# The Compendium

Jack Graham
Texas A&M University

November 6, 2017

# Contents

# Chapter 1

# The Basics

## 1.1 Setup

### 1.1.1 Vim

These commands enable syntax highlighting, line-numbering, and auto indentation, and change tab spacing to 4.

```
 $ vim  /.vimrc
syntax on
set ai
set number
set ts=4
```

### 1.1.2 Using the Template

Use `mkdir /tmp/code/ && cd /tmp/code/ && vim template.cpp`, then type the following:

```cpp
#include <iostream>
#include <sstream>
#include <iomanip>
#include <algorithm>
#include <cmath>
#include <vector>
#include <map>

using namespace std;

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

    // code goes here
```

```
    return 0;
}
```

Then, initialize the directory as follows:

```
for x in a b c d e f g h i j k; do cp template.cpp $x.cpp && touch
$x.dat; done
```

### 1.1.3 Makefile

Use `vim Makefile`, then enter the following:

```
CXX=g++
CXXFLAGS=-Wall -static -g -O2 -std=gnu++14
a:  a.cpp
...
k:  k.cpp
```

This allows you to type `make a && ./a < a.dat` to make and run problem A in one go. The compiler flags used are the same ones used by the SCUSA regionals judge in 2017, plus `-Wall` to help with debugging.

# Chapter 2

# Number Theory

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

## 2.1 Definitions

- A number $p$ is *prime* if its only divisors are 1 and itself.
- Two numbers $a$ and $b$ are *coprime* if $gcd(a, b) = 1$.
- A *perfect number* $n$ is equal to the sum of its proper positive divisors (i.e. the divisors less than $n$). All known perfect numbers are even, and of the form $q(q + 1)/2$ where $q$ is a *Mersenne prime* (a prime of the form $2^p - 1$ for some prime $p$).

## 2.2 Primes

### 2.2.1 Primes Library

```
// Jack Graham 2017
// primes.cpp

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

/*  prime_sieve sieves for all primes < n
    PRECONDITION:
        sizeof(sieve) < n, n > 0
    RUNTIME:
        O (n log log n)
```

```
*/
void prime_sieve(bool* sieve, int 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;
}

/*  primes_to returns a vector of all primes <= n

    PRECONDITION:
        n > 0
    RUNTIME:
        O(n log log n)
*/
vector<int> primes_to(int n) {
    vector<int> v;
    bool sieve[++n];
    prime_sieve(&sieve[0], n);
    for (int i = 0; i < n; ++i)
        if (sieve[i]) v.push_back(i);
    return v;
}

/*  Euler's totient function
    PRECONDITION:
        n > 0
    RUNTIME:
        O(prime factors of n + sqrt(n))
*/
int phi(int n)
{
    int result = n;
    int s = (int)sqrt(n);
    bool sieve[s];
    prime_sieve(&sieve[0], s);
    for (int p = 0; p < s; ++p) {
        if (sieve[p]) { // p is prime
            while (n % p == 0) {
                n /= p;
                result -= result / p;
            }
        }
    }
    if (n > 1)
        result -= result / n;
    return result;
}
```

### 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 $O(n \log \log n)$.

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 $O(\log n)$) through continuous division.

### 2.2.3 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 $O(\log n)$ 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 $a$, $n$.

#### The Prime Number Theorem

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

## 2.3 Modular Arithmetic

### 2.3.1 GCD and LCM

```
// Fast GCD(a, b), O(log(min(a,b)))
int gcd(int a, int b) {
    int t;
    while (b != 0) {
        t = b;
        b = a % b;
        a = t;
    }
    return a;
}

// LCM(a, b), O(log(min(a,b)))
int lcm(int a, int b) {
    return a*b / gcd(a, b);
}
```

### 2.3.2 Euclid Codebase

```
// Jack Graham 2017
// euclid.cpp

#include <vector>

typedef long long u64;

using namespace std;

/*  Efficient extended Euclid's algorithm
    Returns GCD(a, b) with integer solutions for
    xa + by == GCD(a, b)
    PRECONDITION:
        a, b > 0
    RUNTIME:
        O(log(min(a, b)))
*/
int euclid(int a, int b, int *x, int *y) {
    if (a == 0) { // base case
        *x = 0;
        *y = 1;
        return b;
    }
    int x1, y1; //store recursive call here
    int gcd = euclid(b%a, a, &x1, &y1);

    *x = y1 - x1*b/a; // these calls bubble down
    *y = x1;

    return gcd;
}

/*  Modular inverse using euclid()
    Returns x s.t (x*a)%m == 1
    PRECONDITION:
        a, m > 0
    RUNTIME:
        O(log(min(a, m)))
*/
int inv(int a, int m) {
    int x, y;
    euclid(a, m, &x, &y);
    return x;
}

/*  Chinese Remainder Theorem implementation
    Finds the smallest x such that x%d[i] = a[i] for all i
```

```
    PRECONDITIONS:
        d.size()==a.size()
        all elements in d are coprime with all others
    RUNTIME:
        O(n log P) where P is product of all numbers in d
*/
int crt(vector<int> d, vector<int> a) {
    u64 product = 1; //product across all d
    for (int i = 0; i < d.size(); ++i) {
        product *= d[i];
    }
    u64 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 (int)(result % product);
}
```

---

### 2.3.3 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 $ax + by = gcd(a, b)$.

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 $ax \equiv 1 \pmod{m}$.

### 2.3.4 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.

### 2.3.5 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 $\sum \left\lfloor \frac{n}{p^i} \right\rfloor$.

# Chapter 3

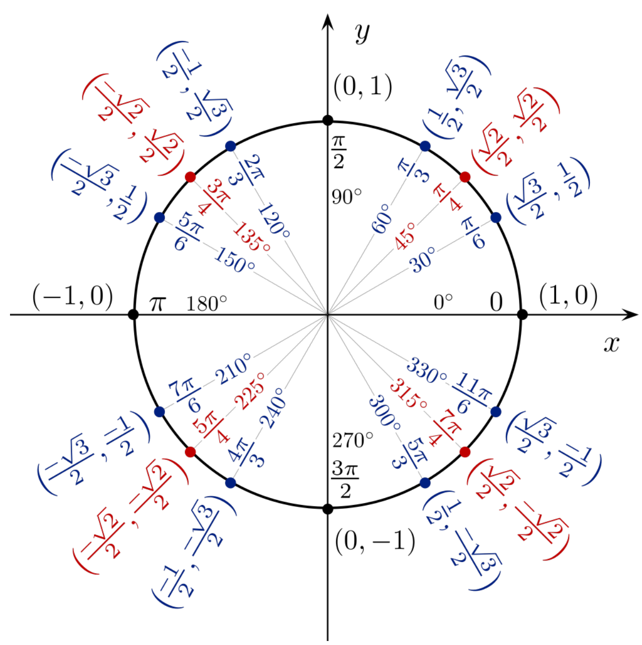
# Geometry

## 3.1 Points

## 3.2 Lines

## 3.3 Trigonometry

### 3.3.1 The Unit Circle



]

## 3.4 Polygons

## Chapter 4

# Linear Algebra