Famous Functions

From Pythagoras to Binary Search

Brendan Shea, PhD

Programming and Problem Solving • Rochester Community and Technical College

Introduction

Every programmer stands on the shoulders of giants. The functions we write today—whether calculating distances, searching through data, or solving equations—have roots stretching back thousands of years. Ancient mathematicians discovered the formulas; pioneering computer scientists figured out how to express them as algorithms; and now you're learning to implement them in Java.

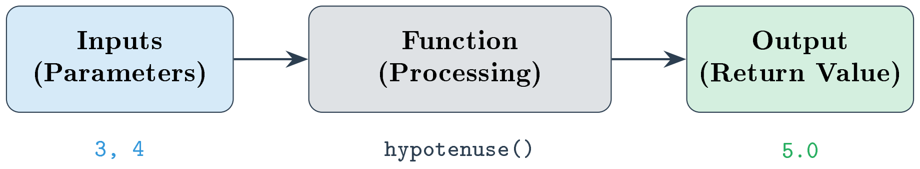
This case study explores several "famous functions": mathematical and computational procedures so useful that they've been implemented billions of times across every programming language. Along the way, you'll learn not just *how* to write functions, but *why* functions are structured the way they are, and how to design your own.

What You'll Learn

- What functions are and why they're fundamental to programming
- The history and mathematics behind classic algorithms
- How to implement the Pythagorean theorem, quadratic formula, factorial, and search algorithms
- Principles for designing, implementing, and testing your own functions

1 What Are Functions?

A **function** (also called a *method* in Java) is a reusable block of code that performs a specific task. You give it inputs, it does some work, and it returns an output. Think of a function like a machine: raw materials go in, a product comes out.



Why do we use functions? Three crucial reasons:

Reusability. Once you write a function to calculate a hypotenuse, you can use it anywhere in your program—or in future programs—without rewriting the logic. Write once, use forever.

Abstraction. Functions let you hide complexity. When you call `Math.sqrt(x)`, you don't need to know *how* the square root is calculated. You just trust that it works. This lets you think at higher levels.

Organization. Breaking a program into functions makes it easier to understand, test, and debug. Each function has one job, and you can verify that job works correctly in isolation.

1.1 Anatomy of a Java Function

Let's examine the parts of a function:

Anatomy of a Function

```java
//   return type     name          parameters
//        |            |                |
//        v            v                v
    public static double hypotenuse(double a, double b) {
        double cSquared = a * a + b * b;   // body: the work
        return Math.sqrt(cSquared);        // return statement
    }
```

The **return type** (`double`) declares what kind of value the function produces. The **parameters** (`double a, double b`) are the inputs the function expects. The **body** contains the actual code that does the work. The **return statement** sends the result back to whoever called the function.

When you call this function with `hypotenuse(3, 4)`, Java substitutes 3 for `a` and 4 for `b`, executes the body, and gives you back 5.0.

2 The Pythagorean Theorem

Pythagoras of Samos (c. 570–495 BCE)

Pythagoras was a Greek philosopher and mathematician who founded a religious movement based on mathematics. His followers, the Pythagoreans, believed that numbers were the ultimate reality underlying the universe. While Pythagoras himself may not have discovered the theorem bearing his name—Babylonian mathematicians knew it a thousand years earlier—his school provided the first known proof, elevating it from an observed pattern to a demonstrated truth.

The **Pythagorean theorem** states that in a right triangle, the square of the hypotenuse (the side opposite the right angle) equals the sum of the squares of the other two sides:

The Formula

$$a^2 + b^2 = c^2$$

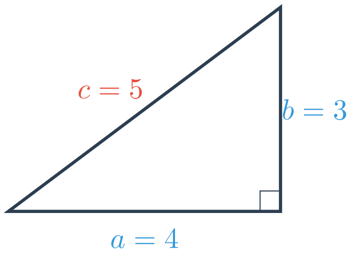
Therefore: $c = \sqrt{a^2 + b^2}$

This formula is everywhere: calculating distances on maps, determining screen dimensions, physics simulations, game development, and anywhere else geometry matters. Let's implement it:

Pythagorean Theorem in Java

```java
/**
 * Calculates the length of the hypotenuse of a right triangle.
 * @param a length of one leg
 * @param b length of other leg
 * @return length of the hypotenuse
 */
public static double hypotenuse(double a, double b) {
    return Math.sqrt(a * a + b * b);
}

// Example usage:
double distance = hypotenuse(3, 4);  // returns 5.0
```



$$c = \sqrt{4^2 + 3^2}$$
$$c = \sqrt{16 + 9}$$
$$c = \sqrt{25} = 5$$

Notice the **Javadoc comment** (the comment starting with `/**`). This documents what the function does, what parameters it expects, and what it returns. Good documentation is part of writing good functions.

3 The Quadratic Formula

From Babylon to al-Khwarizmi

Babylonian mathematicians solved quadratic equations around 2000 BCE, but only for specific cases and without a general formula. The Persian mathematician Muhammad ibn Musa al-Khwarizmi (c. 780–850 CE) wrote a systematic treatment of equation-solving in his book *Al-Kitab al-mukhtasar*. His name gave us the word “algorithm,” and his methods evolved into the quadratic formula we use today.

A **quadratic equation** has the form $ax^2 + bx + c = 0$. The quadratic formula finds the values of x that satisfy this equation:

The Quadratic Formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The expression $b^2 - 4ac$ is called the **discriminant**.

The **discriminant** tells us how many solutions exist: if it’s positive, there are two real solutions; if zero, one solution; if negative, no real solutions (only complex numbers).

Implementing this in Java requires us to handle these cases:

Quadratic Formula in Java

```java
/**
 * Solves ax^2 + bx + c = 0 and returns the solutions.
 * @return array of solutions (0, 1, or 2 elements)
 */
public static double[] solveQuadratic(double a, double b, double c) {
    double discriminant = b * b - 4 * a * c;

    if (discriminant < 0) {
        // No real solutions
        return new double[0];  // empty array
    } else if (discriminant == 0) {
        // One solution (repeated root)
        double x = -b / (2 * a);
        return new double[] { x };
    } else {
        // Two solutions
        double sqrtDisc = Math.sqrt(discriminant);
        double x1 = (-b + sqrtDisc) / (2 * a);
        double x2 = (-b - sqrtDisc) / (2 * a);
        return new double[] { x1, x2 };
    }
}

// Example: solve x^2 - 5x + 6 = 0
double[] roots = solveQuadratic(1, -5, 6);  // returns {3.0, 2.0}
```

This function returns an *array* because the number of solutions varies. The function's design reflects the mathematical reality: sometimes there's no answer, sometimes one, sometimes two.

4 The Factorial Function

Factorial Through History

Factorial calculations appear in ancient Indian mathematics (the Jain text *Anuyogadwara Sutra*, c. 200 BCE) in the context of counting permutations. The modern notation $n!$ was introduced by Christian Kramp in 1808. Factorials are fundamental to combinatorics (counting arrangements), probability theory, and appear unexpectedly in calculus through Taylor series.

The **factorial** of a positive integer n, written $n!$, is the product of all positive integers from 1 to n:

Factorial Definition

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

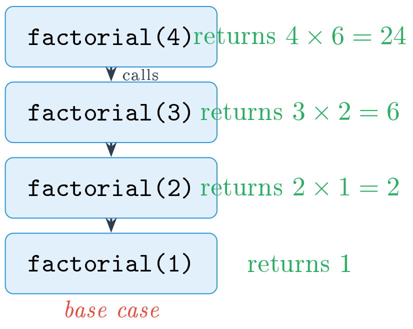
By convention, $0! = 1$.
Examples: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Factorial has a beautiful property: $n! = n \times (n-1)!$. This means we can define factorial in terms of itself—a technique called **recursion**.

Factorial: Iterative vs. Recursive

```java
// Iterative approach: use a loop
public static long factorialIterative(int n) {
    long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;  // multiply result by each number
    }
    return result;
}

// Recursive approach: function calls itself
public static long factorialRecursive(int n) {
    if (n <= 1) {
        return 1;  // base case: 0! = 1! = 1
    } else {
        return n * factorialRecursive(n - 1);  // recursive case
    }
}
```



Every recursive function needs a **base case**—a condition where it stops calling itself and returns a direct answer. Without a base case, the function would call itself forever (until your computer runs out of memory).

Note that we use `long` instead of `int` because factorials grow extremely fast. Even so, 21! exceeds what a `long` can hold!

5 Linear Search

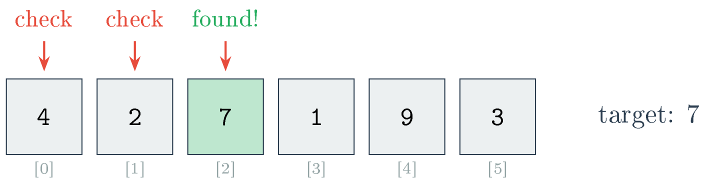
Now we move from mathematics to computer science. **Searching**—finding a specific item in a collection—is one of computing's most fundamental operations. The simplest approach is **linear search**: check each element one by one until you find what you're looking for.

Linear Search

```java
/**
 * Searches for a target value in an array.
 * @return index of target, or -1 if not found
 */
```

```
public static int linearSearch(int[] array, int target) {
    for (int i = 0; i < array.length; i++) {
        if (array[i] == target) {
            return i;  // found it! return the index
        }
    }
    return -1;  // searched everything, not found
}

// Example:
int[] numbers = {4, 2, 7, 1, 9, 3};
int index = linearSearch(numbers, 7);  // returns 2
```



Linear search is simple and works on any array, sorted or not. But it's slow for large collections: if you have a million elements, you might need to check all million. Computer scientists describe this as **O(n)** or "linear time"—the time grows proportionally with the input size.

6 Binary Search

The Birth of Binary Search

Binary search is ancient in concept—it's how you naturally search a dictionary or phone book. But implementing it correctly in code proved surprisingly tricky. Jon Bentley reported in his 1986 book *Programming Pearls* that when he asked professional programmers to write binary search, 90% got it wrong. The first bug-free published binary search appeared in 1962, sixteen years after the first description of the algorithm!

If the array is *sorted*, we can do much better than linear search. **Binary search** repeatedly divides the search space in half:

1. Look at the middle element.
2. If it's your target, you're done.
3. If your target is smaller, search the left half.
4. If your target is larger, search the right half.
5. Repeat until found or the search space is empty.

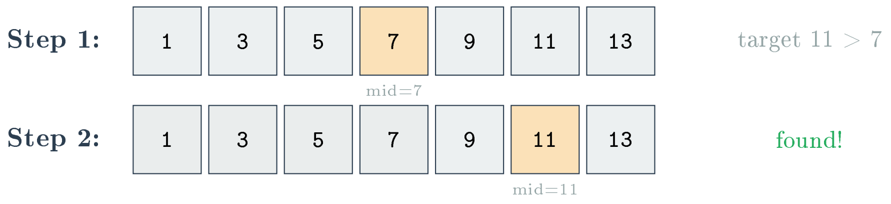
Binary Search

```
/**
 * Searches for target in a SORTED array.
 * @return index of target, or -1 if not found
```

```
  */
public static int binarySearch(int[] array, int target) {
    int left = 0;
    int right = array.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;   // find middle

        if (array[mid] == target) {
            return mid;   // found it!
        } else if (array[mid] < target) {
            left = mid + 1;   // target is in right half
        } else {
            right = mid - 1;   // target is in left half
        }
    }
    return -1;   // not found
}
```



Binary search is O(log n)—logarithmic time. For a million elements, linear search might need a million comparisons; binary search needs at most 20. For a billion elements: 30 comparisons. This dramatic improvement is why sorting data is often worthwhile—it enables fast searching.

7 Designing Your Own Functions

Now that you've seen famous functions, how do you create your own? Here's a systematic approach:

7.1 Step 1: Define the Contract

Before writing code, answer these questions:

- What does this function do? (One clear task)
- What inputs does it need? (Parameters and their types)
- What output does it produce? (Return type)
- What are the edge cases? (Empty arrays, zero, negative numbers, etc.)

Write the function signature and Javadoc *before* the implementation:

Define the Contract First

```
/**
 * Calculates the average of an array of numbers.
 * @param numbers the array to average (must not be empty)
```

```
  * @return the arithmetic mean
  * @throws IllegalArgumentException if array is empty
  */
public static double average(double[] numbers) {
    // TODO: implement
}
```

7.2 Step 2: Write Test Cases

Before implementing, think about how you'll know the function works. What inputs should produce what outputs?

Think About Tests First

```
// Test cases for average():
// average({10}) should return 10.0
// average({1, 2, 3}) should return 2.0
// average({-5, 5}) should return 0.0
// average({}) should throw an exception
```

This practice is called **test-driven development**: tests guide your implementation and verify correctness.

7.3 Step 3: Implement

Now write the code. Start simple—get the basic case working, then handle edge cases:

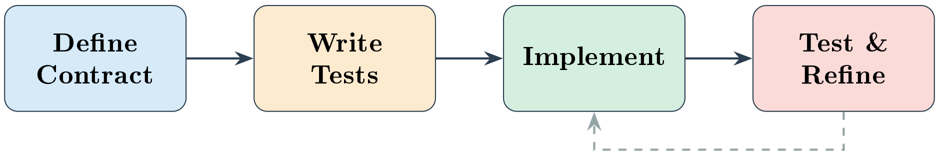
Implementation

```
public static double average(double[] numbers) {
    if (numbers.length == 0) {
        throw new IllegalArgumentException("Cannot average empty
   array");
    }

    double sum = 0;
    for (double num : numbers) {
        sum += num;
    }
    return sum / numbers.length;
}
```

7.4 Step 4: Test and Refine

Run your test cases. If something fails, debug. Once tests pass, consider: Is the code readable? Could it be simpler? Are there edge cases you missed?



8 Principles of Good Functions

Across all our famous functions, certain principles emerge:

Do one thing well. A function should have a single, clear purpose. If you can't describe what a function does in one sentence, it's probably doing too much.

Use meaningful names. `hypotenuse()` is better than `calc()`. `linearSearch()` tells you exactly what algorithm is used. Names are documentation.

Handle edge cases. What happens with empty input? Zero? Negative numbers? The quadratic formula handles the case of no real solutions; factorial handles 0 and 1. Anticipate the unusual.

Document your assumptions. Binary search *requires* a sorted array. If your function has preconditions, state them clearly in comments.

Prefer clarity over cleverness. Code is read far more often than it's written. A simple, obvious implementation beats a clever, confusing one—even if the clever version is slightly faster.

The functions we've explored have endured because they're not just correct—they're clear, well-defined, and focused. These qualities matter whether you're implementing ancient mathematics or inventing something new.

Discussion Questions

Discussion Questions

1. **Iteration vs. Recursion:** We saw both iterative and recursive implementations of factorial. What are the advantages of each approach? When might you prefer one over the other? Are there problems that are more naturally expressed recursively?

2. **The Sorted Requirement:** Binary search only works on sorted data. In what situations would it be worth sorting an array just to use binary search? When would linear search be preferable despite being slower?

3. **Edge Cases:** The quadratic formula must handle three cases (two solutions, one, or none). What would happen if we didn't check the discriminant and just applied the formula blindly? Why is handling edge cases important?

4. **Historical Algorithms:** Many algorithms we use today were discovered centuries or millennia ago. Why do you think these mathematical insights have remained useful? What does it suggest about the nature of mathematical truth?

5. **Testing Challenges:** How would you test the `binarySearch` function thoroughly? What inputs would you try? How many test cases are "enough"? Is it possible to prove a function is correct through testing alone?

Key Terms

Glossary of Key Terms

Base Case	In recursion, the condition that stops the function from calling itself, providing a direct answer.
Binary Search	A search algorithm for sorted arrays that repeatedly halves the search space, achieving O(log n) time.
Discriminant	In the quadratic formula, the expression $b^2 - 4ac$ that determines the number of real solutions.
Factorial	The product of all positive integers up to n, written $n!$, fundamental to counting and probability.
Function	A reusable block of code that takes inputs (parameters), performs a task, and returns an output.
Javadoc	A documentation format for Java using special comments (`/** */`) to describe functions, parameters, and return values.
Linear Search	A search algorithm that checks each element sequentially, working on any array but taking O(n) time.
O(log n)	Logarithmic time complexity, where the time grows with the logarithm of the input size—very efficient.
O(n)	Linear time complexity, where the time grows proportionally with the input size.
Parameter	A variable in a function definition that receives a value when the function is called.
Pythagorean Theorem	The relation $a^2 + b^2 = c^2$ for right triangles, used to calculate distances.
Quadratic Equation	An equation of the form $ax^2 + bx + c = 0$, solved by the quadratic formula.
Recursion	A technique where a function calls itself to solve smaller instances of the same problem.
Return Type	The data type of the value a function produces, declared before the function name.
Signature	The function's name, return type, and parameter list—its external interface.
Test-Driven Development	A practice of writing test cases before implementing the code they test.