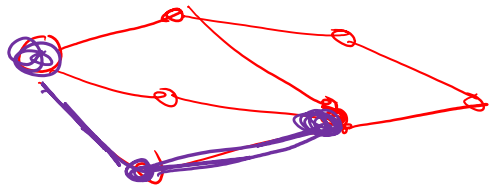


Backtracking

BACK TRACKING

- ✓ **Backtracking** is a general algorithm for finding all (or some) solutions to some computational problem, that *incrementally builds candidates to the solutions*, and abandons each partial candidate '*c*' ("backtracks") as soon as it determines that '*c*' cannot possibly be completed to a valid solution.

- ✓ Backtracking is an important tool for solving constraint satisfaction problems, such as *crosswords*, *verbal arithmetic*, *Sudoku*, *and many other puzzles*.

- ✓ It is also the basis of the so-called logic programming languages such as *Planner and Prolog*.
- ✓ The term "backtrack" was coined by American mathematician D. H. Lehmer in the 1950s.
- ✓ The pioneer string-processing language SNOBOL (1962) may have been the first to provide a built-in general backtracking facility.

Backtracking

Backtracking is one of the techniques that can be used to solve the problem.

It uses the Brute force search to solve the problem, and the brute force search says that for the given problem,

Backtracking is an algorithm that finds all the possible solutions and selects the desired solution from the given set of solutions.

This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems. In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.

Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

When to use a Backtracking algorithm?

When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:

- A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions.
- Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.

Applications of Backtracking

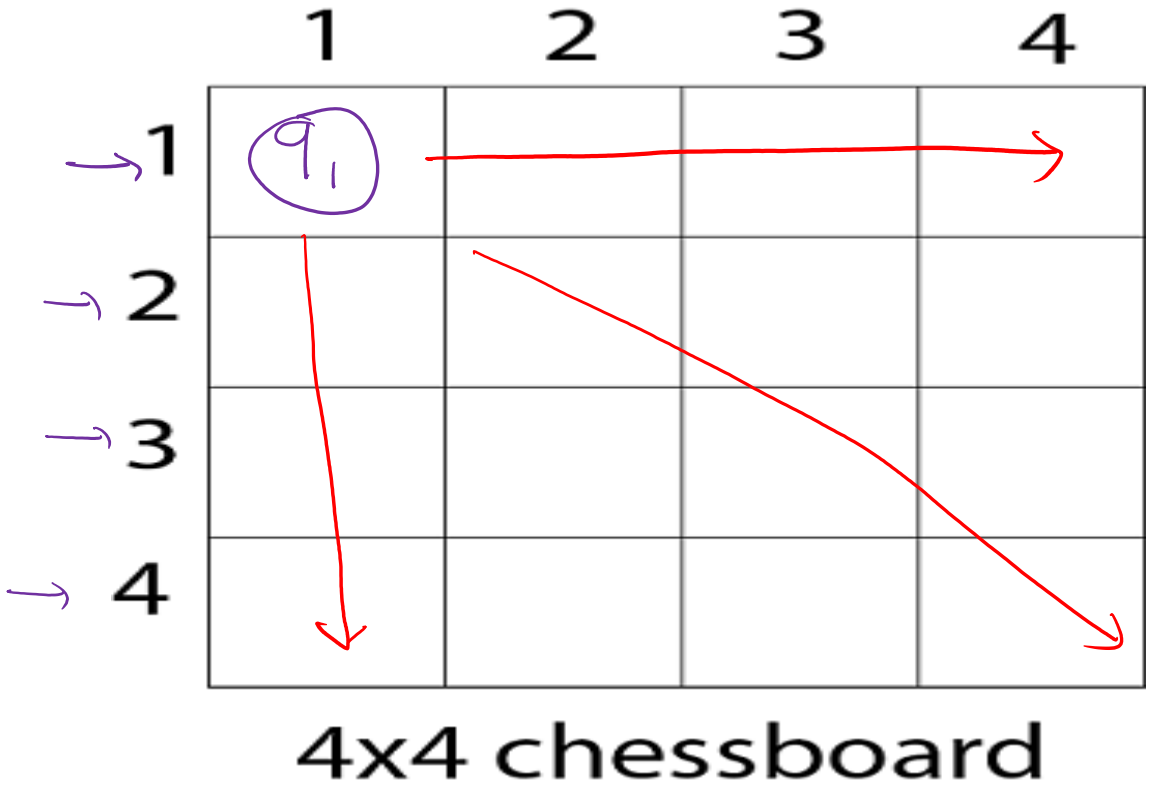
- N-queen problem
- Sum of subset problem
- Graph coloring
- Hamiliton cycle

N-Queens Problem

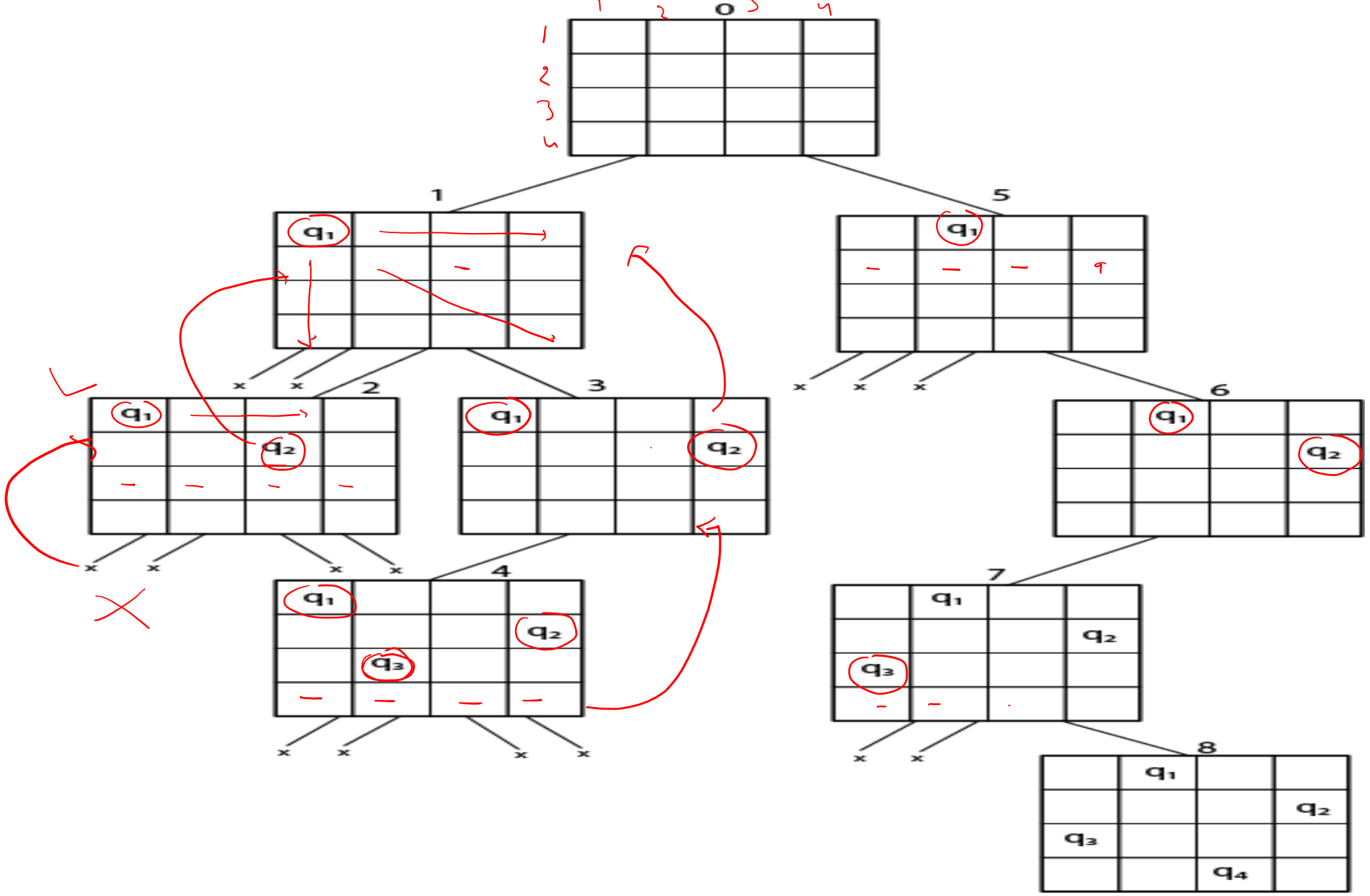
n x n

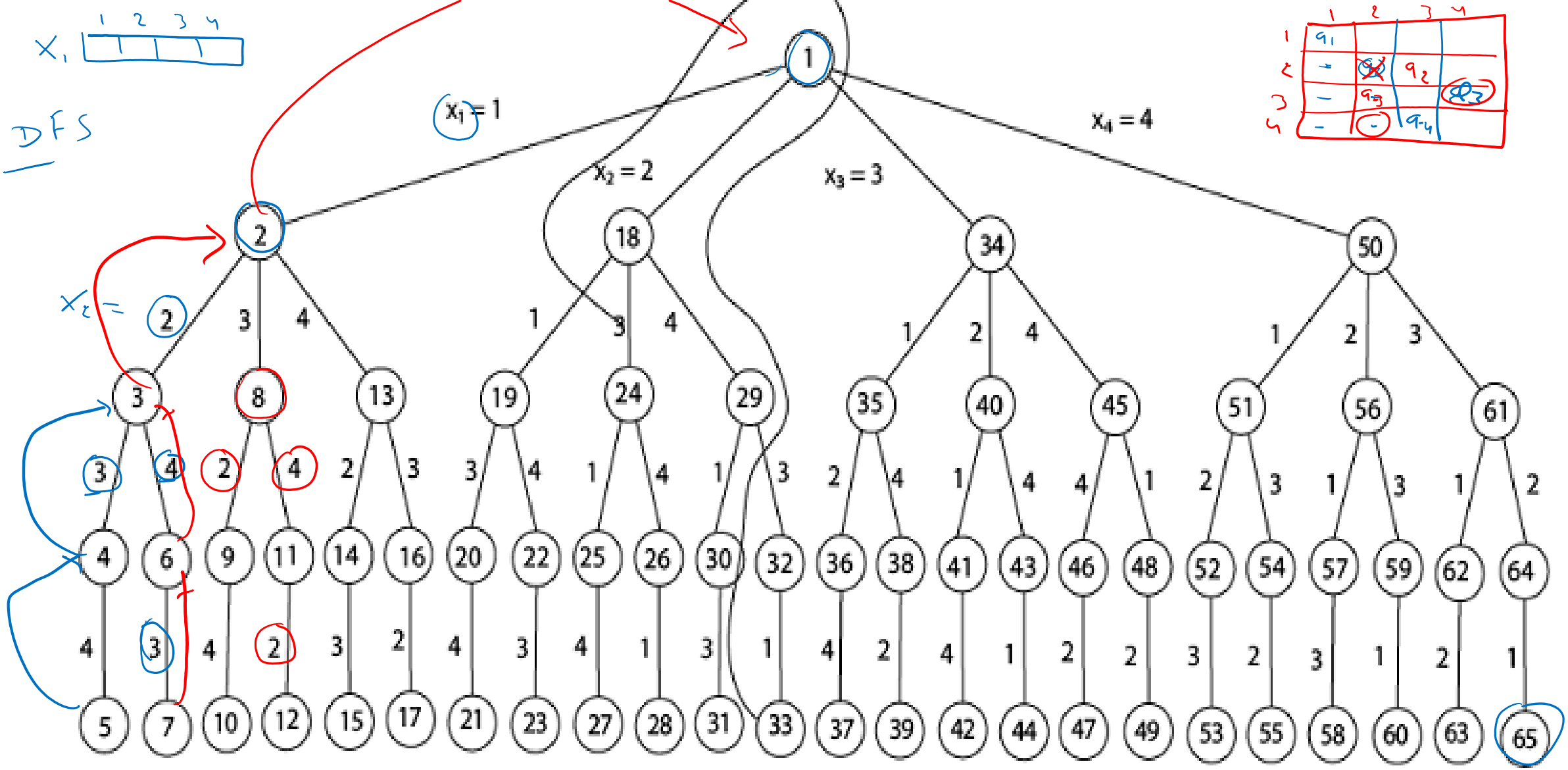
N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

4- queen



4x4 chessboard

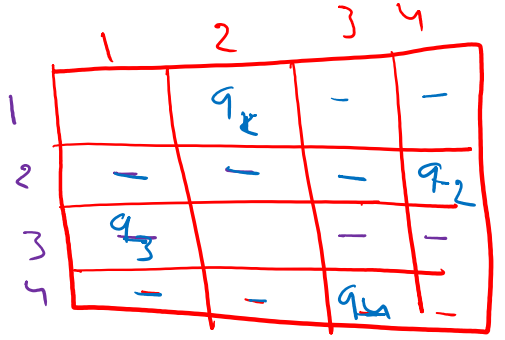

0
1
5
2
3
6
4
7
8
q1
q2
q3
q4
x



4 - Queens solution space with nodes numbered in DFS

STEPS REVISITED - BACKTRACKING

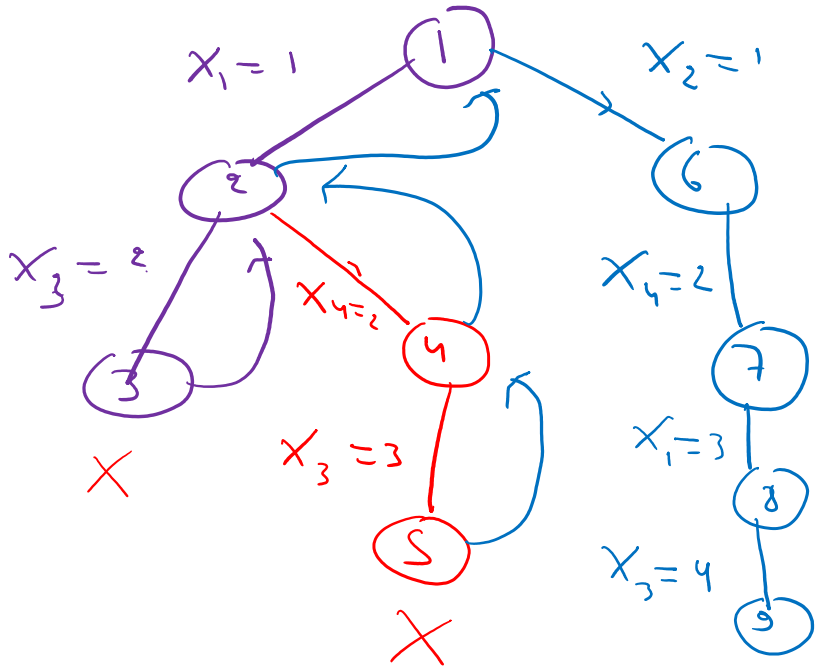
1. Place the first queen in the left upper corner of the table.
2. Save the attacked positions.
3. Move to the next queen (which can only be placed to the next line).
4. Search for a valid position. If there is one go to step 8.
5. There is not a valid position for the queen. Delete it (the x coordinate is 0).
6. Move to the previous queen.
7. Go to step 4.
8. Place it to the first valid position.
9. Save the attacked positions.
10. If the queen processed is the last stop otherwise go to step 3.



Algorithm for N queen problem:-

Initialize an empty chessboard of size NxN.

1) Start in the leftmost column
2) If all queens are placed
   return true
3) Try all rows in the current column.
   Do following for every tried row.
   a) If the queen can be placed safely in this row
      then mark this [row, column] as part of the
      solution and recursively check if placing
      queen here leads to a solution.
   b) If placing the queen in [row, column] leads to
      a solution then return true.
   c) If placing queen doesn't lead to a solution then
      unmark this [row, column] (Backtrack) and go to
      step (a) to try other rows.
4) If all rows have been tried and nothing worked,
   return false to trigger backtracking.



Subset Sum Problem

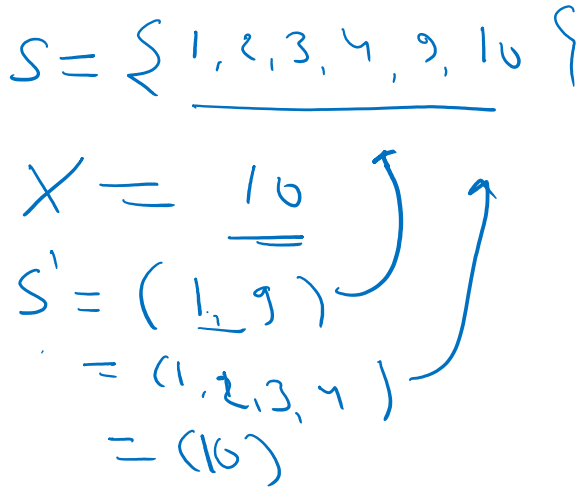
In the subset sum problem we have to find a subset s' of the given set S={S1,S2,S3……Sn} Where the elements of the set S are n Positive integers in such a manner that s' ∈ S and sum of the elements of s' is equal to some positive integer X.

This problem can be solved using following algorithms:

Recursive method

Backtracking

Dynamic Programing



It is one of the most important problems in complexity theory.

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values.

Given Set S= [3, 4, 5, 6] and sum = 9.Obtain the subset sum using Back Tracking approach.

D 187, 190, 194, 200, 204, 205, 211, 218, 222, 225, 236, 239,

I 485, 486, 501, 504, 505, 506, 507, 508, 510, 511, 513, 514, 515, 521, 524, 603, 607, 609, 612, 615, 616, 626, 629, 634, 636, 648,

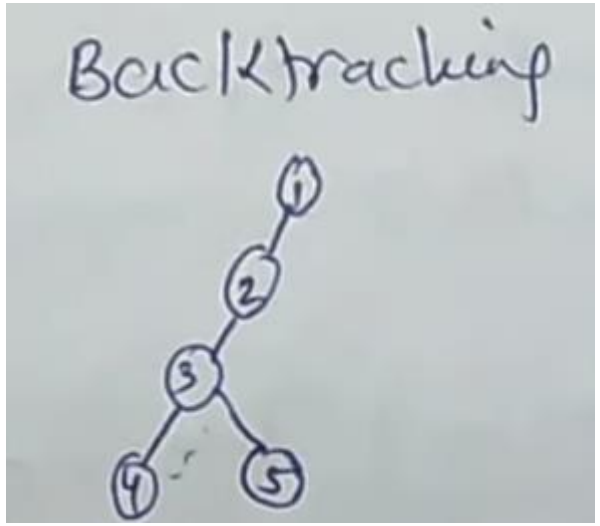
Given Set S= [3, 4, 5, 6] and sum = 9.Obtain the subset sum using Back Tracking approach.

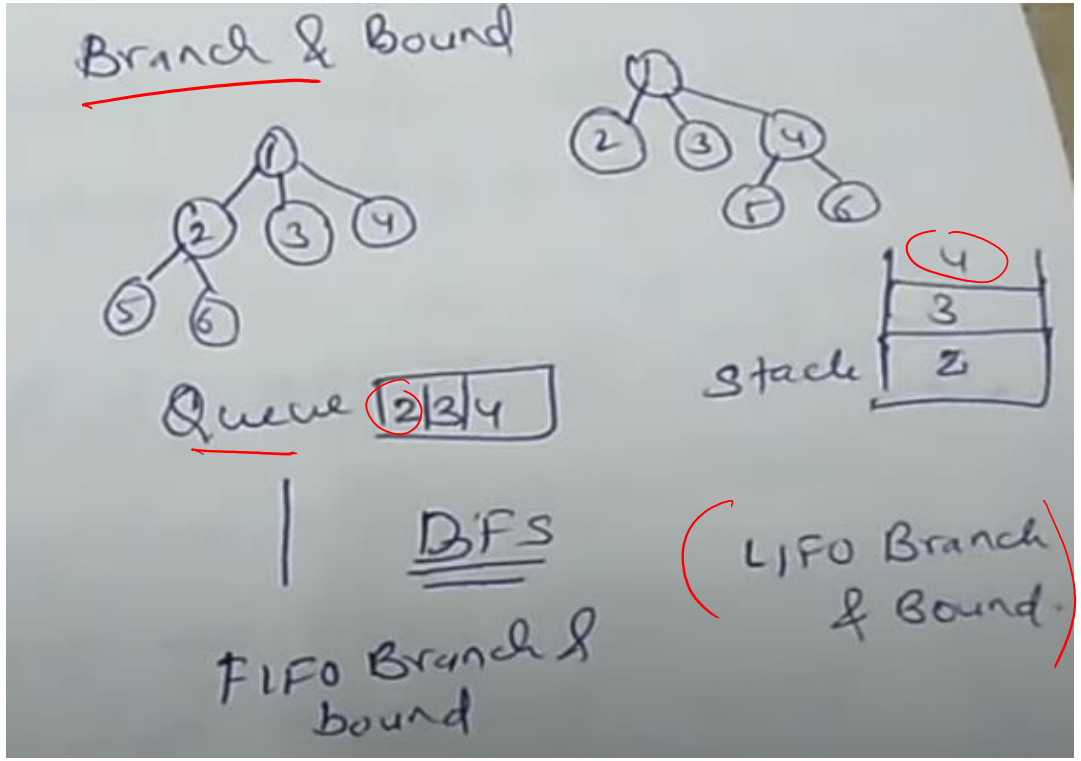
1. divide & conq
2. greedy
3. dynamic
4. B&C
5. B

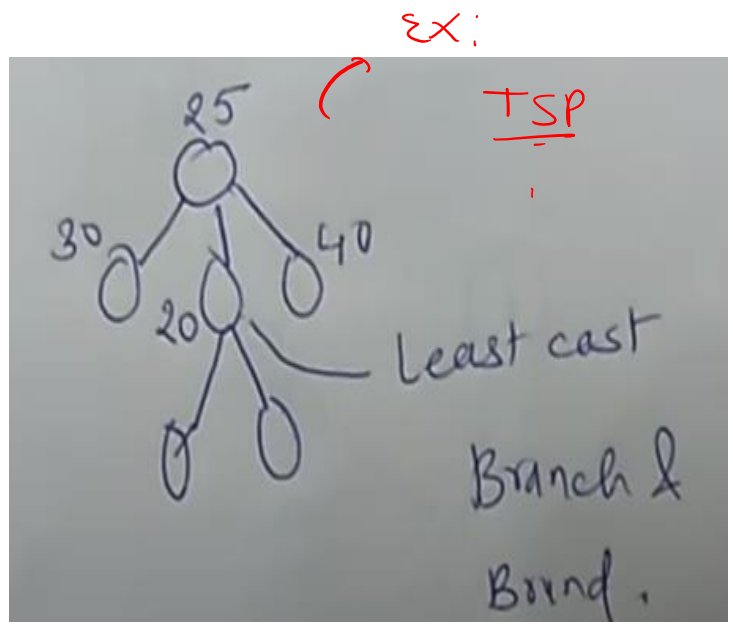
Branch and bound

Branch and bound algorithm is similar to backtracking but is used for solving the optimization problems (minimization).

It perform a graph traversal on the state space tree using BFS instead DFS which is used by backtracking.



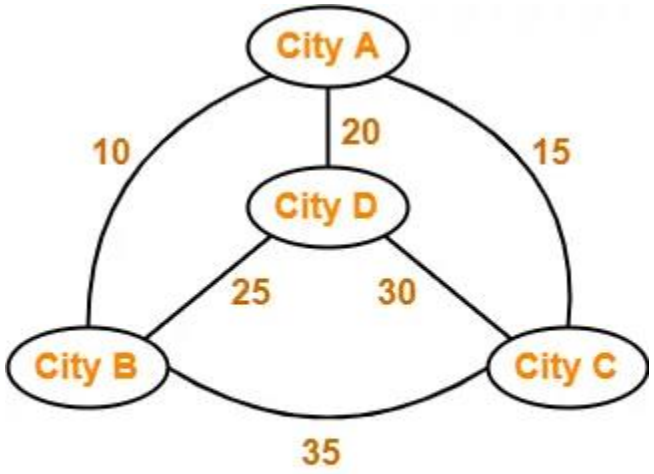




Travelling Salesman Problem | Branch & Bound

Travelling Salesman Problem states-

- A salesman has to visit every city exactly once.
- He has to come back to the city from where he starts his journey.
- What is the shortest possible route that the salesman must follow to complete his tour?



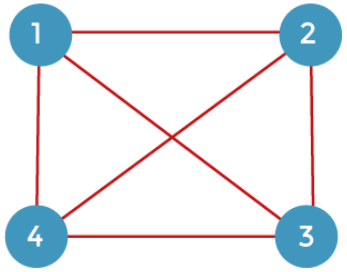
Travelling Salesman Problem

If salesman starting city is A, then a TSP tour in the graph is-

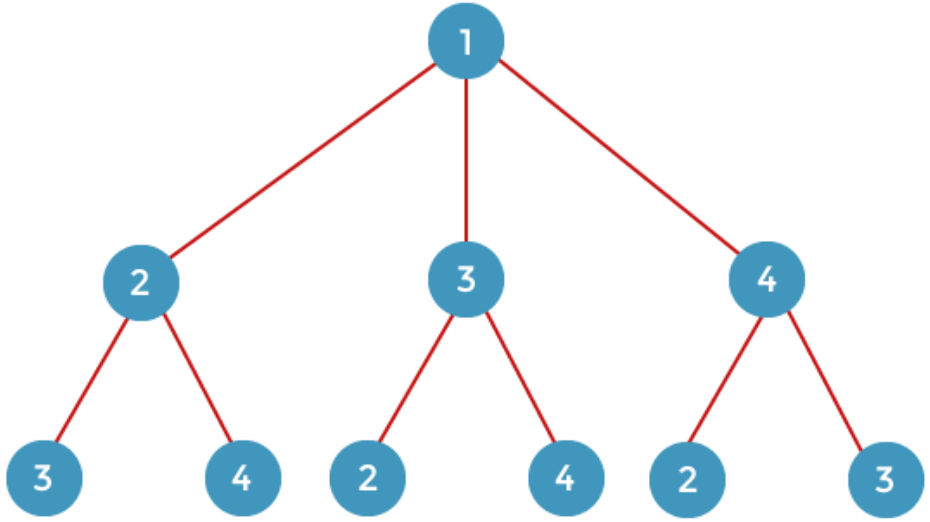
A → B → D → C → A

Cost of the tour

= 10 + 25 + 30 + 15

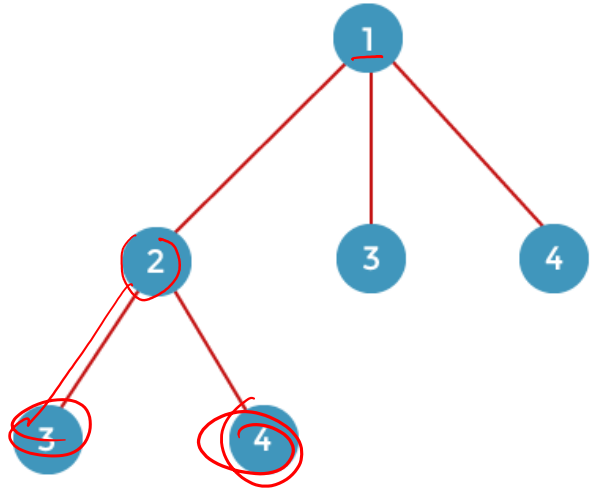


Graph

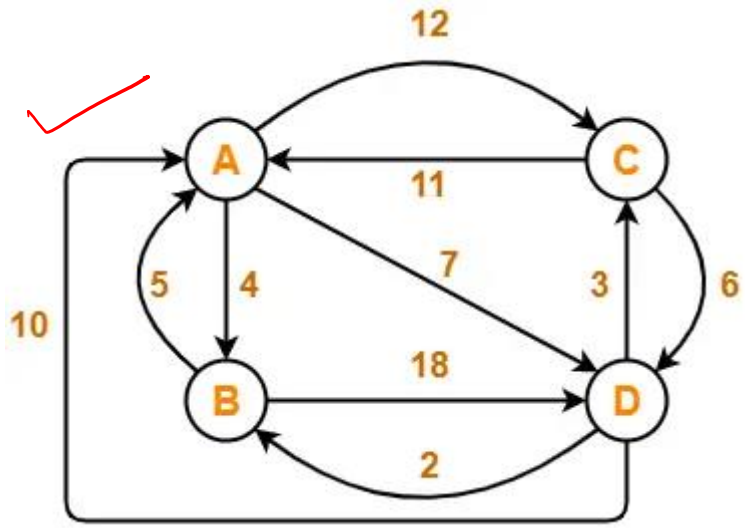


State Space Tree
(All Possible Path)

Branch & Bound (BFS)



Solve Travelling Salesman Problem using Branch and Bound Algorithm in the following graph

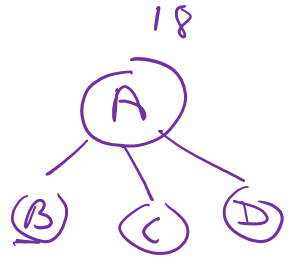


Reduced matrix

Row

	A	B	C	D	
A	∞	4	12	7	4
B	5	∞	∞	18	5
C	11	∞	∞	6	6
D	10	2	3	∞	2

total = 17



Row Reduction

	A	B	C	D	
A	∞	0	8	3	$R_1 - 4$
B	0	∞	∞	13	$R_2 - 5$
C	5	∞	∞	0	$R_3 - 0$
D	8	0	1	∞	$R_4 - 2$
	0	0	1	0	

Column Reduction

Reduced matrix

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞
			$C_3 - 1$	

= C

Cost(A) = 17 + 1 = 18

Choosing To Go To Vertex-B: Node-2 (Path A → B)

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

(1) Set Row A → ∞
(2) Set column B → ∞
(3) Set cost(B, A) → ∞

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	13
C	5	∞	∞	0
D	8	∞	0	∞

13

Row Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	5	∞	∞	0
D	8	∞	0	∞

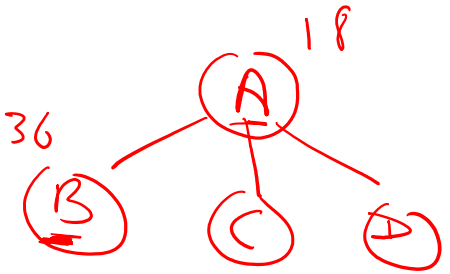
$R_2 - 13$

5

Column Reduction-

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	0	∞	∞	0
D	3	∞	0	∞

$R_1 - 5$



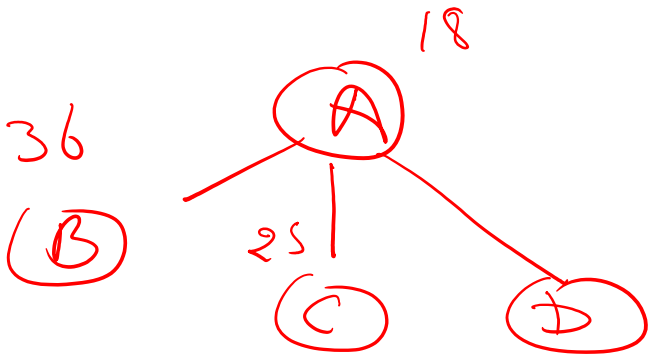
Cost(B)
= Cost(A) + Sum of reduction elements + C[A,B]
= 18 + (13 + 5) + 0 = 36

Choosing To Go To Vertex-C: Node-3 (Path A → C)

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

1. Set Row A to ∞
2. set column C to ∞
3. set (C, A) to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞



Cost(C)
= Cost(A) + Sum of reduction elements + C[A,C]
= 18 + 0 + 7
= 25

Choosing To Go To Vertex-D: Node-4 (Path A → D)

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

(1) set A to ∞

(2) set row D to ∞

(3) set C(D,A) to ∞

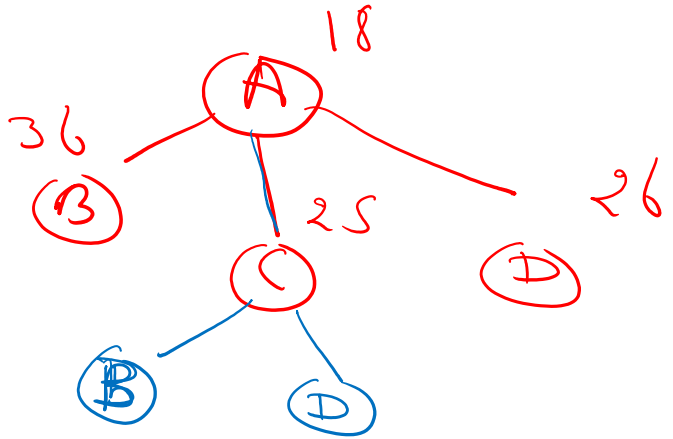
	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	5	∞	∞	∞
D	∞	0	0	∞

5

Row Reduction

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	0	∞	∞	∞
D	∞	0	0	∞

$R_3 - 5$



Cost(D)
= Cost(A) + Sum of reduction elements + C[A,D]
= 18 + 5 + 3
= 26

Choosing To Go To Vertex-B: Node-5 (Path A → C → B)

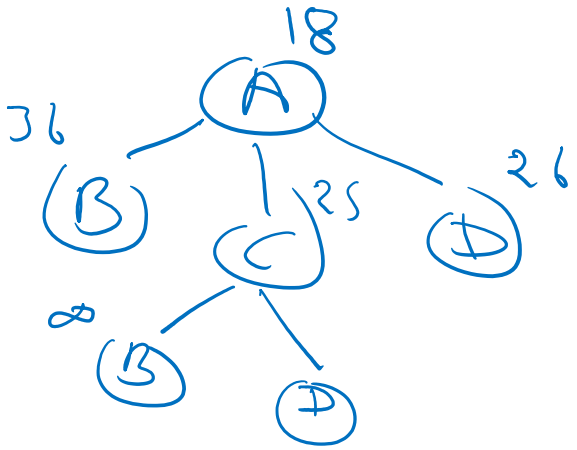
	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

(1) Set Row C to ∞
(2) Col^n B to ∞
(3) Set C(B,A) to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	13
C	∞	∞	∞	∞
D	8	∞	∞	∞

Row Reduction-

	A	B	C	D	
A	∞	∞	∞	∞	
B	∞	∞	∞	0	$R_2 - 13$
C	∞	∞	∞	∞	
D	0	∞	∞	∞	$R_4 - 8$



Cost(B)
= cost(C) + Sum of reduction elements + C[C,B]
= 25 + (13 + 8) + ∞
= ∞

Choosing To Go To Vertex-D: Node-6 (Path A → C → D)

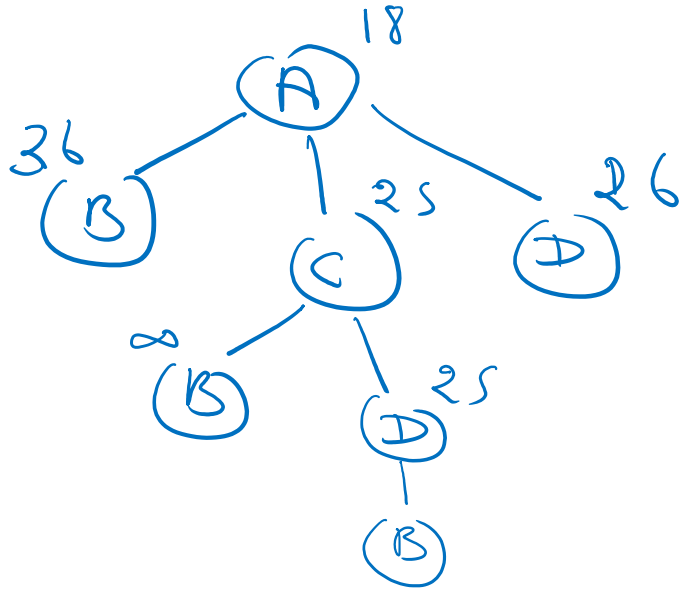
	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

(1) Set Row C to ∞
(2) Set col^n D to ∞
(3) Set C(D, A) to ∞

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	∞	0	∞	∞

Reduced matrix

Cost(D)
= cost(C) + Sum of reduction elements + C[C,D]
= 25 + 0 + 0
= 25



Cost(D) = 25

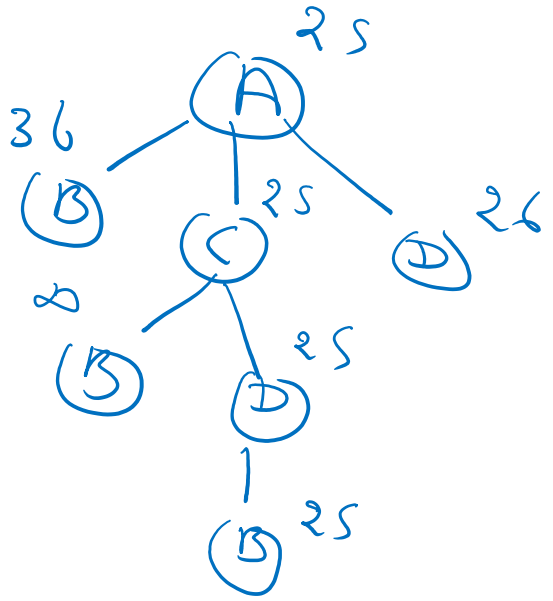
Choosing To Go To Vertex-B: Node-7 (Path A → C → D → B)

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	∞	0	∞	∞

(1) set Row D to ∞
(2) set col^h B to ∞
(3) set C(B, A) to ∞

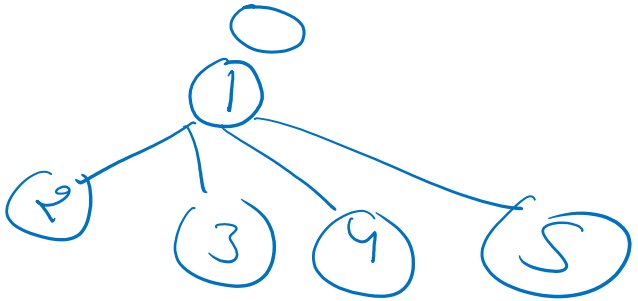
	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	∞
C	∞	∞	∞	∞
D	∞	∞	∞	∞

Cost(B)
= cost(D) + Sum of reduction elements + C[D,B]
= 25 + 0 + 0
= 25



Thus,
•Optimal path is: **A → C → D → B → A**
•Cost of Optimal path = **25 units**

	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	30	10	11
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞



Problem

Polynomial

A problem is said to be polynomial if there exists an algorithm that solves the problem in time $T(n) = O(n^c)$, where c is constant

Example

- Sorting : $O(n \log n) = O(n^2)$
- Searching $O(n)$

Exponential

A problem is said to be exponential if no polynomial time algorithm can be developed for it or algorithm solves the problem in time $T(n) = O(2^n)$ or $O(k^n)$ k-constant, n

Example-

- 0/1 knapsack problem
- Travelling Salesman

P

P Class Problem

Solution and verification is done in polynomial time)

NP Class Problem

Solution is done in Exponential time
Verification is done in polynomial time)

n = 8
b = 2x
b = 2x 8
b = 16

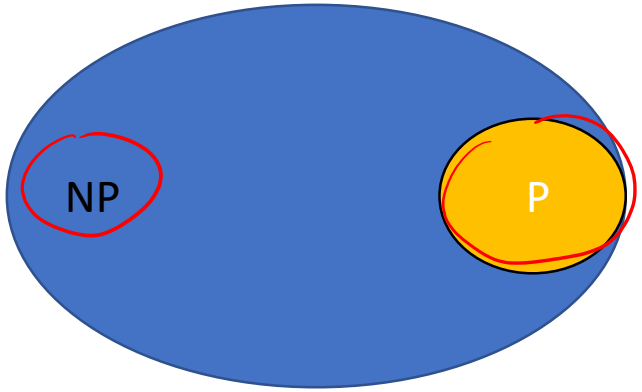
P Class

A Problem that can be solved in Polynomial time is known as P Class Algorithm.

Example : Insertion Sort ($O(n^2)$),
Merge Sort (O(nlogn)),
Sequential Search (O(n)) , etc.

O D+M

1) P ⊆ NP



NP Class

The NP in NP class stands for Non-deterministic Polynomial Time. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.

Problems of NP can be verified by a Turing machine in polynomial time.

Example: **Hamiltonian Path Problem.**

- **Graph colouring**
- **Suduko Game**

Reduction

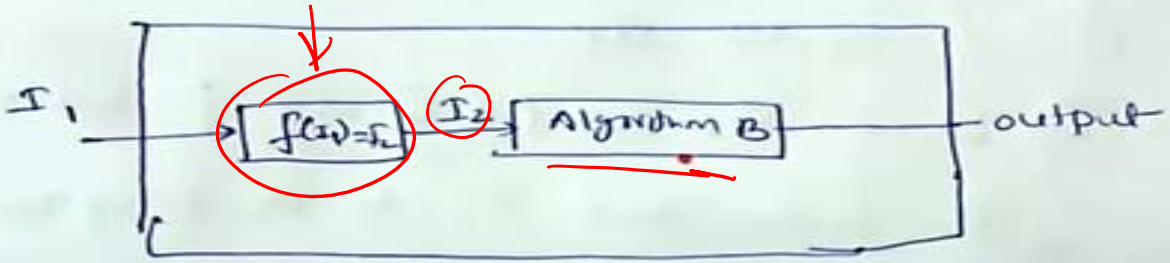
Let we have 2 decision problem P1 and P2

Yes
No

P_1 Problem [Input I_1
Algorithm A(?)

P_2 problem [Input I_2
Algorithm B
(Written)

Suppose Algorithm B can be used to Solve problem P_1



Problem P_1 is reducible to problem P_2, if there is function which convert input (instance) of A into Input (instance) of B & Solution of that instance provide solution to problem.

$P_1 \propto P_2$

Here we assume conversion cost is also in Polynomial time

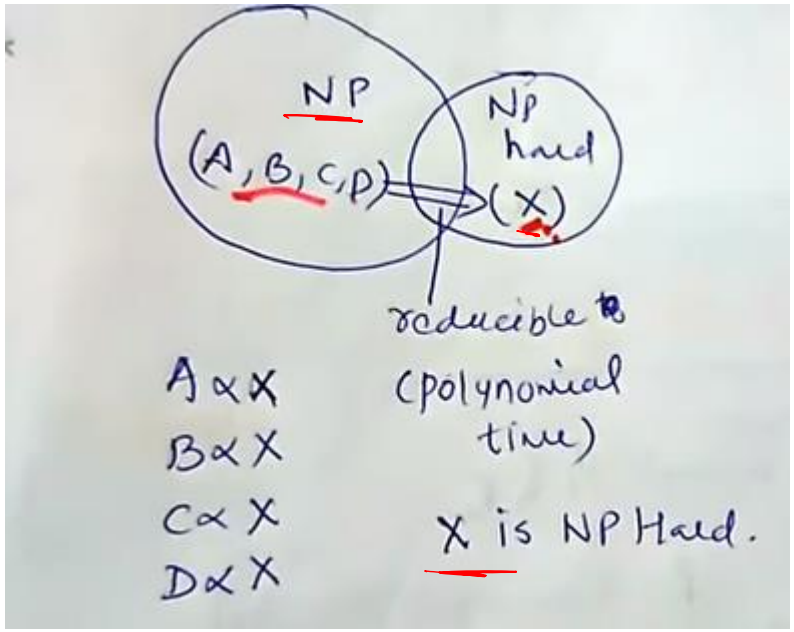
NP-hard class

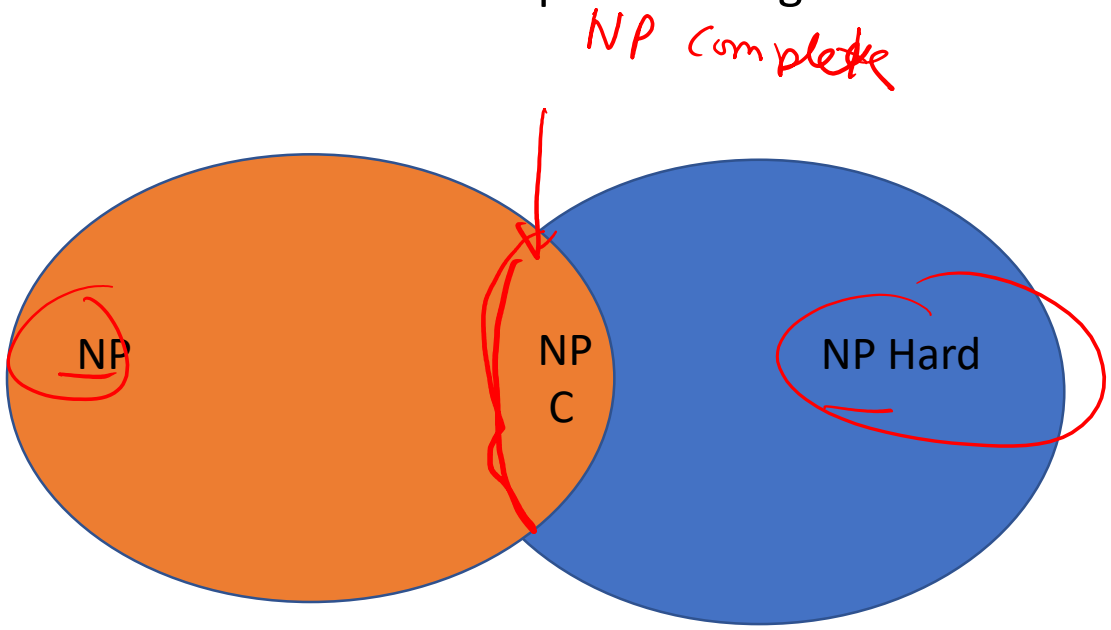
An NP-hard problem is at least as hard as the hardest problem in NP and it is a class of problems such that every problem in NP reduces to NP-hard.

Features:

All NP-hard problems are not in NP.

It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.





NP-complete class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

- Easy $\rightarrow \mathcal{P}$
- Medium $\rightarrow \mathcal{NP}$
- Hard $\rightarrow \mathcal{NP}\text{-}Complete$
- Hardest $\rightarrow \mathcal{NP}\text{-}Hard$