ECE 220: Computer Systems & Programming

Lecture 15: Problem Solving with recursion: Recursion with Backtracking

Thomas Moon

February 29, 2024

Maze Solver

	X	X	
X	*	*	X
*	*	X	
E	*		

one of solutions

	X	X	
X			X
		X	
E			

Wall

Starting point

Exit

- GOAL
  - *(if exists)* find a path from the starting point to the exit → Return 1 and mark '*'
  - One solution is enough (may not be the shortest path)

	X	X	
X			
		X	
E			

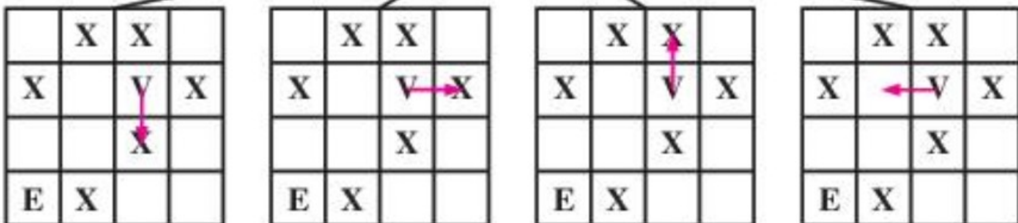
circle!

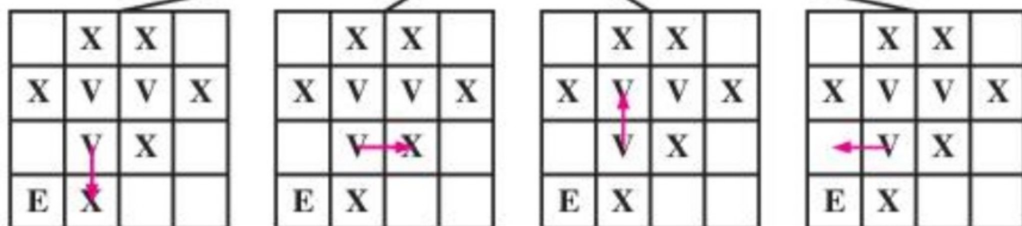
- IDEA
  - Try 1 of 4 moves (U/D/L/R)
  - Solve it from there (recursively!)
  - Mark 'V', if visited (avoid circling)

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
```

Initial Maze Configuration with Starting Point



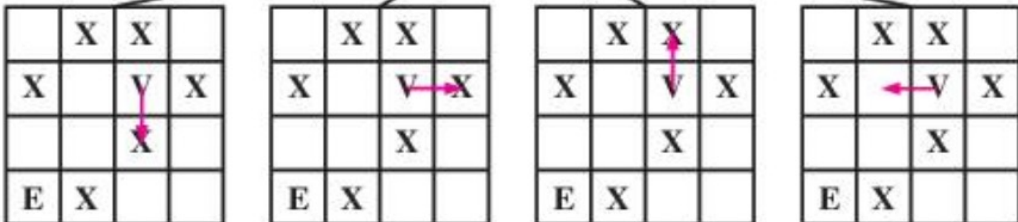


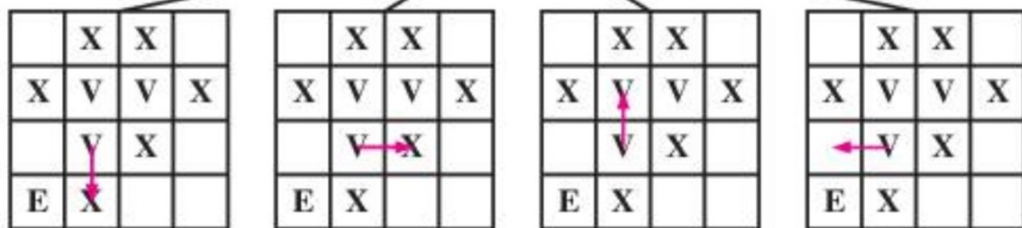
Exit Found!

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
```

Initial Maze Configuration with Starting Point





Exit Found!

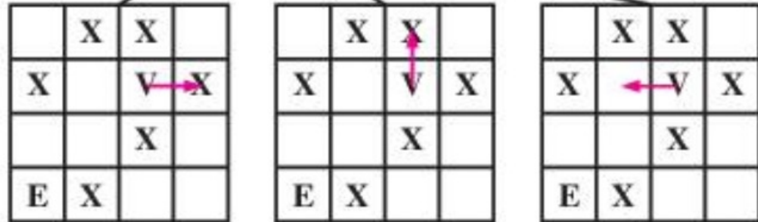
Initial Maze Configuration with Starting Point

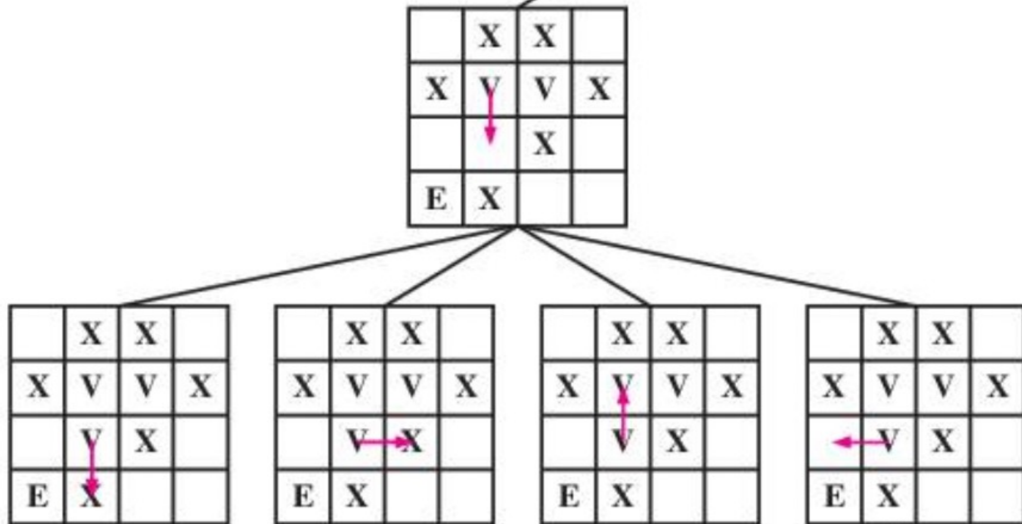
```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 2)
```

Exit Found!

Initial Maze Configuration with Starting Point





Exit Found!

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 2)
```

```
if(maze[xpos][ypos] == 'V' ||
maze[xpos][ypos] == 'X')
        return 0;
```

Initial Maze Configuration with Starting Point

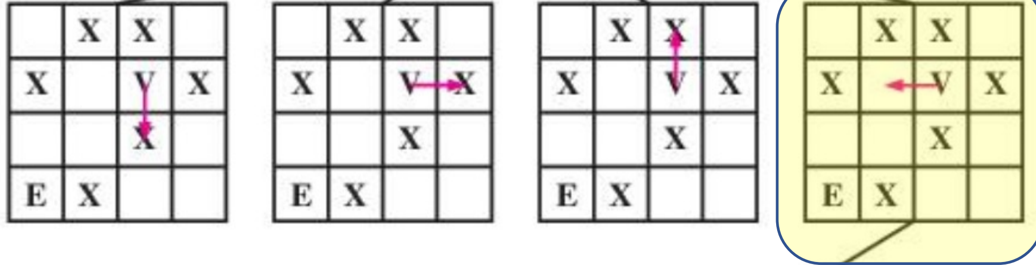
Exit Found!

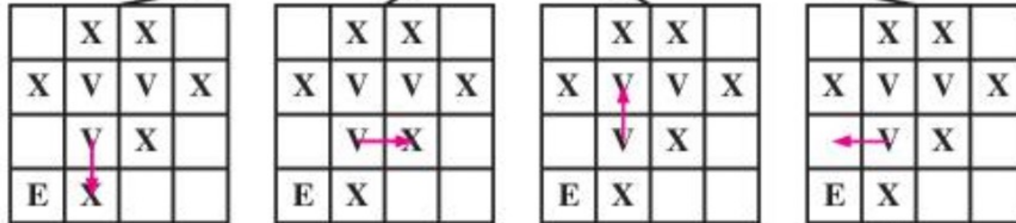
```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos + 1) == 1 )
```

```
solve(maze, 1, 3)
```

```
if(maze[xpos][ypos] == 'V' ||
maze[xpos][ypos] == 'X')
        return 0;
```





Initial Maze Configuration with Starting Point

Exit Found!

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

```
maze[xpos][ypos] = 'V';
```

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
```

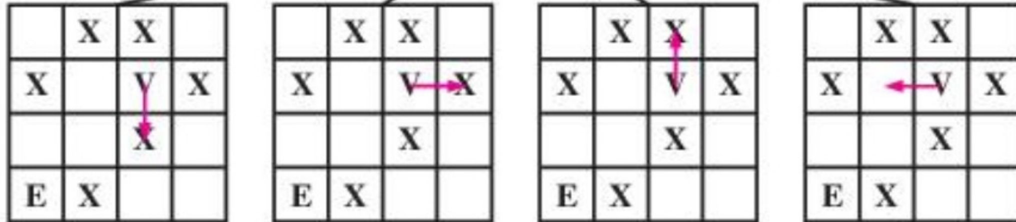
Initial Maze Configuration with Starting Point

Exit Found!

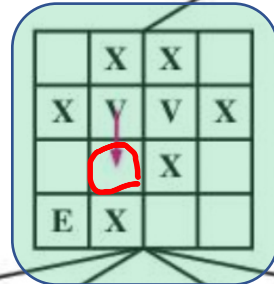
```
res = solve(maze, 1, 2);
```

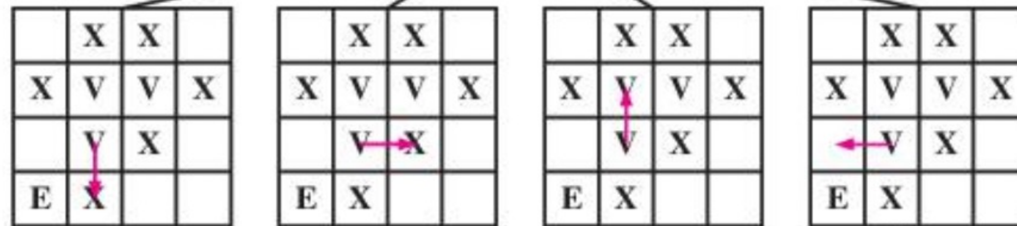
Initial Maze Configuration with Starting Point

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

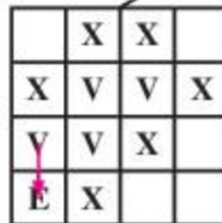


```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)
```





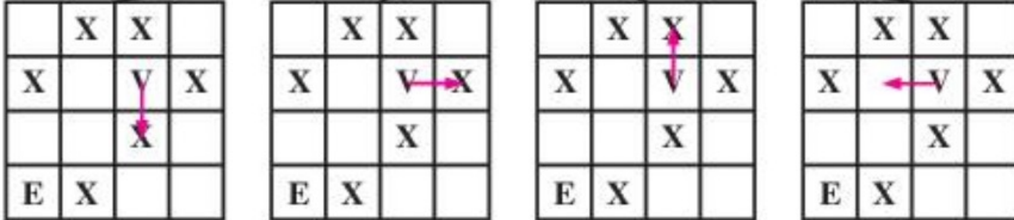
Exit Found!



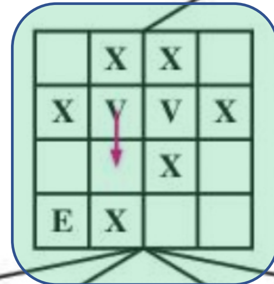
```
res = solve(maze, 1, 2);
```

Initial Maze Configuration with Starting Point

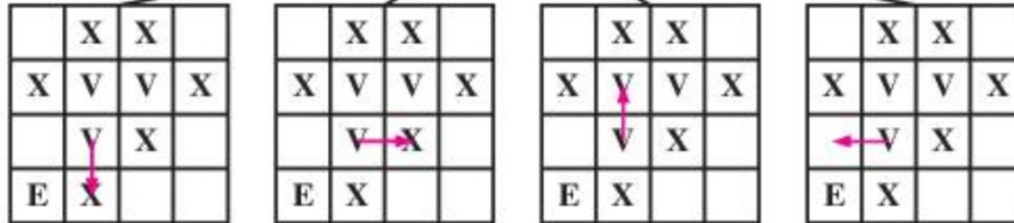
```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```



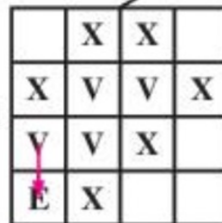
```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)
```



```
maze[xpos][ypos] = 'V';
```



Exit Found!



Initial Maze Configuration with Starting Point

```
res = solve(maze, 1, 2);

maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)

  maze[xpos][ypos] = 'V';
  if( solve(maze, xpos + 1, ypos) == 1 )
    solve(maze, 2, 1)

    maze[xpos][ypos] = 'V';
    if( solve(maze, xpos + 1, ypos) == 1 )
      solve(maze, 3, 1)
```

Exit Found!

Initial Maze Configuration with Starting Point

Exit Found!

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 3, 1)
```

```
if(maze[xpos][ypos] == 'V' ||
maze[xpos][ypos] == 'X')
        return 0;
```

Initial Maze Configuration with Starting Point

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 2, 0)
```

Exit Found!

```
res = solve(maze, 1, 2);
```

Initial Maze Configuration with Starting Point

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 2, 0)
```

```
maze[xpos][ypos] = 'V';
```

Exit Found!

Initial Maze Configuration with Starting Point

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 2, 0)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
```

Exit Found!

Initial Maze Configuration with Starting Point

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 2, 0)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 3, 0)
```

```
if(maze[xpos][ypos] == 'E')
        return 1;
```

Exit Found!

Initial Maze Configuration with Starting Point

```
res = solve(maze, 1, 2);

maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)

maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)

maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 2, 0)

maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 3, 0)
  1

if(maze[xpos][ypos] == 'E')
        return 1;
```

Exit Found!

Initial Maze Configuration with Starting Point

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 2, 0)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 ){
       maze[xpos][ypos] = '*';
       return 1;
}
```

Exit Found!

Initial Maze Configuration with Starting Point

Exit Found!

```
res = solve(maze, 1, 2);

maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)

maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)

maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 2, 0)
```

1

Initial Maze Configuration with Starting Point

Exit Found!

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 )
  solve(maze, 2, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 ) {
      maze[xpos][ypos] = '*';
       return 1;
}
```

1

```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 )
  solve(maze, 1, 1)
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos + 1, ypos) == 1 ) {
        maze[xpos][ypos] = '*';
        return 1;
}
```

1

Initial Maze Configuration with Starting Point

Exit Found!

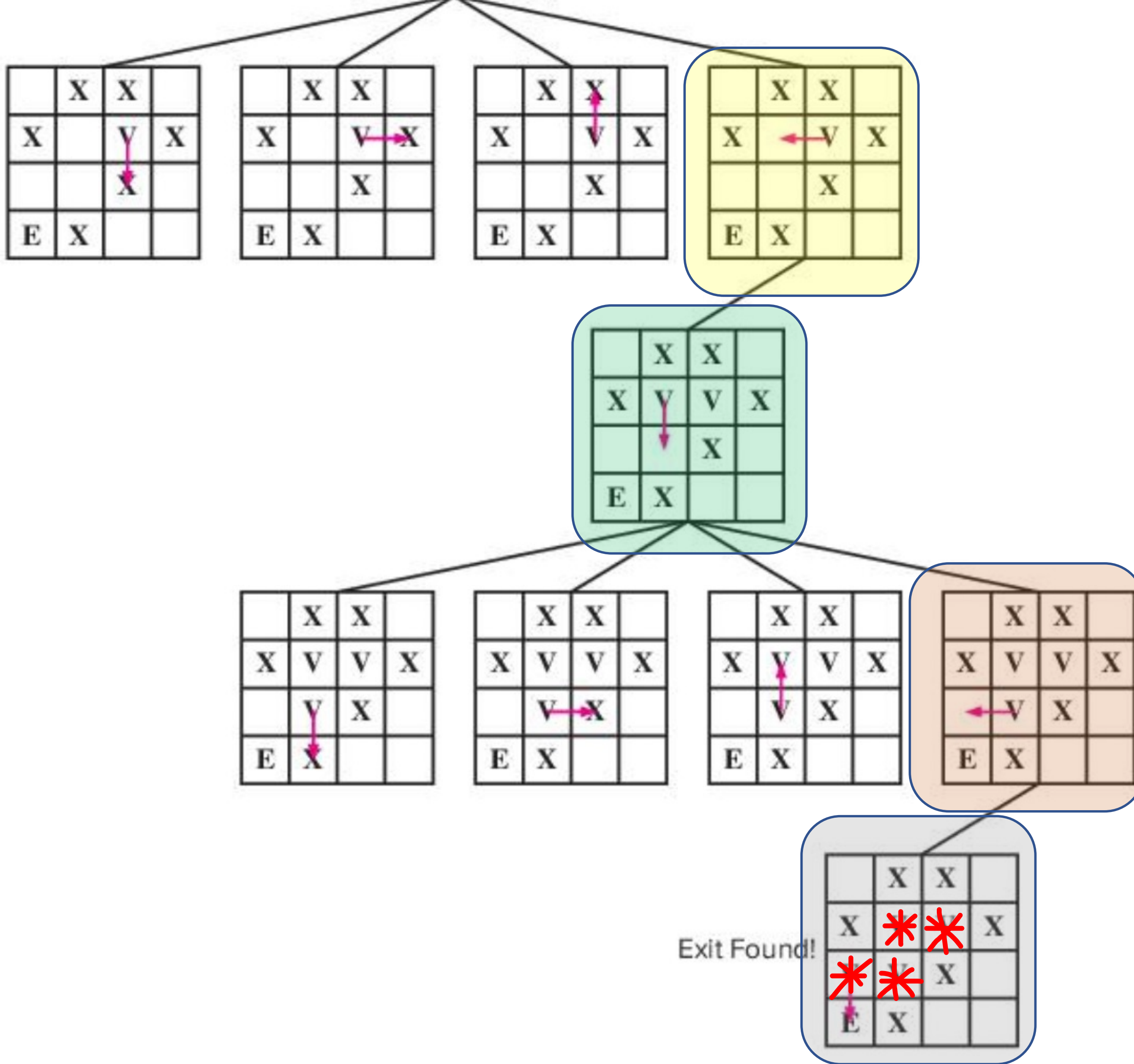
```
res = solve(maze, 1, 2);
```

```
maze[xpos][ypos] = 'V';
if( solve(maze, xpos, ypos - 1) == 1 ) {
        maze[xpos][ypos] = '*';
        return 1;
}
```

1

Initial Maze Configuration with Starting Point

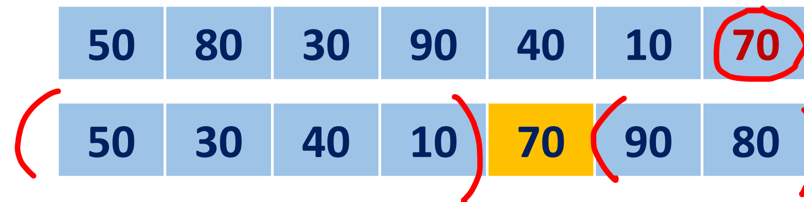
Exit Found!

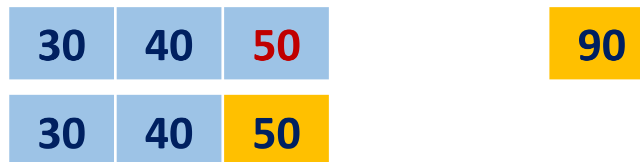



```
res = solve(maze, 1, 2);
```

Quick Sort (divide-and-conquer)

1. Pick a <u>pivot</u> and <u>partition</u> array into 2 subarrays (smaller elements than the pivot in the left and greater elements in the right)
2. Sort the 2 subarrays using the same method



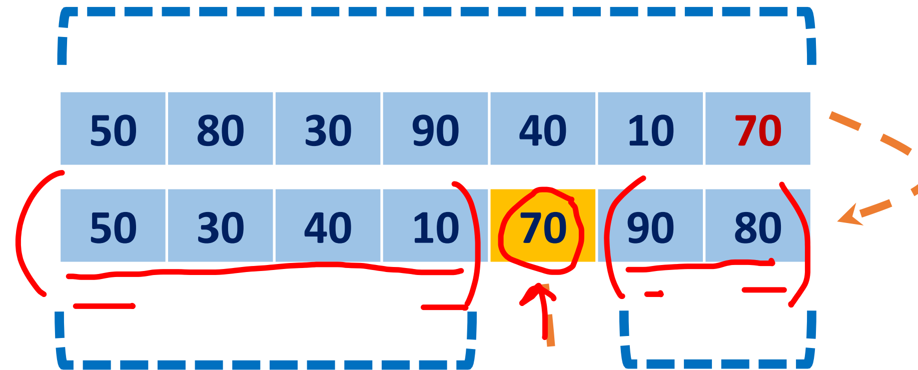


⋮

Quick Sort : Recursive case

```
void quickSort(int array[], int low, int high)
int partition(int array[], int low, int high)
```

quickSort(array, low, high)



partition

After partition,

1. Array partitioned around the pivot
2. Pivot index returned

quickSort(array, low, pivot_idx-1)

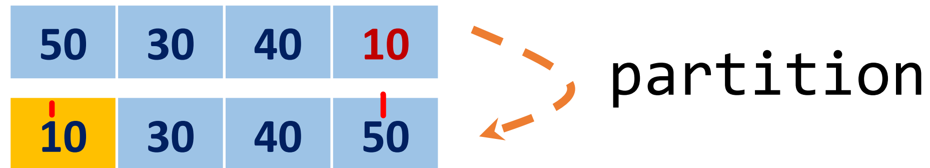
quickSort(array, pivot_idx+1, high)

pivot_idx

Quick Sort: Base case

```
quickSort(array, low, high)
```

```
low=0, high=3
```




```
pivot_idx = 0
```

0 -1

```
quickSort(array, low, pivot_idx-1);
quickSort(array, pivot_idx+1, high);
```

```
low > high
```

```
low=0, high=0
```

90

```
low == high
```

All together, the base case is

```
low >= high
```

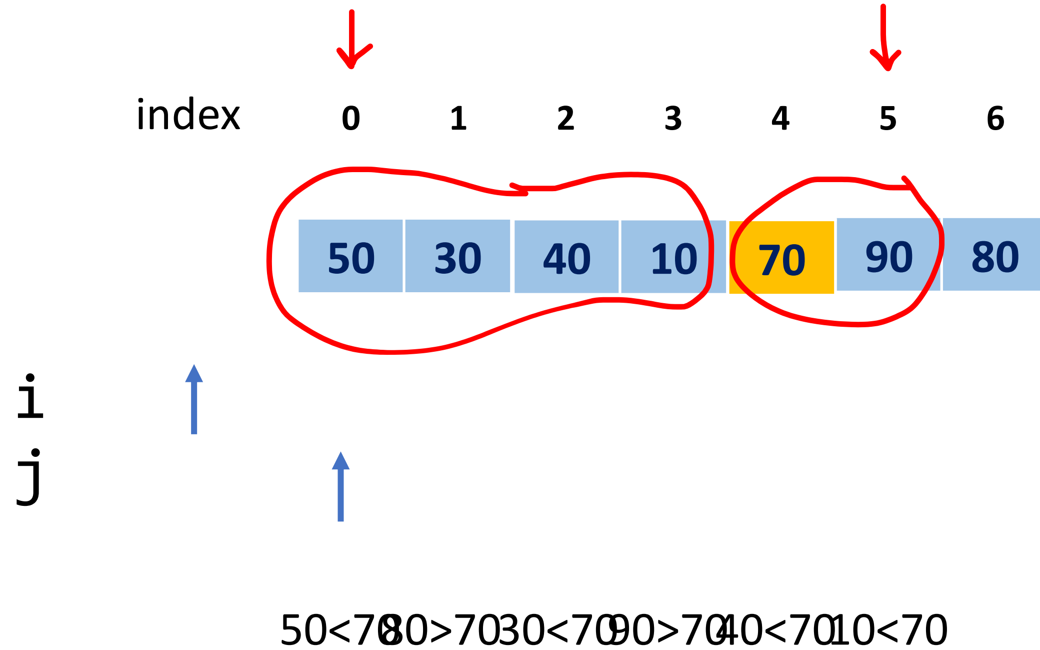
Implement Quick Sort

```
void quickSort(int array[], int low, int high)
{
	//base case
	if(low > = high)
		return;

	//recursive case
	int pivot_index = partition(array, low, high);
	quickSort(array, low, pivot_index – 1);// first half
	quickSort(array, pivot_index + 1, high);//second half
}
```

Quick Sort: Partition

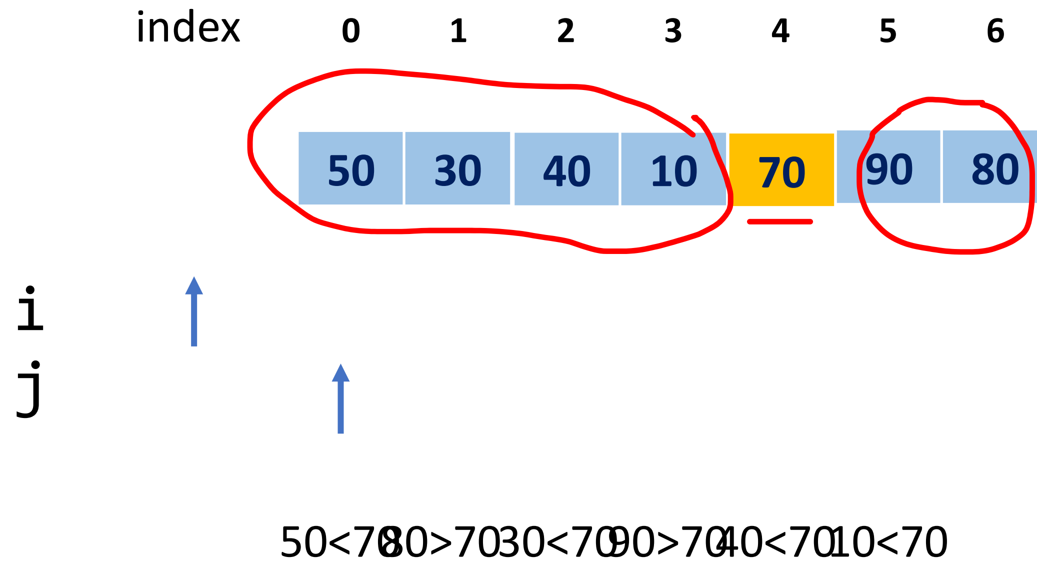
```
i: index of smaller elements (i=-1)
j: loop index (j=0)
```




```
for j from low to high-1{
    if array[j] < pivot
        i = i + 1;
        swap array[i] and array[j]
}
swap array[i+1] and pivot
return i+1 as the pivot index
```

Quick Sort: Partition

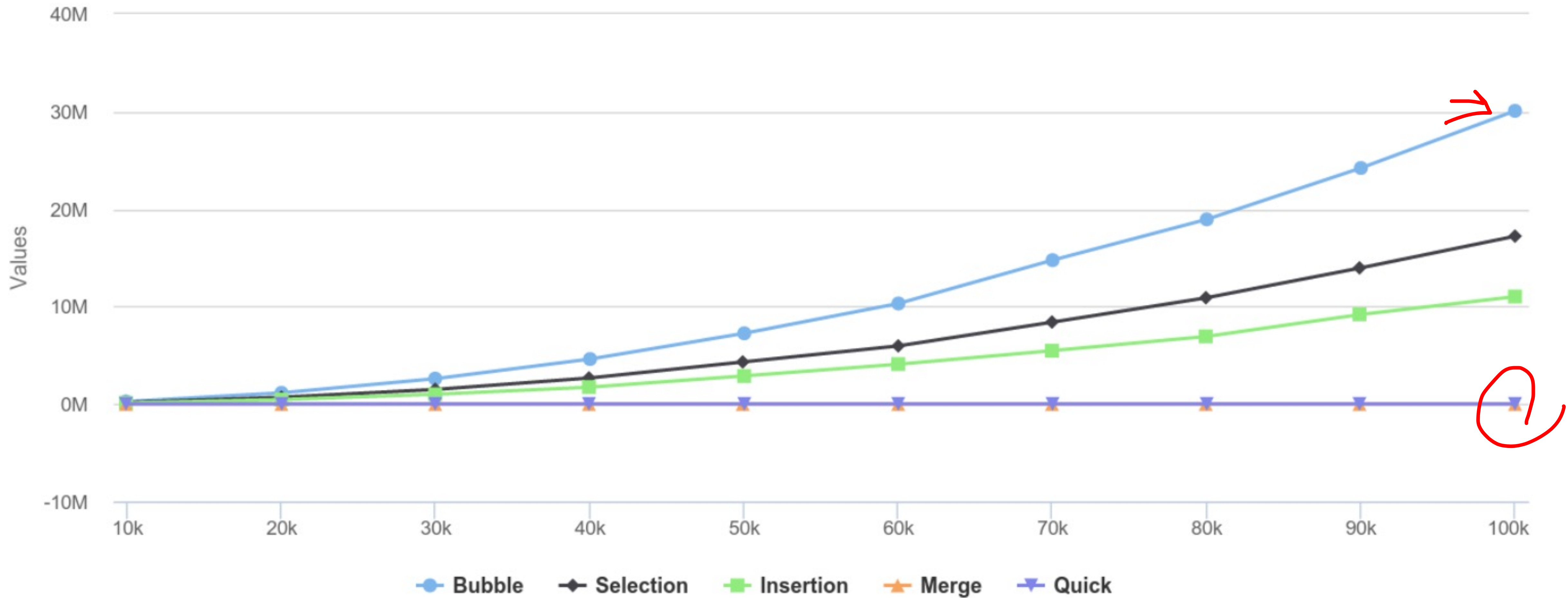
```
i: index of smaller elements (i=-1)
j: loop index (j=0)
```




```
for j from low to high-1{
    if array[j] < pivot
        i = i + 1;
        swap array[i] and array[j]
}
swap array[i+1] and pivot
```

```
At the end of loop,
1. array[0] … array[i] < pivot
2. return i+1  (pivot index)

Use the pivot index for the recursive call
```


40M
30M
20M
10M
0M
-10M
Values
10k
20k
30k
40k
50k
60k
70k
80k
90k
100k
Bubble
Selection
Insertion
Merge
Quick

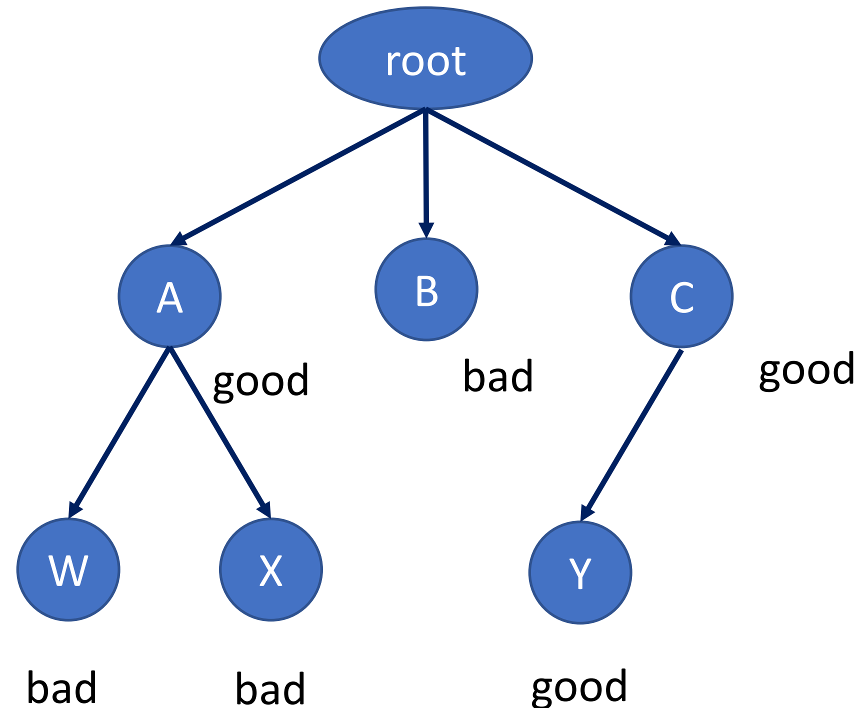
N-Queen Problem

- Place N chess queens in an NxN chessboard so that none of the queens are threatened.

	0	1	2	3
0			Q	
1	Q			
2				Q
3		Q		

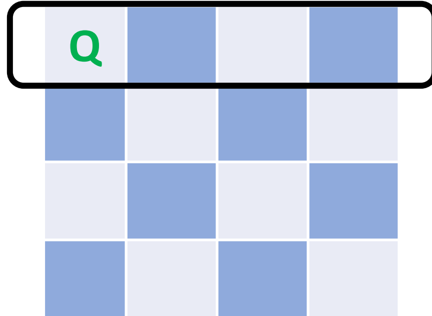
Recursive with Backtracking

- **Backtracking**:
  Solve problem recursively by trying to build a solution incrementally (one piece at a time), removing the solutions that fail to satisfy the constraints of problem.

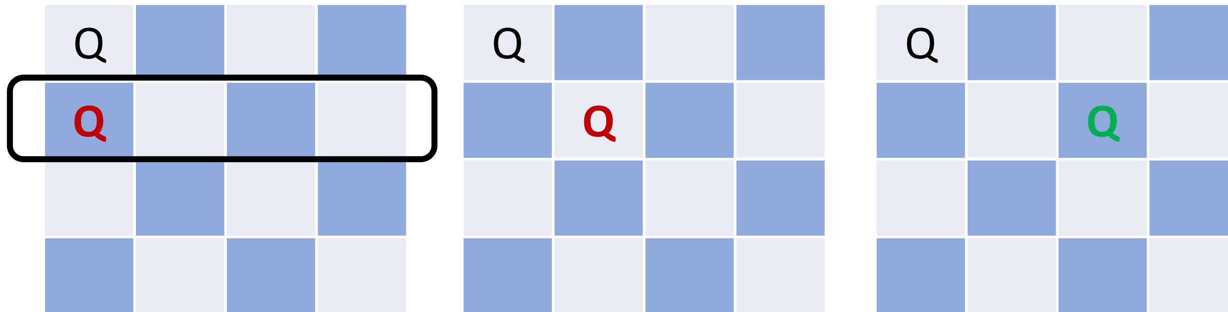


Example: 4x4 N-Queen

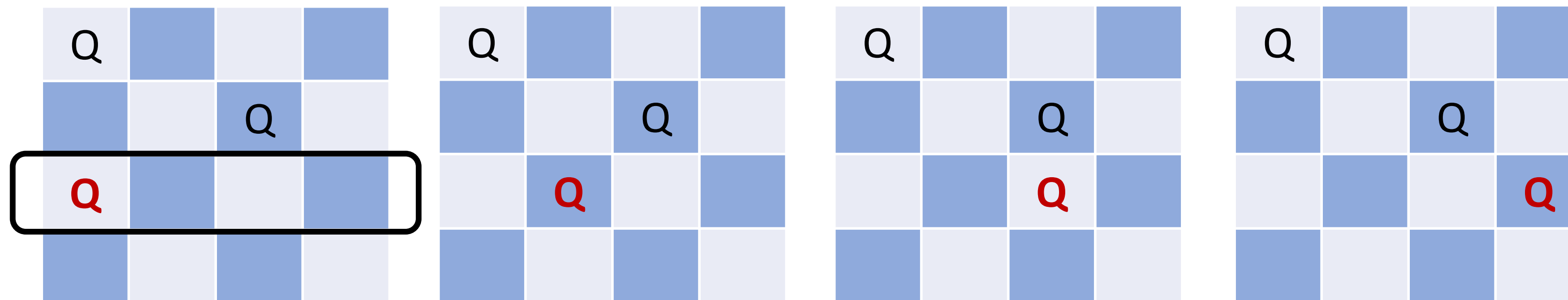
row 0



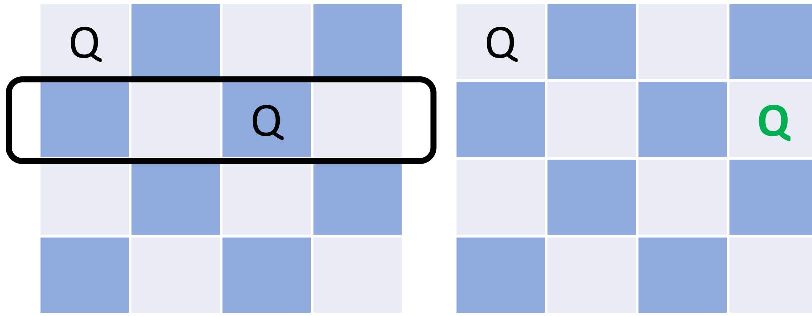
row 1



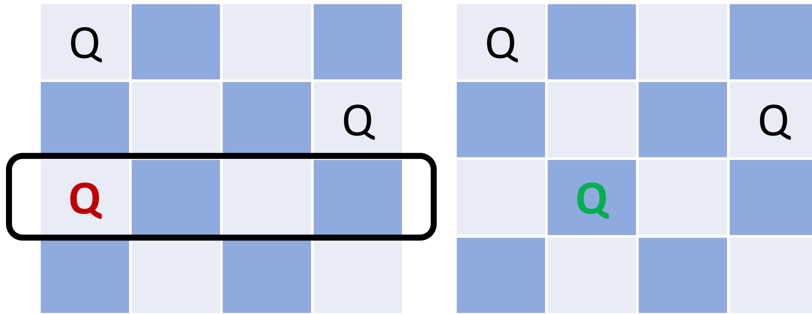
row 2



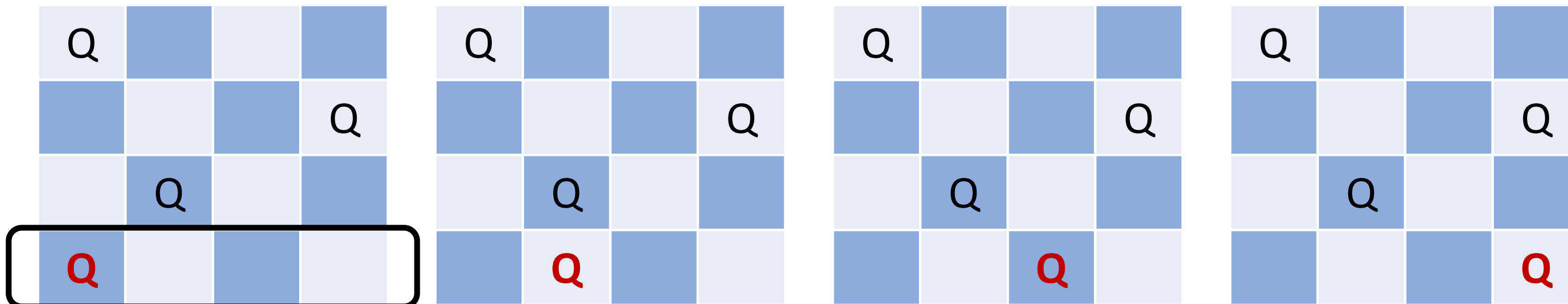
Backtrack to row1 and make a new choice



row 2

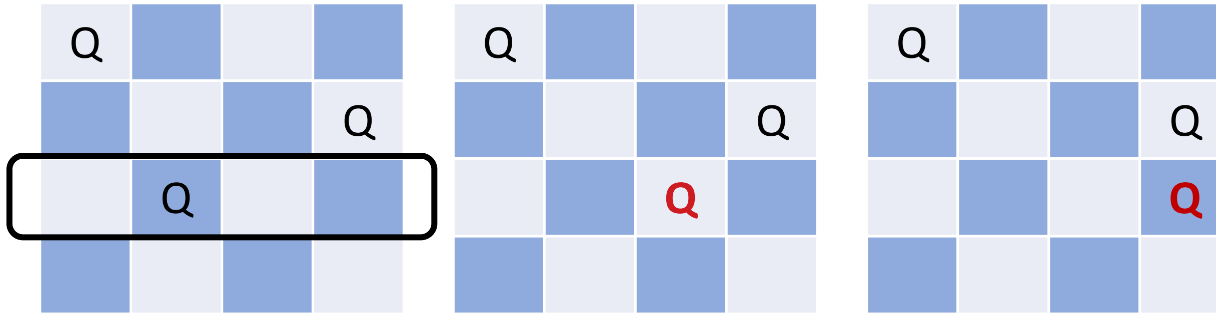


row 3



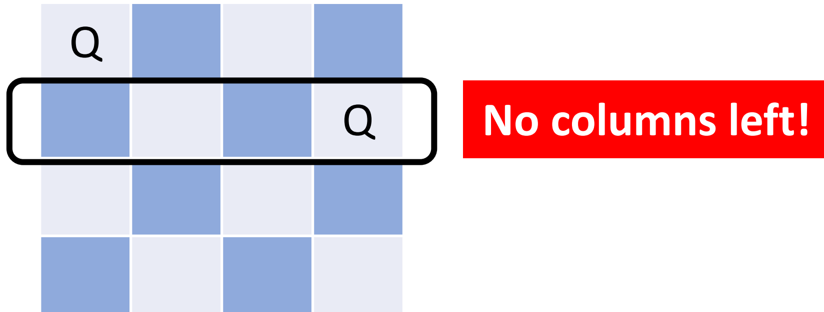
No safe column!

Backtrack to row2
and make
a new choice

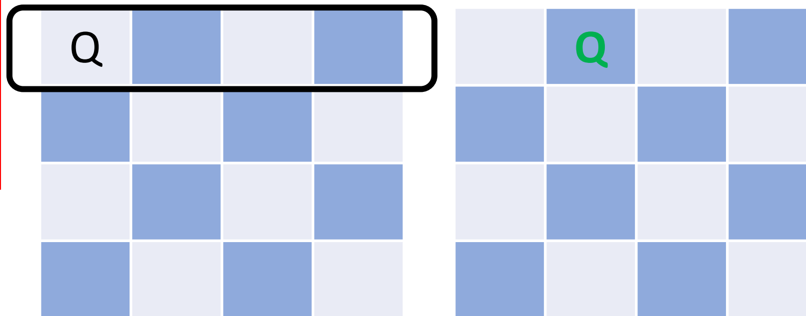


No safe column!

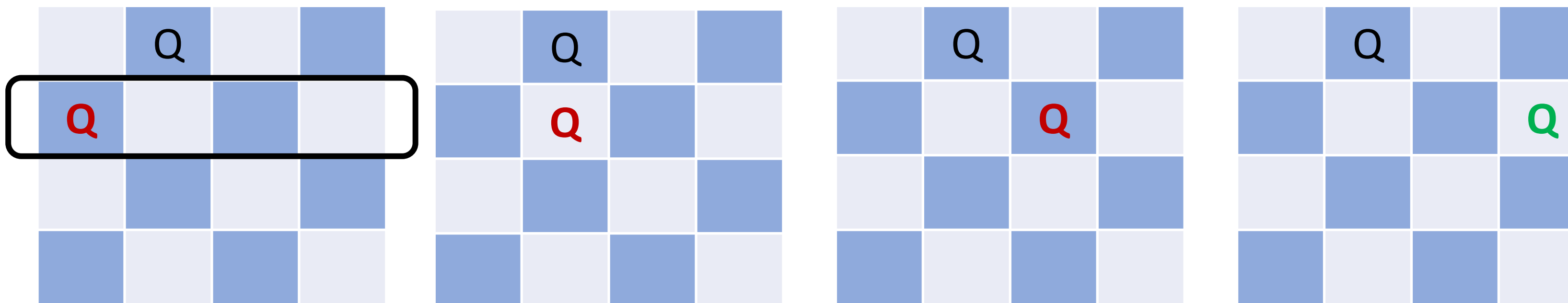
Backtrack to row1
and make
a new choice



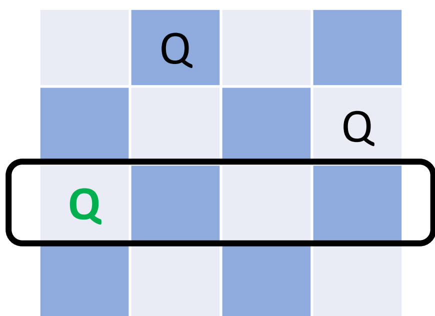
Backtrack to row0
and make
a new choice



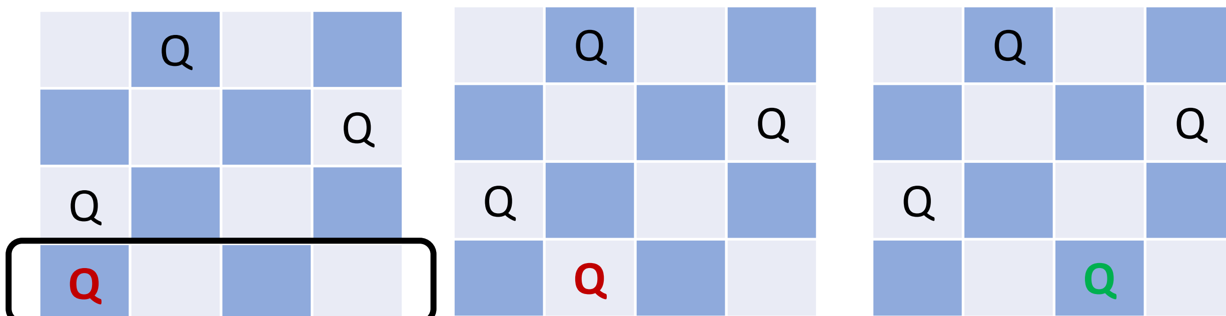
row 1



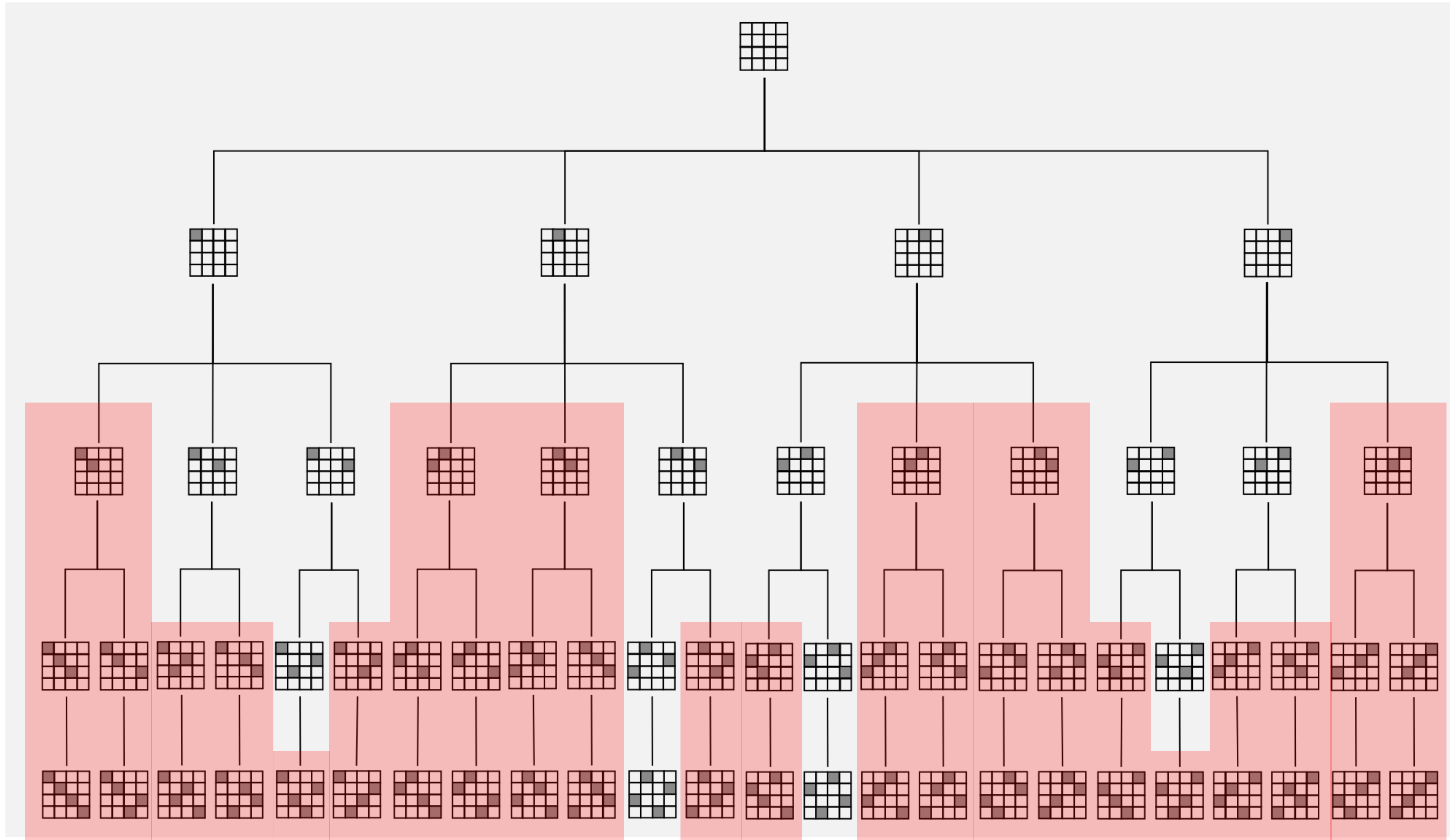
row 2



row 3



Recursive Tree



Recursive with Backtracking

- N-Queen Problem by Backtracking
  1. **Decision**
     Place a queen at a safe place.
  2. **Recursion**
     Explore the solution for the next row.
  3. **Backtrack (Undo)**
     Remove the queen if no solution for the next row.
  4. **Base case**
     Reach the goal.

Implement N-Queen Solver

```
// Solve N-queen probelm by placing (N-row) queens starting from row
int Solve(int board[N][N], int row);
int main()
{
    int board[N][N] = {0};
    if( Solve(board, 0) == false )
        printf("Solution does not exist. \n");

    printf("solution is\n");
    printSolution(board);
}
```

Implement N-Queen Solver

```c
// Utility function to check if a queen can be placed on board[row][col]
int isSafe(int board[N][N], int row, int col);

int Solve(int board[N][N], int row)
{
    int j;//index for column
    if(row == N)
        return true;
    else{
        for(j=0;j<N;j++){
            if(isSafe(board, row, j) == true){
                board[row][j] = 1;
                // increment row to place the next queen
                if(Solve(board, row+1)== true)
                    return true;
                else
                    board[row][j] = 0; //remove the queen
            }
        }
        return false;
    }
}
```

<table>
<tr><td></td><td>Q</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>Q</td></tr>
<tr><td>Q</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

Excercise

- You have a pile of wood sticks with 3 different lengths: 3, 7, and 10 feet. You wants to connect them and make an X-feet long stick using at most 10 sticks.

eg) To make 33-feet long stick,

→ 4 x 3feet + 3 x 7feet (a solution)

→ 11 x 3feet (used more than 10 sticks, not a solution)

To make 11-feet long stick,

→ No solution exists.

Use backtracking algorithm to find a solution.

```c
// solution[N]: stores the solution
// idx: index for the solution matrix
// total: remaining length
int solve(int solution[N], int idx,  int total);

#define N 10
#define M 3
const int set[M] = {3,7,10};
int main(){
    int solution[N] = {0};
    int total;

    printf("Enter total length: ");
    scanf("%d", &total);

    solve(solution, 0, total));
```