Recursive backtracking

1 What is Recursive Backtracking?

Recursive backtracking uses recursion to explore solutions and abandons (backtracks) partial solutions when they fail.

Goals:

- Determine whether a solution exists
- Find a solution
- Find the best solution
- Count the number of solutions
- Print or find all solutions

Applications:

- Puzzle solving (e.g. Sudoku, crosswords)
- Game playing (e.g. Chess, solitaire)
- Constraint satisfaction (e.g. scheduling, matching)

2 The Recursion Checklist

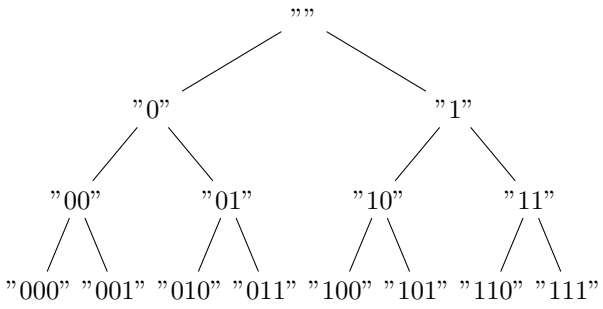
- What information do we track?
- What is the base case?
- What is the recursive step?
- Are all inputs handled?

3 The Backtracking Checklist

- What are our choices at each step?
- Make a choice and recurse
- Undo the choice
- What happens at the base case?

—

4 Example: printAllBinary




```
void printAllBinary(int numDigits) {
    printAllBinaryHelper(numDigits, "");
}

void printAllBinaryHelper(int digits, string soFar) {
    if (digits == 0) {
        cout << soFar << endl;
    } else {
        printAllBinaryHelper(digits - 1, soFar + "0");
        printAllBinaryHelper(digits - 1, soFar + "1");
    }
}
```

5 Example: Dice Rolls

The dice rolls problem asks us to generate and print all possible sequences of rolls of a given number of dice, where each die has 6 sides. This is a classic recursive backtracking problem where:

- Each recursive call represents choosing a value for one die.
- We explore all possibilities by trying values 1 through 6 at each level of recursion.
- After exploring, we backtrack by undoing the choice to try alternatives.

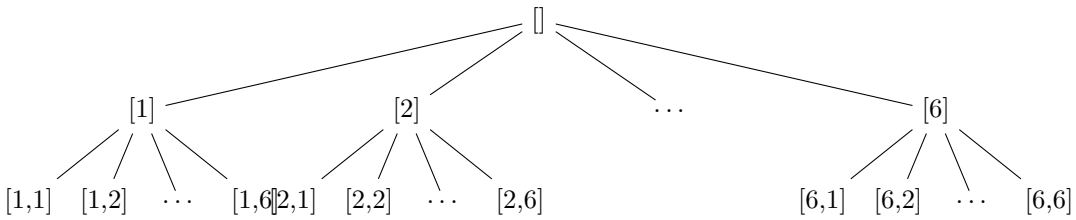
The total number of sequences for rolling n dice is 6^n.

Code for Dice Rolls

```
void diceRolls(int dice) {
    Vector<int> chosen;
    diceRollHelper(dice, chosen);
}

void diceRollHelper(int dice, Vector<int>& chosen) {
    if (dice == 0) {
        cout << chosen << endl;
    } else {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);
            diceRollHelper(dice - 1, chosen);
            chosen.remove(chosen.size() - 1);
        }
    }
}
```

Call tree (2 digits):



6 Example: Dice Roll Sum

The dice roll sum problem is a refinement of the dice rolls problem. Here, we want to generate all possible combinations of rolls of n dice that sum to a specified target value.

Recursive backtracking works well:

- At each step, choose a die value (1 through 6).
- Track the current sum of the chosen values.
- Prune the search space: if the current sum + minimum possible future sum exceeds the target or cannot reach the target, stop early.
- Backtrack by removing the last choice.

This problem demonstrates how backtracking can efficiently prune unnecessary recursive calls by applying constraints at each level.

```
void diceSum(int dice, int desiredSum) {
    Vector<int> chosen;
    diceSumHelper(dice, 0, desiredSum, chosen);
}

void diceSumHelper(int dice, int sum, int desiredSum, Vector<int>& chosen) {
    if (dice == 0) {
        if (sum == desiredSum) {
            cout << chosen << endl;
        }
    } else if (sum + 1 * dice <= desiredSum && sum + 6 * dice >= desiredSum) {
        for (int i = 1; i <= 6; i++) {
            chosen.add(i);
            diceSumHelper(dice - 1, sum + i, desiredSum, chosen);
            chosen.remove(chosen.size() - 1);
        }
    }
}
```

7 Example: Permute Vector

The permute vector problem asks us to generate and print all permutations of a given vector (or list) of elements.

In recursive backtracking:

- At each step, choose an element from the remaining list.
- Remove the chosen element from the remaining list and add it to the current permutation.
- Recurse on the smaller list.
- After recursion, undo the choice by removing it from the permutation and restoring it to the list.

The total number of permutations of a list of n elements is $n!$.

Example Permutation Table for {a, b, c, d}

$\{a, b, c, d\}$	$\{b, a, c, d\}$	$\{c, a, b, d\}$	$\{d, a, b, c\}$
$\{a, b, d, c\}$	$\{b, a, d, c\}$	$\{c, a, d, b\}$	$\{d, a, c, b\}$
$\{a, c, b, d\}$	$\{b, c, a, d\}$	$\{c, b, a, d\}$	$\{d, b, a, c\}$
$\{a, c, d, b\}$	$\{b, c, d, a\}$	$\{c, b, d, a\}$	$\{d, b, c, a\}$
$\{a, d, b, c\}$	$\{b, d, a, c\}$	$\{c, d, a, b\}$	$\{d, c, a, b\}$
$\{a, d, c, b\}$	$\{b, d, c, a\}$	$\{c, d, b, a\}$	$\{d, c, b, a\}$

```
void permute(Vector<string>& v) {
    Vector<string> chosen;
    permuteHelper(v, chosen);
}

void permuteHelper(Vector<string>& v, Vector<string>& chosen) {
    if (v.isEmpty()) {
        cout << chosen << endl;
    } else {
        for (int i = 0; i < v.size(); i++) {
            string s = v[i];
            v.remove(i);
            chosen.add(s);
            permuteHelper(v, chosen);
            chosen.remove(chosen.size() - 1);
            v.insert(i, s);
        }
    }
}
```

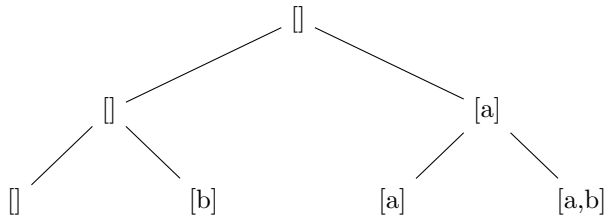
8 Example: Sublists

The sublists problem involves generating all possible subsets (or sublists) of a given list or vector. This is a classic recursive backtracking problem where, at each step, we decide whether to include or exclude each element. The total number of sublists for a list of size n is 2^n, as each element has two choices: include it or not.

```
void sublists(Vector<string>& v) {
    Vector<string> chosen;
    sublistsHelper(v, 0, chosen);
}

void sublistsHelper(Vector<string>& v, int i, Vector<string>& chosen) {
    if (i >= v.size()) {
        cout << chosen << endl;
    } else {
        sublistsHelper(v, i + 1, chosen);
        chosen.add(v[i]);
        sublistsHelper(v, i + 1, chosen);
        chosen.remove(chosen.size() - 1);
    }
}
```

Call tree ([a, b]):



—

9 Example: 8 Queens

The 8 Queens problem asks us to place 8 queens on a standard 8×8 chessboard so that no two queens attack each other. That is, no two queens share the same row, column, or diagonal.

Recursive backtracking is well-suited for this problem. We try placing a queen in each column (or row) and recursively attempt to solve the smaller subproblem. If a conflict is detected, we backtrack and try a different position.

```
bool solveQueens(Board& board) {
    return solveHelper(board, 0);
}

bool solveHelper(Board& board, int col) {
    if (!board.isValid()) {
        return false;
    } else if (col >= board.size()) {
        return true;
    } else {
        for (int row = 0; row < board.size(); row++) {
            board.place(row, col);
            if (solveHelper(board, col + 1)) {
                return true;
            }
            board.remove(row, col);
        }
    }
    return false;
}
```

Example board states (4x4 shown for clarity):

Initial empty board:

_	_	_	_
_	_	_	_
_	_	_	_
_	_	_	_

After placing (0,0), (1,2):

Q	_	_	_
_	_	Q	_
_	_	_	_
_	_	_	_

Final solution:

Q	_	_	_
_	_	Q	_
_	Q	_	_
_	_	_	Q