Complete Recursion & Backtracking Handbook (C++)

Table of Contents

What is Recursion?

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, similar subproblems. It's like solving a big puzzle by solving smaller versions of the same puzzle first.

Key Components of Recursion

1. **Base Case**: The condition that stops the recursion
2. **Recursive Case**: The function calling itself with modified parameters
3. **Progress**: Each recursive call should move closer to the base case.

```
#include <iostream>
using namespace std;
void simpleRecursion(int n) {
    // Base case - stops the recursion
    if (n <= 0) {
        return;
    }
    // Print current number
    cout << n << " ";
    // Recursive case - call itself with smaller value
    simpleRecursion(n - 1);  // Progress: n becomes smaller
}
```

Important Terms & Concepts

1. Base Case

- The terminating condition that prevents infinite recursion

- **Analogy**: Like the ground floor in an elevator - you stop going down

2. Recursive Case

- The part where the function calls itself
- **Analogy**: Like looking into two mirrors facing each other

3. Call Stack

- Memory structure that keeps track of function calls
- **Analogy**: Like a stack of plates - last one on, first one off

4. Stack Overflow

- Error when call stack becomes too deep
- **Analogy**: Like stacking too many books until they fall over

5. Tail Recursion

- When the recursive call is the last operation in the function
- More memory efficient in some languages

6. Tree Recursion

- When a function makes multiple recursive calls
- **Analogy**: Like a family tree branching out

Real-Life Analogies

1. Russian Nesting Dolls (Matryoshka)

Open a doll:

- If it's the smallest doll (base case): Stop

- Otherwise: Open the next smaller doll inside (recursive case)

2. Searching for Keys

To find your keys in a messy room:

- If room has no more places to search (base case): Keys not here

- Otherwise: Search one area, then search the rest of the room (recursive case)

3. Making Photocopies

To make 100 copies:

- If you need 0 copies (base case): Stop

- Otherwise: Make 1 copy, then make (n-1) more copies (recursive case)

4. Factorial Explanation

5! = "How many ways can 5 people line up?"

- Person 1 can be in any of 5 positions

- For each position of Person 1, the remaining 4 people can arrange in 4! ways

- So 5! = 5 × 4!

How Recursion Works

Call Stack Visualization :

```
int factorial(int n) {
   if (n <= 1) {
      return 1;
   }
   return n * factorial(n - 1);
}


// factorial(3) execution:
```

Step 1: factorial(3) calls factorial(2)

Step 2: factorial(2) calls factorial(1)

Step 3: factorial(1) returns 1

Step 4: factorial(2) returns 2 × 1 = 2

Step 5: factorial(3) returns 3 × 2 = 6

Basic Recursion Problems

---

1. Factorial

```
#include <iostream>
using namespace std;
int factorial(int n) {
    // Base case - smallest problem we can solve directly
    if (n <= 1) {
        return 1;
    }
    // Recursive case - break into smaller problem
    return n * factorial(n - 1);
}
int main() {
    int num = 5;
    cout << "Factorial of " << num << " is: " << factorial(num) << endl;
    return 0;
}
```

```
// Time Complexity: O(n)
// Space Complexity: O(n) - due to call stack
```

---

2. Fibonacci

```
#include <iostream>
using namespace std;

// Simple recursive approach (inefficient)
int fibonacci(int n) {
    // Base cases
    if (n <= 1) {
        return n;}
    // Recursive case
    return fibonacci(n - 1) + fibonacci(n - 2);}
// Optimized version with memoization
int fibonacciMemo(int n, int memo[]) {
    // Check if already calculated
    if (memo[n] != -1) {
        return memo[n];}
    // Base cases
    if (n <= 1) {
        return n;}
    // Calculate and store result
    memo[n] = fibonacciMemo(n - 1, memo) + fibonacciMemo(n - 2, memo);
```

```
    return memo[n];
}
int main() {
    int n = 10;
    // Simple version - Time: O(2^n), Space: O(n)
    cout << "Fibonacci (simple): " << fibonacci(n) << endl;
    // Optimized version - Time: O(n), Space: O(n)
    int memo[100];
    for (int i = 0; i < 100; i++) memo[i] = -1;  // Initialize with -1
    cout << "Fibonacci (memoized): " << fibonacciMemo(n, memo) << endl;
    return 0;
}
```

---

3. Sum of Array

```
#include <iostream>
#include <vector>
using namespace std;

int arraySum(vector<int>& arr, int index = 0) {
    // Base case - reached end of array
    if (index >= arr.size()) {
        return 0;
    }
```

```
    // Recursive case - current element + sum of rest
    return arr[index] + arraySum(arr, index + 1);
}

// Alternative approach - reducing array size
int arraySumV2(vector<int> arr) {
    // Base case - empty array
    if (arr.empty()) {
        return 0;
    }

    // Recursive case - first element + sum of remaining elements
    int first = arr[0];
    vector<int> remaining(arr.begin() + 1, arr.end());
    return first + arraySumV2(remaining);
}

int main() {
    vector<int> arr = {1, 2, 3, 4, 5};
    cout << "Array sum: " << arraySum(arr) << endl;
    return 0;
}
```

---

4. Power Function

```
#include <iostream>
using namespace std;


// Optimized power function using divide and conquer
int power(int base, int exp) {
    // Base cases
    if (exp == 0) {
        return 1;}
    if (exp == 1) {
        return base;}
    // Recursive case - divide and conquer approach
    if (exp % 2 == 0) {
        // Even exponent: base^exp = (base^(exp/2))^2
        int half = power(base, exp / 2);
        return half * half;
    } else {
        // Odd exponent: base^exp = base * base^(exp-1)
        return base * power(base, exp - 1);
    }
}
int main() {
```

```
    int base = 2, exp = 10;
    cout << base << "^" << exp << " = " << power(base, exp) << endl;
    return 0;}
// Time Complexity: O(log n)
// Space Complexity: O(log n)
```

---

5. Reverse String

```
#include <iostream>
#include <string>
using namespace std;
string reverseString(string s) {
    // Base case - string with 0 or 1 character
    if (s.length() <= 1) {
        return s;
    }
    // Recursive case - last character + reverse of remaining string
    return s[s.length() - 1] + reverseString(s.substr(0, s.length() - 1));
}

// Alternative approach using indices
void reverseStringInPlace(string& s, int start = 0, int end = -1) {
    if (end == -1) {
        end = s.length() - 1;
```

```
    }
    // Base case - pointers meet or cross
    if (start >= end) {
        return;
    }
    // Swap characters and recurse
    swap(s[start], s[end]);
    reverseStringInPlace(s, start + 1, end - 1);
}
int main() {
    string str = "hello";
    cout << "Original: " << str << endl;
    cout << "Reversed: " << reverseString(str) << endl;

    string str2 = "world";
    reverseStringInPlace(str2);
    cout << "In-place reversed: " << str2 << endl;

    return 0;}
```

Intermediate Problems

1. Binary Search

```
#include <iostream>
#include <vector>
using namespace std;
int binarySearch(vector<int>& arr, int target, int left = 0, int right = -1) {
    // Initialize right boundary if not set
    if (right == -1) {
        right = arr.size() - 1;
    }
    // Base case - element not found
    if (left > right) {
        return -1;
    }
    int mid = left + (right - left) / 2;  // Avoid overflow
    // Found the target
    if (arr[mid] == target) {
        return mid;
    }
    // Search in right half
    if (arr[mid] < target) {
        return binarySearch(arr, target, mid + 1, right);
    }
    // Search in left half
    else {
```

```
        return binarySearch(arr, target, left, mid - 1);
    }
}
int main() {
    vector<int> arr = {1, 3, 5, 7, 9, 11, 13, 15};
    int target = 7;

    int result = binarySearch(arr, target);
    if (result != -1) {
        cout << "Element found at index: " << result << endl;
    } else {
        cout << "Element not found" << endl;
    }
    return 0;
}
// Time Complexity: O(log n)
// Space Complexity: O(log n)
```

---

2. Generate Parentheses

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
```

```
void generateParentheses(int n, string current, int open, int close, vector<string>&
result) {
   // Base case - we have used all parentheses
   if (current.length() == 2 * n) {
      result.push_back(current);
      return;
   }
   // Add opening parenthesis if we haven't used all
   if (open < n) {
      generateParentheses(n, current + "(", open + 1, close, result);
   }
   // Add closing parenthesis if it won't make string invalid
   if (close < open) {
      generateParentheses(n, current + ")", open, close + 1, result);
   }
}

vector<string> generateParenthesesWrapper(int n) {
   vector<string> result;
   generateParentheses(n, "", 0, 0, result);
   return result;
}
```

```
int main() {
    int n = 3;
    vector<string> result = generateParenthesesWrapper(n);

    cout << "All valid parentheses combinations for n = " << n << ":" << endl;
    for (const string& s : result) {
        cout << s << endl;
    }
    return 0;
}
// Time Complexity: O(4^n / √n)
// Space Complexity: O(4^n / √n)
```

---

4. Merge Sort

```
#include <iostream>
#include <vector>
using namespace std;

// Merge two sorted arrays
void merge(vector<int>& arr, int left, int mid, int right) {
    // Create temporary arrays for left and right subarrays
    vector<int> leftArr(arr.begin() + left, arr.begin() + mid + 1);
    vector<int> rightArr(arr.begin() + mid + 1, arr.begin() + right + 1);
```

```
    int i = 0, j = 0, k = left;

    // Merge the two arrays back into arr[left..right]
    while (i < leftArr.size() && j < rightArr.size()) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements
    while (i < leftArr.size()) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    while (j < rightArr.size()) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
```

```
void mergeSort(vector<int>& arr, int left, int right) {
    // Base case - single element or empty array
    if (left >= right) {
        return;
    }
    // Divide - find middle point
    int mid = left + (right - left) / 2;

    // Conquer - recursively sort both halves
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);

    // Combine - merge the sorted halves
    merge(arr, left, mid, right);
}

int main() {
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};

    cout << "Original array: ";
    for (int x : arr) cout << x << " ";
    cout << endl;

    mergeSort(arr, 0, arr.size() - 1);

    cout << "Sorted array: ";
```

```
    for (int x : arr) cout << x << " ";
    cout << endl;

    return 0;
}
```

// Time Complexity: O(n log n)

// Space Complexity: O(n)

---

5. Quick Sort

```
#include <iostream>
#include <vector>
using namespace std;

// Partition function - places pivot in correct position
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];  // Choose last element as pivot
    int i = low - 1;       // Index of smaller element

    for (int j = low; j < high; j++) {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
```

```
    // Place pivot in correct position
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<int>& arr, int low, int high) {
    // Base case - single element or empty array
    if (low < high) {
        // Partition the array and get pivot index
        int pivotIndex = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

int main() {
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};

    cout << "Original array: ";
    for (int x : arr) cout << x << " ";
    cout << endl;

    quickSort(arr, 0, arr.size() - 1);

    cout << "Sorted array: ";
```

```
    for (int x : arr) cout << x << " ";
    cout << endl;

    return 0;
}
```

// Average Time Complexity: O(n log n)

// Worst Time Complexity: O(n²)

// Space Complexity: O(log n)

Backtracking

What is Backtracking?

Backtracking is a refined recursion technique that builds solutions incrementally and abandons partial solutions that cannot lead to a valid complete solution.

Common Backtracking Problems:

1. Subsets (Power Set)

```
#include <iostream>
#include <vector>
using namespace std;

void subsets(vector<int>& nums, int start, vector<int>& current, vector<vector<int>>& result) {
    // Add current subset to result
    result.push_back(current);

    // Try adding each remaining element
    for (int i = start; i < nums.size(); i++) {
```

```
        // Include current element
        current.push_back(nums[i]);

        // Recurse with next starting position
        subsets(nums, i + 1, current, result);

        // Backtrack - remove current element
        current.pop_back();
    }
}
vector<vector<int>> subsetsWrapper(vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> current;
    subsets(nums, 0, current, result);
    return result;
}

int main() {
    vector<int> nums = {1, 2, 3};
    vector<vector<int>> result = subsetsWrapper(nums);

    cout << "All subsets:" << endl;
    for (const vector<int>& subset : result) {
        cout << "[";
        for (int i = 0; i < subset.size(); i++) {
            cout << subset[i];
            if (i < subset.size() - 1) cout << ", ";
```

```
        }
        cout << "]" << endl;
    }
    return 0;
}
```

// Time Complexity: O(2^N × N)

// Space Complexity: O(N)

---

2. Palindrome Partitioning

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

bool isPalindrome(const string& s, int start, int end) {
    while (start < end) {
        if (s[start] != s[end]) {
            return false;
        }
        start++;
        end--;
    }
    return true;
}

void palindromePartition(const string& s, int start, vector<string>& current,
```

```
                    vector<vector<string>>& result) {
    // Base case - processed entire string
    if (start == s.length()) {
        result.push_back(current);
        return;
    }

    // Try all possible end positions
    for (int end = start; end < s.length(); end++) {
        // If current substring is palindrome
        if (isPalindrome(s, start, end)) {
            // Add to current partition
            current.push_back(s.substr(start, end - start + 1));

            // Recurse for remaining string
            palindromePartition(s, end + 1, current, result);

            // Backtrack
            current.pop_back();
        }
    }
}

vector<vector<string>> palindromePartitionWrapper(string s) {
    vector<vector<string>> result;
    vector<string> current;
    palindromePartition(s, 0, current, result);
```

```
    return result;
}

int main() {
    string s = "aab";
    vector<vector<string>> result = palindromePartitionWrapper(s);

    cout << "Palindrome partitions of '" << s << "':" << endl;
    for (const vector<string>& partition : result) {
        cout << "[";
        for (int i = 0; i < partition.size(); i++) {
            cout << "\"" << partition[i] << "\"";
            if (i < partition.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }
    return 0;
}
```

---

3. Letter Combinations of Phone Number

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
using namespace std;

void letterCombinations(const string& digits, int index, string& current,
```

```
                const unordered_map<char, string>& phone, vector<string>& result) {
    // Base case - processed all digits
    if (index == digits.length()) {
        result.push_back(current);
        return;
    }

    // Get letters for current digit
    char digit = digits[index];
    const string& letters = phone.at(digit);

    // Try each letter
    for (char letter : letters) {
        current.push_back(letter);          // Choose
        letterCombinations(digits, index + 1, current, phone, result);  // Recurse
        current.pop_back();                 // Backtrack
    }
}

vector<string> letterCombinationsWrapper(string digits) {
    if (digits.empty()) return {};

    unordered_map<char, string> phone = {
        {'2', "abc"}, {'3', "def"}, {'4', "ghi"}, {'5', "jkl"},
        {'6', "mno"}, {'7', "pqrs"}, {'8', "tuv"}, {'9', "wxyz"}
    };
```

```
    vector<string> result;
    string current;
    letterCombinations(digits, 0, current, phone, result);
    return result;
}

int main() {
    string digits = "23";
    vector<string> result = letterCombinationsWrapper(digits);

    cout << "Letter combinations of '" << digits << "':" << endl;
    for (const string& combination : result) {
        cout << combination << " ";
    }
    cout << endl;

    return 0;
}
```

Common Patterns & Techniques

1. Divide and Conquer

- Break problem into smaller subproblems
- Solve subproblems recursively
- Combine solutions

Template:

```
// Template
int divideAndConquer(problem) {
    // Base case
    if (problem is small enough) {
        return solve directly;
    }

    // Divide
    subproblem1 = divide(problem);
    subproblem2 = divide(problem);

    // Conquer
    solution1 = divideAndConquer(subproblem1);
    solution2 = divideAndConquer(subproblem2);

    // Combine
    return combine(solution1, solution2);
}
```

Advanced Problems

1. N-Queens Problem

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

bool isSafe(vector<string>& board, int row, int col, int n) {
    // Check column
```

```
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 'Q') {
            return false;
        }
    }
    // Check diagonal (top-left to bottom-right)
    for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 'Q') {
            return false;
        }
    }
    // Check diagonal (top-right to bottom-left)
    for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
        if (board[i][j] == 'Q') {
            return false;
        }
    }

    return true;
}

void solveNQueens(vector<string>& board, int row, int n, vector<vector<string>>& result) {
    // Base case - all queens are placed
    if (row == n) {
        result.push_back(board);
        return;
    }
```

```
    // Try placing queen in each column of current row
    for (int col = 0; col < n; col++) {
        if (isSafe(board, row, col, n)) {
            // Place queen
            board[row][col] = 'Q';

            // Recursively place rest of the queens
            solveNQueens(board, row + 1, n, result);

            // Backtrack - remove queen
            board[row][col] = '.';
        }
    }
}

vector<vector<string>> solveNQueensWrapper(int n) {
    vector<vector<string>> result;
    vector<string> board(n, string(n, '.'));
    solveNQueens(board, 0, n, result);
    return result;
}

void printBoard(const vector<string>& board) {
    for (const string& row : board) {
        cout << row << endl;
    }
```

```
    cout << endl;
}

int main() {
    int n = 4;
    vector<vector<string>> solutions = solveNQueensWrapper(n);

    cout << "Total solutions for " << n << "-Queens: " << solutions.size() << endl << endl;

    for (int i = 0; i < solutions.size(); i++) {
        cout << "Solution " << (i + 1) << ":" << endl;
        printBoard(solutions[i]);
    }

    return 0;
}
```

---

2. Combination Sum

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void combinationSum(vector<int>& candidates, int target, int start,
                    vector<int>& current, vector<vector<int>>& result) {
    // Base case - found valid combination
    if (target == 0) {
```

```
        result.push_back(current);
        return;
    }

    // Base case - target became negative
    if (target < 0) {
        return;
    }

    // Try each candidate starting from 'start' index
    for (int i = start; i < candidates.size(); i++) {
        // Include current candidate
        current.push_back(candidates[i]);

        // Recurse with same start index (can reuse same element)
        combinationSum(candidates, target - candidates[i], i, current, result);

        // Backtrack - remove current candidate
        current.pop_back();
    }
}

vector<vector<int>> combinationSumWrapper(vector<int>& candidates, int target) {
    vector<vector<int>> result;
    vector<int> current;
    sort(candidates.begin(), candidates.end());  // Sort for optimization
    combinationSum(candidates, target, 0, current, result);
```

```
    return result;
}

void printCombinations(const vector<vector<int>>& combinations) {
    for (const vector<int>& combination : combinations) {
        cout << "[";
        for (int i = 0; i < combination.size(); i++) {
            cout << combination[i];
            if (i < combination.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }
}

int main() {
    vector<int> candidates = {2, 3, 6, 7};
    int target = 7;

    vector<vector<int>> result = combinationSumWrapper(candidates, target);

    cout << "Combinations that sum to " << target << ":" << endl;
    printCombinations(result);

    return 0;
}
```

---

3. Permutations

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void permutations(vector<int>& nums, vector<int>& current,
                  vector<bool>& used, vector<vector<int>>& result) {
    // Base case - permutation is complete
    if (current.size() == nums.size()) {
        result.push_back(current);
        return;
    }

    // Try each unused number
    for (int i = 0; i < nums.size(); i++) {
        if (!used[i]) {
            // Choose
            current.push_back(nums[i]);
            used[i] = true;

            // Recurse
            permutations(nums, current, used, result);

            // Backtrack
            current.pop_back();
            used[i] = false;
        }
```

```
    }
}

vector<vector<int>> permutationsWrapper(vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> current;
    vector<bool> used(nums.size(), false);
    permutations(nums, current, used, result);
    return result;
}

// Alternative approach using swapping
void permutationsSwap(vector<int>& nums, int start, vector<vector<int>>& result) {
    // Base case
    if (start == nums.size()) {
        result.push_back(nums);
        return;
    }

    for (int i = start; i < nums.size(); i++) {
        // Swap current element with start
        swap(nums[start], nums[i]);

        // Recurse for remaining elements
        permutationsSwap(nums, start + 1, result);

        // Backtrack - swap back
```

```
        swap(nums[start], nums[i]);
    }
}

void printPermutations(const vector<vector<int>>& permutations) {
    for (const vector<int>& perm : permutations) {
        cout << "[";
        for (int i = 0; i < perm.size(); i++) {
            cout << perm[i];
            if (i < perm.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }
}

int main() {
    vector<int> nums = {1, 2, 3};

    cout << "Method 1 - Using extra space:" << endl;
    vector<vector<int>> result1 = permutationsWrapper(nums);
    printPermutations(result1);

    cout << "\nMethod 2 - Using swapping:" << endl;
    vector<vector<int>> result2;
    permutationsSwap(nums, 0, result2);
    printPermutations(result2);
```

```
    return 0;
}
```

---

4. Word Search

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

bool dfs(vector<vector<char>>& board, string word, int i, int j, int index) {
    // Base case - found the complete word
    if (index == word.length()) {
        return true;
    }

    // Check boundaries and character match
    if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size() ||
        board[i][j] != word[index]) {
        return false;
    }

    // Mark current cell as visited
    char temp = board[i][j];
    board[i][j] = '#';  // Use # to mark visited

    // Explore all 4 directions
    bool found = dfs(board, word, i + 1, j, index + 1) ||  // Down
                 dfs(board, word, i - 1, j, index + 1) ||  // Up
```

```
            dfs(board, word, i, j + 1, index + 1) ||  // Right
            dfs(board, word, i, j - 1, index + 1);    // Left

    // Backtrack - restore original character
    board[i][j] = temp;

    return found;
}

bool wordSearch(vector<vector<char>>& board, string word) {
    for (int i = 0; i < board.size(); i++) {
        for (int j = 0; j < board[0].size(); j++) {
            if (dfs(board, word, i, j, 0)) {
                return true;
            }
        }
    }
    return false;
}

int main() {
    vector<vector<char>> board = {
        {'A', 'B', 'C', 'E'},
        {'S', 'F', 'C', 'S'},
        {'A', 'D', 'E', 'E'}
    };
```

```
    string word1 = "ABCCED";
    string word2 = "SEE";
    string word3 = "ABCB";

    cout << "Word '" << word1 << "' found: " << (wordSearch(board, word1) ? "Yes" : "No") << endl;
    cout << "Word '" << word2 << "' found: " << (wordSearch(board, word2) ? "Yes" : "No") << endl;
    cout << "Word '" << word3 << "' found: " << (wordSearch(board, word3) ? "Yes" : "No") << endl;

    return 0;
}
```

Interview Tips & Tricks

1. Always Start with Base Case

```
int recursiveFunction(/* parameters */) {
    // ALWAYS start with base case
    if (/* termination condition */) {
        return /* base result */;
    }
    // Then handle recursive case
    return /* recursive logic */;
}
```

---

2. Trace Through Small Examples

```
// Example: factorial(3)
// Call stack:
// factorial(3) -> 3 * factorial(2)
```

```
// factorial(2) -> 2 * factorial(1)
// factorial(1) -> 1 (base case)
// Returns: 1 -> 2*1=2 -> 3*2=6
```

---

3. Common Recursion Mistakes to Avoid

```
// ❌ MISTAKE 1: No base case
int badRecursion(int n) {
    return badRecursion(n - 1);  // Will cause stack overflow
}
// ❌ MISTAKE 2: Wrong base case
int badFib(int n) {
    if (n == 0) return 0;  // Missing n == 1 case
    return badFib(n - 1) + badFib(n - 2);  // Will go negative
}
// ❌ MISTAKE 3: No progress toward base case
int badRecursion2(int n) {
    if (n == 0) return 0;
    return badRecursion2(n);  // n never changes
}
// ✅ CORRECT: Proper base case and progress
int goodRecursion(int n) {
    if (n <= 0) return 0;  // Clear base case
    return n + goodRecursion(n - 1);  // Makes progress
}
```

---

4. Time and Space Complexity Analysis

// Fibonacci - Simple Recursion

// Time: O(2^n) - each call makes 2 more calls

// Space: O(n) - maximum depth of call stack

// Fibonacci - With Memoization

// Time: O(n) - each subproblem solved once

// Space: O(n) - memo table + call stack

// Binary Search

// Time: O(log n) - problem size halves each time

// Space: O(log n) - depth of recursion

---

5. When to Use Recursion vs Iteration

Use Recursion When:

// ✅ Tree/Graph problems

```
void traverseTree(TreeNode* root) {
    if (!root) return;
    traverseTree(root->left);
    traverseTree(root->right);
}
```

// ✅ Divide and conquer

```
int binarySearch(vector<int>& arr, int target, int left, int right);
```

// ✅ Backtracking

```
void generatePermutations(vector<int>& nums, int start);
```

Use Iteration When:

```
// ✅ Simple counting/accumulation
int sumIterative(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}


// ✅ When recursion might cause stack overflow
int fibIterative(int n) {
    if (n <= 1) return n;
    int prev = 0, curr = 1;
    for (int i = 2; i <= n; i++) {
        int next = prev + curr;
        prev = curr;
        curr = next;
    }
    return curr;
}
```

---

6. Converting Recursion to Iteration

```
// Recursive approach
int sumRecursive(int n) {
```

```
    if (n <= 0) return 0;
    return n + sumRecursive(n - 1);
}

// Iterative equivalent
int sumIterative(int n) {
    int result = 0;
    while (n > 0) {
        result += n;
        n--;
    }
    return result;
}

// Using stack to simulate recursion
int sumUsingStack(int n) {
    stack<int> st;
    int result = 0;

    while (n > 0) {
        st.push(n);
        n--;
    }

    while (!st.empty()) {
        result += st.top();
        st.pop();
```

```
    }

    return result;
}
```

---

7. Testing Recursion Functions

```
void testRecursionFunction() {
    // Test base cases
    assert(factorial(0) == 1);
    assert(factorial(1) == 1);

    // Test simple cases
    assert(factorial(3) == 6);
    assert(factorial(5) == 120);

    // Test edge cases
    // (depends on implementation)

    cout << "All tests passed!" << endl;
}
```

---

8. Common Interview Question Patterns

Pattern 1: Tree Problems

```
// Maximum depth of binary tree
int maxDepth(TreeNode* root) {
    if (!root) return 0;
```

```
    return 1 + max(maxDepth(root->left), maxDepth(root->right));
}
```

Pattern 2: Array/String Division

```
// Check if string is palindrome
bool isPalindrome(string s, int left, int right) {
    if (left >= right) return true;
    if (s[left] != s[right]) return false;
    return isPalindrome(s, left + 1, right - 1);
}
```

Pattern 3: Backtracking for Combinations

```
// Generate all combinations of size k
void combine(int n, int k, int start, vector<int>& current, vector<vector<int>>& result) {
    if (current.size() == k) {
        result.push_back(current);
        return;
    }

    for (int i = start; i <= n; i++) {
        current.push_back(i);
        combine(n, k, i + 1, current, result);
        current.pop_back();
    }
}
```

---

9. Optimization Strategies

Memoization Template:

```
#include <unordered_map>

unordered_map<string, int> memo;

int recursiveFunction(/* parameters */) {
    // Create unique key from parameters
    string key = /* create key from parameters */;

    // Check if already computed
    if (memo.find(key) != memo.end()) {
        return memo[key];
    }

    // Base case
    if (/* base condition */) {
        return /* base value */;
    }

    // Recursive case
    int result = /* recursive computation */;

    // Store and return result
    memo[key] = result;
    return result;
}
```

---

10. Debug Techniques

```
// Add debug prints to trace execution
int factorial(int n, int depth = 0) {
    // Print current state
    for (int i = 0; i < depth; i++) cout << "  ";
    cout << "factorial(" << n << ")" << endl;
    if (n <= 1) {
        for (int i = 0; i < depth; i++) cout << "  ";
        cout << "return 1" << endl;
        return 1;
    }
    int result = n * factorial(n - 1, depth + 1);
    for (int i = 0; i < depth; i++) cout << "  ";
    cout << "return " << result << endl;

    return result;
}
```

Summary

Key Points for Mastering Recursion:

1. **Understand the Structure**: Every recursive problem has a base case and recursive case
2. **Think Small**: Solve the simplest version first, then build up
3. **Trust the Process**: Assume recursive calls work correctly
4. **Practice Patterns**: Tree traversal, divide & conquer, backtracking

5. **Optimize When Needed**: Use memoization for overlapping subproblems
6. **Know When to Stop**: Some problems are better solved iteratively

Common C++ Features for Recursion:

- vector for dynamic arrays and results
- string for text manipulation
- Pass by reference (&) to avoid copying
- STL algorithms like sort() for preprocessing
- unordered_map for memoization

Interview Success Tips:

- Always start with base case
- Walk through examples manually
- Discuss time/space complexity
- Consider optimizations
- Test with edge cases
- Know iterative alternatives

Remember: Recursion is like a mathematical proof by induction - solve the base case, then show how to build larger solutions from smaller ones!