# Programming Exercises: Exponentials, Roots, and Search

These exercises challenge you to implement the mathematical concepts and search algorithms we have discussed, using Python.

## Section 1: Exponentials and Powers (Rules)

### Exercise 1: Power Calculation

Write a Python function `calculate_power(base, exponent)` that calculates the result of the base raised to a positive integer exponent **without using Python's built-in `**` operator or `math.pow()`**. Use a loop and the principles of repeated multiplication.

### Exercise 2: Power of a Product Rule

Write a function `check_product_rule(a, b, n)` that verifies the **Power of a Product Rule** $\mathbf{(a \times b)^n = a^n \times b^n}$ by calculating both sides separately and returning `True` if they are equal (within a small tolerance for floating point numbers).

### Exercise 3: Negative Exponent Converter

Write a function `convert_negative_exponent(base, exponent)` that takes a negative integer exponent and converts the expression to its reciprocal form. For example, if the input is `(3, -2)`, the function should return the equivalent positive form as a string: `"1 / 3^2"`.

### Exercise 4: Non-Integer Power Calculation (Fractional Exponents)

This exercise requires you to compute $a^m$ where $m$ is a non-integer (e.g., $5.2$ or $-1.5$). The solution must use the Bisection Method (from Exercise 6) for the root part of the calculation.

Steps:

1. **Fraction Conversion:** Convert the floating-point exponent $m$ into an improper fraction $\frac{p}{q}$.
   - *Example:* $m = 5.2$ becomes $\frac{52}{10}$.
2. **Apply Rule:** Transform the calculation using the fractional exponent rule: $a^{p/q} = \sqrt[q]{a^p}$.
3. **Calculate Power:** Compute the numerator's power, $a^p$.
4. **Calculate Root:** Use the Bisection Method (`nth_root_bisection` from Exercise 6) to find the $q$-th root of the result from Step 3.
5. **Handle Negatives:** If the original exponent $m$ was negative, use the Negative Exponent Rule (Exercise 3) to take the reciprocal of the final result.

*Note: For the most efficient calculation, proceed to Exercise 12 to learn how to simplify the fraction $\frac{p}{q}$ first.*

## Section 2: Roots and the Bisection Method

### Exercise 5: Simple Square Root Checker

Write a function `is_perfect_square_root(n, root)` that checks if a given integer `root` is the correct integer square root of `n`. Example: `is_perfect_square_root(64, 8)` should return `True`.

### Exercise 6: Implementing the Bisection Method

Recreate the `nth_root_bisection(radicand, n, precision)` function. This is the core task: implement the logic to find the $n$-th root by iteratively halving the search interval until the difference between the high and low bounds is less than the specified `precision`.

### Exercise 7: Root Approximation with Iterations

Modify the Bisection Method function from Exercise 6 to stop after a maximum number of **iterations** (e.g., 15) instead of based on precision. Return the current best approximation. This demonstrates the logarithmic speed of the algorithm.

## Section 3: Search Algorithms

### Exercise 8: Brute Force Search (Linear)

Write the Python function `brute_force_search(data_list, target)` that implements the Brute Force (Linear) search algorithm. It should **return the index** of the `target` if found, or **-1** if it is not found. The list does not need to be sorted.

### Exercise 9: Binary Search Implementation

Write the Python function `binary_search_index(sorted_list, target)` that implements the Binary Search algorithm. It must **require that the input list is sorted** and should return the index of the `target` or **-1** if not found.

### Exercise 10: Search Performance Comparison

Write a function `compare_search_speed(list_size, target)` that generates a random, sorted list of a given `list_size`. The function should then run both `brute_force_search` and `binary_search_index` for a random `target` and report the number of comparisons each method took (you will need to modify the search functions to count steps).

## Section 4: Combination

### Exercise 11: Perfect Square/Cube Finder

Write a function `find_perfect_root(n)` that uses **Binary Search principles** to efficiently find the integer root of an integer $n$.

- If $n$ is a perfect square (e.g., 100), return its root (10).

- If $n$ is a perfect cube (e.g., 27), return its root (3).
- If $n$ is neither, return `None`.

*Hint: Use binary search to find a number $x$ such that $x^2 = n$ or $x^3 = n$. You only need to check integers up to $n$ itself.*

**Exercise 12: Fraction Simplification via Prime Factorization and Full Calculation**

This exercise focuses on simplifying the fractional exponent $\frac{p}{q}$ (e.g., $52/10$ to $26/5$) and then using the simplified fraction to calculate the final power.

**Part A: Prime Factorization Helper** Write a function `prime_factorize(n)` that takes an integer $n$ and returns a dictionary or list containing its prime factors.

- *Example:* `prime_factorize(52)` returns `{2: 2, 13: 1}` (meaning $2^2 \times 13^1$).
- *Example:* `prime_factorize(10)` returns `{2: 1, 5: 1}`.

**Part B: Fraction Simplifier** Write a function `simplify_fraction(numerator, denominator)` that uses the `prime_factorize` function to determine the greatest common divisor (GCD) between the two numbers. The function should return the simplified numerator and denominator.

- *Example:* For $52$ and $10$, the only common prime factor is $2$ (once). Divide both by 2 to get $26$ and $5$. The function should return `(26, 5)`.

**Part C: Full Non-Integer Power Calculation (Refined Exercise 4)** Create a final function `calculate_non_integer_power_simplified(base, exponent_decimal)` that is the complete, efficient solution for fractional exponents. The function must:

1. Convert `exponent_decimal` to an improper fraction $(p, q)$.
2. Use `simplify_fraction` (Part B) to get the reduced fraction $(p', q')$.
3. Compute the result $base^{p'/q'} = \sqrt[q']{base^{p'}}$ using your power and bisection root functions.