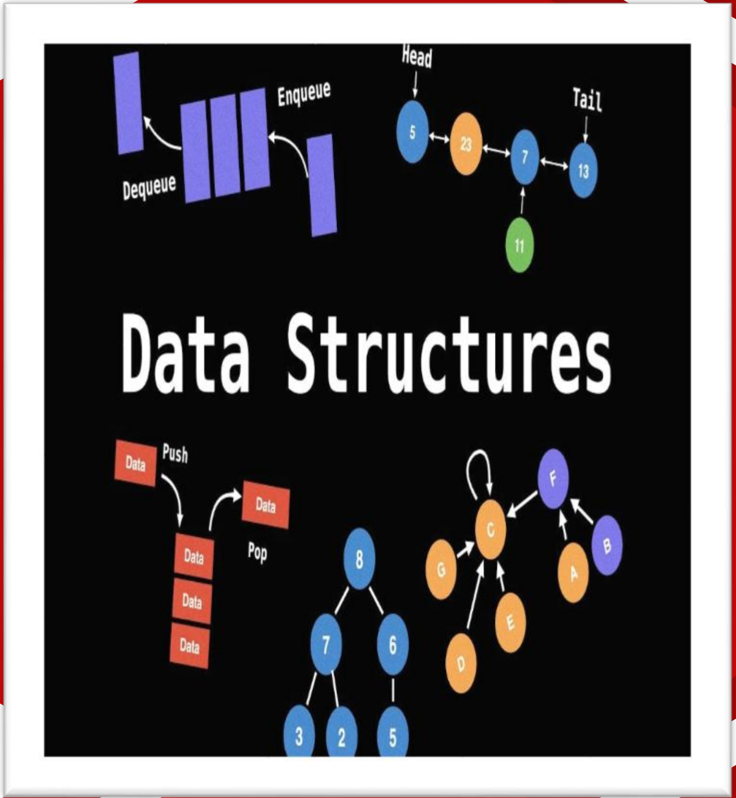
Algorithms and Data Structures

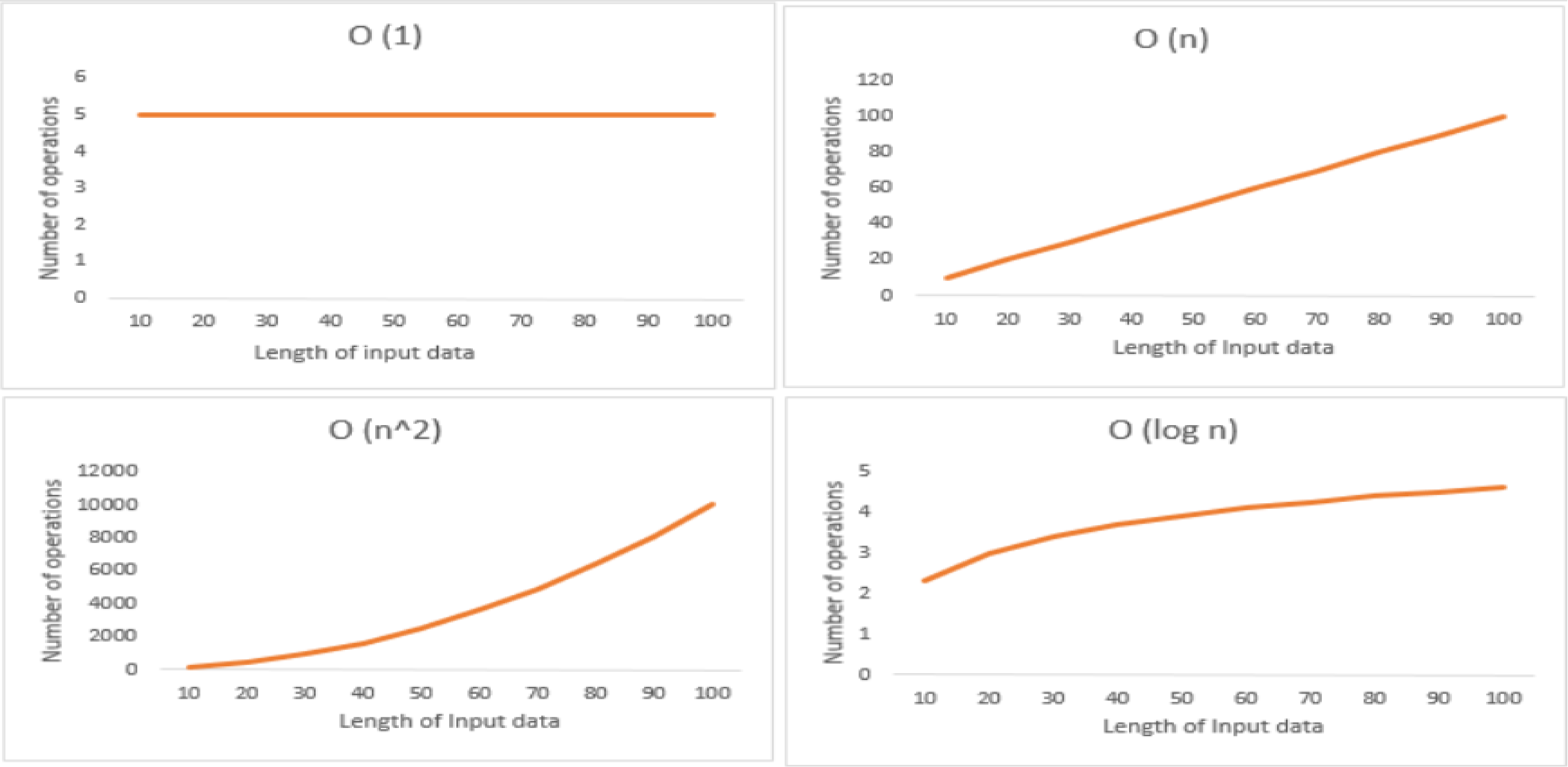
Data Structure
Analysis of Algorithms

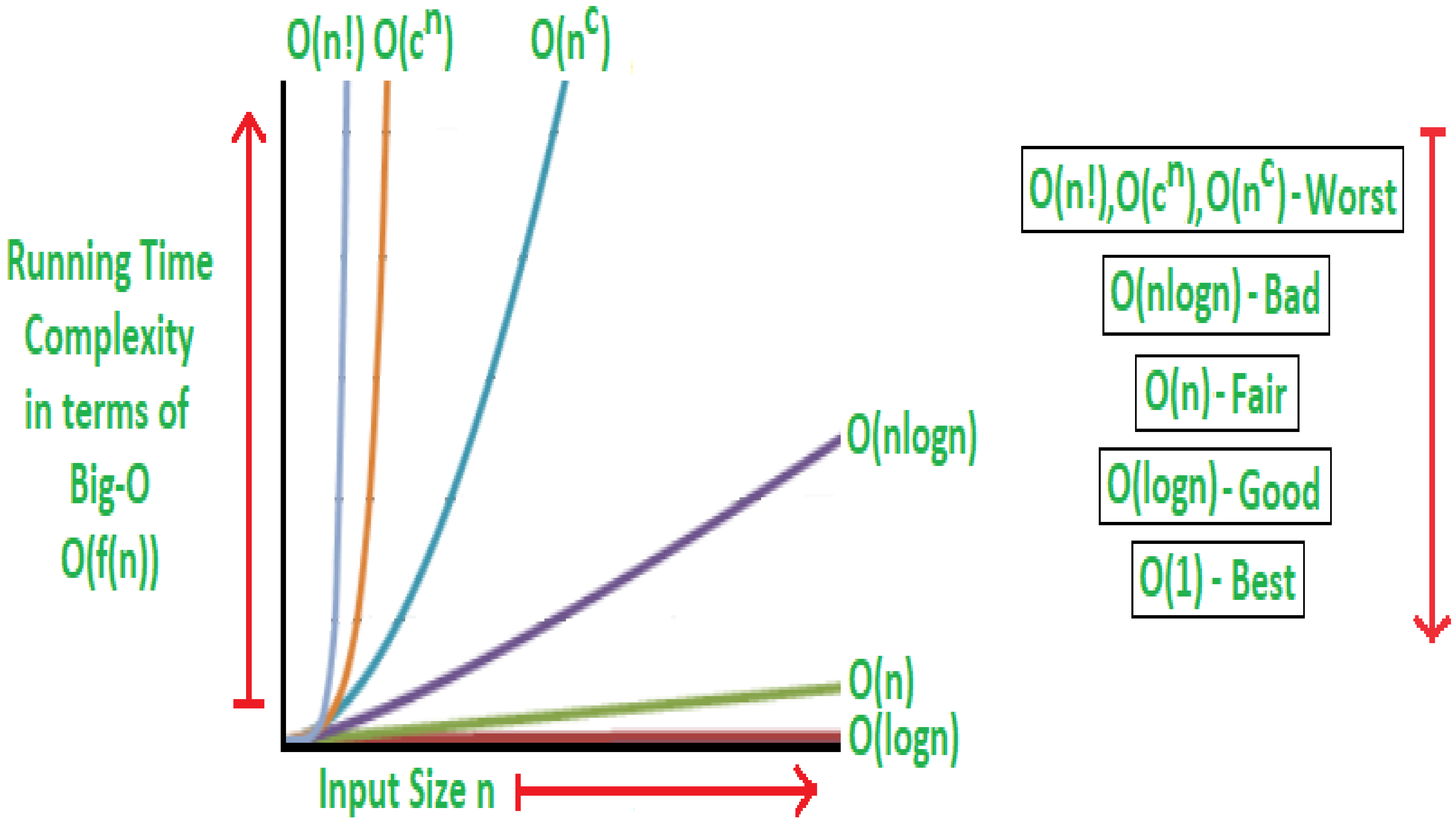
Session : Day4

Dr Kiran Waghmare
CDAC Mumbai



The order of growth for all time complexities are indicated in the graph below:





O(n!) O(c^n) O(n^c)
Running Time Complexity in terms of Big-O O(f(n))
O(nlogn)
O(n)
O(logn)
Input Size n
O(n!),O(c^n),O(n^c) - Worst
O(nlogn) - Bad
O(n) - Fair
O(logn) - Good
O(1) - Best

Commonly Used Functions and Their Comparison

1. **Constant Functions** - $f(n) = 1$ - Whatever is the input size n, these functions take a constant amount of time.
2. **Linear Functions** - $f(n) = n$ - These functions grow linearly with the input size n.
3. **Quadratic Functions** - $f(n) = n^2$ - These functions grow faster than the superlinear functions i.e., $n \log(n)$.
4. **Cubic Functions** - $f(n) = n^3$ - Faster growing than quadratic but slower than exponential.
5. **Logarithmic Functions** - $f(n) = \log(n)$ - These are slower growing than even linear functions.
6. **Superlinear Functions** - $f(n) = n \log(n)$ - Faster growing than linear but slower than quadratic.
7. **Exponential Functions** - $f(n) = c^n$ - Faster than all of the functions mentioned here except the factorial functions.
8. **Factorial Functions** - $f(n) = n!$ - Fastest growing than all these functions mentioned here.

```
void factorialTime(int n, boolean[] used, List<Integer> perm) {
    if (perm.size() == n) {
        System.out.println(perm);
        return;
    }

    for (int i = 0; i < n; i++) {
if (!used[i]) {
            used[i] = true;
            perm.add(i);
            factorialTime(n, used, perm);
            perm.remove(perm.size() - 1);
            used[i] = false;
        }
    }
}
```

Factorial Complexity → O(n!)

```
int exponentialTime(int n) {
    if (n <= 1) return n;  // Base case
    return exponentialTime(n - 1) + exponentialTime(n - 2);}
```

Exponential Complexity → $O(2^n)$

```
void linearRecursion(int n) {
    if (n == 0) return;
    System.out.println(n);
    linearRecursion(n - 1);
}
```

Simple Linear Recursion → O(n)

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Binary Recursion → $O(2^n)$

```
void mergeSort(int[] arr, int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Divide and Conquer → O(n log n)

```
void kRecursion(int n, int k) {
    if (n == 0) return;
    for (int i = 0; i < k; i++) {
        kRecursion(n - 1, k);
    }
}
```

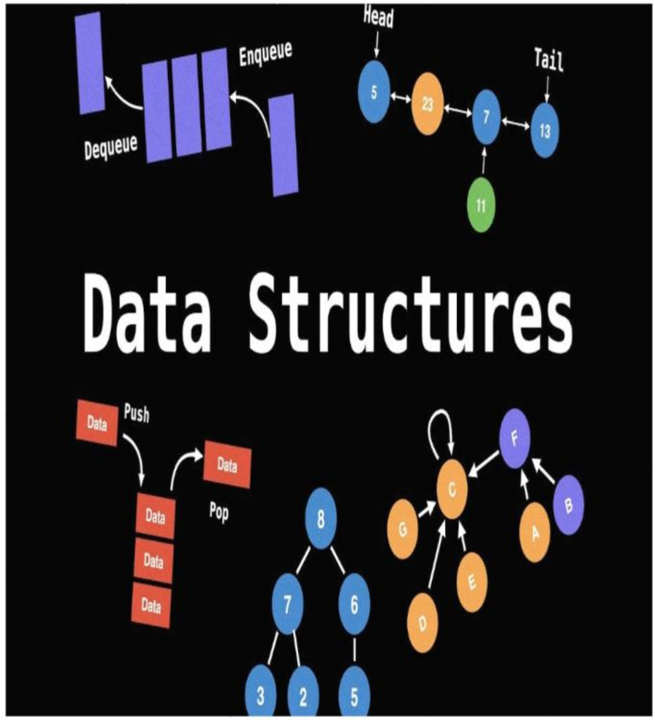
K Recursive Calls → $O(n^k)$

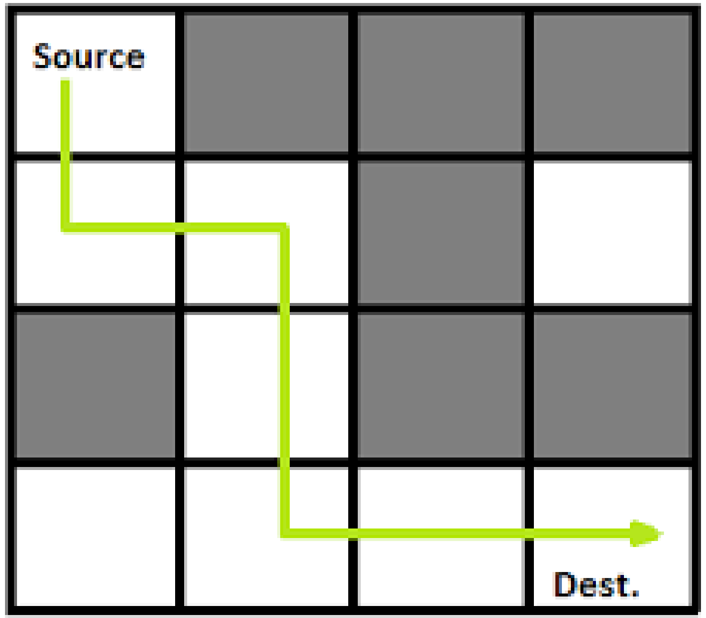
Algorithms and Data Structures

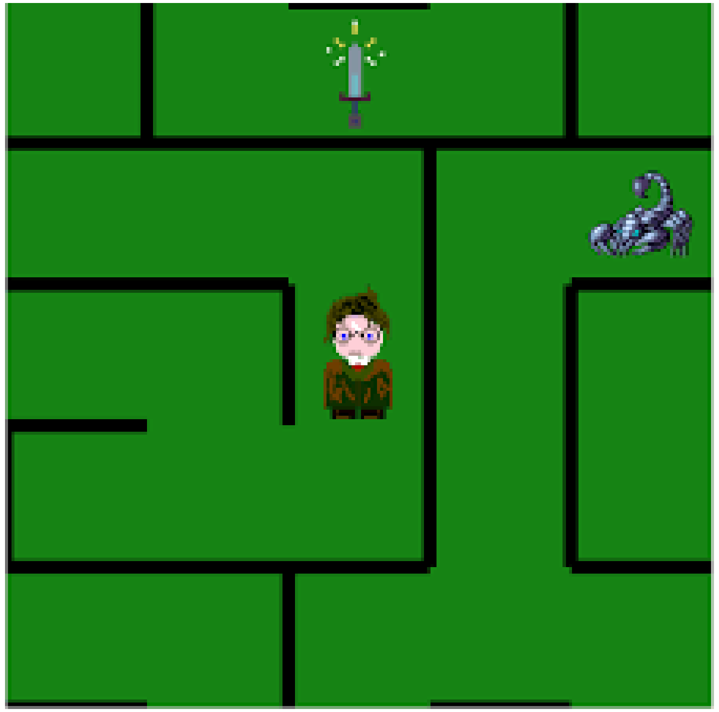
Backtracking

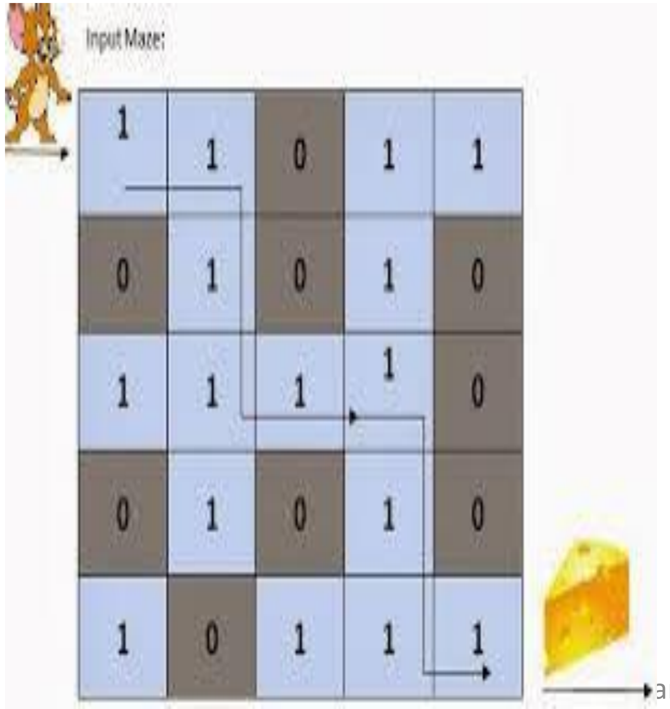
Session : Day3

Dr Kiran Waghmare
CDAC Mumbai



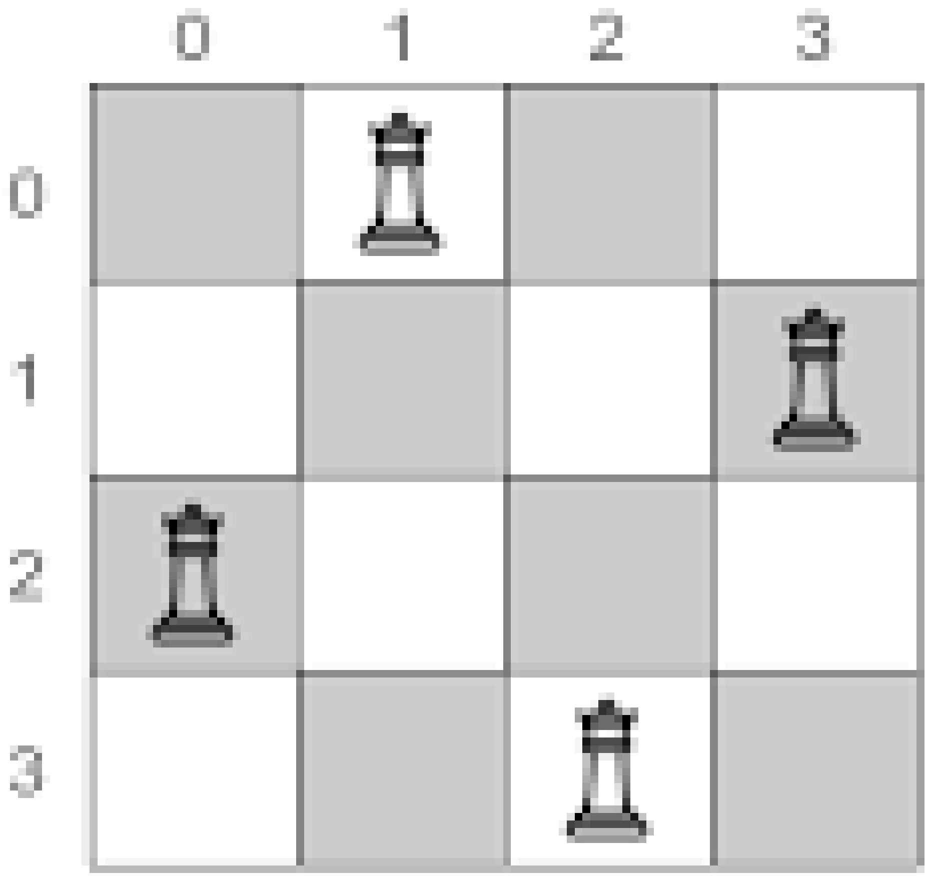

Source
Dest.



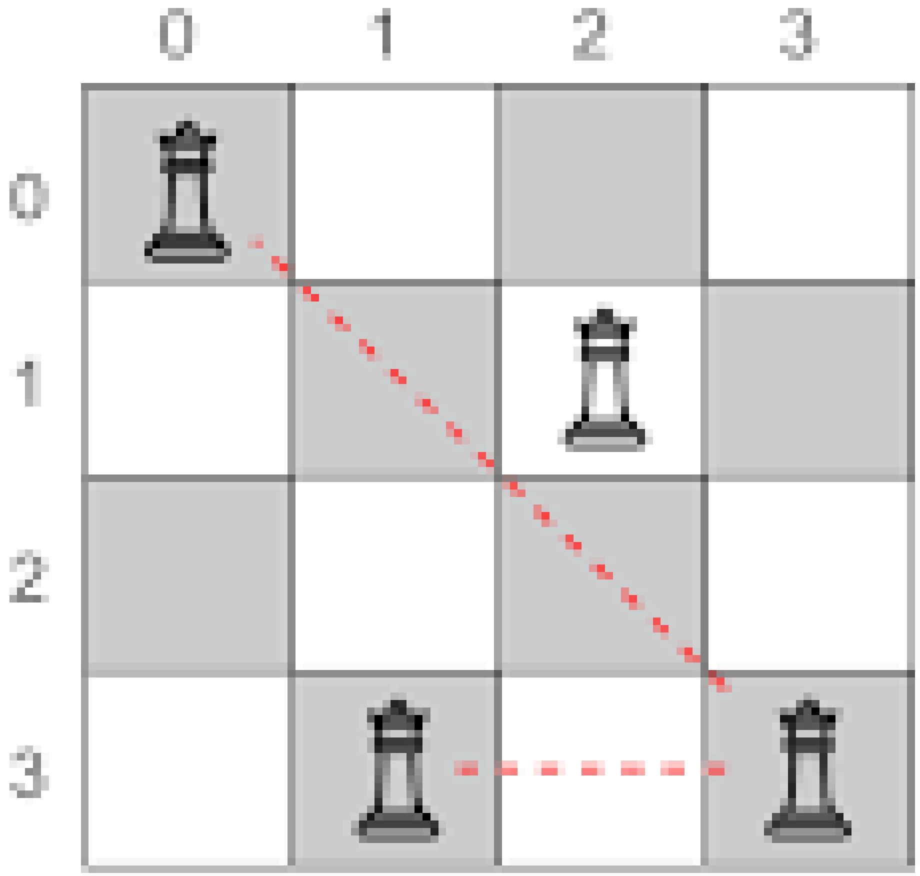

Input Maze:
1 1 0 1 1
0 1 0 1 0
1 1 1 1 0
0 1 0 1 0
1 0 1 1 1


4
20
00:25

are



Valid queen positions



Invalid queen positions

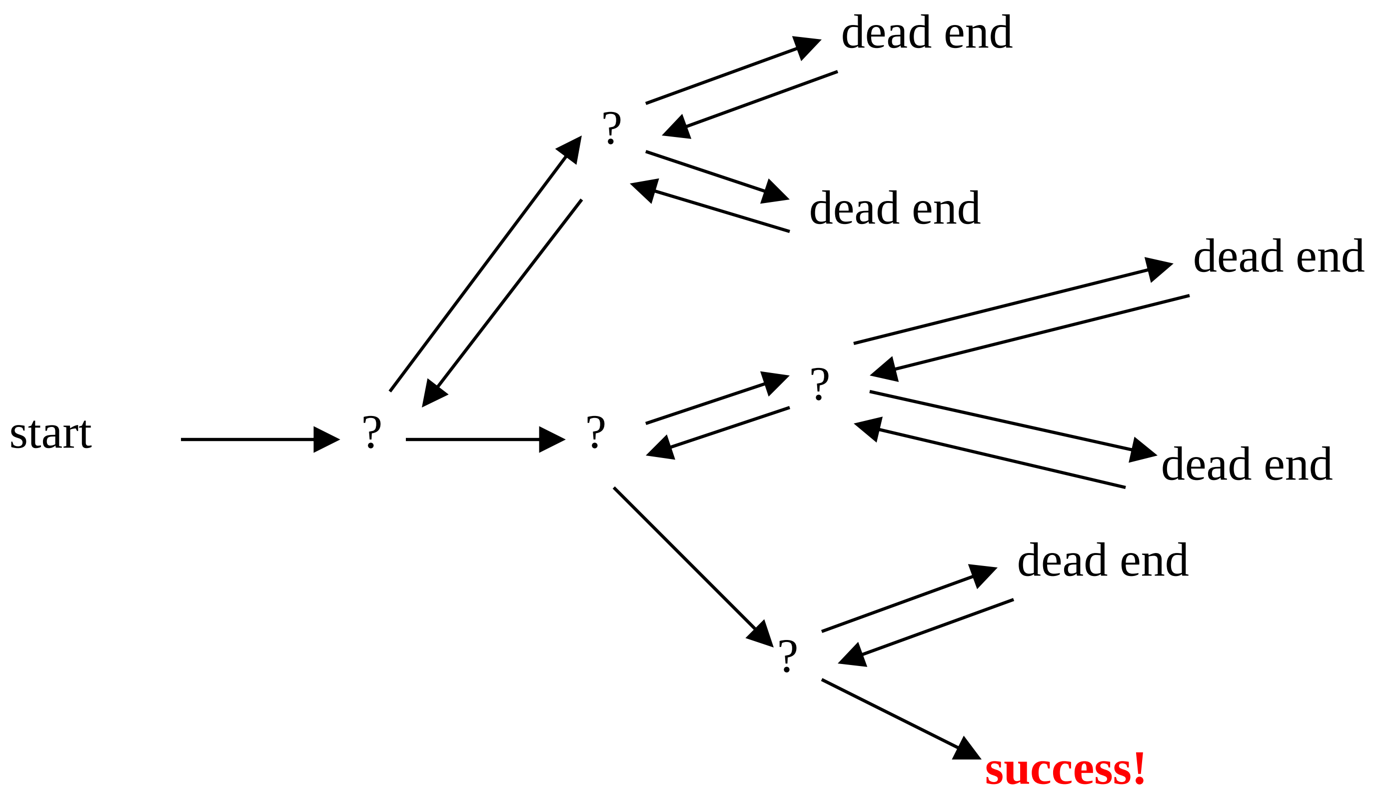
```
def solve_puzzle(game):
  return solved

game = Sudoku()
solve_puzzle(game)
```

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

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

Backtracking (animation)

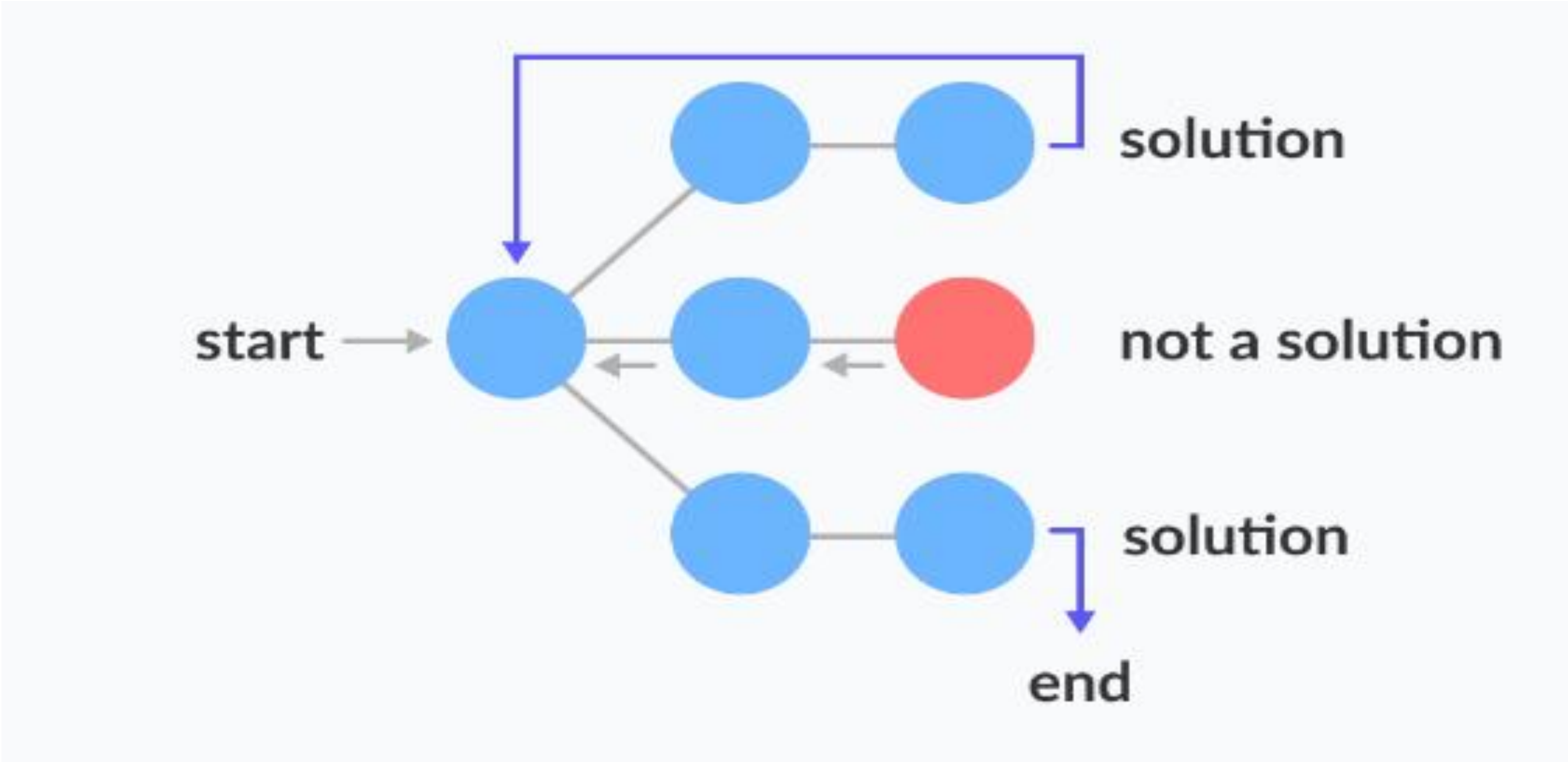


Backtracking

- Backtracking is a **problem-solving technique.**
- It involves systematically **exploring different paths** to find a solution.
- When faced with **multiple choices, backtracking tries each option.**
- It backtracks when it reaches **a dead end**.
- It's akin to navigating through a complex maze.
- **Wrong turns lead to retracing steps until the correct path** is found.
- Backtracking **enables the exploration of various possibilities**.
- It's a **powerful tool for tackling** challenging problems.

State Space Tree

- A space state tree is a tree representing **all the possible states (solution or nonsolution) of the problem** from the root as an initial state to the leaf as a terminal state.



- **Start**: Begin with an initial solution candidate or state.
- **Explore**: Examine all possible next steps or choices from the current state.
- **Constraint check**: Verify whether the current solution candidate satisfies the problem constraints or conditions.
- **Recursion**: If the current candidate satisfies the constraints, recursively explore further by making a choice and moving to the next state.
- **Backtrack**: If the current candidate does not satisfy the constraints, backtrack to the previous state and try a different choice or explore a different path.
- **Repeat**: Repeat steps 2-5 until a valid solution is found or all possible candidates have been explored.

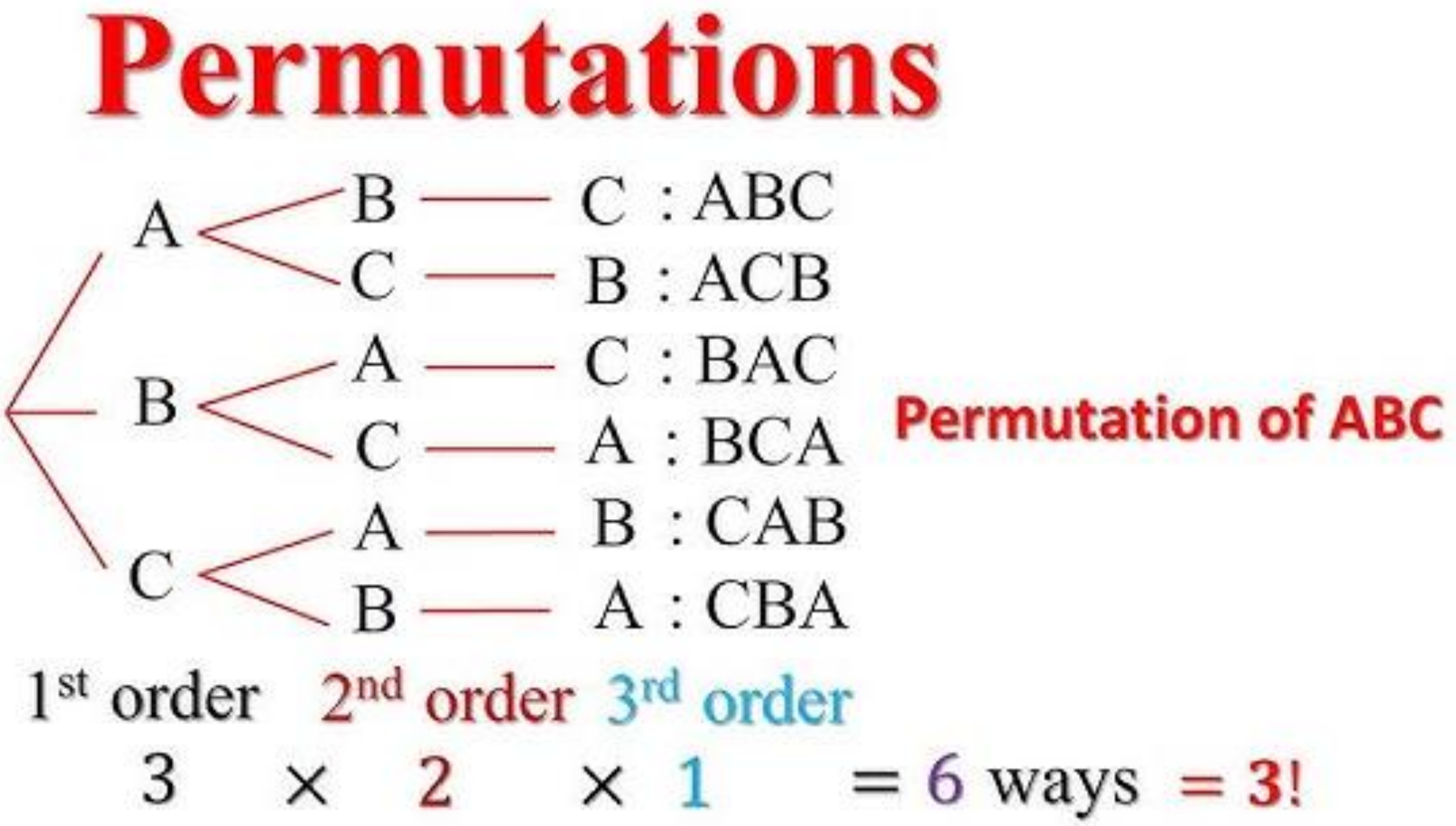
- **Backtracking Algorithm**

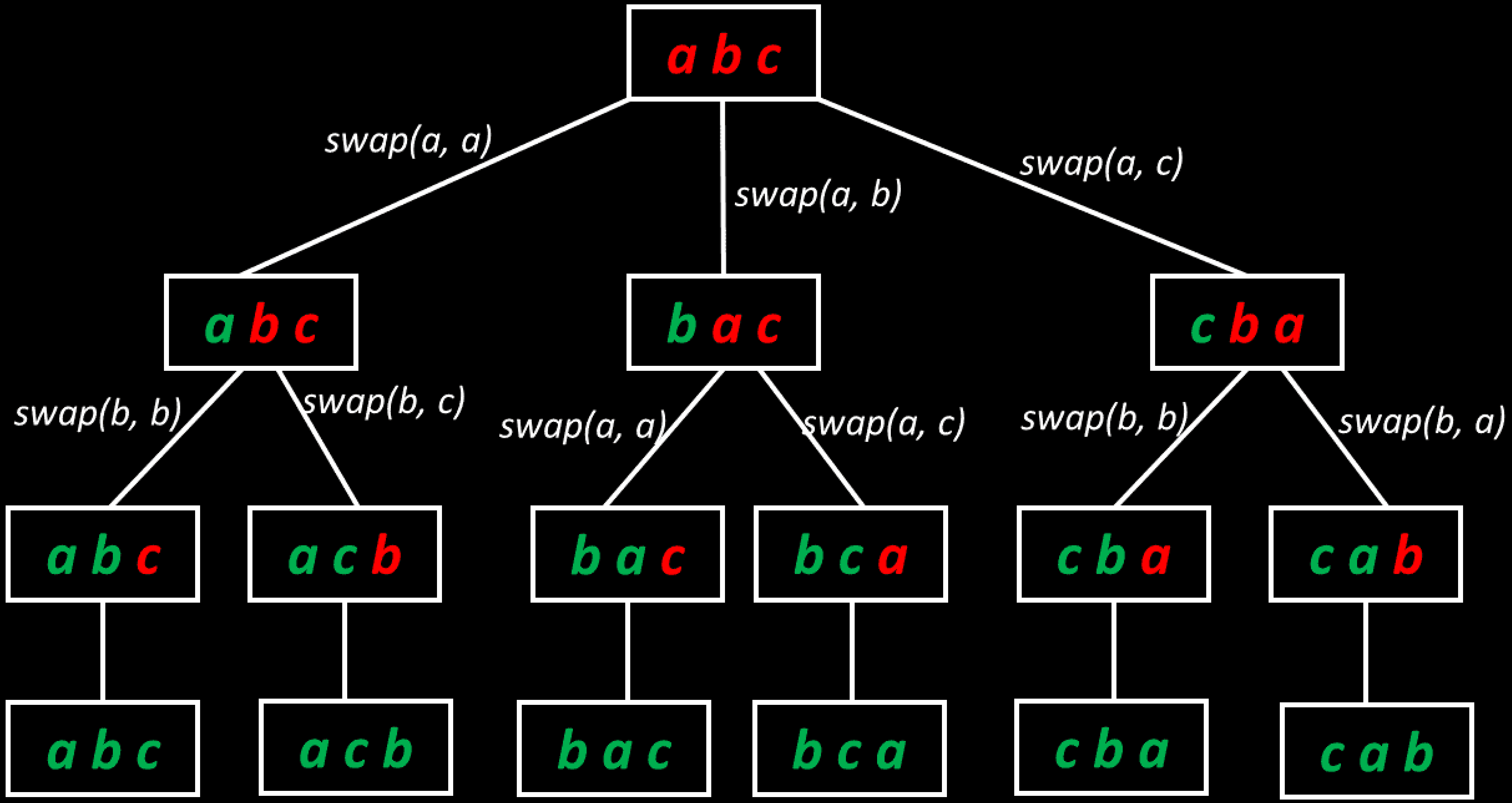
```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
```

- Backtracking **uses recursive algorithms** to explore potential solutions.
- **Each recursive call makes a choice and explores** further.
- If a dead end is reached, the algorithm backtracks to the previous state.

- Backtracking is commonly **used in problem-solving** scenarios.
- It's utilized in **constraint satisfaction problems, combinatorial optimization, sudoku solving, N-queens problem, graph traversal**, etc.

- Backtracking **efficiently searches through a large solution space**.
- It avoids unnecessary computations by pruning branches that cannot lead to a valid solution.


Permutations
A
B
C
B — C : ABC
C — B : ACB
A — C : BAC
C — A : BCA
A — B : CAB
B — A : CBA
Permutation of ABC
1st order 2nd order 3rd order
3 × 2 × 1 = 6 ways = 3!


a b c
swap(a, a)
swap(a, b)
swap(a, c)
a b c
b a c
c b a
swap(b, b)
swap(b, c)
swap(a, a)
swap(a, c)
swap(b, b)
swap(b, a)
a b c
a c b
b a c
b c a
c b a
c a b
a b c
a c b
b a c
b c a
c b a
c a b