The Compendium, v0.1
Texas A&M University

Jack Graham

November 3, 2017

Contents

Chapter 1

The Basics

1.1 Template

The Mother Program

Every C++ program should begin with this. The `#include` statement will bring in the entire C++11 STL when compiled with `g++`, and the strange I/O statements will prevent standard IO from syncing with each other. This will speed up a program that reads from stdout/stdin substantially, at the cost of making `scanf` and `printf` completely unusable, so use only if you plan on using purely `cin` and `cout`.

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    // code goes here

    return 0;
}
```

Makefile

To compile programs, use the following makefile, which incorporates the exact command used by the judges:

```
    main: main.cpp
        g++ -g -lm -lcrypt -O2 -std=c++11 main.cpp -o main
    clean:
        rm main
```

1.1.1 Important Note on Snippets

The snippets in this book are **ALL** written under the assumption that they are being inserted into the skeleton delineated above, i.e. they operate assuming full access to all headers in the C++11 STL using the `std` namespace. **DO NOT USE THEM** without this. You have been warned!

Chapter 2

Number Theory

Unless explicitly stated otherwise, numbers are assumed to belong to $\mathbb{N}$.

2.1 Primes

2.1.1 Prime Facts

Definitions

A number p is *prime* if its only divisors are 1 and itself. Two numbers a and b are *coprime* if $GCD(a, b) = 1$.

The Prime Number Theorem

$\pi(x)$, the number of primes less than some x, grows at almost exactly the same rate as $\frac{x}{\log x - 1}$.

2.1.2 The Sieve of Eratosthenes

The sieve of Eratosthenes is an extremely efficient and useful algorithm, finding all primes up to n in $O(n \log \log n)$.

```
void prime_sieve(bool* sieve, int n) {
    // populates sieve with each prime < n
    // PRECONDITION: sieve has size < n
    sieve[0] = false;
    sieve[1] = false;
    for (int i = 2; i < n; ++i) sieve[i] = true;
    for(int i = 2; i < (int)sqrt(n); ++i)
        if (sieve[i]) for (int j = i*i; j < n; j+=i)
            sieve[j] = false;
}
```

Generally, the obvious use is sufficient. However, sometimes, e.g. when checking a few very large numbers for primality, it is best to create a prime sieve of size $\sqrt{n}$, where n is the largest number, then use the sieve to create a sorted vector of all primes up to $\sqrt{n}$ and test each of these individually. This technique is also helpful for efficient factorization of large numbers (in $O(logn)$) through continuous division.

```
vector<int> primes_to(int maxp) {
    // returns a vector of all primes <= maxp
    ++maxp;
    vector<int> v;
    bool sieve[maxp];
    prime_sieve(&sieve, maxp);
    for (int i = 0; i < maxp; ++i)
        if (sieve[i]) v.push_back(i);
    return v;
}
```

2.1.3 Euler's Totient Function

Euler's totient function $\phi(n)$ for some n is defined as the number of integers less than n that n is relatively prime with, i.e. whose GCD with n is 1. Computing this in $O(\sqrt{n})$ is easy, and the following solution can easily be optimized for many queries.

$\phi(n)$ has many interesting properties, but the most famous and useful is the fact that $a^{\phi(n)} \equiv 1 \pmod{n}$ for coprime a, n.

```
int phi(int n)
{
    int result = n;
    for (int p : primes_to((int)sqrt(n)))
    {
        while (n % p == 0) {
            n /= p;
            result -= result / p;
        }
    }

    if (n > 1)
        result -= result / n;
    return result;
}
```

2.2 Modular Arithmetic

2.2.1 Euclidean Algorithm

Simple $O(\log min(a, b))$ algorithm to find the GCD of two numbers a and b, or LCM.

```
int gcd(int a, int b) {
    int t;
    while (b != 0) {
        t = b;
        b = a % b;
        a = t;
    }
    return a;
}

int lcm(int a, int b) {
    return a*b / gcd(a, b);
}
```

2.2.2 Modular Inverse

The **extended Euclidean algorithm** retains the sublinear time complexity of Euclid's simple algorithm, and for almost no extra cost finds integer coefficients x, y for the equation $ax + by = GCD(a, b)$.

```
int euclid(int a, int b, int *x, int *y) {
    // Extended Euclidean algorithm
    // base case
    if (a == 0) {
        *x = 0;
        *y = 1;
        return b;
    }

    int x1, y1; //store recursive call
    int gcd = euclid(b%a, a, &x1, &y1);

    *x = y1 - (b/a) * x1;
    *y = x1;

    return gcd;
}
```

This equation is useful because it allows us to find the **modular inverse** of two numbers a and m, i.e. the number x such that $ax \equiv 1 \pmod{m}$, which can then be used for other powerful results, like the Chinese remainder theorem. The reason this works is that a modular inverse for $a \pmod{m}$ is only possible if

a and m are coprime, i.e. $gcd(a, m) = 1$. Thus, if $ax + my = 1$, $ax - 1 = (-y)m$, and thus $ax \equiv 1 \pmod{m}$.

```
int inv(int a, int m) {
    int x, y;
    euclid(a, m, &x, &y);
    return x;
}
```

2.2.3 Chinese Remainder Theorem

The **Chinese remainder theorem** enables us to solve for the lowest possible x satisfying a system of equations of the form $x \equiv a \pmod{d}$, for two equally-sized vectors a and d, provided each pair of elements in d is coprime. This is possible using Euclid's extended algorithm to find the inverse GCD.

```
int crt(vector<int> d, vector<int> a) {
    // finds smallest x such that x%d[i] = a[i] for all i
    //PRECONDITION: d.size()==a.size()
    // and all elements in d are coprime with all others
    int product = 1;
    for (int i = 0; i < d.size(); ++i) {
        product *= d[i];
    }
    int result, pp;
    result = 0;
    for (int i = 0; i < d.size(); ++i) {
        pp = product / d[i];
        result += a[i] * inv(pp, d[i]) * pp;
    }
    return result % product;
}
```

2.2.4 Legendre's Formula

Given some prime number p and some n, the largest number x such that $n!$ is evenly divisible by p^x is

$$\sum_i \left\lfloor \frac{n}{p^i} \right\rfloor$$

.

Chapter 3

Geometry

Chapter 4