MCM : ① Catalan numbers. → Binary Trees | Ex. Catalan numbers

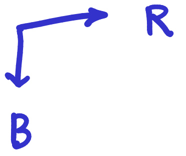
Template 4: DP on grid

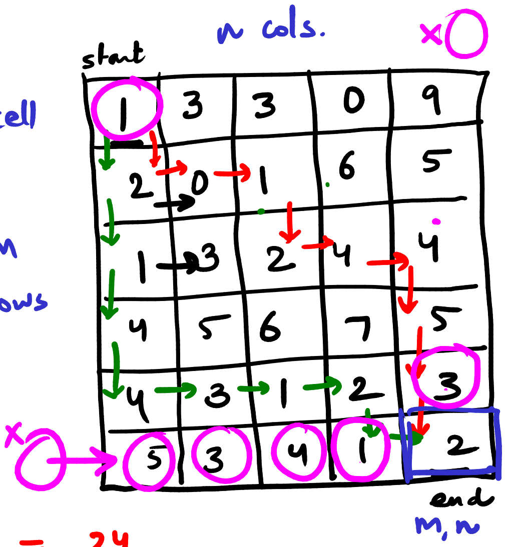
1. Fib. 2. Knapsack 3. LCS 4. MCM 5. grid

(0,0) → (m-1, n-1)
start   end

step on a cell
→ pay cost = value of cell

Move in 2 directions





1	3	3	0	9
2	0	1	6	5
1	3	2	4	4
4	5	6	7	5
4	3	1	2	3
5	3	4	1	2

min cost to go from start to end ?

1. Greedy ?   cost = 1+2+0+1+2+4+4+5+3+2 = 24

cost = 1+2+1+4+4+3+1+2+1+2 = 21 (better than greedy)

2. Recursion: (m,n) → (m-1,n) OR (m,n-1)

Recurrence ↓

$$cost(m,n) = grid[m][n] + \min(cost(m-1,n), cost(m,n-1))$$

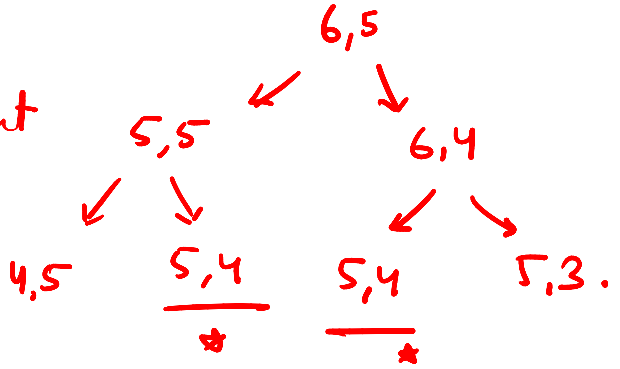
memo[m][n] = {-1}

DP:

```
int cost (grid, int m, int n) :
    if m<0 || n<0 : return 10e8 ;
    if m==0 and n==0 : return grid[0][0]
    if memo[m][n] != -1 : return memo[n][n]
    ~~return~~ grid[m][n] + min (cost(grid, m-1, n), cost(grid, m, n-1))
memo[m][n] =
    return memo[m][n] ;
```

cost (grid, row-1, col-1)

1. Before memoize, give it a 1s thought overlap is happening.



6 rows 5 col

variations : values.   -1 values which are blocked & you can't step on cells.

alphabets → min vowels.

more templates → No. Yes.

Idea : You go through temps & alot of variations → feel comfortable in topic.

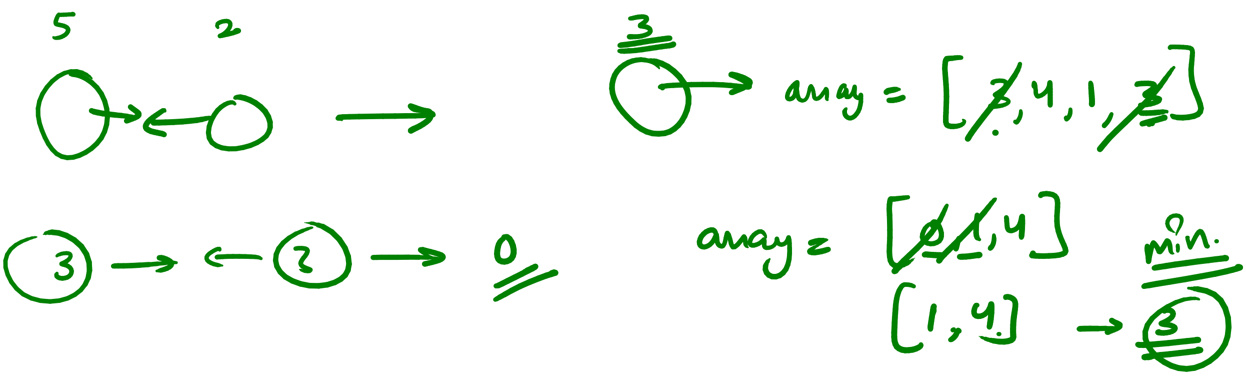
Random Question : 1) sense foresight → greedy x → recursion.

2) Try to find closest problem you've solved before.

Demo: array = [3, ~~5~~, 4, 1, ~~2~~] ← stones.

| leetcode stone crash

2 pick stones → crash them against each other.



sorting → greedy. → optimal x

(5 4 3)(1 2)

1 1

| 2 5 - 4,3,1 |

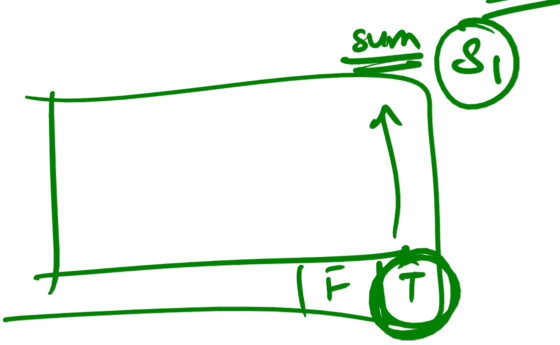
2

{+ve element subset} - 1 {+ve element subset2} = diff should be minimum.

MCM x LCS x Knapsack

arr

$|S_1 - S_2| = \text{diff} \downarrow$ min



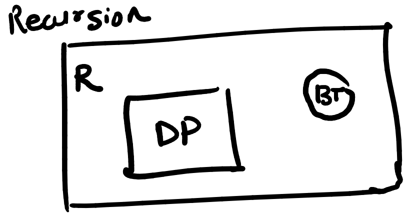
$|\text{sum(arr)} - 2(S_1)| = \text{diff}$

* DP concludes. —x—

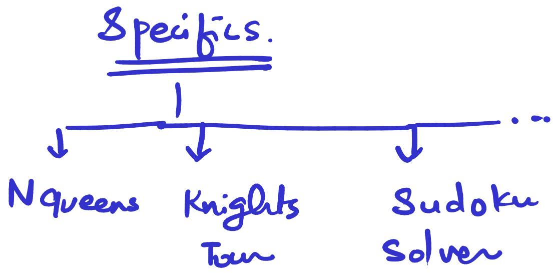
Set of problems → Recursion. | no memoization ×

Backtracking

all recursion calls are unique.



Generic sense.


Specifics.

Nqueens

Knights Tour

Sudoku solver

...

Template vs. Classic

↓ Variations

famous standard

CS concept of states | graphs.

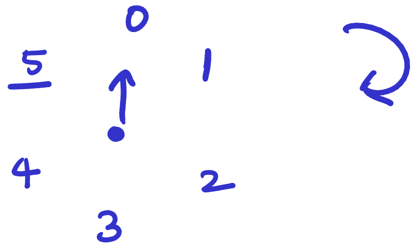
↓ all possible situations/states/values/positions in which your actor can be.

Eg. For switch regulator

$S_0\ S_1\ S_2\ S_3\ S_4\ S_5$

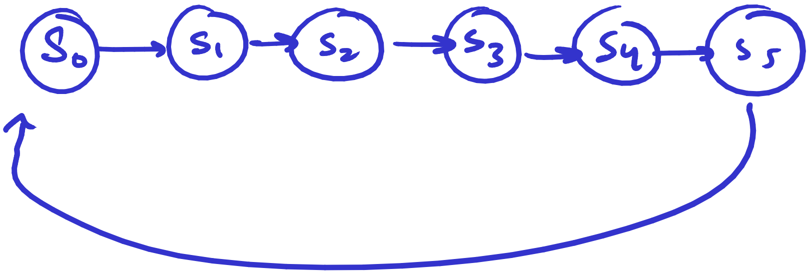
↓ ↓ ↓ ↓ ↓ ↓

↑ → actor states = {0, 1, 2, 3, 4, 5}



transitions = { how states are changing. 0→1, 1→2, 2→3, ... }

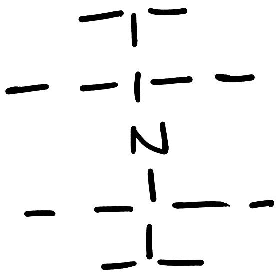
Diag. Automata diagram



state level possibilities. constraints ← understanding.

Eg. Knights Tour Problem. start → touch all cells of grid without stepping on any cell again.

Knight moves.



print matrix →

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

actor : knight.

chess board → 25 cells

1				
		11		
	12			

#states = 25

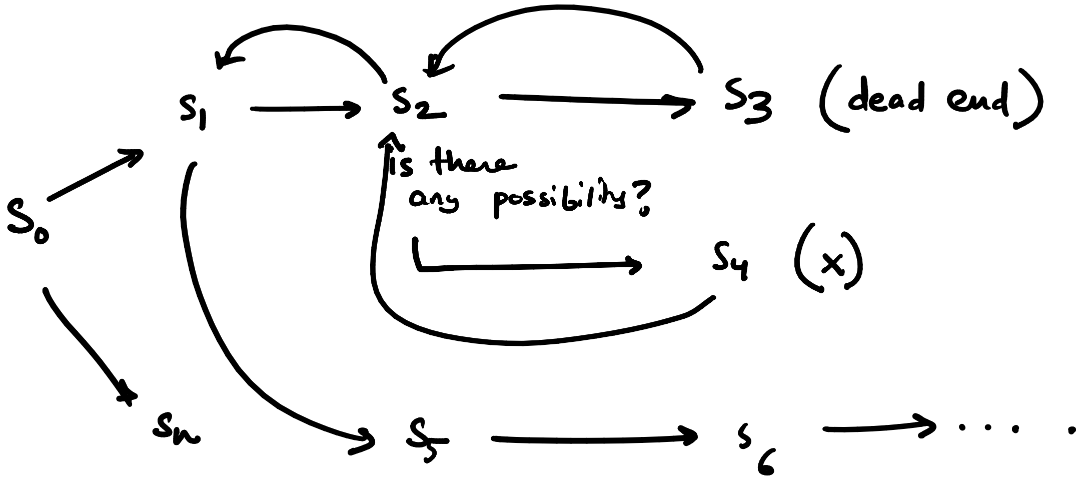
possibilities = 8

actual possibilities = 2

directions can a knight move? (general)

constraints : - cant go outside the board
- cant step on same cell twice

levels → steps you perform before you reach the solution.



25 steps

found sol^n

STOP.

Example 2 : N queens problems → N×N chessboard place N queens such that no queen attacks other queen.

N=1

Q

N = 2 no sol^n

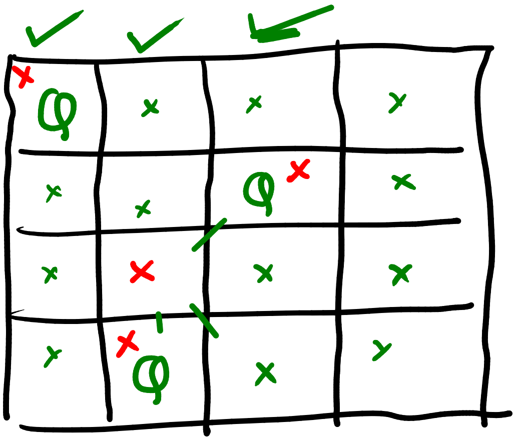
Q	x
x	x

N= 3

Q	x	x
x	x	x
x	Q	x

no sol^n

N=4.



If you put a queen in a col, you can't put another queen in same col^n. N queens for N×N board → 1 queen per column.

× → possibility discarded

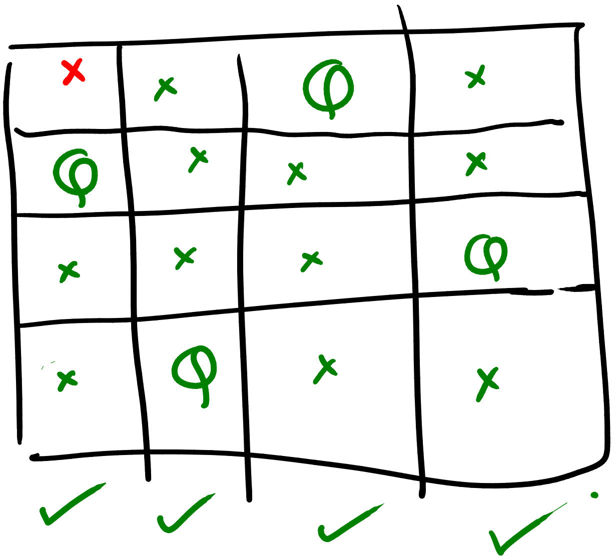
levels = 4

1 col : queen anywhere   # possibilities = 4

2 col : # possibilities = 2 ·1 ×
(actual)

3 col : # poss = 0 (dead end) | # poss = φ

4 col : # pos = 0 (dead end)



1 : col ✓

2^nd col : 1

3^rd col : ✓

4^th col : ✓

levels == N

sol^n

states : all cells on the board

possibilites : N for each col.

Constraints : — cell should not be attacked by previous queens.

levels : N

Knights : states : all cells of board.   levels : N*N

Possibilities : 8

Constraint : - step on cell twice
- should not cross boundary

Sudoku : State : all cells unfilled

possibilities : 1 to 9

constraints : - no repetition in row
col
3x3 subgrid

level : all cells unfilled.

Some code :

```
bool find_sol ( level , other params ...) :
    if ( found_a_solution ) :
        print a sol?
        return True
    bool is_valid = False        // you dont have an ans | assume
    for all possibilities at this level :
        if (constrains are satisfied) :
            save current possibility as my potential ans
            if find_solution ( level +1, other params...) :
                is_valid = True
                return is_valid
            remove current possibility
    return is_valid
```

what if it returns False

Backtracking Template.