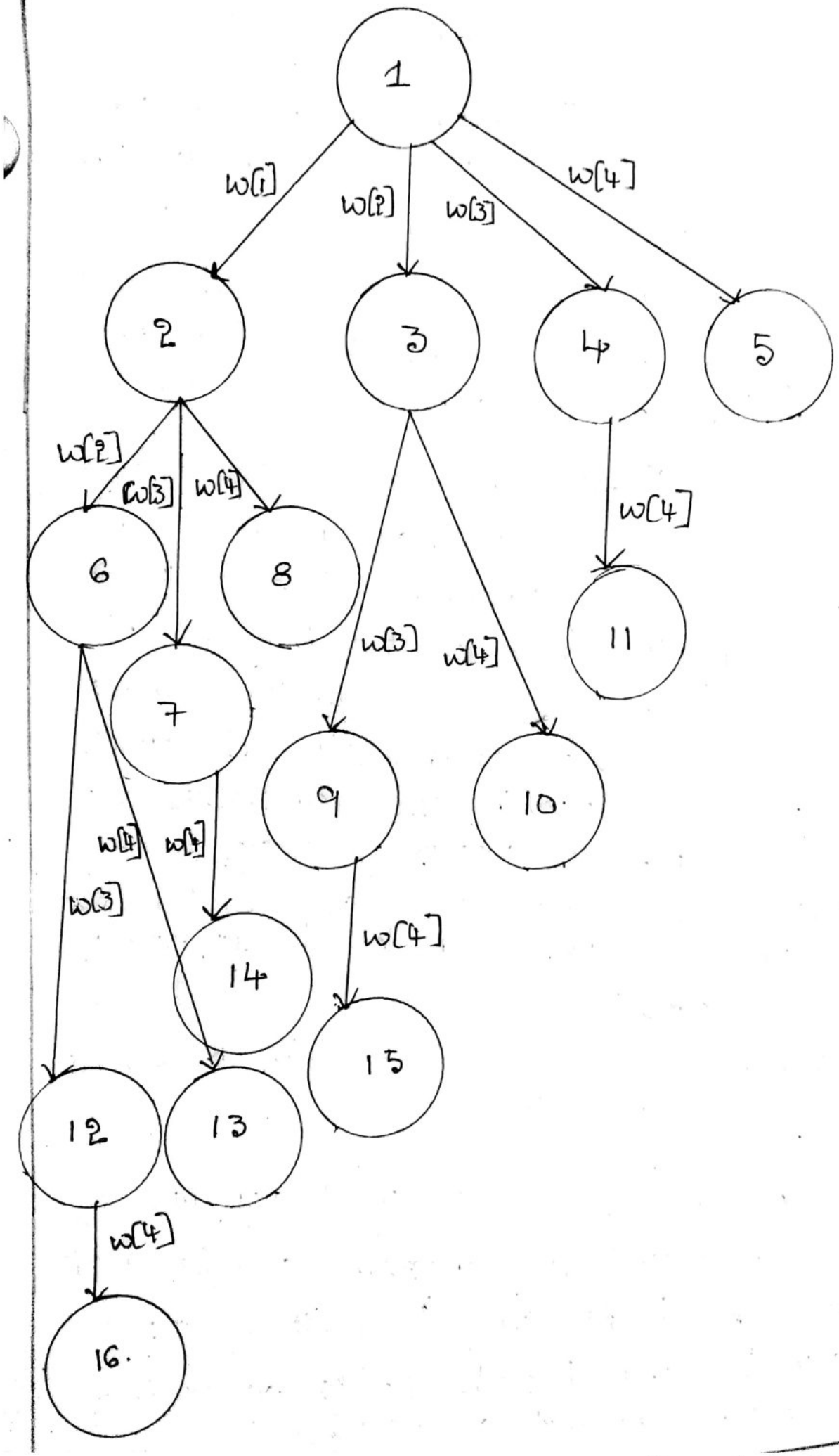
Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number k. we are considering the set contains non-negative values. It is assumed that the input set is unique.

In a state space tree the rootnode represents a function called and the branch represents the candidate element as we go down the depth of the tree we add elements until we satisfy the explict constraints. we continue to add child's nodes for wherever the constraints are not satisfy the further generations of subtree that node are stop and backtra-ck to the previous node to explore the other nodes.

General state space tree with 4 elements [w[1] - - - w[4]).

Assume given set of 4 elements, say w[1] - - - w[4]. tree diagrams can be used to design backtracking algorithms. the following tree diagram depicts approach of generating variable sized tuple.

Scanned with CamScanner



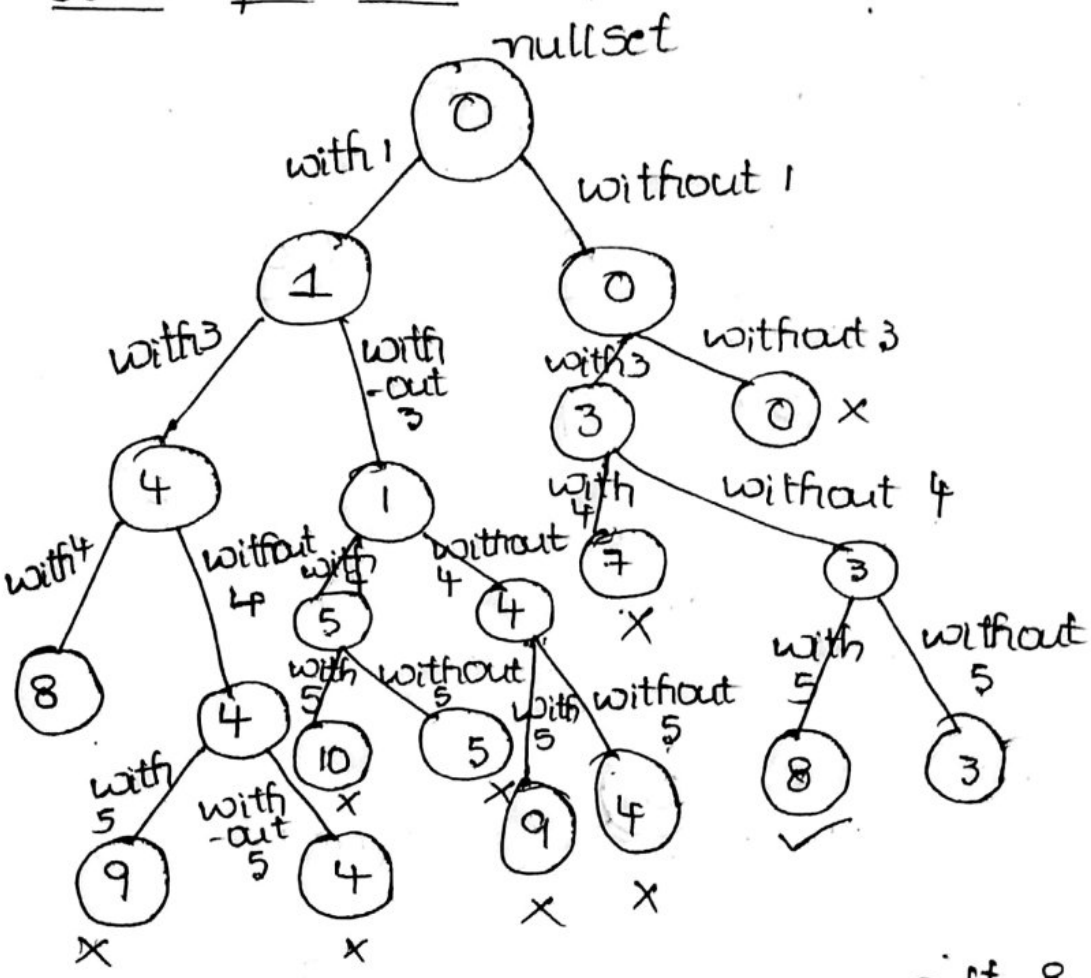
Scanned with CamScanner

Example-1:

consider Given set of elements $\{1,3,4,5\}$ where $K = 8$.

Sol:- $\{\{1,3,4\}\ \{3,5\}\}$

State space tree



∴ The subsets whose sum result 8 are $\{1,3,4\}$, $\{3,5\}$.

Here move from top to end until reach our K value.

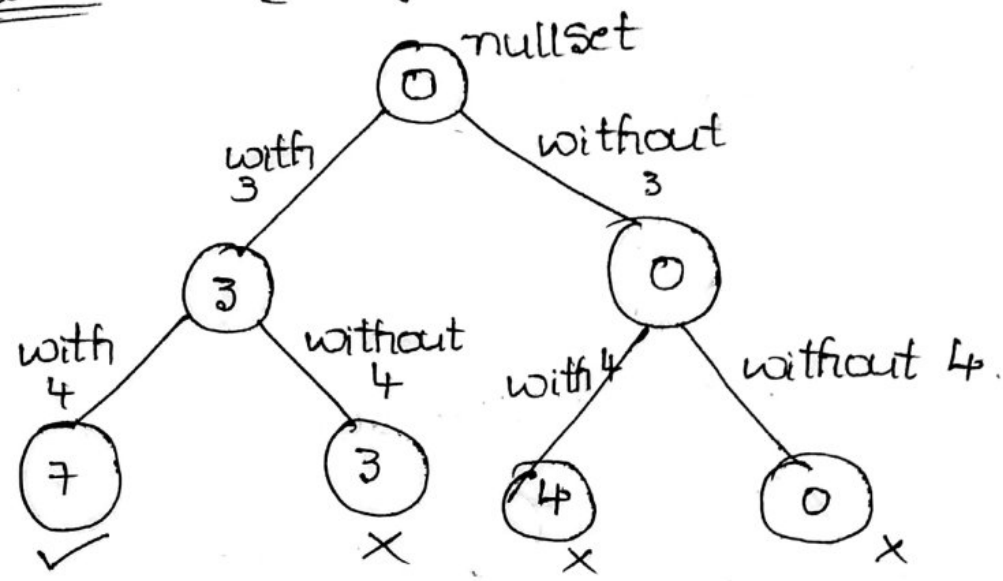
we are unable to get the solution then kill the tree.

Scanned with CamScanner

Example-2:

Consider the subset {1,2,3,4} and K=7.

Solution: {3,4}.



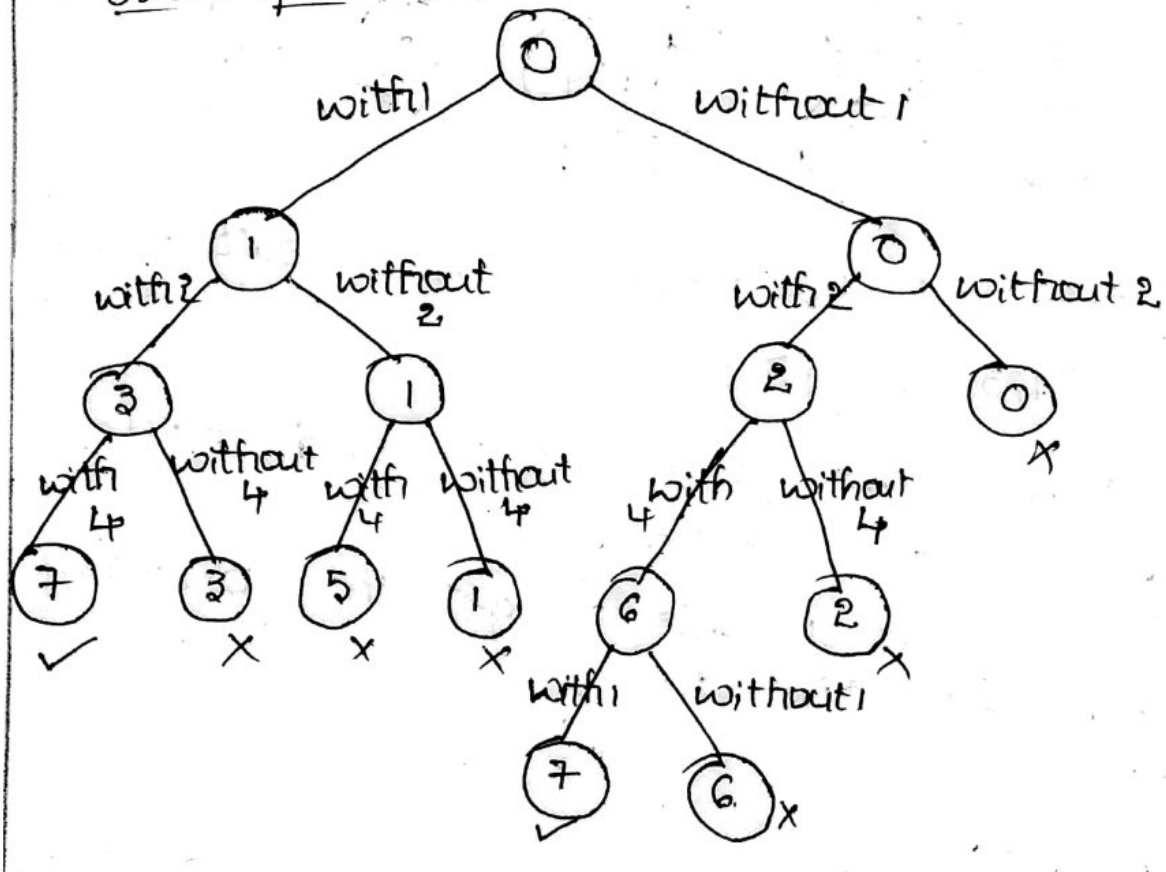
State space tree.

Example-3:

Consider the subset {1,2,4,6} K=7.

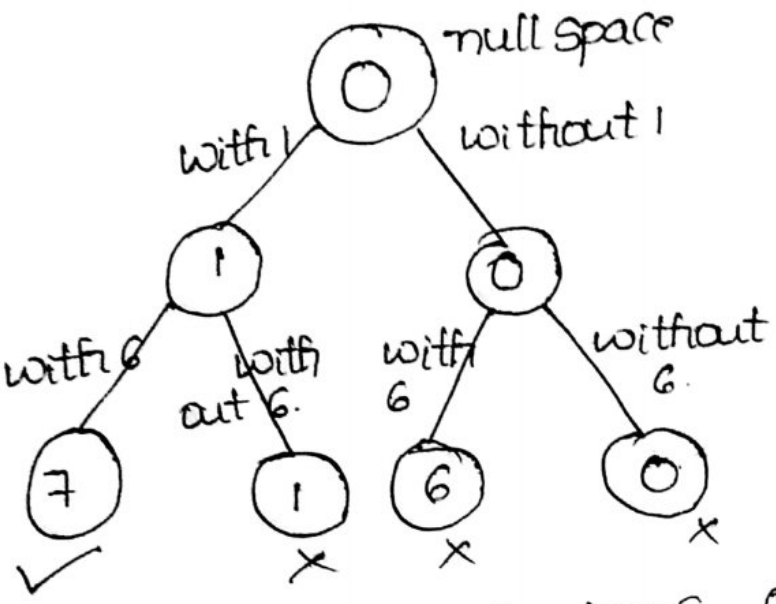
Ans: {1,2,4}, {1,6}.

State space tree



Scanned with CamScanner

State space tree.



Advantages and Applications of subset-sum problem.

* It is specially constructed problem which is known to be easy.
* There are security problems other than public-key codes for which subset-problems are useful.
* It is used in computer passwords.
* It is used in message verification.

Scanned with CamScanner

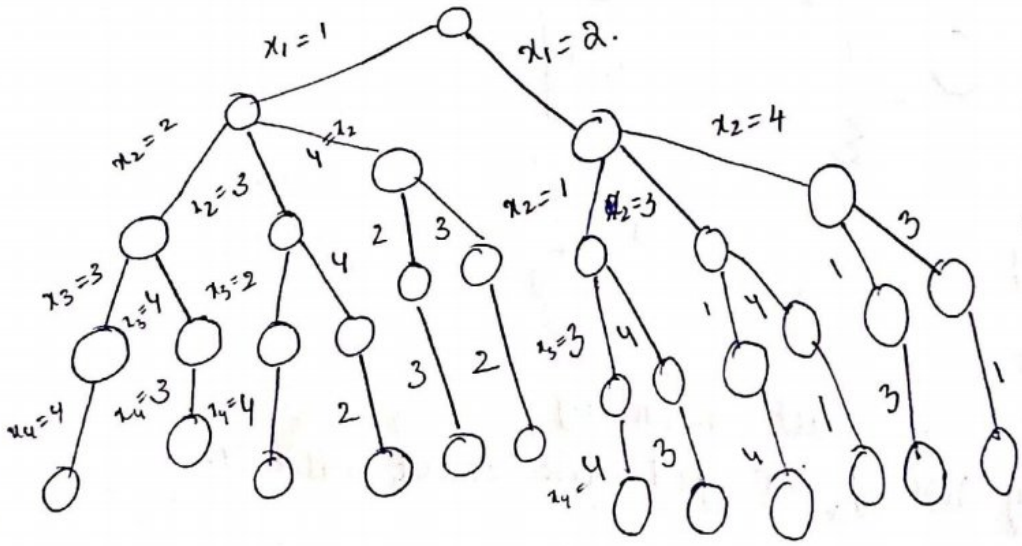
N-Queens Problem

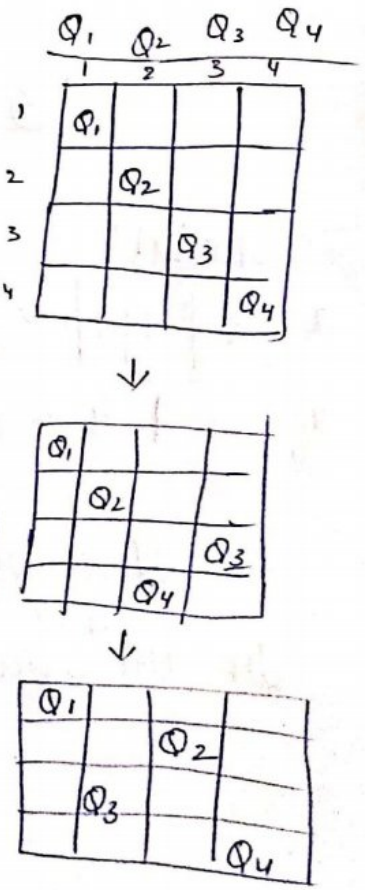
→ There will be a chess board given with n×n cells. n Queens will be given. In a chess board a Queen can move horizontally, vertically and diagonally. In this problem also we have to ~~keep~~ place N Queens such that no Queen attacks another.

→ for this problem, we may have many solutions and we need all the solutions which satisfies the condition. So, for these type of getting all solutions backtracking is used. (for getting optimal solution we have to choose dynamic Programming).

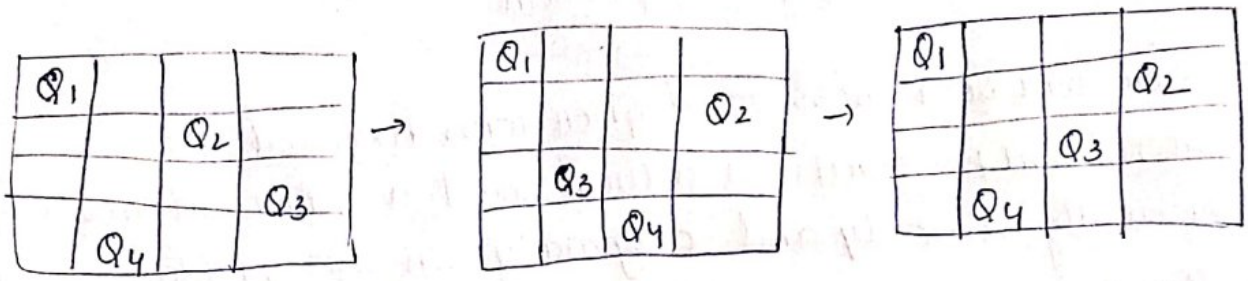
→ for example consider n=4 and for reducing the problem i^{th} Queen is placed in the i^{th} row where i ranges from 1 to 4. So we have to choose its appropriate column.

State space tree

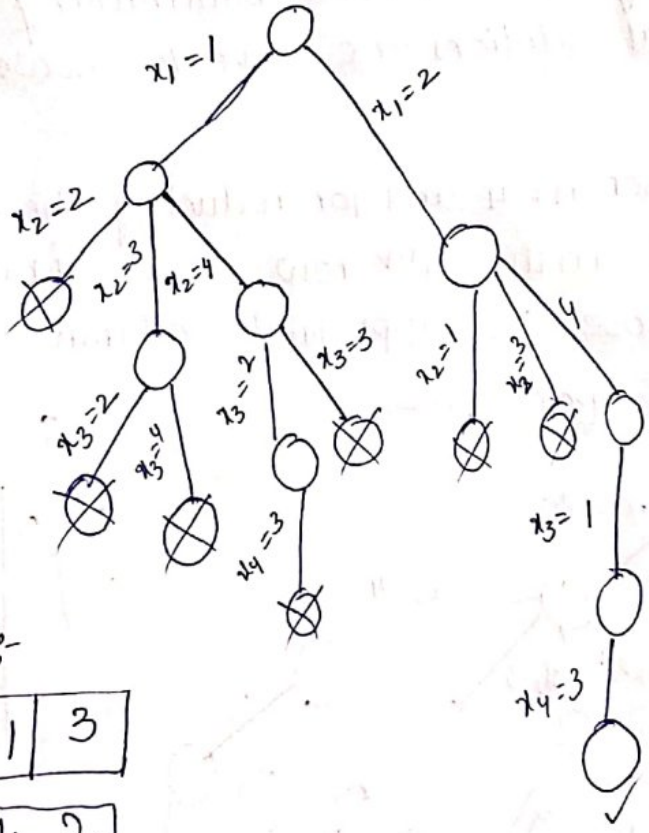




Scanned with CamScanner



Similarly we can draw for x_1 placed at 3rd and 4th columns. Now by using Bounding function we get the solutions for N-Queens problem.



Solution :-

x	2	4	1	3
x	3	1	4	2

Now by using, back tracking we get the above solution

In the state space tree there are

Scanned with CamScanner

$$1 + 4 + 4\times3 + 4\times3\times2 + 4\times3\times2\times1$$

$$= 1 + \sum_{i=0}^{3}\left[\prod_{j=0}^{i}(4-j)\right] \quad \text{for 4 Queens}$$

$$= 1 + \sum_{i=0}^{N-1}\left[\prod_{j=0}^{i}(N-j)\right] \quad \text{for N Queens the}$$

total nodes which are possible to place the Queens without checking all the conditions (diagonal check is not done) in state space tree are

$$1 + \sum_{i=0}^{N-1}\left[\prod_{j=0}^{i}(N-j)\right]$$

Scanned with CamScanner

Introduction to Branch-and-bound technique:

Branch-and-bound is a state-space search method that can be visualized as an improved form of backtracking. In the branch-and-bound technique, a set of feasible solutions are partitioned, and the subsets that do not have optimal solutions are deleted. Branch and bound is an algorithm design paradigm which is generally used for solving the optimization problems. Branch-and-bound approach has 2 major steps - branching and bounding. These 2 steps are repeated till the problem is solved.

Branching involves division of a given problem into two or more subproblems. These subproblems are similar to the original problem but smaller in size.

Let us assume that $f(x)$ is a objective function, S is the state-space tree that has all the solutions. Hence, S is a set of all solutions also called as feasible region. S is divided into k feasible subregions S_i. The union of these subregions will give S. i.e $S = \bigcup_{i=1}^{k} S_i$.

In the branch-and-bound technique, one has to search the state-space tree for finding optimal solution. Hence, it uses bounds for limiting the growth of the state-space tree exponentially.

Second step, bounding step aids in limiting the growth of the state space. In this step, the best solution of the subproblems is identified and used as lower bound.

For every node i of the state space, the lower bound needs to be calculated. Similarly, the upper bound needs to be computed. This includes the minimum amount of work required to compute the result, the best feasible solution.

In branching step, state space is divided into k feasible subregions, uses a lower bound for minimization problem and an upper bound for maximization problem.

The success of branch-and-bound algorithm depends on the quality of the lower and upper bounds. Implementation of the bounding step is more difficult than branching step.

Branch-and-bound technique terminates when:

1. The subproblem of given node gives an optimal solution.
2. The subproblem of given node yields a sub-optimal solution.
3. The subproblems turn out to be infeasible.

The objective of branch-and-bound is similar to that of backtracking method, difference is that the branch-and-bound technique calculates a bound of a possible solution at each and every stage.

This approach is used for a number of NP-hard problems like

Integer programming
Nonlinear programming
Travelling salesman problem (TSP)
0/1 knapsack problem etc.

Like the backtracking technique, branch-and-bound algorithms also construct a state-space tree and employ search techniques to search for optimal solution.

Differences between backtracking and branch-and-bound:

Backtracking	Branch-and-bound
uses only DFS technique	Uses BFS, DFS, least-cost search.
explores state-space trees partially, potential solutions may sometimes be ignored.	checks completely for an optimal solution, always potential solutions are obtained.
can be used for enumerative, decision and optimization problems.	can be used only for an optimization problem.

Backtracking:

Backtracking is a systematic method for searching or more solutions for a given problem. It is a redefined brute force technique used for solving problems. It was proposed by D.H. Lehmer and later refined by R.J. Walker.

Backtracking can effectively solve multi-decision problems, where the final solution is visualized as a set of decisions. The execution of a decision or choice leads to another set of decisions or choices. These decisions ~~gave be~~ encountered until an successful solution of a problem arrives. One can back-track and try other alternatives to achieve the aim. Thus the overall strategy may end in a successful or an unsuccessful outcome. A Backtracking design paradigm can solve following three types of problems:

Enumeration problems:

In an enumeration problem, all solutions are listed for a given problem.

Decision problems:

In an decision problem, a solution is given in terms of Yes/no.

Scanned with CamScanner

Optimization problems:

In optimization problems, optimal solutions are required, which maximize or minimize the given objective function as per constraints of given problem.

Example:

Consider a scenario of searching the torch light in a dark room. If one encounters a wall, it is a dead end. Hence, one has to then backtrack and continue the search process till the torch light is found if it is really present in the room. Other examples where backtracking can be used for puzzles such as mazes and sudoku. It can be observed that many trail and error processes are required to find solutions for these puzzles.

The brute force technique can solve a given problem by listing out all its possible solutions, from which the optimal solution can be picked. Brute force algorithm leads to exponential time complexity. The backtracking approach can solve most of these problems in polynomial time.

Scanned with CamScanner

A backtracking algorithm solves problems incrementally by adding candidate solutions till the final solution is obtained. If partial solution is not leading to a solution, then it is rejected along with its other partial solutions; this process is called domino principle. The backtracking process is continued till goal state is reached, otherwise the search is termed as unsuccessful. Backtracking is a depth first search with some bounding functions. Bounding functions represents constraints of given problem. First, the backtracking process defines a solution vector as n-tuple vector $(x_1, x_2 \ldots x_n)$ for the given problem. Hence n is number of components of solution vector ~~as A tuple vector $(x_1, x_2 \ldots$~~ for each x_i, where i ranges from 1 to n represented a partial solution. These partial solution components x_i are generated based on concept of constraints.

Scanned with CamScanner

Backtracking algorithm:

step 1: While there are many choices left out, perform step 2

step 2: Generate a state-space tree using DFS approach.

2a: check next configuration using bounding functions.

2b: If solution is promising then

if solution is obtained then

print the solution

else

backtrack to the parent of the node and try again.

step 3: end.

Complexity:

It is difficult to evaluate backtracking algorithm analytically. Donald E. knuth suggested a method where random path can be generated from root to a leaf of a statespace tree. if c_1 children are encountered for first compononent of solution vector and so on, Then no. of children encoured for solution of random vector is given by relation $T(n) = 1 + c_1 + c_1c_2 + \cdots + c_1c_2 \cdots c_n$.

Scanned with CamScanner

HAMITONIAN CYCLES

A Hamiltonian cycle, also called a Hamiltonian circuit, Hamilton cycle, or Hamilton circuit, is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once . A graph possessing a Hamiltonian cycle is said to be a Hamiltonian graph. By convention, the singleton graph K_1 is considered to be Hamiltonian even though it does not possess a Hamiltonian cycle, while the connected graph on two nodes K_2 is not.

The Hamiltonian cycle is named after Sir William Rowan Hamilton, who devised a puzzle in which such a path along the polyhedron edges of an dodecahedron was sought (the Icosian game).

In general, the problem of finding a Hamiltonian cycle is NP-complete (Karp 1972; Garey and Johnson 1983, p. 199), so the only known way to determine whether a given general graph has a Hamiltonian cycle is to undertake an exhaustive search. Rubin (1974) describes an efficient search procedure that can find some or all Hamilton paths and circuits in a graph using deductions that greatly reduce backtracking and guesswork. A probabilistic algorithm due to Angluin and Valiant (1979), described by Wilf (1994), can also be useful to find Hamiltonian cycles and paths.

Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

Input:
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
Output:
An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.
For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}.

```
(0)--(1)--(2)
|  / \  |
```

```
| /  \ |
|/    \|
(3)-------(4)
```

And the following graph doesn't contain any Hamiltonian Cycle.

```
(0)--(1)--(2)
|  / \  |
| /  \ |
|/    \|
(3)     (4)
```

Naive Algorithm

Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! (n factorial) configurations.

```
while there are untried conflagrations
{
  generate the next configuration
  if ( there are edges between two consecutive vertices of this
     configuration and there is an edge from the last vertex to
     the first ).
  {
     print this configuration;
     break;
  }
}
```

Backtracking Algorithm

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

Implementation of Backtracking solution

Following are implementations of the Backtracking solution.

```
/* C program for solution of Hamiltonian Cycle problem
   using backtracking */
#include<stdio.h>
#define V 5

void printSolution(int path[]);

bool isSafe(int v, bool graph[V][V], int path[], int pos)
{

    if (graph [ path[pos-1] ][ v ] == 0)
        return false;

    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    if (pos == V){
        if ( graph[ path[pos-1] ][ path[0] ] == 1 )
            return true;
        else
            return false;
    }
    for (int v = 1; v < V; v++){
        if (isSafe(v, graph, path, pos)){
            path[pos] = v;
            if (hamCycleUtil (graph, path, pos+1) == true)
                return true
            path[pos] = -1;
        }
    }
    return false;
}
```

```
/* This function solves the Hamiltonian Cycle problem using
Backtracking.*/
bool hamCycle(bool graph[V][V])
{
   int *path = new int[V];
   for (int i = 0; i < V; i++)
      path[i] = -1;
   path[0] = 0;
   if ( hamCycleUtil(graph, path, 1) == false )
   {
      printf("\nSolution does not exist");
      return false;
   }

   printSolution(path);
   return true;
}
void printSolution(int path[])
{
   printf ("Solution Exists:"
         " Following is one Hamiltonian Cycle \n");
   for (int i = 0; i < V; i++)
      printf(" %d ", path[i]);
   printf(" %d ", path[0]);
   printf("\n");
}
int main()
{
  bool graph1[V][V] = {{0, 1, 0, 1, 0},
                       {1, 0, 1, 1, 1},
                       {0, 1, 0, 0, 1},
                       {1, 1, 0, 0, 1},
                       {0, 1, 1, 1, 0},
                      };

   hamCycle(graph1);
   bool graph2[V][V] = {{0, 1, 0, 1, 0},
                       {1, 0, 1, 1, 1},
                       {0, 1, 0, 0, 1},
```

```
                {1, 1, 0, 0, 0},
                {0, 1, 1, 0, 0},
               };
    hamCycle(graph2);

    return 0;
}
```

Output:

```
Solution Exists: Following is one Hamiltonian Cycle
0 1 2 4 3 0

Solution does not exist
```

The above code always prints cycle starting from 0. The starting point should not matter as the cycle can be started from any point. If you want to change the starting point, you should make two changes to the above code.
Change “path[0] = 0;” to “path[0] = s;” where s is your new starting point. Also change loop “for (int v = 1; v < V; v++)" in hamCycleUtil() to "for (int v = 0; v < V; v++)".

TRAVELLING SALESMAN PROBLEM

The following graph shows a set of cities and distance between every pair of cities-If salesman starting city is A, then a TSP tour in the graph is-

A → B → D → C → A

Cost of the tour

= 10 + 25 + 30 + 15

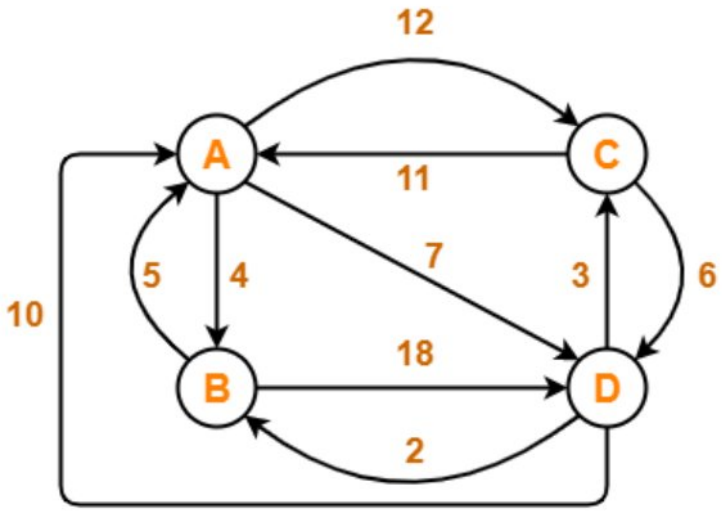
= **80 units**

In this article, we will discuss how to solve travelling salesman problem using branch and bound approach with example.

PRACTICE PROBLEM BASED ON TRAVELLING SALESMAN PROBLEM USING BRANCH AND BOUND APPROACH-

Problem-

Solve Travelling Salesman Problem using Branch and Bound Algorithm in the following graph-



Solution-

Step-01:

Write the initial cost matrix and reduce it-

	A	B	C	D
A	∞	4	12	7
B	5	∞	∞	18
C	11	∞	∞	6
D	10	2	3	∞

After row reduction Performing

this, we obtain the following row-reduced matrix-

	A	B	C	D
A	∞	0	8	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	1	∞

Column Reduction-

Performing this, we obtain the following column-reduced matrix-

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

Finally, the initial distance matrix is completely reduced.

Now, we calculate the cost of node-1 by adding all the reduction elements.

Cost(1)

= Sum of all reduction elements

= 4 + 5 + 6 + 2 + 1

= 18

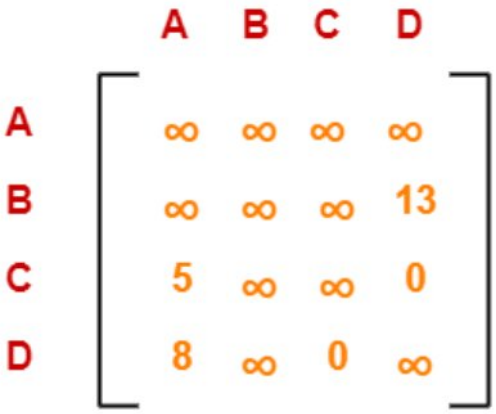
Step-02:

- We consider all other vertices one by one.
- We select the best vertex where we can land upon to minimize the tour cost.

Choosing To Go To Vertex-B: Node-2 (Path A → B)

- From the reduced matrix of step-01, M[A,B] = 0
- Set row-A and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-



Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- Reduce all the elements of row-2 by 13.
- There is no need to reduce row-3.
- There is no need to reduce row -4

Column Reduction-

- Reduce the elements of column-1 by 5.
- We can not reduce column-2 as all its elements are ∞.
- There is no need to reduce column-3.
- There is no need to reduce column -4

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-2.

Cost(2)

= Cost(1) + Sum of reduction elements + M[A,B]

= 18 + (13 + 5) + 0

= 36

Choosing To Go To Vertex-C: Node-3 (Path A → C)

- From the reduced matrix of step-01, M[A,C] = 7

- Set row-A and column-C to ∞
- Set M[C,A] = ∞

Now, resulting cost matrix is-

Now,

- We reduce this matrix.
- Then, we find out the cost of node-03.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- There is no need to reduce row-2.
- There is no need to reduce row-3.
- There is no need to reduce row-4.

Thus, the matrix is already row-reduced.

Column Reduction-

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- We can not reduce column-3 as all its elements are ∞.
- There is no need to reduce column-4.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-3.

Cost(3)

= Cost(1) + Sum of reduction elements + M[A,C]

= 18 + 0 + 7

= 25

Choosing To Go To Vertex-D: Node-4 (Path A → D)

- From the reduced matrix of step-01, M[A,D] = 3
- Set row-A and column-D to ∞
- Set M[D,A] = ∞

Now, resulting cost matrix is-

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	5	∞	∞	∞
D	∞	0	0	∞

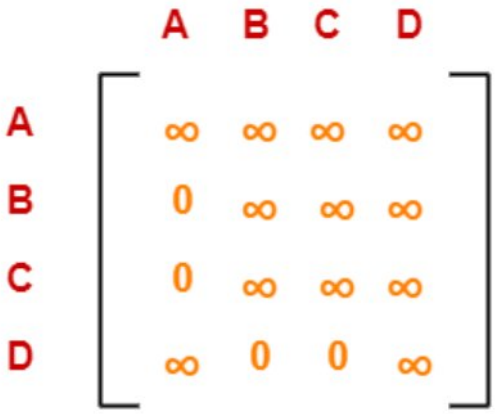
Now,

- We reduce this matrix.
- Then, we find out the cost of node-04.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- There is no need to reduce row-2.
- Reduce all the elements of row-3 by 5.
- There is no need to reduce row-4.

Performing this, we obtain the following row-reduced matrix-



Column Reduction-

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- There is no need to reduce column-3.
- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column-reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-4.

Cost(4)

= Cost(1) + Sum of reduction elements + M[A,D]

= 18 + 5 + 3

= 26

Thus, we have-

- Cost(2) = 36 (for Path A → B)
- Cost(3) = 25 (for Path A → C)
- Cost(4) = 26 (for Path A → D)

We choose the node with the lowest cost.

Since cost for node-3 is lowest, so we prefer to visit node-3.

Thus, we choose node-3 i.e. path **A → C**.

Step-03:

We explore the vertices B and D from node-3.

We now start from the cost matrix at node-3 which is- We now start from the cost matrix at node-3 which is-

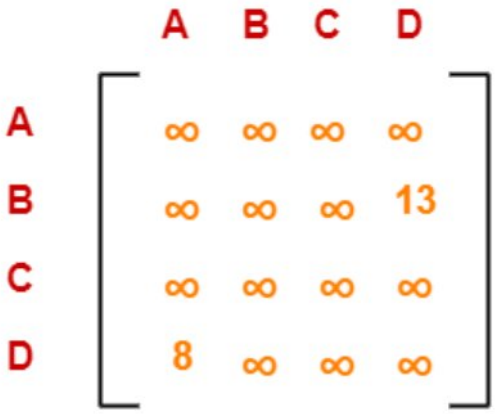
	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

Cost(3) = 25

Choosing To Go To Vertex-B: Node-5 (Path A → C → B)

- From the reduced matrix of step-02, M[C,B] = ∞
- Set row-C and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-



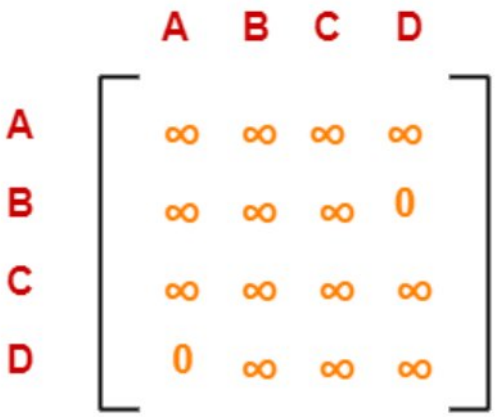
Now,

- We reduce this matrix.
- Then, we find out the cost of node-5.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- Reduce all the elements of row-2 by 13.
- We can not reduce row-3 as all its elements are ∞.
- Reduce all the elements of row-4 by 8.

Performing this, we obtain the following row-reduced matrix-



Column Reduction-

- There is no need to reduce column-1.
- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.
- There is no need to reduce column-4.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-5.

Cost(5)

= cost(3) + Sum of reduction elements + M[C,B]

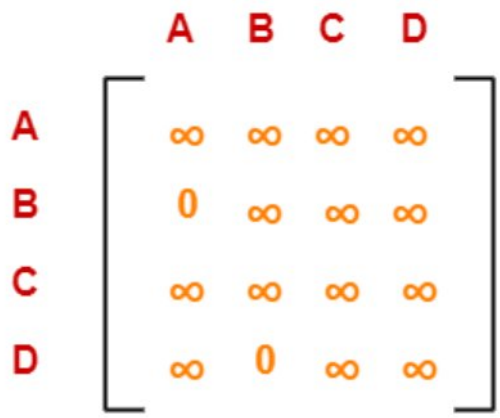
= 25 + (13 + 8) + ∞

= ∞

Choosing To Go To Vertex-D: Node-6 (Path A → C → D)

- From the reduced matrix of step-02, M[C,D] = ∞
- Set row-C and column-D to ∞
- Set M[D,A] = ∞

Now, resulting cost matrix is-



Now,

- We reduce this matrix.
- Then, we find out the cost of node-6.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- There is no need to reduce row-2.
- We can not reduce row-3 as all its elements are ∞.
- We can not reduce row-4 as all its elements are ∞.

Thus, the matrix is already row reduced.

Column Reduction-

- There is no need to reduce column-1.
- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.
- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-6.

Cost(6)

= cost(3) + Sum of reduction elements + M[C,D]

= 25 + 0 + 0

= 25

Thus, we have-

- Cost(5) = ∞ (for Path A → C → B)
- Cost(6) = 25 (for Path A → C → D)

We choose the node with the lowest cost.

Since cost for node-6 is lowest, so we prefer to visit node-6.

Thus, we choose node-6 i.e. path **C → D**.

Step-04:

We explore vertex B from node-6.

We start with the cost matrix at node-6 which is-

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	∞	0	∞	∞

Cost(6) = 25

Choosing To Go To Vertex-B: Node-7 (Path A → C → D → B)

- From the reduced matrix of step-03, M[D,B] = 0
- Set row-D and column-B to ∞
- Set M[B,A] = ∞

Now, resulting cost matrix is-

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	∞
C	∞	∞	∞	∞
D	∞	∞	∞	∞

Now,

- We reduce this matrix.
- Then, we find out the cost of node-7.

Row Reduction-

- We can not reduce row-1 as all its elements are ∞.
- We can not reduce row-2 as all its elements are ∞.
- We can not reduce row-3 as all its elements are ∞.
- We can not reduce row-4 as all its elements are ∞.

Column Reduction-

- We can not reduce column-1 as all its elements are ∞.
- We can not reduce column-2 as all its elements are ∞.
- We can not reduce column-3 as all its elements are ∞.

- We can not reduce column-4 as all its elements are ∞.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

All the entries have become ∞.

Now, we calculate the cost of node-7.

Cost(7)

= cost(6) + Sum of reduction elements + M[D,B]

= 25 + 0 + 0

= 25

Thus,

- Optimal path is: **A → C → D → B → A**
- Cost of Optimal path = **25 units**

0/1 Knapsack using Branch and Bound

- Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and Bound solve these problems relatively quickly.
- The branch-and-bound technique can be applied to the Knapsack problem. The Knapsack problem is about filling a knapsack with 'n' items. Each item is associated with a profit v1 and weight w1.
- The procedure for solving the knapsack problem using the branch-and-bound technique is:

  Step 1: Arrange the items such that v1/w1>=v2/w2>=……>=vn/wn.

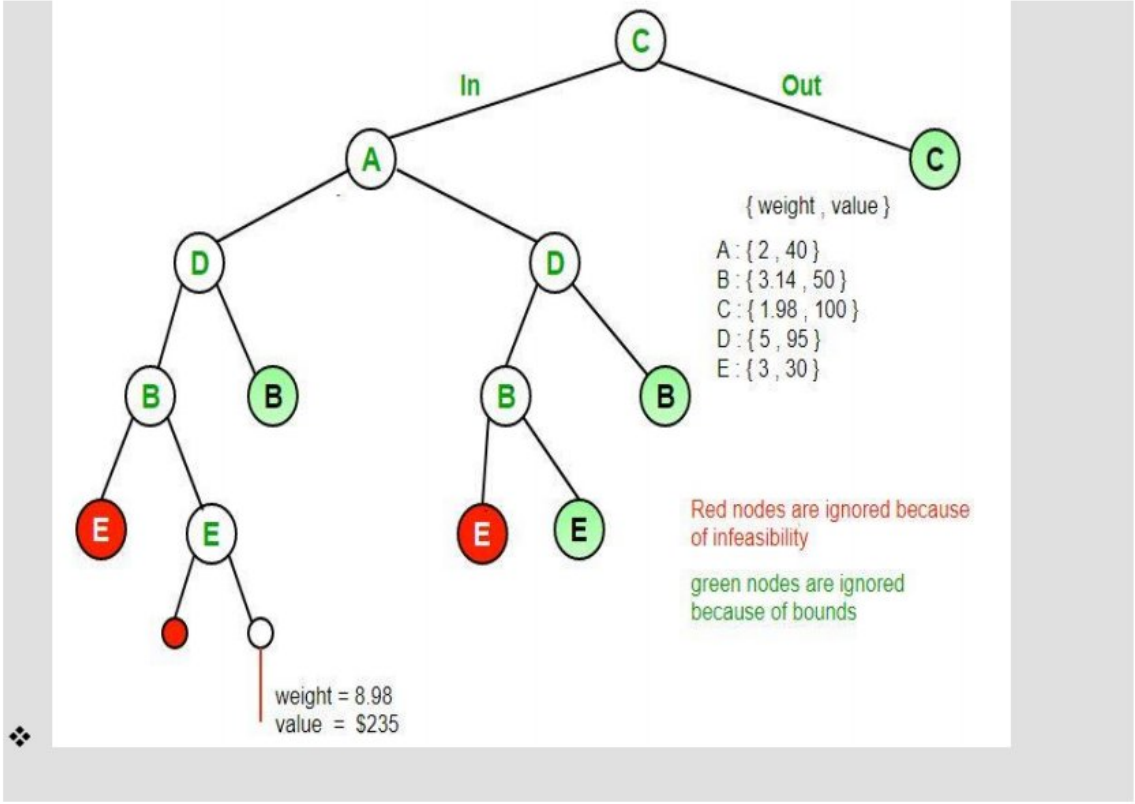
  Step 2: construct the state space tree as a binary tree. Take a node; branching to the left indicates that the item is included and branching to the right indicates that the item is excluded.

  Step 3: Compute the Lower bound as follows :

  ub=V+(W-w)*Vi+1/Wi+1.

  Here ,W is the capacity of the knapsack ,w is the weight of the item; And Vi+1 and Wi+1 are the value and weight of the next item,respectively.
- We can do better (than backtracking) if we know a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, we can simply ignore this node and its subtrees. So we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.

```
Input:
// First thing in every pair is weight of item
// and second thing is value of item
Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
              {5, 95}, {3, 30}};
Knapsack Capacity W = 10

Output:
The maximum possible profit = 235

Below diagram shows illustration. Items are
considered sorted by value/weight.
```

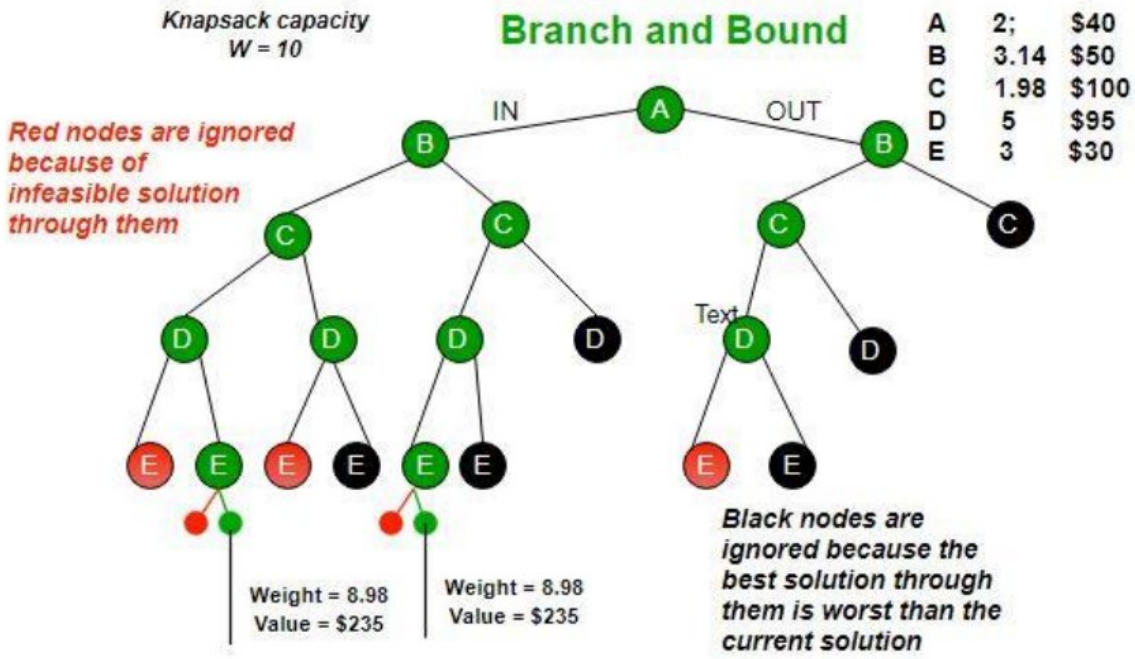


- **Example bounds used in below diagram are, A down can give $315, B down can $275, C down can $225, D down can $125 and E down can $30.**
- To check if a particular node can give us a better solution or not, we compute the optimal solution (through the node) using Greedy approach. If the solution computed by Greedy approach itself is more than the best so far, then we can't get a better solution through the node.

Complete Algorithm:

1. Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
2. Initialize maximum profit, maxProfit = 0
3. Create an empty queue, Q.
4. Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
5. Do following while Q is not empty.
   - Extract an item from Q. Let the extracted item be u.
   - Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
   - Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.

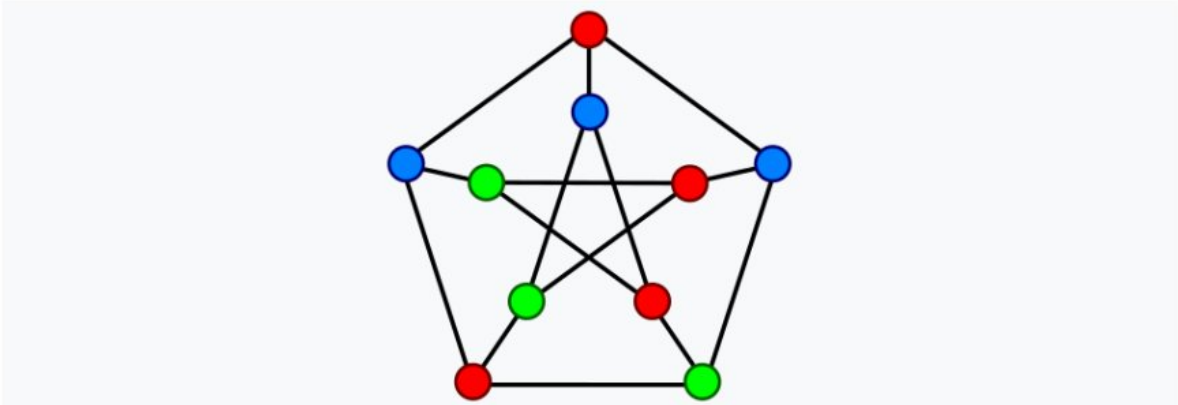
- Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.



GRAPH COLORING

In this problem, an undirected graph is given. There is also provided m colors. The problem is to find if it is possible to assign nodes with m different colors, such that no two adjacent vertices of the graph are of the same colors. If the solution exists, then display which color is assigned on which vertex.

Starting from vertex 0, we will try to assign colors one by one to different nodes. But before assigning, we have to check whether the color is safe or not. A color is not safe whether adjacent vertices are containing the same color.



A proper vertex coloring of the graph with 3 colors, the minimum number possible.

In graph theory, **graph coloring** is a special case of **graph labeling**; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices are of the same color; this is called a **vertex coloring**.

The problem can be solved as

• Assign colors to the vertices of a graph so that no adjacent vertices share the same color – Vertices i, j are adjacent if there is an edge from vertex i to vertex j.

• Find all m-colorings of a graph – Find all ways to color a graph with at most m colors. – m is called chromatic number

• Assign colors to the vertices of a graph so that no adjacent vertices share the same color – Vertices i, j are adjacent if there is an edge from vertex i to vertex j.

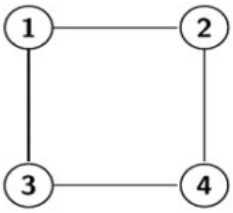
• Find all m-colorings of a graph – Find all ways to color a graph with at most m colors. – m is called chromatic number

Graph coloring Them-Coloring problem Finding all ways to color an undirected graph using at most m different colors, so that no two adjacent vertices are the same color.

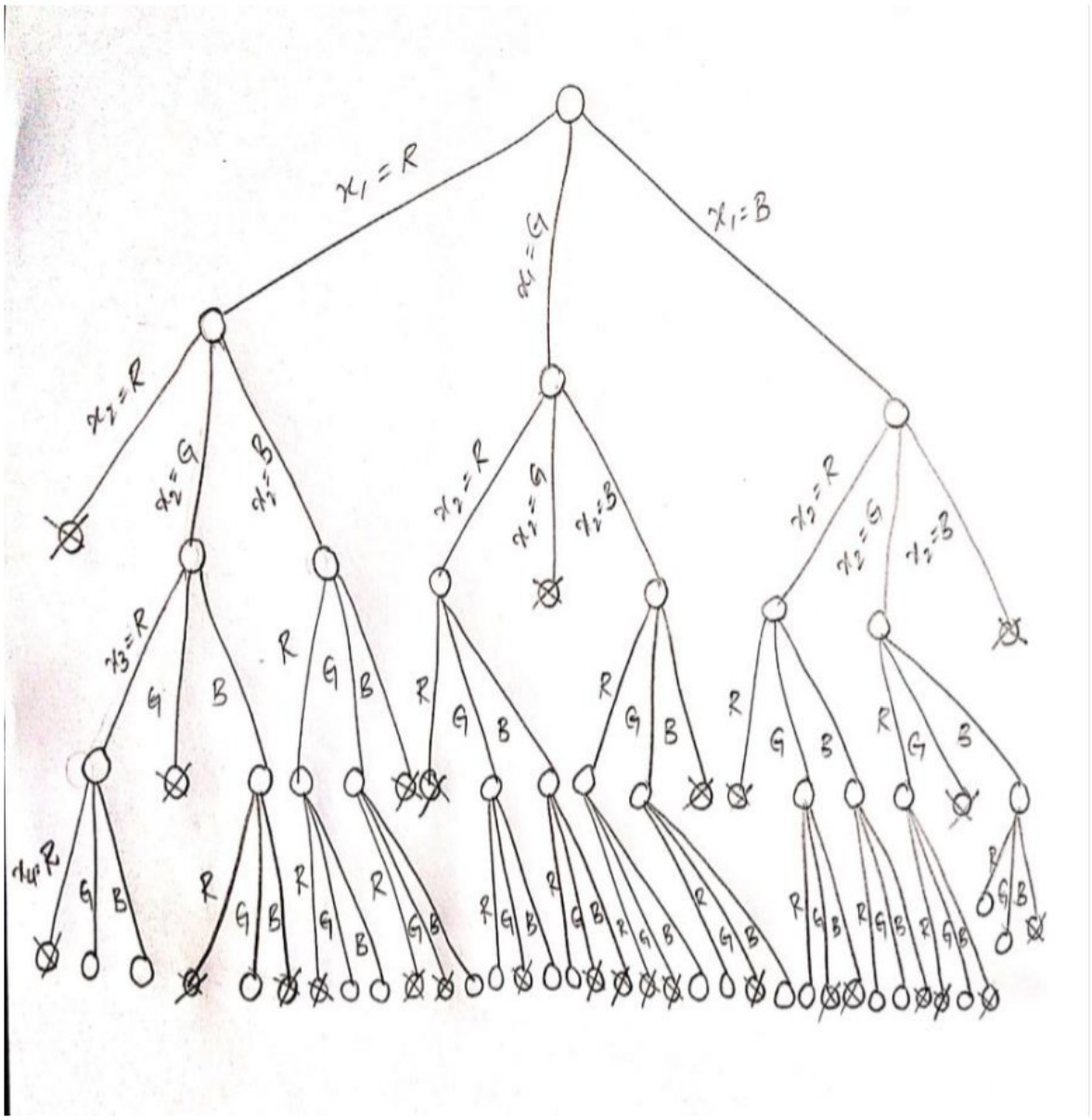
Usually the m-Coloring problem consider as a unique problem for each value of m.

GRAPH COLORING

Consider a graph having four vertices. Now we have to color this graph with given three colors red, green and blue.



m = 3 , that means number of colors



GRAPH COLORING

The possible combinations by using three colors are

RGRG	RGRB	RGBG
RBRG	RBRB	RBGB
GRGR	GRGB	GRBR
GBRB	GBGR	GBGB
BRGR	BRBR	BRBG
BGRG	BGBR	BGBG

Applications of graph coloring:

- Making schedule or time table.
- Mobile radio frequency assignment.
- Sudoku.
- Register allocation.
- Bipartite graph.
- Map coloring.s

GRAPH COLORING

0/1 Knapsack Problem using Branch and Bound

- In knapsack problem, consider 'n' no of objects each object having a profit 'p_i', weight 'w_i' and a knapsack capacity of 'm'.
- The objective is to place this objects 'n' into the knapsack 'm' to get maximum profit i.e., $\sum p_i x_i$ is maximum where x_i is to check whether the object is placed into knapsack or not.

i.e., $x_i = 1$ – object is placed or included.

$x_i = 0$ – object is not placed or not included.

Knapsack problem is maximization problem but whereas Branch and Bound is a minimization problem. By using minimization problem finding the solution to the maximization problem.

To solve this problem first the process is converted into negatives. So, whenever the positive profits are converted into negative profits, we are getting minimization problem.

Branch and Bound is solved in two ways:

1. Least Cost (LC)
2. First in First Out (FIFO)

0/1 Knapsack Problem using LC Branch and Bound:

Least Cost means at each stage having two decisions i.e., consider the objects or not.

Example:

Consider objects n = 4 and capacity m = 15

$(P_1, P_2, P_3, P_4) = (10,10,12,18)$

$(W_1, W_2, W_3, W_4) = (2,4,6,9)$

Step 1: Convert the profit into negatives

$(P_1, P_2, P_3, P_4) = (-10, -10, -12, -18)$

Step 2: Calculate Lower Bound and Upper Bound for each and every node.

- In Lower Bound fractions are allowed and it is represented by **ĉ**, where as in Upper Bound fractions are not allowed and represented by **û.**

Node 1:

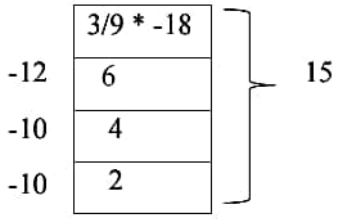
Given Knapsack having capacity 15, object 1 weight is 2 placed into knapsack getting profit

-10.

After placing first object remaining space is 13 so place object 2 into knapsack getting profit -10 and object 3 is placed into the knapsack getting profit -12.

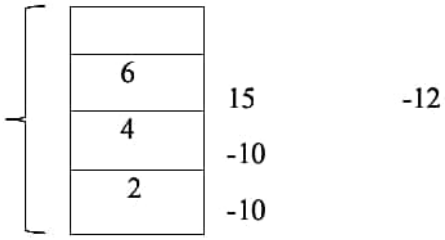
After placing object 3 remaining space is 3 but object 4 weight is 9 so complete object is not placed into the knapsack because given capacity is 15.

To calculate lower bound, fractions are allowed i.e., $\left\lceil \frac{\text{remaining space}}{\text{weight of the object}} * \text{profit} \right\rceil$



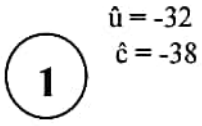
Lower Bound = -32-6 = -38

Similarly, Calculate Upper Bound for node 1 place object 1 getting profit -10, object 2 getting profit -10, object 3 getting profit -12 and remaining space is only 3 but object 4 weight is 9. So, object 4 is not placed into the knapsack.

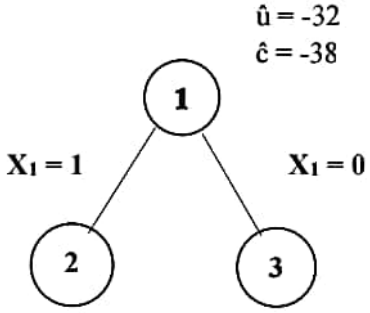


Upper Bound = -32

- In LB the space is not free, but in UB if there is a space, ignore the space.

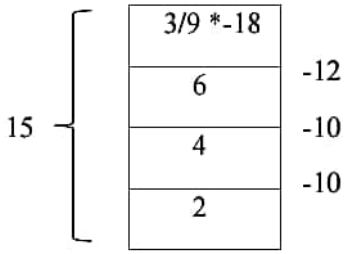


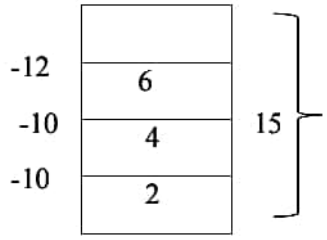
- From node 1, check whether object 1 is placed into the knapsack or not.



Node 2:

In node 2, $X_1 = 1$ i.e., object 1 should be placed 1st in the knapsack. After placing the 1st object, if any space is available place remaining objects into knapsack.





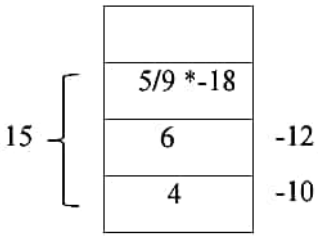
Lower Bound = -32-6 = -38

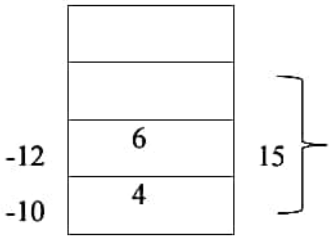
Upper Bound = -32

Node 3:

In node 3, $X_1 = 0$ i.e., first object is not placed into the knapsack. Place 2nd object getting profit -10, 3rd object getting profit -12 and remaining space is 5 so, (5/9) * -18 =

-10.

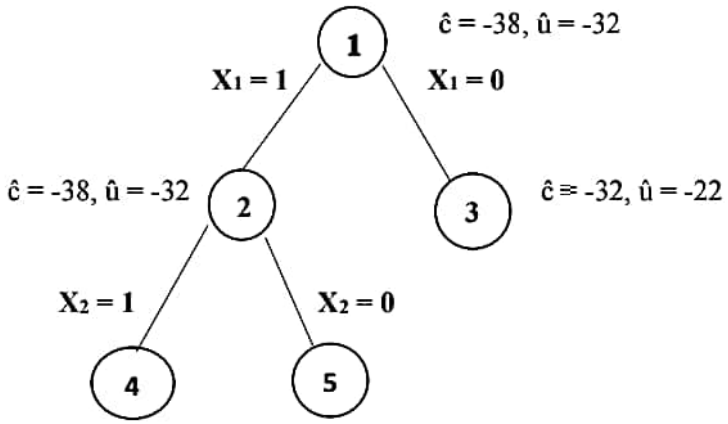




Lower Bound = -22-10 ≅ -32

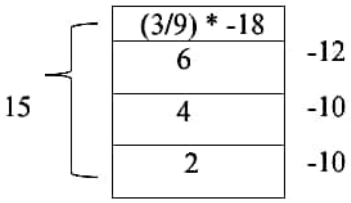
Upper Bound ≅ -22

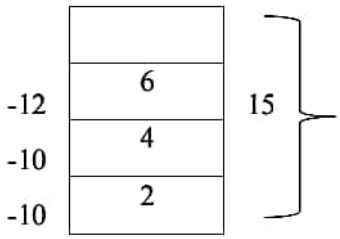
Consider minimum lower bound among node 2 and 3. Here node 2 has LB = -38 and node 3 has LB ≅ -32. consider node 2 because -38 is smaller than -32.



Node 4:

In node 4, $X_1 = 1$, $X_2 = 1$ i.e., compulsory 1st and 2nd object should be placed into the knapsack. If any space available then only place remaining objects.



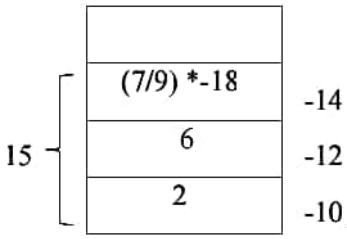


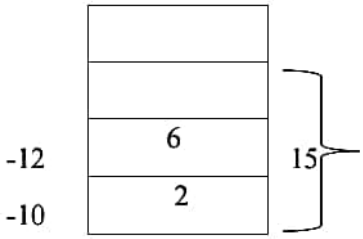
Lower Bound = -32-6 = -38

Upper Bound = -32

Node 5:

In node 5 $X_1 = 1$, $X_2 = 0$ i.e., 1st object is placed into knapsack and ignore the 2nd object and moves to next object.

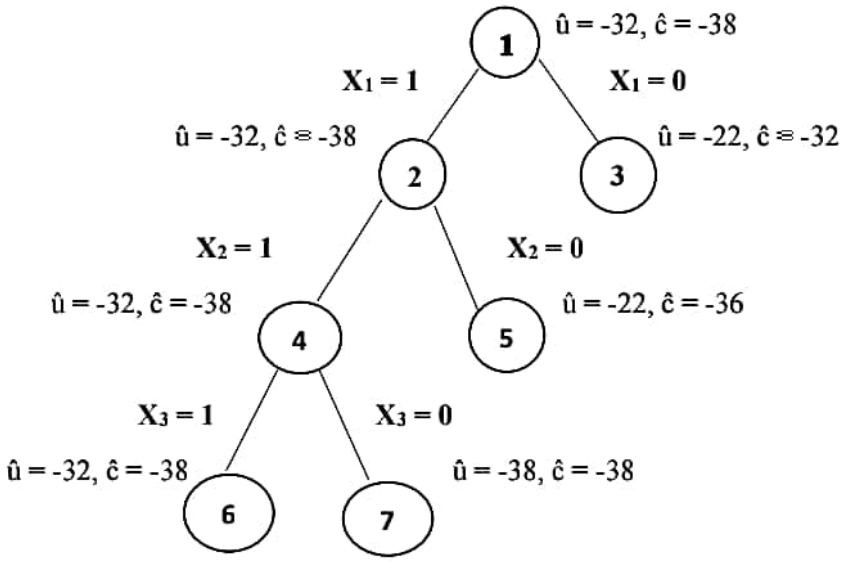




Lower Bound = -22 – 14 = -36

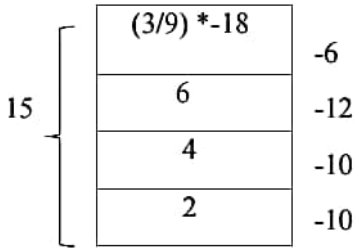
Upper Bound = -22

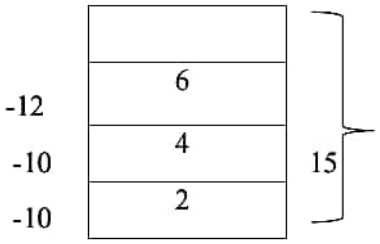
- Among nodes 4 and node 5, node 4 having minimum lower bound = -32



Node 6:

In node 6 $X_1 = 1$, $X_2 = 1$, $X_3 = 1$ i.e., 1st ,2nd and 3rd objects are placed into the knapsack.



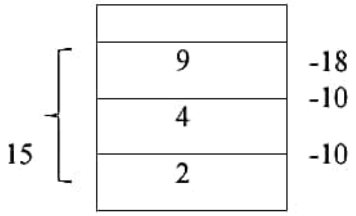


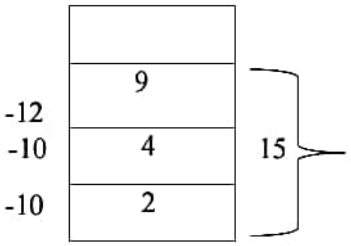
Lower Bound = -32 – 6 = -38

Upper Bound = -32

Node 7:

In node 7 $X_1 = 1$, $X_2 = 1$, $X_3 = 0$ i.e., 1st and 2nd object are placed into the knapsack. Ignore 3rd object, if any space is available place 4th object.

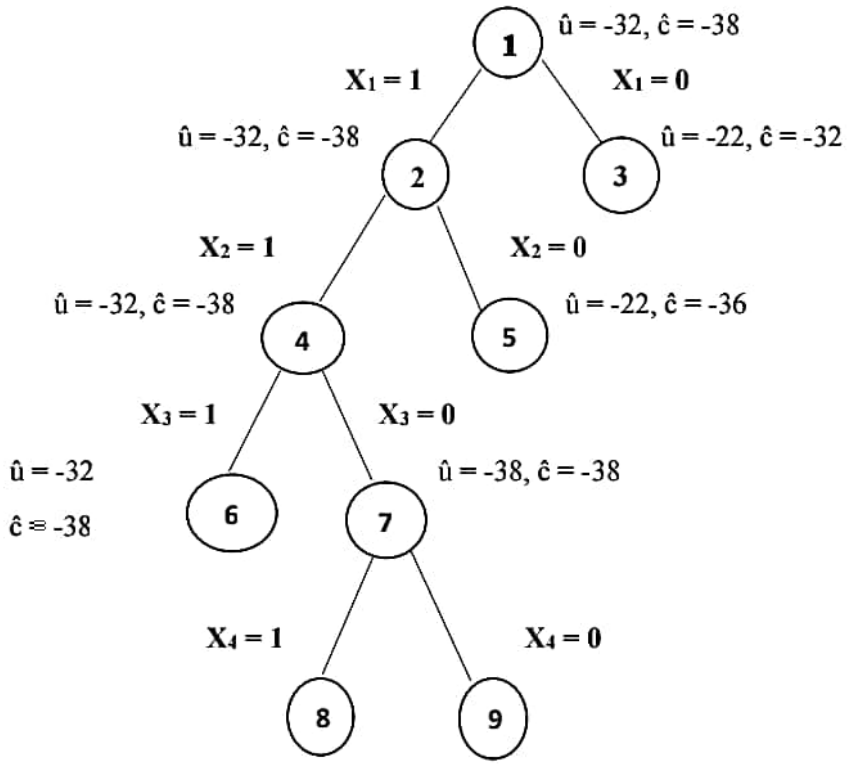




Lower Bound = -38

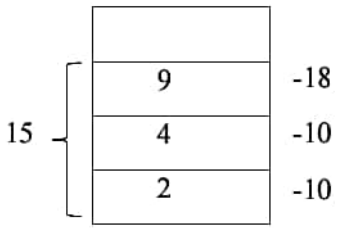
Upper Bound = -38

Among node 6 & 7, both node 6 & 7 lower bound are equal. If both lower bounds are equal then consider lower upper bound. Here. node 7 having lower upper bound = -38.

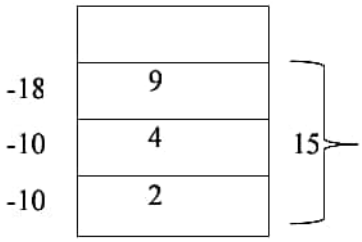


Node 8:

In node 8 $X_1 = 1$, $X_2 = 1$, $X_3 = 0$, $X_4 = 1$ i.e., 1st, 2nd and 4th objects are placed into the knapsack. Ignore the 3rd object.



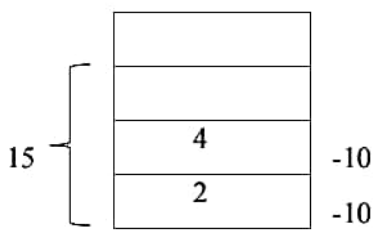
Lower Bound = -38

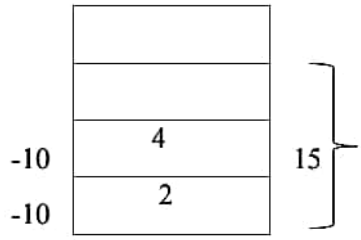


Upper Bound = -38

Node 9:

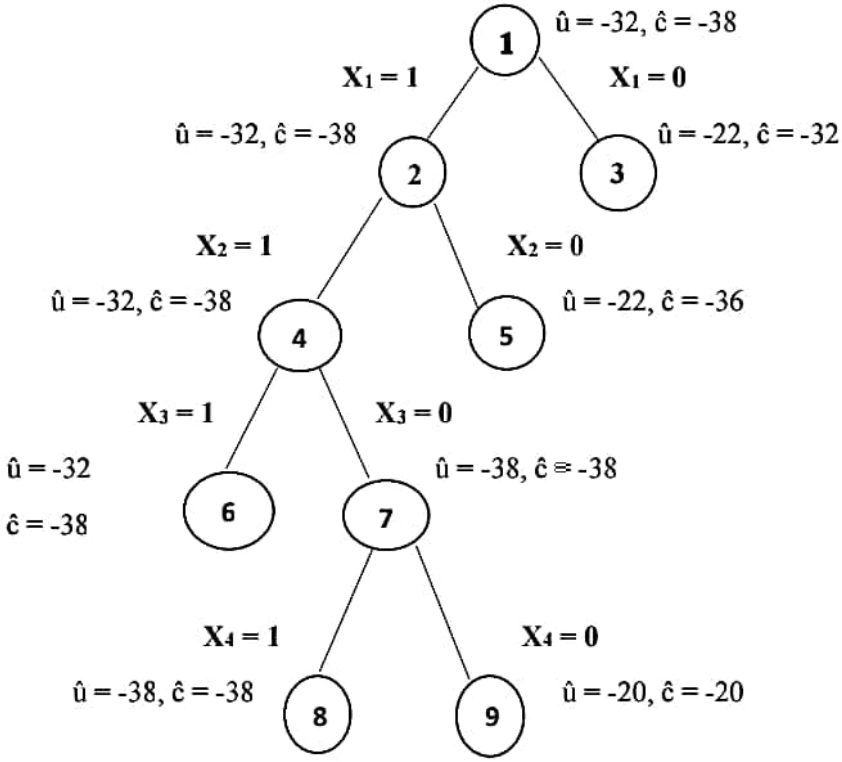
In node 9 $X_1 = 1$, $X_2 = 1$, $X_3 = 0$, $X_4 = 0$ i.e., 1st and 2nd objects are placed into the knapsack. Ignore the 3rd and 4th object and if place is available then simply ignore.





Lower Bound = -20

Upper Bound = -20



- Among nodes 8 and node 9, node 8 having minimum lower bound = -38. So, select node 8 and ignore node 9.
- Therefore, Knapsack instances are $(X_1, X_2, X_3, X_4) = (1, 1, 0, 1)$ and

  $(P_1, P_2, P_3, P_4) = (-10, -10, 0, -18)$
- Maximum profit = (-10) + (-10) + 0 + (-18) = -38. Since branch and bound is minimization problem so total profit = 38.

DESIGN AND ANALYSIS OF ALGORITHMS

(DAA)

UNIT – 6

NP-Hard and NP-Complete Problems:

Basic Concepts

Non Deterministic Algorithms

NP Hard and NP Complete Classes

Cook's Theorem.

SHRI VISHNU ENGINEERING COLLEGE FOR WOMEN : : BHIMAVARAM

(AUTONOMOUS)

UNIT – 6

NP Hard and NP Complete

Polynomial Time	**Exponential Time**
Linear search-n	0/1 knapsack - 2^n
Binary search-log n	Travelling sp -2^n
Insertion sort- n^2	sum of subsets – 2^n
Merge sort – n log n	Graph colouring – 2^n
Matrix multiplication -n^3	Hamiltonian cycle -2^n

Actually this topic is research topic. The algorithms are categorized into two types:

1. Polynomial time taking algorithms
2. Exponential time taking algorithms

Here we have linear search algorithm which takes O(n) time and faster than that is binary search algorithm which takes O(log n) time , and for sorting algorithm merge sort which takes O(n log n) which takes less time and we need faster algorithms which is faster than this one.

Similarly for these exponential time algorithms we need polynomial time taking algorithms. All these algorithms which taking exponential time we want faster and easy method to solve them in just polynomial time because 2^n, n^n , etc.. are much bigger than polynomial time.

As these exponential algorithms are very time consuming so we need polynomial time for them, this is our requirement.

This is a research area and the person from computer science and mathematics can solve these problems so people have been doing research on this one but there is no particular solution found so far yet for solving these problems in polynomial time.

Then ,when the research work is going fruitless we want something such that whatever the work we done should be useful so these are the guidelines or framework made for doing research on these type of problems i.e., exponential type problems and that framework is NP Hard and NP Complete.

Let us see the basic idea behind that so there are two points on which the entire topic is based on these two points only.

1. When you are unable to solve these exponential problems, that you are unable to get solution in polynomial time algorithm for them at least you do the work such that you try to show the similarities between them so that if one problem is solved the other can also be solved. We will not be doing research work individually on each and every problem like one is working on knapsack problem and the other on travelling sp problem.

Let us combine them collectively and put some effort such that if one problem is solved all the other problems should be solved. So far that we have to show the relationship between them.

"Try to relate the problem either solve it or at least related"

2. When we are unable to write algorithm for them that is deterministic, why don't we write non deterministic algorithm.

Deterministic Algorithm:

Each and every statement how it works we know it clearly. We are sure how they work i.e., we know the working of the algorithm. This type of algorithms are called deterministic algorithms.

Non deterministic Algorithms

Here, we don't know how the algorithm works. Then how can we write the algorithms which doesn't know? While writing an algorithm some of the statements may not be figuring out,how to make them polynomial so leave them blanks and say this is non deterministic and when we know how it has to be filled ,we fill up that statements.

In a non-deterministic algorithm also most of the statements may be deterministic, but some of the statements are nondeterministic

By writing this non-deterministic algorithms we can preserve our research work so that in future somebody else work on this same problem. He/she can make the nondeterministic part as deterministic. This is the main idea in writing non deterministic algorithms.

Let us take an example :

Non deterministic Search Algorithm

```
Algorithm Nsearch (A, n, key)
{
        J=choice();
        If(key=A[i])
        {
                Write(j);
                Success();
        }
        else{
                write(0);
                failure();
        }
}
```

Assume that all these statements (choice(), success(), failure()) takes just one unit of time.

This algorithm is about searching a key from an array of N elements.

While seeing this algorithm we can say that the algorithm takes O(1) time.this is the constant time algorithm for searching and the fastest searching algorithm is binary search that takes O(log n) time. And this algorithm takes O(1) and is really faster , we need this one but it is non deterministic.

In the above algorithm choice() gives the index of key element and we are checking the key element is present at index j ,then search is successful otherwise failure and if key element is not present at index 'j' then key element is not present in the array then search is unsuccessful

Now the question is how this choice() knows that the key element is present at j th index.that's what it is non deterministic.Once we know the position of key element in constant time then we will fill up the choice() line. Choice() line is like a blank statement for us,now once we fill this we have an algorithm of O(1) for searching.

Example: we have an array .

10	8	5	9	4	2

If key=5. Then choice() will give directly index 2.but how we doesn't know

Similarly, we write non deterministic algorithms for exponential problems with this we define two classes

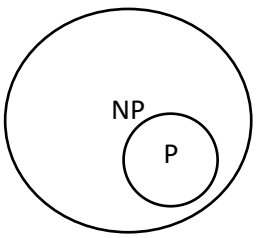
P and NP classes:

P - ' P ' is a set of those deterministic algorithms which are taking polynomial time

Example linear search, binary search, bubble sort etc…

NP -These algorithms are non deterministic but they take polynomial time. We actually write these algorithms for exponential time taking algorithms.

NP- Non deterministic Polynomial time taking algorithms.



' P ' is shown as subset of ' NP ' which means the deterministic algorithms that we know today were the part of non deterministic algorithms.

We completed the first thing that is writing non deterministic polynomial time taking algorithms. How to write is completed

Now the second part if you are unable to solve , atleast show the relationship between them such that if one problem it shoulb be easy for solving the other problems also in polynomial time .we will not work on each problem individually so how to relate them that is the next thing we are going to discuss

So for relating them together we need some problem as a base problem and the base problem is satisfiability problem

satisfiability:-

- In polynomial time, if you unable to solve at least show the relationship between them such that if one problem is solved, then it should be easy for solving other problems also in polynomial time.
- We will not work on each and every problem individually.so for relating them together we need some problem as base problem.
- The base problem is Satisfiability.
- Let x_1, x_2, x3…. xn denotes Boolean variables
- .Let xi denotes the relation of x_i.
- A literal is either a variable or its negation
- .A formula in the prepositional calculus is an expression that can be constructed using literals and the operators and Λ or v.
- A clause is a formula with at least one positive literal.
- The satisfiability problem is to determine if a formula is true for some assignment of truth values to the variables.
- It is easy to obtain a polynomial time non determination algorithm that terminates successfully if and only if a given prepositional formula E (x_1, x_2……x_n) is satiable.
- Such an algorithm could proceed by simply choosing (non deterministically) one of the 2 n possible assignments of truth values to (x_1, x_2…x_n) and verify that E (x_1, x_2…x_n) is true for that assignment.

The satisfiability problem:-

The logical formula:

X_1 v X_2 v X_3

& - x_1

& - x_2

The assignment:

$x_1 \leftarrow$ F, $x_2 \leftarrow$ F, $x_3 \leftarrow$ T

will make the above formula ∀≠true.

(-x_1, -x_2, x_3) represents $x_1 \leftarrow$ F, $x_2 \leftarrow$ F, $x_3 \leftarrow$ T

- If there is at least one assignment which satisfies a formula, then we say that this formula is satisfiable; otherwise, it is unsatisfiable.

- An unsatisfiable formula:

x_1 v x_2

& x1 v –x

& -x_1 v x_2

& -x_1 v -x_2

• Definition of the satisfiability problem:

Given a Boolean formula, determine whether this formula is satisfiable or not.

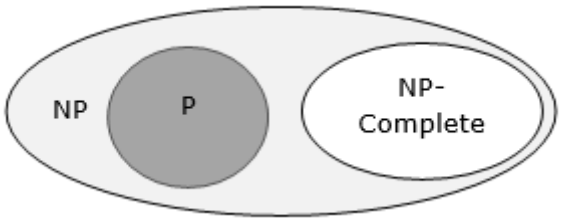
• A literal: x_i or -x_i

• A clause: x_1 v x_2 v -x_3 Ci

• A formula: conjunctive normal form (CNF) C1& C2& … & Cm

NP hard and NP complete Classes:

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness:

A language **B** is ***NP-complete*** if it satisfies two conditions

- **B** is in NP
- Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof

To prove ***TSP is NP-Complete***, first we have to prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus, ***TSP belongs to NP***.

Secondly, we have to prove that ***TSP is NP-hard***. To prove this, one way is to show that ***Hamiltonian cycle*** $\leq_p$ ***TSP*** (as we know that the Hamiltonian cycle problem is NPcomplete).

Assume ***G = (V, E)*** to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph $G' = (V, E')$, where

$$E'=\{(i ,j): i , j \in V \text{ and } i \neq j$$

Thus, the cost function is defined as follows −

$$t(i, j)= \{0 \text{ if}(I ,j)\in E$$

$$1 \quad \text{Otherwise,}$$

Now, suppose that a Hamiltonian cycle ***h*** exists in ***G***. It is clear that the cost of each edge in ***h*** is **0** in G' as each edge belongs to ***E***. Therefore, ***h*** has a cost of **0** in G'. Thus, if graph ***G*** has a Hamiltonian cycle, then graph G' has a tour of **0** cost.

Conversely, we assume that G' has a tour h' of cost at most **0**. The cost of edges in E' are **0** and **1** by definition. Hence, each edge must have a cost of **0** as the cost of h' is **0**. We therefore conclude that h' contains only edges in E.

We have thus proven that G has a Hamiltonian cycle, if and only if G' has a tour of cost at most **0**. TSP is NP-complete.

NP-HARD GRAPH AND SCHEDULING PROBLEMS

Some NP-hard Graph Problems:

The strategy to show that a problem L_2 is NP-hard is

I. Pick a problem L_1 already known to be NP-hard.

II. Show how to obtain an instance I^1 of L_2 from any instance I of L_1 such that from the solution of I^1 - We can determine (in polynomial deterministic time) the solution to instance I of L_1.

III. Conclude from (ii) that $L_1 \alpha L_2$.

IV. Conclude from (i), (ii), and the transitivity of α that Satisfiabilityα L1
$L_1 \alpha L_2$
Therefore, Satisfiability L_2
and L_2 is NP-hard

1. Chromatic Number Decision Problem (CNP)

- A coloring of a graph G = (V, E) is a function f: V → {1,2, …, k} ∀ I ϵ V.
- If (U, V) ϵ E then f(u) ≠f(v).
- The CNP is to determine if G has a coloring for a given K.
- Satisfiability with at most three literals per clause α chromatic number problem. Therefore, CNP is NP-hard.

2. Directed Hamiltonian Cycle (DHC): -

- Let G = (V, E) be a directed graph and length n = 1V1
- The DHC is a cycle that goes through every vertex exactly once and then returns to the starting vertex.
- The DHC problem is to determine if G has a directed Hamiltonian Cycle.

Theorem: CNF (Conjunctive Normal Form) satisfiabilityα DHC.
Therefore, DHC is NP-hard.

3. Problem (TSP) :Travelling Salesperson Decision:

• The problem is to determine if a complete directed graph G = (V, E) with edge costs C(u, v) has a tour of cost at most M.

Theorem: Directed Hamiltonian Cycle (DHC) αTSP

• But from problem (2) satisfiabilityα DHC

Therefore, Satisfiability α TSP
and TSP is NP-hard.

NP-HARD SCHEDULING PROBLEMS

Sum of subsets: -

The problem is to determine if A= {a_1, a_2 ……., a_n} (a1, a_2 …….,a_n are positive integers) has a subset S that sums to a given integer M. 2.

Schedule identical processors: -

- Let P_i $1 \leq i \leq m$ be identical processors or machines Pi.
- Let J_i $1 \leq i \leq n$ be n jobs.
- Jobs $J_i$requires t_i processing time.
- A schedule S is an assignment of jobs to processors.
- For each job Ji, S specifies the time intervals and the processors on which this job is to be processed.
- A job cannot be processed by more than one processor at any given time. The problem is to find a minimum finish time non-preemptive schedule.
- The finish time of S is FT(S) = max $\{T_i\}$ $1 \leq i \leq m$.
- Where T_i is the time at which processor Pi finishes processing all jobs (or job segments) assigned to it.

Cook's Theorem:

- The Cook-Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem whether a Boolean formula is satisfiable.
- The theorem is named after Stephen Cook and Leonid Levin.
- An important consequence of this theorem is that if there exists a deterministic polynomial time algorithm for solving Boolean satisfiability, then every NP problem can be solved by a deterministic polynomial time algorithm.

The work shows that Cook's theorem is the origin of the loss of non-determinism in terms of the equivalence of the two definitions of NP, the one defining NP as the class of problems solvable by a nondeterministic Turing machine in polynomial time, and the other defining N P as the class of problems verifiable by a deterministic Turing machine in polynomial time. Therefore, we argue that fundamental difficulties in understanding P versus NP lie firstly at cognitionlevel, then logic level.

Following are the four theorems by Stephen Cook −

Theorem-1

If a set **S** of strings is accepted by some non-deterministic Turing machine within polynomial time, then **S** is P-reducible to {DNF tautologies}.

Theorem-2

The following sets are P-reducible to each other in pairs (and hence each has the same polynomial degree of difficulty): {tautologies}, {DNF tautologies}, D3, {sub-graph pairs}.

Theorem-3

- For any $T_Q(k)$ of type **Q**, TQ(k)k√(logk)2TQ(k)k(logk)2 is unbounded
- There is a $T_Q(k)$ of type **Q** such that TQ(k)⩽2k(logk)2TQ(k)⩽2k(logk)2

Theorem-4

If the set S of strings is accepted by a non-deterministic machine within time $T(n) = 2^n$, and if $T_Q(k)$ is an honest (i.e. real-time countable) function of type **Q**, then there is a constant **K**, so **S** can be recognized by a deterministic machine within time $T_Q(K8^n)$.

- First, he emphasized the significance of polynomial time reducibility. It means that if we have a polynomial time reduction from one problem to another, this ensures that any polynomial time algorithm from the second problem can be converted into a corresponding polynomial time algorithm for the first problem.
- Second, he focused attention on the class NP of decision problems that can be solved in polynomial time by a non-deterministic computer. Most of the intractable problems belong to this class, NP.
- Third, he proved that one particular problem in NP has the property that every other problem in NP can be polynomially reduced to it. If the satisfiability problem can be

solved with a polynomial time algorithm, then every problem in NP can also be solved in polynomial time. If any problem in NP is intractable, then satisfiability problem must be intractable. Thus, satisfiability problem is the hardest problem in NP.

- Fourth, Cook suggested that other problems in NP might share with the satisfiability problem this property of being the hardest member of NP.

In order to prove this, we require a uniform way of representing NP problems. Remember that what makes a problem NP is the existence of a polynomial-time algorithm more specifically, a Turing machine for checking candidate certificates.

Assume, then, that we are given an NP decision problem D. By the definition of NP, there is a polynomial function P and a Turing machine M which, when given any instance I of D, together with a candidate certificate c, will check in time no greater than P(n), where n is the length of I, whether or not c is a certificate of I

Let us assume that M has q states numbered 0, 1, 2, . . . , q − 1, and a tape alphabet a1, a2, . . . , as. We shall assume that the operation of the machine is governed by the functions T, U, and D. We shall further assume that the initial tape is inscribed with the problem instance on the squares 1, 2, 3, . . . , n, and the putative certificate on the squares −m, . . . , −2, −1. Square zero can be assumed to contain a designated separator symbol. We shall also assume that the machine halts scanning square 0, and that the symbol in this square at that stage will be a1 if and only if the candidate certificate is a true certificate. Note that we must have $m \leq P(n)$. This is because with a problem instance of length n the computation is completed in at most P(n) steps; during this process, the Turing machine head cannot move more than P(n) steps to the left of its starting point. We define some atomic propositions with their intended interpretations as follows:

1. For i = 0, 1, . . . , P(n) and j = 0, 1, . . . , q − 1, the proposition Qij says that after i computation steps, M is in state j.

2. For i = 0, 1, . . . , P(n), j = −P(n), . . . , P(n), and k = 1, 2, . . . , s, the proposition Sijk says that after i computation steps, square j of the tape contains the symbol ak.

3. i = 0, 1, . . . , P(n) and j = −P(n), . . . , P(n), the proposition Tij says that after i computation steps, the machine M is scanning square j of the tape.

Next, we define some clauses to describe the computation executed by M:

1. At each computation step, M is in at least one state. For each i = 0, . . . , P(n) we have the clause

$$Qi0 \vee Qi1 \vee \cdots \vee Qi(q-1),$$

giving (P(n) + 1)q = O(P(n)) literals altogether.

2. At each computation step, M is in at most one state. For each i = 0, . . . , P(n) and for each pair j, k of distinct states, we have the clause

$$\neg(Qij \wedge Qik),$$

giving a total of q(q − 1)(P(n) + 1) = O(P(n)) literals altogether

3. At each step, each tape square contains at least one alphabet symbol. For each i = 0, . . . , P(n) and −P(n) ≤ j ≤ P(n) we have the clause

$$Sij1 \vee Sij2 \vee \cdots \vee Sijs,$$

giving (P(n) + 1)(2P(n) + 1)s = O(P(n) 2) literals altogether.

4. At each step, each tape square contains at most one alphabet symbol. For each i = 0, . . . , P(n) and −P(n) ≤ j ≤ P(n), and each distinct pair ak, al of symbols we have the clause

$$\neg(Sijk \wedge Sijl),$$

giving a total of (P(n) + 1)(2P(n) + 1)s(s − 1) = O(P(n) 2) literals altogether

5. At each step, the tape is scanning at least one square. For each i = 0, . . . , P(n), we have the clause

$$Ti(-P(n)) \vee Ti(1-P(n)) \vee \cdots \vee Ti(P(n)-1) \vee TiP(n),$$

giving (P(n) + 1)(2P(n) + 1) = O(P(n) 2) literals altogether.

6. At each step, the tape is scanning at most one square. For each i = 0, . . . , P(n), and each distinct pair j, k of tape squares from −P(n) to P(n), we have the clause

$$\neg(Tij \wedge Tik),$$

giving a total of 2P(n)(2P(n) + 1)(P(n) + 1) = O(P(n) 3) literals.

7. Initially, the machine is in state 1 scanning square 1. This is expressed by the two clauses

$$Q01,\ T01,$$

giving just two literals.

8. The configuration at each step after the first is determined from the configuration at the previous step by the functions T, U, and D defining the machine M. For each i = 0, . . . , P(n), −P(n) ≤ j ≤ P(n), k = 0, . . . , q − 1, and l = 1, . . . , s, we have the clauses

$$Tij \wedge Qik \wedge Sijl \rightarrow Q(i+1)T(k,l)$$
$$Tij \wedge Qik \wedge Sijl \rightarrow S(i+1)jU(k,l)$$
$$Tij \wedge Qik \wedge Sijl \rightarrow T(i+1)(j+D(k,l))$$
$$Sijk \rightarrow Tij \vee S(i+1)jk$$

The fourth of these clauses ensures that the contents of any tape square other than the currently scanned square remains the same (to see this, note that the given clause is equivalent to the formula Sijk ∧ ¬Tij → S(i+1)jk). These clauses contribute a total of (12s + 3)(P(n) + 1)(2P(n) + 1)q = O(P(n) 2) literals.

9. Initially, the string ai1 ai2 . . . ain defining the problem instance I is inscribed on squares 1, 2, . . . , n of the tape. This is expressed by the n clauses

$$S01i1\ ,\ S02i2\ ,\ \ldots\ ,\ S0nin\ ,$$

a total of n literals.

10. By the P(n)th step, the machine has reached the halt state, and is then scanning square 0, which contains the symbol a1. This is expressed by the three clauses

$$QP(n)0,\ SP(n)01,\ TP(n)0,$$

giving another 3 literals.

Altogether the number of literals involved in these clauses is $O(P(n)^3)$ (in working this out, note that q and s are constants, that is, they depend only on the machine and do not vary with the problem instance; thus they do not contribute to the growth of the the number of literals with increasing problem size, which is what the O notation captures for us). It is thus clear that the procedure for setting up these clauses, given the original machine M and the instance I of problem D, can be accomplished in polynomial time.

We must now show that we have succeeded in converting D into SAT. Suppose first that I is a positive instance of D. This means that there is a certificate c such that when M is run with inputs c, I, it will halt scanning symbol a1 on square 0. This means that there is some sequence of symbols that can be placed initially on squares $-P(n), \ldots, -1$ of the tape so that all the clauses above are satisfied. Hence those clauses constitute a positive instance of SAT.

Conversely, suppose I is a negative instance of D. In that case there is no certificate for I, which means that whatever symbols are placed on squares $-P(n), \ldots, -1$ of the tape, when the computation halts the machine will not be scanning a1 on square 0. This means that the set of clauses above is not satisfiable, and hence constitutes a negative instance of SAT.

Thus from the instance I of problem D we have constructed, in polynomial time, a set of clauses which constitute a positive instance of SAT if and only I is a positive instance of D. In other words, we have converted D into SAT in polynomial time. And since D was an arbitrary NP problem it follows that any NP problem can be converted to SAT in polynomial time.