



Ahmad Albarasy · Follow

Last edited by Ahmad Albarasy on Dec 6, 2025 

Contributed by



DSaA Lab 4 - Recursion

The Islamic University of Gaza
Engineering Faculty
Department of Computer Engineering

Prepared by: Eng. Ahmad Albarasy

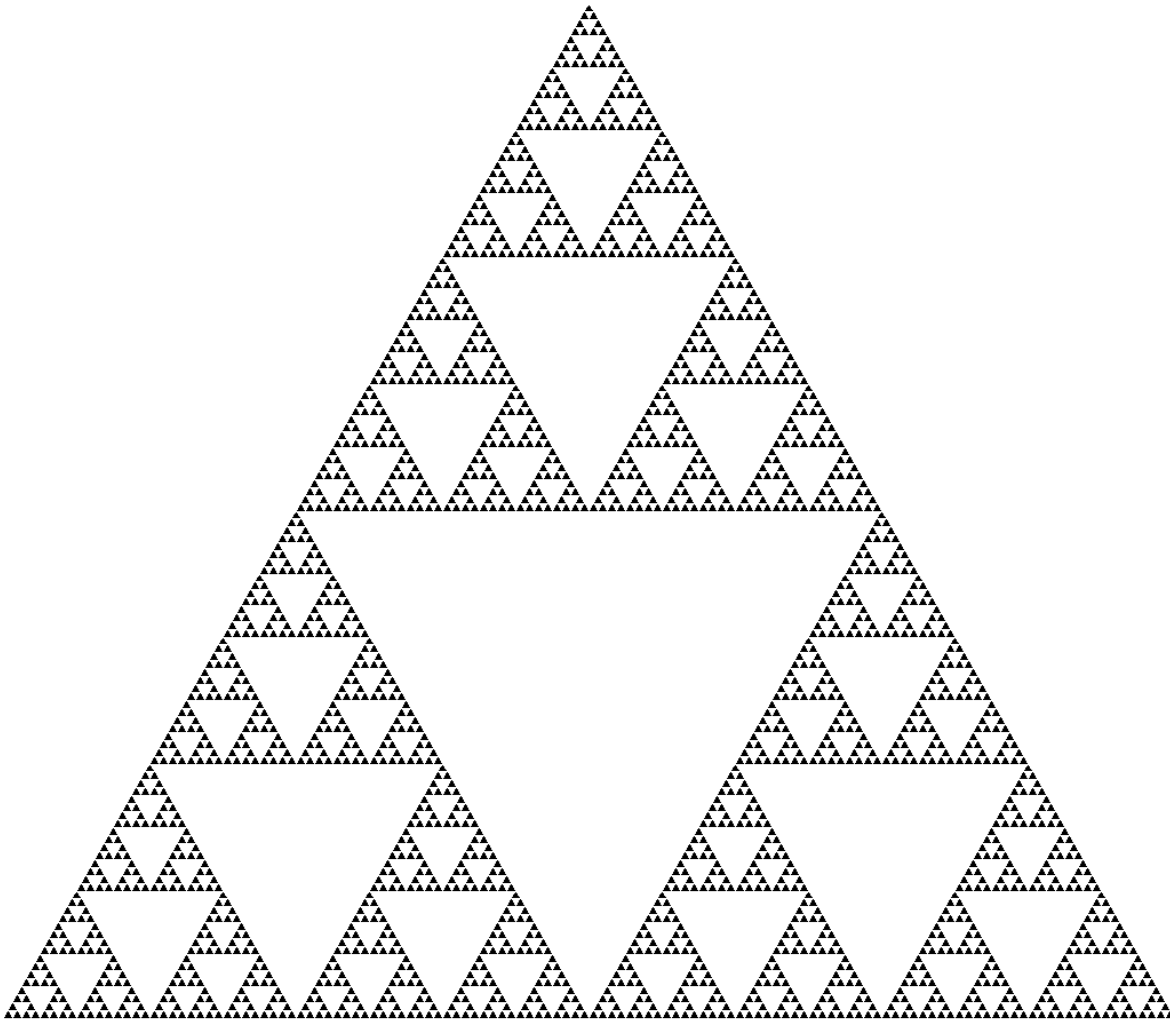
Under the supervision of: Prof. Ahmad Mahdi

In this lab, we are going to learn about recursion and how can we use it to solve interesting problems. We will also explore the different types of recursion, compare between solving problems iteratively and recursively, and lastly, we will discuss common pitfalls you might fall into while solving problems recursively.

Definition

Recursion is a technique by which a method makes one or more calls to itself during execution, or a concept or process that depends on a simpler or previous version of itself.

There are many examples of recursion in art and nature. For example, fractal patterns are naturally recursive. A physical example of recursion used in art is in the Russian Matryoshka dolls. Each doll is either made of solid wood, or is hollow and contains another Matryoshka doll inside it.



Structure of a Recursive Function

To better understand how a recursive function are structured, lets take a look at the factorial function example:

```
1 public static int factorial(int n){  
2     if(n < 0)  
3         throw new IllegalArgumentException();  
4     else if (n == 0)  
5         return 1; // base case  
6     else  
7         return n * factorial(n-1); // recursive case  
8 }
```

To build a recursive function, we typically need two important parts:

1. Recursive case: This part defines how the function calls itself with modified arguments. Each recursive call should reduce the problem in some way, ensuring that the function moves closer to the base case

2. Base case: This part defines when the recursion should terminate, providing a direct solution for the simplest instance of the problem. A recursive function could have one or more base cases, depending on the problem

Where Recursion Is Used

Recursion is an important technique in the study of data structures and algorithms, and it is used naturally to solve problems that can be divided into smaller subproblems following the same pattern as the original. For instance, in the upcoming labs, we will see how recursion can be leveraged to traverse hierarchical structures such as trees and graphs, and how it can be applied in divide-and-conquer strategies to break down problems into smaller, manageable pieces, as in the merge sort and quicksort algorithms.

⚠ Caution: Avoid recursion in critical systems

Recursion is generally avoided in critical systems because each recursive call consumes stack memory. If recursion is too deep or an edge case occurs where the base case is never reached, it can lead to unpredictable behavior or even a system crash. Iterative approaches are usually preferred in such cases for reliability and control over resource usage

Iteration vs. Recursion

In the previous sections, we explored recursion and how it works. In this section, we will compare recursive and iterative approaches, highlighting the strengths and weaknesses for each one of them.

We might encounter problems that can be solved using either approach, while other problems may be much easier to solve with one approach than the other. For instance, finding the sum of an array can be implemented both iteratively and recursively, whereas traversing a binary tree in an in-order fashion, as we will see in an upcoming lab, would be more natural and easy to understand when done in a recursive approach.

```
1 public class Factorial {
2     public static int factorialIterative(int n) {
3         int result = 1;
4         for (int i = 2; i <= n; i++) {
5             result *= i;
6         }
7         return result;
8     }
9
10    public static int factorialRecursive(int n) {
11        if (n == 0 || n == 1) return 1;
12        return n * factorial(n - 1);
13    }
14 }
```

In terms of memory usage, especially the stack, the recursive approach can consume a significant amount of memory because each function call requires a new frame in the call stack to store the function's parameters, local variables, and return address. As the recursion depth increases, the stack grows accordingly, which can lead to high memory usage or even a stack overflow if the recursion is too deep.

On the other hand, the iterative approach typically use a fixed amount of stack memory regardless of the number of iterations, making them more predictable and efficient in terms of resource usage.

Types of Recursion

One way to classify recursive functions is based on the number of times a function calls itself during each execution. Under this classification, recursion falls into three main categories:

1. **Single Recursion:** refers to functions containing only a single recursive call per activation
2. **Binary Recursion** refers to functions containing two recursive calls per activation
3. **Multiple Recursion** refers to functions containing more than two recursive calls per activation

Standard examples of single recursion include list traversal (such as a linear search) or computing the factorial of a number. Binary recursion often appears in divide-and-conquer algorithms such as merge sort, or in computing the Fibonacci sequence, where each call branches into two smaller subproblems. Multiple recursion shows up in problems like tree traversal, such as depth-first search on a node with multiple children, where each node may lead to several recursive calls.

Single recursion is typically more efficient and can often be replaced by an iterative computation running in linear time and constant space. Binary and multiple recursion, by contrast, can create large recursion trees that require significantly more time and memory. For example, the naive recursive Fibonacci implementation grows exponentially as each call produces two more calls. These forms of recursion are more inherently recursive and generally cannot be converted into simple loops without explicitly managing your own stack.

Tasks

Task 1 (4 points)

Given an integer n , return `true` if it is a power of four. Otherwise, return `false`.

An integer n is a power of four, if there exists an integer x such that $n == 4^x$.

Question link on [LeetCode](#)

Example 1:

Input: $n = 16$

Output: `true`

Example 2:

Input: $n = 5$

Output: `false`

Example 3:

Input: $n = 1$

Output: `true`

Constraints:

- $-2^{31} \leq n \leq 2^{31}-1$

Task 2 (6 points)

Implement `pow(x, n)`, which calculates x raised to the power n (i.e., x^n).

Question link on [LeetCode](#)

Example 1:

Input: $x = 2.00000$, $n = 10$

Output: 1024.00000

Example 2:

Input: $x = 2.10000$, $n = 3$

Output: 9.26100

Example 3:

Input: $x = 2.00000$, $n = -2$

Output: 0.25000

Explanation: $2^{-2} = 1/2^2 = 1/4 = 0.25$

Constraints:

- $-100.0 < x < 100.0$
- $-231 \leq n \leq 231-1$
- Either x is not zero or $n > 0$
- $-104 \leq x^n \leq 104$

Tasks Submission Guide

You will solve both problems on LeetCode and you have to make sure that your solution passes all test cases and that LeetCode accepts it. Once you are done, its time to submit your solution on GradeScope.

Firstly, make sure to submit **only** two files:

1. TaskOne.java**2. TaskTwo.java**

Secondly, for each question, you have to follow these steps to make sure GradeScope's autograder runs without issues:

For Task 1:

1. Copy your whole solution into a file named `TaskOne.java`
2. Change the class name from `Solution` to `TaskOne`, and make sure that the class is public
3. add the following package identifier to the beginning of the file, make sure to copy it and **don't write it manually to avoid typos**:

```
1 | package com.gradescope.labFour;
```

Here's an example solution for reference:

```
1 package com.gradescope.labFour;
2
3 public class TaskOne {
4     public boolean isPowerOfFour(int n) {
5         // your solution here
6     }
7     /* if any helper methods are needed, make sure to include them */
8 }
```

For Task 2:

1. Copy your whole solution into a file named `TaskTwo.java`
2. Change the class name from `Solution` to `TaskTwo`, and make sure that the class is `public`
3. add the following package identifier to the beginning of the file, make sure to copy it and **don't write it manually to avoid typos**:

```
1 package com.gradescope.labFour;
```

Here's an example solution for reference:

```
1 package com.gradescope.labFour;
2
3 public class TaskTwo {
4     public double myPow(double x, int n) {
5         // your solution here
6     }
7     /* if any helper methods are needed, make sure to include them */
8 }
```

Happy Coding 😊

Last changed by

Ahmad Albarasy · Follow

0 38

Add a comment

Read more

Data Structures and Algorithms Lab 3 - Algorithm Analysis

In this lab, we are going to learn how to analyze our algorithms in terms of time and space complexity, whi...
Dec 3, 2025

Data Structures and Algorithms Lab 2 - Linked Lists

In this lab, we are going to learn about Linked Lists. What they are, how can we implement them, their key...
Dec 3, 2025

Data Structures and Algorithms Lab 1 - Arrays

In this lab, we are going to dive into Arrays, one of the most basic yet fundamental data structures that we us...
Dec 3, 2025

Data Structures and Algorithms Lab 0 - Introduction

Welcome one and all to the Data Structures and Algorithms lab! This lab is one of the most important...
Dec 3, 2025

[Read more from Ahmad Albarasy](#)

Published on  HackMD