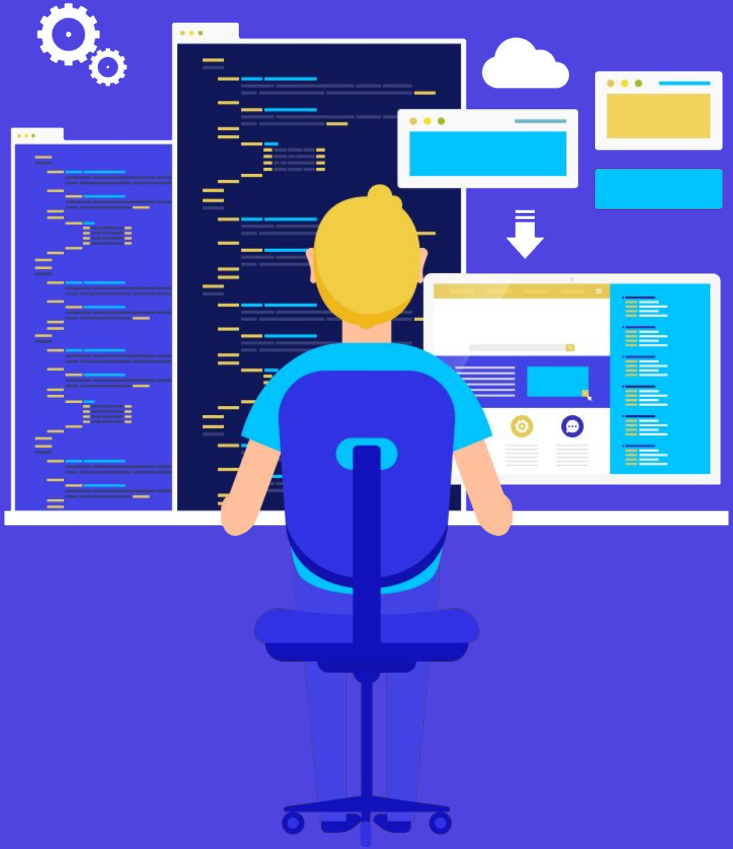
Advanced Problem Solving: Recursion



Relevel
by Unacademy

What does it take?

As we would be concentrating on hands-on advanced problem-solving in this session, please revise the concepts covered in the last two sessions:

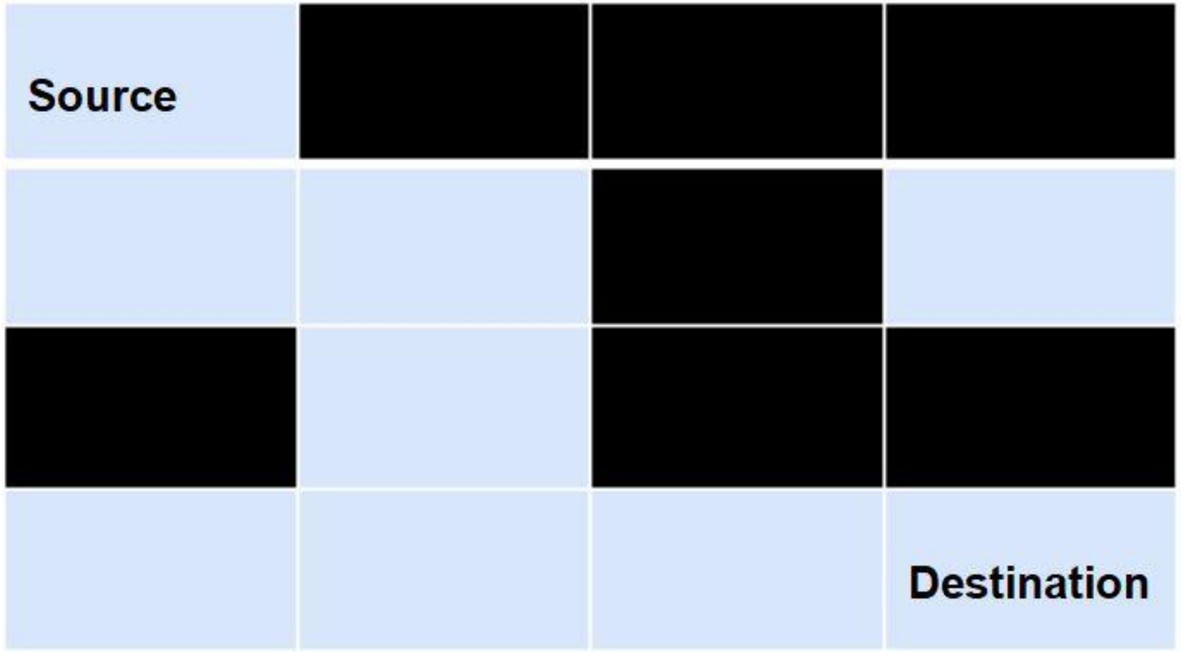
- Recursion
- Call Stack
- Base cases

List of Problems Involved

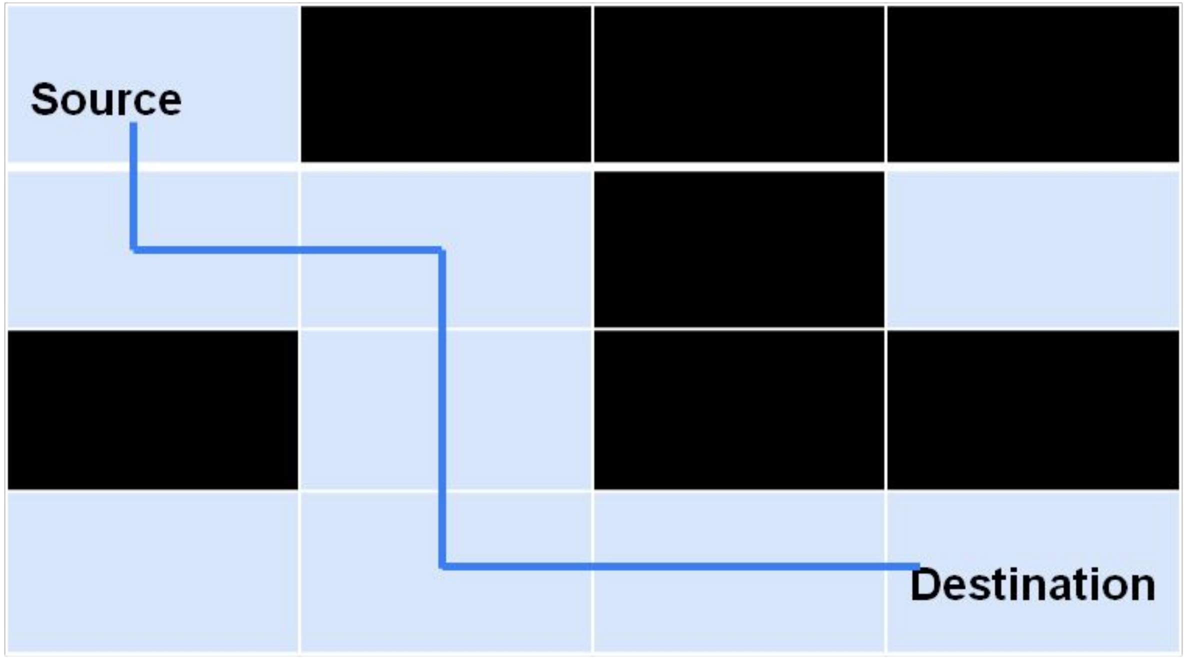
- Rat In a Maze
- N Queen
- Sudoku Solver
- Knights tour
- Squareful array

Rate In a Maze

Rat in a Maze – A maze is an N*N binary matrix of blocks where the upper left block is known as the Source block, and the lower rightmost block is known as the Destination block. If we consider maze, then maze[0][0] is source and maze[N-1][N-1] is destination. Our main problem task is to reach the destination from the source. We have considered a rat as a character that can move either forward or down.
In the maze matrix, a few dead blocks will be denoted by 0, and active blocks will be denoted by 1. A rat can move only in the active blocks.



```
{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}
```



```
{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}
```

Rate In a Maze

Intuition – We can solve this problem using recursion. We need a path from source to destination to try multiple paths. When we use any random path, and if it fails to reach the destination, we can backtrack and try another path. Since we are trying the same logic multiple times, we can use the recursive technique. We call this technique when we backtrack and try another path a Backtracking Algorithm.

Approach – Create a recursive function that will follow a path, and if that path fails, then backtrack and try another path. Let's see each step –

1. Create an output matrix having all values as 0
2. Create a recursive function, which will take input matrix, output matrix, and rat position (i,j)
3. If the position is not valid, then return.
4. Make output[i][j] as one and verify if the current position is the destination or not. If yes, return the output matrix
5. Recursively call the function for position (i+1, j) and (i,j+1)
6. Make output[i][j] as 0.

Code Link - https://jsfiddle.net/ke6a3yzo/

Time Complexity –

If there are N rows and N columns. Then complexity will be O(2^(N^2))

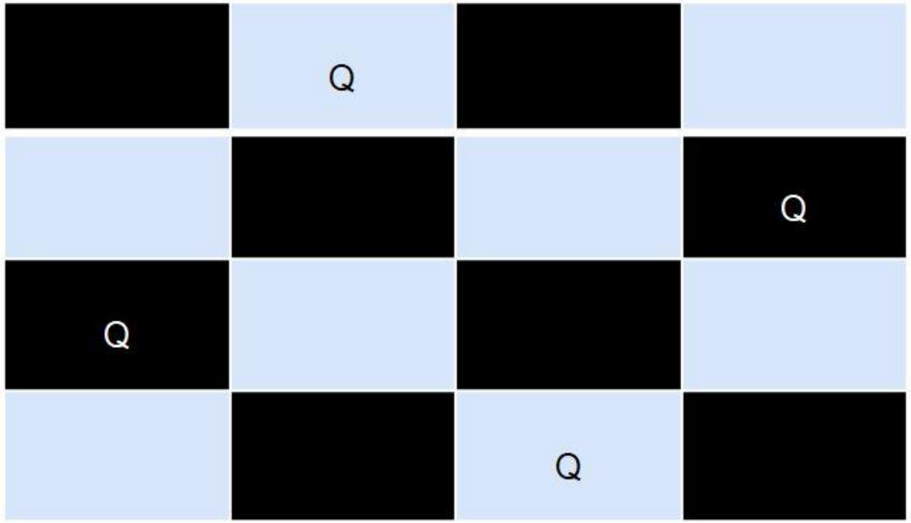
Space Complexity –

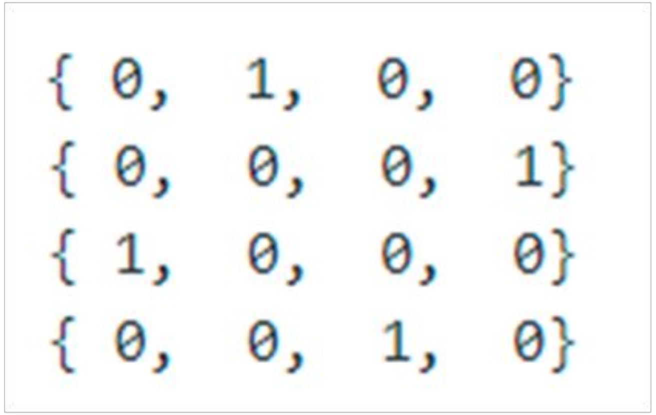
If there are N rows and N columns. Then complexity will be O(N^2)

N Queen

N Queen – Consider an N*N chessboard. N Queen's problem is to place N queens on the N*N chessboard such that no two queens can attack each other.

Example - Input - N = 4





```
{ 0,  1,  0,  0}
{ 0,  0,  0,  1}
{ 1,  0,  0,  0}
{ 0,  0,  1,  0}
```

N Queen

Naive Approach – Try to generate all possible combinations and print the combination satisfying the condition.
Efficient Approach – This can be solved using recursion.
Intuition - Our main task is to place a queen with no clash with another queen. We will check this condition for all columns one by one. We will backtrack and check for another column if there is any clash. This technique is a backtracking algorithm.
Steps -

1. Start from the leftmost column
2. If all queens are placed, then return true. (base case)
3. Iterate through every row for the current column
4. If the queen is safely placed in the current row, mark [row, column] as a solution.
5. Verify if placing the queen in her current position is safe, return true
6. If placing queen is not safe, unmark [row, column], then backtrack and go to step 4 to check other rows
7. If all the rows have been checked and are not fulfilling the condition, return false and backtrack again.

Code Link -> https://jsfiddle.net/br6m2j3n/

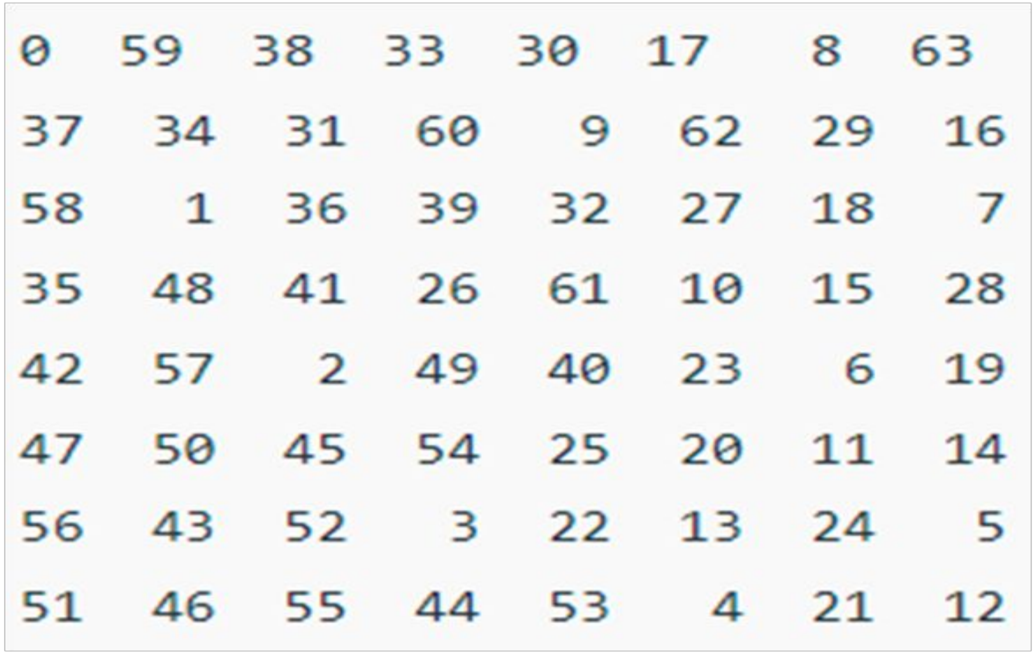
Time Complexity –
If there are N Queens. Then complexity will be O(N!)

Space Complexity –
If there are N Queens. Then complexity will be O(N)

Knight's Tour

Knight's Tour – Consider an N*N chessboard. Knight's Tour problem is to print order when Knight visits that block of the chessboard. Initially, the knight will be placed at the first block of the chessboard. The rule is that the knight visits each block exactly once.

Example - Input - N = 8



```
0   59  38  33  30  17   8  63
37  34  31  60   9  62  29  16
58   1  36  39  32  27  18   7
35  48  41  26  61  10  15  28
42  57   2  49  40  23   6  19
47  50  45  54  25  20  11  14
56  43  52   3  22  13  24   5
51  46  55  44  53   4  21  12
```

Knight’s Tour

Naive Approach – Try to generate all possible tours and print the tour that satisfies the condition.
Efficient Approach - We can solve this problem using recursion
Intuition – We need output such that the knight visited the block exactly once. So we will take an empty vector that will take knight moves, and if that move violates the rule, we will backtrack and try for another move.
Steps –

1. Create a solution vector
2. Verify if all blocks visited, print the solution
3. If not, add the next move to the solution vector, recursively check if this move satisfies the condition.
4. If the above move is not valid, remove this from the solution vector and try another move.
5. If none of the moves works, then return false.

Code Link -> https://jsfiddle.net/3gtc0sa4/

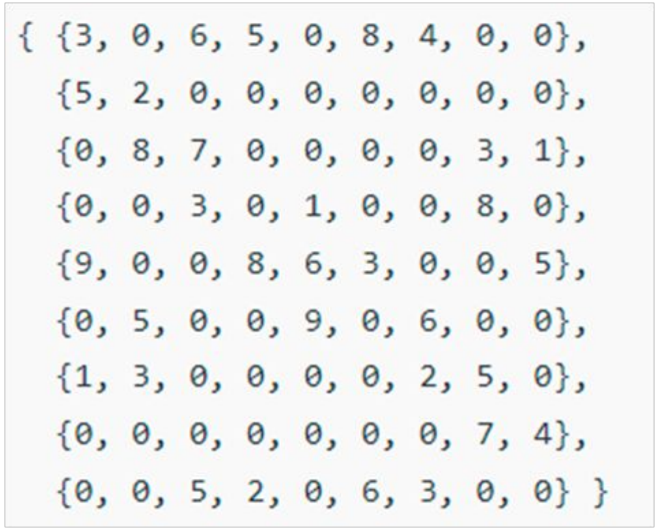
Time Complexity –
If there are N rows and N columns. Then complexity will be O(8^(N^2))

Space Complexity –
If there are N rows and N columns. Then complexity will be O(N^2)

Sudoku Solver

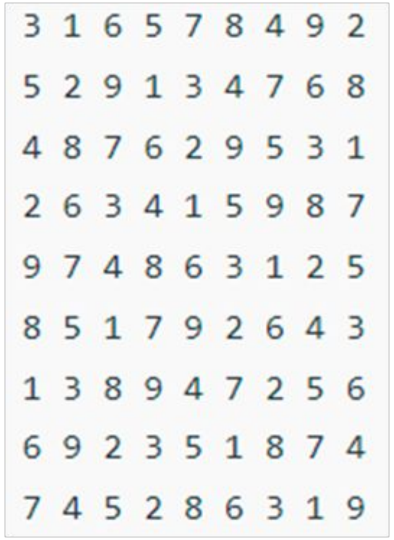
Sudoku Solver – Consider a 9*9 2D array grid partially filled with numbers from 1 to 9. Sudoku Solver problem is to fill remaining blocks with numbers from 1 to 9, such that every row, column, and subgrid (3*3) contains exactly one instance of digits(1 to 9).

Input



```
{ {3, 0, 6, 5, 0, 8, 4, 0, 0},
  {5, 2, 0, 0, 0, 0, 0, 0, 0},
  {0, 8, 7, 0, 0, 0, 0, 3, 1},
  {0, 0, 3, 0, 1, 0, 0, 8, 0},
  {9, 0, 0, 8, 6, 3, 0, 0, 5},
  {0, 5, 0, 0, 9, 0, 6, 0, 0},
  {1, 3, 0, 0, 0, 0, 2, 5, 0},
  {0, 0, 0, 0, 0, 0, 0, 7, 4},
  {0, 0, 5, 2, 0, 6, 3, 0, 0} }
```

Output



```
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

Sudoku Solver

Naive Approach – Try to generate all possible combinations and print the combination satisfying the condition.
Efficient Approach - We can solve this problem using recursion
Intuition – We need output such that rows, columns, and subgrids have one instance from 1 to 9. We can assign a number to the empty cell and check if it is safe to assign the value. If that assignment violates the rule, we will backtrack and try for another assignment of number.
Steps –

1. Create a function that will check if the assignment of the number to the cell is safe or not.
2. Create a recursive function that will take grid as input
3. Assign a number to an unfilled cell, and check if it is safe to assign; if yes, then recursively call the function for all safe cases. If any recursive call returns true, then return true. If no recursive call returns true, then return false.
4. If there is no unassigned cell, return true.

Code Link -> https://jsfiddle.net/zrv0Lmjg/

Time Complexity –

Time complexity will be O(9^(N*N))

Space Complexity –

Space complexity will be O(N*N)

Squareful Arrays

Squareful Arrays – An array is known as a squareful array if for every pair of the adjacent numbers, the sum of numbers is a perfect square.

Squareful Arrays Problem - We need to find several permutations of the input array: squareful arrays.

Example -

Input - [1,17,8]

Output - 2

[1,8,17] and [17,8,1] are 2 squareful arrays

Squareful Arrays

Intuition – Since we need to check all adjacent pair sums when we are at i position, we need to check for i-1 position and i+1 position number.

Steps -

1. Create a temporary and boolean array that will keep track of visited nodes.
2. Sort the input array.
3. Create a recursive function that will take a sorted input array, a temporary array, and a boolean array
4. If temporary array size is equal to input array, increment the output and return
5. Iterate through the input array
6. If the current number and previous number are equal and the previous number is not visited, then continue
7. Check if the sum is a perfect square; if not, continue.
8. If the current number has already been visited, continue
9. Add the number to the temporary array and mark the visited array as true
10. Call recursive function again to check

Code Link - https://jsfiddle.net/byL4tfqw/

Time Comple`xity –

Time complexity will be O(N^N)

Space Complexity –

Space complexity will be O(N)

MCQ Questions

1) What happens when we reach to final solution in Backtracking algorithm?
   a) It continues searching for other solutions [Correct Answer]
   b) Recursively traverses via the same route
   c) It backtracks to the base condition
   d) None

2) Backtracking algorithm is used to solve which kind of problems?
   a) Combinatorial Problems [Correct Answer]
   b) Numerical Problems
   c) Exhaustive search
   d) None

3) Backtracking Algorithm is faster then naive approach?
   a) true [Correct Answer]
   b) false

MCQ Questions

4) Which of the following is similar to recursion?

1. If-else
2. Loop [Correct Answer]
3. Switch case
4. If elif else

5) In recursion, which of the following case will stop function to calling itself?

1. Base case [Correct Answer]
2. Worst case
3. Best case
4. None

Practice Questions

1) Given an array arr and sum s. Find all unique combinations from array whose sum is equal to s.
2) Find all subsets of an array arr using backtracking algorithm

Upcoming Class Teaser

- Order of Growth
- Asymptotic Analysis
- Various types of Complexity Notations
- Big O notation
- Time Complexity
- Space Complexity
- Calculating Time and Space Complexity

THANK YOU