Backtracking — Coding Interview Notes (Light Theme)

General Pattern Template

```
def backtrack(curr, OTHER_ARGUMENTS...):
    if (BASE_CASE):
        # modify the answer
        return

    ans = 0
    for (ITERATE_OVER_INPUT):
        # modify the current state
        ans += backtrack(curr, OTHER_ARGUMENTS...)
        # undo the modification of the current state

    return ans
```

Concept:
Backtracking is a general algorithmic technique for exploring all possible configurations of a problem space by building a solution incrementally and abandoning (“backtracking”) as soon as it becomes invalid.

It is commonly used for generating combinations, permutations, solving constraint problems (N-Queens, Sudoku, word search), and optimization with pruning.

Time Complexity: Exponential (varies per problem)
Space Complexity: O(depth of recursion)

Key Ideas

1. Backtracking builds partial solutions step-by-step using recursion.
2. At each recursive call, iterate over choices; make a choice, explore deeper, and then undo the choice.
3. The **BASE_CASE** defines when a valid or complete solution is reached.
4. Use pruning (early exits) to reduce the search space when possible.
5. Typical structure: modify → recurse → undo modification.

Example 1: Generate All Subsets (Power Set)

Goal: Return all possible subsets of the input array.
Approach: At each index, choose to include or exclude the element.

```
def subsets(nums):
```

```
    result = []

    def backtrack(start, path):
        result.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()

    backtrack(0, [])
    return result

# Example
print(subsets([1,2,3]))  # Output: [[], [1], [1,2], [1,2,3], [1,3], [2], [2,3], [3]]
```

Example 2: Generate All Permutations

Goal: Return all possible permutations of the input array.
Approach: At each step, choose a number not yet in the current path.

```
def permute(nums):
    result = []

    def backtrack(path):
        if len(path) == len(nums):
            result.append(path[:])
            return
        for num in nums:
            if num not in path:
                path.append(num)
                backtrack(path)
                path.pop()

    backtrack([])
    return result

# Example
print(permute([1,2,3]))  # Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

Example 3: N-Queens Problem

Goal: Place N queens on an N×N chessboard so that no two queens attack each other.
Approach: Try placing queens row by row; backtrack when a conflict occurs.

```
def solve_n_queens(n):
    result = []
    cols = set()
    pos_diagonals = set()  # r + c
    neg_diagonals = set()  # r - c
    board = [["."] * n for _ in range(n)]
```

```
    def backtrack(r):
        if r == n:
            result.append(["".join(row) for row in board])
            return
        for c in range(n):
            if c in cols or (r+c) in pos_diagonals or (r-c) in neg_diagonals:
                continue
            cols.add(c)
            pos_diagonals.add(r+c)
            neg_diagonals.add(r-c)
            board[r][c] = "Q"

            backtrack(r+1)

            cols.remove(c)
            pos_diagonals.remove(r+c)
            neg_diagonals.remove(r-c)
            board[r][c] = "."

    backtrack(0)
    return result

# Example
print(solve_n_queens(4))
```

Summary Table

ProblemChoices per StepBase CaseOutputComplexity SubsetsInclude / Exclude elementReached end of arrayAll subsetsO(2^n) PermutationsPick unused elementslen(path)==nAll permutationsO(n!) N-QueensAll columns in each rowrow==nAll valid boardsO(n!) (approx)