```
{
    if place (k, i) then
    {
        x[k] = i ;
        if (k=n) then
            write (x[1 ... n]), else
            NQueen(k+1, n)
    }
}
```

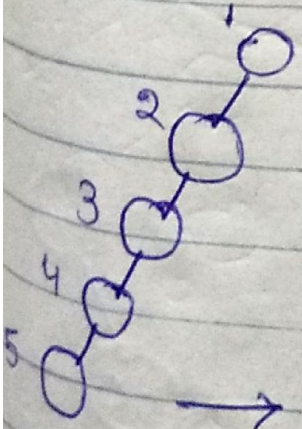
3/10/18

Optimal Binary Search Tree.

N → keys

kth key — r_k is the frequency or probability of appearing.

d_k is the distance of the kth key from root.



→ $d_k = 5$

We have to minimize this term:-

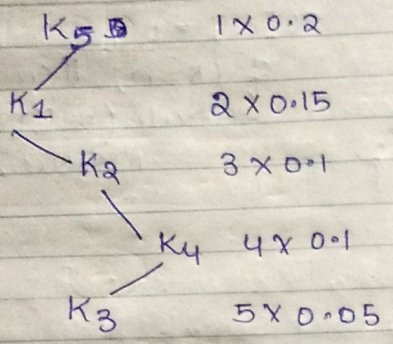
$$\min \sum_{k \in N} r_k d_k$$

↳ this will give minimum cost

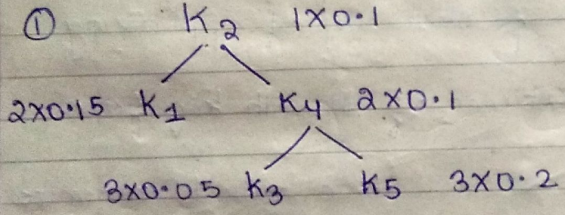
We will assume that all the keys given are sorted ie, $k_1 < k_2 < k_3 < k_4 < k_5$

Prob. → 0.15 0.1 0.05 0.1 0.2

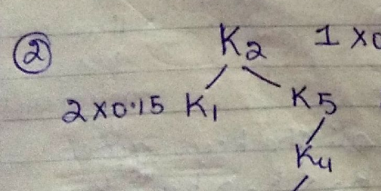
$n = 5$.



Any random tree

① 

Searching cost = 1.35.

② 

Searching cost = 1.3.

So, it is not necessary that a tree with maximum level will be optimal or with a minimum level will be optimal.

So this is how we find optimal search tree

keys:	A	B	C	D
freq/prob:	23	10	8	12

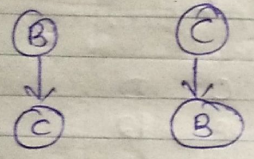
	A	B	C	D
A	23			
B	43	10		
C	–	26	8	
D	–	–	28	12

M[i, j] → It will give cost of searching if we includes keys from i to j

	A	B	C	D
A	23			
B	43	10		
C	67	26	8	
D	104	52	28	12

	A	B	C	D
A	A			
B	A	B		
C	A	B	C	
D	B	C	D	D

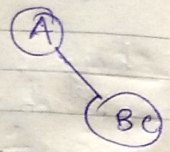
2-probabilities



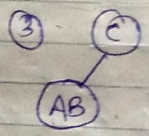
= B + 2xC
= 10 + 2x8
= 26

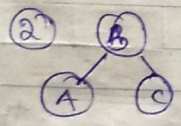
= C + 2xB
= 8 + 2x10
= 28

ABC

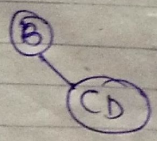


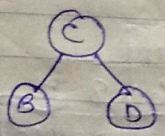
= BC + B + C + A
= 26 + 10 + 8 + 23
= 67.

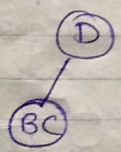




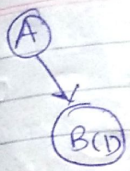
BCD



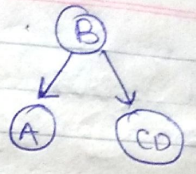




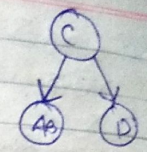
ABCD.



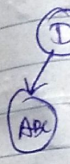
= BCD + B + C + D + A



= 2A + CD + C + D + B

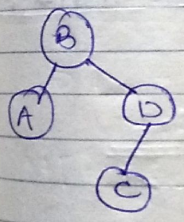


= AB + 2D + A + B + C



= D + A + B + C + ABC

Ans = 104 = Search cost of the tree.



This is the optimal binary search tree.

i	1	2	3	4	5
	0.15	0.1	0.05	0.1	0.2

There will be two trees, make both of them and find if the ans for both of them is same ie, searching cost of both trees are same.

0·15 0·35 0·1
0·2 0·3

Sol-

1	2	3	4	5
0·15	0·1	0·05	0·1	0·2

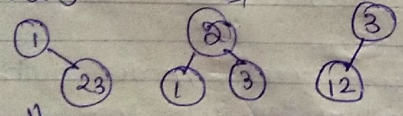
	1	2	3	4	5
1	0·15				
2	–	0·1			
3	–	–	0·05		
4	–	–	–	0·1	
5	–	–	–	–	0·2

	1	2	3	4	5
1	0·15				
2	0·35	0·1			
3	–	0·2	0·05		
4	–	–	0·2	0·1	
5	–	–	–	0·4	0·2

	1	2	3	4	5
1	1				
2	1	2			
3	1 or 2	2	3		
4	1 or 3	3 or 4	4	4	
5	2 or 4	4	5	5	5

	1	2	3	4	5
1	0·15				
2	0·35	0·1			
3	0·5	0·2	0·05		
4	–	0·45	0·2	0·1	
5	–	–	0·55	0·4	0·2

123



$\Rightarrow 0·3+0·1+0·1=0·5$

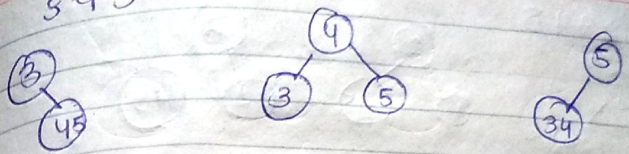
$= 0·2+0·1+0·05+0·15$

$= 0·5$

$= 0·35+0·15+0·1+0·05$

$= 0·65$

345



$= 0·05+0·1+0·2+0·4$

$= 0·75$

$= 0·1+0·1+0·4$

$= 0·6$

$= 0·2+0·1+0·05+0·2$

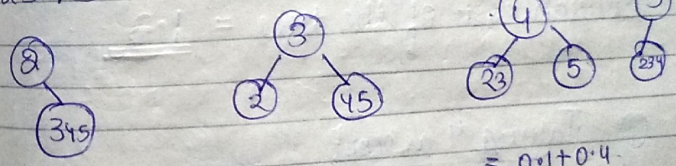
$= 0·55$

	1	2	3	4	5
1	0·15				
2	0·35	0·1			
3	0·5	0·2	0·05		
4	0·85	0·45	0·2	0·1	
5	1·3	0·85	0·55	0·4	0·2

$= 0·9$

$0·2+0·45+0·1+0·05+0·1$

2345



$= 0·55+0·05+0·1+0·2+0·1$

$= 1·0$

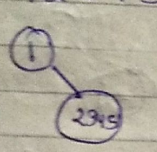
$= 0·2+0·05+0·4+0·1+0·2$

$= 0·95$

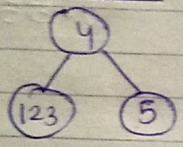
$= 0·1+0·4+0·2+0·1+0·05$

$= 0·85$

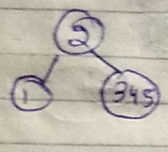
12345



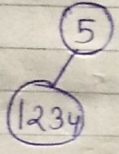
= 0.15 + 0.85 + 0.45
= 1.45



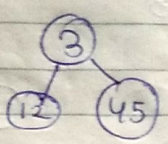
= 0.5 + 0.5 + 0.3
= 1.3



= 0.4 + 0.55 + 0.35
= 1.3



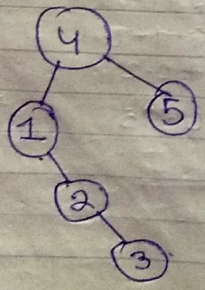
= 0.2 + 0.4 + 0.85
= 1.45



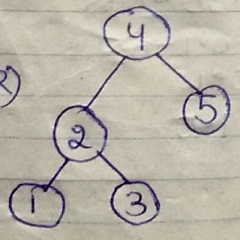
= 0.05 + 0.3 + 0.25 + 0.75
= 1.35

Search cost of the tree = 1.3

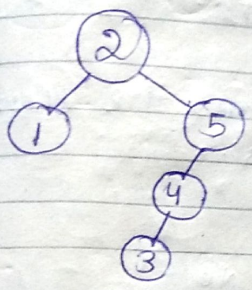
① Taking 4 as root



(OR)



② Taking 2 as root



24/10/19

Sum of Subset problem

You are given some positive numbers n and the weight of ith item is w_i and an integer m is given, so you want to find all the subsets of w_i whose sum is equal to m.

Eg: n = 3 m = 6 $w_1 = 2$; $w_2 = 4$; $w_3 = 6$

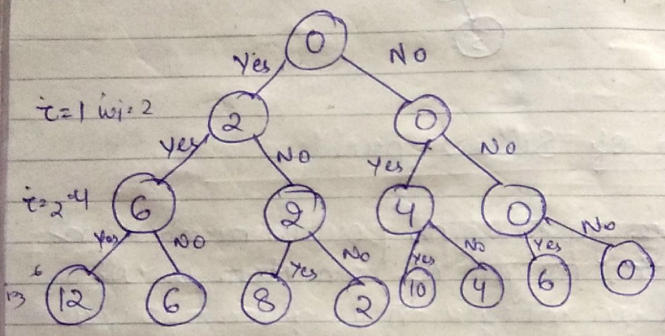
Sol: {2, 4} or {6}

If we have n numbers then depth of tree = n.

We will have two branches, left branch ⇒ include

→ All the branches are space state and the branch leading to solution is solution state.

→ The node contains the sum of elements so far.



This is a variable length tupple solution as solution tupple can be {2,4} (2 tupple soln) and {6} (1 tupple soln).

We can give fixed length tupple solution of the problem (having tupple length = n).
as tupple value can be 0 or 1,
ie, 0 → exclude
1 → include.

{2,4} → {1,1,0}
{6} → {0,0,1}
] fixed length tupple solution

Pruned state space tree → ~~The tree branches~~ The tree left after backtracking or removing non-promising nodes.

We call a node non-promising if it cannot lead to a feasible (or optimal) solution, otherwise promising

Promising → move further
non-promising → backtrack to node's parent.

Q $w_1 = 3$; $w_2 = 4$; $w_3 = 5$
$w_4 = 6$; $S = 13$

Weight so far = weight of node, i.e., sum of numbers included in partial solution node represents.

Total Possible Left = weight of the remaining items $i+1$ to n (for a node at depth i)

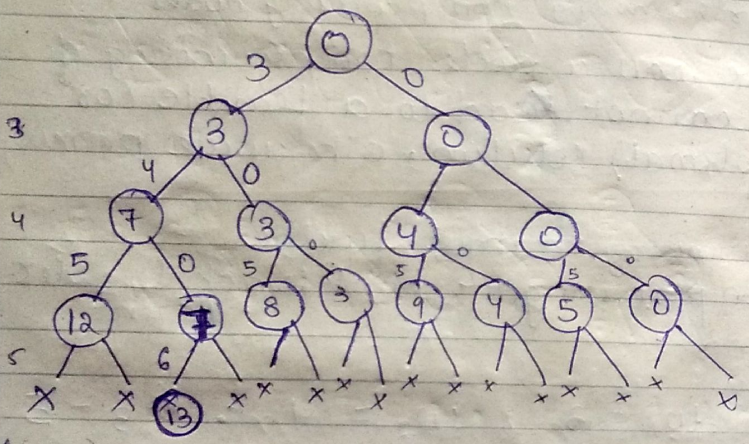
A node at depth i is a non-promising

if (WSF + TPL < S)

or (WSF + W[i+1] > S)

WSF → weightsofar.

TPL → Totalpossibleleft.



sumofSubsets (i, weightsofar, TPL)

if (promising(i)) // may lead to soln

then if (WSF == S)

then print include[I] to include[i]

// found soln.

else // expand the node when WSF < S.

include[i+1] = "Yes"

sumofSubsets(i+1, WSF + W[i+1], TSF - W[i+1])

include[i+1] = "no" // try excluding

sumofSubsets(i+1, WSF, TPL - W[i+1])

---

boolean promising(i)

return (WSF + TPL >= S) &&

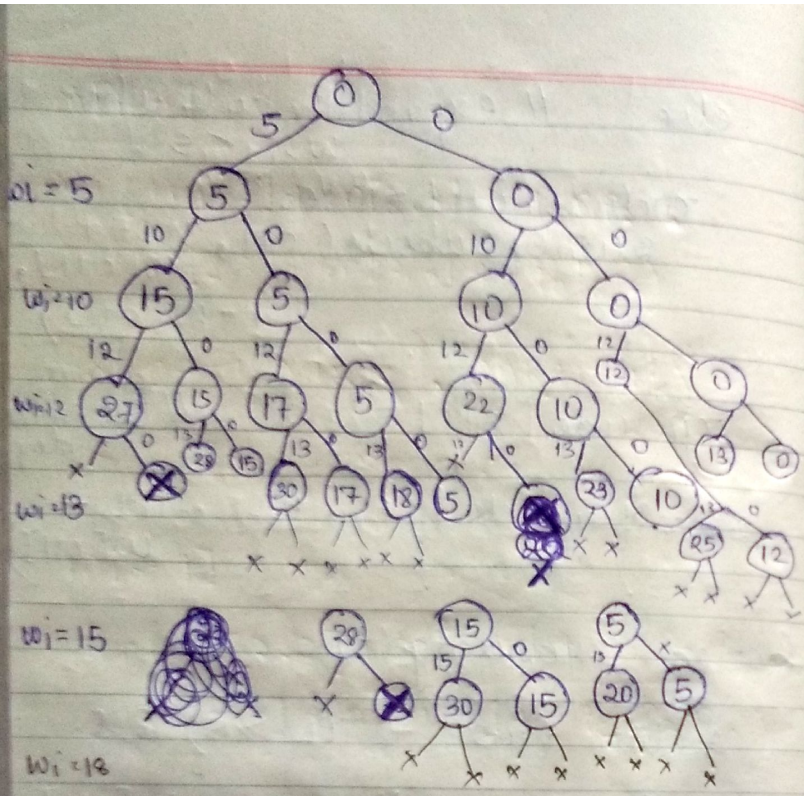
(WSF == S || WSF + W[i+1] < S)

Print all solutions.

n = 6 ; m = 30.

W[1...6] = {12, 5, 13, 10, 18, 15}

Sorted order - 5, 10, 12, 13, 15, 18



Tutorial

Q We are given four matrices M_1, M_2, M_3 and M_4. Dimensions are:-

$M_1 \rightarrow p \times q$ $\quad M_3 = r \times s$

$M_2 \rightarrow q \times r$ $\quad M_4 = s \times t$

where $p = 10$, $q = 100$, $r = 20$, $s = 5$, $t = 80$.

Find the optimal parenthesis for the matrix multiplication from M_1 to M_4.

$p_0 = p$
$p_1 = q$
$p_2 = r$
$p_3 = s$
$p_4 = t$

$A_1 = p \times q$
$A_2 = q \times r$
$A_3 = r \times s$
$A_4 = s \times t$

	1	2	3	4
1	0	20K	15K	19K
2		0	10K	50K
3			0	8K
4				0

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

$m[1-2] = p_0 \times p_1 \times p_2 = 10 \times 100 \times 20 = 20000$

$m[2-3] = p_1 \times p_2 \times p_3$
$= 100 \times 20 \times 5 = 10000$

$m[3-4] = 20 \times 5 \times 80 = 8000$

m[1-3]

$\min(m[1,1] + m[2,3] + P_0P_1P_3,$
$m[1,2] + m[3,3] + P_0P_2P_3)$

$= \min(0 + 10K + 5K, 20K + 0 + 1K)$

$= 15K = 15000$

m[2-4]

$= \min(m[2,2] + m[3,4] + P_1P_2P_4,$
$m[2,3] + m[4,4] + P_1P_3P_4)$

$= \min(0 + 8K + 160K, 10K + 0 + 40K)$

$= 50K$

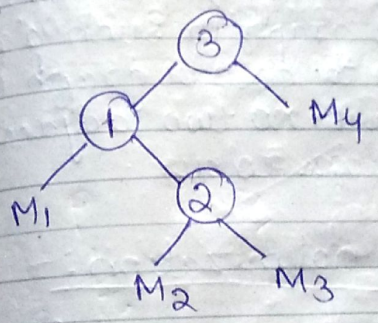
m[1-4]

$= \min(m[1,1] + m[2,4] + P_0P_1P_4,$
$m[1,2] + m[3,4] + P_0P_2P_4,$
$m[1,3] + m[4,4] + P_0P_3P_4)$

$= \min(0 + 50K + 80K, 20K + 8K + 16K, 15K + 0 + 4K)$

$= 19K$

$(M_1(M_2M_3))M_4$



26/10/

Graph coloring

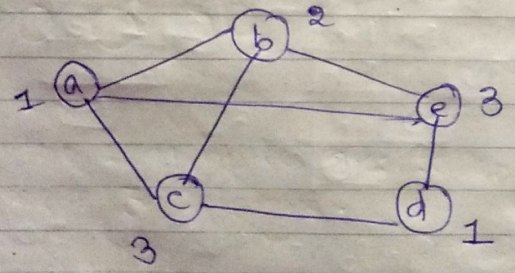
There are two versions of this problem:-

① M-colorability Decision

↳ we have a graph G and an integer m (m no:- of colours). We want to colour the graph in such a manner that we have to colour graph with m colours with no two adjacent nodes have same colour.

② M-colorability optimization:-

→ Here we have to color graph with minimum no:- of colours.



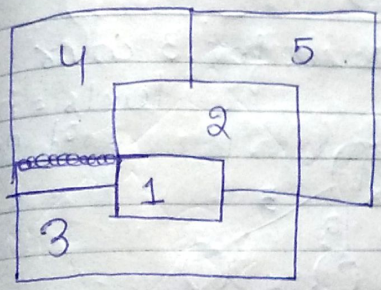
For m-colorability optimization problem, we have to use 3 colours.

So m = 3 (chromatic

4- color problem

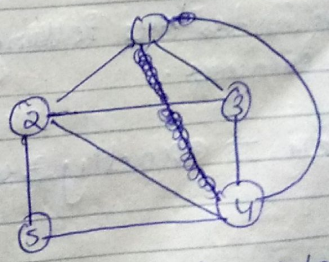
(Planar graph). → In which no two edges cross each other.

Application → For coloring of maps.



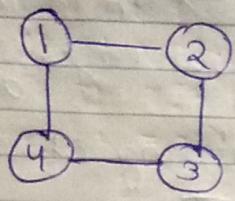
Each region of a graph can be denoted as nodes.

→ We will have edges b/w two nodes, when the regions are adjacent to each other.



* All the maps can be coloured by

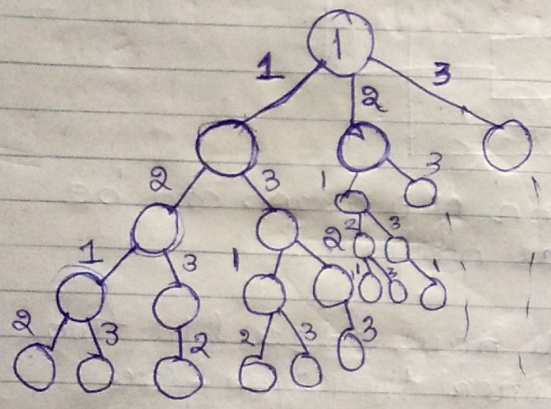
Eg:- 3-coloring problem



(We can use 3 colors at most)

We have 3 choices for 3 color problem.

levels = no:- of nodes + 1



All the leaf nodes are telling → atmost 3 colours.

12 solution → exactly 3 colours

adjacency matrix.

G[i, j] = 1. (1 → edge is there).

(1 ... m) → colours.

$(x_1, x_2 \cdots x_n)$ n-tupple solution where x_i is colour of node.

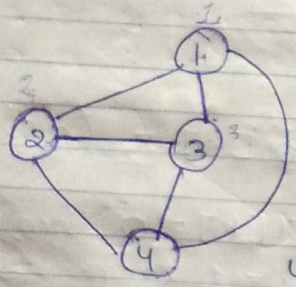
x[ ] with 0.

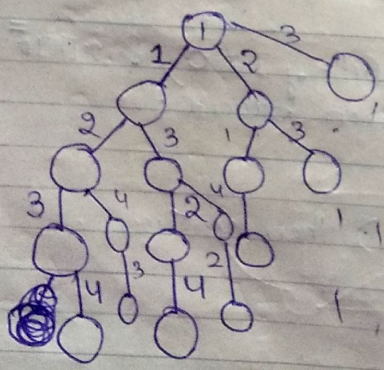
```
Next value (k)        // k is for kth node
    x[1] . . . . . x[k-1]. // colours are assigned to all the previous node
{
repeat
{
        x[k] = (x[k]+1) mod (m+1);
    if x[k] = 0 then return
    for j = 1 to n do
    {
        if (G[k,j] ≠ 0) & x[k] = x[j]
        then break;
    }
    if (j = n+1) then return.
```

→ m coloring(1) (Initial)

```
m coloring (k)
repeat.
{
    next value (k)
        if x[k]=0 then return
    if (k = n)
        write (x[1 ··· n])
    else mcoloring (k+1);
} until (false);
```

Q





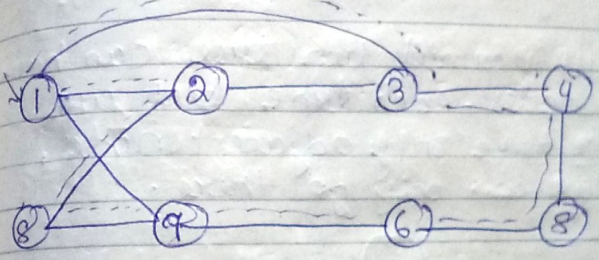
This is a 4 coloring problem and always all the nodes will be assigned 4 different colors.

3/10/18

Hamiltonian Cycle.

It is a round trip path in a graph G that visit every vertex once and return to the starting point.

→ Can be more than one.



→ If we don't reach the starting point, then it is hamiltonian path.

→ Travelling salesman problem is also a hamiltonian path but it is a minimum hamiltonian path.

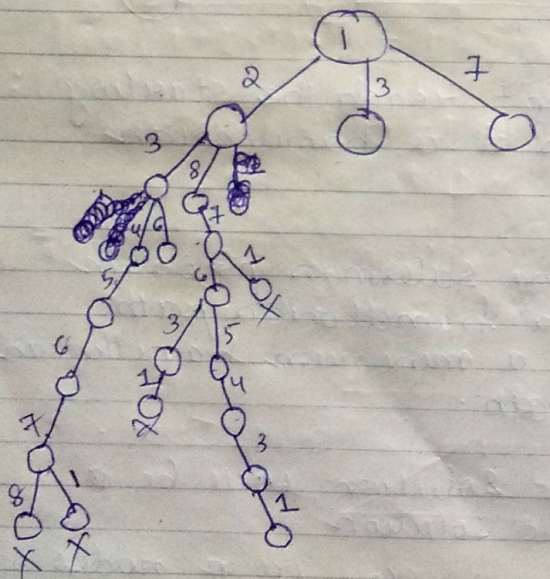
→ Here, the solution will be a n-tupple solution $(x_1, x_2 \cdots x_n)$ are n nodes.

The solution tupple can represent -
x_i → sequence of vertices you visited.

(2, 8, 7, 6, 5, 4, 3, 1)

↳ We started from 1 and the sequence is 2, 8, 7 ... 1.

→ Full permutation tree can have all nodes, each time but we will take only adjacent nodes, as we can only go there.



→ Bounding function should have 2 conditions.

1. next node should be adjacency list
2. distinct node
3. leaf node should be 1
4. n distinct nodes should be traversed

→ First we will check that out of all nodes, which nodes are in the adjacency list, then we will see all the adjacency nodes should not be traversed previously, then we will check that leaf node should be 1 and finally check that all the n nodes should be traversed and then it is again pointing to 1.

Algorithm

Graph → G[1...n]

Solution tupple $(x_1, x_2, x_3, x_4 \cdots x_n)$

$(1, 0, 0, 0 \cdots 0)$

↓

Starting node.

Branch & Bound

Hamiltonian (2)

```
repeat
{
  next value(k);
  if (x[k]=0) then exit
    if (k=n)
      Print x[1-n]
    else Hamiltonian (k+1)
} until (false);
}
```

Next value (k)

```
repeat
{
x[k] = (x[k]+1) mod (n+1)
if x[k] = 0 then return (// Cycle not found)
if (G(x[k-1], x[k]) ≠ 0);
{
 for j = 1 to k-1
 do if x[j] = x[k]
      then break;
 if (j = k)
 if ((k<n) or (k=n)
   or G(x[k], x[1]) ≠ 0)
   then return
}
} until (false).
```

NP hard and NP complete problems

→ Non-deterministic polynomial time problems.

The problem goes like that, you have a set of n jobs and size of these jobs is $S_1, S_2 \ldots S_n$. If we have a (S_i) job then it will take (S_i) time to complete. You have two processors P_1 and P_2 (both are identical); so you have to write a program to schedule this jobs onto a processor and scheduling is done in such a way that last job finish fastest.

→ One attempt we can do is giving all jobs to one processor.

P_1: S_1, S_2, S_3, S_4, S_5 — Total time for S_5 = $S_1+S_2+S_3+S_4+S_5$.

→ One attempt is

	P_1	P_2	
(1)	S_1	S_2 (100	All even and all odd.
(1)	S_3	S_4 (100	S_5 → 300 time unit
(1)		(100)	→ 3 time unit.

This also fails.

→ Another attempt is giving jobs to lightly loaded processor

P_1	P_2	
S_1	S_2	if $S_1 > S_2$
S_4	S_3	$S_1 < (S_2 + S_3)$

But this also fails as we have given jobs arbitrarily. So we have to think of some optimal solution.

→ Another attempt, arrange all the jobs in ascending order.

Eg: S_i = 2 units & 3 units.

arrange $2 < 2 < 2 < 3 < 3$.

P_1	P_2
2	2
2	3
3	
7 units	

But without arranging.

P_1	P_2
2	3
2	3
2	(6)
6	

So this is also not optimal.

Another solution is Brute-Force approach.

Taking all the possible ~~solutions~~ subset. Then we will calculate for each subset. then find the fastest.

But suppose if we 1000 jobs then combinations = $2^{1000}!$

→ It will take years to solve this.

→ We cannot solve NP hard problems, we can only say that whether it is NP hard or not.

→ Till now whatever we have done, in that

① Hamiltonian (If large permutation tree)

② Subset problems

, etc.

02/11/18

Reduction :-

If you have any efficient algo for π_1 then you can find an efficient algo for π_2 (which uses π_1)

Efficient :-

For the reasonable set of I/Ps you can simulate your problem in reasonable time.

→ Brute force algo is not efficient.

→ If you have n inputs, then we have 2^n solution for Brute force

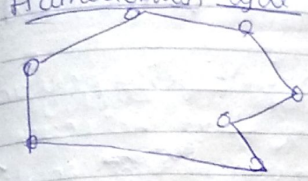
→ Efficient means that running time of your algo is bounded by polynomial in the I/P size.
For some constant c, running time of n $T(n)$ is $O(n^c)$.

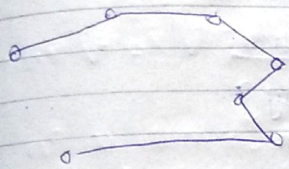
If $c = 1$, the our problem will run in $O(n)$ time.

$O(n)$; $O(n^2)$, $O(n^3)$, $O(n \log n)$ Vs 2^n

Efficient (polynomial) — not efficient

Hamiltonian cycle





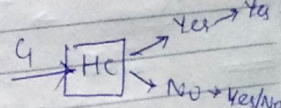
Hamiltonian path.

→ If there is a hamiltonian cycle in a graph then there will be Hamiltonian path.
(But vice-versa is not true always)

Reduction

Eff HC
↓
Eff HP.

Does a G has HP.



→ All the problems with O/P Yes/No (one bit) are known as Decision Problem.

→ All the problems with more than one bit output are known as Search problem.

For class of NP problems we need 2 persons:-

① Prover (Knowledgeous person)
② Verifier (Hard believer)
have time constraint.

Prover	Verifier
Yes	Why?

Eg: I/P → integer t.
O/P → t_1 & t_2
$t = t_1 \times t_2$ Composite no.-

Prover → It is a composite no:-
Verifier → How
Prover → Took t_1 & t_2 and showed the verifier that $t = t_1 \times t_2$. So the verifier believed.

Eg 2 Prover → Yes this graph has HC
Verifier → Why
P → Showed the edges.

Now, he took a bigger graph.
Prover → NO, the graph is not HC
Verifier → Why
Prover → He will have to show all possible combination (Brute force)

∴ here proving YES is easy but proving NO is difficult.
A decision prob is said to be in class NP, if for all YES I/Ps there is a proof using which it can be verified quickly/efficiently, that I/P is indeed a YES input.

→ There are 2 problems
① which can be solved in polynomial time
② which cannot be solved in polynomial time

Now we have 4 categories

Solve \ Proved	P	N
P	①	②
N	③	④

① Solve in polynomial time & prove in polynomial time
② " " but cannot " " "
③ Solve in non polynomial time but prove in Polynomial time

Imp

SAT : Satisfiability

→ Boolean variable → Yes / No
→ Literals → $x, \bar{x}$
→ Clause → OR of literals

$C_1 = Eg:-(x_1 \vee \bar{x_2} \vee \bar{x_3} \vee x_4 \cdots \vee x_n)$

→ A Boolean ~~function~~ Formula is in CNF (Conjunctive Normal form).
- AND of clauses

Eg $C_1 \wedge C_2 \wedge C_3 \wedge \cdots \wedge C_n$.

[BF in CNF is said to be satisfiable such that assignment to variable, such that formula evaluates to True]

Prob: SAT
input :- B.F.

Here to proving YES is also difficult
As to prove YES, we have to take all the combinations.
So, this is the hardest problem till now.
This is a ~~problem~~ known as

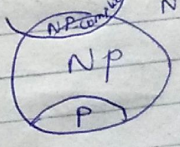
Cook's Theorem

If you can find the solution of the SAT problem, then you can find the solution of all NP problems.

① If there exist a polynomial time algo for SAT, then there is one for all NP problems

Red
Eff for SAT
↓
Eff for NP.

The problems which we can prove in polynomial time is NP problems



P → Polynomial time

$P \subseteq NP$.

NP-complete → which belongs NP-hard and NP.
{ which YES can be solved in P time
but NO cannot " " " " }

NP hard :- ~~Which~~ whose YES/NO, both cannot be solved in polynomial time. You can never solve this problem.

To deal with NP-complete problems we have a class of approximation algo. (which give near optimum solution).

→ We verify these problems on Non-deterministic Turing machine and is known as non-deterministic polynomial time problem.