DESIGN AND ANALYSIS OF ALGORITHMS

UNIT-I

INTRODUCTION:
Algorithm,pseudocode for expressing algorithms,performance analysis- Time complexity and space complexity, asymptotic noatation- O notation Omega notation and Theta notation and little on notation,probabilistic analysis,amortized complexity.
DIVIDE AND CONQUER:
General Method, applications-Binary search,merge sort, quick sort, strassen's matrix multiplication.

What is an Algorithm?

Just to understand better, let us consider the problem of preparing an omelet. For preparing omelet, general steps which we follow are:

1) Get the frying pan.
2) Get the oil.
    a. Do we have oil?
        i. If yes, put it in the pan.
        ii. If no, do we want to buy oil?
            1. If yes, then go out and buy.
            2. If no, we can terminate.
3) Turn on the stove.
4) Etc...

The above steps says that, for a given problem (in this case, preparing an omelet), giving step by step procedure for solving it. So, the definition of algorithm can be given as:

An algorithm is the step-by-step instructions to a given problem.

Why Analysis of Algorithms?

If we want to go from city *"A"* to city *"B"*. There can be many ways of doing this: by flight, by bus, by train and also by cycle. Depending on the availability and convenience we choose the one which suits us. Similarly, in computer science there can be multiple algorithms exist for solving the same problem (for example, sorting problem has lot of algorithms like insertion sort, selection sort, quick sort and many more). Algorithm analysis helps us determine which of them is efficient in terms of time and space consumed.

Goal of Analysis of Algorithms?

The goal of *analysis of algorithms* is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer's effort etc.)

What is Running Time Analysis?

It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is number of elements in the input and depending on the problem type the input may be of different types. In general, we encounter the following types of inputs.

- Size of an array
- Polynomial degree
- Number of elements in a matrix
- Number of bits in the binary representation of the input
- Vertices and edges in a graph

How to Compare Algorithms?

To compare algorithms, let us define some *objective measures*.

Execution times?

Not a good measure as execution times are specific to a particular computer.

Number of statements executed?

Not a good measure since the number of statements varies with the programming language as well as the style of the individual programmer.

Ideal Solution?

Let us assume that we expressed running time of given algorithm as a function of the input size n (i.e., $f(n)$). We can compare these different functions corresponding to running times and this kind of comparison is independent of machine time, programming style, etc..

What is Rate of Growth?

The rate at which the running time increases as a function of input is called rate of growth. Let us assume that you went to a shop for buying a car and a cycle. If your friend sees you there and asks what you are buying then in general we say *buying a car*. This is because cost of car is too big compared to cost of cycle (approximating the cost of cycle to cost of car).

Total Cost = cost_of_car + cost_of_cycle
Total Cost ≈ cost_of_car (approximation)

For the above example, we can represent the cost of car and cost of cycle in terms of function and for a given function we ignore the low order terms that are relatively insignificant (for large value of input size, n). As an example in the below case, n^4, $2n^2$, $100n$ and 500 are the individual costs of some function and we approximate it to n^4. Since, n^4 is the highest rate of growth.

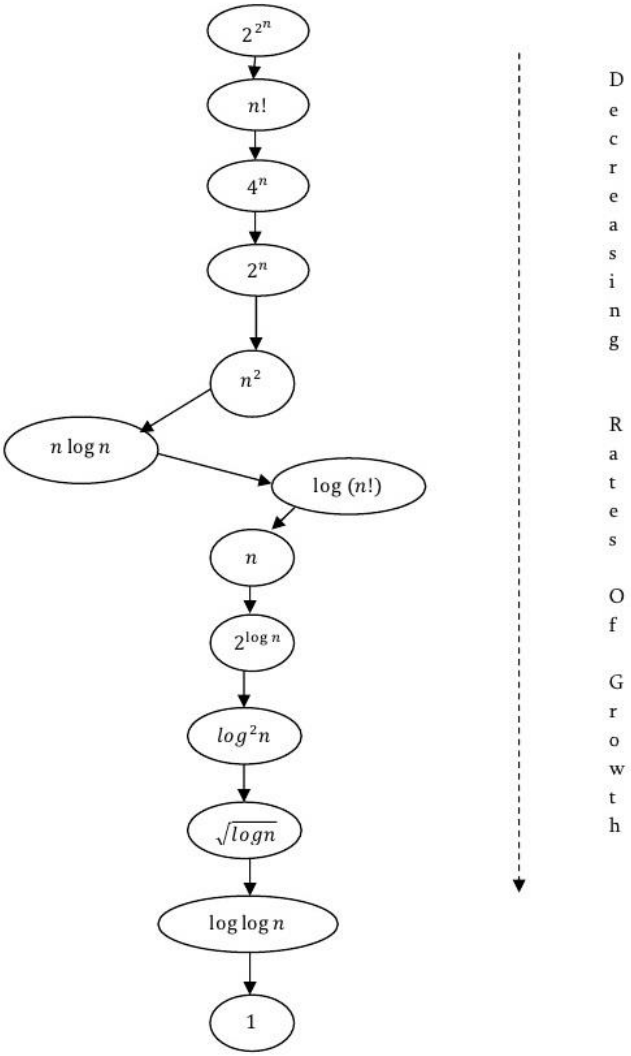
$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

Commonly used Rate of Growths

Below is the list of rate of growths which come across.

Time complexity	Name	Example
$O(1)$	Constant	Adding an element to the front of a linked list
$O(logn)$	Logarithmic	Finding an element in a sorted array
$O(n)$	Linear	Finding an element in an unsorted array
$O(nlogn)$	Linear Logarithmic	Sorting n items by 'divide-and-conquer'-Mergesort
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	The Towers of Hanoi problem

Below diagram shows the relationship between different rates of growth.



Types of Analysis

Assume that we have an algorithm for a problem and want to know on what inputs the algorithm is taking less time (performing well) and on what inputs the algorithm is taking huge time.

We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions: one for case where it is taking the less time and other for case where it is taking the more time. In general the first

case is called the best case for the algorithm and second case is called the worst case for the algorithm.

To analyze an algorithm we need some kind of syntax and that forms the base for asymptotic analysis/notation. There are three types of analysis:

- **Worst case**
  - Defines the input for which the algorithm takes huge time.
  - Input is the one for which the algorithm runs the slower.
- **Best case**
  - Defines the input for which the algorithm takes lowest time.
  - Input is the one for which the algorithm runs the fastest.
- **Average case**
  - Provides a prediction about the running time of the algorithm
  - Assumes that the input is random

$$\text{Lower Bound} <= \text{Average Time} <= \text{Upper Bound}$$

For a given algorithm, we can represent best case, worst case, average case analysis in the form of expressions. As an example, let $f(n)$ be the function which represents the given algorithm. Then, the following expression for best case and worst case.

$f(n) = n^2 + 500$, for worst case
$f(n) = n + 100n + 500$, for best case

Similarly, for average case too. The expression defines the inputs with which the algorithm takes the average running time (or memory).

Asymptotic notation?

Having the expressions for best case, average case and worst case, to find the upper bounds and lower bounds for each of them: for all the three cases we need to identify the upper bound, lower bounds. In order to represent these upper bound and lower bounds we need some syntax and that creates the base for following discussion.

For the below discussions, let us assume that the given algorithm is represented in the form of function $f(n)$.

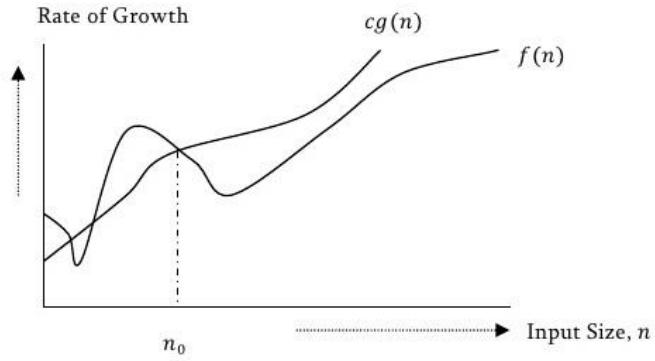
Big-O notation

This notation gives the *tight* upper bound of the given function. Generally we represent it as $f(n) = O(g(n))$. That means, at larger values of n, the upper bound of $f(n)$ is $g(n)$.

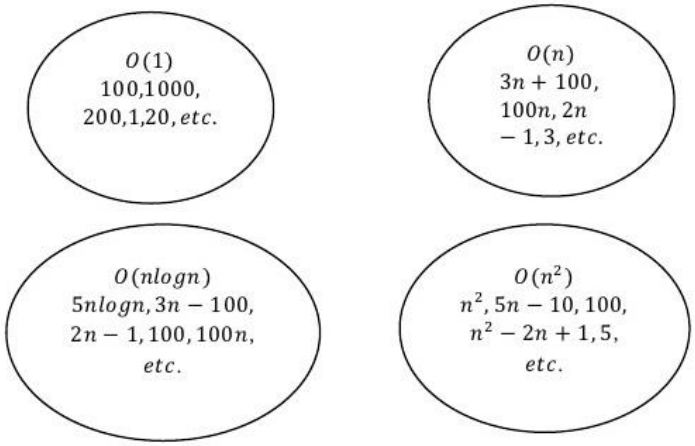
For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 is $g(n)$. That means $g(n)$ gives the maximum rate of growth for $f(n)$ at larger values of n.

Now, let us see the O −notation with little more detail. O −notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give some rate of growth $g(n)$ which is greater than given algorithms rate of growth $f(n)$.

In general, we do not consider lower values of n. That means the rate of growth at lower values of n is not important. In the below figure, n_0 is the point from which we consider the rate of growths for a given algorithm. Below n_0 the rate of growths may be different.



Big-O Visualization



$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$. For example, $O(n^2)$ includes $O(1), O(n), O(nlogn)$ etc..

Note: Analyze the algorithms at larger values of n only. What this means is, below n_0 we do not care for rates of growth.

Big-O examples

Example-1 Find upper bound for $f(n) = 3n + 8$

Solution: $3n + 8 \leq 4n$, for all $n \geq 1$
$\therefore 3n + 8 = O(n)$ with c = 4 and $n_0 = 8$

Example-2 Find upper bound for $f(n) = n^2 + 1$

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$
$\therefore n^2 + 1 = O(n^2)$ with c = 2 and $n_0 = 1$

Example-3 Find upper bound for $f(n) = n^4 + 100n^2 + 50$

Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 1$
$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with c = 2 and $n_0 = 100$

Example-4 Find upper bound for $f(n) = 2n^3 - 2n^2$

Solution: $2n^3 - 2n^2 \leq 2n^3$, for all $n \geq 1$
$\therefore 2n^3 - 2n^2 = O(2n^3)$ with c = 2 and $n_0 = 1$

Example-5 Find upper bound for $f(n) = n$

Solution: $n \leq n^2$, for all $n \geq 1$
$\therefore n = O(n^2)$ with c = 1 and $n_0 = 1$

Example-6 Find upper bound for $f(n) = 410$

Solution: $410 \leq 410$, for all $n \geq 1$
$\therefore 100 = O(1)$ with c = 1 and $n_0 = 1$

No uniqueness?

There is no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n + 5 = O(n^2)$. For this function there are multiple n_0 and c.

Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n^2$ for all $n \geq 5, n_0 = 5$ and $c = 101$ is a solution.

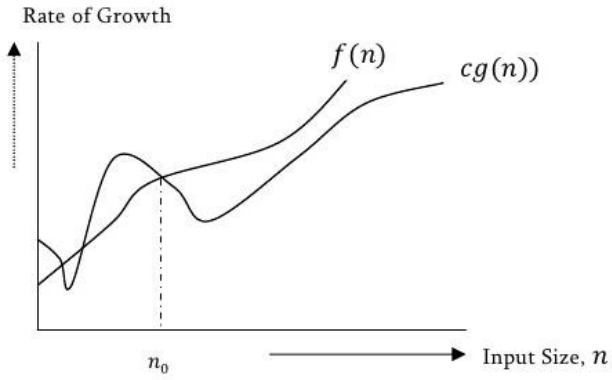
Solution2: $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$ for all $n \geq 1, n_0 = 1$ and $c = 105$ is also a solution.

Omega-Ω notation

Similar to above discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as $f(n) = \Omega(g(n))$. That means, at larger values of n, the tighter lower bound of $f(n)$ is $g(n)$.

For example, if $f(n) = 100n^2 + 10n + 50$, $g(n)$ is $\Omega(n^2)$.

The Ω notation as be defined as $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic lower bound for $f(n)$. $\Omega(g(n))$ is the set of functions with smaller or same order of growth as $f(n)$.



Ω Examples

Example-1 Find lower bound for $f(n) = 5n^2$

Solution: $\exists c$, n_0 Such that: $0 \leq cn \leq 5n^2 \Rightarrow cn \leq 5n^2 \Rightarrow c = 1$ and $n_0 = 1$
$\therefore 5n^2 = \Omega(n)$ with $c = 1$ and $n_0 = 1$

Example-2 Prove $f(n) = 100n + 5 \neq \Omega(n^2)$

Solution: $\exists$ c, n_0 Such that: $0 \leq cn^2 \leq 100n + 5$
$100n + 5 \leq 100n + 5n\ (\forall n \geq 1) = 105n$

$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$

Since n is positive $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$

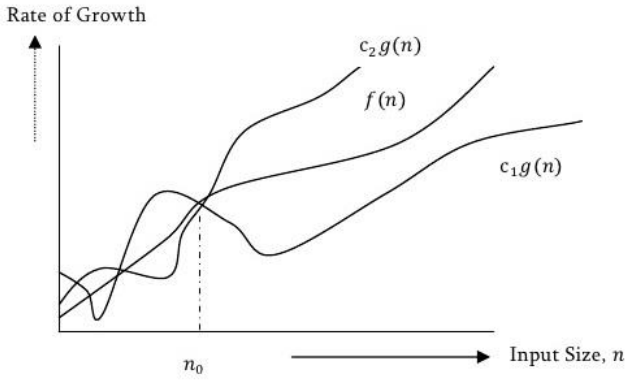
$\Rightarrow$ Contradiction: n cannot be smaller than a constant

Example-3 $n = \Omega(2n),\ n^3 = \Omega(n^2),\ n = \Omega(logn)$

Theta-θ notation

This notation decides whether the upper and lower bounds of a given function are same or not. The average running time of algorithm is always between lower bound and upper bound. If the upper bound (O) and lower bound (Ω) gives the same result then θ notation will also have the same rate of growth. As an example, let us assume that $f(n) = 10n + n$ is the expression. Then, its tight upper bound $g(n)$ is $O(n)$. The rate of growth in best case is $g(n) = O(n)$. In this case, rate of growths in best case and worst are same. As a result, the average case will also be same.

Note: For a given function (algorithm), if the rate of growths (bounds) for O and Ω are not same then the rate of growth θ case may not be same.



Now consider the definition of θ notation. It is defined as $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight bound for $f(n)$. $\theta(g(n))$ is the set of functions with the same order of growth as $g(n)$.

θ Examples

Example-1 Find θ bound for $f(n) = \frac{n^2}{2} - \frac{n}{2}$

Solution: $\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$, for all, n≥ 1

$\therefore \frac{n^2}{2} - \frac{n}{2} = \theta(n^2)$ with $c_1 = 1/5, c_1 = 1$ and $n_0 = 1$

Example-2 Prove $n \neq \theta(n^2)$

Solution: $c_1 n^2 \leq n \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq 1/c_1$

$\therefore n \neq \Theta(n^2)$

Example-3 Prove $6n^3 \neq \theta(n^2)$

Solution: $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq c_2/6$

$\therefore 6n^3 \neq \Theta(n^2)$

Example-4 Prove $n \neq \theta(\log n)$

Solution: $c_1 \log n \leq n \leq c_2 \log n \Rightarrow c_2 \geq \frac{n}{\log n}, \forall n \geq n_0$ – Impossible

Important Notes

For analysis (best case, worst case and average) we try to give upper bound (O) and lower bound (Ω) and average running time (θ). From the above examples, it should also be clear that, for a given function (algorithm) getting upper bound (O) and lower bound (Ω) and average running time (θ) may not be possible always.

For example, if we are discussing the best case of an algorithm, then we try to give upper bound (O) and lower bound (Ω) and average running time (θ).

In the remaining chapters we generally concentrate on upper bound (O) because knowing lower bound (Ω) of an algorithm is of no practical importance and we use θ notation if upper bound (O) and lower bound (Ω) are same.

Little Oh Notation

The little Oh is denoted as o. It is defined as : Let, f(n) and g(n) be the non negative functions then

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

such that f(n)= o(g(n)) i.e f of n is little Oh of g of n.

f(n) = o(g(n)) if and only if f(n) = o(g(n)) and f(n) ≠ θ (g(n))

Guidelines for asymptotic analysis?

There are some general rules to help us in determining the running time of an algorithm. Below are few of them.

1) **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
// executes n times
for (i=1; i<=n; i++)
{
    m = m + 2; // constant time, c
}
```

Total time = a constant $c \times n = cn = O(n)$.

2) **Nested loops:** Analyze from inside out. Total running time is the product of the sizes of all the loops.

```
//outer loop executed n times
for (i=1; i<=n; i++)
{
    // inner loop executed n times
    for (j=1; j<=n; j++)
    {
            k = k+1; //constant time
    }
}
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

3) **Consecutive statements:** Add the time complexities of each statement.

```
x = x +1; //constant time

// executed n times
for (i=1; i<=n; i++)
```

```
{
    m = m + 2; //constant time
}

//outer loop executed n times
for (i=1; i<=n; i++)
{
    //inner loop executed n times
    for (j=1; j<=n; j++)
    {
            k = k+1; //constant time
    }
}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

4) **If-then-else statements:** Worst-case running time: the test, plus *either* the *then* part *or* the *else* part (whichever is the larger).

```
//test: constant
if (length ( ) != otherStack. length ( ) )
{
    return false; //then part: constant
}
else
{
    // else part: (constant + constant) * n
    for (int n = 0; n < length( ); n++)
    {
            // another if : constant + constant (no else part)
            if (!list[n].equals(otherStack.list[n]))
            //constant
            return false;
    }
}
```

Total time = $c_0 + c_1 + (c_2 + c_3) * n = O(n)$.

5) **Logarithmic complexity:** An algorithm is $O(logn)$ if it takes a constant time to cut the problem size by a fraction (usually by ½).

As an example let us consider the following program:

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

If we observe carefully, the value of i is doubling every time. That means, initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4, 8$ and so on.

Let us say the loop is executing some K times. That means at $K - th$ step $2^i = n$ and we come out of loop. So, if we take logarithm on both sides,

$log(2^i) = logn$
$ilog2 = logn$
$i = logn$ //if we assume base-2

So the total time = $O(logn)$.

Note: Similarly, for the below case also the worst case rate of growth is $O(logn)$. That means, the same discussion holds good for decreasing sequence also.

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

Commonly used logarithms and summations

Logarithms

$$\log x^y = y \log x$$
$$\log n = \log_e^n$$
$$\log xy = \log x + \log y$$
$$\log^k n = (\log n)^k$$
$$\log \log n = \log (\log n)$$
$$\log \frac{x}{y} = \log x - \log y$$
$$a^{\log_b^x} = x^{\log_b^a} \quad \log_b^x = \frac{\log_a^x}{\log_a^b}$$

Arithmetic series

$$\sum_{K=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Geometric series

$$\sum_{k=1}^{n} x^k = 1 + x + x^2 \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \log n$$

Other important formulae

$$\sum_{k=1}^{n} \log k \approx n \log n$$
$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p \approx \frac{1}{p+1} n^{p+1}$$

Divide and conquer

General Method

- In divide and conquer method, a given problem is,
  i) Divided into smaller subproblems.
  ii) These subproblems are solved independently.
  iii) Combining all the solutions of subproblems into a solution of the whole.
- If the subproblems are large enough then divide and conquer is reapplied.
- The generated subproblems are usually of same type as the original problem. Hence recurssive algorithms are used in divide and conquer strategy.

```
Algorithm DC(P)
{
  if P is too small then
  return solution of P.
  else
  {
       Divide (P) and obtain P1,P2,.....Pn
       where n ≥ 1
       Apply DC to each subproblem
       return combine (DC(P1),DC(P2)... DC(Pn));
  }
}
```

Master Theorem for Divide and Conquer

In all efficient divide and conquer algorithms we will divide the problem into subproblems, each of which is some part of the original problem, and then perform some additional work to compute the final answer. As an example, if we consider merge sort [for details, refer *Sorting* chapter], it operates on two problems, each of which is half the size of the original, and then uses $O(n)$ additional work for merging. This gives the running time equation:

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

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program or algorithm, first we try to find the recurrence relation for the problem. If the recurrence is of below form then we directly give the answer without fully solving it.

If the reccurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(nklog^p n)$, where $a \geq 1, b > 1, k \geq 0$ and p is a real number, then we can directly give the answer as:

1) If $a > b^k$, then $T(n) = \Theta(n^{\log_b^a})$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta(n^{\log_b^a} log^{p+1} n)$
   b. If $p = -1$, then $T(n) = \Theta(n^{\log_b^a} \text{loglogn})$
   c. If $p < -1$, then $T(n) = \Theta(n^{\log_b^a} s)$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

Problems on Master Theorem for Divide and Conquer

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Problem-1 $T(n) = 3T(n/2) + n^2$

Solution: $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-2 $T(n) = 4T(n/2) + n^2$

Solution: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 logn)$ (Master Theorem Case 2.a)

Problem-3 $T(n) = T(n/2) + 2^n$

Solution: $T(n) = T(n/2) + 2^n \Rightarrow \Theta(2^n)$ (Master Theorem Case 3.a)

Problem-4 $T(n) = 2^n T(n/2) + n^n$

Solution: $T(n) = 2^n T(n/2) + n^n$ => Does not apply (a is not constant)

Problem-5 $T(n) = 16T(n/4) + n$

Solution: $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-6 $T(n) = 2T(n/2) + n \log n$

Solution: $T(n) = 2T(n/2) + n\,logn \Rightarrow T(n) = \Theta(nlog^2n)$ (Master Theorem Case 2.a)

Problem-7 $T(n) = 2T(n/2) + n/logn$

Solution: $T(n) = 2T(n/2) + n/logn \Rightarrow T(n) = \Theta(n\text{loglogn})$ (Master Theorem Case 2.b)

Problem-8 $T(n) = 2T(n/4) + n^{0.51}$

Solution: $T(n) = 2T(n/4) + n^{0.51} \Rightarrow T(n) = O(n^{0.51})$ (Master Theorem Case 3.b)

Problem-9 $T(n) = 0.5T(n/2) + 1/n$

Solution: $T(n) = 0.5T(n/2) + 1/n$ => Does not apply ($a < 1$)

Problem-10 $T(n) = 6T(n/3) + n^2 \log n$

Solution: $T(n) = 6T(n/3) + n^2logn \Rightarrow T(n) = \Theta(n^2logn)$ (Master Theorem Case 3.a)

Problem-11 $T(n) = 64T(n/8) - n^2logn$

Solution: $T(n) = 64T(n/8) - n^2logn$ => Does not apply (function is not positive)

Problem-12 $T(n) = 7T(n/3) + n^2$

Solution: $T(n) = 7T(n/3) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.as)

Problem-13 $T(n) = 4T(n/2) + logn$

Solution: $T(n) = 4T(n/2) + logn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-14 $T(n) = T(n/2) + n(2 - cosn)$

Solution: $T(n) = T(n/2) + n(2 - cosn)$ => Does not apply. We are in Case 3, but the regularity condition is violated. (Consider n = 2Пk, where k is odd and arbitrarily large. For any such choice of n, you can show that $c >= 3/2$, thereby violating the regularity condition.)

Problem-15 $T(n) = 16T(n/4) + n!$

Solution: $T(n) = 16T(n/4) + n! \Rightarrow T(n) = \Theta(n!)$ (Master Theorem Case 3.a)

Problem-16 $T(n) = \sqrt{2}T(n/2) + logn$

Solution: $T(n) = \sqrt{2}T(n/2) + log\, n \Rightarrow T(n) = \Theta(\sqrt{n})$ (Master Theorem Case 1)

Problem-17 $T(n) = 3T(n/2) + n$

Solution: $T(n) = 3T(n/2) + n \Rightarrow T(n) = \Theta(n^{log3})$ (Master Theorem Case 1)

Problem-18 $T(n) = 3T(n/3) + \sqrt{n}$

Solution: $T(n) = 3T(n/3) + \sqrt{n} \Rightarrow T(n) = \Theta(n)$ (Master Theorem Case 1)

Problem-19 $T(n) = 4T(n/2) + cn$

Solution: $T(n) = 4T(n/2) + cn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-20 $T(n) = 3T(n/4) + nlogn$

Solution: $T(n) = 3T(n/4) + nlogn \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 3.a)

Problem-21 $T(n) = 3T(n/3) + n/2$

Solution: $T(n) = 3T(n/3) + n/2 \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 2.a)

Problem-26 What is the complexity of the below program:

```
void function(int n)
{
    int i, j, k , count =0;
    for(i=n/2; i<=n; i++)
        for(j=1; j + n/2<=n; j= j++)
            for(k=1; k<=n; k= k * 2)
                count++;
}
```

Solution: Let us consider the comments in the following function.

```
void function(int n)
{
    int i, j, k , count =0;
    //outer loop execute n/2 times
    for(i=n/2; i<=n; i++)
        //Middle loop executes n/2 times
        for(j=1; j + n/2<=n; j= j++)
            //outer loop execute logn times
            for(k=1; k<=n; k= k * 2)
                count++;
}
```

The complexity of the above function is $O(n^2logn)$.

Problem-27 What is the complexity of the below program:

```
void function(int n)
{
    int i, j, k , count =0;
    for(i=n/2; i<=n; i++)
        for(j=1; j<=n; j= 2 * j)
            for(k=1; k<=n; k= k * 2)
                count++;
}
```

Solution: Let us consider the comments in the following function.

```
void function(int n)
{
    int i, j, k , count =0;
    //outer loop execute n/2 times
    for(i=n/2; i<=n; i++)
        //Middle loop executes logn times
        for(j=1; j<=n; j= 2 * j)
            //outer loop execute logn times
            for(k=1; k<=n; k= k*2)
                count++;
}
```

The complexity of the above function is $O(nlog^2n)$.

Problem-28 Find the complexity of the below program.

```
function( int n )
{
    if ( n == 1 ) return ;
    for( i = 1 ; i <= n ; i + + )
    {
        for( j= 1 ; j <= n ; j + + )
        {
            print("*" ) ;
            break;
        }
    }
}
```

Solution: Let us consider the comments in the following function.

```
function( int n )
```

```
{
        //constant time
        if ( n == 1 ) return ;
        //outer loop execute n times
        for( i = 1 ; i <= n ; i + + )
        {
                // inner loop executes only time due to break statement.
                for( j= 1 ; j <= n ; j + + )
                {
                        print("*" ) ;
                        break;
                }
        }
}
```

The complexity of the above function is $O(n)$. Even though the inner loop is bounded by n, but due to the break statement it is executing only once.

Problem-29 Write a recursive function for the running time $T(n)$ of the function function, whose code is below. Prove using the iterative method that $T(n) = \Theta(n^2)$.

```
function( int n )
{
        if ( n == 1 ) return ;
        for( i = 1 ; i <= n ; i + + )
                for( j = 1 ; j <= n ; j + + )
                        print("*" ) ;
        function( n-3 );
}
```

Solution: Consider the comments in below function:

```
function (int n)
{
        //constant time
        if ( n == 1 ) return ;

        //outer loop execute n times
        for( i = 1 ; i <= n ; i + + )
                //inner loop executes n times
                for( j = 1 ; j <= n ; j + + )
```

```
                    //constant time
                    print("*" ) ;
            function( n-3 );
    }
```

The recurrence for this code is clearly $T(n) = T(n-3) + cn^2$ for some constant $c > 0$ since each call prints out n^2 asterisks and calls itself recursively on n - 3. Using the iterative method we get:

$$T(n) = T(n-3) + cn^2$$

Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(n^3)$.

Problem-30 Determine Θ bounds for the following recurrence relation:

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

Solution: Using Divide and Conquer master theorem, we get $O(n\,log^2 n)$.

Problem-31 Determine Θ bounds for the following recurrence

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

Solution: Substituting in the recurrence equation, we get:

$$T(n) \leq c1 * \frac{n}{2} + c2 * \frac{n}{4} + c3 * \frac{n}{8} + cn$$
$$\leq k * n, \text{ where k is a constant.}$$

Problem-32 Determine Θ bounds for the following recurrence relation:

$$T(n) = T(\lceil n/2 \rceil) + 7$$

Solution: Using Master Theorem we get $\Theta(logn)$.

Running time of following program?

```
function(int n)
{
        for( i = 1 ; i ≤ n/3 ; i + + )
                for( j = 1 ; j ≤ n ; j += 4 )
                        print( "*" ) ;
}
```

Solution: Consider the comments in below function:

```
function(int n)
{
        // this loops executes n/3 times
        for( i = 1 ; i ≤ n/3 ; i + + )
                // this loops executes n/4 times
                for( j = 1 ; j ≤ n ; j += 4)
                        print( "*" ) ;
}
```

```
{
        //constant time
        if ( n == 1 ) return ;
        //outer loop execute n times
        for( i = 1 ; i <= n ; i + + )
        {
                // inner loop executes only time due to break statement.
                for( j= 1 ; j <= n ; j + + )
                {
                        print("*" ) ;
                        break;
                }
        }
}
```

The complexity of the above function is $O(n)$. Even though the inner loop is bounded by n, but due to the break statement it is executing only once.

Problem-29 Write a recursive function for the running time $T(n)$ of the function function, whose code is below. Prove using the iterative method that $T(n) = \Theta(n^2)$.

```
function( int n )
{
        if ( n == 1 ) return ;
        for( i = 1 ; i <= n ; i + + )
                for( j = 1 ; j <= n ; j + + )
                        print("*" ) ;
        function( n-3 );
}
```

Solution: Consider the comments in below function:

```
function (int n)
{
        //constant time
        if ( n == 1 ) return ;

        //outer loop execute n times
        for( i = 1 ; i <= n ; i + + )
                //inner loop executes n times
                for( j = 1 ; j <= n ; j + + )
```

Binary Search

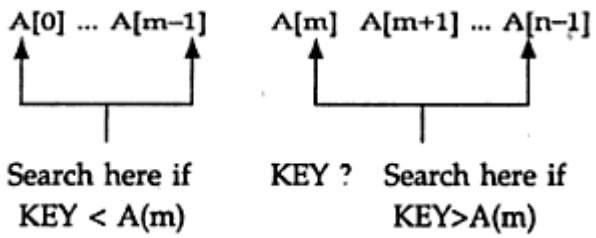
Binary search is an efficient searching method. While searching the elements using this method the most essential thing is that the elements in the array should be sorted one.

An element which is to be searched from the list of elements stored in array A[0...n – 1] is called KEY element.

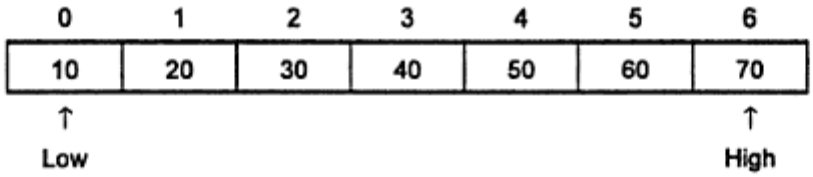
Let A[m] be the mid element of array A. Then there are three conditions that needs to be tested while searching the array using this method

1. If KEY = A[m] then desired element is present in the list.
2. Otherwise if KEY < A[m] then search the left sublist
3. Otherwise if KEY > A[m] then search the right sublist.

This can be represented as



Consider a list of elements stored in array A as,



The KEY element (i.e. the element to be searched) is 60.

Now to obtain middle element we will apply a formula :

$$m = (low + high)/2$$

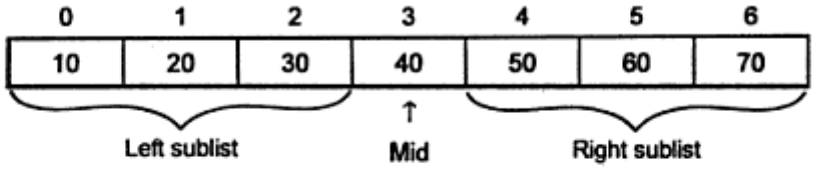
$$m = (0 + 6)/2$$

$$m = 3$$

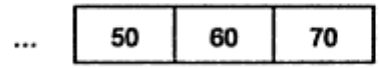
Then Check $A[m] \stackrel{?}{=} KEY$

i.e. $A[3] \stackrel{?}{=} 60$ NO $A[3] = 40$ and $40 < 60$

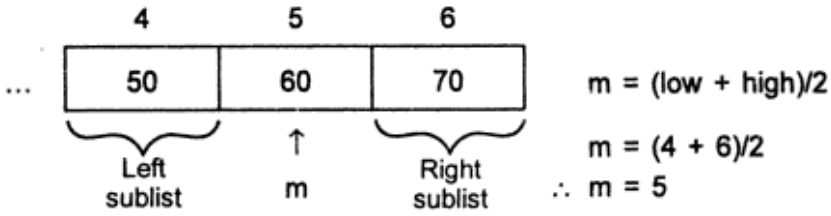
∴ Search the right sublist.



The right sublist is



Now we will again divide this list and check the mid element.



is $A[m] \stackrel{?}{=} KEY$

i.e. $A[5] \stackrel{?}{=} 60$ Yes, i.e. the number is present in the list.

Thus we can search the desired number from the list of elements.

Algorithm

```
Algorithm BinSearch(A[0…n-1],KEY)
//Problem Description: This algorithm is for searching the
//element using binary search method.
//Input: An array A from which the KEY element is to be
//searched.
//Output: It returns the index of an array element if it is
//equal to KEY otherwise it returns -1
low:=0
high:=n-1
while(low<high) do
{
 m:=(low+high)/2    //mid of the array is obtained
```

```
    if(KEY=A[m]) then
    return m
  else if(KEY<A[m]) then
     high:= m-1        //search the left sublist
  else
     low:=m+1          //search the right sublist
}
return-1               //if element is not present in the list
```

Analysis

The basic operation in binary search is comparison of search key (i.e. KEY) with the array elements. To analyze efficiency of binary search we must count the number of times the search key gets compared with the array elements. The comparison is also called a three way comparison because algorithm makes the comparison to determine whether KEY is smaller, equal to or greater than A[m].

In the algorithm after one comparison the list of n elements is divided into n/2 sublists. The worst case efficiency is that the algorithm compares all the array elements for searching the desired element. In this method one comparison is made

and based on this comparison array is divided each time in n/2 sublists. Hence the worst case time complexity is given by

$$C_{worst}(n) = \underbrace{C_{worst}(n/2)}_{\text{Time required to compare left sublist or right sublist}} + \underbrace{1}_{\text{One comparison made with middle element}} \quad \text{for } n > 1$$

Also, $C_{worst}(1) = 1$

But as we consider the rounded down value when array gets divided the above equations can be written as

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1 \quad \ldots (1)$$

$$C_{worst}(1) = 1 \quad \ldots (2)$$

The above recurrence relation can be solved further. Assume $n = 2^k$ the equation (1) becomes,

$$C_{worst}(2^k) = C_{worst}(2^k/2) + 1$$

$$C_{worst}(2^k) = C_{worst}(2^{k-1}) + 1 \quad \text{... (3)}$$

Using backward substitution method, we can substitute

$$C_{worst}(2^{k-1}) = C_{worst}(2^{k-2}) + 1$$

Then equation (3) becomes

$$C_{worst}(2^k) = \left[C_{worst}(2^{k-2}) + 1\right] + 1$$

$$= C_{worst}(2^{k-2}) + 2$$

Then $C_{worst}(2^k) = \left[C_{worst}(2^{k-3}) + 1\right] + 2$

$\therefore$ $C_{worst}(2^k) = C_{worst}(2^{k-3}) + 3$

...

...

...

$$C_{worst}(2^k) = C_{worst}(2^{k-k}) + k$$

$$= C_{worst}(2^0) + k$$

$$C_{worst}(2^k) = C_{worst}(1) + k \quad \text{... (4)}$$

But as per equation 2,

$C_{worst}(1) = 1$ then we get equation (4),

$C_{worst}(2^k) = 1 + k$

$\therefore$ $C_{worst}(n) = 1 + \log_2 n$

$\therefore$ $C_{worst}(n) = \log_2 n$ for $n > 1$

As we have assumed

$n = 2^k$

Taking logarithm (base 2) on both sides

$\log_2 n = \log_2 2^k$

$\log_2 n = k \cdot \log_2 2$

$\log_2 n = k(1)$ $\because \log_2 2 = 1$

$\therefore$ $k = \log_2 n$

The worst case time complexity of binary search is $\Theta(\log_2 n)$.

Merge Sort

The merge sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out.

Merge sort on an input array with n elements consists of three steps:

Divide : partition array into two sub lists s1 and s2 with n/2 elements each.

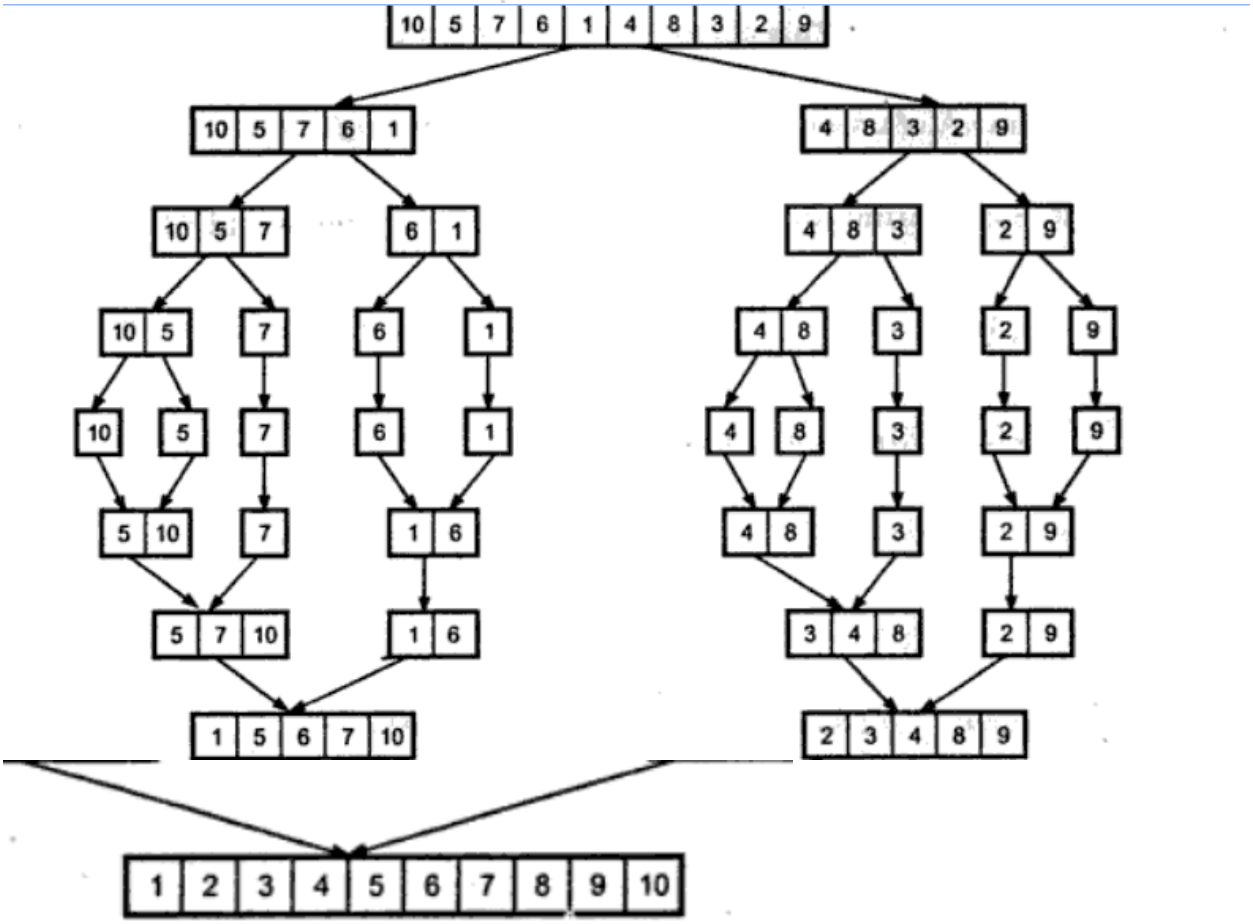
Conquer : Then sort sub list s1 and sub list s2.

Combine : merge s1 and s2 into a unique sorted group.

Example : Consider the elements as

70, 20, 30, 40, 10, 50, 60

Now we will split this list into two sublists.



```
Algorithm MergeSort(int A[0…n–1],low,high)
//Problem Description: This algorithm is for sorting the
//elements using merge sort
//Input: Array A of unsorted elements, low as beginning
//pointer of array A and high as end pointer of array A
//Output: Sorted array A[0…n – 1]
        if(low < high)then
        {
           mid ← (low+high)/2        // split the list at mid
           MergeSort(A,low,mid)      // first sublist
           MergeSort(A,mid+1,high) // second sublist
           Combine(A,low,mid,high) // merging of two sublists
        }
Algorithm Combine(A[0…n–1],low, mid, high)
{
        k ← low;    // k as index for array temp
        i ← low;    // i as index for left sublist of array A
        j ← mid+1 // j as index for right sublist of array A
        while(i <= mid and j <= high)do
        {
          if(A[i]<=A[j])then
          // if smaller element is present in left sublist
          {
                // copy that smaller element to temp array
                temp[k] ← A[i]
                i ← i+1
                k ← k+1
           }
             else  //smaller element is present in right sublist
          {
             //copy that smaller element to temp array
             temp[k] ← A[j]
```

```
            j ← j+1
            k ← k+1
        }
    }

    //copy remaining elements of left  sublist to temp
    while(i<=mid)do
    {
            temp[k] ← A[i]


            i ← i+1
            k ← k+1
        }
        //copy remaining elements of right sublist to temp
        while(j<=high)do
        {
            temp[k] ← A[j]
            j ← j+1
            k ← k+1
        }
```

Analysis

In merge sort algorithm the two recursive calls are made. Each recursive call focuses on n/2 elements of the list. After two recursive calls one call is made to combine two sublists i.e. to merge all n elements. Hence we can write recurrence relation as

$$T(n) = \underbrace{T(n/2)}_{\text{Time taken by left sublist to get sorted}} + \underbrace{T(n/2)}_{\text{Time taken by right sublist to get sorted}} + \underbrace{cn}_{\text{Time taken for combining two sublists}}$$

where $n > 1$ $T(1) = 0$

We can obtain the time complexity of Merge Sort using two methods

1. Master theorem
2. Substitution method

1. Using master theorem :

Let, the recurrence relation for merge sort is

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

i.e. $T(n) = 2T(n/2) + cn$ ... (1)

$T(1) = 0$ ... (2)

As per Master theorem

$$T(n) = \Theta(n^d \log n) \quad \text{if } a = b$$

As in equation 1,

$a = 2$, $b = 2$ and $f(n) = cn$ i.e. n^d with $d = 1$.

and $a = b^d$

i.e. $2 = 2^1$

This case gives us

$$T(n) = \Theta(n \log_2 n)$$

Hence the average and worst case time complexity of merge sort is $\Theta(n \log_2 n)$.

Quick Sort

Quick sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out. The three steps of quick sort are as follows :

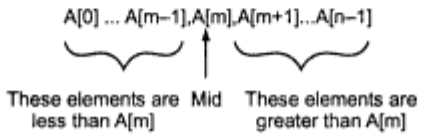
- **Divide :** Split the array into two sub arrays that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element. The splitting of the array into

two sub arrays is based on pivot element. All the elements that are less than pivot should be in left sub array and all the elements that are more than pivot should be in right sub array.

- **Conquer :** Recursively sort the two sub arrays.
- **Combine :** Combine all the sorted elements in a group to form a list of sorted elements.

In merge sort the division of array is based on the positions of array elements, but in quick sort this division is based on actual value of the element. Consider an array A[i] where i is ranging from 0 to n – 1 then we can formulize the division of array elements as

Let us understand this algorithm with the help of some example.



Algorithm

The quick sort algorithm is performed using following two important functions - *Quick* and *partition*. Let us see them-

```
Algorithm Quick(A[0...n–1], low,high)
//Problem Description : This algorithm performs sorting of
//the elements given in Array A[0...n–1]
//Input: An array A[0...n–1] in which unsorted elements are
//given. The low indicates the leftmost element in the list
//and high indicates the rightmost element in the list
//Output: Creates a sub array which is sorted in ascending
//order
if(low<high)then
//split the array into two sub arrays
m ← partition(A[low...high])// m is mid of the array
Quick(A[low...m–1])
Quick(A[mid+1...high])
```

In above algorithm call to partition algorithm is given. The *partition* performs arrangement of the elements in ascending order. The recursive *quick* routine is for dividing the list in two sub lists. The pseudo code for *Partition* is as given below -

```
Algorithm Partition (A[low...high])
//Problem Description: This algorithm partitions the
//subarray using the first element as pivot element
//Input: A subarray A with low as left most index of the
//array and high as the rightmost index of the array.
//Output: The partitioning of array A is done and pivot
//occupies its proper position. And the rightmost index of
//the list is returned
pivot ← A[low]
i ← low
j ← high+1
while(i<=j) do
{
```

```
while(A[i]<=pivot) do
  i ← i+1
while(A[j]>=pivot) do
  j ← j-1;
if(i<=j) then
    swap(A[i],A[j])//swaps A[i] and A[j]
}
swap(A[low],A[j])//when i crosses j swap A[low] and A[j]
return j//rightmost index of the list
```

The Partition function is called to arrange the elements such that all the elements that are less than pivot are at the left side of pivot and all the elements that are greater than pivot are all at the right of pivot. In other words pivot is occupying its proper position and the partitioned list is obtained in an ordered manner.

Analysis

Best case (split in the middle)

If the array is always partitioned at the mid, then it brings the best case efficiency of an algorithm.

The recurrence relation for quick sort for obtaining best case time complexity is,

$$C(n) = C(n/2) + C(n/2) + n \quad \text{... equation (1)}$$

$$C(n) = \underbrace{C(n/2)}_{\text{Time required to sort left sub array}} + \underbrace{C(n/2)}_{\text{Time required to sort right sub array}} + \underbrace{n}_{\text{Time required for partitioning the sub array}} \quad \text{... equation (1)}$$

and $C(1) = 0$

Method 1 : Using Master theorem

We will solve equation (1) using master theorem.

The master theorem is

If $f(n) \in \Theta(n^d)$ then

1. $T(n) = \Theta(n^d)$ if $a < b^d$
2. $T(n) = \Theta(n^d \log n)$ if $a = b^d$
3. $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$

We get,

$$C(n) = 2\,C(n/2) + n$$

Here $f(n) \in n^1$ $\therefore d = 1$

Now, a = 2 and b = 2.

As from case 2 we get $a = b^d$ i.e. $2 = 2^1$, we get

$$T(n) \text{ i.e. } C(n) = \Theta(n^d \log n)$$

$$\therefore \quad C(n) = \Theta(n \log n)$$

Thus,

Best case time complexity of quick sort is $\Theta(n \log_2 n)$.

Worst case

We can write it as

$$C(n) = C(n-1) + n$$

or

$$C(n) = n + (n-1) + (n-2) + \ldots + 2 + 1$$

But as we know

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

$$\therefore \quad C(n) = \Theta(n^2)$$

The time complexity of worst case of quick sort is $\Theta(n^2)$.

Strassen matrix multiplication

Strassen showed that 2 × 2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions.

The divide and conquer approach can be used for implementing Strassen's matrix multiplication.

- **Divide** : Divide matrices into sub-matrices : A0 , A1, A2 etc.
- **Conquer** : Use a group of matrix multiply equations.
- **Combine** : Recursively multiply sub-matrices and get the final result of multiplication after performing required additions or subtractions.

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

$$S1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$S2 = (A_{21} + A_{22}) \times B_{11}$$
$$S3 = A_{11} \times (B_{12} - B_{22})$$
$$S4 = A_{22} \times (B_{21} - B_{11})$$
$$S5 = (A_{11} + A_{12}) \times B_{22}$$

$$S6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$
$$S7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

$$C_{11} = S1 + S4 - S5 + S7$$
$$C_{12} = S3 + S5$$
$$C_{21} = S2 + S4$$
$$C_{22} = S1 + S3 - S2 + S6$$

Algorithm

Here we are dividing matrices in sub-matrices and recursively multiplying sub-matrices

```
Algorithm St_Mul(int *A, int *B, int *C, int n)
{
 if (n == 1)then
 {
   (*C) =(*C)+ (*A) * (*B);
 }
 else
 {
   St_Mul(A,B,C,n/4);
   St_Mul(A, B+(n/4), C+(n/4), n/4);
   St_Mul(A+2 *(n/4), B, C+2 *(n/4), n/4);
   St_Mul(A+2 *(n/4), B+(n/4), C+3 *(n/4), n/4);
   St_Mul(A+(n/4), B+2*(n/4), C, n/4);
```

```
    St_Mul(A+(n/4), B+3*(n/4), C+(n/4), n/4);
    St_Mul(A+3*(n/4), B+2*(n/4), C+2*(n/4), n/4);
    St_Mul(A+3*(n/4), B+3*(n/4), C+3*(n/4), n/4);
  }
}
```

Analysis of Algorithm

$T(1) = 1$ assume $n = 2^k$

$T(n) = 7\ T(n/2)$

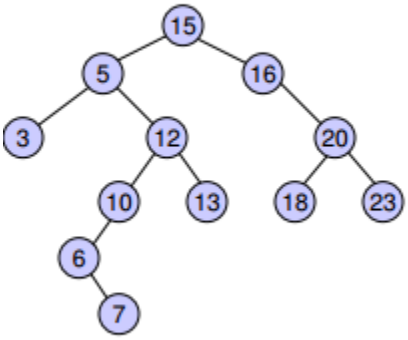
$T(n) = 7^k\ T(n/2^k)$

$T(n) = 7^{\log n}$

$T(n) = n^{\log 7} = n^{2.81}$

UNIT-2

Efficient non recursive tree traversal algorithms



- in-order: (left, root, right)
  3, 5, 6, 7, 10, 12, 13,
  15, 16, 18, 20, 23
- pre-order: (root, left, right)
  15, 5, 3, 12, 10, 6, 7,
  13, 16, 20, 18, 23
- post-order: (left, right, root)
  3, 7, 6, 10, 13, 12, 5,
  18, 23, 20, 16, 15

Non recursive Inorder traversal algorithm

1. Start from the root, let's it is current.
2. If current is not NULL, push the node on to stack.
3. Move to left child of current and go to step 2.
4. If current is NULL, and stack is not empty, pop node from the stack.
5. Print the node value and change current to right child of current.
6. Go to step 2.

So we go on traversing all left node, as we visit the node, we will put that node into stack. (remember we need to visit parent after the child and as we will encounter parent first when start from root, it's case for LIFO :) and hence the stack). Once we reach NULL node, we will take the node at the top of the stack, last node which we visited. Print it.

Check if there is right child to that node. If yes, move right child to stack and again start traversing left child node and put them on to stack. Once we have traversed all node, our stack will be empty.

Non recursive preorder traversal algorithm

Idea is to print parent node first, then left node and then last visit right node. We get the idea that once we visit the node, next node to be taken is left, and then right, when we push them onto stack, we need to put right first and then left, so when we pop node out of stack, we get left.

1. Start from root and push on to stack
2. Pop from stack and print the node.
3. Push right child onto to stack.
4. Push left child onto to stack.
5. Repeat Step 2 to 5 till stack is not empty.

Non recursive postorder traversal algorithm

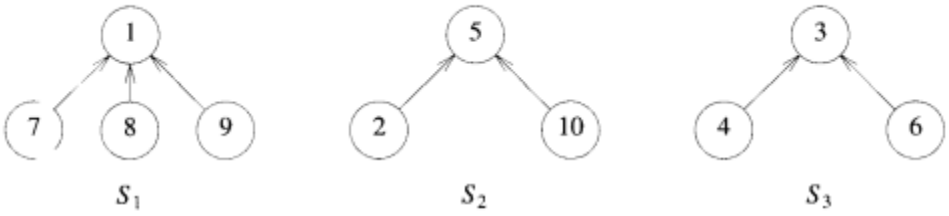
Left node, right node and last parent node.

```
1.1 Create an empty stack
2.1 Do following while root is not NULL
    a) Push root's right child and then root to stack.
    b) Set root as root's left child.
2.2 Pop an item from stack and set it as root.
    a) If the popped item has a right child and the right child
       is at top of stack, then remove the right child from stack,
       push the root back and set root as root's right child.
    b) Else print root's data and set root as NULL.
2.3 Repeat steps 2.1 and 2.2 while stack is not empty.
```

SETS AND DISJOINT SET UNION

Introduction

In this section we study the use of forests in the representation of sets. We shall assume that the elements of the sets are the numbers $1, 2, 3, \ldots, n$. These numbers might, in practice, be indices into a symbol table in which the names of the elements are stored. We assume that the sets being represented are pairwise disjoint (that is, if S_i and S_j, $i \neq j$, are two sets, then there is no element that is in both S_i and S_j). For example, when $n = 10$, the elements can be partitioned into three disjoint sets, $S_1 = \{1, 7, 8, 9\}$, $S_2 = \{2, 5, 10\}$, and $S_3 = \{3, 4, 6\}$. Figure shows one possible representation for these sets. In this representation, each set is represented as a tree. Notice that for each set we have linked the nodes from the children to the parent, rather than our usual method of linking from the parent to the children. The reason for this change in linkage becomes apparent when we discuss the implementation of set operations.



Possible tree representation of sets

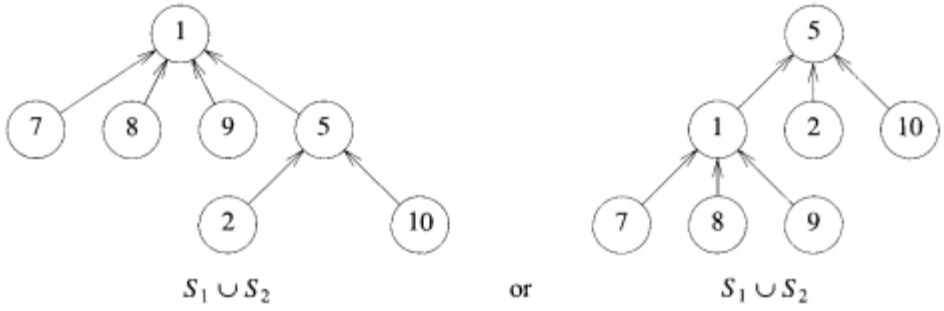
The operations we wish to perform on these sets are:

1. **Disjoint set union.** If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = all elements x such that x is in S_i or S_j. Thus, $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$. Since we have assumed that all sets are disjoint, we can assume that following the union of S_i and S_j, the sets S_i and S_j do not exist independently; that is, they are replaced by $S_i \cup S_j$ in the collection of sets.

2. **Find**(i). Given the element i, find the set containing i. Thus, 4 is in set S_3, and 9 is in set S_1.

Union and Find Operations

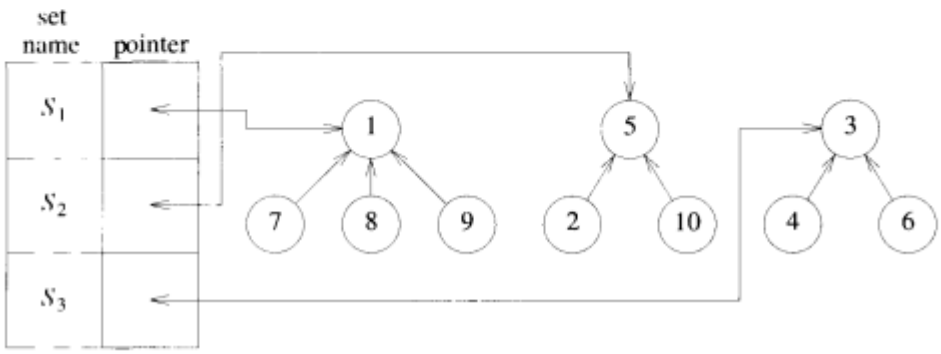
Let us consider the union operation first. Suppose that we wish to obtain the union of S_1 and S_2 . Since we have linked the nodes from children to parent, we simply make one of the trees a subtree of the other. $S_1 \cup S_2$ could then have one of the representations of Figure 2.18.



Possible representations of $S_1 \cup S_2$

To obtain the union of two sets, all that has to be done is to set the parent field of one of the roots to the other root. This can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set. If, in addition, each root has a pointer to the set name, then to determine which set an element is currently in, we follow parent links to the root of its tree and use the pointer to the set name. The data representation for S_1, S_2, and S_3 may then take the form shown in Figure .

In presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them. This simplifies



the discussion. The transition to set names is easy. If we determine that element i is in a tree with root j, and j has a pointer to entry k in the set name table, then the set name is just $name[k]$. If we wish to unite sets S_i and S_j, then we wish to unite the trees with roots FindPointer(S_i) and FindPointer(S_j). Here FindPointer is a function that takes a set name and determines the root of the tree that represents it. This is done by an examination of the [set name, pointer] table. In many applications the set name is just the element at the root. The operation of $Find(i)$ now becomes: Determine the root of the tree containing element i. The function $Union(i, j)$ requires two trees with roots i and j be joined. Also to simplify, assume that the set elements are the numbers 1 through n.

Since the set elements are numbered 1 through n, we represent the tree nodes using an array $p[1 : n]$, where n is the maximum number of elements. The ith element of this array represents the tree node that contains element i. This array element gives the parent pointer of the corresponding tree node. Figure shows this representation of the sets S_1, S_2, and S_3 of Figure Notice that root nodes have a parent of -1.

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

Array representation of S_1, S_2, and S_3

We can now implement $Find(i)$ by following the indices, starting at i until we reach a node with parent value -1. For example, $Find(6)$ starts at 6 and then moves to 6's parent, 3. Since $p[3]$ is negative, we have reached the root. The operation $Union(i, j)$ is equally simple. We pass in two trees with roots i and j. Adopting the convention that the first tree becomes a subtree of the second, the statement $p[i] := j$; accomplishes the union.

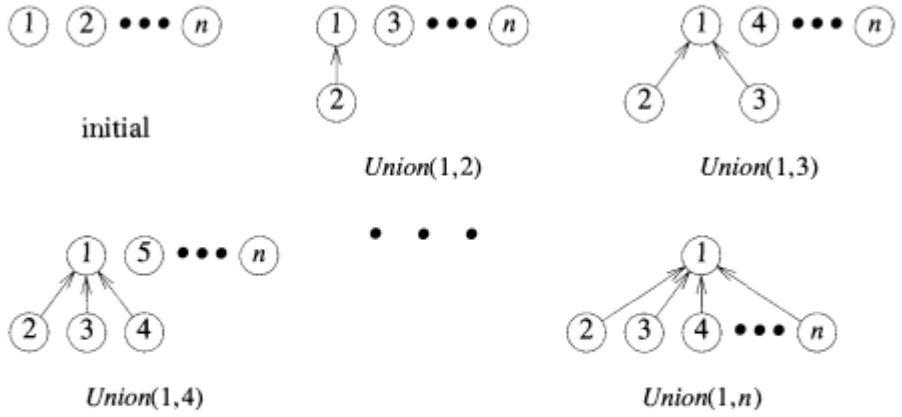
```
1   Algorithm SimpleUnion(i, j)
2   {
3       p[i] := j;
4   }

1   Algorithm SimpleFind(i)
2   {
3       while (p[i] ≥ 0) do i := p[i];
4       return i;
5   }
```

$Union(1,2),\ Union(2,3),\ Union(3,4),\ Union(4,5), \ldots,\ Union(n-1,n)$

$Find(1),\ Find(2), \ldots,\ Find(n)$

Degenerate tree

Definition [Weighting rule for *Union(i,j)*] If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i; otherwise make i the parent of j. □



Trees obtained using the weighting rule

```
1   Algorithm WeightedUnion(i, j)
2   // Union sets with roots i and j, i ≠ j, using the
3   // weighting rule. p[i] = −count[i] and p[j] = −count[j].
4   {
5       temp := p[i] + p[j];
6       if (p[i] > p[j]) then
7       { // i has fewer nodes.
8           p[i] := j; p[j] := temp;
9       }
10      else
11      { // j has fewer or equal nodes.
12          p[j] := i; p[i] := temp;
13      }
14  }
```

Definition : [Collapsing rule]: If j is a node on the path from i to its root and $p[i] \neq root[i]$, then set $p[j]$ to $root[i]$. □

$$Find(8),\ Find(8), \ldots,\ Find(8)$$

If SimpleFind is used, each $Find(8)$ requires going up three parent link fields for a total of 24 moves to process all eight finds. When CollapsingFind is used, the first $Find(8)$ requires going up three links and then resetting two links. Note that even though only two parent links need to be reset, CollapsingFind will reset three (the parent of 5 is reset to 1). Each of the remaining seven finds requires going up only one link field. The total cost is now only 13 moves. □

```
1   Algorithm CollapsingFind(i)
2   // Find the root of the tree containing element i. Use the
3   // collapsing rule to collapse all nodes from i to the root.
4   {
5       r := i;
6       while (p[r] > 0) do r := p[r]; // Find the root.
7       while (i ≠ r) do  // Collapse nodes from i to root r.
8       {
9           s := p[i]; p[i] := r; i := s;
10      }
11      return r;
12  }
```

Graph Traversals

In breadth first search we start at a vertex v and mark it as having been reached (visited). The vertex v is at this time said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from v are visited next. These are new unexplored vertices. Vertex v has now been explored. The newly visited vertices haven't been explored and are put onto the end of a list of unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left.

```
1   Algorithm BFS(v)
2   // A breadth first search of G is carried out beginning
3   // at vertex v. For any node i, visited[i] = 1 if i has
4   // already been visited. The graph G and array visited[ ]
5   // are global; visited[ ] is initialized to zero.
6   {
7       u := v; // q is a queue of unexplored vertices.
8       visited[v] := 1;
9       repeat
10      {
11          for all vertices w adjacent from u do
12          {
13              if (visited[w] = 0) then
14              {
15                  Add w to q; // w is unexplored.
16                  visited[w] := 1;
17              }
18          }
19          if q is empty then return; // No unexplored vertex.
20          Delete u from q; // Get first unexplored vertex.
21      } until(false);
22  }
```

A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At this time the exploration of the new vertex u begins. When this new vertex has been explored, the exploration of v continues. The search terminates when all reached vertices have been fully explored.

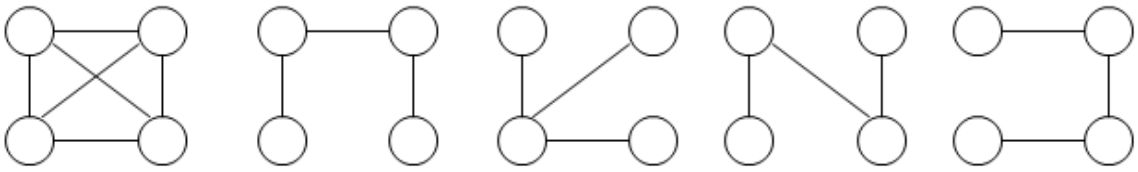
```
1   Algorithm DFS(v)
2   // Given an undirected (directed) graph G = (V, E) with
3   // n vertices and an array visited[ ] initially set
4   // to zero, this algorithm visits all vertices
5   // reachable from v. G and visited[ ] are global.
6   {
7       visited[v] := 1;
8       for each vertex w adjacent from v do
9       {
10          if (visited[w] = 0) then DFS(w);
11      }
12  }
```

A *spanning tree* for a graph G is a sub-graph of G which is a tree that includes every vertex of G.

Notes.

* A spanning tree of a graph G is a "maximal" tree contained in the graph G.

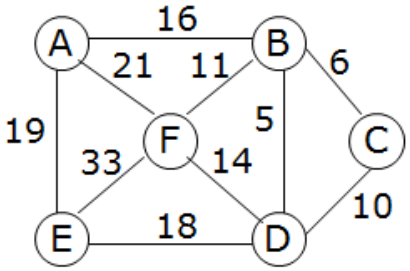
* When you have a spanning tree T for a graph G, you cannot add another edge of G to T without producing a circuit.



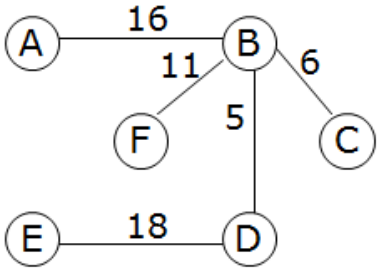
A connected, undirected graph

Four of the spanning trees of the graph

- Suppose you have a connected undirected graph with a weight (or cost) associated with each edge
- The cost of a spanning tree would be the sum of the costs of its edges
- A minimum-cost spanning tree is a spanning tree that has the lowest cost



A connected, undirected graph



A minimum-cost spanning tree

- There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms

- **Kruskal's algorithm:** Start with *no* nodes or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle
  - Here, we consider the spanning tree to consist of edges only

- **Prim's algorithm:** Start with any *one node* in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.
  - Here, we consider the spanning tree to consist of both nodes and edges

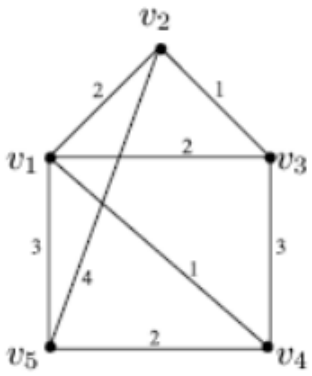
Kruskal's Algorithm

Example.

Find the minimal spanning tree for the following connected weighted graph *G*.

The starting point of Kruskal's Algorithm is to make an "edge" list, in which the edges are listed in order of increasing weights.

G:

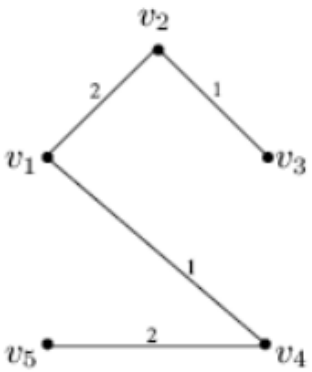


Edge Table

Edge	Weight
$e_1 = (v_2, v_3)$	1
$e_2 = (v_1, v_4)$	1
$e_3 = (v_2, v_1)$	2
$e_4 = (v_1, v_3)$	2
$e_5 = (v_5, v_4)$	2
$e_6 = (v_1, v_5)$	3
$e_7 = (v_3, v_4)$	3
$e_8 = (v_2, v_5)$	4

Edge	Weight	Will adding edge make circuit?	Action Taken	Cumulative Weight of Subgraph
$e_1 = (v_2, v_3)$	1	No	Added	1
$e_2 = (v_1, v_4)$	1	No	Added	2
$e_3 = (v_2, v_1)$	2	No	Added	4
$e_4 = (v_1, v_3)$	2	Yes	Not Added	4
$e_5 = (v_5, v_4)$	2	No	Added	6
$e_6 = (v_1, v_5)$	3	Yes	Not Added	6
$e_7 = (v_3, v_4)$	3	Yes	Not Added	6
$e_8 = (v_2, v_5)$	4	Yes	Not Added	6

The minimum spanning tree is drawn below.



Kruskal's Algorithm for finding minimum spanning trees for weighted graphs (Epp's version) is then:

Input: G a connected weighted graph with n vertices.

Algorithm Body:

(Build a sub-graph T of G to consist of all of the vertices of G with edges added in order of increasing weight. At each stage, let m be the number of edges of T.)

1. Initialise T to have all of the vertices of G and no edges.
2. Let E be the set of all edges of G and let $m = 0$. (pre-condition: G is connected.)
3. While $(m < n - 1)$

a. Find an edge e in E of least weight.

b. Delete e from E.

c. If addition of e to the edge set of T does not produce a circuit then add e to the edge set of T and set $m = m + 1$

End While (post-condition: T is a minimum spanning tree for G.)

Output: T (a graph)

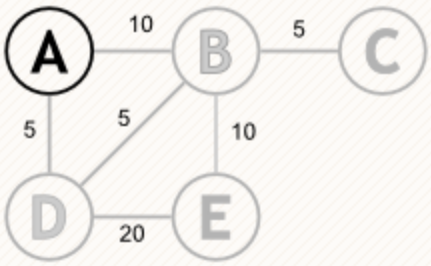
End Algorithm

Prims algorithm

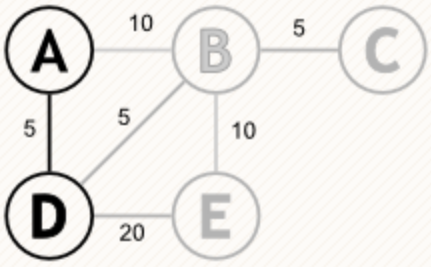
1. Start with a tree which contains only one node.
2. Identify a node (outside the tree) which is closest to the tree and add the minimum weight edge from that node to some node in the tree and incorporate the additional node as a part of the tree.
3. If there are less then $n - 1$ edges in the tree, go to **2**

```
T = a spanning tree containing a single node s;
E = set of edges adjacent to s;
while T does not contain all the nodes {
        remove an edge (v, w) of lowest cost from E
        if w is already in T then discard edge (v, w)
        else {
                add edge (v, w) and node w to T
                add to E the edges adjacent to w
        }
        }
```

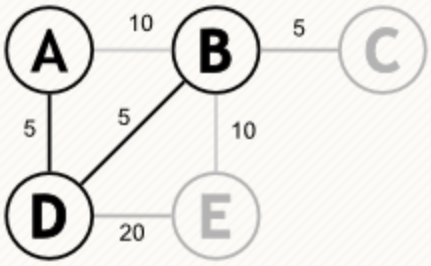
Start with only node A in the tree.

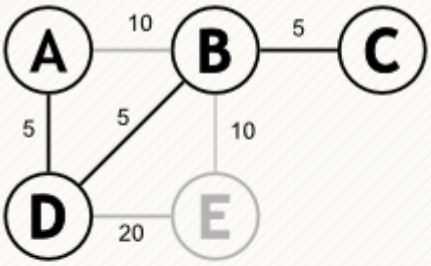


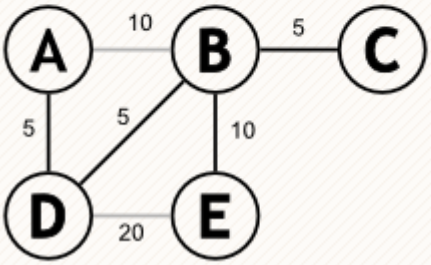
Find the closest node to the tree, and add it.



Repeat until there are $n - 1$ edges in the tree.



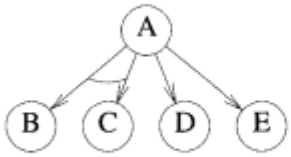




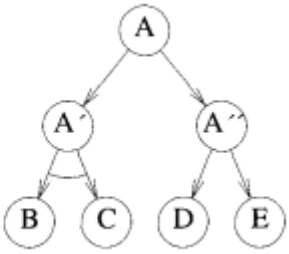
Time complexity is **O(n²)** time.

AND/OR Graph

Many complex problems can be broken down into a series of subproblems such that the solution of all or some of these results in the solution of the original problem. These subproblems can be broken down further into sub-subproblems, and so on, until the only problems remaining are sufficiently primitive as to be trivially solvable. This breaking down of a complex problem into several subproblems can be represented by a directed graphlike structure in which nodes represent problems and descendents of nodes represent the subproblems associated with them.

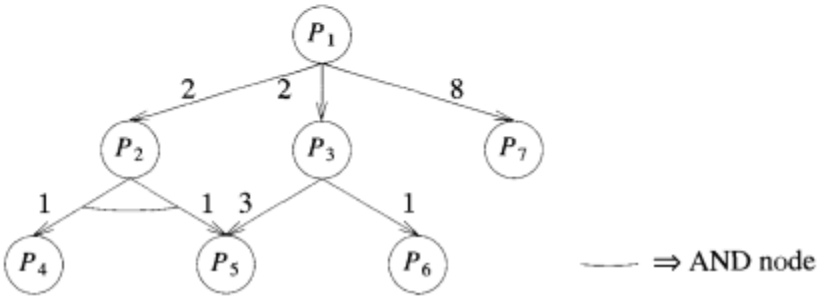


(a)



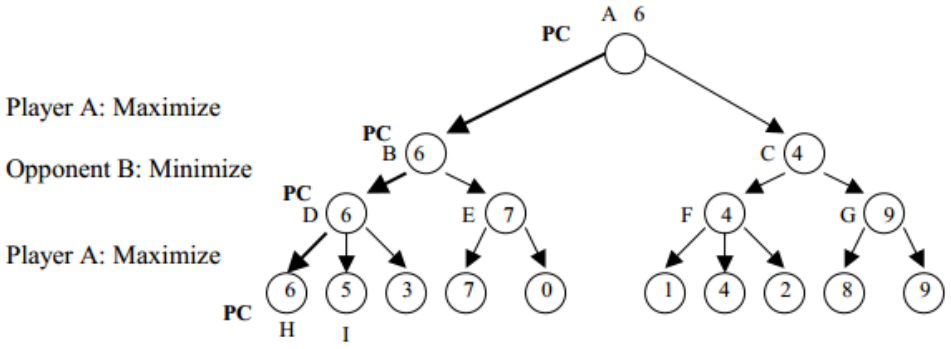
Nodes of the first type are called AND nodes and those of the latter type OR nodes. Nodes A and A'' of Figure are OR nodes whereas node A' is an AND node. The AND nodes are drawn with an arc across all edges leaving the node. Nodes with no descendents are called *terminal*. Terminal nodes represent primitive problems and are marked either solvable or not solvable. Solvable terminal nodes are represented by rectangles. An AND/OR graph need not always be a tree.

Consider the directed graph of Figure . The problem to be solved is P_1. To do this, one can solve node P_2, P_3, or P_7, as P_1 is an OR node. The cost incurred is then either 2, 2, or 8 (i.e., cost in addition to that of solving one of P_2, P_3, or P_7). To solve P_2, both P_4 and P_5 have to be solved, as P_2 is an AND node. The total cost to do this is 2. To solve P_3, we can solve either P_5 or P_6. The minimum cost to do this is 1. Node P_7 is free. In this example, then, the optimal way to solve P_1 is to solve P_6 first, then P_3, and finally P_1. The total cost for this solution is 3. □



Game Trees: All games which require only mental effort would always have number of possible options at any position of the game. For each position, there would be number of counter moves. The repetitive pattern results in what is known a game tree.

Game playing programs depend on game tree search to find the best move for the current position, assuming the best play of the opponent. In a two-person game, two players choose a legal move alternately, and both of them intuitively try to maximize their advantage. Because of this reason, finding the best move for a player must assume the opponent also plays his/her best moves. In other words, if the leaves are evaluated on the viewpoint of player A, player A will always play moves that maximize the value of the resulting position, while the opponent B plays moves that minimize the value of the resulting position. This gives us the MiniMax algorithm.



MiniMax Search Tree

Figure 2.1 simulates a MiniMax search in a game tree. Every leaf has a corresponding value, which is approximated from player A's viewpoint. When a path is chosen, the value of the child will be passed back to the parent. For example, the value for D is 6, which is the maximum value of its children, while the value for C is 4, which is the minimum value of F and G. In this example, the best sequence of moves found by the maximizing / minimizing procedure is the path through nodes A, B, D and H, which is called the principal continuation **[7]**. The nodes on the path are denoted as PC (principal continuation) nodes.

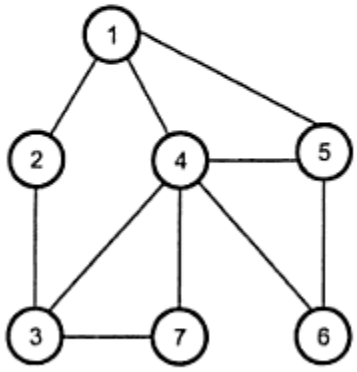
```
int findMax (struct tree_node *p)
{
    int node_data, leftmax, rightmax, max;

    max = -1
//assume all values in the tree are positive
integers

    if (p !=   NULL)
    {  node_data = p -> data;
       leftmax = findMax(p -> left_child);
      rightmax = findMax(p->right_child);

       //find the largest of the tree values.

      if (leftmax > rightmax)
        max = leftmax;
          else
        max = rightmax;
          if (node_data > max)
        max = node_data;
    }
```

return max;

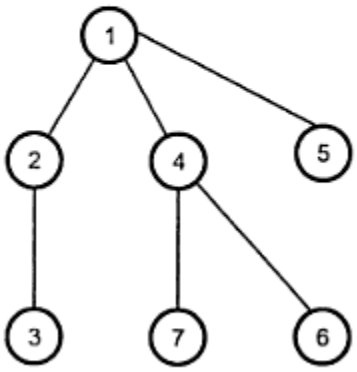
Connected Components

If G is a connected undirected graph, then we can visit all the vertices of the graph in the first call to BFS. The subgraph which we obtain after traversing the graph using BFS represents the connected component of the graph.

For example : Consider the graph G whose connected component can be given as below -



Graph G



Connected component of G obtained by breadth first search

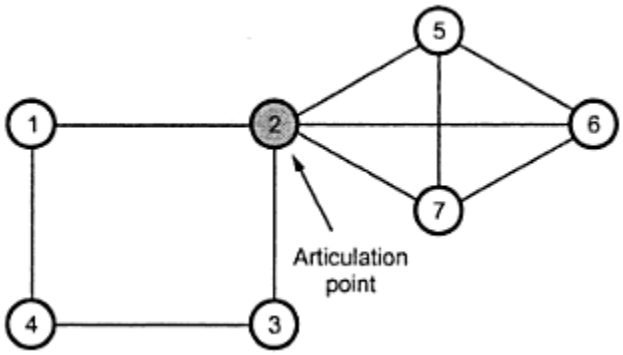
Thus **BFS** can be used to determine whether G is connected. All the newly visited vertices on call to BFS, represent the vertices in connected component of graph G. The subgraph formed by these vertices make the **connected component.**

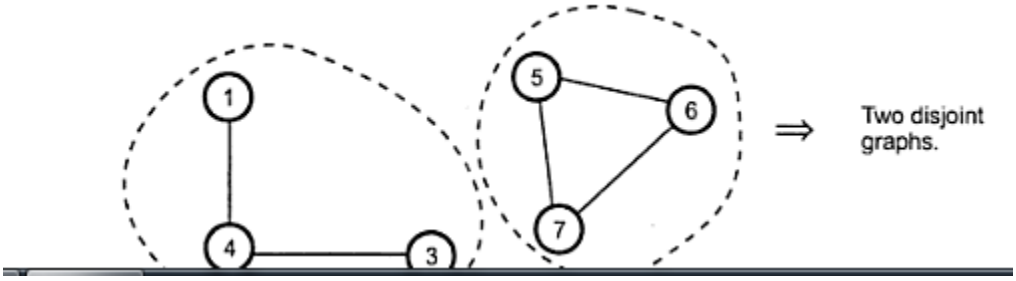
If adjacency list graph is used then the BFS that will obtain the connected component will require $\theta(n+e)$ time where **n** is total number of vertices and **e** represent total number of edges, in the graph.

Bi-connected Components

In this section we will understand two important concepts those are articulation point and bi-connected components. We will also learn how depth first search helps in finding an articulation point and bi-connected components.

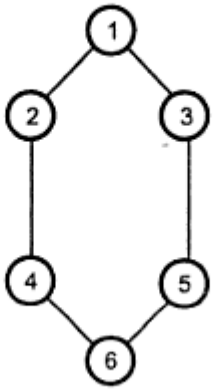
Definition of Articulation Point : Let $G = (V, E_0)$ be a connected undirected graph, then an articulation point of graph G is a vertex whose removal disconnects graph G. This articulation point is a kind of cut-vertex.





- A graph G is said to be bi-connected if it contains no articulation points.

For example



Even though we remove any single vertex we do not get disjoint graphs.

- To identify articulation points following observations can be made.

i) The root of the DFS tree is an articulation if it has two or more children.

ii) A leaf node of DFS tree is not an articulation point.

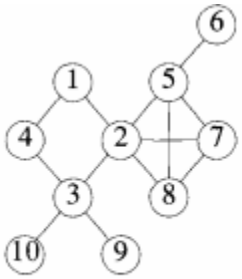
iii) If u is any internal node then it is not an articulation point if and only if from every child w of u it is possible to reach an ancestor of u using only a path made up of descendents of w and back edge.

This observation leads to a simple rule as,

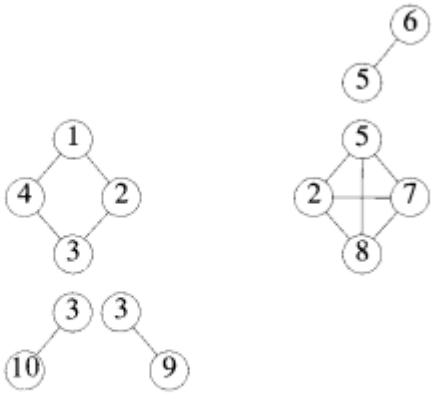
Low [u] = min { dfn [u], min {Low [w]/w is a child of u}, min {dfn [w]/ (u,w) is a back edge}}

Where Low [u] is the lowest depth first number that can be reached from u using a path of descendents followed by at most one back edge. The vertex u is an articulation point if u is child of w such that

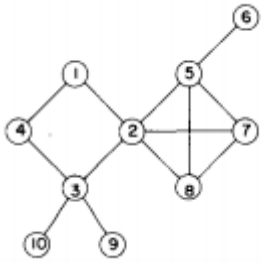
L[w] ≥ dfn[u].



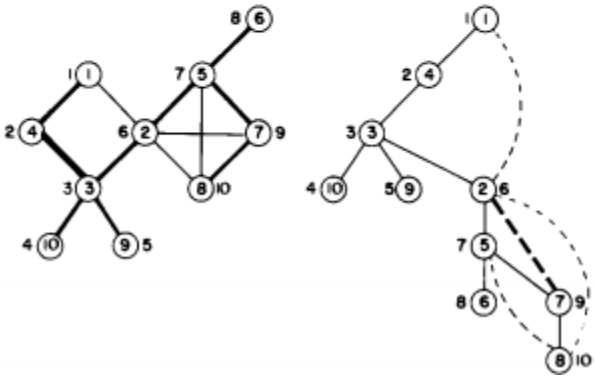
(a) Graph *G*



Biconnected components of graph of



Figures (a) and (b) show a depth first spanning tree of the graph of In each figure there is a number outside each vertex. These numbers correspond to the order in which a depth first search visits these vertices. This number will be referred to as the *depth first number* (DFN) of the vertex. Thus, DFN(1) = 1, DFN(4) = 2 and DFN(6) = 8. In Figure 6.29(b) solid edges form the depth first spanning tree. These edges are called *tree edges*. Broken edges (i.e. all remaining edges) are called *back edges*.



Depth first spanning trees have a property that is very useful in identifying articulation points and biconnected components. This property is that if (u, v) is any edge in G then relative to the depth first spanning tree T either u is an ancestor of v or v is an ancestor of u. So, there are no cross edges relative to a depth first spanning tree ((u, v) is a *cross* edge relative to T iff neither u is an ancestor of v nor v an ancestor of u). To see this, assume that $(u, v) \in E(G)$ and (u, v) is a cross edge. (u, v) cannot be a tree edge

as otherwise u is the parent of v or vice versa. So, (u, v) must be a back edge. Without loss of generality, we may assume DFN(u) < DFN(v). Since vertex u is visited first, its exploration cannot be complete until vertex v is visited. From the definition of depth first search, it follows that u is an ancestor of all vertices visited until u is completely explored. Hence u is an ancestor of v in T and (u, v) cannot be a cross edge.

We next observe that the root node of a depth first spanning tree is an articulation point iff it has at least two children. Furthermore, if u is any other vertex then it is not an articulation point iff from every child w of u it is possible to reach an ancestor of u using only a path made up of descendents of w and a back edge. Note that if this cannot be done for some child w of u then the deletion of vertex u will leave behind at least two nonempty components (one containing the root and the other containing vertex w). This observation leads to a simple rule to identify articulation points. For each vertex u define $L(u)$ as follows:

$$L(u) = \min\{\text{DFN}(u), \min\{L(w) \mid w \text{ is a child of } u\}, \min\{\text{DFN}(w) \mid (u, w) \text{ is a back edge}\}\}$$

It should be clear that $L(u)$ is the lowest depth first number that can be reached from u using a path of descendents followed by at most one back edge. From the preceding discussion it follows that if u is not the root then u is an articulation point iff u has a child w such that $L(w) \geq \text{DFN}(u)$. For the spanning tree of Figure the L values are $L(1{:}10) = (1, 1, 1, 1, 6, 8, 6, 6, 5, 4)$. Vertex 3 is an articulation point as child 10 has $L(10) = 4$ while $\text{DFN}(3) = 3$. Vertex 2 is an articulation point as child 5 has $L(5) = 6$ and $\text{DFN}(2) = 6$. The only other articulation point is vertex 5; child 6 has $L(6) = 8$ while $\text{DFN}(5) = 7$.

```
1   Algorithm Art(u, v)
2   // u is a start vertex for depth first search. v is its parent if any
3   // in the depth first spanning tree. It is assumed that the global
4   // array dfn is initialized to zero and that the global variable
5   // num is initialized to 1. n is the number of vertices in G.
6   {
7       dfn[u] := num; L[u] := num; num := num + 1;
8       for each vertex w adjacent from u do
9       {
10          if (dfn[w] = 0) then
11          {
12              Art(w, u); // w is unvisited.
13              L[u] := min(L[u], L[w]);
14          }
15          else if (w ≠ v) then L[u] := min(L[u], dfn[w]);
16      }
17  }
```

Algorithm Pseudocode to compute dfn and L

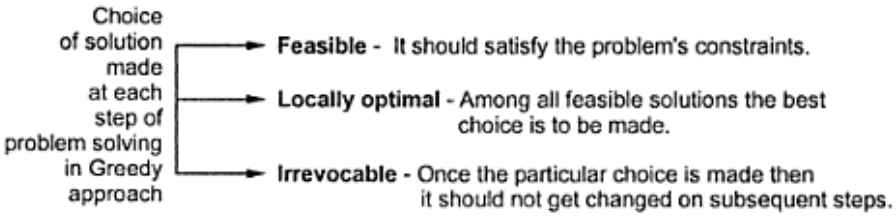
```
1       Algorithm BiComp(u, v)
2       // u is a start vertex for depth first search. v is its parent if
3       // any in the depth first spanning tree. It is assumed that the
4       // global array dfn is initially zero and that the global variable
5       // num is initialized to 1. n is the number of vertices in G.
6       {
7           dfn[u] := num; L[u] := num; num := num + 1;
8           for each vertex w adjacent from u do
9           {
9.1             if ((v ≠ w) and (dfn[w] < dfn[u]))then
9.2                 add (u, w) to the top of a stack s;
10              if (dfn[w] = 0) then
11              {
11.1                if (L[w] ≥ dfn[u]) then
11.2                {
11.3                    write ("New bicomponent");
11.4                    repeat
11.5                    {
11.6                        Delete an edge from the top of stack s;
11.7                        Let this edge be (x, y);
11.8                        write (x, y);
11.9                    } until (((x, y) = (u, w)) or ((x, y) = (w, u)));
11.10               }
12                  BiComp(w, u); // w is unvisited.
13                  L[u] := min(L[u], L[w]);
14              }
15              else if (w ≠ v) then L[u] := min(L[u], dfn[w]);
16          }
17      }
```

Algorithm Pseudocode to determine bicomponents

UNIT-3 NOTES

Greedy method

In an algorithmic strategy like Greedy, the decision of solution is taken based on the information available. The Greedy method is a straightforward method. This method is popular for obtaining the optimized solutions. In Greedy technique, the solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. At each step the choice made should be,



In Greedy method following activities are performed.

1. First we select some solution from input domain.
2. Then we check whether the solution is feasible or not.
3. From the set of feasible solutions, particular solution that satisfies or nearly satisfies the objective of the function. Such a solution is called optimal solution.
4. As Greedy method works in stages. At each stage only one input is considered at each time. Based on this input it is decided whether particular input gives the optimal solution or not.

Job Sequencing with Deadlines

Consider that there are n jobs that are to be executed. At any time t=1,2,3,.. only exactly one job is to be executed.The profits pi are given.These profits are gained by corresponding jobs.For obtaining feasible solution we should take care that the jobs get completed within their given deadlines.

Let n = 4

n	pi	di
1	70	2
2	12	1
3	18	2
4	35	1

We will follow following rules to obtain the feasible solution

- Each job takes one unit of time.
- If job starts before or at its deadline, profit is obtained, otherwise no profit.
- Goal is to schedule jobs to maximize the total profit.
- Consider all possible schedules and compute the minimum total time in the system.

The feasible solutions are obtained by various permutations and combinations of jobs.

n	pi
1	70
2	12
3	18
4	35
1,3	88
2,1	82
2,3	30
3,1	88
4,1	105
4,3	53

Each job takes only one unit of time. Deadline of job means a time on which or before which the job has to be executed. The sequence {2,4} is not allowed because both have deadline 1. If job 2 is started at 0 it will be completed on 1 but we can not start job 4 on 1 since deadline of job 4 is 1. The feasible sequence is a sequence that allows all jobs in a sequence to be executed within their deadlines and highest profit can be gained.

- The optimal solution is a feasible solution with maximum profit.
- In above example sequence 3,2 is not considered as $d_3 > d_2$ but we have considered the sequence 2,3 as feasible solution because $d_2 < d_3$.
- We have chosen job 1 first then we have chosen job 4.The solution 4,1 is feasible as the order of execution is 4 then 1.Also $d_4 < d_1$. If we try {1,3,4} then it is not a feasible solution, hence reject 3 from the set. Similarly if we add job 2 in the sequence then the sequence becomes {1,2,4}.This is also not a feasible solution hence reject it. Finally the feasible sequence is 4,1. This sequence is optimum solution as well.

```
1   Algorithm GreedyJob(d, J, n)
2   // J is a set of jobs that can be completed by their deadlines.
3   {
4       J := {1};
5       for i := 2 to n do
6       {
7           if (all jobs in J ∪ {i} can be completed
8               by their deadlines) then J := J ∪ {i};
9       }
10  }
```

```
line  procedure JS(D, J, n, k)
        //D(i) ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs are//
        //ordered such that p1 ≥ p2 ≥ ... ≥ pn. J(i) is the ith job in//
        //the optimal solution, 1 ≤ i ≤ k. Also, at termination D(J(i))//
        //≤ D(J(i + 1)), 1 ≤ i < k.//
        integer D(0:n), J(0:n), i, k, n, r
        D(0) ← J(0) ← 0   //initialize//
        k ← 1; J(1) ← 1 //include job 1//
        for i ← 2 to n do //consider jobs in nonincreasing order of pi//
          //Find position for i and check feasibility of insertion//
          r ← k
          while D(J(r)) > D(i) and D(J(r)) ≠ r do
            r ← r - 1
          repeat
          if D(J(r)) ≤ D(i) and D(i) > r then
            //insert i into J//
            for l ← k to r + 1 by - 1 do
              J(l + 1) ← J(l)
            repeat
            J(r + 1) ← i; k ← k + 1
          endif
        repeat
      end JS
```

Time complexity is $O(n^2)$

KNAPSACK PROBLEM

The 0-1 Knapsack problem is defined as follows: Given a knapsack of capacity c and n items of weights $\{ w_1, w_2 \ldots w_n\}$ and profits $\{ p_1, p_2 \ldots p_n\}$, the objective is to choose a subset of n objects that fits into the knapsack and that maximizes the total profit.

Consider a knapsack (bag) with a capacity of c. We select items from a list of n items. Each item has both a weight of w_i and a profit of p_i. In a feasible solution, the sum of the weights must not exceed knapsack capacity (c), and an optimal solution is both feasible and reaches the maximum profit.

An optimal packing is a feasible solution one with a maximum profit:

$$p_1x_1 + p_2x_2 + p_3x_3 + \ldots + p_nx_n = \sum_{i=1}^{n} p_ix_i$$

subject to constraints: $w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_nx_n = \sum_{i=1}^{n} w_ix_i \le c$

and $x_i = 1$ or 0, $1 \quad i \quad n$

We have to find the values of x_i where $x_i = 1$ if ith item is packed into the knapsack and $x_i = 0$ if ith item is not packed.

Greedy strategies for the knapsack problem are:

(1) From the remaining items, select the item with maximum profit that fits into the knapsack.

(2) From the remaining items, select the item that has minimum weight and also fits into the knapsack.

(3) From the remaining items, select the one with maximum p_i/w_i that fits into the knapsack.

There are four items that have a profit and weight list below. The knapsack capacity is 4 kgs. We apply the above three greedy strategies.

item (i)	p_i	w_i
1	5	1
2	9	3
3	4	2
4	8	2

Figure A Knapsack problem

Method 1 chooses the item with maximum profit. First choose the item 2 which has a maximum profit of Rs 9 with weight 3 kgs. If we next select item 4, then total profit will be Rs 17. But, it violates the capacity constraint of 4 kgs.

Method 2 selects the item that has minimum weight. First select item 1, and next item 4. If we select third one as item 3, the total weight will be 5 kgs (1+2+2). This violates the 4 kgs constraint.

Method 3 chooses the item with maximum p_i/w_i that fits into the knapsack. We first compute p_i/w_i and make the following table.

item (i)	p_i	w_i	p_i/w_i
1	5	1	5
2	9	3	3
3	4	2	2
4	8	2	4

Sort items in decreasing order of p_i/w_i

item (i)	p_i	w_i	p_i/w_i
1	5	1	5
4	8	2	4
2	9	3	3
3	4	2	2

The order of items selected is 1 and 4. If we include item 2, then it violates the capacity constraint.

These three methods do not guarantee an optimal solution. This example illustrates the 0-1 knapsack problem which exhibits no greedy choice property and hence no greedy algorithm exists. Only dynamic programming algorithm exists.

$Number\ of\ objects;\ n = 3$
Capacity of Knapsack; M=20
$(p_1, p_2, p_3) = (25,24,15)$
$(w_1, w_2, w_3) = (18,15,10)$

To solve this problem, Greedy method may apply any one of the following strategies:

- From the remaining objects, select the object with maximum profit that fit into the knapsack.
- From the remaining objects, select the object that has minimum weight and also fits into knapsack.
- From the remaining objects, select the object with maximum p_i/w_i that fits into the knapsack.

Let us apply all above 3 approaches on the given knapsack instance:

Approach	(x_1, x_2, x_3)	$\sum_{i=1}^{3} w_i x_i$	$\sum_{i=1}^{3} p_i x_i$
1	$(1, \frac{2}{15}, 0)$	18+2+0=20	28.2
2	$< 0, \frac{2}{3}, 1 >$	0+10+10=20	31.0
3	$< 0, 1, \frac{1}{2} >$	0+15+5=20	31.5

Approach 1: (selection of object in decreasing order of profit):

In this approach, we select those object first which has maximum profit, then next maximum profit and so on. Thus we select 1st object (since its profit is 25, which is maximum among all profits) first to fill into a knapsack, now after filling this object ($w_1 = 18$) into knapsack remaining capacity is now 2 (i.e. 20-18=2). Next we select the 2nd object, but its weight w_2=15, so we take a fraction of this object (i.e. $x_2 = \frac{2}{15}$). Now knapsack is full (i.e. $\sum_{i=1}^{3} w_i x_i = 20$) so 3rd object is not selected. Hence we get total profit $\sum_{i=1}^{3} p_i x_i = 28$ units and the solution set $(x_1, x_2, x_3) = (1, \frac{2}{15}, 0)$

Approach 2: (Selection of object in increasing order of weights).
In this approach, we select those object first which has minimum weight, then next minimum weight and so on. Thus we select objects in the sequence 2nd then 3rd then 1st. In this approach we have total profit $\sum_{i=1}^{3} p_i x_i = 31.0$ units and the solution set $(x_1, x_2, x_3) = (0, \frac{2}{3}, 1)$.

Consider a knapsack problem of finding the optimal solution where, M=15, (p1,p2,p3...p7) = (10, 5, 15, 7, 6, 18, 3) and (w1, w2, ..., w7) = (2, 3, 5, 7, 1, 4, 1)
In order to find the solution, one can follow three different srategies.

Strategy 1 : non-increasing profit values

Let (a, b, c, d, e, f, g) represent the items with profit (10,5,15,7,6,18,3) then the sequence of objects with non-increasing profit is (f, c, a, d, e, b, g).

Item chosen for inclusion	Quantity of item included	Remaining space in M	PiXi
f	1 full unit	15-4=11	18*1=18
C	1 full unit	11-5=6	15*1=15
A	1 full unit	6-2=4	10*1=10
d	4/7 unit	4-4=0	4/7*7=04

Profit= 47 units
The solution set is (1,0,1,4/7,0,1,0).

Strategy 2: non-decreasing weights

The sequence of objects with non-decreasing weights is (e, g, a, b, f, c, d).

Item chosen for inclusion	Quantity of item included	Remaining space in M	PiXI
E	1 full unit	15-1=14	6*1=6
G	1 full unit	14-1=13	3*1=3
A	1 full unit	13-2=11	10*1=10
b	1 full unit	11-3=8	5*1=05
f	1 full unit	8-4=4	18*1=18
c	4/5 unit	4-4=0	4/5*15=12

Profit= 54 units
The solution set is (1,1,4/5,0,1,1,1).

a: P_1/w_1 =10/2 = 5
b: P_2/w_2 =5/3=1.66
c: P_3/w_3 =15/5 = 3
d: P_4/w_4 =7/7=1
e: P_5/w_5 =6/1=6
f: P_6/w_6 =18/4 = 4.5
g: P_7/w_7 =3/1=3

Hence, the sequence is (e,a,f,c,g,b,d)

Item chosen for inclusion	Quantity of item included	Remaining space in M	PiXI
E	1 full unit	15-1=14	6*1=6
A	1 full unit	14-2=12	10*1=10
F	1 full unit	12-4=8	18*1=18
C	1 full unit	8-5=3	15*1=15
g	1 full unit	3-1=2	3*1=3
b	2/3 unit	2-2=0	2/3*5=3.33

Profit= 55.33 units
The solution set is (1,2/3,1,0,1,1,1).

Approach 3: (Selection of object in decreasing order of the ratio p_i/w_i).
In this approach, we select those object first which has maximum value of p_i/w_i , that is we select those object first which has maximum profit per unit weight .
Since $(p_1/w_1, p_2/w_2, p_3/w_3)$=(1.3, 1.6, 1.5). Thus we select 2nd object first , then 3rd object then 1st object. In this approach we have total profit $\sum_{i=1}^{3} p_i x_i = 31.5$ units and the solution set $(x_1, x_2, x_3) = (0,1,\frac{1}{2},)$.

Thus from above all 3 approaches, it may be noticed that

- Greedy approaches **do not always yield an optimal solution**. In such cases the greedy method is frequently the basis of a heuristic approach.
- Approach3 **(**Selection of object in decreasing order of the ratio p_i/w_i) gives a optimal solution for knapsack problem.

Algorithm:

```
procedure GREEDY_KNAPSACK(P, W, M, X, n)
  //P(1:n) and W(1:n) contain the profits and weights respectively of the n//
  //objects ordered so that P(i)/W(i) ≥ P(i + 1)/W(i + 1). M is the//
  //knapsack size and X(1:n) is the solution vector//
  real P(1:n), W(1:n), X(1:n), M, cu;
  integer i, n;
  X ← 0   //initialize solution to zero//
  cu ← M   //cu = remaining knapsack capacity//
  for i ← 1 to n do
    if W(i) > cu then exit endif
    X(i) ← 1
    cu ← cu − W(i)
  repeat
  if i ≤ n then X(i) ← cu/W(i) endif
end GREEDY_KNAPSACK
```

Time complexity is O(n).

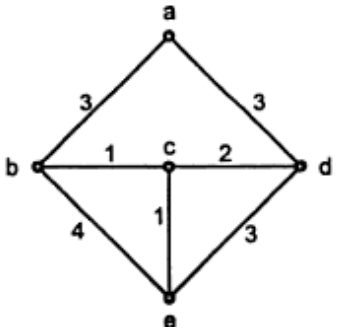
Spanning Tree

A Spanning tree S is a subset of a tree T in which all the vertices of tree T are present but it may not contain all the edges.

The minimum spanning tree of a weighted connected graph G is called minimum spanning tree if its weight is minimum.

We are going to discuss the two popular algorithms of minimum spanning tree, namely Prim's algorithm and Kruskal's algorithm.

1. Prim's Algorithm : In Prim's algorithm the pair with the minimum weight is to be chosen. Then adjacent to these vertices which ever is the edge having minimum weight is selected. This process will be continued till all the vertices are not be covered. The necessary condition in this case is that the circuit should not be formed. From figure we will build the minimum spanning tree as follows-



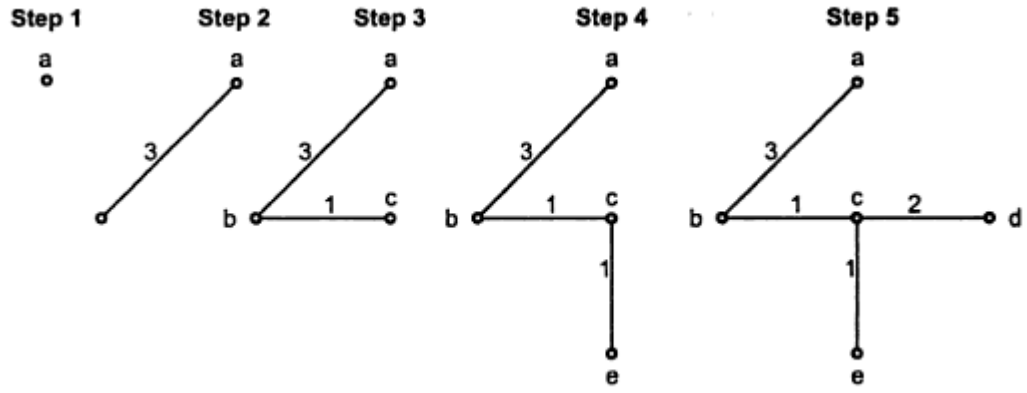


Fig. Prim's minimum spanning tree for 'g'

Minimum weight is 7

Prims algorithm

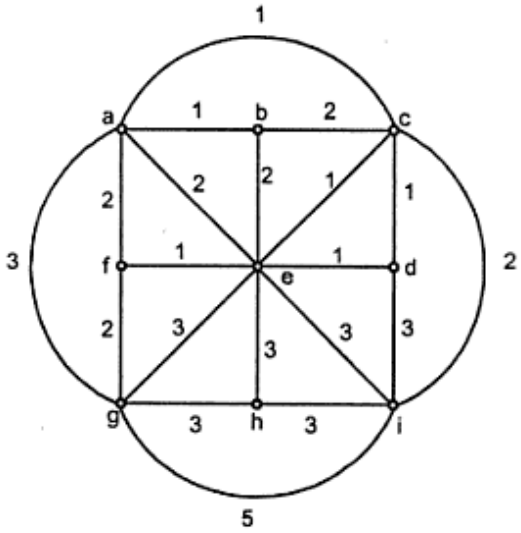
```
A = ∅
foreach v ∈ V:
   KEY[v] = ∞
   PARENT[v] = null
KEY[r] = 0
Q = V
while Q != ∅:
   u = min(Q) by KEY value
   Q = Q - u
   if PARENT(u) != null:
      A = A ∪ (u, PARENT(u))
   foreach v ∈ Adj(u):
     if v ∈ Q and w(u,v) < KEY[v]:
       PARENT[v] = u
       KEY[v] = w
return A
```

Time complexity= $O(n^2)$

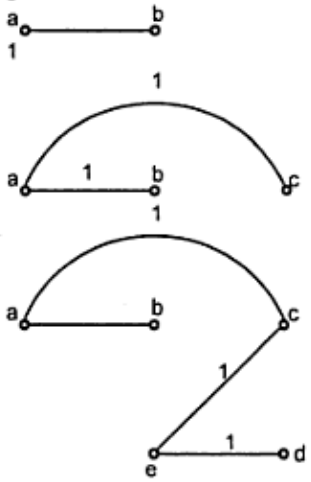
n=number of nodes

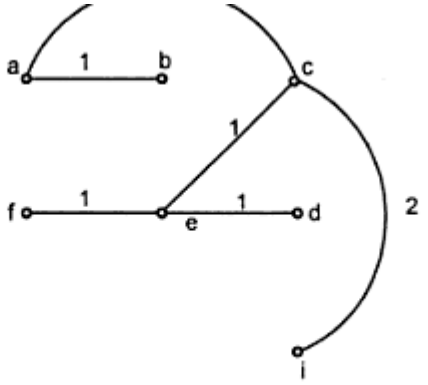
2. Kruskal's Algorithm : In Kruskal's algorithm the **minimum** weight is obtained. In this algorithm also the **circuit should not be formed**. Each time the edge of minimum weight has to be selected, from the graph.It is **not necessary** in this algorithm to have edges of minimum weights to be **adjacent**. Let us solve one example by Kruskal's algorithm.

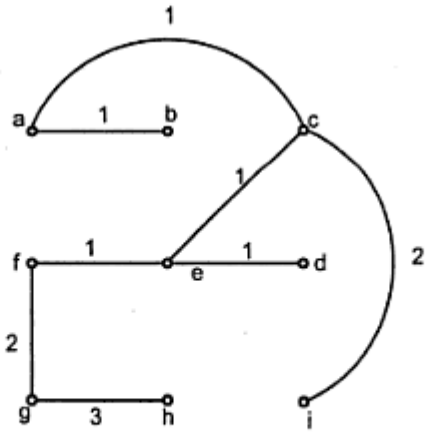
Find the minimum spanning tree for the following figure using Kruskal's algorithm.



In Kruskal's algorithm, we will start with some vertex, and will cover all the vertices with minimum weight. The vertices need not be adjacent.

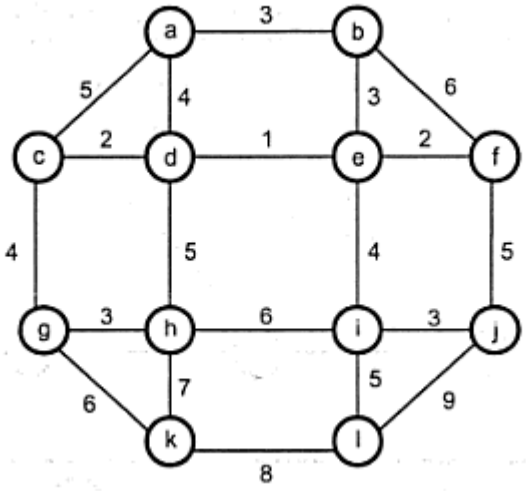


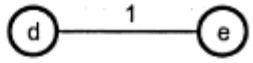
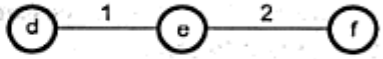
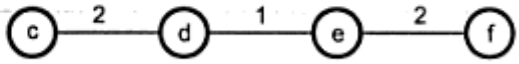
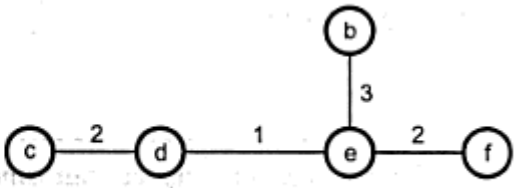
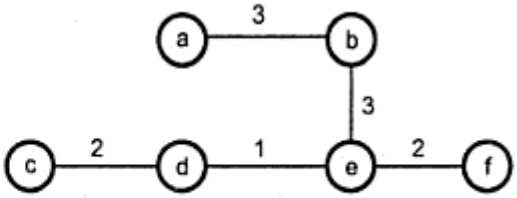
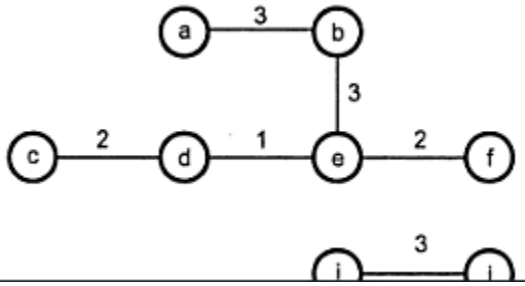




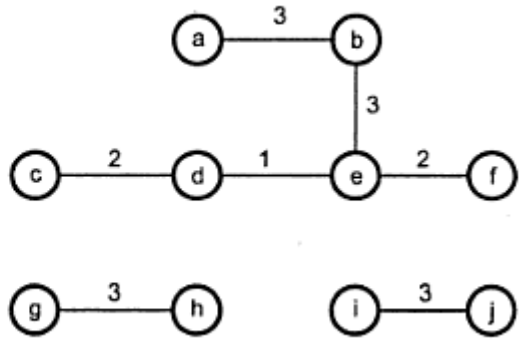
The total weight = 12

Apply Kruskal's algorithm to find minimum spanning tree of following graph.

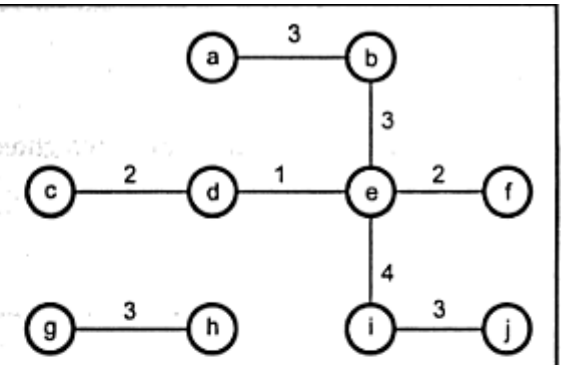
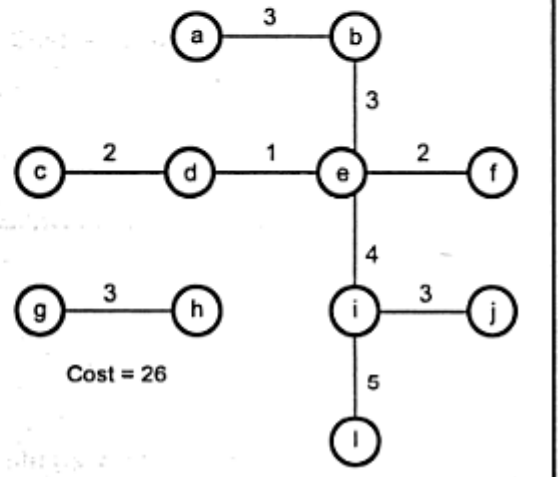


We first select an edge with minimum weight.	
Then we select the next minimum weighted edge. It is not necessary that selected edge is adjacent.	
Then select next minimum weighted edge.	 Cost = 5
Then select next minimum weighted edge.	 Cost = 8
Then select next minimum weighted edge.	 Cost = 11
Then select next minimum weighted edge.	

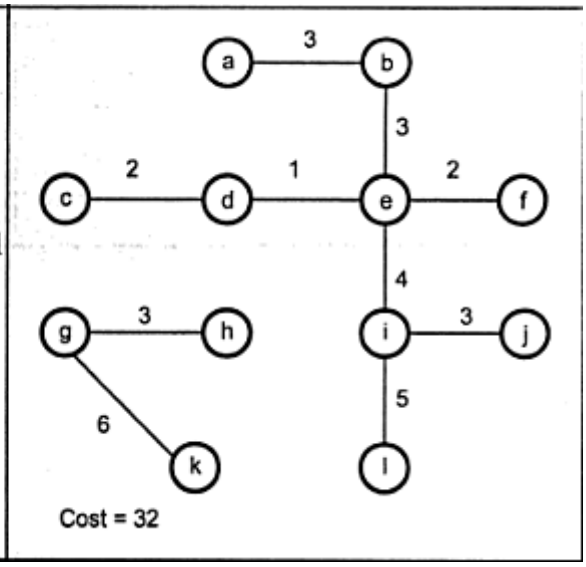
Cost=14

Then select next minimum weighted edge.	

Cost=17

Then select next minimum weighted edge in order to make the graph connected.	 Cost = 21
Then select next minimum weighted edge.	 Cost = 26

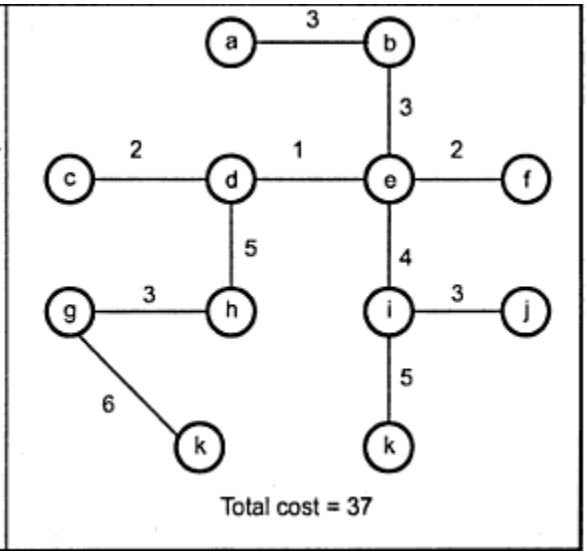
Then select next minimum weighted edge.



In order to make graph connected, select the minimum weighted edge.

Thus we get a spanning tree.

Total cost = 37



Kruskal algorithm

```
Kruskal(V, E)

A = ∅
foreach v ∈ V:
   Make-disjoint-set(v)
Sort E by weight increasingly
foreach (v1, v2) ∈ E:
   if Find(v1) ≠ Find(v2):
      A = A ∪ {(v1, v2)}
      Union(v1, v2)
return A
```

Time complexity=O(E log V)

E is total number of edges.

V=total number of vertices.

n=number of nodes

Single Source Shortest Path

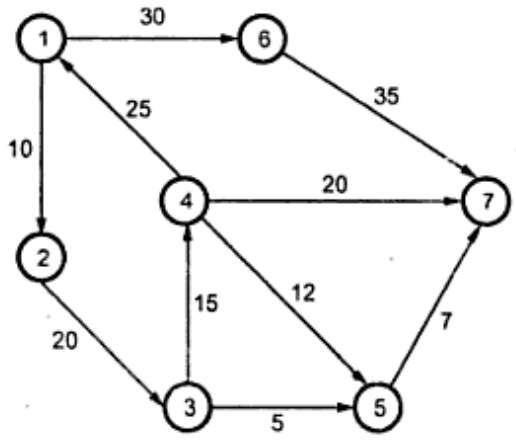
Many times, Graph is used to represent the distances between two cities. Everybody is often interested in moving from one city to other as quickly as possible. The single source shortest path is based on this interest.

In single source shortest path problem the shortest distance from a single vertex called source is obtained.

Let G(V, E) be a graph, then in single source shortest path the shortest paths from vertex v_0 to all remaining vertex is determined. The vertex v_0 is then called as source and the last vertex is called destination.

It is assumed that all the distances are positive.

Example : Consider a graph G as given below.



We will start from source vertex 1. Hence set S[1] = 1.

Now shortest distance from vertex 1 is 10. i.e. $1 \rightarrow 2 = 10$. Hence {1, 2} and min = 10.

From vertex 2 the next vertex chosen is 3.

{1, 2} = 10

{1, 3} = ∞

{1, 5} = ∞

{1, 6} = 30

{1, 7} = ∞

Now

{1, 2, 3} = 30

{1, 2, 4} = ∞

{1, 2, 7} = ∞

{1, 2, 5} = ∞

{1, 2, 6} = ∞

Hence select 3.

∴ S[3] = 1

Now

{1, 2, 3, 4} = 45

{1, 2, 3, 5} = 35

{1, 2, 3, 6} = ∞

{1, 2, 3, 7} = ∞

Hence select next vertex as 5.

∴ S[5] = 1

Now

{1, 2, 3, 5, 6} = ∞

{1, 2, 3, 5, 7} = 42

Hence vertex 7 will be selected. In single source shortest path if destination vertex is 7 then we have achieved shortest path 1 - 2 - 3 - 5 - 7 with path length 42. The single source shortest path from each vertex is summerised below -

1, 2	10
1, 2, 3	30
1, 2, 3, 4	45
1, 2, 3, 5	35
1, 2, 3, 4, 7	65
1, 2, 3, 5, 7	42
1, 6	30
1, 6, 7	65

```
    Procedure Dijk(u, Cost(N,N), Dist(N), N)
      /* Cost(i,j) = weight of edge (i,j), u = source vertex, Dist(i)
= Shortest distance of vertex i from u, N = number of vertices in
the graph */
    Array Cost(N,N), Dist(N)
      For i = 1 to N Do
        S(i) = 0; /* i is not included in the shortest path */
        Dist(i) = Cost(u,i);
          /* weight of edge (u, i) or ∞ if no edge exists, 0 if i = u */

 Endfor
 S(u) = 1; /* Vertex u is put in set S */

 For k = 2 to N Do
   Choose v such that Dist(v) = min{Dist(w)} and S(w)=0;
   S(v) = 1; /* put vertex v in set S */
 For all w with S(w) = 0 Do
   Dist(w) = min(Dist(w), Dist(v) + Cost(v,w));
 Endfor
 Endfor
End Dijk
```

Complexity analysis

The time taken by Dijkstra's algorithm on a graph with N vertices is $O(N^2)$ because the *k-loop* is repeated $N - 1$ times and for each repetition of *k-loop* v is searched from a list which is decreasing in length starting from $N - 1$ with every iteration of k. The time $T(N)$ may be expressed as $T(N) = (N - 1) + (N - 2) + \cdots + 1 = N(N - 1)/2$.

Dynamic Programming

Introduction

Dynamic programming is typically applied to optimization problem. This technique is invented by a U.S. Mathematician Richard Bellman in 1950. In the word dynamic programming the word programming stands for **planning** and it does not mean by computer programming.

- Dynamic programming is technique for solving problems with overlapping subproblems.
- In this method each subproblem is solved only once. The result of each subproblem is recorded in a table from which we can obtain a solution to the original problem.

Steps of Dynamic Programming

Dynamic programming design involves 4 major steps :

1. Characterize the structure of optimal solution. That means develop a mathematical notation that can express any solution and subsolution for the given problem.
2. Recursively define the value of an optimal solution.
3. By using bottom up technique compute value of optimal solution. For that you have to develop a recurrence relation that relates a solution to its subsolutions, using the mathematical notation of step 1.
4. Compute an optimal solution from computed information.

Principle of Optimality

The dynamic programming algorithm obtains the solution using principle of optimality.

The principle of optimality states that "in an optimal sequence of decisions or choices, each subsequence must also be optimal."

Difference between Divide and Conquer and Dynamic Programming

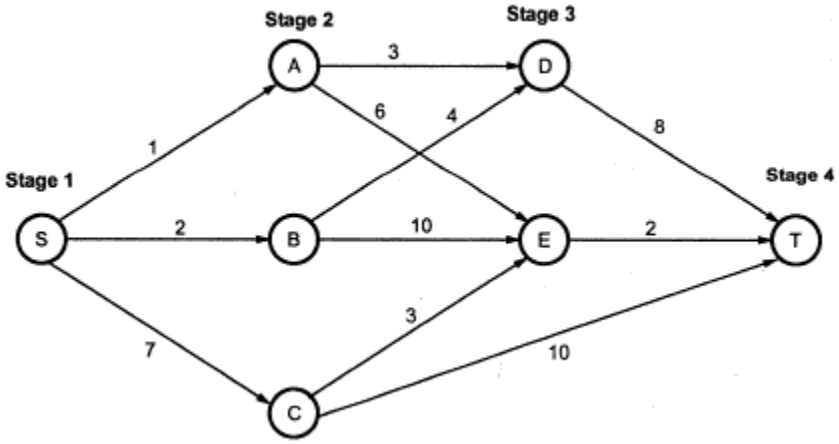
1.	The problem is divided into small subproblems. These subproblems are solved independently. Finally all the solutions of sub-problems are collected together to get the solution to the given problem.	In dynamic programming many decision sequences are generated and all the overlapping sub-instances are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

Comparison between Greedy Algorithm and Dynamic Programming

1.	Greedy method is used for obtaining **optimum solution.**	Dynamic programming is also for obtaining **optimum solution.**
2.	In Greedy method a set of **feasible solutions** and the picks up the optimum solution.	There is **no** special set of **feasible solutions** in this method.
3.	In Greedy method the optimum selection is **without revising previously** generated solutions.	Dynamic programming considers **all possible sequences** in order to obtain the optimum solution.
4.	In Greedy method there is **no** as such **guarantee** of getting **optimum solution.**	It is guaranteed that the dynamic programming will generate optimal solution using **principle of optimality.**

Multistage Graphs

A multistage graph G = (V, E) which is a directed graph. In this graph all the vertices are partitioned into the k stages where k ≥ 2. In multistage graph problem we have to find the shortest path from source to sink. The cost of each path is calculated by using the weight given along that edge. The cost of a path from source (denoted by S) to sink (denoted by T) is the sum of the costs of edges on the path. In multistage graph problem we have to find the path from S to T. There is set of vertices in each stage. The multistage graph can be solved using forward and backward approach. Let us solve multistage problem for both the approaches with the help of some example.



There is a single vertex in stage 1, then 3 vertices in stage 2, then 2 vertices in stage 3 and only one vertex in stage 4 (this is a target stage).

Backward approach

$$d(S, T) = \min \{1 + d(A, T), 2 + d(B, T), 7 + d(C, T)\} \quad \ldots (1)$$

We will now compute d(A, T), d(B, T) and d(C,T).

$$d(A, T) = \min \{3 + d(D, T), 6 + d(E, T)\} \quad \ldots (2)$$

$$d(B, T) = \min \{4 + d(D, T), 10 + d(E, T)\} \quad \ldots (3)$$

$$d(C, T) = \min \{3 + d(E, T), d(C, T)\} \quad \ldots (4)$$

Now let us compute d(D, T) and d(E, T).

$$\mathbf{d(D, T) = 8}$$

$$\mathbf{d(E, T) = 2} \qquad \textbf{backward vertex = E}$$

Let us put these values in equations (2), (3) and (4).

$$d(A, T) = \min \{3 + 8, 6 + 2\}$$

$$\mathbf{d(A, T) = 8} \qquad \textbf{A - E - T}$$

$$d(B, T) = \min \{4 + 8, 10 + 2\}$$

$$\mathbf{d(B, T) = 12} \qquad \textbf{A - D - T}$$

$$d(C, T) = \min \{3 + 2, 10\}$$

d(C, T) = 5 **C - E - T**

d(S, T) = min {1 + d(A, T), 2 + d(B, T), 7 + d(C, T)}
= min {1 + 8, 2 + 12, 7 + 5}
= min {9, 14, 12}
d(S, T) = 9 **S - A - E - T**

The path with minimum cost is S - A - E - T with the cost 9.

Forward approach

d(S, A) = 1
d(S, B) = 2
d(S, C) = 7
d(S, D) = min {1 + d(A, D), 2 + d(B, D)}
= min {1 + 3, 2 + 4}
d(S, D) = 4

d(S, E) = min {1 + d(A, E), 2 + d(B, E), 7 + d(C, E)}
= min {1 + 6, 2 + 10, 7 + 3}
= min {7, 12, 10}
d(S, E) = 7 i.e. **Path S - A - E is chosen.**

d(S, T) = min {d(S, D) + d(D, T), d(S, E) + d(E, T), d(S, C) + d(C, T)}
= min {4 + 8, 7 + 2, 7 + 10}
d(S, T) = 9 i.e. **Path S - E, E - T is chosen.**

∴ The minimum cost = 9 with the path **S - A - E - T.**

All Pairs Shortest Paths

Problem Description

When a weighted graph, 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 the all pairs shortest path.

Step 1 : We will decompose the given problem into subproblems.

Let,

$A^k_{(i,j)}$ be the length of shortest path from node i to node j such that the label for every intermediate node will be <= k.

We will compute A^k for k = 1 ..... n for n nodes.

Step 2 : For solving all pair shortest path, the principle of optimality is used. That means any subpath of shortest path is a shortest path between the end nodes. Divide the paths from i node to j node for every intermediate node, say 'k'. Then there arises two cases.

i) Path going from i to j via k.

ii) Path which is not going via k.

Select only shortest path from two cases.

Step 3 : The shortest path can be computed using bottom up computation method. Following is recursion method.

Initially : $A^0 = W[i, j]$

Next computations :

$$A^k_{(i,j)} = \min\left\{ A^{k-1}_{(i,j)}, A^{k-1}_{(i,k)} + A^{k-1}_{(k,j)} \right\} \qquad \text{where } 1 \le k \le n$$

Example : *Obtain all pair shortest paths for following graph.*

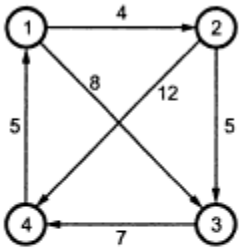


Fig.

Solution :

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 4 & 8 & \infty \\ \infty & 0 & 5 & 12 \\ \infty & \infty & 0 & 7 \\ 5 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 4 & 8 & \infty \\ \infty & 0 & 5 & 12 \\ \mathbf{12} & \infty & 0 & 7 \\ 5 & \mathbf{9} & \mathbf{13} & 0 \end{bmatrix}$$

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 4 & 8 & 16 \\ \mathbf{17} & 0 & 5 & 12 \\ 12 & \infty & 0 & 7 \\ 5 & 9 & 13 & 0 \end{bmatrix}$$

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 4 & 8 & 16 \\ 17 & 0 & 5 & 12 \\ 12 & \mathbf{16} & 0 & 7 \\ 5 & 9 & 13 & 0 \end{bmatrix}$$

Thus shortest distances between all pairs is obtained.

Algorithm

```
Algorithm All_Pair(W,A)
{
  for i=1 to n do
      for j=1 to n do
        A[i,j] := W[i,j]; //copy the weights as it is in matrix A
  for k=1 to n do
  {
   for i=1 to n do
       {
        for j=1 to n do
        {
         A[i,j]=min ( A[i,j],A[i,k] + A[k,j]);
        }
       }
   }
 }
```

Analysis

From above algorithm
The first double for-loop takes $O(n^2)$ time.
The nested three for loops take $O(n^3)$ time.
Thus, the whole algorithm takes $O(n^3)$ time.

Traveling Salesperson Problem

Problem Description

Let G be directed graph denoted by (V, E) where V denotes set of vertices and E denotes set of edges. The edges are given along with their cost C_{ij}. The cost $C_{ij} > 0$ for all i and j. If there is no edge between i and j then $C_{ij} = \infty$.

A tour for the graph should be such that all the vertices should be visited only once and cost of the tour is sum of cost of edges on the tour. The traveling salesperson problem is to find the tour of minimum cost.

Dynamic programming is used to solve this problem.

Step 1 : Let the function C(1, V–{1}) is the total length of the tour terminating at 1. The objective of TSP problem is that the cost of this tour should be minimum.

Let d[i, j] be the shortest path between two vertices i and j.

Step 2: Let $V_1, V_2 \ldots V_n$ be the sequence of vertices followed in optimal tour. Then $(V_1, V_2, \ldots V_n)$ must be a shortest path from V_1 to V_n which passes through each vertex exactly once.

Here the principle of optimality is used. The path $V_i, V_{i+1}, \ldots V_j$ must be optimal for all paths beginning at V(i), ending at V(j), and passing through all the intermediate vertices $\{V_{(i+1)} \ldots V_{(j-1)}\}$ once.

Step 3: Following formula can be used to obtain the optimum cost tour.

$Cost(i, S) = \min \{d[i, j] + Cost(j, S–\{j\})\}$ where $j \in S$ and $i \notin S$.

Consider one example to understand solving of TSP using dynamic programming approach.

Example : *For the given diagraph, obtain optimum cost Tour.*

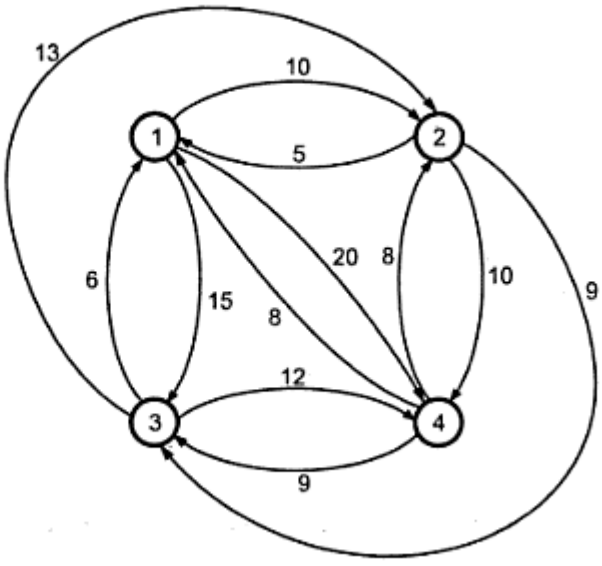


Fig.

Solution : The distance matrix can be given by,

from ↓ / to →	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

First we will select any arbitrary vertex say select 1.

Now process for intermediate sets with increasing size.

Step 1 : Let S = ϕ then,

$$\text{Cost}(2, \phi, 1) = d(2, 1) = 5$$

$$\text{Cost}(3, \phi, 1) = d(3, 1) = 6$$

$$\text{Cost}(4, \phi, 1) = d(4, 1) = 8$$

That means we have obtained dist (2, 1), dist (3, 1) and dist (4, 1).

Step 2 :

$$\text{Candidate (S)} = 1$$

Apply the formula,

$$\text{Cost}(i, S) = \min \{d[i, j] + \text{Cost}(j, S - \{j\})$$

Hence from vertex 2 to 1, vertex 3 to 1 and vertex 4 to 1 by considering intermediate path lengths we will calculate total optimum cost.

$$\text{Cost}(2, \{3\}, 1) = d(2, 3) + \text{Cost}(3, \phi, 1)$$

$$= 9 + 6$$

$$\text{Cost}(2, \{3\}, 1) = 15$$

$$\text{Cost}(2, \{4\}, 1) = d(2, 4) + \text{Cost}(4, \phi, 1)$$

$$= 10 + 8$$

$$\text{Cost}(2, \{4\}, 1) = 18$$

$$\text{Cost}(3, \{2\}, 1) = d(3, 2) + \text{Cost}(2, \phi, 1)$$

$$= 3 + 5$$

$$\text{Cost}(3, \{2\}, 1) = 18$$

$$\text{Cost}(3, \{4\}, 1) = d(3, 4) + \text{Cost}(4, \phi, 1)$$
$$= 12 + 8$$
$$\text{Cost}(3, \{4\}, 1) = 20$$
$$\text{Cost}(4, \{2\}, 1) = d(4, 2) + \text{Cost}(2, \phi, 1)$$
$$= 8 + 5$$
$$\text{Cost}(4, \{2\}, 1) = 13$$
$$\text{Cost}(4, \{3\}, 1) = d(4, 3) + \text{Cost}(3, \phi, 1)$$
$$= 9 + 6$$
$$\text{Cost}(4, \{3\}, 1) = 15$$

Step 3 : Consider candidate (S) = 2

$$\text{Cost}(2, \{3, 4\}, 1) = \min\{[d(2, 3) + \text{Cost}(3, \{4\}, 1)], [d(2, 4) + \text{Cost}(4, \{3\}, 1)]\}$$
$$= \min\{[9 + 20], [10+15]\}$$
$$\text{Cost}(2, \{3, 4\}, 1) = 25$$
$$\text{Cost}(3, \{2, 4\}, 1) = \min\{[d(3, 2) + \text{Cost}(2, \{4\}, 1)], [d(3, 4) + \text{Cost}(4, \{2\}, 1)]\}$$
$$= \min\{[13+18], [12+13]\}$$
$$\text{Cost}(3, \{2, 4\}, 1) = 25$$
$$\text{Cost}(4, \{2, 3\}, 1) = \min\{[d(4, 2) + \text{Cost}(2, \{3\}, 1)], [d(4, 3) + \text{Cost}(3, \{2\}, 1)]\}$$
$$= \min\{[8 + 15], [9 + 18]\}$$

$$\text{Cost}(4, \{2, 3\}, 1) = 23$$

Step 4 : Consider candidate (S) = 3. i.e. Cost (1, {2, 3, 4}) but as we have chosen vertex 1 initially the cycle should be completed i.e. starting and ending vertex should be 1.

∴ We will compute,

$$\text{Cost}(1, \{2, 3, 4\}, 1) = \min\begin{Bmatrix}[d(1,2) + \text{Cost}(2, \{3,4\},1)], [d(1,3) + \text{Cost}(3, \{2,4\},1)] \\ [d(1,4) + \text{Cost}(4, \{2,3\},1)]\end{Bmatrix}$$
$$= \min([10 + 25], [15 + 25], [20 + 23])$$
$$= 35$$

Thus the optimal tour is of path length 35.

Consider step 4 now, from vertex 1 we obtain the optimum path as d(1, 2). Hence select vertex 2. Now consider step 3, in which from vertex 2 we obtain optimum cost from d(2, 4). Hence select vertex 4. Now in step 2 we get remaining vertex 3 as d(4, 3) is optimum. Hence optimal tour is 1, 2, 4, 3, 1.

Time complexity

Let the vertices of the graph be numbered 1 through n. Without loss of generality, we regard a tour to be a simple path that starts and ends at the vertex 1. Every tour consists of an edge

$(1, k)$ for some $k \in V - \{1\}$ and a path from vertex k to vertex 1. The path from vertex k to vertex 1 goes through each vertex in $V - \{1, k\}$ exactly once.

If the tour is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices in $V - \{1, k\}$. Hence the principle of optimality holds. Let $sp(i, S)$ be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at the vertex 1.

The function $sp(1, V - \{1\})$ is the length of an optimal salesperson tour. From the principle of optimality, it follows that

$$sp(1, V - \{1\}) = \min_{2 \le k \le n} \{W_{1k} + sp(k, V - \{1, k\})\} \quad (4.1)$$

where W_{ij} represents the length/cost of the edge (i, j) and n is the number of cities.

Generalizing, we obtain (for $i \notin S$),

$$sp(i, S) = \min_{j \in S} \{W_{ij} + sp(j, S - \{j\})\} \quad (4.2)$$

Equation (4.1) can be solved for $sp(1, V - \{1\})$ if we know $sp(k, V - \{1, k\})$ for all choices of k. The sp values are obtained by using Eq. (4.2). Clearly, $sp(i, \phi) = W_{i,1}$, $1 \le i \le n$. Hence, we can use Eq. (4.2) to obtain $sp(i, S)$ for all S of size 1. Then we can obtain $sp(i, S)$ for S with $|S| = 2$ and so on. When $|S| < n - 1$, the values of i and S for which $sp(i, S)$ is needed are such that $i \neq 1$, $1 \notin S$, and $i \notin S$.

Complexity analysis

Let M be the number of $sp(i, S)$'s that have to be computed before Eq. (4.1) can be used to compute $sp(1, V - \{1\})$. For each value of $|S|$ there are $n - 1$ choices for i. The number of distinct sets S of size k not including 1 and i is ${}^{n-2}C_k$. Hence:

$$M = \sum_{k=0}^{n-2} (n - 1) \binom{n-2}{k} = (n - 1)2^{n-2}$$

The computation of $sp(i, S)$ with $|S| = k$ requires $k - 1$ comparisons when solving Eq. (4.2). The time T required to find an optimal tour using Eqs. (4.1) and (4.2) is

$$T = \sum_{k=1}^{k=n-2} (k - 1)(n - 1)\, {}^{n-2}C_k = (n - 1) \sum_{k=1}^{k=n-2} (k - 1)\, {}^{n-2}C_k$$

But $$\sum_{k=1}^{k=n-2} (k - 1)\, {}^{n-2}C_k = \sum_{k=1}^{k=n-2} k\, {}^{n-2}C_k - \sum_{k=1}^{k=n-2} {}^{n-2}C_k$$

$$= \sum_{k=1}^{k=n-2} (n - 2)\, {}^{n-3}C_{k-1} - \sum_{k=1}^{k=n-2} {}^{n-2}C_k,$$

since for integer k, $k\, {}^{r}C_k = r\, {}^{r-1}C_{k-1}$

$$= (n - 2) \sum_{k=1}^{k=n-2} {}^{n-3}C_{k-1} - (2^{n-2} - 1), \text{ since } \sum_{k=0}^{k=n} {}^{n}C_k = 2^n$$

$$= (n - 2)2^{n-3} - 2^{n-2} + 1$$

Hence, $$T = (n - 1)\left\{\frac{(n - 2)2^n}{8} - \frac{2^n}{4} + 1\right\} = \frac{(n - 1)(n - 2)2^n}{8} - \frac{(n - 1)2^n}{4} + (n - 1)$$

$$= O(n^2 2^n)$$

0/1 Knapsack Problem

Problem Description : If we are given n objects and a Knapsack or a bag in which the object i that has weight w_i is to be placed. The Knapsack has a capacity W. Then the profit that can be earned is $p_i x_i$. The objective is to obtain filling of Knapsack with maximum profit earned.

$$\text{maximized } \sum_{n}^{1} p_i x_i \text{ subject to constraint } \sum_{n}^{1} w_i x_i \leq W$$

where $1 \leq i \leq n$ and n is total number of objects. And $x_i = 0$ or 1

The greedy method does not work for this problem.

To solve this problem using dynamic programming method we will perform following steps.

Step 1 : The notations used are

Let,

$f_i(y_j)$ be the value of optimal solution.

Then S^i is a pair (p, w) where $p = f(y_j)$ and $w = y_j$

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

We can compute S^{i+1} from S^i

These computations of S^i are basically the sequence of decisions made for obtaining the optimal solutions.

Step 2 : We can generate the sequence of decisions in order to obtain the optimum selection for solving the Knapsack problem.

Let x_n be the optimum sequence. Then there are two instances $\{x_n\}$ and $\{x_{n-1}, x_{n-2} \ldots x_1\}$. So from $\{x_{n-1}, x_{n-2} \ldots x_1\}$ we will choose the optimum sequence with respect to x_n. The selection of sequence from remaining set should be such that we should be able to fulfill the condition of filling Knapsack of capacity W with maximum profit. Otherwise $x_n, \ldots x_1$ is not optimal.

This proves that 0/1 Knapsack problem is solved using principle of optimality.

Step 3 : The formulae that are used while solving 0/1 Knapsack is

Let, $f_i(y_j)$ be the value of optimal solution. Then

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

Initially compute

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

$$S_1^i = \{(P, W) \mid (P - p_i, W - w_i) S^i\}$$

S_1^{i+1} can be computed by merging S^i and S_1^i

Purging rule

If S^{i+1} contains (P_j, W_j) and (P_k, W_k); these two pairs such that

$P_j \leq P_k$ and $W_j \geq W_k$, then (P_j, W_j) can be eliminated. This purging rule is also called as dominance rule. In purging rule basically the dominated tuples gets purged. In short, remove the pair with less profit and more weight.

Example 4.3 : *Solve Knapsack instance $M = 8$, and $n = 4$. Let P_i and W_i are as shown below.*

i	P_i	W_i
1	1	2
2	2	3
3	5	4
4	6	5

Solution : Let us build the sequence of decision S^0, S^1, S^2.

$$S^0 = \{(0, 0)\} \text{ initially}$$

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

That means while building S_1^0 we select the next i^{th} pair. For S_1^0 we have selected first (P, W) pair which is (1, 2).

Now

$$S^1 = \{\text{Merge } S^0 \text{ and } S_0^1\}$$

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

$$S_1^1 = \{\text{ Select next (P, W) pair and add it with } S^1\}$$

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

S_1^1 = {(2, 3), (3, 5)} ∵ Repetition of (2, 3) is avoided.

S^2 = {Merge candidates from S^1 and S_1^1}

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

S_1^2 = {Select next (P, W) pair and add it with S^2}

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

Now S^3 = {Merge candidates from S^2 and S_1^2}

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

Note that the pair (3, 5) is purged from S^3. This is because, let us assume $(P_j, W_j) = (3, 5)$ and $(P_k, W_k) = (5, 4)$. Here $P_j \le P_k$ and $W_j > W_k$ is true hence we will eliminate pair (P_j, W_j) i.e. (3, 5) from S^3.

S_1^3 = {(6, 5), (7, 7), (8, 8), (11, 9), (12, 11), (13, 12), (14, 14)}

S^4 = {(0,0), (1,2), (2,3), (5,4),(6,6),(7,7), (8,9)
(6,5), (8,8), (11,9), (12,11), (13,12),(14,14)}

Now we are interested in M = 8. We get pair (8, 8) in S^4. Hence we will set $x_4 = 1$. Now to select next object $(P - P_4)$ and $(W - W_4)$.

i.e (8 – 6) and (8 – 5).

i.e. (2, 3)

Pair (2, 3) ∈ S^2. Hence set x_2 =1. So we get the final solution as (0, 1, 0, 1).

Example : Consider the Knapsack for the instance n = 4. (W_1, W_2, W_3, W_4) = (10, 15, 6, 9) and (P_1, P_2, P_3, P_4) = (2, 5, 8, 1) and m = 30 then we can generate sequence of decisions.

$S^i = \{P_i, W_i\}$

S^0 = {(0, 0)}, S_1^0 = {(2, 10)}

S^1 = {(0, 0), (2, 10)}; S_1^1 = {(5, 15), (7, 25)}

S^2 = {(0, 0), (2, 10), (5, 15), (7, 25)}; S_1^2 = {(8, 6), (10, 16), (13, 21), (15, 31)}

S^3 = {(0, 0), (8, 6), (10, 16), (13, 21) (15, 31) }; By pruging rule (7, 25), (2, 10) and (5, 15) can be eliminated.

S_1^3 = {(1, 9), (9, 15), (11, 25), (14, 30), (16, 40)}

S^4 = {(0, 0), (8, 6), (1, 9) (9, 15), (11, 25), (14, 30) (16, 40)}; We eliminated (10, 16), (13, 21) and (15, 31)

Now for m = 30 we will search for a tuple in which value of W is 30. We obtain such a tuple (14, 30) in S^4. Hence set $x_4 = 1$.

∴ We will do (13, 21) – (P_3, W_3).

i.e. (14 – 1) and 30–9.

We will get (13, 21). As 13, 21 ∈ S^3 set $x_3 = 1$.

Now ((13 – 80), (21 – 6))= (5, 15). We will search for

pair (5, 15). It ∈ S^2. Hence set $x_2 = 1$

Hence the final solution is (0, 1, 1, 1). In other words select item x_2, x_3 and x_4 for the Knapsack.

EXAMPLE Consider $n = 3$, $(w_1, w_2, w_3) = (2, 3, 3)$, $(p_1, p_2, p_3) = (1, 2, 4)$ and $m = 6$. For these data we have:

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

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

$S^2 = \{(0,0), (1,2), (2,3), (3,5)\}$; $S_1^2 = \{(4,3), (5,5), (6,6)\}$

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

Note that the pair (3,5) has been eliminated from S^3 as a result of the purging rule. When generating the S^i's, we can also purge all pairs (P, W) with $W > m$ as these pairs determine the value of $f_n(y)$ only for $y > m$. Since the knapsack capacity is m, we are not interested in the behavior of f_n for $y > m$. When all pairs (P_j, W_j) with $W_j > m$ are purged from the S^i's, $f_n(m)$ is given by the P value of the last pair in S^n (note that S^i's are ordered sets). By computing S^n, we can find the solutions to all the knapsack problems *KNAPSACK* $(1, n, y)$, $0 \le y \le m$ and not just *KNAPSACK*$(1, n, m)$. Since, we want a solution only to *KNAPSACK*$(1, n, m)$, we can dispense with the computation of S^n. The last pair in S^n is

either the last one in S^{n-1} or it is $(P_j + p_n, W_j + w_n)$, where $(P_j, W_j) \in S^{n-1}$ such that $W_j + w_n \le m$ and W_j is maximum. If $(P1, W1)$ is the last tuple in S^n, a set of 0/1 values for the x_i's such that $\Sigma p_i x_i = P1$ and $\Sigma w_i x_i = W1$ can be determined by carrying out a search through the S^i's. We can set $x_n = 0$ if $(P1, W1) \in S^{n-1}$. If $(P1, W1) \notin S^{n-1}$, then $(P1 - p_n, W1 - w_n) \in S^{n-1}$ and we can set $x_n = 1$. This leaves us to determine how either $(P1, W1)$ or $(P1 - p_n, W1 - w_n)$ was obtained in S^{n-1}. This can be done recursively. With $m = 6$, the value of $f_3(6)$ is given by the tuple (6, 6) in S^3. The tuple (6, 6) $\notin S^2$, and so we must set $x_3 = 1$. The pair (6, 6) came from the pair $(6 - p_3, 6 - w_3) = (2, 3)$ and hence, $(2, 3) \in S^2$. Since $(2, 3) \notin S^1$, we can set $x_2 = 1$ and $x_1 = 0$. Hence the optimal solution is $(x_1, x_2, x_3) = (0, 1, 1)$.

Time complexity

Considering all subsets of the set of n items, computing the total weight of each subset and finding a feasible subset (not all subsets are feasible) of maximum profit leads to a $O(2^n)$ Let S^i represent the possible states resulting from the 2^i decision sequences for $x_1, \ldots, x_i$. A state refers to a pair (P_j, W_j), where W_j is the total weight of objects included in the knapsack and P_j is the corresponding profit. We note that $S^0 = \{(0, 0)\}$. To obtain S^{i+1}, we note that the possibilities for x_{i+1} are $x_{i+1} = 0$ or $x_{i+1} = 1$. When $x_{i+1} = 0$, the resulting states are the same as for S^i. When $x_{i+1} = 1$, the resulting states are obtained by adding (P_{i+1}, W_{i+1}) to each state in S^i. We call this set of additional states S_1^i. Now, S^{i+1} can be computed by merging the states in S^i and S_1^i together. If S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \le P_k$ and $W_j \ge W_k$, then the pair (P_j, W_j) can be discarded because the other pair gives more profit with no additional weight.

Complexity analysis

Each S^i, $i > 0$ is obtained by merging S^{i-1} and S_1^{i-1}. $|S_1^{i-1}| \le |S^{i-1}|$ because of the presence of dominated pairs in S_1^{i-1}, $|S^i| \le 2|S^{i-1}|$. In the worst-case no pair is removed. The time required to generate S^i from S^{i-1} is $|S^{i-1}| + |S_1^{i-1}| = O(|S^{i-1}|)$. So the time required to compute all the S^is, $1 \le i \le n$, is $O(\Sigma|S^{i-1}|) = O(2^n)$, since in the worst-case $|S^i| = 2^i$.

RELIABILITY DESIGN PROBLEM

In this section, we present the dynamic programming approach to solve a problem with multiplicative constraints. Let us consider the example of a computer network, in which a set of nodes are connected with each other. Let r_i be the reliability of a node, i.e., the probability at which the node forwards the packets correctly is r_i. Then, the reliability of the path connecting from one node s to another node d is $\prod_{i=1}^{k} r_i$, where k is the number of intermediate nodes. Similarly, we can

also consider a system with n devices connected in series, where the reliability of device i is r_i. The reliability of the system is $\prod_{i=1}^{k} r_i$. For example, if there are 5 devices connected in series and the reliability of each device is 0.99, then the reliability of the system is 0.99 0.99 0.99 0.99 0.99

= 0.951. Hence, it is desirable to connect multiple copies of the same devices in parallel through the use of switching circuits. The switching circuits determine the devices in any group functions properly. Then they make use of one such device at each stage.

Let m_i be the number of copies of device D_i in stage i, then the probability that all m_i have malfunctions is $(1 - r_i)^{m_i}$. Hence, the reliability of stage i becomes $1 - (1 - r_i)^{m_i}$. Thus, if $r_i = 0.99$ and $m_i = 2$, then the reliability of stage i is 0.9999. However, in practical situations, it becomes less because the switching circuits are not fully reliable. Let us assume that the reliability of stage i in $\phi_i(m_i)$, $i \le n$. Thus the reliability of the system is $\prod_{i=1}^{n} \phi_i(m_i)$, where n is the number of stages.

Figure shows n devices connected in series and Figure 7.10 shows the multiple copies of the devices connected in parallel at each stage.

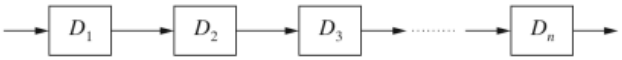


Figure n devices $D_1, D_2, \ldots, D_n$ connected in series.

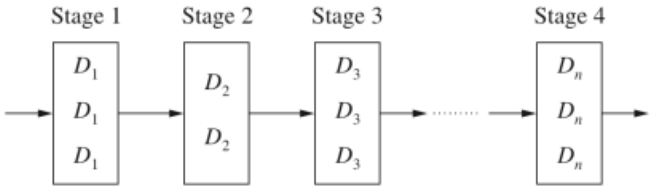


Figure Multiple copies of the devices connected in parallel at each stage.

The reliability design problem is to use multiple copies of the devices at each stage to increase reliability. However, this is to be done under a cost constraint. Let c_i be the cost of each unit of device D_i, and let c be the cost constraint. Then the objective is to maximise the reliability under the condition that the total cost of the system will be less than c. Mathematically, we can write

$$\text{Maximise} \prod_{1 \le i \le n} \phi_i(m_i)$$

$$\text{Subject to} \sum_{1 \le i \le n} c_i m_i \le c$$

$$m_i \ge 1 \text{ and } 1 \le i \le n$$

We can assume that each $c_i > 0$, and so each m_i, must be in the range $1 \le m_i \le u_i$, where

$$u_i = \left\lfloor \frac{c - \sum_{j=1 \text{ and } j \ne i}^{n} c_j}{c_i} \right\rfloor$$

The dynamic programming approach finds the optimal solution for $m_1, m_2, \ldots, m_n$. An optimal sequence of decision, i.e., a decision for each m_i, can result an optimal solution.

Let $f_n(c)$ be the maximum reliability of the system, i.e., maximum $\prod_{i=1}^{n} \phi_i(m_i)$, subject to the constraint $\sum_{1 \le i \le n} c_i m_i \le c$ and $1 \le m_i \le u_i$, $1 \le i \le n$. Let us take a decision on the value of m_n from $\{1, 2, \ldots, u_n\}$. Then,the value of the remaining m_n, $1 \le i \le n$, can be chosen in such a way that $\phi_i(m_i)$ for $1 \le i \le n-1$ can be maximised under the cost constraint $c - c_n m_n$. Thus, the principle of optimality holds and we can write

$$f_n(c) = \max_{1 \le m_n \le u_n} \{\phi_n(m_n)\, f_{n-1}(c - m_n c_n)\} \qquad 1)$$

We can generalise Eq. (1) as

$$f_j(x) = \max_{1 \le m_j \le u_j} \{\phi_j(m_j)\, f_{j-1}(x - m_j c_j)\} \qquad (\ 2)$$

It is clear that $f_0(x) = 1$ for all x, $0 \le x \le c$. Let S^i consists of tuples of the form (f, x) where $f = f_i(x)$. There is at most one tuple for each different x that remits from a sequence of decisions on $m_1, m_2, \ldots, m_n$. If there are two tuples (f_1, x_1) and (f_2, x_2) such that $f_1 \ge f_2$ and $x_1 \le x_2$ then (f_2, x_2) is said to be dominated tuple and discarded from S^i.

Let us design a three-stage system with devices D_1, D_2 and D_3 having costs ₹ 30, ₹ 15 and ₹ 20 respectively. The cost constraint of the system is ₹ 105. The reliabilities of the devices D_1, D_2 and D_3 are 0.9, 0.8 and 0.5 respectively. If stage i has m_i devices in parallel, then

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

We can write $c_1 = 30$, $c_2 = 15$, $c_3 = 20$, $c = 105$, $r_1 = 0.9$, $r_2 = 0.8$ and $r_3 = 0.5$. We can calculate the value of u_i, for $1 \le i \le 3$.

$$x_1 = \left\lfloor \frac{105 - (15 + 20)}{30} \right\rfloor = \left\lfloor \frac{70}{30} \right\rfloor = 2$$

$$x_2 = \left\lfloor \frac{105 - (30 + 20)}{15} \right\rfloor = \left\lfloor \frac{55}{15} \right\rfloor = 3$$

$$x_3 = \left\lfloor \frac{105 - (30 + 15)}{20} \right\rfloor = \left\lfloor \frac{60}{20} \right\rfloor = 3$$

Then, we start with $S^0 = \{(1, 0)\}$. We can obtain each S^i from S^{i-1} by trying out all possible values for m_i and combining the resulting tuples together.

$S_1^1 = \{(0.9,30)\}$, $S_2^1 = \{(0.99,60)\}$ and S^1 is obtained after combining S_1^1 and S_2^1.

Thus $S^1 = S_1^1 \cup S_2^1 = \{(0.9,30),\ (0.99,60)\}$

are the tuples at stage 1 with 1 and 2 devices connected in parallel respectively.

Considering 1 device at stage 1, we can write as follows:

$$S_1^2 = \{(0.9 \times 0.8, 30 + 15), (0.99 \times 0.8, 60 + 15)\}$$

$$= \{(0.72, 45), (0.792, 75)\}$$

Considering 2 devices of D_2 at stage 2, we can compute the reliability as

$$\phi_2(m_2) = 1 - (1 - 0.8)^2 = 0.96, \quad \text{Cost at stage } 2 = 2 \times 15 = 30$$

Hence, we can write

$$S_2^2 = \{(0.9 \times 0.96, 30 + 30), (0.99 \times 0.96, 60 + 30)\}$$
$$= \{(0.864, 60), (0.9504, 90)\}$$

The tuple (0.9504, 90) is removed, as it left only ₹ 15 and the maximum cost of the third stage is 20.

Now, we can consider 3 devices of D_2 at stage 2 and compute the reliability as

$$\phi_2(m_2) = 1 - (1 - 0.8)^3 = 1 - 0.008 = 0.992.$$

Hence, we can write

$$S_3^2 = \{(0.9 \times 0.992, 30 + 45), (0.99 \times 0.992, 60 + 45)\}$$
$$= \{(0.8928, 75), (0.98208, 105)\}$$

The tuple (0.98208, 105) is discarded as there is no cost left for stage 3. Combining S_1^2, S_2^2 and S_3^2, we get

$$S^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.8928, 75)\}$$

The tuple (0.792, 75) is discarded from S^2, as it is dominated by (0.864, 60).

Now, we can compute S_1^3 assuming 1 device at Stage 3. Thus,

$$S_1^3 = \{(0.72 \times 0.5, 45 + 20), (0.864 \times 0.5, 60 + 20), (0.8928 \times 0.5, 75 + 20)\}$$
$$= \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$$

If there are 2 devices at Stage 3, then

$$\phi_3(m_3) = (1 - (1 - 0.5)^2) = 0.75$$

We can write S_2^3 as follows:

$$S_2^3 = \{(0.72 \times 0.75, 45 + 40), (0.864 \times 0.75, 60 + 40), (0.8928 \times 0.75, 75 + 40)\}$$

$$= \{(0.54, 85), (0.648, 100)\}$$

(tuple $(0.8928 \times 0.75, 115)$ is discarded as cost constraint is 105).

If there are 3 devices at Stage 3, then

$$\phi_3(m_3) = (1 - (1 - 0.5)^3) = 1 - 0.125 = 0.875$$

Hence, we can write

$$S_3^3 = \{(0.72 \times 0.875, 45 + 60)\} = \{(0.63, 105)\}$$

Combining S_1^3, S_2^3 and S_3^3, we can write S^3 discarding the dominant tuples as given below.

$$S^3 = \{(0.36, 65), (0.432, 80), (0.54, 85), (0.648, 100)\}$$

The best design has the reliability 0.648 and a cost of ₹100. Now, we can track back to find the number of devices at each stage. The tuple (0.648, 100) is taken from S_2^3 that is with 2 devices at stage 2. Thus $m_3 = 2$. The tuple (0.648, 100) was derived from the tuple (0.864, 60) taken from S_2^2 and computed with considering 2 devices at stage 2. Thus $m_2 = 2$. The tuple (0.864, 60) is derived

from the tuple (0.9, 30) taken from S_1^1 and computed with 1 device at Stage 1. Thus $m_1 = 1$.

m1=1,m2=2,m3=2 as a solution to reliability design.

Optimal binary search tree

Consider 4 elements $a_1 < a_2 < a_3 < a_4$ with $q_0 = 0.25$, $q_1 = 3/16$, $q_2 = q_3 = q_4 = 1/16$. $p_1 = 1/4$, $p_2 = 1/8$, $p_3 = p_4 = 1/16$.

a) Construct the optimal binary search tree as a minimum cost tree.

b) Construct the table of values W_{ij}, C_{ij}, V_{ij} computed by the algorithm to compute the roots of optimal subtrees.

For convenience we will multiply the probabilities q_i and p_i by 16. Hence p_i and q_i values are $(p_1, p_2, p_3, p_4) = (4, 2, 1, 1)$.

$$(q_0, q_1, q_2, q_3, q_4) = (4, 3, 1, 1, 1)$$

Let us compute values of W first. For W_{ii} computation we will use,

$$W_{ii} = q_i$$

Hence

$$W_{00} = q_0$$

$$\therefore \quad W_{00} = 4$$

$$W_{11} = q_1$$

$$\therefore \quad W_{11} = 3$$

$$W_{22} = q_2$$

$\therefore \quad W_{22} = 1$

$W_{33} = q_3$

$\therefore \quad W_{33} = 1$

$W_{44} = q_4$

$\therefore \quad W_{44} = 1$

Compute W_{ii+1} using formula

$$W_{ii+1} = q_i + q_{i+1} + p_{i+1}$$

Hence,

$W_{01} = q_0 + q_1 + p_1$

$= 4 + 3 + 4$

$\therefore \quad W_{01} = 11$

$W_{12} = q_1 + q_2 + p_2$

$= 3 + 1 + 2$

$\therefore \quad W_{12} = 6$

$W_{23} = q_2 + q_3 + p_3$

$= 1 + 1 + 1$

$\therefore \quad W_{23} = 3$

$W_{34} = q_3 + q_4 + p_4$

$= 1 + 1 + 1$

$\therefore \quad W_{34} = 3$

For computing W_{ij} we will use following formula

$$W_{ij} = W_{ij-1} + p_j + q_j$$

$\therefore \quad W_{02} = W_{01} + p_2 + q_2$

$= 11 + 2 + 1$

$\therefore \quad W_{02} = 14$

$\therefore \quad W_{13} = W_{12} + p_3 + q_3$

$= 6 + 1 + 1$

$\therefore \quad W_{13} = 8$

$\therefore \quad W_{24} = W_{23} + p_4 + q_4$

$= 3 + 1 + 1$

$\therefore \quad W_{24} = 5$

Now,

$W_{03} = W_{02} + p_3 + q_3$

$= 14 + 1 + 1$

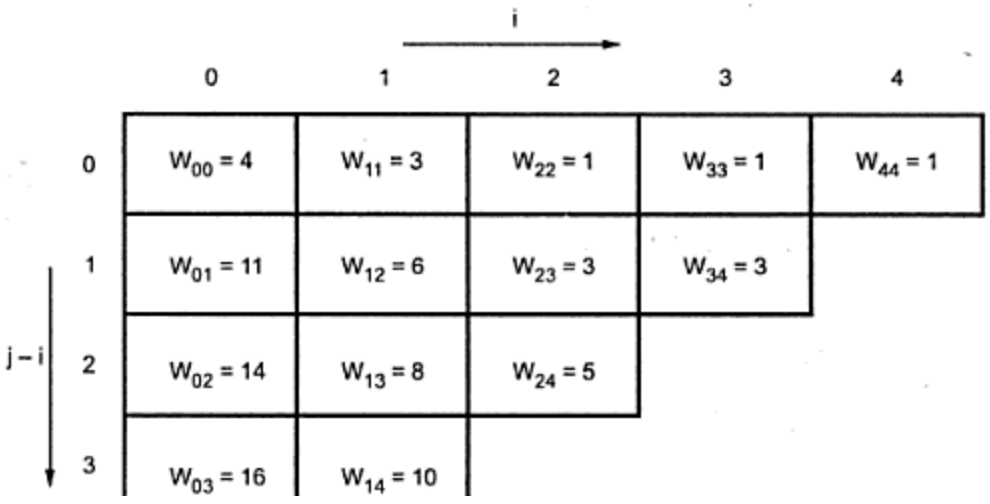
$\therefore \quad W_{03} = 16$

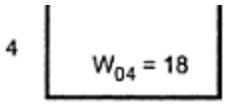
$W_{04} = W_{03} + p_4 + q_4$

$= 16 + 1 + 1$

$\therefore \quad W_{04} = 18$

To summarize W values





To compute C and r values we will use $C_{ii} = 0$ and $r_{ii} = 0$.

Hence

$C_{00} = 0$
$r_{00} = 0$
$C_{11} = 0$
$r_{11} = 0$
$C_{22} = 0$
$r_{22} = 0$
$C_{33} = 0$
$r_{33} = 0$
$C_{44} = 0$
$r_{44} = 0$

To compute C_{ii+1} and r_{ii+1} we will use following formulae

$$r_{i\,i+1} = i + 1$$

$$C_{i\,i+1} = q_i + q_{i+1} + p_{i+1}$$

Hence

$$r_{01} = 1$$

$$C_{01} = q_0 + q_1 + p_1$$
$$= 4 + 3 + 4$$

$\therefore$ $C_{01} = 11$

$$r_{12} = 2$$

$$C_{12} = q_1 + q_2 + p_2$$
$$= 3 + 1 + 2$$

$\therefore \quad C_{12} = 6$

$r_{23} = 3$

$C_{23} = q_2 + q_3 + p_3$

$= 1 + 1 + 1$

$\therefore \quad C_{23} = 3$

$r_{34} = 4$

$C_{34} = q_3 + q_4 + p_4$

$= 1 + 1 + 1$

$\therefore \quad C_{34} = 3$

To compute C_{ij} and r_{ij} we will compute the values of k first.

The value of k lies between values of $r_{i, j-1}$ to $r_{i+1, j}$. The min $\{C_{i\,k-1} + C_{kj}\}$ decides value of k.

Consider i = 0 and j = 2.

To decide value of k, the range for k is $r_{01} = 1$ to $r_{12} = 2$ when k = 1 and i = 0, j = 2. We will compute C_{ij} using formula

$$C_{ij} = C_{ik-1} + C_{kj}$$

$\therefore \quad C_{02} = C_{00} + C_{02}$

$C_{02} = 6 \quad \rightarrow$ Minimum value of C_{02}.

When k = 2, and i = 0, j = 2.

$C_{02} = C_{01} + C_{22}$

$C_{02} = 11 + 0$

$\therefore \quad C_{02} = 11$

In above computations we get minimum value of C_{02} when k = 1. Hence the value of k becomes 1.

As $r_{ij} = k$

$$r_{02} = 1$$

For $C_{ij} = W_{ij} + \min \{C_{i,k-1} + C_{kj}\}$

$\therefore$ $C_{02} = W_{02} + C_{02}$

$$C_{02} = 14 + 6 = 20$$

Now for r_{13} and C_{13}

k is between $r_{12} = 2$ to $r_{23} = 3$ then when $k = 2$ and $i = 1, j = 3$.

$$C_{13} = C_{11} + C_{23}$$

$$= 0 + 3$$

$\therefore$ $C_{13} = 3 \rightarrow$ Minimum value of C_{13}

When $k = 3$ and $i = 1, j = 3$

$$C_{13} = C_{12} + C_{33}$$

$$= 6 + 0$$

$\therefore$ $C_{13} = 6$

Hence $k = 2$ gives minimum value of C_{13}.

$\therefore$ $r_{13} = 2$

and $C_{13} = W_{13} + C_{13}$ (Minimum value)

$$= 8 + 3$$

$\therefore$ $C_{13} = 11$

Now for r_{24} and C_{24}

k is between $r_{23} = 3$ to $r_{34} = 4$.

$\therefore i = 2, j = 4$ when $k = 3$.

$$C_{24} = C_{22} + C_{34}$$

$$= 0 + 3$$

$\therefore$ $C_{24} = 3$

When k = 4, i = 2, j = 4.

$$C_{24} = C_{23} + C_{44}$$
$$= 3 + 0$$
$$\therefore \quad C_{24} = 3$$

That means value of k can be 3. Let us consider k = 3. Then,

$$r_{24} = 3$$

and
$$C_{34} = W_{34} + C_{34}$$
$$= 5 + 3$$
$$\therefore \quad C_{34} = 8$$

Now for r_{03} and C_{03}

Value of k lies between $r_{02} = 1$ to $r_{13} = 2$

When k = 1 and i = 0, j = 3.

$$C_{03} = C_{00} + C_{13}$$
$$= 0 + 11$$
$$\therefore \quad C_{03} = 11 \rightarrow \text{Minimum value of } C_{03}.$$

When k = 2 and i = 0, j = 3.

$$C_{03} = C_{01} + C_{23}$$
$$= 11 + 3$$
$$\therefore \quad C_{03} = 14$$

Hence value of k = 1.

$$\therefore \quad r_{03} = 1$$

and
$$C_{03} = W_{03} + C_{03}$$
$$= 16 + 11$$
$$\therefore \quad C_{03} = 27$$

Now for r_{14} and C_{14}

Value of k is between $r_{13} = 2$ to $r_{24} = 3$.

When k = 2 and i = 1, j = 4 then

$$C_{14} = C_{11} + C_{24}$$
$$= 0 + 8$$

$\therefore$ $C_{14} = 8 \rightarrow$ Minimum value of C_{14}

When k = 3 and i = 1, j = 4 then

$$C_{14} = C_{12} + C_{34}$$
$$= 6 + 3$$

$\therefore$ $C_{14} = 9$

Hence value of k = 2

$\therefore$ $r_{14} = 2$

and $C_{14} = W_{14} + C_{14}$

$$= 10 + 8$$

$$C_{14} = 18$$

Now for computing r_{04} and C_{04}

Value of k is between $r_{03} = 1$ to $r_{14} = 2$ then,

When k = 1 and i = 0, j = 4.

$$C_{04} = C_{00} + C_{14}$$
$$= 0 + 18$$

$\therefore$ $C_{04} = 18 \rightarrow$ Minimum value of C_{04}

When k = 2 and i = 0, j = 4.

$$C_{04} = C_{01} + C_{24}$$
$$= 11 + 8$$

$\therefore \quad C_{04} = 19$

Hence value of k = 1

$r_{04} = 1$

$\therefore \quad C_{04} = W_{04} + C_{04}$

$= 18 + 18$

$\therefore \quad C_{04} = 36$

To summarize

	0	1	2	3	4
0	$C_{00} = 0$ $r_{00} = 0$	$C_{11} = 0$ $r_{11} = 0$	$C_{22} = 0$ $r_{22} = 0$	$C_{33} = 0$ $r_{33} = 0$	$C_{44} = 0$ $r_{44} = 0$
1	$C_{01} = 11$ $r_{01} = 1$	$C_{12} = 6$ $r_{12} = 2$	$C_{23} = 3$ $r_{23} = 3$	$C_{34} = 3$ $r_{34} = 4$	
2	$C_{02} = 20$ $r_{02} = 1$	$C_{13} = 11$ $r_{13} = 2$	$C_{24} = 8$ $r_{24} = 3$		
3	$C_{03} = 27$ $r_{03} = 1$	$C_{14} = 18$ $r_{14} = 2$			
4	$C_{04} = 36$ $r_{04} = 1$				

To build the OBST

$r_{04} = 1$

Hence a_1 becomes root node

(a_1) $[r_{04}]$

To compute children of a_1 we will apply following formula

For node r_{ij} = k then

left child is r_{ik-1} and right child is r_{kj}

Hence for r_{04} = 1, for node a_1

i = 0, j = 4 and k = 1.

Left child = r_{00} = 0. That is empty node.

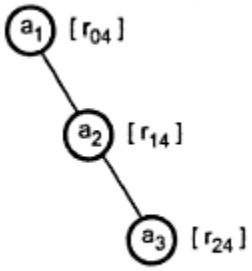
Right child = r_{14}= 2. That means a_2 is right child of a_1.

For node $r_{14} = 2$ for node a_2

$i = 1, j = 4$ and $k = 2$.

Left child = $r_{11} = 0$. That is empty node.

Right child = $r_{24} = 3$. That means a_3 is right child of a_2.



For node $r_{24} = 3$ i.e. for node a_3 $i = 2, j = 4$, and $k = 3$.

Left child = $r_{22} = 0$. That means no left child.

Right child = $r_{34} = 4$. That means a_4 is right child of node a_3.

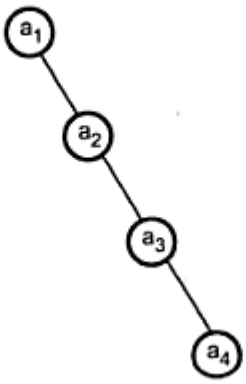


Fig. OBST

Previous papers questions

4.a) Solve the following by using job sequencing with deadlines n=5, (p1…..p5)=(20, 15,10, 5, 1) and (d1,…..d5)=(2, 2, 1, 3,3)

b) Write an algorithm of Kruskal's minimum cost spanning tree. [15]

5.a) Obtain all pair shortest paths for following graph (Figure.1):

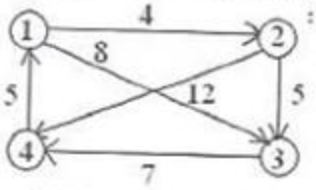


Figure.1

b) Derive the mathematical formulation in reliability design. [15]

---

4. (a) Write an algorithm of prim's minimum spanning tree.
   (b) Find the optimal solution of the Knapsack instance n= 7, M=15, (p1,p2,….p7) = (10,5,15,7,6,18,3) and (w1,w2,…..w7)=(2,3,5,7,1,4,1)

5. (a) Define merging and purging rules in o/1 Knapsack problem.
   (b) Write an algorithm for all pairs shortest path. Explain with an example

5. Consider 4 elements $a_1 < a_2 < a_3 < a_4$ with $q_0=0.25$, $q_1=3/16$, $q_2=q_3=q_4=1/16$. $p_1=1/4$, $p_2=1/8$ $p_3=p_4=1/16$.
   (a) Construct the optimal binary search tree as a minimal cost tree.
   (b) Construct the table of values W_{ij}, C_{ij}, V_{ij} computed by the algorithm to compute the roots of optimal subtrees

4. (a) What do you mean by minimum spanning tree? write and explain algorithm for minimal spanning tree with an example.
   (b) Differentiate between Divide and Conquer algorithm & greedy Algorithm

5. (a) Solve the all-pair shortest path problem for given adjacency matrix graph using Floyd's algorithm

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & \infty \end{bmatrix}$$

   (b) Distinguish the following: (i) Dynamic Programming vs. Divide and Conquer
   (ii) Dynamic Programming vs. Greedy method

UNIT-4

Backtracking is one of the most general technique. In this technique, we search for the set of solutions or optimal solution which satisfies some constraints. One way of solving a problem is by exhaustive search, we enumerate all possible solutions and see which one produces the optimum result. For example, for the Knapsack problems, we could look at every possible subset objects, and find out one which has the greatest profit value and at the same time not greater than the weight bound. Backtracking is a variation of exhaustive search, where the search is refined by eliminating certain possibilities. Backtracking is usually faster method than an exhaustive search.

General Method

- In the backtracking method,
  1. The desired solution is expressible as an n tuple $(x_1, x_2,...x_n)$ where x_i is chosen from some finite set S_i.
  2. The solution maximizes or minimizes or satisfies a criterion function C $(x_1, x_2,...x_n)$.
- The problem can be categorized into three categories.

For instance - For a problem P let C be the set of constraints for P. Let D be the set containing all solutions satisfying C then,

Finding whether there is any feasible solution? - is the decision problem.

What is the best solution ? - is the optimization problem.

Listing of all the feasible solution - is the enumeration problem.

- The basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success.
- The major advantage of this algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.
- Backtracking algorithm determines the solution by systematically searching the solution space (i.e. set of all feasible solutions) for the given problem.
- Backtracking is a depth first search with some bounding function. All solutions using backtracking are required to satisfy a complex set of constraints. The constraints may be explicit or implicit.
- Explicit constraints are rules, which restrict each vector element to be chosen from the given set. Implicit constraints are rules, which determine which of the tuples in the solution space, actually satisfy the criterion function.

Some Terminologies used in Backtracking

Backtracking algorithms determine problem solutions by systematically searching for the solutions using **tree structure.**

For example -

Consider a 4-queen's problem. It could be stated as "there are 4 queens that can be placed on 4 × 4 chessboard. Then no two queens can attack each other".

Following figure shows tree organization for this problem.

See Fig. 5.1 on next page.

- Each node in the tree is called **a problem state.**
- All paths from the root to other nodes define the **state space of** the problem.

- The solution states are those problem states s for which the path from root to s defines a **tuple** in the solution space.

In some trees the leaves define the **solution states.**

- **Answer states :** These are the leaf nodes which correspond to an element in the set of solutions, these are the states which satisfy the implicit constraints.

- A node which is been generated and all whose children have not yet been generated is called **live node.**
- The live node whose children are currently being expanded is called **E-node.**
- A **dead node** is a generated node which is not to be expanded further or all of whose children have been generated.

- There are two methods of generating state search tree -
  i) **Backtracking** - In this method, as soon as new child C of the current E-node N is generated, this child will be the new E-node.
  The N will become the E-node again when the subtree C has been fully explored.
  ii) **Branch and Bound** - E-node will remain as E-node until it is dead.
- Both bracktracking and branch and bound methods use **bounding functions.** The bounding functions are used to kill live nodes without generating all their children. The care has to be taken while killing live nodes so that at least one answer node or all answer nodes are obtained.

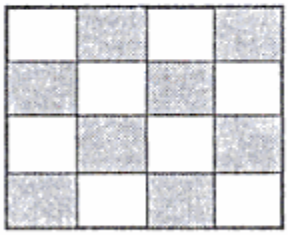
N-Queen's Problem

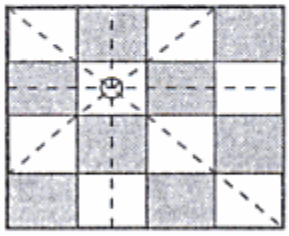
The n-queen's problem can be stated as follows.

Consider a n × n chessboard on which we have to place n queens so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

For example -

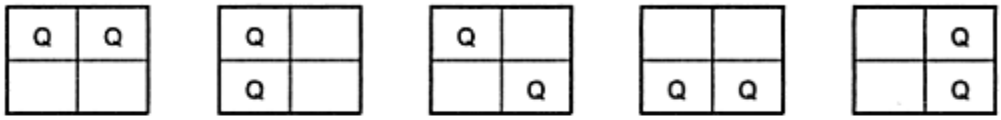
Consider 4 × 4 board.



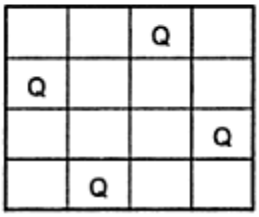


The next queen - if is placed on the paths marked by dotted lines then they can attack each other

- 2-Queen's problem is not solvable - Because 2-queens can be placed on 2 × 2 chessboard as



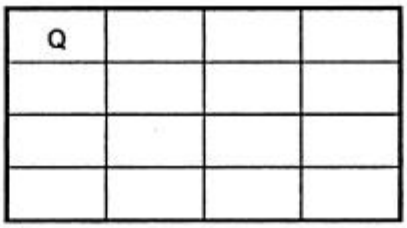
- But 4-queen's problem is solvable.



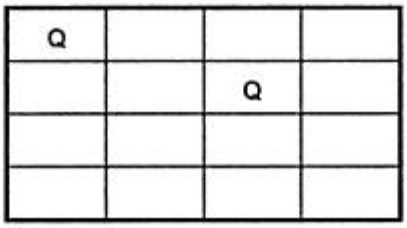
⇐ Note that no two queens can attack each other.

Let us take 4-queens and 4 × 4 chessboard.

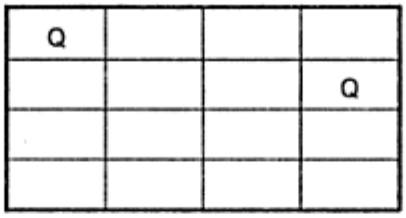
- Now we start with empty chessboard.
- Place queen 1 in the first possible position of its row. i.e. on 1st row and 1st column.



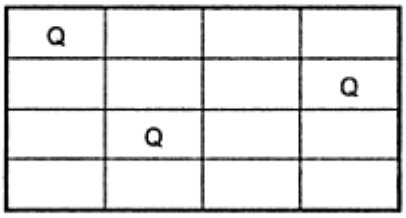
- Then place queen 2 after trying unsuccessful place - 1(1, 2), (2, 1), (2, 2) at (2, 3) i.e. 2nd row and 3rd column.



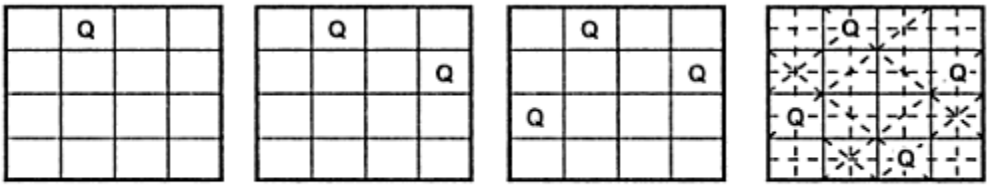
- This is the dead end because a 3rd queen cannot be placed in next column, as there is no acceptable position for queen 3. Hence algorithm **backtracks** and places the 2nd queen at (2, 4) position.



- The place 3rd queen at (3, 2) but it is again another dead end as next queen (4th queen) cannot be placed at permissible position.



- Hence we need to backtrack all the way upto queen 1 and move it to (1, 2).
- Place queen 1 at (1, 2), queen 2 at (2, 4), queen 3 at (3, 1) and queen 4 at (4, 3).



THE 8-QUEEN PROBLEM

Take a chess-board. As we know that it contains 64 cells or squares (having two colours, viz. black and white alternatively) arranged in 8 rows and 8 columns. The problem is to place 8 queens in the chessboard squares so that no two of them attack each other, that is no two of them are on the same row, column, or diagonal. For future reference we put the numbers 1, 2, ..., 8 to the rows (from left to right) and columns (from top to bottom) of the chessboard as shown below. We also label the queens as 1, 2, ..., 8. There will be no loss of generality if we assume that queen i is placed on row i. The solution set to the 8-queen problem is then the set of all 8-tuples $(x_1, \ldots, x_8)$, where $1 \le x_i \le 8$ and x_i is the column on which queen i is placed.

1. The explicit constraints of the problem are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, for $1 \le i \le 8$ and
2. The implicit constraints are that neither two queens are on the same column nor on the same diagonal.

The solution space consists of 8^8 elements, each of 8-tuples. The explicit constraints reduce the number of possible solutions to be the permutations of the 8-tuple (1, 2, 3, 4, 5, 6, 7, 8). This observation reduces the solution space from 8^8 tuples to 8!. One valid solution to the 8-queen problem is: (4, 6, 8, 2, 7, 1, 3, 5) and is shown in Figure

	1	2	3	4	5	6	7	8
1					Q			
2		Q						
3						Q		
4	Q							
5							Q	
6				Q				
7								Q
8			Q					

Figure Solution to the 8-queens problem.

Sum of Subsets Problem

Problem Statement

Let, $S = \{S_1, ... S_n\}$ be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.

It is always convenient to sort the set's elements in ascending order. That is,

$$S_1 \leq S_2 \leq ..... \leq S_n$$

Let us first write a general algorithm for sum of subset problem.

Algorithm

Let, S be a set of elements and d is the expected sum of subsets. Then -

Step 1 : Start with an empty set.

Step 2 : Add to the subset, the next element from the list.

Step 3 : If the subset is having sum d then stop with that subset as solution.

Step 4 : If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.

Step 5 : If the subset is feasible then repeat step 2.

Step 6 : If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

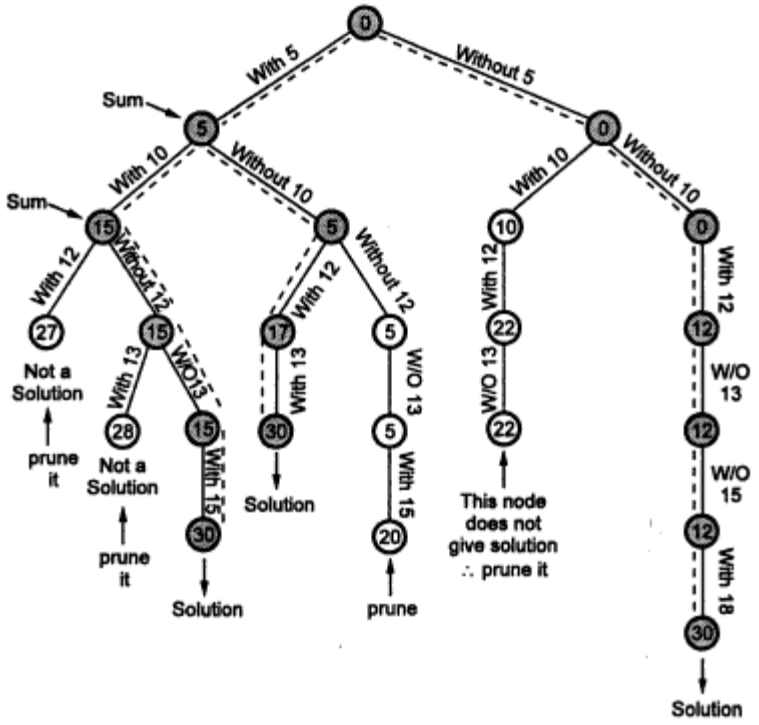
This problem can be well understood with some example.

➠ **Example :** *Consider a set S = {5, 10, 12, 13, 15, 18} and d = 30. Solve it for obtaining sum of subset.*

Solution :

Initially subset = { }	Sum = 0	–
5	5	Then add next element.
5, 10	15 ∵ 15 < 30	Add next element.
5, 10, 12	27 ∵ 27 < 30	Add next element.
5, 10, 12, 13	40	Sum exceeds d = 30 hence backtrack.
5, 10, 12, 15	42	Sum exceeds d = 30 ∴ Backtrack
5, 10, 12, 18	45	Sum exceeds d ∴ Not feasible. Hence backtrack.

5, 10		
5, 10, 13	28	
5, 10, 13, 15	33	Not feasible ∴ Backtrack.
5, 10		
5, 10, 15	30	Solution obtained as sum = 30 = d



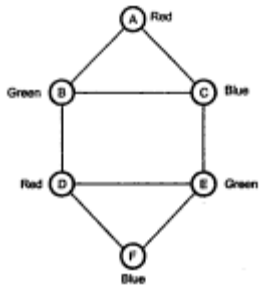
Algorithm

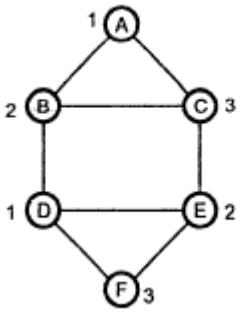
```
Algorithm Sum_Subset(sum, index, Remaining_sum)
//sum is a variable that stores the sum of all the
//selected elements
//index denotes the index of chosen element from the given
//Remaining_sum is initially sum of all the elements.
//selection of some element from the set subtracts the
//chosen value
//from Remaining_sum each time.
//w[1...n] represents the set containing n elements
//a[j] represents the subset where 1 ≤ j ≤ index
//sum = Σ_{j=1}^{index-1} w[j]*a[j]
//Remaining_sum = Σ_{j=index}^{n} = w[j].
// w[j] is sorted in non-decreasing order
// For a feasible sequence assume that w[1] ≤ d and
     Σ_{i=1}^{n} w[i] ≥ d
  // Generate left child until sum + w[index] is ≤ d

     a[index]←1
if(sum + w[index] = d) then
  write(a[1...index]) //subset is found          ← The subset is printed

  else if (sum + w[index] + w[index+1] d) then
   Sum_Subset((sum+w[index]), (index+1),(Remaining_sum – w[index]));
            // Generate right child
  if(sum+Remaining_sum – w[index] ≥ d) AND          ← Search the next element which can make sum ≤ d
          (sum+w[index+1]≤ d)) then
  {
        a[index] ← 0
   Sum_Subset(sum, (index+1), (Remaining_sum – w[index]));
}
```

Graph Coloring

Graph coloring is a problem of coloring each vertex in graph in such a way that no two adjacent vertices have same color and yet m-colors are used. This problem is also called m-coloring problem. If the degree of given graph is d then we can color it with d + 1 colors. The least number of colors needed to color the graph is called its chromatic number. For example : Consider a graph given in Fig.

As given in Fig. we require three colors to color the graph. Hence the chromatic number of given graph is 3. We can use backtracking technique to solve the graph coloring problem as follows -





Algorithm

The algorithm for graph coloring uses an important function Gen_Col_Value() for generating color index. The algorithm is -

```
Algorithm Gr_color(k)

 //The graph G[1 : n,1 : n] is given by adjacency matrix.
//Each vertex is picked up one by one for coloring
//x[k] indicates the legal color assigned to the vertex
{
 repeat
 {
   // produces possible colors assigned
    Gen_Col_Value(k);

 if(x[k] = 0) then
    return; //Assignment of new color is not possible.
 if(k=n) then // all vertices of the graph are colored
   write(x[1:n]); //print color index
 else
```

Takes O(nm) time

```
    Gr_color(k+1) // choose next vertex
   }until(false);
  }
  The algorithm used assigning the color is given as below
  Algorithm Gen_Col_Value(k)
   //x[k] indicates the legal color assigned to the vertex
   // If no color exists then x[k] = 0
   {
  // repeatedly find different colors
     repeat
      {
        x[k] ← (x[k]+1) mod (m+1) ; //next color index when it is
          //highest index
        if(x[k] = 0) then // no new color is remaining
          return;
       for (j ← 1 to n) do
       {
         // Taking care of having different colors for adjacent
      // vertices by using two conditions i) edge should be
      // present between two vertices
        // ii) adjacent vertices should not have same color
         if(G[k,j]!=0) AND (x[k]→x[j])) then
          break;
       }
      //if there is no new color remaining
       if ( j= n+1) then
          return;


    }until(false);
  }
```

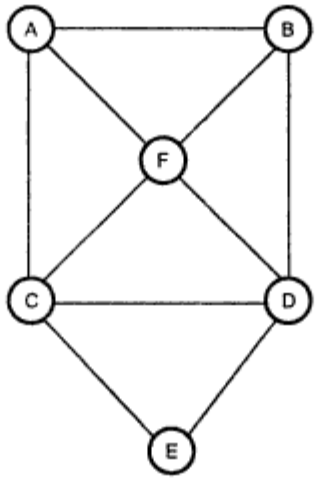
The Gr_color takes computing time of $\sum_{i=0}^{n-1} m^i$. Hence total computing time for this gorithm is $O(nm^n)$.

Hamiltonian Cycle

Problem -

Given an undirected connected graph and two nodes x and y then find a path from x to y visiting each node in the graph exactly once.

For example : Consider the graph G -



Then the Hamiltonian cycle is A - B - D - E - C - F - A. This problem can be solved using backtracking approach. The state space tree is generated in order to find all the Hamiltonian cycles in the graph. Only distinct cycles are output of this algorithm. The Hamiltonian cycle can be identified as follows -

```
1   Algorithm Hamiltonian(k)
2   // This algorithm uses the recursive formulation of
3   // backtracking to find all the Hamiltonian cycles
4   // of a graph. The graph is stored as an adjacency
5   // matrix G[1 : n, 1 : n]. All cycles begin at node 1.
6   {
7       repeat
8       { // Generate values for x[k].
9           NextValue(k); // Assign a legal next value to x[k].
10          if (x[k] = 0) then return;
11          if (k = n) then write (x[1 : n]);
12          else Hamiltonian(k + 1);
13      } until (false);
14  }
```

Algorithm Finding all Hamiltonian cycles

```
1   Algorithm NextValue(k)
2   // x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
3   // no vertex has as yet been assigned to x[k]. After execution,
4   // x[k] is assigned to the next highest numbered vertex which
5   // does not already appear in x[1 : k − 1] and is connected by
6   // an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
7   // in addition x[k] is connected to x[1].
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
12          if (x[k] = 0) then return;
13          if (G[x[k − 1], x[k]] ≠ 0) then
14          { // Is there an edge?
15              for j := 1 to k − 1 do if (x[j] = x[k]) then break;
16                              // Check for distinctness.
17              if (j = k) then // If true, then the vertex is distinct.
18                  if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
19                      then return;
20          }
21      } until (false);
22  }
```

Algorithm Generating a next vertex

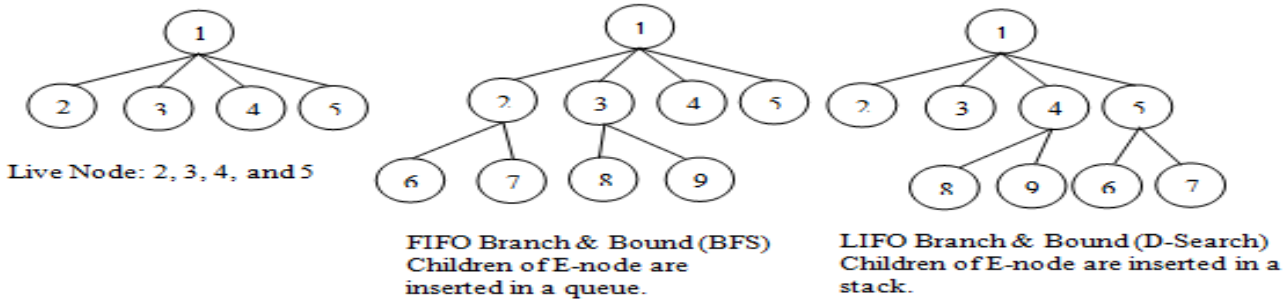
Branch and Bound

Introduction

Branch and bound is a general algorithmic method for finding optimal solutions of various optimization problems. Branch and bounding is a general optimization technique that applies where the greedy method and dynamic programming fail. However, it is much slower. Indeed, it often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average. In this chapter we will discuss how **bounding functions** help in determining the expansion of a node in a **state space tree**. This illustration can be done with the help of 0/1 knapsack problem. Finally we will get introduced with NP hard and NP complete problems.

Branch & Bound (B & B) is general algorithm (or Systematic method) for finding optimal solution of various optimization problems, especially in discrete and combinatorial optimization.

- The B&B strategy is very similar to backtracking in that a state space tree is used to solve a problem.
- The differences are that the B&B method
  - ✓ Does not limit us to any particular way of traversing the tree.
  - ✓ It is used only for optimization problem
  - ✓ It is applicable to a wide variety of discrete combinatorial problem.
- B&B is rather general optimization technique that applies where the greedy method & dynamic programming fail.
- It is much slower, indeed (truly), it often (rapidly) leads to exponential time complexities in the worst case.
- The term B&B refers to all state space search methods in which all children of the "E-node" are generated before any other "live node" can become the "E-node"
  - ✓ **Live node→** is a node that has been generated but whose children have not yet been generated.
  - ✓ **E-node→**is a live node whose children are currently being explored.
  - ✓ **Dead node→** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.



- Two graph search strategies, BFS & D-search (DFS) in which the exploration of a new node cannot begin until the node currently being explored is fully explored.
- Both BFS & D-search (DFS) generalized to B&B strategies.
  - ✓ **BFS→**like state space search will be called FIFO (First In First Out) search as the list of live nodes is "First-in-first-out" list (or queue).
  - ✓ **D-search (DFS)→** Like state space search will be called LIFO (Last In First Out) search as the list of live nodes is a "last-in-first-out" list (or stack).
- In backtracking, bounding function are used to help avoid the generation of sub-trees that do not contain an answer node.
- We will use 3-types of search strategies in branch and bound

1) FIFO (First In First Out) search

2) LIFO (Last In First Out) search
3) LC (Least Count) search

General Method

- In branch and bound method a state space tree is built and all the children of E nodes (a live node whose children are currently being generated) are generated before any other node can become a live node.
- For exploring new nodes either a BFS or D-search technique can be used.

- In Branch and bound technique, BFS-like state space search will be called FIFO (First In First Out) search. This is because the list of live node is First In First Out list(queue). On the other hand the D-search like state space search will be called LIFO search because the list of live node is Last in First out list(stack).
- In this method a space tree of possible solutions is generated. Then partitioning (called as branching) is done at each node of the tree. We compute lower bound and upper bound at each node. This computation leads to selection of answer node.
- Bounding functions are used to avoid the generation of subtrees that do not contain an answer node.

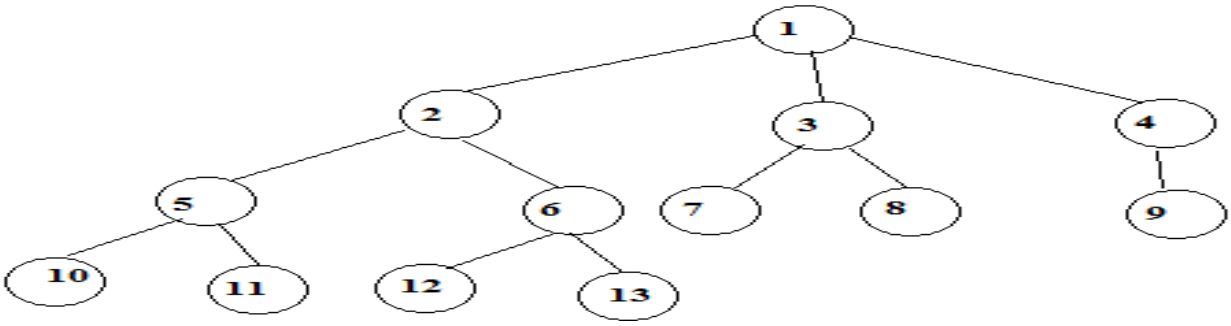
FIFO B&B:

FIFO Branch & Bound is a BFS.

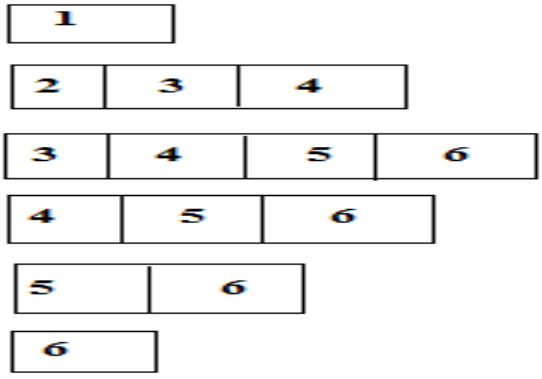
In this, children of E-Node (or Live nodes) are inserted in a queue.

Implementation of list of live nodes as a queue

- ✓ Least()→ Removes the head of the Queue
- ✓ Add()→ Adds the node to the end of the Queue



Assume that node '12' is an answer node in FIFO search, 1st we take E-node has '1'



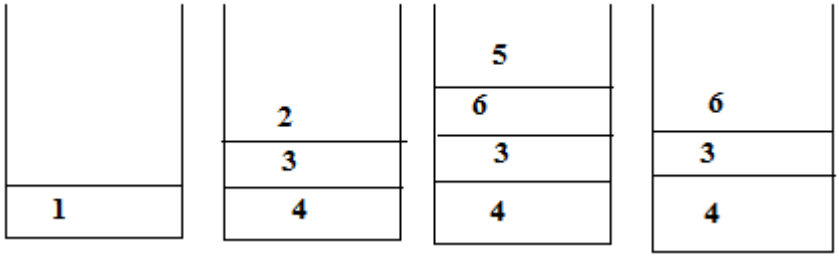
LIFO B&B:

LIFO Brach & Bound is a D-search (or DFS).

In this children of E-node (live nodes) are inserted in a stack

Implementation of List of live nodes as a stack

- ✓ Least()→ Removes the top of the stack
- ✓ ADD()→Adds the node to the top of the stack.



Least Cost (LC) Search:

The selection rule for the next E-node in FIFO or LIFO branch and bound is sometimes "blind". i.e., the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can often be speeded by using an "intelligent" ranking function. It is also called an approximate cost function "**Ĉ**".

Expended node (E-node) is the live node with the best **Ĉ** value.

Branching: A set of solutions, which is represented by a node, can be partitioned into mutually (jointly or commonly) exclusive (special) sets. Each subset in the partition is represented by a child of the original node.

Lower bounding: An algorithm is available for calculating a lower bound on the cost of any solution in a given subset.

Each node X in the search tree is associated with a cost: **Ĉ(X)**

C=cost of reaching the current node, X(E-node) form the root + The cost of reaching an answer node form X.

Ĉ=g(X)+H(X).

Example:

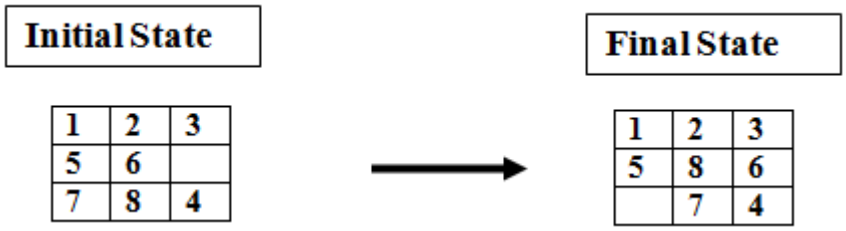
8-puzzle

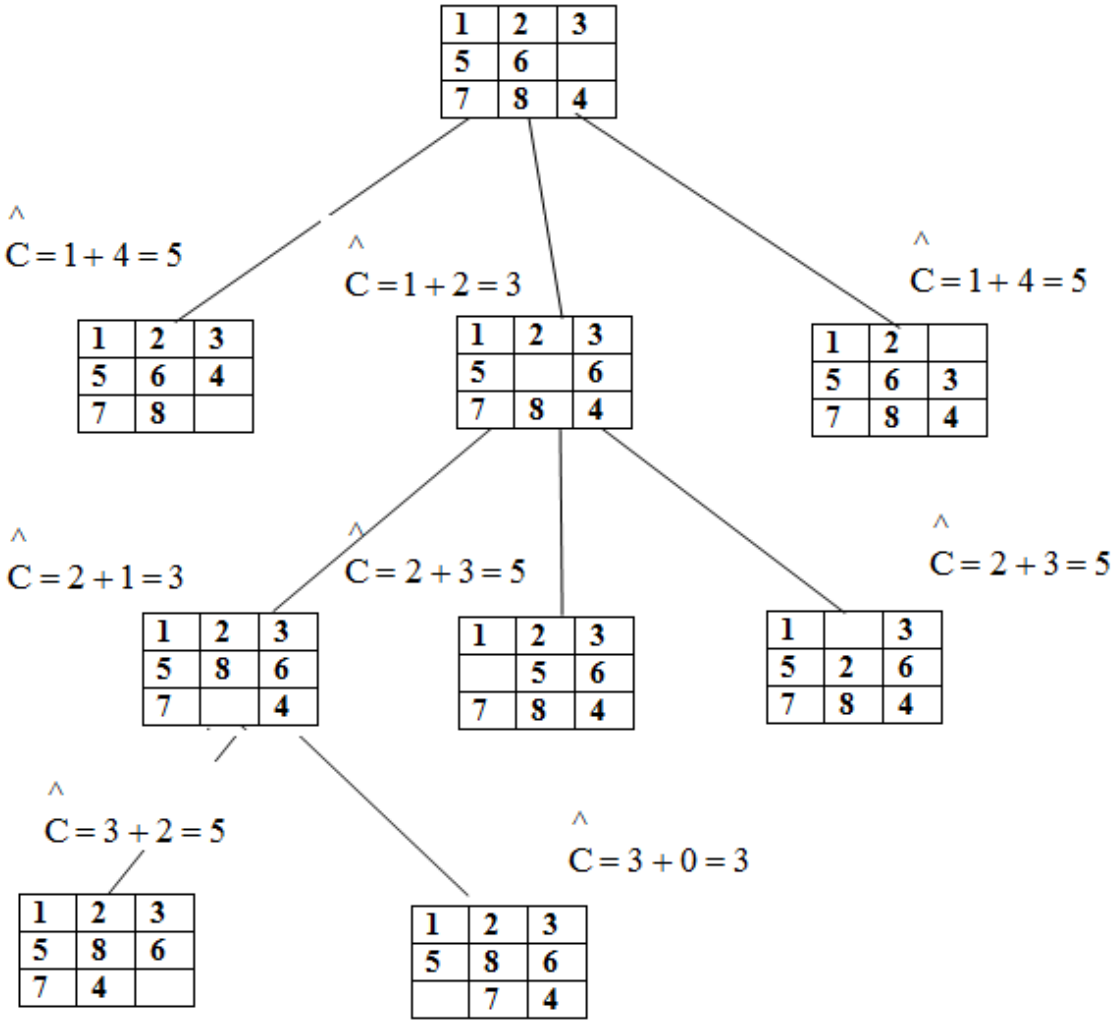
Cost function: **Ĉ** = g(x) +h(x)

where h(x) = the number of misplaced tiles

and g(x) = the number of moves so far

Assumption: move one tile in any direction cost 1.





Note: In case of tie, choose the leftmost node.

Travelling Salesman Problem:

Def:- Find a tour of minimum cost starting from a node S going through other nodes only once and returning to the starting point S.

Time Conmlexity of TSP for Dynamic Programming algorithm is $O(n^2 2^n)$

B&B algorithms for this problem, the worest case complexity will not be any better than $O(n^2 2^n)$ but good bunding functions will enables these B&B algorithms to solve some problem instances in much less time than required by the dynamic programming alogrithm.

Let $G=(V,E)$ be a directed graph defining an instances of TSP.

Let C_{ij} → cost of edge <i, j>

$C_{ij} = \infty$ if <i, j> $\notin$ E

$|V|=n$ → total number of vertices.

Assume that every tour starts & ends at vertex 1.

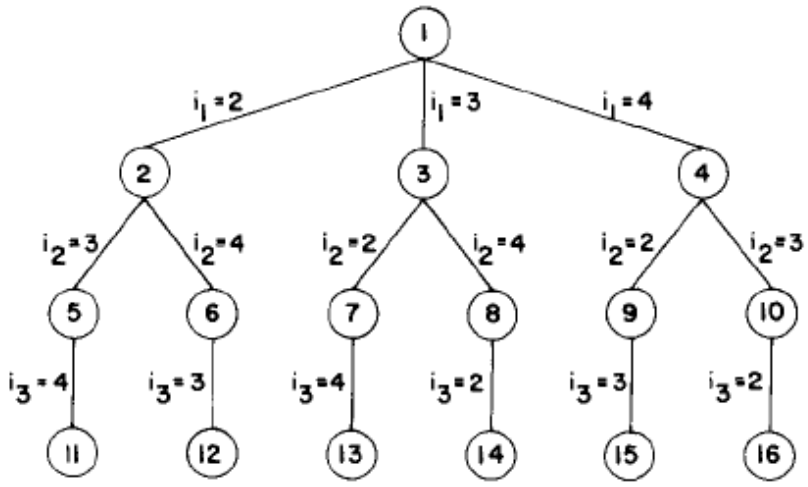
Solution Space S= {1, Π , 1 / Π is a permutation of (2, 3. 4. ----n) } then $|S|=(n-1)!$

The size of S reduced by restricting S

Sothat $(1, i_1, i2, \text{-----} i_{n-1}, 1) \in S$ iff $<i_j, i_{j+1}> \in E$. $O \le j \le n-1$, $i_0 - i_n = 1$

S can be organized into "State space tree".

Consider the following Example



State space tree for the travelling salesperson problem with n=4 and $i_0=i_4=1$

The above diagram shows tree organization of a complete graph with |V|=4.

Each leaf node 'L' is a solution node and represents the tour defined by the path from the root to L.

Node 12 represents the tour.

$i_0=1, i_1=2, i_2=4, i_3=3, i_4=1$

Node 14 represents the tour.

$i_0=1, i_1=3, i_2=4, i_3=2, i_4=1.$

TSP is solved by using LC Branch & Bound:

To use LCBB to search the travelling salesperson "State space tree" first define a cost function C(.) and other 2 functions $\hat{C}(.)$ & $u(.)$

Such that $\hat{C}(r) \le C(r) \le u(r)$ (1) for all nodes r.

Cost C(.)→ is the solution node with least C(.) corresponds to a shortest tour in G.

C(A)={Length of tour defined by the path from root to A if A is leaf

Cost of a minimum-cost leaf in the sub-tree A, if A is not leaf }

Fro(1) $\hat{C}(r) \le C(r)$ then $\hat{C}(r)$ → is the length of the path defined at node A.

From previous example the path defined at node 6 is i_0, i_1, i_2=1, 2, 4 & it consists edge of <1,2> & <2,4>

Abetter $\hat{C}(r)$ can be obtained by using the reduced cost matrix corresponding to G.

- A row (column) is said to be reduced iff it contains at least one zero & remaining entries are non negative.
- A matrix is reduced iff every row & column is reduced.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost Matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced Cost Matrix

L = 25

Given the following cost matrix:

$$\begin{bmatrix} inf & 20 & 30 & 10 & 11 \\ 15 & inf & 16 & 4 & 2 \\ 3 & 5 & inf & 2 & 4 \\ 19 & 6 & 18 & inf & 3 \\ 16 & 4 & 7 & 16 & inf \end{bmatrix}$$

- The TSP starts from node 1: **Node 1**
- Reduced Matrix: To get the lower bound of the path starting at node 1

Row # 1: reduce by 10

$$\begin{bmatrix} inf & 10 & 20 & \\ 15 & inf & 16 & \\ 3 & 5 & inf & 2 \\ 19 & 6 & 18 & in \\ 16 & 4 & 7 & 16 \end{bmatrix}$$

Row #2: reduce 2

$$\begin{bmatrix} inf & 10 & 20 & 0 \\ 13 & inf & 14 & 2 \\ 3 & 5 & inf & 2 \\ 19 & 6 & 18 & inf \\ 16 & 4 & 7 & 16 \end{bmatrix}$$

Row #3: reduce by 2

$$\begin{bmatrix} inf & 10 & 20 & 0 \\ 13 & inf & 14 & 2 \\ 1 & 3 & inf & 0 \\ 19 & 6 & 18 & inf \\ 16 & 4 & 7 & 16 \end{bmatrix}$$

Row # 4: Reduce by 3:

$$\begin{bmatrix} inf & 10 & 20 & 0 \\ 13 & inf & 14 & 2 \\ 1 & 3 & inf & 0 \\ 16 & 3 & 15 & inf \\ 16 & 4 & 7 & 16 \end{bmatrix}$$

Row # 5: Reduce by 4

$$\begin{bmatrix} inf & 10 & 20 & 0 \\ 13 & inf & 14 & 2 \\ 1 & 3 & inf & 0 \\ 16 & 3 & 15 & inf \\ 12 & 0 & 3 & 12 \end{bmatrix}$$

Column 1: Reduce by 1

$$\begin{bmatrix} inf & 10 & 20 & 0 \\ 12 & inf & 14 & 2 \\ 0 & 3 & inf & 0 \\ 15 & 3 & 15 & inf \\ 11 & 0 & 3 & 12 \end{bmatrix}$$

Column 2: It is reduced.

Column 3: Reduce by 3

$$\begin{bmatrix} inf & 10 & 17 & 0 & 1 \\ 12 & inf & 11 & 2 & 0 \\ 0 & 3 & inf & 0 & 2 \\ 15 & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & 12 & inf \end{bmatrix}$$

Column 4: It is reduced.
Column 5: It is reduced.

The reduced cost is: RCL = 25
So the cost of node 1 is: Cost (1) = 25

The reduced matrix is:

Cost (1) = 25

$$\begin{bmatrix} inf & 10 & 17 & 0 & 1 \\ 12 & inf & 11 & 2 & 0 \\ 0 & 3 & inf & 0 & 2 \\ 15 & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & 12 & inf \end{bmatrix}$$

- **Choose to go to vertex 2: Node 2**

- Cost of edge <1,2> is: A(1,2) = 10
- Set row #1 = inf since we are choosing edge <1,2>
- Set column # 2 = inf since we are choosing edge <1,2>
- Set A(2,1) = inf
- The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & 11 & 2 & 0 \\ 0 & inf & inf & 0 & 2 \\ 15 & inf & 12 & inf & 0 \\ 11 & inf & 0 & 12 & inf \end{bmatrix}$$

- The matrix is reduced:
- RCL = 0
- The cost of node 2 (Considering vertex 2 from vertex 1) is:

Cost(2) = cost(1) + A(1,2) = 25 + 10 = 35

- **Choose to go to vertex 3: Node 3**

- Cost of edge <1,3> is: A(1,3) = 17 (In the reduced matrix
- Set row #1 = inf since we are starting from node 1
- Set column # 3 = inf since we are choosing edge <1,3>
- Set A(3,1) = inf
- The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & inf & 2 & 0 \\ inf & 3 & inf & 0 & 2 \\ 15 & 3 & inf & inf & 0 \\ 11 & 0 & inf & 12 & inf \end{bmatrix}$$

Reduce the matrix: Rows are reduced

The columns are reduced except for column # 1:

Reduce column 1 by 11:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 1 & inf & inf & 2 & 0 \\ inf & 3 & inf & 0 & 2 \\ 4 & 3 & inf & inf & 0 \\ 0 & 0 & inf & 12 & inf \end{bmatrix}$$

The lower bound is: RCL = 11

The cost of going through node 3 is:

cost(3) = cost(1) + RCL + A(1,3) = 25 + 11 + 17 = 53

- Choose to go to vertex 4: **Node 4**

Remember that the cost matrix is the one that was reduced at the starting vertex 1

Cost of edge <1,4> is: A(1,4) = 0

Set row #1 = inf since we are starting from node 1

Set column # 4 = inf since we are choosing edge <1,4>

Set A(4,1) = inf

The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & inf & 0 \\ 0 & 3 & inf & inf & 2 \\ inf & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows are reduced

Columns are reduced

The lower bound is: RCL = 0

The cost of going through node 4 is:

cost(4) = cost(1) + RCL + A(1,4) = 25 + 0 + 0 = 25

➢ **Choose to go to vertex 5: Node 5**

- Remember that the cost matrix is the one that was reduced at starting vertex 1
- Cost of edge <1,5> is: A(1,5) = 1
- Set row #1 = inf since we are starting from node 1
- Set column # 5 = inf since we are choosing edge <1,5>
- Set A(5,1) = inf
- The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & 2 & inf \\ 0 & 3 & inf & 0 & inf \\ 15 & 3 & 12 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

Reduce the matrix:

Reduce rows:

Reduce row #2: Reduce by 2

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 10 & inf & 9 & 0 & inf \\ 0 & 3 & inf & 0 & inf \\ 15 & 3 & 12 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

Reduce row #4: Reduce by 3

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 10 & inf & 9 & 0 & inf \\ 0 & 3 & inf & 0 & inf \\ 12 & 0 & 9 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

Columns are reduced

The lower bound is: RCL = 2 + 3 = 5

The cost of going through node 5 is:

cost(5) = cost(1) + RCL + A(1,5) = 25 + 5 + 1 = 31

In summary:

So the live nodes we have so far are:

- ✓ 2: cost(2) = 35, path: 1->2
- ✓ 3: cost(3) = 53, path: 1->3
- ✓ 4: cost(4) = 25, path: 1->4
- ✓ 5: cost(5) = 31, path: 1->5

Explore the node with the lowest cost: Node 4 has a cost of 25

Vertices to be explored from node 4: 2, 3, and 5

Now we are starting from the cost matrix at node 4 is:

Cost (4) = 25

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & inf & 0 \\ 0 & 3 & inf & inf & 2 \\ inf & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & inf & inf \end{bmatrix}$$

- ➢ **Choose to go to vertex 2: Node 6 (path is 1->4->2)**

Cost of edge <4,2> is: A(4,2) = 3

Set row #4 = inf since we are considering edge <4,2>

Set column # 2 = inf since we are considering edge <4,2>

Set A(2,1) = inf

The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & 11 & inf & 0 \\ 0 & inf & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & inf & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows are reduced

Columns are reduced

The lower bound is: RCL = 0

The cost of going through node 2 is:

cost(6) = cost(4) + RCL + A(4,2) = 25 + 0 + 3 = 28

➢ **Choose to go to vertex 3: Node 7 (path is 1->4->3)**

Cost of edge <4,3> is: A(4,3) = 12
Set row #4 = inf since we are considering edge <4,3>
Set column # 3 = inf since we are considering edge <4,3>
Set A(3,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & inf & inf & 0 \\ inf & 3 & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & 0 & inf & inf & inf \end{bmatrix}$$

Reduce the matrix:

Reduce row #3: by 2:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & inf & inf & 0 \\ inf & 1 & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 11 & 0 & inf & inf & inf \end{bmatrix}$$

Reduce column # 1: by 11

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 1 & inf & inf & inf & 0 \\ inf & 1 & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 0 & 0 & inf & inf & inf \end{bmatrix}$$

The lower bound is: RCL = 13
So the RCL of node 7 (Considering vertex 3 from vertex 4) is:
Cost(7) = cost(4) + RCL + A(4,3) = 25 + 13 + 12 = 50

➢ Choose to go to vertex 5: **Node 8** (path is 1->4->5)

Cost of edge <4,5> is: A(4,5) = 0
Set row #4 = inf since we are considering edge <4,5>
Set column # 5 = inf since we are considering edge <4,5>
Set A(5,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & inf & inf \\ 0 & 3 & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & 0 & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix:
Reduced row 2: by 11

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 1 & inf & 0 & inf & inf \\ 0 & 3 & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & 0 & 0 & inf & inf \end{bmatrix}$$

Columns are reduced
The lower bound is: RCL = 11
So the cost of node 8 (Considering vertex 5 from vertex 4) is:
Cost(8) = cost(4) + RCL + A(4,5) = 25 + 11 + 0 = 36

In summary: So the live nodes we have so far are:

- ✓ 2: cost(2) = 35, path: 1->2
- ✓ 3: cost(3) = 53, path: 1->3
- ✓ 5: cost(5) = 31, path: 1->5
- ✓ 6: cost(6) = 28, path: 1->4->2
- ✓ 7: cost(7) = 50, path: 1->4->3
- ✓ 8: cost(8) = 36, path: 1->4->5

➢ Explore the node with the lowest cost: Node 6 has a cost of 28
➢ Vertices to be explored from node 6: 3 and 5

- Now we are starting from the cost matrix at node 6 is:

Cost (6) = 28

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & 11 & inf & 0 \\ 0 & inf & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & inf & 0 & inf & inf \end{bmatrix}$$

- **Choose to go to vertex 3: Node 9 (path is 1->4->2->3)**

Cost of edge <2,3> is: A(2,3) = 11
Set row #2 = inf since we are considering edge <2,3>
Set column # 3 = inf since we are considering edge <2,3>
Set A(3,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & inf & inf & inf & inf \end{bmatrix}$$

Reduce the matrix: Reduce row #3: by 2

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 11 & inf & inf & inf & inf \end{bmatrix}$$

Reduce column # 1: by 11

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 0 & inf & inf & inf & inf \end{bmatrix}$$

The lower bound is: RCL = 2 +11 = 13
So the cost of node 9 (Considering vertex 3 from vertex 2) is:
Cost(9) = cost(6) + RCL + A(2,3) = 28 + 13 + 11 = 52

➢ **Choose to go to vertex 5: Node 10 (path is 1->4->2->5)**

Cost of edge <2,5> is: A(2,5) = 0
Set row #2 = inf since we are considering edge <2,3>
Set column # 3 = inf since we are considering edge <2,3>
Set A(5,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ 0 & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows reduced
Columns reduced
The lower bound is: RCL = 0
So the cost of node 10 (Considering vertex 5 from vertex 2) is:
Cost(10) = cost(6) + RCL + A(2,3) = 28 + 0 + 0 = 28

In summary: **So the live nodes we have so far are:**

- ✓ 2: cost(2) = 35, path: 1->2
- ✓ 3: cost(3) = 53, path: 1->3
- ✓ 5: cost(5) = 31, path: 1->5
- ✓ 7: cost(7) = 50, path: 1->4->3
- ✓ 8: cost(8) = 36, path: 1->4->5
- ✓ 9: cost(9) = 52, path: 1->4->2->3
- ✓ 10: cost(2) = 28, path: 1->4->2->5
- ➢ Explore the node with the lowest cost: Node 10 has a cost of 28
- ➢ Vertices to be explored from node 10: 3
- ➢ Now we are starting from the cost matrix at node 10 is:

Cost (10)=28

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ 0 & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & 0 & inf & inf \end{bmatrix}$$

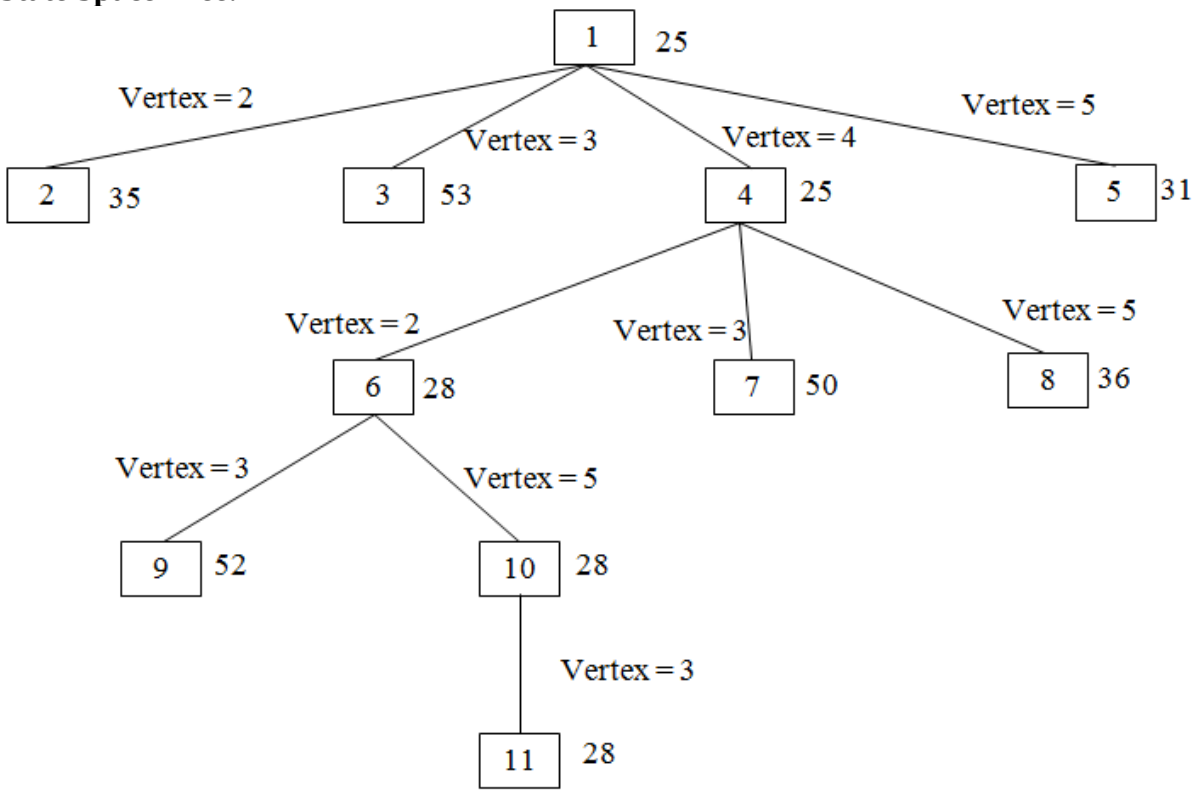
➢ **Choose to go to vertex 3: Node 11 (path is 1->4->2->5->3)**

Cost of edge <5,3> is: A(5,3) = 0
Set row #5 = inf since we are considering edge <5,3>
Set column # 3 = inf since we are considering edge <5,3>
Set A(3,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows reduced
Columns reduced
The lower bound is: RCL = 0
So the cost of node 11 (Considering vertex 5 from vertex 3) is:
Cost(11) = cost(10) + RCL + A(5,3) = 28 + 0 + 0 = 28

State Space Tree:



0/1 knapsack problem

The 0/1 Knapsack problem states that - There are 'n' objects given and capacity of Knapsack is 'm'. Then select some objects to fill The Knapsack in such a way that it should not exceed the capacity of Knapsack and maximum profit can be earned. The Knapsack problem is a maximization problem. That means we will always seek for maximum $P_i x_i$(where P_i represents profit of object x_i). We can also get $\sum P_i x_i$ maximum iff $-\sum P_i x_i$ is minimum.

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

$$\text{Subject to} \quad \sum_{i=1}^{n} W_i x_i$$

$$\text{Such that } \sum W_i x_i \le m \text{ and}$$

$$x_i = 0 \text{ or } 1 \text{ where } 1 \le i \le n$$

We will discuss the branch and bound strategy for 0/1 Knapsack problem using fixed tuple size formulation. We will design the state space tree and compute c^(.) and u(.) where c^(x) represents the approximate cost used for computing the least cost c(x). Clearly u(x) denotes the upper bound. As we know upper bound is used to kill those nodes in the state space tree which cannot lead to the answer node.

Let, x be the node at level j. Then we will draw the state space tree for fixed tuple formulation having levels $1 \le j \le n+1$.

LC branch and bound solution

Cosider knapsack istace =4 with capacity m=15 such that pi={10,10,12,18},wi={2,4,6,9}

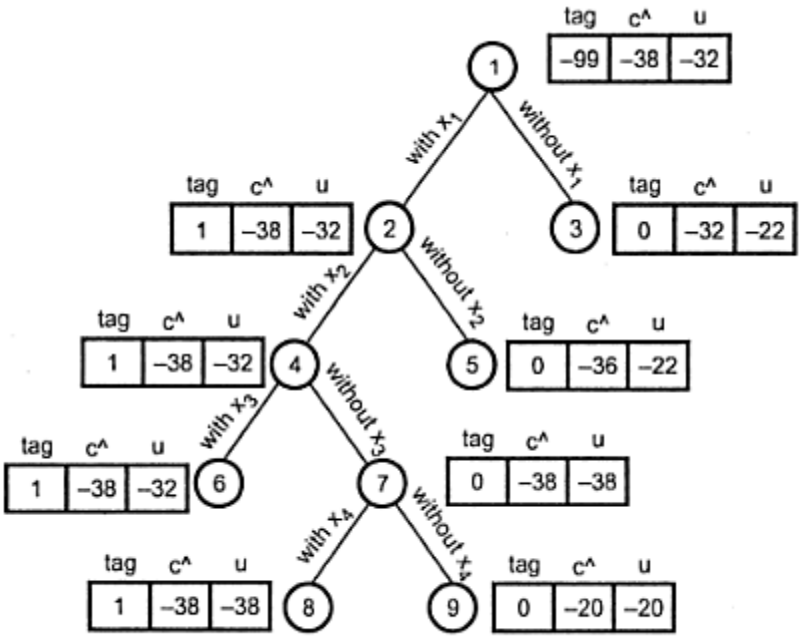


Fig. LCBB solution space tree

In Fig. at each node a structure is drawn in which computation of c^(·) and u(·) is given. The tag field is useful for tracing the path.

Using algorithm U_Bound u(x) is computed and using algorithm C_Bound c^(x) is computed.

u(1) can be computed as -

for i = 1, 2 and 3

$$\therefore \quad -\sum P_i = -(10 + 10 + 12)$$

$$\therefore \quad u(1) = -32$$

If we select i = 4, then it will exceed capacity of Knapsack.

The computation of c^(1) can be done as

$$c^{\wedge}(1) = u(1) - \left[\frac{m - \text{Current total weight}}{\text{Actual weight of remaining object}}\right] * \left[\begin{matrix}\text{Actual profit}\\ \text{of remaining object}\end{matrix}\right]$$

$$c^{\wedge}(1) = \left[-32 - \left[\frac{15-(2+4+6)}{9}\right] * 18\right]$$

$$= -32 - \frac{3}{9} * 18$$

$$c^{\wedge}(1) = -38$$

In this way considering each possibility of object being in Knapsack or not being in Knapsack. c^(x) and u(x) is computed. Each time minimum $-\sum P_I x_i$ will become E-node an we will get the answer node as node 8. If we trace the tag field we will get tag(2) – tag (4) – tag (7) – tag (8). i.e. 1101. Hence $x_4 = 1$, $x_3 = 0$, $x_2 = 1$ and $x_1 = 1$. We will select object x_4, x_2 and x_1 to fill up the Knapsack and gain maximum profit.

FIFO Branch-and-Bound Solution

The space tree with variable tuple size formulation can be drawn and c^(.) and u(.) is computed (We have considered the same Knapsack problem which is discussed

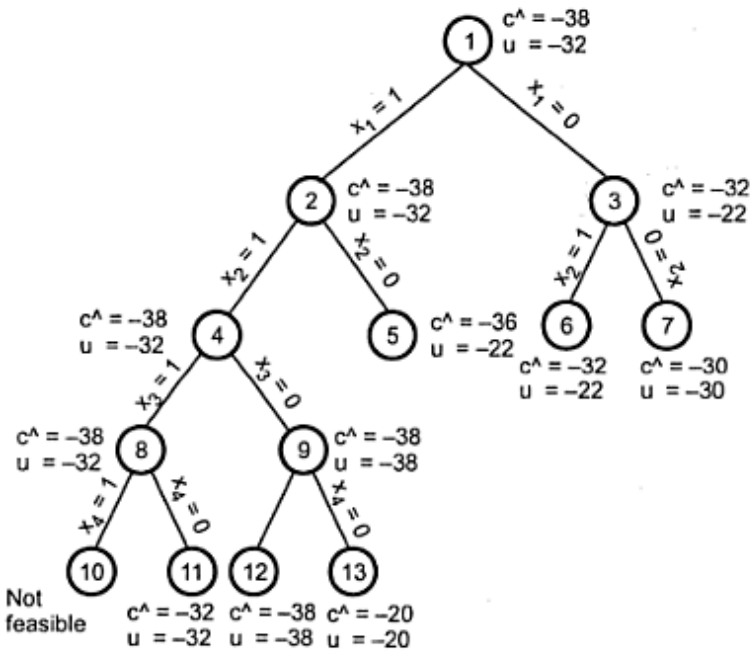


Fig. FIFOBB space tree

Initially upper = u(1) = – 32. Then children of node 1 are generated. Node 2 becomes E-node and hence children 4 and 5 are generated. Node 4 and 5 are added in the list of live nodes. Next, node 3 becomes E-node and children 6 and 7 are generated. As c^(7) > upper we will kill node 7. Hence node 6 will be added in the list of live nodes. Node 4 is E-node and children 8 and 9 are generated. The upper is updated and it is now upper = u(9) = – 38. Nodes 8 and 9 are added in the list of live nodes. Node 5 and 6 becomes the next E-node but as c^(5) > upper and c^(6) >upper, kill nodes 5 and 6. Node 8 becomes next E-node and children 10 and 11 are generated. As node 10 is infeasible do not consider it. c^(11) > upper. Hence kill node 11. Node 9 becomes next E-node and upper = –38. Children 12 and 13 are generated. But c^(13) > upper. So kill node 13. Finally node 12 becomes an answer node. Therefore solution is $x_1 = 1$, $x_2 = 1$, $x_3 = 0$ and $x_4 = 1$.

Important Questions for exam point of view:

1. (a) Write FIFOBB algorithm for the 0/1 knapsack problem
(b) Explain the general method of Branch and Bound.

2. Apply the Branch and Bound algorithm to solve the TSP, for the following cost matrix.

$$\begin{bmatrix} \infty & 11 & 10 & 9 & 6 \\ 8 & \infty & 7 & 3 & 4 \\ 8 & 4 & \infty & 4 & 8 \\ 11 & 10 & 5 & \infty & 5 \\ 6 & 9 & 5 & 5 & \infty \end{bmatrix}$$

3. (a) Explain how the traveling salesperson problem is solved by using LC Branch and Bound.
(b) Write the general algorithm for Branch and Bound.

4. What is traveling sales person problem? Solve the following sales person problem instance using branch and bound.

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

5. (a) Draw the portion of the state space tree generated by FIFOBB using the variable tuple size for the knapsack instances, n = 5 (P1, P2, P5) = (10, 15, 6, 8, 4),
(w1, w2,..., w5) = (4, 6, 3, 4, 2) and M = 12.
(b) Write the control abstraction of LC search.

6. a) What do you mean by bounding? Explain how these bound are useful in branch and bound methods.
b) Explain the principles of
i) FIFO branch and Bound
ii) LC Branch and Bound

7. a) Explain the method of reduction to solve TSP problem using Branch and Bound.
b) Explain the principles of FIFO Branch and Bound.

8. (a) Explain the principles of the following:
i) FIFO branch and Bound ii) LC Branch and Bound
(b)Draw the portion of the state space tree generated by LCBB for the knapsack instances:
n=5, (P1, P2,… , P5) = (12,10,5,9,3), (w1, w2, ...,w5)=(3, 5, 2, 5, 3)
and M =12.

2 marks questions

1. Define backtracking.
2. What is meant by optimization problem?
3. Define Hamiltonian circuit problem.
4. What is Hamiltonian cycle in an undirected graph?
5. Define 8queens problem.
6. List out the application of backtracking.
7. Give the explicit and implicit constraint for 8-queen problem.
8. How can we represent the solution for 8-queen problem?
9. Give the categories of the problem in backtracking.
10. Differentiate backtracking and over exhaustive search.
11. What is state space tree?

NP-Hard and NP-Complete Problems

Most algorithms we have studied so far have polynomial-time running times – that is, finding the shortest paths and minimum spanning trees in graphs, sorting of arrays, matrix multiplication, and so on. All these algorithms are efficient, because in each case their time requirement grows as a polynomial function (such as n, n^2, or n^3) of the size of the input.

This chapter gives a general introduction of P, NP and NP-complete problems. We will study the definition and some classic examples of NP-complete problems. More importantly, we will learn how to show that a problem is not easier than another problem by polynomial-time reduction. Polynomial-time reduction is the general method to prove that a new problem is NP-complete, so that we have an excuse not to find the optimal solution. As we encounter more NP-complete problems, we will learn some general techniques for reductions.

Polynomial-time algorithms can be considered tractable (easy or not so difficult) for the following reasons:

Although a problem which has a running time of say $O(n^k)$ (for large value of k) can be called intractable (difficult), there are very few practical problems with such orders of polynomial complexity.

For reasonable models of computation, a problem that can be solved in polynomial time in one model can also be solved in polynomial time on another.

The class of polynomial-time solvable problems has nice closure properties (since polynomials are closed under addition, multiplication, etc.)

P, NP, and NP-complete

Somehow we believe that an efficient algorithm should run in polynomial time, that is, $O(n^k)$ for input size n and some constant k. Consequently, problems that can be solved in polynomial time are treated as "easy" problems (although k can be arbitrarily large), and we use P to denote the set of polynomial-time-solvable problems. Here is the formal definition:

P: the set of problems that can be solved in polynomial time.

Some examples in P:

(1) Finding the shortest-path in a graph between two vertices such that the sum of the weights of its constituent edges is minimized.
(2) Testing whether given number is prime.
(3) Finding the maximum element in a given array.

Some problems that we can not solve in polynomial time so far:

(1) Factorization: given n and m, determine whether n has any prime factors less than m. For example, let $n = 324$ and $m = 5$. 324 is the product of the prime factors 2 x 2 x 3 x 3 x 3 x 3 = $2^2 \times 3^4$. 2 and 3 are prime factors less than 5.
(2) Polynomial Identities Testing: determining whether a given multivariate polynomial is identically equal to 0. For example, consider a polynomial in two variables x and y: $x^3y^2 + xy - 10$. For $x = 2$ and $y = 1$, the value of this polynomial is 0. Hence,
(3) Graph Coloring: given k, determine whether a graph has a k vertex coloring.

The problems that can not be solved in polynomial time so far are considered as "hard" problems. For the "hard" problems, if (say by guessing) we are given a solution for a problem, we may be able to check whether the solution is correct very efficiently (i.e. in polynomial time). This leads to the definition of NP problems:

NP: the set of problems for which the "yes" answer (with a certificate – that is, a solution) can be verified in polynomial time. NP stands for "Non-deterministic Polynomial time".

One quick observation is that any problem in P is also in NP, because we can verify its answer by solving the problem (and so do not even need a certificate). On the other hand, many "hard" problems mentioned above are also in NP: although we do not have a polynomial-time algorithm for it, we can verify a solution in polynomial time. To understand the relation between P and NP, it is useful to have a "hardest" problem in NP. Here comes the definition for NP-completeness:

A problem is NP-complete if (1) it is in NP; and (2) if it can be solved in polynomial time, then any problem in NP can be solved in polynomial time.

Generally NP-complete (NPC) problems are considered as the "hard" problems. The interesting thing is that we never say NPC problems can not be solved in polynomial time in the formal definition, because we do not have a proof for it. Instead, we use the second property in the definition to link all the NPC problems together: any NPC problem is at least not easier than any other NPC problem. The technique to show that a problem A is not easier than another problem B is Polynomial-time Reduction, which will be shown in next section.

Based on the above description (and our belief), the relation of P, NP and NPC problems are shown in Figure Note that we do not know for sure whether NP or NPC problems are polynomial-time-solvable.

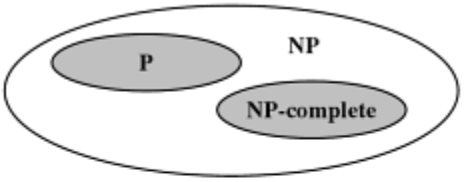


Figure : What we think the world looks like…

It is important to know the rudiments of NP-completeness for anyone to design "sound" algorithms for problems. If one can establish a problem as NP-complete, there is strong reason to believe that it is intractable. We would then do better by trying to design a good approximation algorithm rather than searching endlessly seeking an exact solution. An example of this is the Traveling Salesman Problem (TSP), which is intractable. A practical strategy to solve TSP therefore would be to design a good approximation algorithm. Another important reason to have good familiarity with NP-completeness is many natural interesting and safe-looking problems that on the surface seem no harder than sorting or searching, are in fact NP-complete.

Definition 1 *Any problem for which the answer is either zero or one is called a* ***decision problem****. An algorithm for a decision problem is termed a* ***decision algorithm****.*

Definition 2 *Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an* ***optimization problem****. An* ***optimization algorithm*** *is used to solve an optimization problem.*

Examples of some Intractable Problems

Before giving examples on intractable problems, let us define a decision problem. A *decision problem* determines whether or not there exists a decision procedure or algorithm for a class S of questions requiring a Boolean value (i.e., a true or false, or yes or no). These are also known as yes-or-no questions. For example, the decision problem for the class of questions "Does x divide y without remainder?" is decidable because there exists a mechanical procedure, which allows us to determine for any x and y whether the answer for "Does x divide y without remainder?" is yes or no.

Traveling Salesman Problem:
A Hamiltonian cycle in an undirected graph is a simple cycle that passes through every vertex exactly once. Optimization Problem: Given a complete, weighted graph, find a minimum-weight Hamiltonian Cycle. Decision Problem: Given a complete, weighted graph and an integer k, does there exist a Hamiltonian cycle with total weight at most k.

Subset Sum:
The input is a positive integer C and n objects whose values are positive integers $x_1, x_2, \ldots, x_n$. Optimization Problem: Among all subsets of objects with sum at most C, what is the largest subset sum? Decision Problem: Is there a subset of objects whose values add up to exactly C?

Knapsack Problem:
This is a generalization of the subset sum problem. Consider a knapsack of capacity C where C is a positive integer and n objects with positive integer weights $w_1, w_2, \ldots, w_n$ and positive integer profits $p_1, p_2, \ldots, p_n$. Optimization Problem: Find the largest total profit of any subset of the

objects that fits in the knapsack. Decision Problem: Given k, is there a subset of the objects that fits in the knapsack and has total profit at least k?

Satisfiability (SAT):

A propositional variable (Boolean variable) is one that can be assigned the value "true" or "false". A literal is a propositional variable or its negation (such as $\overline{p}$). A clause is a sequence of literals separated by the logical OR operator ( ). A propositional formula is said to be in *conjunctive normal form* (CNF) if it consists of a sequence of clauses separated by the logical AND operator ( ). For example,

$$(p \vee q \vee s) \wedge (\overline{q} \vee r) \wedge (\overline{p} \vee \overline{q} \vee r \vee \overline{s})$$

This is a Boolean formula in CNF. A truth assignment for a set of propositional variables is an assignment of true or false value to each propositional variable. A truth assignment is said to satisfy a formula if it makes the value of the formula true. In other words, given a Boolean formula in CNF, either find a satisfying truth assignment or else report that none exists.

3SAT (Decision Problem): Given a CNF formula in which each clause is permitted to contain at most three literals, is there a truth assignment to its variables that satisfies it?

CLASSES P AND NP

An algorithm is said to be polynomially bounded if its worst-case complexity is bounded by a polynomial function of the input size. A problem is said to be polynomially bounded if there is a polynomially bounded algorithm for it.

P is the class of all decision problems that are polynomially bounded. The implication is that a decision problem X P can be solved in polynomial time on a deterministic computation model (such as a deterministic Turing machine).

NP represents the class of decision problems which can be solved in polynomial time by a non-deterministic model of computation. That is, a decision problem X NP can be solved in polynomial-time on a non-deterministic computation model (such as a non-deterministic Turing machine). A non-deterministic model can make the right guesses on every move and race towards the solution much faster than a deterministic model.

A deterministic machine, at each point in time, executes an instruction. Depending on the outcome of executing the instruction, it then executes some next instruction, which is unique. A non-deterministic machine on the other hand has a choice of next steps. It is free to choose any that it wishes. For example, it can always choose a next step that leads to the best solution for the problem. A non-deterministic machine thus has the power of extremely good, optimal guessing.

As an example, let us consider the decision version of TSP: Given a complete, weighted graph and an integer k, does there exist a Hamiltonian cycle with total weight at most k?

A smart non-deterministic algorithm for the above TSP problem starts with a vertex, guesses the correct edge to choose, proceeds to the next vertex, guesses the correct edge to

choose there, etc. and in polynomial time discovers a Hamiltonian cycle of least cost and provides an answer to the above problem. This is the power of non-determinism. A deterministic algorithm here will have no choice but take super-polynomial time to answer the above question.

Another way of viewing the above is that given a candidate Hamiltonian cycle (call it certificate), one can verify in polynomial time whether the answer to the above question is YES or NO. Thus to check if a problem is in NP, it is enough to prove in polynomial time that any YES instance is correct. We do not have to worry about NO instances since the program always makes the right choice.

It is easy to show that P NP. However, it is unknown whether P = NP. In fact, this question is perhaps the most celebrated of all open problems in computer science.

Nondeterministic algorithms

Non-Deterministic Algorithm

- The algorithm in which every operation is uniquely defined is called deterministic algorithm.
- The algorithm in which every operation may not have unique result, rather there can be specified set of possibilities for every operation. Such an algorithm is called non-deterministic algorithm. Non-deterministic means that no particular rule is followed to make the guess.
- The non-deterministic algorithm is a two stage algorithm -

i) Non-deterministic (Guessing) stage - generate an arbitrary string that can be thought of as a candidate solution.

ii) Deterministic ("Verification") stage - In this stage it takes as input the candidate solution and the instance to the problem and returns *yes* if the candidate solution represents actual solution.

```
Algorithm Non_Determin()
// A[1:n] is a set of elements
// we have to determine the index i of A at which element x is
//located.
{
  // The following for-loop is the guessing stage
     for i=1 to n do
       A[i] := choose(i);
```

```
// Next is the verification(deterministic) stage
 if (A[i] = x) then
 {
    write(i);
    success();
 }
 write(0);
 fail();
}
```

In the above given non deterministic algorithm there are three functions used –

1) Choose - arbitrarily chooses one of the element from given input set.
2) Fail - indicates the unsuccessful completion.
3) Success - indicates successful completion.

The algorithm is of non deterministic complexity O(1), when A is not ordered then the deterministic search algorithm has a complexity Ω (n).

➠ **Example** : *The non-deterministic algorithm for sorting elements in non decreasing order is as given below –*

```
Algorithm Non_DSort(A,n)
// A[1:n] is an array that stores n elements which are positive //integers B[1:n] be
an auxiliary array in which elements are put //at appropriate positions
{
// guessing stage
   for i:=1 to n do
      B[i]:=0;//initialize B to 0
   for i:=1 to n do
   {
      j:=choose(1,n)//select any element from the input set
      if(B[j]!=0) then
         fail();
      B[j]:=A[i];
   }
//verification stage
   for i:=1 to n–1 do
      if(B[i]>B[i+1] then
         fail();// not sorted elements
      write(B[1:n]);// print the sorted list of elements
      success();
}
```

- Non-deterministic polynomial time algorithm for knapsack problem

```
algorithm nd_knapsack ( p, w, n, m, r, x )
{
    W = 0;
    P = 0;
    for ( i = 1; i <= n; i++ )
    {
        x[i] = choice ( 0, 1 );
        W += x[i] * w[i];
        P += x[i] * p[i];
    }


    if ( ( W > m ) || ( P < r ) )
        failure();
    else
        success();
}
```

- The `for` loop selects or discards each of the `n` items
- It also recomputes the total weight and profit coresponding to the selection
- The `if` statement checks to see the feasibility of assignment and whether the profit is above a lower bound `r`
- The time complexity of the algorithm is $O(n)$
- If the input length is q in binary, time complexity is $O(q)$

Definition *$\mathcal{P}$ is the set of all decision problems solvable by deterministic algorithms in polynomial time. $\mathcal{NP}$ is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.*

Definition *Let A and B be problems. Problem A **reduces** to B (written as $A \propto B$) if and only if there is a way to solve A by a deterministic polynomial time algorithm using a deterministic algorithm that solves B in polynomial time.*

* If we have a polynomial time algorithm for B, then we can solve A in polynomial time
* Reducibility is transitive
  - $A \propto B \wedge B \propto C \Rightarrow A \propto C$

- The group of problems is further subdivided into two classes

$\mathcal{NP}$-complete. A problem that is $\mathcal{NP}$-complete can be solved in polynomial time iff all other $\mathcal{NP}$-complete problems can also be solved in polynomial time

$\mathcal{NP}$-hard. If an $\mathcal{NP}$-hard problem can be solved in polynomial time then all $\mathcal{NP}$-complete problems can also be solved in polynomial time

- All $\mathcal{NP}$-complete problems are $\mathcal{NP}$-hard but some $\mathcal{NP}$-hard problems are known not to be $\mathcal{NP}$-complete

$$\mathcal{NP}\text{-complete} \subset \mathcal{NP}\text{-hard}$$

- $\mathcal{P}$ vs $\mathcal{NP}$ problems
  - The problems in class $\mathcal{P}$ can be solved in $O(N^k)$ time, for some constant k (polynomial time)
  - The problems in class $\mathcal{NP}$ can be *verified* in polynomial time
    - If we are given a *certificate* of a solution, we can verify that the certificate is correct in polynomial time in the size of input to the problem
  - Some polynomial-time solvable problems look very similar to $\mathcal{NP}$-complete problems

NP Complete Problems

To prove whether particular problem is NP complete or not we use polynomial time reducibility. That means if

$$A \xrightarrow{\text{Poly}} B \text{ and } B \xrightarrow{\text{Poly}} C \text{ then } A \xrightarrow{\text{Poly}} C.$$

1. CNF - SAT problem

This problem is based on Boolean formula. The Boolean formula has various Boolean operations such as OR(+), AND ($\cdot$) and NOT. There are some notations such as $\rightarrow$ (means implies) and $\leftrightarrow$ (means if and only if).

A Boolean formula is in **Conjuctive Normal Form (CNF)** if it is formed as collection of subexpressions. These subexpressions are called **clauses**.

For example

$(\bar{a} + b + d + \bar{g})\ (c + \bar{e})\ (\bar{b} + d + \bar{f} + h)\ (a + c + e + \bar{h})$

This formula evaluates to 1 if b, c, d are 1.

The CNF-SAT is a problem which takes Boolean formula in CNF form and checks whether any assignment is there to Boolean values so that formula evaluates to 1.

Theorem : CNF-SAT is in NP complete.

Proof : Let S be the Boolean formula for which we can construct a simple non-deterministic algorithm which can guess the values of variables in Boolean formula and then evaluates each clause of S. If all the clauses evaluate S to 1 then S is satisfied. Thus CNF-SAT is in NP-complete.

2. A 3SAT problem

A 3SAT problem is a problem which takes a Boolean formula S in CNF form with each clause having **exactly three literals** and check whether S is satisfied or not. [Note that CNF means each literal is ORed to form a clause, and each clause is ANDed to form Boolean formula S].

Following formula is an instance of 3SAT problem :

$(\bar{a} + b + \bar{g})\ (c + \bar{e} + f)\ (\bar{b} + d + \bar{f})\ (a + e + \bar{h})$

Theorem : 3SAT is in NP complete.

Proof : Let S be the Boolean formula having 3 literals in each clause for which we can construct a simple non-deterministic algorithm which can guess an assignment of Boolean values to S. If the S is evaluated as 1 then S is satisfied. Thus we can prove that 3SAT is in NP-complete.

Some other NP complete problems

1. **The 0/1 Knapsack Problem** - It can be proved as NP complete by reduction from SUM OF SUBSET problem.
2. **Hamiltonian Cycle** - It can be proved as NP complete, by reduction from vertex cover.
3. **Travelling Salesperson Problem** - It can be proved as NP complete, by reduction from Hamiltonian cycle.

Properties of NP-Complete and NP-Hard Problems

- As we know, P denotes the class of all deterministic polynomial language problems and NP denotes the class of all non-deterministic polynomial language problems. Hence
  $P \subseteq NP$.
- The question of whether or not
  $P = NP$
  holds, is the most famous outstanding problem in the computer science.
- Problems which are known to lie in P are often called as *tractable*. Problems which lie outside of P are often termed as *intractable*. Thus, the question of whether $P = NP$ or $P \neq NP$ is the same as that of asking whether there exist problems in NP which are intractable or not.

The relationship between P and NP is depicted by

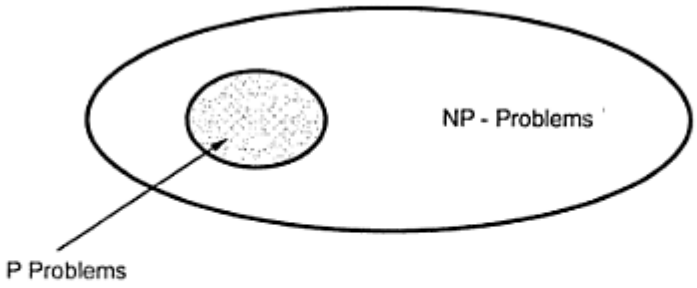


Fig. 7.16 P and NP problems assuming P ≠ NP

- We don't know if P = NP. However in 1971 S. A. Cook proved that a particular NP problem known as SAT(Satisfiability of sets of Boolean clauses) has the property that, if it is solvable in polynomial time, so are all NP problems. This is called a "NP-complete" problem.

- Let A and B are two problems then problem A reduces to B if and only if there is a way to solve A by deterministic polynomial time algorithm using a deterministic algorithm that solves B in polynomial time.

  A reduces to B can be denoted as $A \propto B$. In other words we can say that if there exists any polynomial time algorithm which solves B then we can solve A in polynomial time. We can also state that if $A \propto B$ and $B \propto C$ then $A \propto C$.

- A NP problem such that, if it is in P, then NP = P. If a (not necessarily NP) problem has this same property then it is called "NP-hard". Thus the class of NP-complete problem is the intersection of the NP and NP-hard classes.

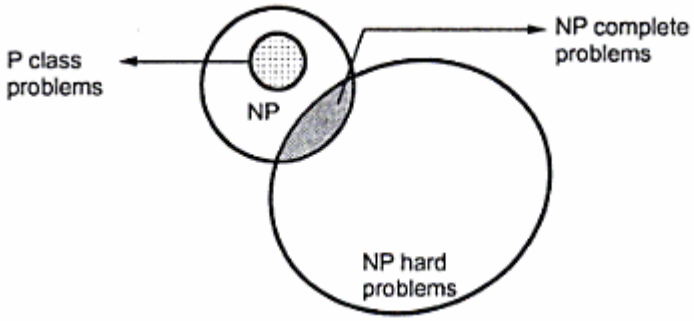
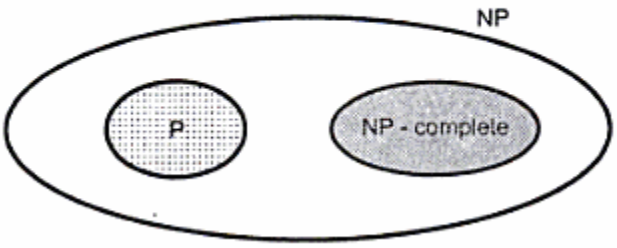


Fig. Relationship between P, NP, NP-complete and NP-hard problems

- Normally the decision problems are NP-complete but optimization problems are NP-hard. However if problem A is a decision problem and B is optimization problem then it is possible that A $\propto$ B. For instance the Knapsack decision problem can be Knapsack optimization problem.
- There are some NP-hard problems that are not NP-complete. For example *halting problem*. The halting problem states that : "Is it possible to determine whether an algorithm will ever halt or enter in a loop on certain input?"
- Two problems P and Q are said to be polynomially equivalent if and only if P $\propto$ Q and Q $\propto$ P.



NP-HARD GRAPH PROBLEMS

The strategy we shall adopt 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 (in polynomial deterministic time) an instance I' of L_2 from any instance I of L_1 such that from the solution of I' we can determine (in polynomial deterministic time) the solution to instance I to L_1.
iii) Conclude from (ii) that $L_1 \propto L_2$.
iv) Conclude from (i), (iii) and the transitivity of $\propto$ that L_2 is NP-hard.

For the first few proofs we shall go through all the above steps. Later proofs will explicitly deal only with steps (i) and (ii). An NP-hard decision problem L_2 can be shown NP-complete by exhibiting a polynomial time nondeterministic algorithm for L_2. All the NP-hard decision problems we shall deal with here are also NP-complete. The construction of polynomial time nondeterministic algorithms for these problems is left as an exercise.

CNF-satisfiability $\propto$ clique decision problem (CDP)

Proof: Let $F = \wedge_{1 \le i \le k} C_i$ be a propositional formula in CNF. Let x_i, $1 \le i \le n$ be the variables in F. We shall show how to construct from F a graph $G = (V, E)$ such that G will have a clique of size at least k iff F is satisfiable. If the length of F is m, then G will be obtainable from F in $O(m)$ time. Hence, if we have a polynomial time algorithm for CDP, then we can obtain a polynomial time algorithm for CNF-satisfiability using this construction.

For any F, $G = (V, E)$ is defined as follows: $V = \{\langle \sigma, i \rangle | \sigma$ is a literal in clause $C_i\}$; $E = \{(\langle \sigma, i \rangle, \langle \delta, j \rangle) | i \neq j$ and $\sigma \neq \bar{\delta}\}$. A sample construction is given in Example 11.11.

If F is satisfiable then there is a set of truth values for x_i, $1 \le i \le n$ such that each clause is true with this assignment. Thus, with this assignment there is at least one literal σ in each C_i such that σ is true. Let $S = \{\langle \sigma, i \rangle | \sigma$ is true in $C_i\}$ be a set containing exactly one $\langle \sigma, i \rangle$ for each i.

S forms a clique in G of size k. Similarly, if G has a clique $K = (V', E')$ of size at least k then let $S = \{\langle \sigma, i \rangle | \langle \sigma, i \rangle \in V'\}$. Clearly, $|S| = k$ as G has no clique of size more than k. Furthermore, if $S' = \{\sigma | \langle \sigma, i \rangle \in S$ for some $i\}$ then S' cannot contain both a literal δ and its complement $\bar{\delta}$ as there is no edge connecting $\langle \delta, i \rangle$ and $\langle \bar{\delta}, j \rangle$ in G. Hence by setting x_i = true if $x_i \in S'$ and x_i = false if $\bar{x}_i \in S'$ and choosing arbitrary truth values for variables not in S', we can satisfy all clauses in F. Hence, F is satisfiable iff G has a clique of size at least k. □

Example Consider $F = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$. The construction of Theorem 11.2 yields the graph:

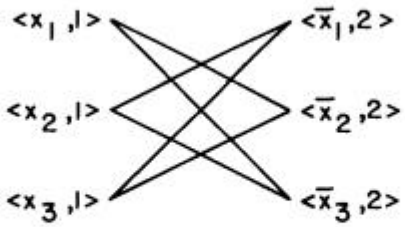


Figure A sample graph and satisfiability

This graph contains six cliques of size two. Consider the clique with vertices $\{\langle x_1, 1 \rangle, \langle \bar{x}_2, 2 \rangle\}$. By setting x_1 = true and $\bar{x}_2$ = true (i.e. x_2 = false) F is satisfied. x_3 may be set either to true or false. □

Node Cover Decision Problem

A set $S \subseteq V$ is a *node cover* for a graph $G = (V, E)$ iff all edges in E are incident to at least one vertex in S. The size of the cover, $|S|$, is the number of vertices in S.

Example Consider the graph:

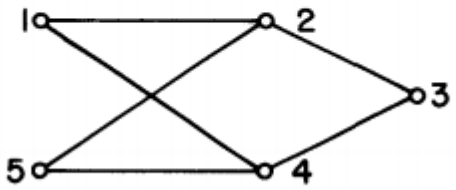


Figure A sample graph and node cover

$S = \{2, 4\}$ is a node cover of size 2. $S = \{1, 3, 5\}$ is a node cover of size 3. □

In the node cover decision problem (NCDP) we are given a graph G and an integer k. We are required to determine if G has a node cover of size at most k.

Theorem Clique decision problem (CDP) $\propto$ node cover decision problem (NCDP)

Proof: Let $G = (V, E)$ and k define an instance of CDP. Assume that $|V| = n$. We shall construct a graph G' such that G' has a node cover of size at most $n - k$ iff G has a clique of size at least k. Graph G' is given by $G' = (V, \overline{E})$ where $\overline{E} = \{(u, v) | u \in V, v \in V \text{ and } (u, v) \notin E\}$.

Now, we shall show that G has a clique of size at least k iff G' has a node cover of size at most $n - k$. Let K be any clique in G. Since there are no edges in $\overline{E}$ connecting vertices in K, the remaining $n - |K|$ vertices in G' must cover all edges in $\overline{E}$. Similarly, if S is a node cover of G' then $V - S$ must form a complete subgraph in G.

Since G' can be obtained from G in polynomial time, CDP can be solved in polynomial deterministic time if we have a polynomial time deterministic algorithm for NCDP. □

Note that since CNF-satisfiability $\propto$ CDP, CDP $\propto$ NCDP and $\propto$ is transitive, it follows that NCDP is NP-hard.

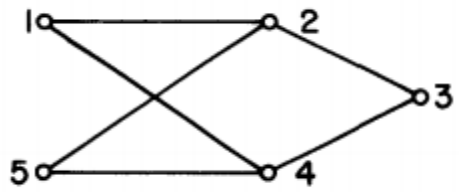
Chromatic Number Decision Problem (CN)

A coloring of a graph $G = (V, E)$ is a function $f:V \rightarrow \{1, 2, \ldots, k\}$ defined for all $i \in V$. If $(u, v) \in E$ then $f(u) \neq f(v)$. The *chromatic number decision problem* (CN) is to determine if G has a coloring for a given k.

Example A possible 2-coloring of the graph of Figure is: $f(1) = f(3) = f(5) = 1$ and $f(2) = f(4) = 2$. Clearly, this graph has no 1-coloring. □

In proving CN to be NP-hard we shall make use of the NP-hard problem SATY. This is the CNF satisfiability problem with the restriction that each clause has at most three literals. The reduction CNF-satisfiability $\propto$ SATY is left as an exercise.

Consider the graph:



Figure

COOK'S THEOREM

Before presenting Cook's theorem, let us state some of the related definitions, though we have defined previously: A decision problem is in NP if it can be solved by a non-deterministic algorithm in polynomial time. An instance of the Boolean satisfiability problem is a Boolean expression that combines Boolean variables using Boolean operators. An expression is satisfiable if there is some assignment of truth values to the variables that makes the entire expression true.

> Cook's Theorem states that any problem in class NP can be reduced to an instance of SAT in polynomial time.

Alternatively, it 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 a problem of determining whether a Boolean formula is satisfiable.

An important consequence of the theorem is this: if there were a deterministic polynomial time algorithm for solving Boolean satisfiability, then there would exist a deterministic polynomial time algorithm for solving all problems in NP. Crucially, the same follows for any NP complete problem as these are also in NP.

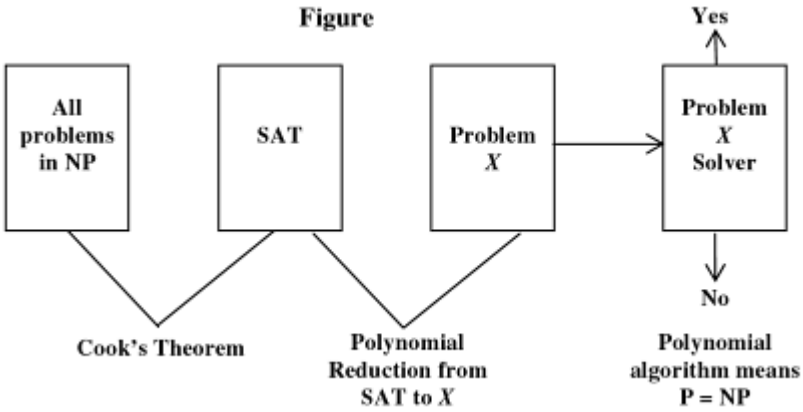
The question of whether such an algorithm exists is called the P = NP problem and it is widely considered the most important unsolved problem in theoretical computer science.

A decision problem is NP-hard if the time complexity on a deterministic machine is within a polynomial factor of the complexity of any problem in NP.

A problem is NP-complete if it is NP-hard and in NP.

Cook's theorem proved SATISFIABILITY was NP-hard by using a polynomial time reduction translating each problem in NP into an instance of SAT. Since a polynomial time algorithm for SAT would imply a polynomial time algorithm for everything in NP, SAT is NP-hard. Since we can guess a solution to SAT, it is in NP and thus NP-complete.

Figure



The proof of Cook's Theorem, while quite clever, was certainly difficult and complicated. We had to show that all problems in NP could be reduced to SAT to make sure we did not miss a hard one. But now that we have a known NP-complete problem in SAT. For any other problem, we can prove it NP-hard by polynomially transforming SAT to it!

Since the composition of two polynomial time reductions can be done in polynomial time, all we need show is that SAT, that is, any instance of SAT can be translated to an instance of *X* in polynomial time.

Definition *A problem A is $\mathcal{NP}$-hard if and only if satisfiability reduces to A (satisfiability $\propto A$). A problem A is $\mathcal{NP}$-complete if and only if A is $\mathcal{NP}$-hard and $A \in \mathcal{NP}$.*

- There are $\mathcal{NP}$-hard problems that are not $\mathcal{NP}$-complete
- Only a decision problem can be $\mathcal{NP}$-complete
- An optimization problem may be $\mathcal{NP}$-hard; cannot be $\mathcal{NP}$-complete
- If A is a decision problem and B is an optimization problem, it is quite possible that $A \propto B$
  - Knapsack decision problem can be reduced to the knapsack optimization problem
  - Clique decision problem reduces to clique optimization problem
- There are some $\mathcal{NP}$-hard decision problems that are not $\mathcal{NP}$-complete
- Example: Halting problem for deterministic algorithms
  - $\mathcal{NP}$-hard decision problem, but not $\mathcal{NP}$-complete
  - Determine for an arbitrary deterministic algorithm A and input I, whether A with input I ever terminates
  - Well known that halting problem is undecidable; there exists no algorithm of any complexity to solve halting problem
    - It clearly cannot be in $\mathcal{NP}$
  - To show that "satisfiability $\propto$ halting problem", construct an algorithm A whose input is a propositional formula X
    - If X has n variables, A tries out all the 2^n possible truth assignments and verifies whether X is satisfiable
    - If X is satisfiable, it stops; otherwise, A enters an infinite loop
    - Hence, A halts on input X iff X is satisfiable
  - If we had a polynomial time algorithm for halting problem, then we could solve the satisfiability problem in polynomial time using A and X as input to the algorithm for halting problem
  - Hence, halting problem is an $\mathcal{NP}$-hard problem that is not in $\mathcal{NP}$

Definition *Two problems A and B are said to be **polynomially equivalent** if and only if $A \propto B$ and $B \propto A$.*

- To show that a problem B is $\mathcal{NP}$-hard, it is adequate to show that $A \propto B$, where A is some problem already known to be $\mathcal{NP}$-hard
- Since $\propto$ is a transitive relation, it follows that if satisfiability $\propto A$ and $A \propto B$, then satisfiability $\propto B$
- To show that an $\mathcal{NP}$-hard decision problem is $\mathcal{NP}$-complete, we have just to exhibit a polynomial time nondeterministic algorithm for it