BigInteger and String
When do you need `BigInteger`

- Numbers with 20 or more digits (e.g., if ever > $10^{20}$)
- Factorials over 20! ($2,432,902,008,176,640,000$ is 19 digits)

`BigInteger` also convenient for:

- Number base conversion
- Greatest common divisors
- Modular arithmetic
- Large prime numbers
When do you need `BigInteger`:
- Numbers with 20 or more digits (e.g., if ever > $10^{20}$)
- Factorials over 20! ($2,432,902,008,176,640,000$ is 19 digits)

`BigInteger` is also convenient for:
- Number base conversion
- Greatest common divisors
- Modular arithmetic
- Large prime numbers

Some languages like Ruby and Python seamlessly switch between integers and big integers.
Getting started

- Constructors
  - BigInteger(byte[] val) (two’s-complement)
  - BigInteger(String val) (string in base 10)
  - ...
  - BigInteger.valueOf(long val) (64-bit integer)

- Static constants
  - BigInteger.ONE
  - BigInteger.TEN
  - BigInteger.ZERO

Implementation
- BI objects are immutable
- Sign is stored as an int
- Magnitude is stored as an int[]
Binary Operations

- **Bitwise**
  - and(BigInteger val)
  - andNot(BigInteger val)
  - not()
  - or(BigInteger val)
  - xor(BigInteger val)
  - shiftLeft(int n)
  - shiftRight(int n)

- **Size**
  - bitCount()
  - bitLength()

- **One at a time**
  - clearBit(int n)
  - flipBit(int n)
  - getLowestSetBit()
  - testBit(int n)
  - setBit(int n)
Bonus features

- Number base conversion
  - `BigInteger(String val, int radix)`
  - `toString(int radix)`

- Greatest common divisor
  - `gcd(BitInteger val)`

- Modular arithmetic
  - `divideAndRemainder(BigInteger val)`
  - `mod(BigInteger m) // non-negative`
  - `modInverse(BigInteger m)`
  - `modPow(BigInteger exponent, BigInteger m)`
  - `remainder(BigInteger val)`
Large prime numbers

- Probabilistic test
  - `isProbablePrime(int certainty)`
  - true is very likely prime
  - false is definitely composite

- Trade-off: time vs. accuracy
  - \( P(\text{prime}) = 1 - 1 / 2^{\text{certainty}} \)
  - 10 is usually good enough (\( P > 0.999 \))

- Other methods
  - \( P(\text{prime}) = 1 - 1 / 2^{100} \)
  - `nextProbablePrime()`
  - `probablePrime(int bitLength, Random rnd)`
Simple Addition

Yup, yet another trivial problem. You just have to add two integers! Oh, not easy enough? Sure, let’s say that they’re positive as well. Now it should be trivial, right?

Input

The input consists of two lines, each containing a positive integer less than $10^{10,000}$. Oh right... I forgot to mention that the integers can be quite large.

Output

One line containing the sum of the two integers.

<table>
<thead>
<tr>
<th>Sample Input 1</th>
<th>Sample Output 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1337 42</td>
<td>1379</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sample Input 2</th>
<th>Sample Output 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 999999999999</td>
<td>10000000000000</td>
</tr>
</tbody>
</table>
JAVA:

import java.util.*;
import java.math.*;

public class Main{
  public static void main(String args[]){
    Scanner scn = new Scanner(System.in);
    BigInteger a = new BigInteger(scn.nextLine());
    BigInteger b = new BigInteger(scn.nextLine());
    System.out.println(a.add(b));
  }
}

Solutions

RUBY:

```ruby
puts gets.to_i + gets.to_i
```

PYTHON:

```python
print int(input()) + int(input())
```

KOTLIN:

```kotlin
fun main(argv:Array<String>){
    println(readLine()!!.toBigInteger() + readLine()!!.toBigInteger())
}
```

SCALA:

```scala
import scala.io.StdIn.{readLine}
object Solution {
    def main(args: Array[String]): Unit = {
        println(BigInt(readLine())+BigInt(readLine()))
    }
}
```
In-class problem

An-anagram Counting

An anagram is a reordering of the letters in a word of phrase. For example, you can rearrange the letters of terraced to get the word rethecer. Rearranging them some more will give you the word cratered. You can even make dactrere and redatrec, which are both anagrams of terraced even if they are not legitimate English words.

Input

Input contains up to 200 words, one per line. Each word consists of upper- and lower-case letters (a–z) and may have as many as 100 characters. Input ends at end of file.

Output

For every input word, output the total number of unique anagrams that can be made from it. For the purpose of this problem, upper- and lower-case letters are considered distinct.

Sample Input 1

```
at
ordeal
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
abcdABCDAabcd
```

Sample Output 1

```
2
5040
4032914611266056355804000000
49229914688306352000000
29937600
```