Algorithms and Problem Solving - II

- Algorithm Efficiency
- Primality Testing
- Improved Primality Testing
- Sieve of Eratosthenes
- Primality Testing - Applications
- Exercises
Algorithm Efficiency

• In the last unit we went through the definition and representation of an algorithm.

• The next question we face is: How to design an efficient algorithm for a given problem?

• Algorithm efficiency is concerned with utilization of two important resources
  – Time
  – Space

• Time complexity refers to the time taken by an algorithm to complete and produce a result. An improvement in time complexity increases the speed with which the algorithm proceeds to completion.

• Space complexity refers to the amount of space taken by an algorithm and produce a result. An improvement in space complexity reduces the amount of space (memory, disk space, etc.) taken by an algorithm.
To illustrate the concept of algorithm efficiency, we take up an important problem in computer science: Primality testing.

An integer \( \geq 2 \) is prime if and only if its only positive divisors are 1 and itself.

The specific version of the problem which we refer to is as follows: Given a positive integer \( n \) find all primes less than or equal to \( n \).

A list of all prime integers less than or equal to 100 is given below:

\[
\begin{array}{cccccccc}
2 & 3 & 5 & 7 & 11 & 13 & 17 \\
19 & 23 & 29 & 31 & 37 & 41 & 43 \\
47 & 53 & 59 & 61 & 67 & 71 & 73 \\
79 & 83 & 89 & 97 & & & \\
\end{array}
\]
Here is our algorithm in pseudocode:

Given an integer \( n \), for all positive integers \( k \) less than or equal to \( n \) do the following:

1. If \( k \) is prime, i.e. \( \text{isPrime}(k) \) is true, print \( k \).

Algorithm \( \text{isPrime}(\text{int } x) \)

1. For a given integer \( x \) do the following.
2. Assume that \( x \) is prime.
3. for all positive integers \( j \) less than \( x \) do the following
4. Divide \( x \) by \( j \). If the remainder after division is equal to zero for any division, \( x \) is not prime.
5. Return the status of primality of \( x \).
import java.io.*;
public class SimplePrimality
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter an integer: ");
        int limit = Integer.parseInt(in.readLine());
        int limit = Integer.parseInt(in.readLine());
        for(int count = 2; count <= limit; count++)
         
            if(isPrime(count)) System.out.print(count + "\t");
    }
    public static boolean isPrime(int number)
    {
        int i = 2;
        while(i < number)
        {
            if(number % i == 0)
                return false;
            i++;
        }
        return true;
    }
}
Testing the Program

- Test the program for input = 100, 1000, 10000, 100000, 1000000.
- On a sample run, it was observed that the algorithm takes much longer to complete for larger inputs. This is because the algorithm tests for primality of a large integer \( n \) by repeated divisions for all integers from 2 to \( n - 1 \), which makes the algorithm take longer time to complete for large values of \( n \).
- Is there any way we can improve this algorithm and make it faster?
Primality Testing - Improvements

• Observe that all prime numbers are odd, except for 2.

• Hence we do not need to divide a number with all even numbers. A division by 2 is enough to eliminate all even numbers.

• A second observation is that we do not need to divide an integer \( n \) by all integers upto \( n - 1 \) to check if it is prime or not.

• Instead we can test for primality only by dividing \( n \) by all odd integers from 3 to \( \sqrt{n} \), to test for primality.

• To see this observe that if a divisor of \( n \) is greater than \( \sqrt{n} \) the corresponding divisor must be less than \( \sqrt{n} \), since if both divisors are greater than \( \sqrt{n} \), then their product will be greater than \( n \), which is a contradiction.

• With these improvements the program becomes as follows:
import java.io.*;
public class FasterPrimality
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter an integer: ");
        int limit = Integer.parseInt(in.readLine());
        for(int count = 2; count <= limit; count++)
            if(isPrime(count)) System.out.print(count + "	");
    }

    public static boolean isPrime(int number)
    {
        boolean prime = true;
        if(number % 2 == 0 && number > 2)
            return false;
        int i = 3;
        int upper = ((int) Math.sqrt(number));
        while(i <= upper)
            {
                if(number % i == 0) return false;   i += 2; }
        return true;
    }
}
Testing the Program again

- Test the program for input = 100, 1000, 10000, 100000, 1000000.

- Observe the time taken by the program to complete. On a sample run, this program is much faster and efficient than the previous version, because the number of trial divisions has sharply reduced by more than half.

- Is there any way we can improve this algorithm further and make it even faster?
Primality Testing – A Classic Algorithm

- The **Sieve of Eratosthenes** (276 BC - 194 BC). is a classic algorithm to find all primes between 1 and $n$. The algorithm is as follows:
  1. Begin with an (unmarked) array of integers from 2 to $n$.
  2. The first unmarked integer, 2, is the first prime.
  3. Mark every multiple of this prime.
  4. Repeatedly take the next unmarked integer as the next prime and repeat step 3.
  5. Stop marking at multiples when the last unmarked multiple is greater than $\sqrt{n}$.
  6. After all markings are done, all unmarked integers are primes between 2 and $n$. 
Sieve of Eratosthenes - Example

• For example, our beginning array is:
  – (Unmarked)  2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  – (After step 1)  2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  – (After step 2)  2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
    (All even multiples of 3 are already marked)
  – (After step 3)  2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
  – The final list of primes is 2, 3, 7, 11, 13, 17, 19.

• Note that this algorithm takes *more space resources* than either of the previous two algorithms.

• Is this algorithm more time efficient than the previous two algorithms?
Primality Testing - Applications

- Primality Testing these days is used in computer security algorithms.

- Most notable of these is the RSA Algorithm which relies on the apparent difficulty of factoring a large integer.

- If you can find the factors of the following number you could win US$200,000:

  2519590847565789349402718324004839857142928212620403202777713783604366202
  07075955562640185258807844069182906412495150821892985914976184502808489
  1200728449926873928072877767359714183472702618963750149718246911650776133
  7985909570009733045974880842840179742910064245869181719511874612151517265
  4632282216869987549182422433637259085141865462043576798423387184774447920
  7399342365848238242811981638150106748104516603773060562016196762561338441
  4360383390441495263443219011465754445417842402092461651572335077870774981
  7125772467962926386356373289912154831438167899885040445364023527381951378
  636564391212010397122822120720357

- Check [www.rsalabs.com](http://www.rsalabs.com) for further details.
Exercises

1. Can you convert both example 1 and example 2 to find and print all composite integers between 2 and $n$? Will it have any effect on the efficiency of the algorithm. Note that a composite integer is an integer that is not prime.

2. Implement the pseudocode for the Sieve of Eratosthenes Algorithm in Java.

3. Check the java library java.math. Is there a method for primality testing? (Hint: Look for the class BigInteger). How slow or fast is it as compared to our method isPrime(int x) in Example 2?