Introduction to Problem Solving

Contents

[I. Why Problem Solving is Important 2](#_Toc199500407)

[1. Problem Statement 2](#_Toc199500408)

[Understanding Factors 2](#_Toc199500409)

[Naive Solution 2](#_Toc199500410)

[Why is This a Problem? 2](#_Toc199500411)

[Why Problem Solving (Optimization) Is Important 3](#_Toc199500412)

[Optimized Approach Using Square Root 4](#_Toc199500413)

[2. Check If a Number is Prime 4](#_Toc199500414)

[Solution 5](#_Toc199500415)

[3. Square Root Calculation Without Library 5](#_Toc199500416)

[4. Floor of Square Root (for any positive integer) 6](#_Toc199500417)

[Floor of a number 6](#_Toc199500418)

[5. Basics of Logarithms 6](#_Toc199500419)

# Why Problem Solving is Important

## Problem Statement

* Given an integer , count the number of its factors.
* **Example:**

For , factors are: → Total = **4**

### **Understanding Factors**

* **Definition**: A number is a factor of if (i.e., divides without a remainder).
* Example:
  + is a factor of because
  + is not a factor of since

### Naive Solution

* Loop from
* For each , check if
* If , increment a counter
* Time Complexity:

### Why is This a Problem?

* Case:
  + Assuming operations per second
* Case:
  + Requires operations
  + With , time taken =
  + Time ≈ 317 years

|  |
| --- |
| int countFactorsNaiveApproach(long long int n) {  int count = 0;  for (long long int i = 1; i <= n; i++) {  if (n % i == 0) {  count++;  }  }  return count;  } |

### Why Problem Solving (Optimization) Is Important

* Brute force approaches might work for small inputs, but fail miserably at scale.
* Recognizing that **factors come in pairs** allows the loop to run a smaller number of times.
* Example 1: Count all the factors of 24

|  |  |  |  |
| --- | --- | --- | --- |
| i | n / i | Factor Pair | count |
| 1 | 24 | (1, 24) | count+=2 |
| 2 | 12 | (2, 12) | count+=2 |
| 3 | 8 | (3, 8) | count+=2 |
| 4 | 6 | (4, 6) | count+=2 |
|  | | |  |
| 6 | 4 | already seen as (4, 6) |  |
| 8 | 3 | already seen as (3, 8) |  |
| 12 | 2 | already seen as (2, 12) |  |
| 24 | 1 | already seen as (1, 24) |  |

**There are 8 factors for 24**

* Example 2: Count all the factors of 16

|  |  |  |  |
| --- | --- | --- | --- |
| i | n / i | Factor Pair | count |
| 1 | 16 | (1, 16) | count+=2 |
| 2 | 8 | (2, 8) | count+=2 |
| 4 | 4 | 4 | count+=1 |

* + Here 16 is a perfect square, and hence there are odd number of factors.
  + There are 5 factors for 16.
* To find till when to run the loop, we can write the following condition:
* This reduces time from to .

### **Optimized Approach Using Square Root**

* Factors appear in pairs
* Only need to iterate from 1 to .
* If is a factor, is also a factor. Increment the by 2.
* When , increment the by (e.g., perfect square).

|  |
| --- |
| int countFactorsOptimizedApproach(long long int n) {  int count = 0;  for (long long int i = 1; i \* i <= n; i++) {  if (n % i == 0) {  if (i == n / i) {  count += 1;  }  else {  count += 2;  }  }  }  return count;  } |

## **Check If a Number is Prime**

* Definition: A number is prime if it has exactly two factors: 1 and itself.
* 1 is not a prime. As it has only 1 factor.
* 0 is not a prime as it has infinitely many divisors.
* A **composite number** is a number with **more than two positive divisors** and must be **greater than 1**.

### Solution

* We can modify the above countFactors() code to find whether a number is prime or not.
  + If , return false (not prime).
  + Initialize to track number of divisors.
  + Loop from to :
    - If :
      * If , increment count by 1 (perfect square).
    - Else, increment by (pair of divisors).
  + If , return false (not prime).
  + After the loop, return true only if .

|  |
| --- |
| bool isPrime(long long int n) {  if (n <= 1) return false;  int count = 0;  for (long long int i = 2; i \* i <= n; i++) {  if (n % i == 0)  if (n / i == i) {  count += 1;  else  count += 2;  if (count > 2) {  return false;  }  }  }  return count == 2;  } |

## **Square Root Calculation Without Library**

* Compute square root of a **perfect square** without using math libraries.

|  |
| --- |
| long long int squareRootOfPerfectSquare(long long int n) {  for (long long int i = 1; i \* i <= n; i++) {  if (i \* i == n) {  return i;  }  }  } |

## **Floor of Square Root (for any positive integer)**

* **Problem Statement**: Given where , find its .
* Example:
  + If n = 35, output
  + If n = 48, output 6

### Floor of a number

* The floor of a number , written as or , is the greatest integer less than or equal to .

| **Number (n)** | **floor(n)** | **Explanation** |
| --- | --- | --- |
| 5.7 | 5 | 5 is the largest integer ≤ 5.7 |
| 3.0 | 3 | Already an integer |
| -2.3 | -3 | -3 is the largest integer ≤ -2.3 |
| -5.0 | -5 | Already an integer |

|  |
| --- |
| long long int floorOfSqrt(long long int n) {  int potential\_ans = 0;  for (long long int i = 1; i <= n; i++) {  if (i \* i <= n) {  potential\_ans = i;  }  else  break;  }  return potential\_ans;  } |

## Basics of Logarithms

* → means
  + Example: since
* Properties:
* If ⇒
* How many times do you divide by until you reach 1?
  + This is essentially