The Compendium

Jack Graham
Texas A&M University

March 25, 2018

Contents

Chapter 1

The Basics

1.1 Setup

1.1.1 Vim

My preferred lightweight setup file. The macro at the end makes F5 save, compile, and run the program (pretty neat!)

```
 $ vim  /.vimrc
syntax on
set ai
set nu
set ts=4
set sw=4
set is
set hs
set bg=dark
nnoremap <silent> <F5> :w <bar> :make <bar> :!./main < in.dat
```

1.1.2 Using the Template

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

```cpp
#include <bits/stdc++.h>‘
using namespace std;

typedef long long ll;
typedef vector<int> vi;

int main() {
    // 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.

1.2 Choice of Programming Language

About 10^6 operations are doable by Python in 30 seconds, while 10^9 operations can be done by C++. Therefore, Python should only be used if integers are very big (Python's integers are arbitrarily large), if decimals require arbitrary precision (`import decimal` will take care of that.), or for problems where speed is not an issue that involve something like string formatting. The judge guarantees problems to be solvable in C++, but does not guarantee them to be solvable in Python. For this reason, the C++ STL is a far better choice for competitive programming, with vast majority of code written here is in C++.

Chapter 2

The C++ STL

2.1 Input and Output

The bread and butter of every program. This is what you use to read from stdin and write to stdout, in the form of `cin` and `cout`. I use the headers `<iostream>`, `sstream`, and `<iomanip>` for this purpose.

2.1.1 Output Tricks

`<iomanip>` is kind enough to provide us with enough functionality for `cout` and other output streams to be as flexible as `printf`. These are passed directly to `cout` or any other output stream (e.g. `cout << setw(6)`), and include (but are not limited to):

- `setbase(int n)`: changes the base that numbers are outputted in to `n`, so long as `n` is 8, 16, or 10. If you want any other base, you're going to have to write your own code.[1]. If you want numbers to display their base (e.g. `0x1c`) you can pass `cout << showbase`, and if you stop wanting that, `cout << noshowbase`.
- `setw(int n)`: sets the width of the next singular object passed to `cout` to `n`. Note that things are right-justified by default; the statement `cout << left` will change that, and `cout << right` will change it back.
- `setfill(char c)`: changes the fill character (`' '` by default).
- `setprecision(int n)`: changes the decimal precision for subsequent floating-point values. By default, this is the number of significant digits which is

[1] Although base 2 is conspicuously absent from `setbase`, there is another solution. The idiomatic way to output an e.g. 32-bit `int`'s binary representation is the moderately clunky `int x = 53; bitset<32>(x) y; cout << y;`.

almost never what you want; pass `cout << fixed` first to alleviate this issue.[2]

- `flush`: flushes the output buffer. I define `endl` to be `"\n"`, trading its flushing functionality for speed, so if you need flushing for some reason, here it is.

2.1.2 Input Tricks

`cin` uses the same `<iomanip>` flags as `cout`, but with a few twists. For one, using `setbase(0)`, which is the default value, allows `cin` to infer the base of integers based on what precedes them (e.g. `0x` for hex). Another useful flag is `ws`; passing this to `cin` eats the whitespace.

Another useful trick for input is `getline(cin, s)` where `s` is some string. This puts the entire line, excluding the trailing newline, into `s`, which can then be checked to see if it is empty. A `stringstream` can then be constructed around `s` to process it; but beware as this is pretty inefficient, so if there's a lot of input it might TLE.

For the annoying problems that don't specify how many test cases there are ahead of time, `getline(cin, s)` or `cin >> x` can be wrapped in a `while` loop's condition, which will terminate at EOF.

2.2 Algorithms

2.2.1 Note on Functors

For many functions in `<algorithms>`, the default behavior can be easily overridden with a custom functor. For example, `sort()` sorts in ascending order by default, but passing the additional argument `greater<T>()` when sorting a `vector<T>` would sort in descending order instead. `greater<T>()` is a functor, which is just a class that defines `operator()` and can thus be used as a function.

If you wanted to sort by a more complicated characteristic, you can define your own functor for that. The following code sorts a `vector<pii>` in descending order of the first element, with ties broken by ascending order of the second element:

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pii;

struct Compare {
  bool operator() (const pii& a, const pii& b) {
    return pii(-a.first,a.second) < pii(-b.first,b.second);
  }
```

[2] If, for some reason, you find yourself wanting to represent floating-point numbers differently, pass `cout << scientific` for scientific notation, `hexfloat` for hexadecimal, or `defaultfloat` for the default float notation.

```
};

int main() {
  vector<pii> pairs = {{1, 2}, {3, 3}, {3, 4}, {1, 3}, {5, 4}};
  sort(pairs.begin(), pairs.end(), Compare());
  // pairs is now {{5, 4}, {3, 3}, {3, 4}, {1, 2}, {1, 3}}
}
```

I denote the functions in `<algorithms>` for which an optional functor argument can be passed at the end with an asterisk.

2.2.2 Sorted Data

For sorted `vector<T>`s `v` and `w`, and `T k`:

- *`binary_search(v.begin(), v.end(), k)` checks if `v` contains `k` ($O(\log n)$)
- *`equal_range(v.begin(), v.end(), k)` returns a pair of iterators in `v` that span the range of values equal to `k` ($O(\log n)$)
- *`lower_bound(v.begin(), v.end(), k)` returns an iterator pointing to the first element in `v` which is not less than `k` [3] ($O(\log n)$)
- *`upper_bound(v.begin(), v.end(), k)` is the same as `lower_bound`, except it finds the first value that compares greater than `k` ($O(\log n)$)
- *`merge(v.begin(), v.end(), w.begin(), w.end(), a.begin())` merges `v` and `w` into one sorted vector, putting the result into `a` ($O(n)$)
- *`set_union(v.begin(), v.end(), w.begin(), w.end(), a.begin())` finds the union of `v` and `w`, putting the result into `a` ($O(n)$). Other useful, similar functions are *`set_intersection`, *`set_difference`, and *`set_symmetric_difference`.

2.3 Containers

[3] An interesting use of this function is, on sorted or unsorted lists, finding the first element `x` that does not satisfy `P(x, k)` for some predicate function `P`.

Chapter 3

The Python Standard Library

3.1 Input and Output

Usually, `input()` is used for input and `print()` is used for output. Sometimes, however, you need to continuously get input until EOF. In this case, you can often use `fileinput` like this:

```python
import fileinput

for line in fileinput.input():
    print(line)
```

3.2 Hello

Chapter 4

Number Theory

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

4.1 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).

4.2 Primes

4.2.1 Primes Library

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

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

using namespace std;

/*  prime_sieve takes a blank sieve and populates it
    with primes
    PRECONDITION:
        sieve.size() >= 2
    RUNTIME:
```

```
        O (n log log n)
*/
void prime_sieve(vector<bool>& sieve) {
    fill(sieve.begin(), sieve.end(), true);
    sieve[0] = false;
    sieve[1] = false;
    for(int i = 2; i*i < sieve.size(); ++i)
        if (sieve[i]) for (int j = i*i; j < sieve.size(); 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<bool> sieve(n);
    vector<int> primes;
    for (int i = 2; i*i < n; ++i) if (!sieve[i]) {
        primes.push_back(i);
        for (int j = i*i; j < n; j+=i) sieve[j] = true;
    }
    return primes;
}

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

```
}
```

4.2.2 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.

4.2.3 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 $\frac{x}{\log x - 1}$.

4.3 Modular Arithmetic

4.3.1 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);
}
```

4.3.2 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);
}
```

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

4.3.4 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.

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

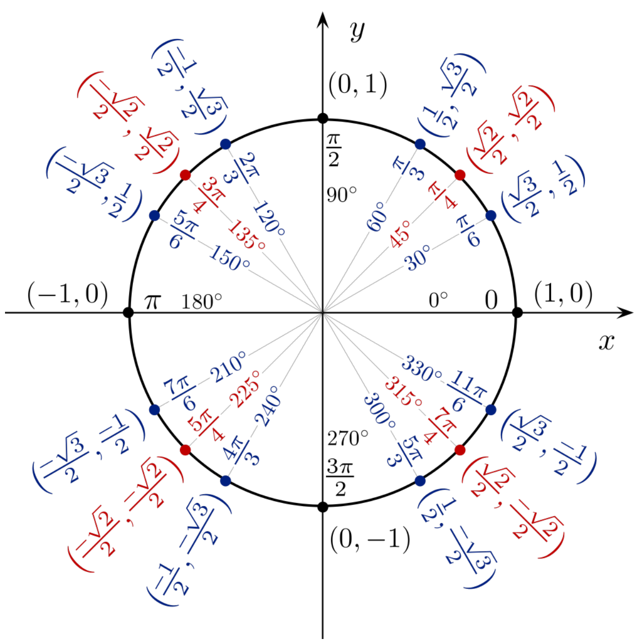
Chapter 5

Geometry

5.1 Euclidean Geometry

5.2 Trigonometry

5.2.1 The Unit Circle



5.3 Polygons

Chapter 6

Linear Algebra