🔄 Backtracking Interview Quick-Check Pattern

A 1-minute cheat sheet to help you master recursion + decision problems under pressure.

---

1. 🧠 When to Use Backtracking?

Ask yourself:

- ❓ Am I asked to **generate all combinations / permutations / subsets**?
- ❓ Do I need to **try all options** to solve something?
- ❓ Does the question ask for **all valid solutions** or **any valid solution**?
- ❓ Am I dealing with **a board/grid**, **recursive decisions**, or **path exploration**?
- ❓ Does the problem sound **combinatorial** or **constraint-based**?

> 💡 Backtracking = **DFS + undo**

---

2. 🧰 Backtracking Patterns by Problem Type

Problem Type	Strategy Used
Permutations	Choose → Explore → Unchoose
Combinations / Subsets	Include / Exclude → Recurse
Word Search / Maze	Move in 4 directions → Mark + unmark
Sudoku / N-Queens	Try value → Check constraints
Restore IPs / Expressions	Backtrack with custom break points
Palindrome Partitioning	Partition if prefix is valid → Recurse

---

3. 🔧 General Backtracking Template

```
function backtrack(path, choices) {
  if (goal reached) {
    result.push([...path]);
    return;
  }

  for (let choice of choices) {
    if (invalid choice) continue;

    // choose
    path.push(choice);

    // explore
    backtrack(path, updated choices);

    // un-choose (backtrack)
    path.pop();
  }
}
```

---

4. ✅ Must-Know Examples

✅ Generate All Subsets (Combinations)

```
function subsets(nums) {
  const result = [];
  function backtrack(start, path) {
    result.push([...path]);
    for (let i = start; i < nums.length; i++) {
      path.push(nums[i]);
      backtrack(i + 1, path);
      path.pop();
    }
  }
```

```
  backtrack(0, []);
  return result;
}
```

✅ Permutations

```
function permute(nums) {
  const result = [];
  function backtrack(path, used) {
    if (path.length === nums.length) {
      result.push([...path]);
      return;
    }
    for (let i = 0; i < nums.length; i++) {
      if (used[i]) continue;
      used[i] = true;
      path.push(nums[i]);
      backtrack(path, used);
      path.pop();
      used[i] = false;
    }
  }
  backtrack([], []);
  return result;
}
```

✅ N-Queens (Constraint-Based)

```
function solveNQueens(n) {
  const result = [], board = Array(n).fill().map(() =>
Array(n).fill('.'));

  function isSafe(r, c) {
    for (let i = 0; i < r; i++) {
      if (board[i][c] === 'Q') return false;
```

```
      if (c - (r - i) >= 0 && board[i][c - (r - i)] === 'Q') return
false;
      if (c + (r - i) < n && board[i][c + (r - i)] === 'Q') return
false;
    }
    return true;
  }

  function backtrack(r) {
    if (r === n) {
      result.push(board.map(row => row.join('')));
      return;
    }
    for (let c = 0; c < n; c++) {
      if (!isSafe(r, c)) continue;
      board[r][c] = 'Q';
      backtrack(r + 1);
      board[r][c] = '.';
    }
  }

  backtrack(0);
  return result;
}
```

---

5. 🧱 Edge Cases to Watch For

- Infinite recursion (missing base case!)
- Using the same element twice (track `used`)
- Deep recursion (stack overflow risk)
- Constraints not applied early → **TLE**
- Return vs collect → `return true` for early exit, `push()` for all

---

6. 🧠 Mental Model

Think of backtracking as:

1. **Explore every path**
2. **Reject invalid paths early**
3. **Undo your choice before trying the next one**

Step	Metaphor
Choose	Take a step
Explore	Move deeper
Un-choose	Backtrack, undo the decision

✅ Final Backtracking Interview Checklist

- Did I define a **clear base case**?
- Am I **tracking used elements** or constraints?
- Do I **backtrack properly** after recursive calls?
- Am I pruning paths early to avoid TLE?
- Do I return a value or collect answers?