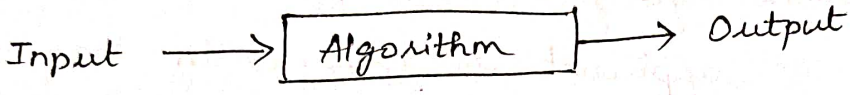
UNIT - V.

INTRODUCTION TO RANDOMIZED ALGORITHM:

* Deterministic Algorithm is an algorithm, which given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of steps.



* The solution produced by the algorithm is correct and the number of computational steps is same for different runs of the algorithm with same input.

Problems in deterministic algorithm:

* Given a computational problem,
  - difficult to formulate an algorithm with good running time
  - the explotion of running time of algorithm with the no. of inputs.

Randomized Algorithm:

* In addition to the input, the algorithm uses a source of pseudo random numbers. During execution, it takes random choices depending on those random numbers.

* The behavior (output) can vary if the algorithm is run multiple times on the same input.

Advantages:

- Simple and easy
- fast with high probability to produce optimum output.

HIRING PROBLEM:

* Assume a company is in need to hire a new office assistant and for this it uses an employment agency.

* The employment agency will send one candidate each day. The company will interview that person and then decide to either hire that person or not. The company has to pay the employment agency a small fee to interview a candidate. To hire an applicant is more costly than interviewing an applicant.

* The procedure Hire-Assistant, provides pseudocode for the hiring problem. It assumes that the candidates for the office assistant job are numbered 1 through n. The procedure, after interviewing candidate i, determine if candidate i is the best candidate seen so far. Therefore, to initialize, the procedure creates a dummy candidate, numbered 0, who is less qualified than each of the other candidates.

Algorithm:

```
Hire-Assistant(n)
best ← 0
for i ← 1 to n do
    interview candidate i
    if candidate i is better than candidate best
        then best ← i
        hire candidate i.
```

* The algorithm is concerned with the cost incurred by interviewing and hiring. The analytical techniques, used are identical whether we analyze cost or running time. In either case, it counts the number of times certain basic operations are executed.

* Interviewing has a low cost, say C_i, whereas hiring is expensive, costing C_h. Let n be the number of people interviewed and m be the number of people hired.

* Then, the total cost associated with this algorithm is

$O(nC_i + mC_h)$

Worst-case analysis:

* In worst case, every candidate interviewed are hired. This situation occurs if candidates come in increasing order of quality, in which case all n times candidates are hired, for a hiring cost of $O(nC_h)$. There is no idea about the order in which candidates arrive nor we have any control over this order.

Randomized Algorithm:

* For randomization, it is necessary to know something about the distribution of the inputs.

* In the hiring problem, it seems as if the candidates are being presented in random order, but no idea whether or not they are really are.

* Thus, to randomize the hiring problem, we must have greater control over the order in which the candidates are interviewed.

* Hence, slight changes are imposed in hiring problem. Assume the employment agency has n candidates and they send the list of candidates to the company in advance. On each day, the company chooses, randomly, which candidate to interview. Instead of relying on a guess that the candidates will come in random order, the company gained control of the process and enforced a random order.

* When randomized algorithm is applied on hiring problem, it first permutes the candidates and then determines the best candidate. On each permutation, it provides several ways of order of arrangement of candidates in the list.

* For the hiring problem, the only change needed in the code is to randomly permute the array.

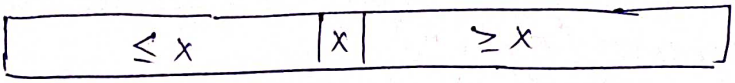
Algorithm:

```
Randomized - Hire - Assistant (n)
randomly permute the list of candidates
best ← 0
for i ← 1 to n do
    interview candidate i
        if candidate i is better than candidate best
            then best ← i
                hire candidate i
```

RANDOMIZED QUICK SORT:

QUICK SORT: DIVIDE AND CONQUER:

1. Divide : Partition the array into two subarrays around a pivot x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

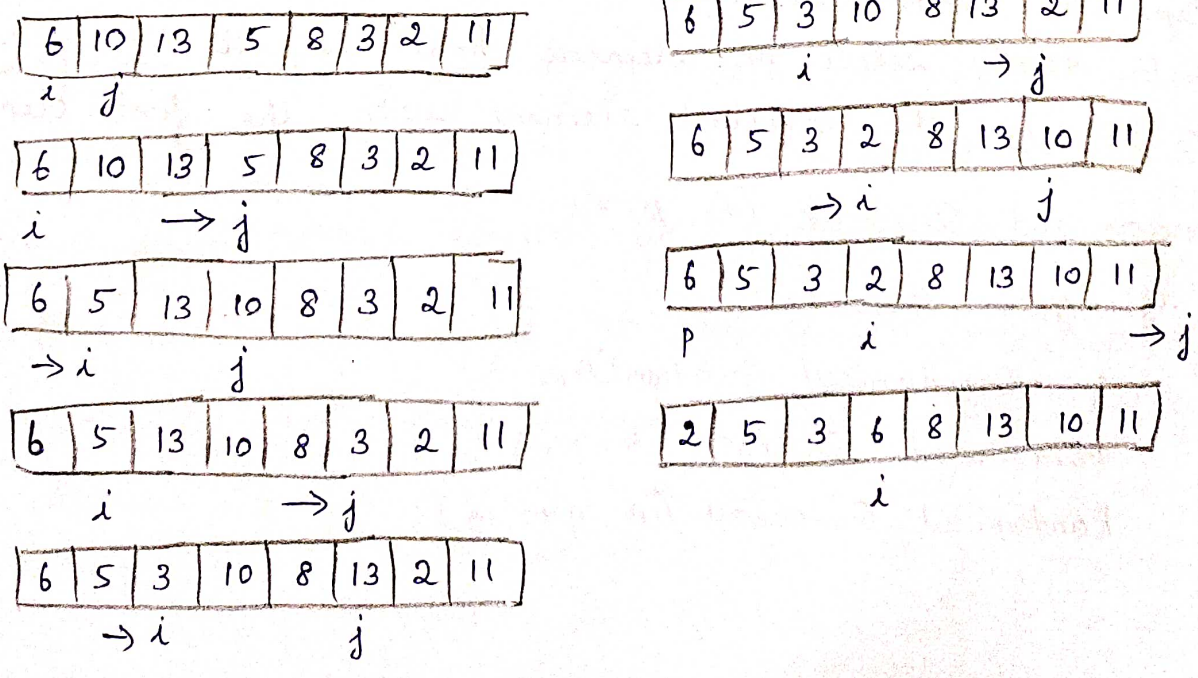


2. Conquer : Recursively sort the two subarrays.
3. Combine : Trivial

Partitioning Subroutine:

```
PARTITION (A, p, q)        ⇒ A[p..q]
                           ⇒ pivot = A[p]
  x ← A[p]
  i ← p
  for j ← p+1 to q
     do if A[j] ≤ x
           then i ← i+1
                exchange A[i] ↔ A[j]
  exchange A[p] ↔ A[i]
  return i
```

Example of partitioning



Pseudocode for Quicksort:

```
QUICKSORT (A, p, r)
    if p < r then
        q ← PARTITION (A, p, r)
        QUICKSORT (A, p, q-1)
        QUICKSORT (A, q+1, r)
```

Initial call : QUICKSORT (A, 1, n)

Worst case of Quicksort : Arises when input is sorted or reverse sorted. One side of partition always has no elements. $O(n^2)$ is worst case analysis of Quicksort.

Best case of Quicksort : Partition splits the array evenly.

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

Randomized Quicksort :

* To get better performance on sorted or nearly sorted data - we can randomize the algorithm to get the same effect as if the input data were random.
* Instead of explicitly permuting the input data, randomization can be accomplished trivially by random sampling of one of the array elements as the pivot.
* Steps in Randomization
  - Randomly select an element from the set
  - Exchange the selected element with the first element.

```
Randomized-Quicksort (A, p, r)
    if p < r then
        q = Randomized-Partition (A, p, r);
        Randomized-Quicksort (A, p, q-1);
        Randomized-Quicksort (A, q+1, r);
```

```
Randomized_Partition (A, p, r)
    i = Random (p, r);
    Exchange A[p] <-> A[i];
    return Partition (A, p, r);
```

STRING MATCHING:

* Given a string 's', the problem of string matching deals with finding whether a pattern 'p' occurs in 's' and if 'p' does occur then return the position in 's' where 'p' occurs.

* Find a given pattern P[1..m] in text T[1..n] with n >= m. Given a pattern P[1..m] and a text T[1..n], find all occurrences of P in T. P occurs with shift s : P[1] = T[s+1], P[2] = T[s+2], ... P[m] = T[s+m]. If so, call s is a valid shift, otherwise, an invalid shift.

O(mn) approach (or) Naive approach:

* One of the most obvious approach towards string matching problem would be to compare the first element of the pattern to be searched 'p', with the first element of the string 's' in which to locate 'p'.

* If the first element of 'p' matches the first element of 's', compare the second element of 'p' with second element of 's'. If match found proceed likewise until entire 'p' is found.

* If a mismatch is found at any position, shift 'p' one position to the right and repeat comparison beginning from first element of 'p'.

Example:

String S: a b c a b a a b c a b a c

Pattern P: a b a a

Step 1: Compare p[1] with S[1]

S: a b c a b a a b c a b a c

P: a b a a

Step 2: Compare p[2] with S[2]

S: a b c a b a a b c a b a c

P: a b a a

Step 3: Compare p[3] with s[3]

S: a b a c a b a a b c a b a c

P: a b a a

mismatch.

Since mismatch is detected, shift 'p' one position to right and repeat matching procedure.

S: a b c a b a a b c a b a c

P: a b a a

Finally, a match is found after shifting 'p' three times to the right.

Drawbacks:

* If 'm' is the length of the pattern 'p' and 'n' is the length of the string 'S', the matching time is of the order O(mn). This is certainly a very slow running algorithm.

* What makes this approach so slow is the fact that element of 'S' with which comparisons had been performed earlier are involved again and again in comparisons in some future iterations.

RABIN KARP ALGORITHM:

* The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared.

* If the hash values are unequal, the algorithm will determine the hash value for next M-char. sequence.

* If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence.

* In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

* Let the text be: ABCCDDAEFG and pattern be: CDD. Let n be the length of text and m be length of pattern. Therefore, n = 10 and m = 3. Let d be the no. of characters in the input set (ie) d = n, d = 10.

* Assign a numerical value for the characters in the input set. A = 1, B = 2, C = 3, D = 4, E = 5, ... etc.

* First, calculate the hash value for the pattern.

Hash value for the pattern = Summation of $(v * d^{m-1}) \bmod q$ where q is a prime number.

* Hash value of CDD (P).

$= (3 * 10^2 + 4 * 10^1 + 4 * 10^0) \% 13 = 344 \% 13 = 6.$

* Calculate the hash value for the text window of size m. Check the strings only if the hash value matches. Else, calculate the hash value of next text window of size m.

* Calculate the hash value of first m characters (ABC). If its hash value matches hash value of pattern, then check the strings. If string does not match, calculate hash value of next m characters BCC.

* Hash value for first text window ABC (t)

$= (1*10^2 + 2*10^1 + 3*10^0) \% 13 = 123 \% 13 = 6$.

* The hash value of the first window matches with p, so perform character matching between ABC and CDD. Since they do not match, go for next window.

Spurious Hit (Limitation of RK Alg)

* When the hash value of pattern matches with the hash value of a window of the text but the window is not the actual pattern then it is called a spurious hit.

* Spurious hit increases the time complexity of the algorithm. In order to minimize spurious hit, modulus are used. It greatly reduces spurious hit.

* Calculate the hash value of the next window by subtracting the first term and adding the next term as shown. The next window contains BCC. Previous window ABC. Remove value of A and add value of C.

* $t = (((1*10^2 + 2*10^1 + 3*10^0) * 10) - (1 \times 10^2) + (3*10^0)) \bmod 13$

$= 233 \bmod 13 = 12$.

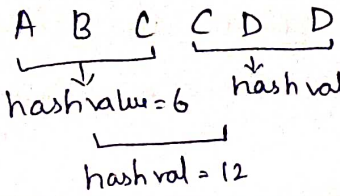
* In order to optimize this process, use the previous hash value in the following way.

$t = ((d * (t - v[\text{char to be removed}] * h) + v[\text{char to be added}]) \bmod 13$ where $h = d^{m-1} \bmod q$

$h = 10^2 \bmod 13 = 100 \% 13 = 9$

$t = (10 * (6 - 1*9) + 3) \% 13 = 12$

* After few searches, we will get the match for the window CDA in the text.



Algorithm:

```
Rabin-Karp-Matcher (T, p, d, q)
  n ← length [T]
  m ← length [p]
  h ← d^(m-1) mod q
  p ← 0
  to ← 0
  for i ← 1 to m
  do p ← (dp + p[i]) mod q
  to ← (dto + T[i]) mod q
  for s ← 0 to n-m
  do  if p = ts
  then if P[1.....m] = T[s+1..... s+m]
  then "Pattern occurs with shift" s
  If s < n-m
  then ts+1 ← (d[ts - T[s+1] h) + T[s+m+1]) mod q
```

Rabin-Karp Algorithm Complexity

* The average case and best case complexity of Rabin-Karp alg. is O(m+n) and worst case complexity is O(mn). The worst-case complexity occurs when spurious hits occur a number for all the windows.

APPROXIMATION ALGORITHMS.

* Approximation algorithms are used to solve optimization problems. An optimization problem is the problem of finding the best solution from all feasible solutions.

* The objective may be either minimum or maximum depending on the problem considered.

* The optimization problems have exponential time complexity and are NP-hard problems. For such problems, it is not possible to design algorithms that can find exactly optimal solutions to all instances of the problem in polynomial time.

* Approximation algorithms are
  - guaranteed to run in polynomial time.
  - guaranteed to get a solution which is close to the optimal solution.

* Challenge - need to prove solutions value is close to the optimum value, without even knowing what is the optimum value.

Definition of Approximation Alg:

* Given an optimization problem P, an algorithm A is said to be an approximation alg. for P, if for any given instance I, it returns an approximate solution, that is a feasible solution.

VERTEX COVER:

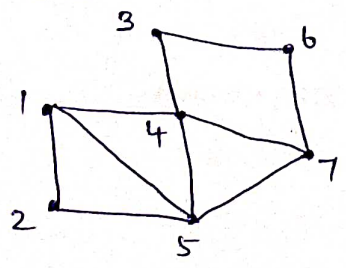
Definition:

Instance: An undirected graph $G = (V, E)$.

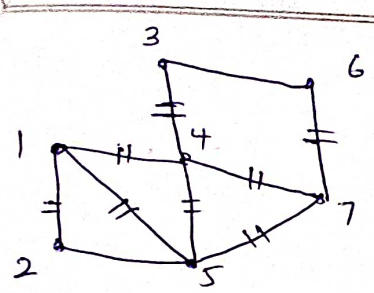
Feasible solution. A subset $C \subseteq V$ such that atleast one vertex of every edge of G belongs to C.

Value: The value of the solution is the size of the cover, $|C|$ and the goal is to minimize it.

Example:

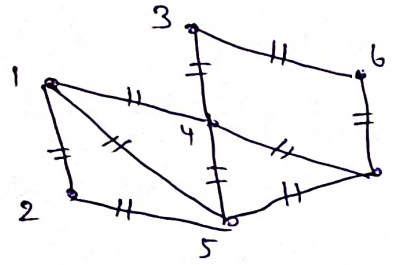


* In the given graph, choose vertices 1, 4 and 7 and check whether it forms a vertex cover. To do so, check all the edges covered by 1, 4 and 7. Edges covered by vertex 1 will be those edges having 1 as one of the endpoint.



* The vertices {1, 4, 7} does not include the edges (2,5) and (3,6). Hence, set of vertices {1, 4, 7} is not a vertex cover.

Eg 2: Consider vertex set {1, 2, 3, 5, 7}



* It includes all the edges. Hence, C = {1, 2, 3, 5, 7} is a vertex cover.
The size of vertex cover is |C| = 5.

* Now, check whether the solution can be minimized. (i.e) the no. of vertices in vertex cover should be minimum.

* Consider vertices {1, 4, 5, 6}

* It also includes all the edges. Hence, C = {1, 4, 5, 6} is a vertex cover and its size is |C| = 4.

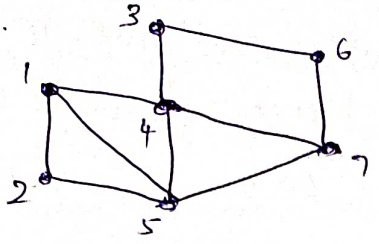
Vertex Cover – Approximation Algorithm:

APPROX – VERTEX – COVER (G)

1. $C = \phi$
2. $E' = G.E$
3. while $E' \neq \phi$
4. let (u, v) be an arbitrary edge of E'
5. $C = C \cup \{u, v\}$
6. remove from E' every edge incident on either u or v
7. return C.

* The brute force approach takes $O(2^n)$.
* Approximation alg takes – $O(V+E)$
$\Rightarrow O(n+n^2) = O(n^2)$.

Eg:



$E' = \{ (1,2), (1,5), (1,4), (2,5), (3,4), (3,6), (4,5), (4,7), (5,7), (6,7) \}$

$C = \{ \}$

Iteration 1: Choose an arbitrary edge say (1,2).

$C = \{1, 2\}$

* Remove edges containing vertices 1 and 2.

$E' = \{ (3,4), (3,6), (4,5), (4,7), (5,7), (6,7) \}$

Iteration 2: Choose an edge say (5,7)

$C = \{1, 2, 5, 7\}$

$E' = \{ (3,4), (3,6) \}$

Iteration 3: Choose an edge say (3,6)

$C = \{1, 2, 5, 7, 3, 6\}$

$E' = \{ \}$

$|C| = 6.$

Vertex Cover Performance Proof:

$|C| <= 2|C^*|$ where $|C|$ is the size of the vertex cover by approximation algorithm and $|C^*|$ is the size of minimum vertex cover.

* In the above eg, $|C| = 6$ and $|C^*| = 4$.

* $|C| \leq 2|C^*|$

$6 \leq 2 \times 4 \Rightarrow 6 \leq 8.$

Hence proved.

P, NP, NP-Complete and NP-Hard

P - Polynomial

NP - Non-deterministic Polynomial

* This is a research topic.

* Many algorithms are discussed throughout the subject. The algorithms are now classified into two types based on their time complexity or running time.

* Polynomial time
  - ~~Bin~~ Linear Search - $O(n)$
  - Binary Search - $O(\log n)$
  - Insertion Sort - $O(n^2)$
  - Merge Sort - $O(n \log n)$
  - Matrix multiplication - $O(n^3)$.

* Exponential time
  - 0/1 Knapsack
  - Travelling Salesman Problem
  - Sum of Subsets
  - Hamiltonian cycle

  } $\Rightarrow O(2^n)$

* Research concept - try to reduce the execution time of the algorithms.

* Therefore, research work concentrates on reducing the running time of exponential time algorithms to polynomial time algorithms. W.K.T the exponential time of 2^n much greater than polynomial time of n.

* Frameworks (or) guidelines are set for the researches who work on designing polynomial time algorithms for exponential time algorithms. These guidelines are NP-hard and NP-complete

* Two points.

1. Show some similarity between the exponential time problems So that, if one problem is solved, using that the other problems can also be solved in the same manner.
   (ie) Solve the problem (or) atleast relate the problems.
2. If it is not possible to write polynomial time deterministic algorithm, atleast try to write polynomial time non-deterministic algorithm.

Deterministic and Non-Deterministic Algorithm:

* Most of the algorithms are deterministic. Each and every statement and the working of the statement is clear and known to the algorithm writer.

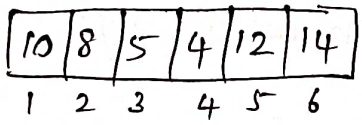
* In contrast, the working of the statements is not known in non-deterministic algorithm.

* So, on trying to write a polynomial-time deterministic algorithm, include both deterministic and non-deterministic statements. Later, some other researcher, may try to convert that non-deterministic statement to polynomial time deterministic statement.

Example: - Non-deterministic Algorithm

```
Algorithm NSearch (A, n, key)
{
    j = choice();        ——— O(1)
    if (key == A[j])
    {
        write(j);
        Success();       ——— O(1)
    }
    write(0);
    failure();           ——— O(1)
}
```

O(1) constant time algorithm.



key = 12
j = 5.

* The statement j = choice() is non-deterministic. The reason is that we dont know how come it selects the index of the key element. Hence, in future, this stmt can be converted into polynomial time deterministic stmt that takes O(1) execution time.

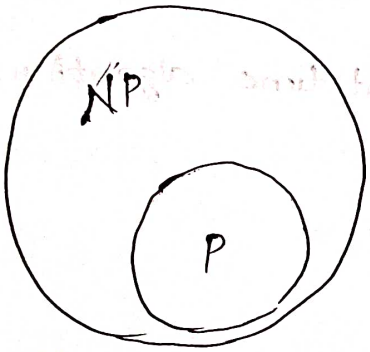
* In this way, for exponential time algorithm, polynomial time algorithms can be written with a mix of deterministic and non-deterministic statements.

Two classes of problems:

P - set of deterministic algorithms that take polynomial time of execution.

eg, linear search, sorting algorithms.
binary search, single source shortest path alg, minimum spanning tree, Huffman Coding.

NP - set of non-deterministic algorithms that take polynomial time of execution.



* In this diagram, P is shown as subset of NP (ie) the deterministic algorithm known today where part of non-deterministic alg.

* The deterministic algorithms at present where part of non-deterministic alg. The non-deterministic algorithms at present may become part of deterministic alg. in future.

* This is given by Cook's theorem. $P \subseteq NP$.

To provide relationship between exponential time algorithms:

* To relate all these problems in exponential time, a base problem is needed. Satisfiability problem acts as the base problem.

Satisfiability Problem:

* This is referred to as CNF-satisfiability problem.

* It uses CNF formula. That is propositional calculus formula using boolean variables. Assume there are 3 boolean variables x_1, x_2 and x_3.

$$x_i = \{ x_1, x_2, x_3 \}$$

* Assume an example CNF formula,

$$CNF = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$$

* The formula has 2 clauses C_1 & C_2 where $C_1 = x_1 \vee \bar{x}_2 \vee x_3$ and $C_2 = \bar{x}_1 \vee x_2 \vee \bar{x}_3$. Clauses are formed using disjunction and clauses are combined using conjunction.

* The satisfiability problem is to find for which values of x_i (x_1, x_2 and x_3), the CNF formula will be true.

* Now, what are the possible values for x_i

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

* We need to try each combination in the formula and find out whether it returns true.

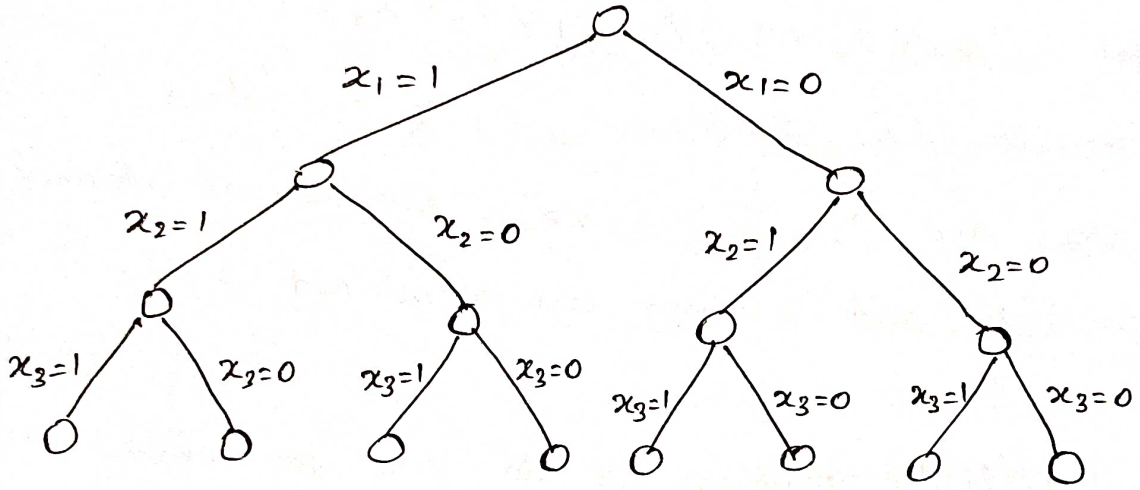
* Since 8 possible values, there are 8 tries.

(ie) 2^3.

For n number of bits, the time to check and get the result is 2^n.

* Hence, satisfiability problem is also an exponential time algorithm. and its execution time is 2^n (exponential time) similar to other exponential time problems.

* The same possible values can be represented as state space tree.



* The paths from the root to the leaf of these tree gives a solution.

* The next step is to show that the satisfiability problem is similar to exponential time algorithm like 0/1 knapsack, TSP, Hamiltonian cycle, sum of subsets problem.

* Hence, if satisfiability problem is solved in polynomial time, then all the exponential time problem can also be solved in polynomial time. Hence, no need to solve problems individually.

* Now, let us relate the satisfiability problem with 0/1 knapsack.

$P = \{10, 8, 12\}$ $\quad n = 3$

$W = \{3, 5, 4\}$ $\quad m = 8$

The solution for this 0/1 knapsack can be represented as

$$x_i = \{ \underline{0/1}, \underline{0/1}, \underline{0/1} \}$$

(ie).

x_1	x_2	x_3
0	0	0
:		
1	1	1.

$\Rightarrow 2^3$ (ie) 2^n.

In satisfiability problem, we need to check the formula to be true whereas in 0/1 knapsack, the profit is to be maximized. Also 0/1 knapsack problem can also be solved using state space tree.

NP_Hard and NP_Complete.

* Exponential time Algorithms - Satisfiability problem, 0/1 knapsack, TSP, Sum of subsets, Hamiltonian cycle.

* Let us consider all these problems as hard problems. Let us assume Satisfiability pblm as NP_hard, then all the problems are also NP_hard.

Reduction:

* Reduction is a technique used to relate problems.

* Let us take 2 problems - Satisfiability problem and 0/1 knapsack problem. Satisfiability problem reduces to 0/1 knapsack problem.

Sat $\propto$ 0/1 knapsack

(I_1) (I_2)

Instance

* To show relationship between 2 problems, we take an example formula for satisfiability and that formula can be converted into 0/1 knapsack. This conversion should be simple and should take only polynomial time.

* If satisfiability is NP-hard, then it reduces 0/1 knapsack problem and hence 0/1 knapsack is also NP-hard.

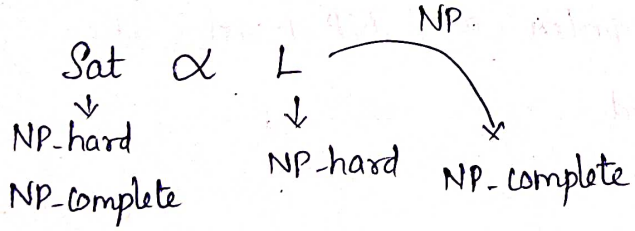
* The reduction has a transitive property.

Sat $\propto L_1$, $L_1 \propto L_2$

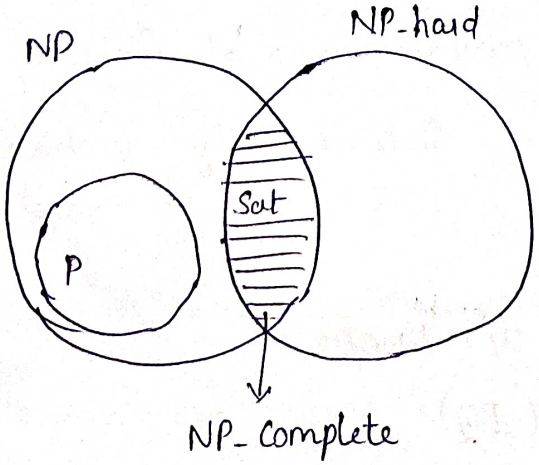
(ie) L_1 is NP-hard and L_2 is also NP-hard.

* If a NP-hard problem has a non-deterministic alg with polynomial time, then it is also NP-complete.

* The base problem (ie) satisfiability problem has a non-deterministic polynomial time alg. Hence, it is both NP-hard and NP-complete.



Diagrammatic Representation.



* The size of P gets increases, as non-deterministic alg converts to deterministic

$P \subseteq NP$

$P = NP$ ⇒ Cook's theorem.

⇓

Satisfiability will be P only if $P = NP$.

(To be proved).