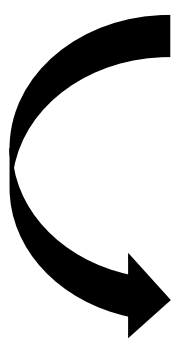


Recursive Backtracking

General Structure of a Back-tracking Algorithm

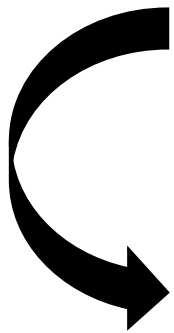
```
class YourClass {  
    Storage for your partial / final solution;  
  
    YourClass(The parameters and constraints of the search);  
  
    bool solve(step index);  
  
    Some function to display / return the result;  
};  
  
bool solve(step index) {  
    if (step is final) return true;  
  
    for (every possible answer to the current step) {  
        Apply this answer for the step to your partial solution;  
        if (check if partial solution is valid) {  
            if (solve(next(stage))) {  
                return true;  
            }  
        }  
        Un-apply the answer from the solution;  
    }  
  
    return false;  
}
```



```
class YourClass {  
    Storage for your partial / final solution;  
    YourClass(The parameters and constraints of the search);  
}
```

```
class QueensSolver {  
private:  
    // The size of the grid we are trying to solve for  
    int gridSize;  
  
    // Storage for intermediate and final result  
    std::vector<int> results;  
  
public:  
    QueensSolver(int gridSize) : gridSize(gridSize) {}  
}
```

```
bool solve(step index) {  
    if (step is final) return true;  
  
    for (every possible answer to the current step) {  
        Apply this answer for the step to your partial solution;  
        if (check if partial solution is valid) {  
            if (solve(next(stage))) {  
                return true;  
            }  
        }  
        Un-apply the answer from the solution;  
    }  
  
    return false;  
}
```



See code on next two slides, or check
<https://godbolt.org/z/Y3WvqPK8e>

```
// Solve for placement of all queens at or below a specific row
bool solveRecursive(int row) {
    // We have finished all rows, return true
    if (row == gridSize) return true;

    // Check each position in this row, check if its threatened by a
    // queen already placed earlier
    for (int column = 0; column < gridSize; ++column) {
        if (canPlaceQueenAt(row, column)) {
            // If not threatened, place it and move to the next row
            results.push_back(column);

            if (solveRecursive(row + 1)) {
                // If placing the rest of the rows also succeeded, return true
                return true;
            }

            // Otherwise un-place the queen and try the next position in this row
            results.pop_back();
        }
    }

    return false;
}
```

```
bool solve(step index) {  
    if (step is final) return true;  
  
    for (every possible answer to the current step) {  
        Apply this answer for the step to your partial solution;  
        if (check if partial solution is valid) {  
            if (solve(next(stage))) {  
                return true;  
            }  
        }  
        Un-apply the answer from the solution;  
    }  
  
    return false;  
}
```

Step index doesn't have to be a single integer (e.g., a row number). It just need to be something that can identify which step you are currently at. Think what you should do for Sudoku.

Sudoku

- A 9-by-9 grid of numbers.
- Each row and column must contain all numbers from 1-9.
- If you divide the 9-by-9 grid into 9 3x3 boxes, they also each must contain all numbers from 1-9.
- Some numbers are given at the start. Others are blank and need to be filled in.

An example
Sudoku
puzzle (and
solution)

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