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`.

```cpp
#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

2.1 Modular Arithmetic

2.1.1 GCD and 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);
}
```

2.1.2 Chinese Remainder Theorem

2.1.3 Legendre's Formula

Given a prime number p and natural number n, the largest natural number x such that $n!$ is evenly divisible by p^x is $\sum_i \left\lfloor \frac{n}{p^i} \right\rfloor$.

2.2 Primes

2.2.1 Prime Facts

The Prime Number Theorem

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

2.2.2 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;
}
```

]