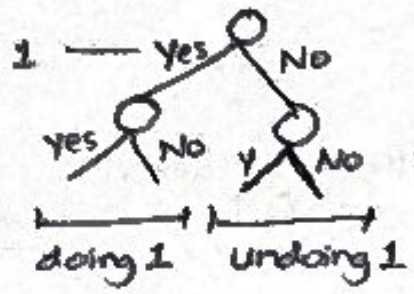
⊙ **Backtracking** : programming Approach where solution is found by exploring all possible solutions in a systematic way.

→ Builds solutions incrementally and discards those that fail to meet the constraints.

→ Thus it backtracks to other solution cases when it finds a dead-end.

⊙ It often uses recursion to explore all solutions simultaneously. Incrementally built solutions and undoing built solutions for other partial solutions.



1) Choose a decision (Yes/No)
2) Explore that decision
3) Undo it (No/Yes)
4) And explore this decision
5) Back track to previous decision if solution doesn't exist.
6) Follow these steps each time a decision is made.
7) Terminate the program when all cases are explored OR solution is found.

---

constraints for solving - Backtracking

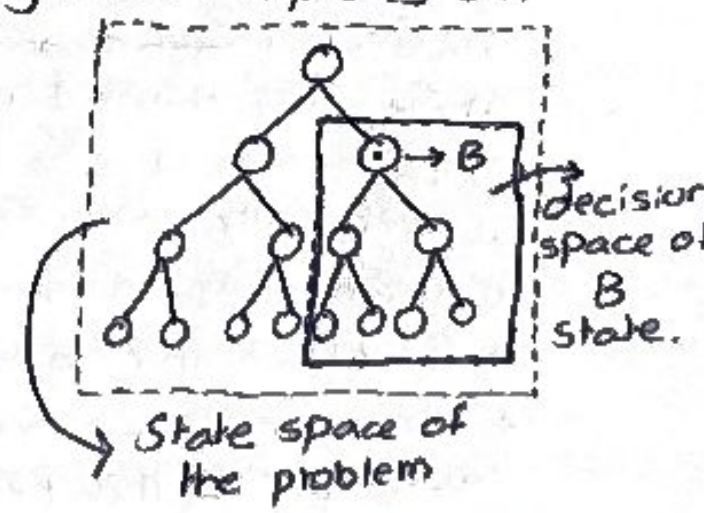
1) Bound constraints (upper or lower values for decision variables)
2) Feasiblity constraints (check wheter solution meets given criteria)
3) optimization constraints (The best solution is only accepted)

Decision space exploration : navigating through set of all possible decisions/moves to find a solution.

→ critical part of Back Tracking as algorithm explores all decision paths, discarding some.

State space : graphical representation of all possible options at each step while solving a problem

Node = state of solution
Branch = decision



Recursive Backtracking Algorithm


```
Backtracking (current_state):
    if is_solution(current_state)
        store_solution(current_state)
        return

    for each choice in available_choices(current_state):
        if is_valid(choice):
            apply_choice(choice, current_state)
            Backtracking(current_state)
            remove_choice(choice, current_state)
```

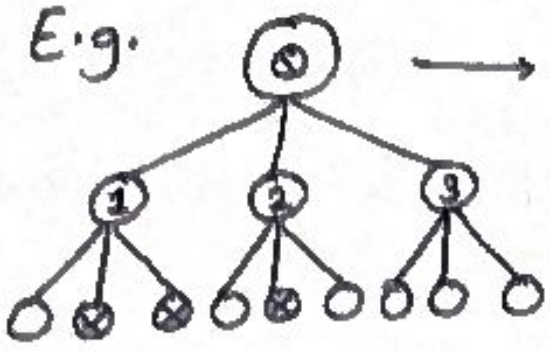
→ Base case

→ Exploring Further

→ Backtracking

1. Start with base/first choice
2. If that choice is enough for being a solution, store it
3. For each next choice we will take from this state, check if it can be considered valid
4. For each valid option, add this choice to current_state making a new state.
5. Recursively call the same function on the new state to explore that state. [If during exploration, new solutions are found, they are added to solution list, if not function returns after exploring all possible cases from that new state]
6. Now, we <u>BACKTRACK</u> by undoing the choice and returning back to current_state! Hence, we are ready to explore next choice.

E.g.



→ current state

→ All valid available choices we can make

→ solutions from them, only some are valid.

Happens Recursively i.e. D.F.S is used

⊗ = solution

- ⊙ we start from ⊙ and find valid choices 1, 2, 3
- ⊙ we apply choice 1 making current state = 1
- ⊙ we explore 1 and find 2 solutions, add it to list of solution
- ⊙ After fully exploring 1, we backtrack to previous state ⊙
- ⊙ Now move to state 2, Find 1 solution, add it to list
- ⊙ Backtrack and go to 3, find no solutions, back track to ⊙
- ⊙ Now since we have no further choices to explore <u>from ⊙</u> we end so the execution of Backtrack(⊙)

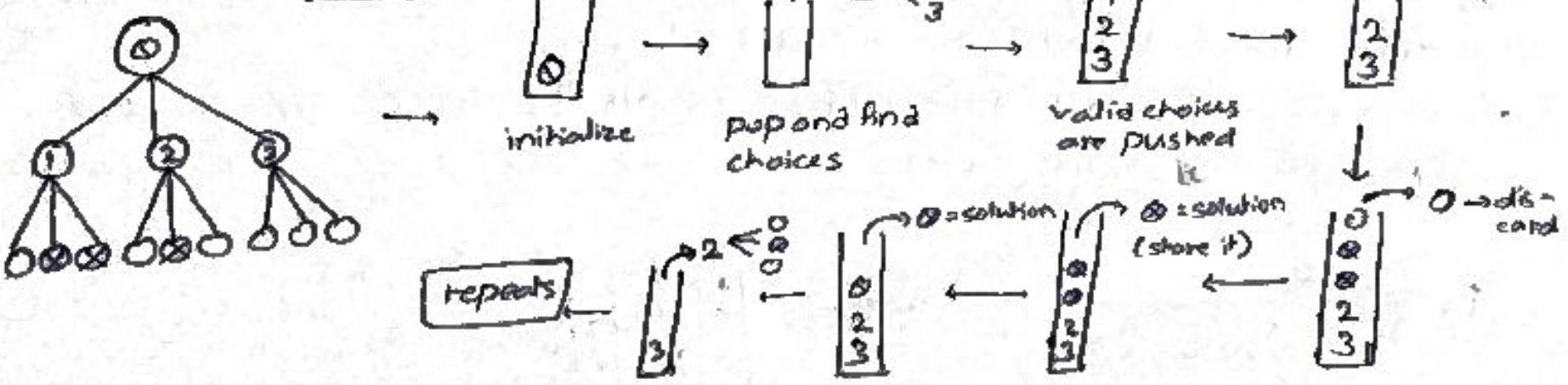
Iterative Backtracking Algorithm

```
Backtracking():
    Stack = new Stack()
    Stack.push(initial_state)

    while(!stack.isempty()):
        current_state = stack.pop()
        if is_solution(current_state):
            store_solution(current_state)
            continue

        choices = generate.choice(current_state)
        for i in choices:
            if is_valid(i):
                new_state = apply_choice(i, current_state)
                stack.push(new_state)
```

1. same exploration is used as recursive algorithm.
2. use of stack instead of recursive calls.
3. Also, a DFS approach.



Recursive B.T.

- → Recursive calls
- → automatic manages states
- → easier to implement
- → Hard to customize/optimize
- → can suffer stack overflow

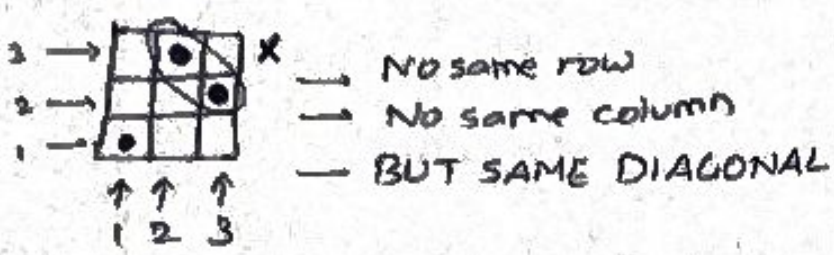
Iterative B.T.

- → Iteratively over an explicit stack
- → manual management required.
- → Hard to implement (code)
- → Easy to customize/optimize
- → No stack overflow as uses its own stack.

(*) N-Queens Problem

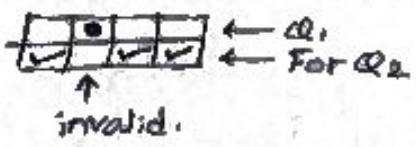
→ placing N queens on a NxN chessboard such that no two queens can attack each other.

i.e. no two queens share a row, a column or a diagonal.

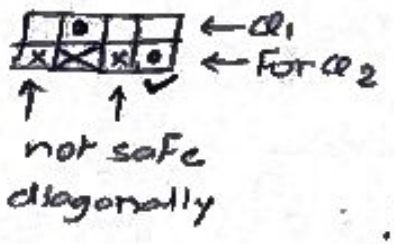


2×2 and 3×3 queens problem has no solution

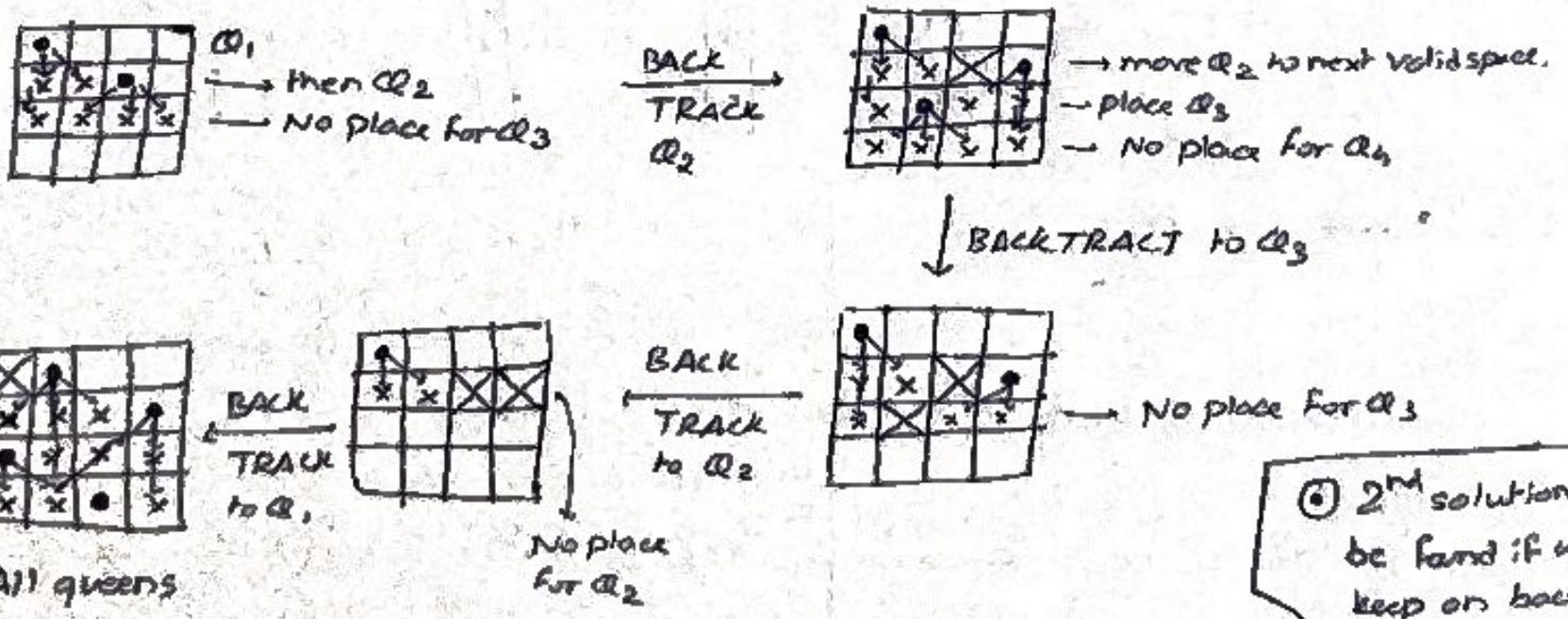
1. Start with empty NxN chessboard
2. Place queens one by one in different rows ( )
   For each row, place a queen such that no previously placed queen shares a column with it.



3. For each placement, check whether it is safe to place it there.
   i.e. check if no previously placed Queen can → attack this queen diagonally



4. If you can place the queen, place it and move to place next queen (in next row)
5. If all queens are placed, store solution
6. If queen cannot be placed, move to previous queen (BACKTRACK) and replace it to the next valid place.
7. Repeat until all safe places of all the queens are checked.
   {Multiple Solutions Exist}

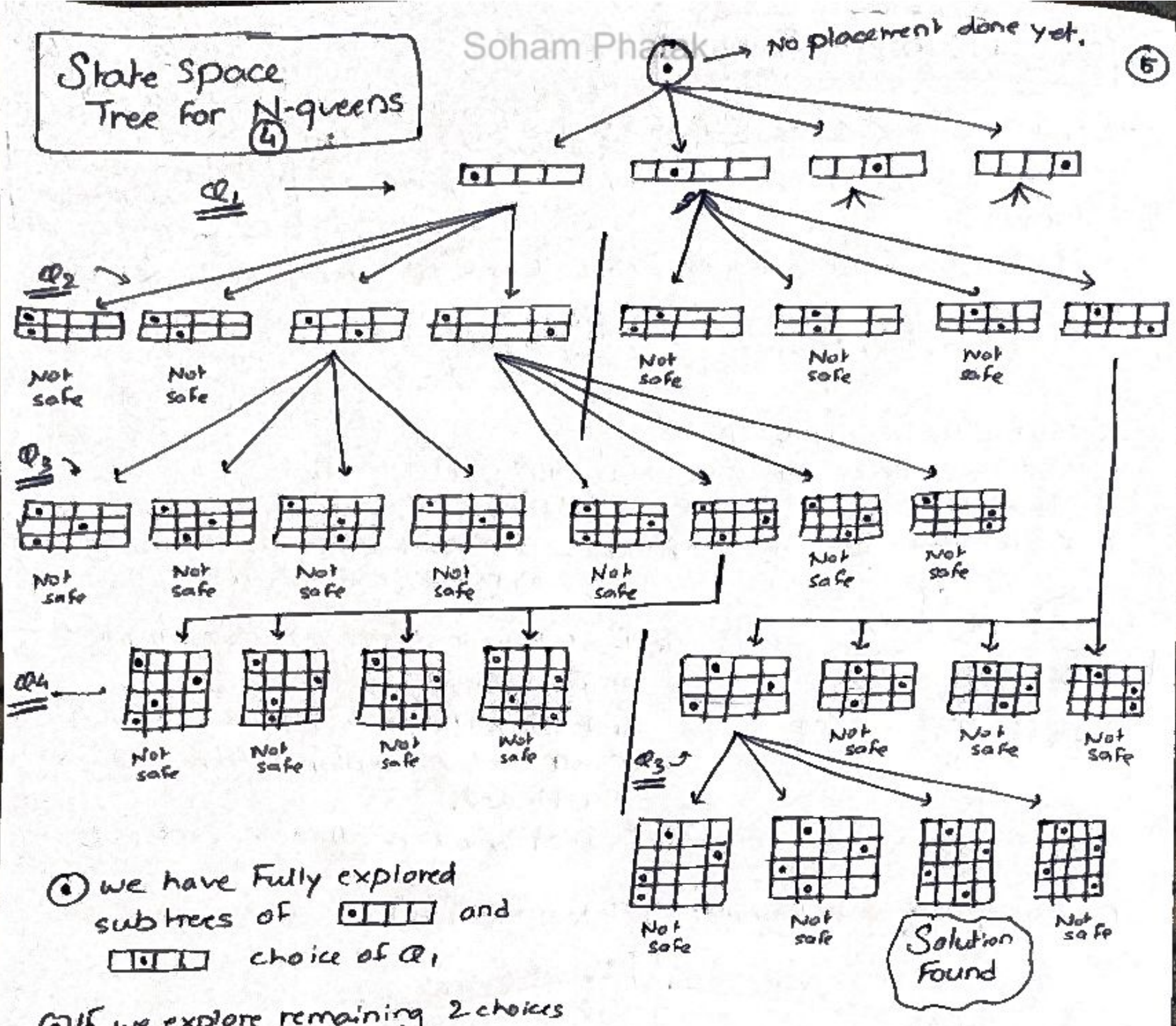


$\{Q_1, Q_2, Q_3, Q_4\} = \{2, 4, 1, 3\}$ ← solution 1 of 2 of 4 queens problem

↑ row number

⊙ 2nd solution can be found if we keep on backtracking and finding safe spaces

State Space Tree for N-queens (4)



- We have Fully explored subtrees of and choice of Q_1
- If we explore remaining 2 choices in same way, we get one more solution  ✓ solution 2
- Tree is pruned when state is (Not safe)

Time complexity : $O(N!)$ → all states are traversed.

Applications :
- → scheduling problems
- → constraint satisfaction problem
- → optimization problems
- → AI & search algorithms

(*) We, do not check same row condition in N-queens. This can lead to more complexity

✱ Sum of Subsets Problem

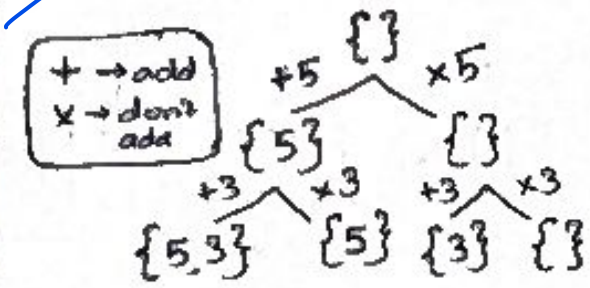
→ finding all subsets of a given set of numbers whose sum (of all elements) is equal to a given target.

— To Find solutions by exploring all options whose sum is ≤ target.

E.g. {5, 3, 10, 2, 4} Target = 12 → {5, 3, 4}, {10, 2} } sum of subsets is 12

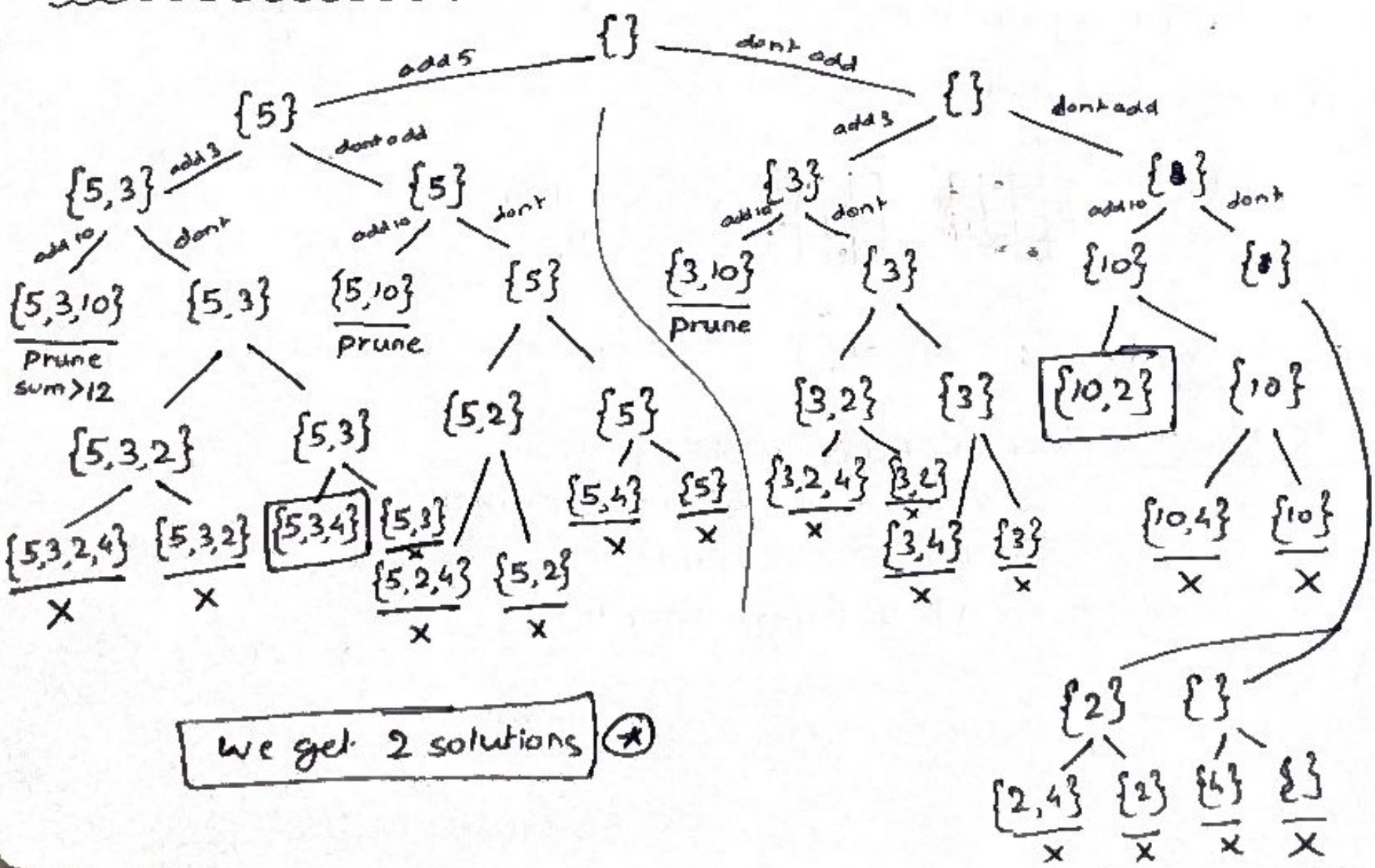
✱ 5 Queens Problem has (12) distinct solutions (mirrors included)

1. Start with an empty set
2. Sort the given set, this helps pruning efficiently
3. At each step — check if sum ≤ target, if not prune that brach.
4. Each state, we have 2 choices — 1) proceed with next no. in sequence
   2) proceed without it.



4. If at any instance, sum > target, stop exploring further, and BACKTRACK To previous state.
5. If all choices explored, then too Backtrack.
6. Exit only when all cases explored.

Proceeding with given Example: (State space Tree)



we get 2 solutions ✱

Time Complexity : $O(2^n)$ → there are 2^n subsets in total.

Applications :
- → Resource allocation
- → Knapsack problem
- → Scheduling & Task allocation
- → Subset selection (in ML)

⊛ Graph-Coloring Problem

(vertices of graph)

→ **m-coloring decision problem :** determine whether a given graph can be colored using m different colours such that no adjacent vertices have same colour.

→ **m-coloring optimization problem :** determine the min. no. of colours required to color a given graph such that no adjacent vertices have same colour.

---

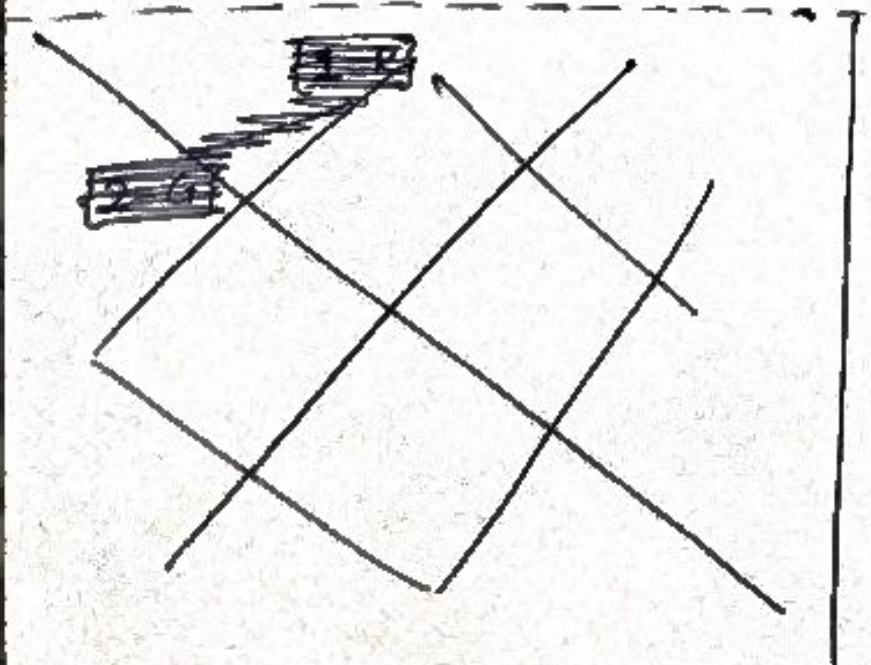
1. Given a graph $G(V,E)$ and a number m
2. assign colours to vertices one by one
3. keep in consideration that assigned colour should be different than the colour of adjacent vertices.
4. If possible, return solution
5. If not possible, backtrack to change colours
6. If not possible after trying all cases, return NO.

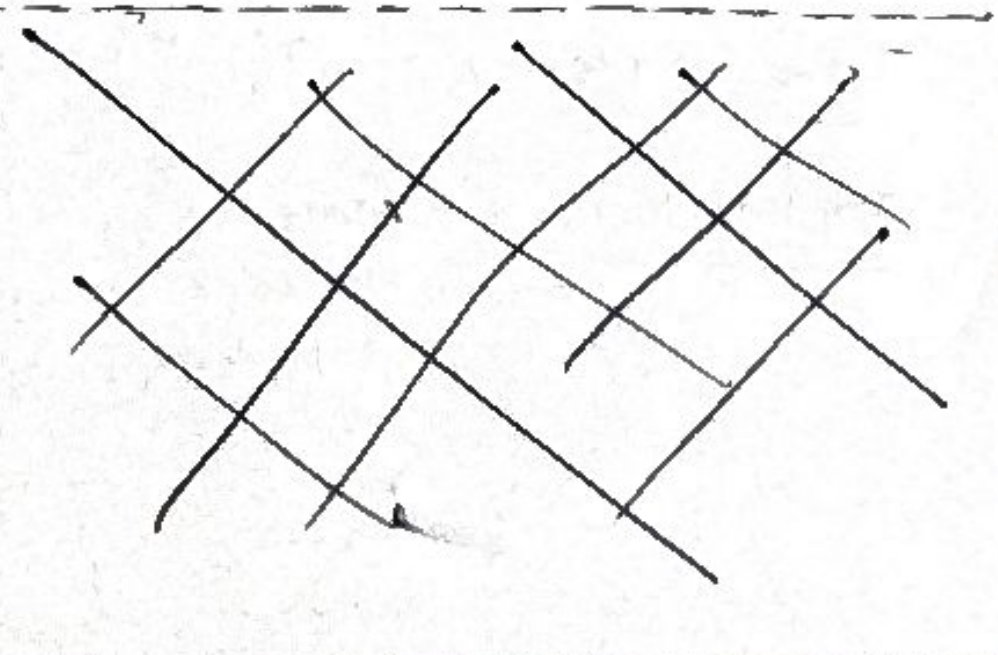
For m-coloring optimization, use N colours and return the no. of colors used.

N = no. of vertices

M-colorability can't be more than N for a N-vertex graph ⊛

E.g.





A,B,C,D = colours

{}

{1=A}

{1=A, 2=B}

{1=A, 2=B, 3=C}

{1=A, 2=B, 3=C, 4=D}

{1=A, 2=B, 3=C, 4=D, 5=A}

(*) remember, while colors, try to use already used colors to keep no. of used colors to minimum

Ans ⇒ m = 4

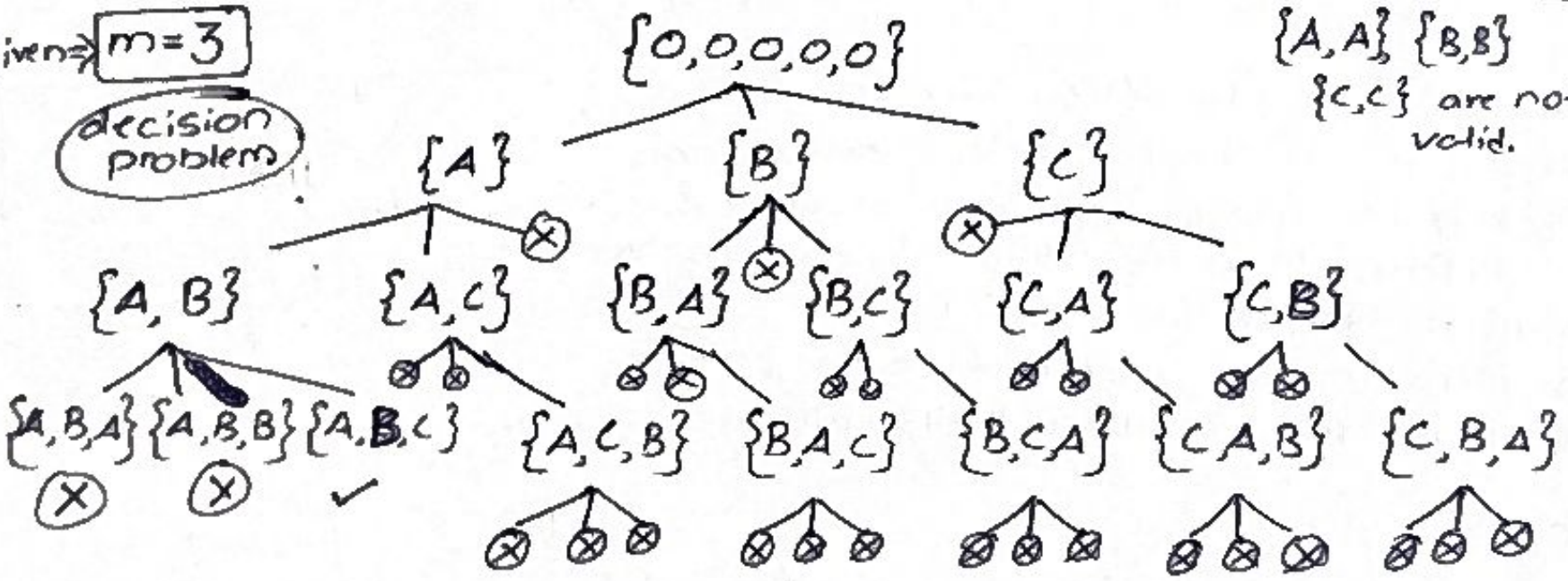
{optimization problem terminates when 1 solution is reached}

optimization problem.

---

given ⇒ m = 3

decision problem



{A, A} {B, B} {C, C} are not valid.

No solution.

Answer = NO

⊗ → pruned choices

Time Complexity = $O(m^N)$ m = no. of colors, n = no. of vertices

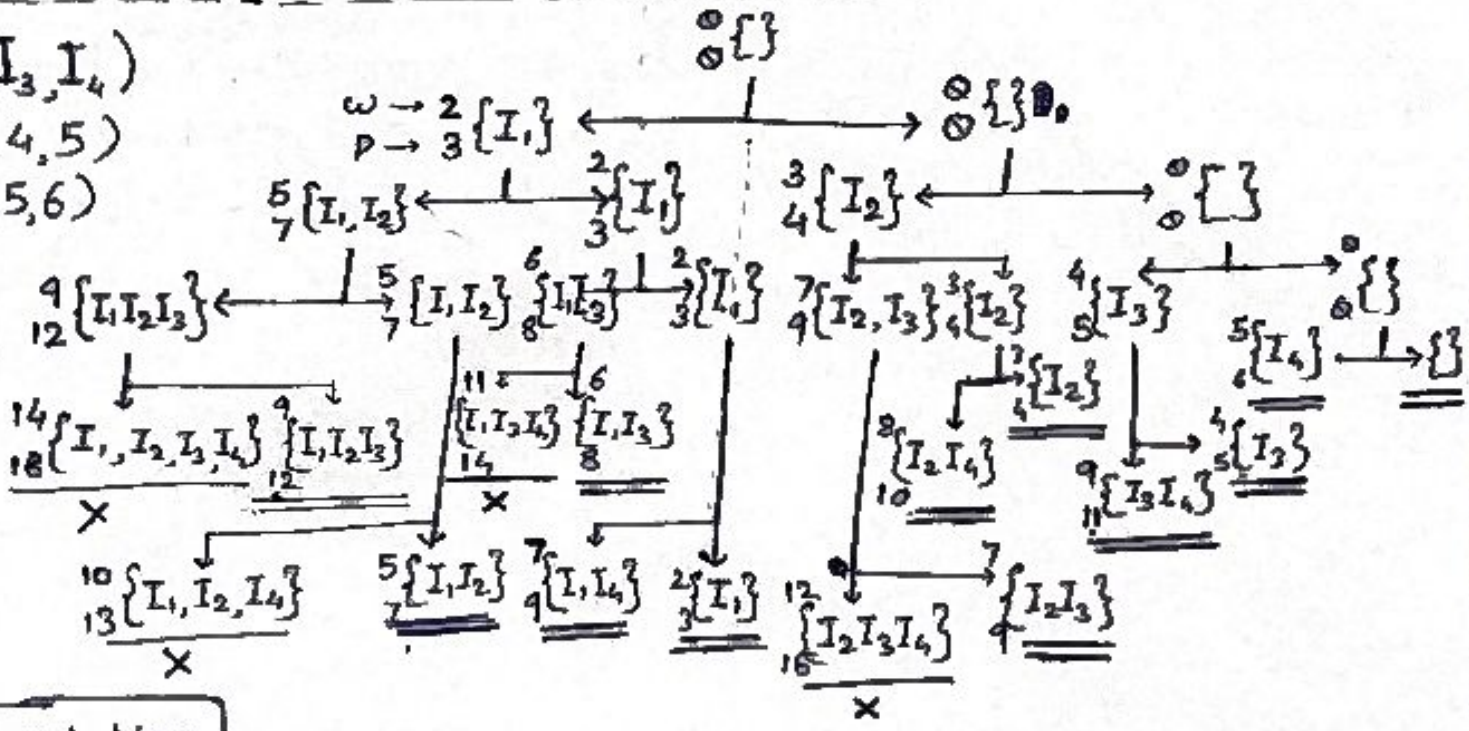
Applications = Map colouring
Network Design
Scheduling Problems.

⊛ 0/1 Knapsack Problem

→ select items in such a way that profit (total) is maximum and weight (total) ≤ capacity

1. Start with 1st item → (sort items according to weights)
2. a. include item and calculate weight, profit
   b. exclude item and calculate weight, profit
3. If including item causes total weight > capacity, stop exploring from that state
4. Continue 1,2,3 for all items from new states achieved.
5. Keep track of max profit & selected items.
6. After all cases are traversed, return the max. profit solution.

Items: (I_1, I_2, I_3, I_4)
weights: (2,3,4,5)
profits: (3,4,5,6)
capacity = 9



we have: 12 solutions

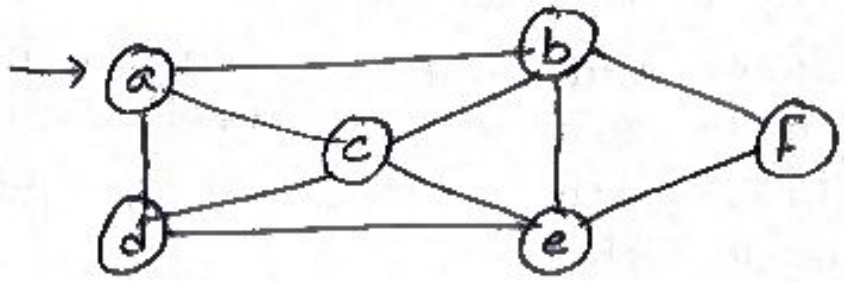
⊛ most optimum solution: $\{I_1, I_2, I_3\}$ < w = 9, p = 12

Time Complexity : $O(2^n)$

Applications : <Same as DP 0/1 knapsack>

Hamiltonian cycle : Visit every (exactly once) graph node and return back to starting node.

e.g.



Hamiltonian cycles

$a \to b \to f \to e \to c \to d \to a$
$a \to b \to f \to e \to d \to c \to a$
$a \to c \to b \to f \to e \to d \to a$
$a \to c \to d \to e \to f \to b \to a$
$a \to d \to c \to e \to f \to b \to a$
$a \to d \to e \to f \to b \to c \to a$