PiE C++ Final Assignment

Name: Duc Nguyen,
student number: s1630512*

November 8, 2015

---

*Email address for correspondence: nguyenmanhduc@student.utwente.nl or ngmaduc@gmail.com

Exercise 1

Question 1.1

Write a *C++* program to compute the first N prime numbers, where N is given by the user. Use dynamic arrays to store the primes and use this information in the mod test.

Answer.

Three functions are used for this question including `bruteForce`, `modTest` and `print_primes` for searching prime numbers and printing out the result in console. Passing by reference are chosen to avoid unnecessary copy of variables.

```cpp
std::vector <unsigned long int> bruteForce (int &n);
std::vector <unsigned long int> modTest (int &n);
void print_primes (int &n, const std::vector <unsigned long int>& primes);
```

The method chosen to create dynamic arrays to store the primes is std::vector. The range $[0, 2147483647]$ of unsigned long int fits to the scope of the question. The idea of `bruteForce`, `modTestDiv` are shown in the following code snippets.

```cpp
bruteForce

        primes.push_back(2);
         unsigned long int c;//need to be the type of primes for mode test
         int count = 1;
         for (int count = 1; count < n; ){//counter from 1; "2" included before
              for (c = 2; c < num; c++){
                   if (num % c == 0) {//mod test from 2 to n
                        break;
                   }
              }
              if (c == num) {//to this point means no divisor up to n, Prime!
                   primes.push_back(num);//push to result vector of Prime
                   count++;//increase counter
              }
              num++;
         }
```

```cpp
modTestDiv

        primes.push_back(2);
         for (int count = 1; count < n; ) {//counter from 1; "2" included ↩
              before
              bool isPrime = true;
              for (int i = 0; i < primes.size(); i++){
```

```
5                    if (num % primes[i] ==0) {//non−primes   are  products  of  primes
6                      isPrime = false;
7                      break;
8                    }
9                }
10               if (isPrime == true) {
11                   primes.push_back(num);
12                   count++;//increase  counter
13               }
14               num++;
15           }
```

Question 1.2

Write to the screen a list of the first 10000 primes in the format below; where $p(n)$ is the nth prime number. Report only the last five lines. Comment on the behaviour of the ratio $n * ln(p(n))/p(n)$ as n gets large.

Answer.

The void `print_ratio` (int &n, const std::vector <unsigned long int>& primes) and prime number search functions together generate the required ratio. As n gets large, the ratio tends to converge to 1. Until $10^5 - th$ prime number, the ratio is 1.103.

The last five lines are and the `print_ratio` are listed below:

```
1  9996       :        104707  :        1.10348856177824
2  9997       :        104711  :        1.10356044403989
3  9998       :        104717  :        1.10361306655082
4  9999       :        104723  :        1.10366568381267
5  10000      :        104729  :        1.10371829582629
```

```
print_ratio

1  void print_ratio (int &n, const std::vector <unsigned long int>& primes){
2      std::cout << "n\t:\t p(n)\t:\t n*ln( p(n) )/p(n) " << std::endl;
3      for (int i = primes.size; i < primes.size(); i++) {
4          std::cout << i+1 << "\t:\t" << primes[i] <<"\t:\t"
5              <<std::fixed << std::setprecision(14) <<
6                  double((i+1)*log(primes[i])/primes[i])<< std::endl;
7      }//set precision used for increase decimal displayed
8  }
```

Question 1.3

Based on question 2, give an estimate of the $10^6 - th$ prime number.

Answer.

We use 1.1 for the value of the ratio with $n = 10^6$:

$$10^6 * ln(p(10^6))/p(10^6) \approx 1.1$$

Using Wolfram Alpha to solve this equation, the estimate of the $10^6 - th$ prime number is:

$$p(10^6) \approx 15022800$$

Question 1.4

Instead of writing to the screen, write to a file (on disk) a list containing just the prime numbers. Print eight numbers per line, such that all numbers have the same space.

Answer.

The `primes_to_file` function for writing to a file (on disk) with eight numbers per line is shown below:

```
primes_to_file
1  void primes_to_file (int &n, const std::vector <unsigned long int>& primes,
2         const std::string& fileName){
3         if(n >= 1){
4             std::ofstream fileOut;
5             fileOut.open(fileName);
6             for (int i = 0; i < primes.size(); i++) {
7                 fileOut << primes[i] << "\t";
8                 if ((i+1) % 8 == 0) {//Print eight numbers per line
9                     fileOut<<std::endl;
10                }
11            }
12        }
13        else{
14            std::cerr << "Invalid Input" << std::endl;
15        }
16 }
```

Question 1.5

Time your code for $N = 10^3; 10^4; 10^5$ and 10^6. Make a log-log plot of run-time against N for both codes. What can we say from the log-log plot? Do this analysis for brute force and suggested speed up and comment on the results.

Answer.

Figure (**??**) is the log-log plot of run-time against N showing that within the range of N the linear relationship of logarithmic values of run-time and N which also mean the running time is propotional to N to the power of the slope of the straight line of the log-log graph. The search function `modTestDiv` is faster than `bruteForce`. However, the `bruteForce` can be simply modified to `bruteForceM` which results in faster running time. The only modification is instead of performing mod test from 2 to n for each number n, we only do mod test from 2 to square root of n, because number n is not a prime number means that it always has a divisor less than or equal square root of n. Otherwise, the number n is a prime number.Measurement of running time is recored by using `std::chrono::time_point<std::chrono::system_clock>`. The running times are then saved to a file for further processing. All the codes are measured with the function `rtime_to_file`. Code snippets for `rtime_to_file` and `bruteForceM` are shown below:
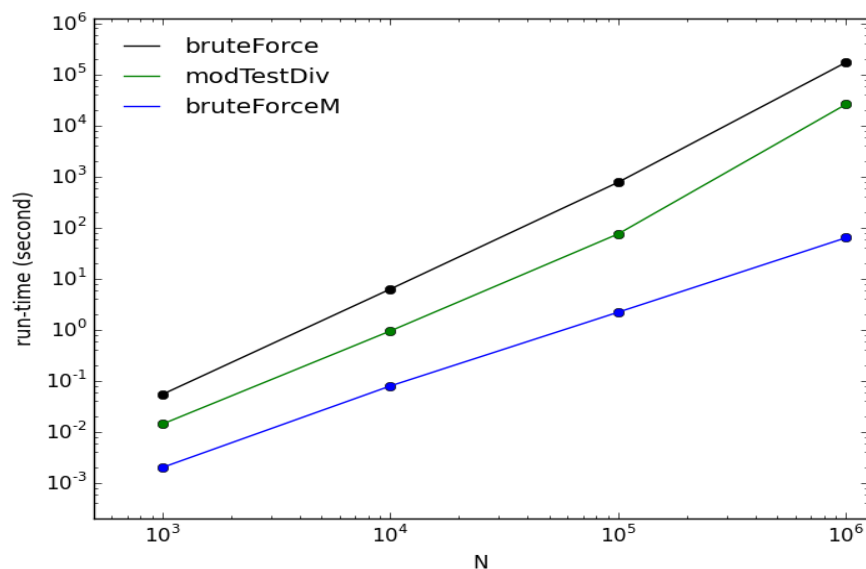
Figure 1: log-log plot of run-time against N

```
rtime_to_file

1 void rtime_to_file (const std::string& fileName,
2     std::function<std::vector <unsigned long int >(double &n)> &f){
3     ...
4     for (int i = 0; i < v.size(); i++) {
5         std::chrono::time_point<std::chrono::system_clock> start, end;
6         start = std::chrono::system_clock::now();
7         std::vector<unsigned long int> primes = f(v[i]);//only measure this ↩
            function
8         end = std::chrono::system_clock::now();
```

```
9            std::chrono::duration<double> elapsed_seconds = end−start;
10     ...
11      }
12  }
```

bruteForceM

```
1  for (int count = 2; count <= n; ){//counter from 1; "2" included before
2              for (c = 2; c <= sqrt(num); c++){//non prime has divisor lt sqrt
3                  ...
4              }
5              if (c > sqrt(num)) {//to this point no divisor up to n, Prime!
6                  ...
7              }
8              num++;
9          }
```

Question 1.6

More efficient ways of computing prime numbers exist. Find and implement one and report
the analysis of part 5 for this algorithm. Comment on the results.

Answer.

Sieve of Eratosthenes is used for illustrating an efficient way to search for prime numbers.
The log-log plot of run-time against N is shown in Figure (**??**).
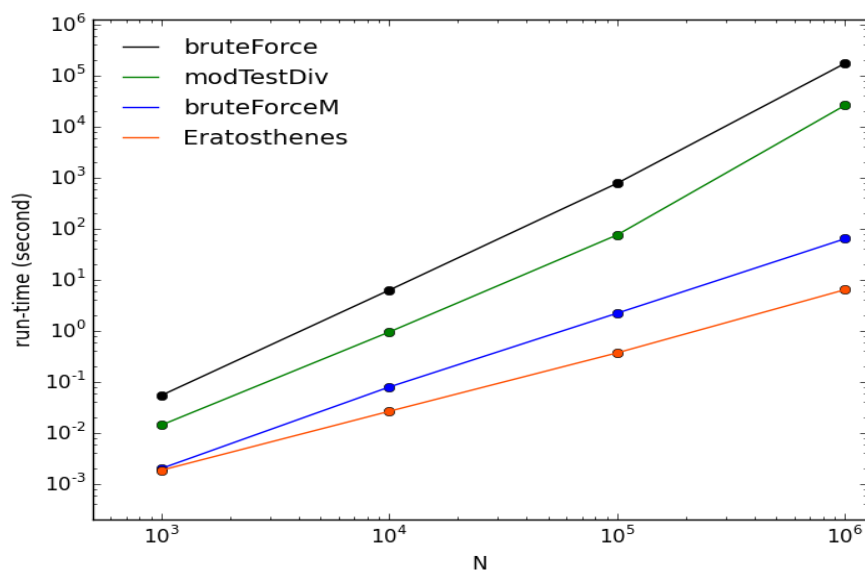

Figure 2: log-log plot of run-time against N

The idea is just to mark all the number in the range of interest as prime then update the mark as non-prime if a number is the multiplication of the previous primes. It illustrate the fundamental property of prime number that any non-prime number can be represented by a multiplication of prime numbers. The code snippet implementing Sieve of Eratosthenes is shown below. The Sieve of Eratosthenes requires the largest number to be found as an input. We can use the ratio from Question 1.3 or Rosser's theorem which states that $p(n) < n*log(n*log(n))$. For searching large prime numbers, the linear relationship will not be held. Further study on complexity of algorithm showing that is beyond the scope of this report. However, a simple test shows that for only 7 digits prime number, high complexity method like `bruteForce` already takes the order of days. Therefore, it should not be used for finding large prime numbers.

```
Eratosthenes
1    unsigned long int max = n * std::log(n*std::log(n)); //Rosser s theorem
2     for (unsigned long int p=2; p < max; p++){ // for all elements in array
3          if (primes.size() > n-1){//keep track first n prime only, vector ↩
               count from 0
4              break;
5          }
6          else if(isPrime[p] == true){ // it is not multiple of any other prime
7              primes.push_back(p);
8          }
9          // mark all multiples of prime selected above as non primes
10         int c=2;
11         int mul = p * c;
12         while(mul <= max){
13             isPrime[mul] = false;
14             c++;
15             mul = p*c;
16         }
17      }
```

Exercise 3

Question 3.1

Use any algorithm to compute the shortest distance between every set of cities and write this information to disk with the route as a list of cities..

Answer.

The question ask for the shortest distance between every set of cities thus Floyd-Warshalls algorithm which is well-known for solving to solve the All-Pairs-Shortest-Path problem is chosen. This report will try to cover key points of the algorithm as well as explain how to

implement it it *C++*. The details of Floyd-Warshalls algorithm used for the code in this exercise is described in Graph Theory reference. (**?**)

The input file given is very suitable for the algorithm because it is already in the form of adjacency matrix (distance matrix) which give all information about the cities and how they are connected. They are also known as nodes and edges' length in graph theory. The distance matrix d is stored by 2-D vector std::vector<std::vector<int>> d. To reconstruct the shortest path, we also need a node sequence matrix s which is stored by a 2-D std::vector<std::vector<int>> s. All the elements of node sequence matrix s which means that initially the shortest path is the direct connection from citi i to city j. The size of both vector are the square of nodes. The functions lines_count and input2vector are used to obtain the number of nodes and create the distance matrix, respectively. We are now ready to implement Floyd-Warshalls algorithm which is done by the function WFI.

```cpp
int lines_count (const std::string& fileName);
std::vector<std::vector<int>> input2vector (const std::string& fileName);
void WFI(int &nodes, std::vector<std::vector<int>>& d,
        std::vector<std::vector<int>>& s);
int main()
{
    ...
    std::vector<std::vector<int>> s(nodes,std::vector<int>(nodes, 0));
    ...
}
```

The code snippet for the implementation of lines_count and input2vector are shown below:

```cpp
lines_count
std::ifstream fileIn(fileName);
    int n =0;
    std::string line;
    while (std::getline(fileIn , line)){
        n++;//increase n after each line
    }
```

```cpp
input2vector
while (std::getline(fileIn , line)){
        std::vector<int> lineData;
        std::istringstream  lineStream(line);
        int value;
        // Read an integer at a time from the line
        while(lineStream >> value){
```

```
7              // Add the integers from a line to a 1D vector
8              lineData.push_back(value);
9          }
10         // When all the integers have been read add the 1D array
11         // into a 2D array (as one line in the 2D array)
12         d.push_back(lineData);
13     }
```

Floyd-Warshall's algorithm is a recursive algorithm which updates distance matrix d and node sequence matrix s in each step. We will run a loop from the first node to the last note. For each iteration, it simply tells whether a city k needs to be included for shortest in the path between city i is city j. It will be updated in node sequence matrix s as part of the shortest path and will be permanently as part of the shortest path. The algorithm behave like a greedy algorithm as it prefers more nodes and shortest path. We need the latest status of distance matrix d since it stores the current shortest distance between city i is city j. It will only be updated when the current shortest distance from is less than the distance if we include a new node k in the shortest path when we compare the distance in each iteration. There is one small problem with this algorithm that it could not record multiple current shortest distance when it happens to be the equality case for the comparision of the iteration. In this case, I choose to include the one with more nodes in the the shortest paths. The implementaion of Floyd-Warshalls algorithm is shown below:

```
WFI
1    for (int k = 1; k <= nodes; k++){
2        for (int i = 1; i <= nodes; i++){
3            for (int j = 1; j <= nodes; j++){
4                //If the path with two edges is less than the path with one ↩
                     edge
5                //node that algorithm is from 1 but vector is from 0
6                if(i!=k && j!=k && i!=j){
7                    if (d[i-1][j-1] >= (d[i-1][k-1] + d[k-1][j-1])){
8                        //choose the one with more nodes
9                        //Set the cost of the edge to be the lesser cost.
10                       d[i-1][j-1] = (d[i-1][k-1] + d[k-1][j-1]);
11                       //This ensures proper path reconstruction.
12                       //at this point increase to k+1 to continue algorithm
13                       s[i-1][j-1] = k;
14                   }
15               }
16           }
17       }
18   }
```

Reconstructing the shortest path is like going in a zig-zag in the latest node sequence matrix s. The element $s[i][j]$ will tell if the node k is needed to be included in the shortest path. If

there is a k, the recuresion will be done for $s[i][k]$ and $s[k][j]$ and it will stops when $k = 0$ which is the base case. The function`path_recon` is used for path reconstruction and it is shown as below:

```
path_recon
```
```cpp
void path_recon(std::ofstream& fileOut1, int& n1,int& n2,
        std::vector<std::vector<int>>& s){
    int k;
    k = s[n1-1][n2-1];//start and destination, remember to -1 in the index
    if (k != 0){
        path_recon(fileOut1,n1,k,s);//recursive
        fileOut1 << " - "<< k;//add - to separate new nodes
        path_recon(fileOut1,k,n2,s);//recursive
    }
}
```

To record all the shortest distance and the shortest path to a file, now all we need to do is to run a loop so that it covers all the pair of nodes then run `WFI` and `path_recon` This will be done in the implemetation below:

```cpp
for (int i = 1; i <= d.size(); i++){
            for (int j = i+1; j <= d[i].size(); j++){
                fileOut1 << i;
                path_recon(fileOut1,i,j,s);
                fileOut1 << " - " <<j;
                fileOut1 << "\t\t" << d[i-1][j-1];
                fileOut1 << std::endl;
            }
        }
```

References

RAY, SANTANU SAHA 2013 *Graph Theory with Algorithms and Its Applications*. Springer, India, Private Ltd.