Detailed Notes on Recursion: Patterns and Coding Style

---

Introduction to Recursion

Recursion is a programming technique where a function calls itself to solve a problem. It is particularly useful for problems that can be divided into similar sub-problems. Recursion relies on:

1. Base Case: The condition under which the recursion stops.

2. Recursive Case: The part where the function calls itself with a smaller or simpler input.

Key components:

- Recursive function: A function that calls itself.

- Call stack: A data structure that stores information about active function calls.

---

Common Recursion Patterns

1. Divide and Conquer

Split the problem into smaller parts, solve each part recursively, and combine results.

Examples:

- Merge Sort

- Quick Sort

- Binary Search

Template:

```java

public Result solve(Problem problem) {

if (baseCaseCondition(problem)) {

return baseCaseResult();

}

List<SubProblem> subProblems = divide(problem);

List<Result> results = new ArrayList<>();

for (SubProblem subProblem : subProblems) {

results.add(solve(subProblem));

}

return combine(results);

}

```

When to Use:

- Problems can be broken into independent sub-problems.

- The solution to the overall problem is a combination of the sub-problems.

Advantages:

- Simplifies problem-solving by focusing on smaller chunks.

- Recursion inherently mirrors the divide-and-conquer paradigm.

2. Backtracking

Explore all possible solutions and backtrack when a solution doesn’t meet the constraints.

Examples:

- N-Queens Problem

- Sudoku Solver

- Permutations and Combinations

Template:

```java

public void solve(State state) {

if (baseCaseCondition(state)) {

processSolution(state);

return;

}

for (Choice choice : generateChoices(state)) {

makeChoice(state, choice);

solve(state);

undoChoice(state, choice);

}

}

```

When to Use:

- Problems require exploring all possible solutions.

- Constraints must be met at each decision point.

Advantages:

- Effective for constraint-satisfaction problems.

- Provides clear logic for exploring and eliminating paths.

3. Dynamic Programming with Recursion (Memoization)

Avoid redundant calculations by storing the results of sub-problems.

Examples:

- Fibonacci Numbers

- Longest Common Subsequence

- Knapsack Problem

Template:

```java

public Result solve(Problem problem, Map<Problem, Result> memo) {

if (memo.containsKey(problem)) {

return memo.get(problem);

}

if (baseCaseCondition(problem)) {

return baseCaseResult();

}

Result result = combine(

divide(problem).stream()

.map(subProblem -> solve(subProblem, memo))

.collect(Collectors.toList())

);

memo.put(problem, result);

return result;

}

```

When to Use:

- Overlapping sub-problems exist.

- The problem exhibits optimal substructure.

Advantages:

- Significant reduction in time complexity.

- Avoids redundant computations.

4. Tree Traversals

Recursively process nodes of a tree structure.

Examples:

- Preorder Traversal

- Inorder Traversal

- Postorder Traversal

Template:

```java

public void traverse(TreeNode node) {

if (node == null) {

return;

}

processNode(node);

traverse(node.left);

traverse(node.right);

}

```

When to Use:

- Problems involve hierarchical or tree-like structures.

Advantages:

- Mirrors natural hierarchical problem structures.

- Simplifies tree-related algorithms like search and modifications.

5. Tail Recursion

A specific type of recursion where the recursive call is the last operation in the function.

Example:

- Factorial Calculation (optimized for tail-recursive languages)

Template:

```java

public Result solve(Problem problem, Accumulator accumulator) {

if (baseCaseCondition(problem)) {

return accumulator.getResult();

}

return solve(smallerProblem(problem), accumulator.update());

}

```

When to Use:

- When recursion depth can grow large.

- For problems that can accumulate results without intermediate calculations.

Advantages:

- Reduces the risk of stack overflow in languages with tail-recursion optimization.

- Mimics iterative solutions with cleaner syntax.

Note: Java does not optimize tail recursion, so use iterative solutions when possible for deep recursion.

---

Best Practices for Writing Recursive Code

1. Define Clear Base Cases:

- Ensure the recursion terminates properly.

- Write and test the base cases first.

2. Avoid Redundant Calculations:

- Use memoization to store results of sub-problems.

- Example: Use `HashMap` in Java for caching.

3. Handle Large Inputs Gracefully:

- Be aware of stack overflow errors.

- Consider converting recursion to iteration for large input sizes.

4. Optimize Tail Recursion:

- Java does not optimize tail recursion, so prefer iteration in these cases.

5. Debug Recursively:

- Add print statements to trace function calls and stack states.

- Example: `System.out.println("Calling solve(" + problem + ")");`.

6. Use Helper Functions:

- Simplify the main function by delegating logic to helper functions with additional parameters.

7. Understand Problem Complexity:

- Analyze the time and space complexity of the recursive solution.

- Example: Avoid exponential complexity where possible (e.g., naive Fibonacci).

8. Iterative Alternatives:

- Consider iterative solutions when recursion is not strictly necessary.

---

Common Mistakes in Recursion

1. Missing Base Case:

- Leads to infinite recursion and stack overflow.

2. Improperly Defined Base Case:

- Incorrect base case logic can produce wrong results or unnecessary recursion.

3. Not Reducing Problem Size:

- Ensure each recursive call works on a simpler or smaller input.

4. Mutating Shared State:

- Be cautious with global variables and mutable data structures.

5. Stack Overflow:

- Recursive depth exceeding language limits (e.g., Java’s recursion depth limit).

---

Examples

Factorial Calculation

```java

public int factorial(int n) {

if (n == 0 || n == 1) {

return 1;

}

return n \* factorial(n - 1);

}

```

Fibonacci with Memoization

```java

public int fibonacci(int n, Map<Integer, Integer> memo) {

if (memo.containsKey(n)) {

return memo.get(n);

}

if (n <= 1) {

return n;

}

int result = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);

memo.put(n, result);

return result;

}

```

Permutations

```java

public List<List<Integer>> permutations(List<Integer> nums) {

if (nums.isEmpty()) {

List<List<Integer>> result = new ArrayList<>();

result.add(new ArrayList<>());

return result;

}

List<List<Integer>> result = new ArrayList<>();

for (int i = 0; i < nums.size(); i++) {

List<Integer> remaining = new ArrayList<>(nums);

remaining.remove(i);

for (List<Integer> perm : permutations(remaining)) {

List<Integer> current = new ArrayList<>();

current.add(nums.get(i));

current.addAll(perm);

result.add(current);

}

}

return result;

}

```

Non-Tail Recursion Example

```java

public int sumOfDigits(int n) {

if (n == 0) {

return 0;

}

return (n % 10) + sumOfDigits(n / 10);

}

```

Explanation:

- The recursive call `sumOfDigits(n / 10)` is not the last operation.

- After the recursive call returns, the result is added to `n % 10`.

- This is an example of non-tail recursion.

---

Conclusion

Recursion is a powerful tool for solving complex problems, especially those with a clear hierarchical or repetitive structure. By mastering patterns, understanding limitations, and adhering to best practices, you can write efficient and elegant recursive code.