Review 1: C++, Time Complexity, Modular Arithmetic

Veteran Track

Gabee De Vera

C++ Prerequisites

C++ Prerequisites

- Before proceeding, you must know the following:

1. Include Directives
2. Input/Output (`cin`, `cout`)
3. Basic Data Types (`bool`, `int`, `long long`)
4. Control Structures (`if-else if-else`, `for`, `while`, `do-while`)
5. Strings and common string methods (`std::string`)
6. Vectors and common vector methods (`std::vector`, `push_back`)
7. Functions

If you are unfamiliar with any of these, please quickly refresh yourself first!

A Quick Warning on Integer Overflow

Remember `int` and `long long`? Both are numerical data types. The only difference is that an `int` x could be as large as around $2 \cdot 10^9$, while a `long long` (or "ll") l could be as large as around $2 \cdot 10^{18}$.

When adding/multiplying two integers results in a number that falls outside this range, the integer **overflows**. It doesn't crash your program but rather causes logic errors, *sometimes without you noticing*!

Because of this, I **strongly recommend that you use long longs instead of int whenever possible**, unless you *really really really* need to constant-optimize your code (i.e., make it faster without changing its *time complexity*, more on that later).

Advanced C++ Features

Advanced C++ Features: Typedef, Using, #Define

- `typedef` allows you to define a new alias for a type.
  - This is useful when a typename is long.
- `using` can also be used to alias types. Its primary use in CompProg is to help you avoid typing namespaces such as `std` over and over again.
- `#define` is a preprocessing directive that acts like a "macro". Note that `#define` works in the *compilation step* of your program, not during its runtime.

Whenever possible, prefer `typedef` over `using` over `#define`.

Advanced C++ Features: Typedef, Using, #Define

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;
typedef vector<ll> vll;
#define INF(dtype) numeric_limits<dtype>::max()

int main() {
    vll my_vec(10ll, INF(ll));

    cout << my_vec[4] << endl;

    return 0;
}
```

Advanced C++ Features: Error Stream

- `cerr` allows you to print debug statements without affecting the output read
- Be careful! This may not always work. It does work on CodeForces though.

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
#define INF(dtype) numeric_limits<dtype>::max()

int main() {
    ll a, b;
    cin >> a >> b;
    cout << a + b << endl;
    cerr << "Anya likes peanuts, Anya hates carrots" << endl;
    return 0;
}
```

Advanced C++ Features: Sort

- `sort` allows you to sort a `vector` or container of items quickly.

```
// ...
typedef long long ll;
typedef vector<ll> vll;
int main() {
    ll n;
    cin >> n;
    vll a(n, 0ll);
    for(ll & v : a) cin >> v;
    sort(a.begin(), a.end(), [](ll first, ll second){return first <= second;});
    return 0;
}
```

Advanced C++ Features: Sort

```
sort(a.begin(), a.end(), [](ll first, ll second){return first <= second;});
```

- `a.begin()` and `a.end()` are iterators, more on them later!
- The third argument of `sort` is a function that returns a boolean. We call this function $g(f, s)$ a "predicate function".
- Consider two adjacent elements a_i and a_{i+1}. You can think of sort as an algorithm that tries its best to ensure that $g(a_i, a_{i+1})$ is true for all integers $i < |a| - 1$, where $|a|$ is the length of the array a.

Advanced C++ Features: Pairs

- `pairs` allow you to store two values of potentially different data types into one variable.

```
pair<string, ll> me = {"Hoshimachi Suisei", 17ll};

cout << me.first << ", eternally " << me.second << endl;
```

Advanced C++ Features: Structs

- `structs` give you even more flexibility. They allow you to store a set of **fields**, assign values to them, and even define functions on the struct (known as methods).
- If you know Object-Oriented Programming (OOP), structs are like `classes`, but are **public by default**. That is, all fields could be accessed from the outside by default.

Advanced C++ Features: Structs

```
struct Song {
    string name;
    string artist;
    int runtime;
    void print_song() {
        cout << name << " by artist " << artist << endl;
    }
};
int main() {
    // Note that we initialize fields in the order they were defined
    Song best_song = {"Stellar Stellar", "Hoshimachi Suisei", 301};
    best_song.print_song();
    best_song.name = "Lumisan";
    best_song.artist = "Mark Alegre";
    best_song.print_song();
    return 0;
}
```

Advanced C++ Features: Classes

- `class` is basically just `struct` but **private by default**.
- For the purposes of CompProg, you can treat them as mostly the same.

```
class Song {
    public:
        string name;
        string artist;
        int runtime;
        void print_song() {
            cout << name << " by artist " << artist << endl;
        }
};
```

Advanced C++ Features

- Pointers are addresses in memory. They specify *where* your data lives. These are created using the *new* keyword.

```
int* a = new int;
*a = 42;
cout << "The meaning of life is " << *a << endl;
int* b = a; // b points to the same place that a does
*b = 3142;
cout << "The meaning of life is now " << *a << endl;
```

Advanced C++ Features

- To get a pointer to a currently existing variable, use "&".

```
int exists = 42;
int* a = &exists;
cout << "The meaning of life is " << *a << endl;
int* b = a; // b points to the same place that a does
*b = 3142;
cout << "The meaning of life is now " << *a << endl;
```

Advanced C++ Features

- Finally, to access and modify properties of an object from its pointer, use `->`.

```
struct Song {
    string name;
    string artist;
    int runtime;
    void print_song() {
        cout << name << " by artist " << artist << endl;
    }
};
int main() {
    Song *best_song = new Song();
    best_song->name = "Lumisan"; best_song->artist = "Mark Alegre";
    best_song->print_song();
    return 0;
}
```

Advanced C++ Features: Others

- Other helpful things to study:
    i. Constructor functions in classes and structs
    ii. Encapsulation in C++
    iii. The different uses of `const`, marking methods as `const`
    iv. `constexpr`, runtime and compile-time expressions
    v. C++ Templates (generics)
    vi. Standard Library (STL) methods (eg., `std::accumulate`, `std::reverse`, etc...)
    vii. Stack vs Heap allocation
    viii. Iterators (`iterator.begin()`, `iterator.end()`, etc...)

Time Complexity

Time Complexity

The **time complexity** describes how quickly an algorithm scales as the input grows. it *ignores constant factors*.

This is denoted using "big-O" notation. For example, $O(1)$ means that the performance of an algorithm scales like the constant function $f(n) = 1$ (i.e., it runs just as fast even for large inputs).

Meanwhile, $O(n)$ means that an algorithm scales like a linear function $f(n) = n$, meaning that every increase in the input by 1 unit corresponds to a constant increase in the time taken.

Time Complexity

For example, adding two numbers is $O(1)$ since it takes the same amount of time to add two fixed-size integers.

```
ll a, b;
cin >> a >> b;
cout << a + b << endl;
```

Time Complexity

As another example, adding the first `n` positive integers using the method below takes $O(n)$:

```
ll n;
cin >> n;
ll ans = 0ll;
for(ll i = 1ll; i <= n; i++) {
    ans += i;
}
cout << ans << endl;
```

Time Complexity

Meanwhile, the code below takes $O(1)$:

```
ll n;
cin >> n;
cout << n * (n + 1) / 2 << endl;
```

Time Complexity

As another example, computing $\sum_{i=1}^{n} \sum_{j=1}^{m} ij$ naively takes $O(nm)$ time since you do two loops -- one outer and one inner:

```
ll n, m;
cin >> n >> m;
ll ans = 0ll;
for(ll i = 1ll; i <= n; i++) {
    for(ll j = 1ll; j <= m; j++) {
        ans += i * j;
    }
}
cout << ans << endl;
```

Time Complexity

As another example, computing $\sum_{i=1}^{n} \sum_{j=1}^{m} ij$ naively takes $O(nm)$ time since you do two loops -- one outer and one inner:

```
ll n, m;
cin >> n >> m;
ll ans = 0ll;
for(ll i = 1ll; i <= n; i++) {
    for(ll j = 1ll; j <= m; j++) {
        ans += i * j;
    }
}
cout << ans << endl;
```

- **Thinking question:** Can you improve the time complexity?

Time Complexity

Finally, `std::sort` takes $O(n \log n)$, where $\log n$ is the logarithm -- the inverse of exponentiation.

The base-2 logarithm or simply the logarithm of n, in computer science, is defined as the unique real number such that the following holds true,

$$2^{\log n} = n$$

Time Complexity

Here are some common time complexities with some examples,

Time Complexity	Name	Examples
$O(1)$	Constant	Adding Two Numbers
$O(\log n)$	Logarithmic	Binary Search
$O(n)$	Linear	Searching Through a List
$O(n \log n)$	Linearithmic	Sorting a List, FFT
$O(n^2)$	Quadratic	Naive Polynomial Multiplication
$O(n^3)$	Cubic	Floyd-Warshall Algorithm

Time Complexity

Here are some common time complexities with some examples,

Time Complexity	Name	Examples
$O(n^k)$	Polynomial	-
$O(b^n)$, constant b	Exponential	Brute Forcing through all subsets
$O(n!)$	Factorial	Brute Forcing through all permutations

Time Complexity: The 10^8 Rule

The time complexity allows you to *estimate* the number of operations your algorithm takes simply by substituting the largest possible values for your inputs.

For example, for a quadratic $O(n^2)$ algorithm, at $n = 5000$, it will perform around $n^2 \approx 25 \cdot 10^6$ operations.

As a rule of thumb, $\leq 10^8$ operations will pass in C++ under the usual time limit of around one second.

Meanwhile, $\leq 5 \cdot 10^8$ operations is a risky AC. There's a chance it works, but there's also a chance it doesn't depending on your constant factor.

Finally, $\geq 10^9$ will highly likely TLE unless your constant factor is low.

Time Complexity: The 10^8 Rule, Example

For example, a cubic $O(n^3)$ algorithm, at $n = 500$, it will perform around $n^3 \approx 125 \cdot 10^6$ operations, which will likely pass as long as the constant factor isn't too high.

However, the same cubic algorithm at $n = 5000$ will perform $n^3 \approx 125 \cdot 10^9$ operations, which will definitely TLE.

Space Complexity

Big-O notation could also be used to determine how memory usage scales as the input increases.

For example, storing n numbers in a list takes $O(n)$ memory, while storing all lattice points whose coordinates are both $\leq n$ takes $O(n^2)$ memory.

Note that the time complexity is always greater than or equal to the space complexity. That is because requiring $O(n^2)$ memory necessitates $O(n^2)$ time to initialize and allocate memory in the first place.

Modular Arithmetic

Modular Arithmetic

In simple terms, **Modular Arithmetic** is an alternative type of arithmetic where you *only retain the remainder*.

To denote that we are only retaining the remainder of a calculation after dividing by a positive integer c, we use the following notation:

$$\mod c$$

Modular arithmetic is very common in Competitive Programming, particularly in combinatorial problems. Learning how to deal with modular arithmetic is a crucial thing to have in your toolbelt.

Modular Arithmetic

For example, since the remainder upon dividing $9 + 10 = 19$ by 3 is 1,

$$9 + 10 \equiv 1 \mod 3$$

This could also be read as "9 plus 10 is 1 modulo 3", or "the answer to 9 + 10 mod 3 is 1".

Note that we use $\equiv$ instead of $=$ to denote "equivalence".

Modular Arithmetic: Implementation

To compute the remainder of a upon dividing by c, use `a % c` in C++.

```
cout << 52 % 15 << endl; // Outputs 7
```

Modular Arithmetic: Properties

The following properties hold:

$$a \mod c + b \mod c \equiv (a + b) \mod c$$
$$a \mod c - b \mod c \equiv (a - b) \mod c$$
$$a \mod c * b \mod c \equiv (a * b) \mod c$$

In other words, taking the modulo first, then adding/subtracting/multiplying is equivalent to performing the operation first then taking the modulo.

In code, this means that `(a % c + b % c) % c == (a + b) % c` is true (in most cases).

Modular Arithmetic

For example, since modulo distributes over addition,

$$
\begin{aligned}
&1+2+3+4+5 \mod 3 \equiv 1+2+0+1+2 \mod 3 \\
&\equiv 3+0+3 \mod 3 \\
&\equiv 0+0+0 \mod 3 \\
&\equiv 0 \mod 3
\end{aligned}
$$

We also have...

$$2 \mod 3 \equiv 5 \mod 3 \equiv 8 \mod 3 \equiv 11 \mod 3$$

Modular Arithmetic

In C++, there can be cases when the result of applying `%` is negative. This can happen when some of the inputs are negative.

To ensure that the result of `%` is positive, you can use the following snippet:

```
ll a, b;
cin >> a >> b;
ll amodb = ((a % b) + b) % b;
cout << amodb << endl;
```

Modular Arithmetic

A more formal discussion on Modular Arithmetic will be given at a later date, but for now, this high-level overview should suffice.

Homework

Homework

See the Reboot Page for your homework this week 😄

Now, we Reflect

Gusto ko nang matulog, it's like 11:30pm now and there's flagcem tomorrow .-.

Now, we Reflect

But... I want to teach people how to do CompProg!