CSE200A : Competitive Programming I
(Summer 2024)
Reading 1

FooBar: Competitive Programming Club of IIITD

May 2024

1 Refresher on useful data structures in C++ STL

These are the data structures whose implementation is provided to us by the programming language. We can directly use the features provided by these data structures without the need of implementing them from scratch.

1.1 Vector

Vectors are the C++ equivalent of Python's dynamic arrays.

Vectors are sequence containers representing arrays that can change in size. In a simple static array, we can not redefine its size. So it is optimal to make static arrays only where the size of the array doesn't change according to the user's needs. Also, the size should be defined before the compilation because stack memory is assigned at the time of compilation of the program. A Vector is a dynamic array that has the ability to modify its shape whenever we perform insertion or deletion in it. Just like an array, Vector elements are placed in contiguous storage so that they can be accessed using iterators.

Here is an example of how to use vectors in C++:

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

int main(){
    int n = 5;

    // an empty vector
    vector<int> v;
    // adding elements to the back of the vector
    v.push_back(3);
    v.push_back(2);
```

```
12      // vectors can be indexed like arrays
13      cout << "Elements: " << v[0] << ' ' << v[1] << '\n';
14      // the size of a vector can be accessed using the size()
           method.
15      cout << v.size() << '\n';
16
17      // this vector will be initialised as a vector with size n=5
18      // and all elements equal to (-1).
19      vector<int> v2(n, -1);
20      // vectors can be resized to another size.
21      v2.resize(4);
22      v2[1] = 2;
23
24      cout << "Before sorting:\n";
25      for(int i = 0; i < n; ++i)
26          cout << v2[i] << ' ';
27      cout << "\nAfter sorting:\n";
28      // sorting a vector
29      sort(v.begin(), v.end());
30      for(const int &i: v2)
31          cout << i << ' ';
32
33      // (the above were 2 different ways to iterate through vector
34      // elements in order. the latter can be used for other C++
35      // containers as well)
36
37      // other useful tricks:
38      // sort(v2.rbegin(), v2.rend()) sorts in reverse order.
39      // reverse(v2.begin(), v2.end()) reverses the order of
           elements.
40
41      // vectors can also have other containers inside them,
42      // including vectors themselves.
43
44      vector<vector<int>> matrix(3, vector<int>(3,0));
45      matrix[2].push_back(3);
46      matrix[1][2] = 9;
47      matrix[2][1] = 7;
48
49      cout << "\nMatrix:\n";
50      for(const auto &i: matrix){
51          for(const auto &j: i){
52              cout << j << ' ';
53          }
54          cout << '\n';
55      }
56  }
```

Refer to the documentation here. Learn about the complexity of various member functions (insert, pushback etc.) by clicking them on the above page. You can also see the examples for each of the above functions on their corresponding pages

1.2 Maps (ordered and unordered)

Maps are the C++ equivalent of Python dictionaries.

Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.

A map (ordered map) is generally implemented using a balanced binary search tree which takes $O(\log(N))$ time to do operations.

An unordered map is generally implemented using hashing with $O(1)$ best case time complexity and $O(N)$ worst case complexity for searching and inserting.

It is recommended to use maps unless unordered maps are absolutely required.

Here is an example of how to use maps:

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

int main(){
    // first argument: type of the key
    // second argument: type of the value
    map<char, int> m;
    // whenever a key is searched,
    // if it exists in the map then the value is returned.
    // otherwise, a default value is returned and inserted
    // into the key,
    // for example 0 for int.
    cout << "Default value of type: " << m['a'] << '\n';
    // this is different from python where a KeyError occurs.

    // what will be the value of m['b'] after the line below?
    m['b'] = 1; m['b'] = 2;

    // iterating through all key-value pairs
    cout << "Our map:\n";
    // size() works in maps as well, returns number of keys.
    cout << "Size: " << m.size() << '\n';
    for(const auto &p: m){
        char key = p.first;
        int val = p.second;
        cout << "Key: " << key << " Value: " << p.second << '\n';
    }
}

```

```
// unordered_map could have also been used here.
// but there are a couple differences:
// - complexities of the operations
// - order in which the key-value pairs are iterated.
//    - in a regular map, keys are in sorted order.
//    - in an ordered map, keys are in random order
//      (determined by the hash function)
```

Refer to the documentation here and here. Learn about the complexity of various member functions by clicking them on the above page. You can also see the examples for each of the above functions on their corresponding pages.

Note- Access time in map is not O(1)!!

The constant factor of ordered map is quite big, so its generally better to use unordered map which has much smaller hidden constant with a good hash function to avoid collisions which can cause hacks in contests.

Do check the following cf blog to avoid getting hacks on unordered map.
https://codeforces.com/blog/entry/62393

Multimaps also exist.

1.3 Sets and multisets

Set is a container which contains unique elements. The element itself is identified by its value. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

A set is generally implemented using a balanced binary search tree which takes $O(log(N))$ time to find an element.

A multiset is almost the same as a set, except that the same element can have multiple instances.

Here is an example on how to use both:

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

int main(){
    set<int> s;
    s.insert(4);
    s.insert(5);
    // erase() removes:
    // - the only instance of the argument from a set
    // - all of the instances of the argument from a multiset.
    s.erase(5);
    // you can also erase from iterators. in this case,
```

```
    // s.begin() points to the minimum element 4, which gets
       removed.
    // this is useful in multisets, because erasing an iterator
    // does not remove other instances of the element pointed to
    // by the iterator.
    s.erase(s.begin());

    // NOTE: in C++17 onwards, the extract() method is a much
    // cleaner way of removing single elements from multisets.

    s.insert(3); s.insert(3);
    cout << s.count(3) << '\n';
    // count() works in both sets and multisets.


    // what does erasing a non-present element do? nothing.
    s.erase(2);

    // finding elements in sets
    if(s.find(2) != s.end()){
        // 2 is in the set
    }else{
        // 2 is not in the set
    }

    // Exercise to the reader:
    // substitute "set" for "multiset" in the above code
    // and see what happens using print statements.


    // a cool trick:

    vector<int> v = {1,1,2,3,1,5,8,8,2};
    set<int> s2(v.begin(), v.end());
    // this set now contains the unique elements from the vector.

    for(const auto &i: s2)
        cout << i << ' ';
}
```

Refer to the documentation here and here.

Just like maps, unordered sets and unordered multisets also exist.

1.4 Priority Queue

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some strict weak ordering criterion.

Refer to the documentation at http://www.cplusplus.com/reference/queue/priorityqueue/

1.5 Binary searching in vectors, sets and maps

If a vector is sorted in non-decreasing order, we can use the inbuilt binary search functions `std::lower_bound` and `std::upper_bound` to find positions of elements and more things.

Suppose in a sorted vector `v`, we want to find the smallest element which is more than or equal to an arbitrary element `k`. We can return the iterator (pointer for containers) to that element with `auto it = lower_bound(v.begin(), v.end(), k)`. We can get the index of this element with `it - v.begin()` and its value with `*it`.

Similarly we can use `upper_bound` if we want strict inequality (pointer to element strictly more than `k`).

Both of these functions work in $O(\log n)$ time and prove to be very very useful.

For sets and maps however, we can use `s.lower_bound(k)` for a set `s` or `m.lower_bound(k)` for a map `m`. In the case of map, this returns a pointer to a `pair`.

1.6 More Useful Resources

TopCoder STL Tutorial
Hackerearth STL Tutorial

1.7 Practice Problems

Fox Dividing Cheese
Monk and Magical Candy Bags
Registration System
Trees in a Row

Challenge problem - Restore Graph

2 Elementary Number Theory

The author assumes that you have elementary mathematics knowledge.
Mathematics, particularly number theory, simplifies complex computations and optimizes algorithms.

2.1 Example: Summing the First n Natural Numbers

- Method 1: Using a loop (Time complexity: $O(n)$)
- Method 2: Using the formula (Time complexity: $O(1)$)

2.2 Prime Factorization

A positive integer a is called a divisor or a factor of a non-negative integer b if b is divisible by a, which means that there exists some integer k such that $b = ka$. An integer $n > 1$ is prime if its only divisors are 1 and n. Integers greater than 1 that are not prime are composite. Every positive integer has a unique prime factorization: a way of decomposing it into a product of primes, as follows:

$$n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$$

where the p_i are distinct primes and the a_i are positive integers.

```
vector<int> factor(int n) {
    vector<int> ret;
    for (int i = 2; i * i <= n; i++) {
        while (n % i == 0) {
            ret.push_back(i);
            n /= i;
        }
    }
    if (n > 1) { ret.push_back(n); }
    return ret;
}
```

This algorithm runs in $O(\sqrt{n})$ time because the for loop iterates up to $\sqrt{n}$ times. Even though there is a while loop inside the for loop, dividing n by i quickly reduces the value of n, resulting in fewer iterations of the outer loop. This efficiency helps speed up the code execution.

Example of this algorithm for $n = 252$:

i	n	ret
2	252	$\{\}$
2	126	$\{2\}$
2	63	$\{2, 2\}$
3	21	$\{2, 2, 3\}$
3	7	$\{2, 2, 3, 3\}$

At this point, the for loop terminates because i is already 3, which is greater than $\lfloor\sqrt{7}\rfloor$. In the last step, we add 7 to the list of factors ret, because it otherwise won't be added, for a final prime factorization of $\{2, 2, 3, 3, 7\}$.

2.3 Counting Divisors

Problem

CSES - Problem Link

Solution

The most straightforward solution is to do what the problem asks us to do - for each x, find the number of divisors of x in $\mathcal{O}(x)$ time.

```cpp
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    for (int q = 0; q < n; q++) {
        int x;
        int div_num = 0;
        cin >> x;
        for (int i = 1; i * i <= x; i++) {
            if (x % i == 0) { div_num += i * i == x ? 1 : 2; }
        }
        cout << div_num << '\n';
    }
}
```

This solution runs in $\mathcal{O}(n\sqrt{x})$ time.we can actually speed this up to get an $\mathcal{O}((x+n)\log x)$ solution!

First, let's discuss an important property of the prime factorization. Consider:

$$x = {p_1}^{a_1}{p_2}^{a_2}\cdots{p_k}^{a_k}$$

Then the number of divisors of x is simply $(a_1+1)\cdot(a_2+1)\cdots(a_k+1)$.

Why is this true? The exponent of p_i in any divisor of x must be in the range $[0, a_i]$ and each different exponent results in a different set of divisors, so each p_i contributes a_i+1 to the product.

x can have $\mathcal{O}(\log x)$ distinct prime factors, so if we can find the prime factorization of x efficiently, we can use it with the above property to answer queries in $\mathcal{O}(\log x)$ time instead of the previous $\mathcal{O}(\sqrt{x})$ time.(How? Sieve taught later in this section)

please read the Codeforces blog entry for another approach to problem here.

2.4 GCD & LCM

Warning

Coding lcm as $a \times b/\gcd(a, b)$ might cause integer overflow if the value of $a \times b$ is greater than the max size of the data type (e.g., the max size of `int` in C++ is $2^{31} - 1$). Be mindful of this when coding and use `long long` if necessary.

Code

```
#include <numeric>
int g = std::gcd(a, b);
int l = std::lcm(a, b);
```

2.5 Modular Arithmetic

Basic Concepts

1. Counting Modulo n:
   - In modulo 12: $0, 1, 2, \ldots, 11, 0, 1, \ldots$ (wrapping around of 12 or calculating remainder by 12)
   - In modulo 5: $0, 1, 2, 3, 4, 0, 1, \ldots$ (wrapping around of 5 or calculating remainder by 5)
2. Counting Backwards:
   - Example in modulo 5: For integers -12 to 0: $3, 4, 0, 1, 2, 3, \ldots$
3. Residue and Congruence:
   - Residue: a is the modulo-m residue of n if $n \equiv a \pmod{m}$ and $0 \leq a < m$.
   - Congruence: $a \equiv b \pmod{n}$ if $a - b$ is a multiple of n.

Practical Applications

Modular Addition

Modular addition has the following property:

$$(a + b) \mod m = ((a \mod m) + (b \mod m)) \mod m$$

Modular Subtraction

Modular subtraction follows this property:

$$(A - B) \mod C = (A \mod C - B \mod C) \mod C$$

Modular Multiplication

Modular multiplication obeys the property:

$$(a \times b) \mod m = ((a \mod m) \times (b \mod m)) \mod m$$

Modular Division

It's important to note that the above rules do not apply to division. In modular arithmetic,

$$\frac{a}{b} \mod P \text{ is not necessarily equal to } \left(\frac{a \mod P}{b \mod P}\right) \mod P$$

Applications

1. Exponentiation:
   - Problem: The last digit of 7^{1000} is 1.
   - Pattern: Powers of 7 modulo 10: $7, 9, 3, 1, 7, 9, 3, 1, \ldots$

Advanced Applications (if you are not doing the course just for credits)

1. Fermat's Little Theorem: Useful for finding large powers modulo a prime.
   Fermat's Little Theorem
2. Euler's Totient Function: Computes the count of positive integers less than n that are coprime to n.
   Euler's Totient Function
3. Euler's Totient Theorem: States that if a and m are coprime positive integers, then $a^{\phi(m)} \equiv 1 \pmod{m}$, where $\phi(m)$ is Euler's Totient Function.
   Euler's Totient Theorem
4. Chicken McNugget Theorem: Provides insights into combinatorial number theory.
   Chicken McNugget Theorem
5. CRT: Chinese Remainder Theorem. Used to solve systems of linear congruences efficiently.
   Chinese Remainder Theorem (CRT)
6. Wilson's Theorem: States that a natural number p is prime if and only if $(p-1)! \equiv -1 \pmod{p}$.
   Wilson's Theorem

2.6 Highly Useful Mathematical Algorithms

1. Binary Exponentiation: Efficiently compute $a^n \mod m$ in $O(\log n)$ time.

$$a^n = \begin{cases} 1 & \text{if } n == 0 \\ \left(a^{\frac{n}{2}}\right)^2 & \text{if } n > 0 \text{ and } n \text{ even} \\ \left(a^{\frac{n-1}{2}}\right)^2 \cdot a & \text{if } n > 0 \text{ and } n \text{ odd} \end{cases}$$

Figure 1: Recurrence relation for binary exponentiation

2. Sieve of Eratosthenes: Determine all prime numbers up to n in $O(n \log \log n)$ time. Sieve of Eratosthenes
   If you still remember the counting divisor problem and its sieve approach, think about it for a moment. Here's how we find the prime factorization of x in $O(\log x)$ time with $O(x \log x)$ preprocessing:

   For each $k \leq 10^6$, find any prime number that divides k. To find this, we can use the Sieve of Eratosthenes, which runs in $O(n \log n)$, where n is the largest number we consider. There's also a version of the sieve that runs in linear time, but we won't be needing it. We can find the prime factorization of x by repeatedly dividing it by the prime numbers we calculated earlier until $x = 1$.

```cpp
#include <iostream>

using namespace std;

const int MAX_N = 1e6;
// max_div[i] contains the largest prime that goes into i
int max_div[MAX_N + 1];

int main() {
    for (int i = 2; i <= MAX_N; i++) {
        if (max_div[i] == 0) {
            for (int j = i; j <= MAX_N; j += i) { max_div[j]
                = i; }
        }
    }

    int n;
    cin >> n;

    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        int div_num = 1;
        while (x != 1) {
            /*
             * get the largest prime that can divide x and
                see
```

```
                * how many times it goes into x (stored in count
                    )
                */
               int prime = max_div[x];
               int count = 0;
               while (x % prime == 0) {
                   count++;
                   x /= prime;
               }
               div_num *= count + 1;
           }
           cout << div_num << '\n';
       }
}
```

3. Extended Euclidean Algorithm: Find coefficients x and y such that $ax+by = \gcd(a, b)$. Extended Euclidean Algorithm

2.7 Few common Facts

Never write a code unless you prove it to be true just on the basis of few cases There are many conjectures involving primes. While widely believed to be true, none have been proven yet. Some famous examples include:

- Goldbach's Conjecture: Every even integer $n > 2$ can be represented as the sum $n = a + b$, where both a and b are primes.
- Twin Prime Conjecture: There are infinitely many pairs of primes of the form $\{p, p+2\}$, where both p and $p + 2$ are primes.
- Legendre's Conjecture: There is always a prime number between n^2 and $(n+1)^2$, where n is any positive integer.

Zeckendorf's Theorem states that every positive integer has a unique representation as a sum of Fibonacci numbers such that no two numbers are equal or consecutive Fibonacci numbers. For example, 74 can be represented as the sum $55 + 13 + 5 + 1$.

Pythagorean Triples: A Pythagorean triple is a triple (a, b, c) that satisfies the Pythagorean theorem $a^2 + b^2 = c^2$. For example, $(3, 4, 5)$ is a Pythagorean triple. A triple is primitive if a, b, and c are coprime. Euclid's formula produces all primitive Pythagorean triples:

$$(n^2 - m^2, 2nm, n^2 + m^2)$$

where $0 < m < n$, n and m are coprime, and at least one of n or m is even.

2.8 Other Useful Results

- Check if kth bit is set: $(n >> (k - 1))\&1$

- Check if power of 2 or not: $n\&(n-1)$
- Least Significant Bit: lambda x: x & (-x)

2.9 Practice Problems

- Sherlock and His Girlfriend: Problem Link
- Almost Prime: Problem Link
- Exponentiation 2: Problem Link
- Divisor Analysis: Problem Link
- T-Primes: Problem Link
- Remainders Game: Problem Link
- Diophantine Equations: Problem Link
- Totient: Problem Link
- Power (x, n): Problem Link
- Alice, Bob, Oranges, and Apples (Hard): Problem Link
- Reducing Fractions (Hard): Problem Link