CSE200A: Competitive Programming I
(Summer 2023)
Reading 1

FooBar: Competitive Programming Club of IIITD

June 2023

1 Number Theory

1.1 Modular Arithmetic

For convenience, I will provide a definition of the "n mod m" notation for integer values of n and m. In this context, "n mod m" is determined by subtracting n/m * m from n, where x represents the largest integer that is less than or equal to x. This definition guarantees that the modulo operation always produces an integer ranging from 0 to m-1, inclusive.

It is important to note that this definition may or may not align with the expression n commonly referred to as the modulo operator, although it may be more accurate to call it the remainder operator in certain cases. If the result of n adjustments are necessary by adding m to any negative values. For instance, if -8 adding m to the negative result is required.

To achieve the desired behavior in programming languages like Java or Kotlin, you can use the Math.floorMod(n, m) function provided in the standard library. Furthermore, I define the "mod" operator to have lower precedence compared to addition or subtraction in order to simplify expression evaluation. Therefore, an expression like "ax + b mod m" is interpreted as "(ax + b) mod m". It is worth noting that this precedence rule may not align with the precedence of the "

The value m that follows the modulo operator is commonly known as the modulus, while the result of the expression n mod m is referred to as the residue of n modulo m.

1.2 Modular Addition

Below are some interesting properties of Modular Addition:
(a + b) mod m = ((a mod m) + (b mod m)) mod m

1.3 Modular Subtraction

Below are some interesting properties of Modular Subtraction :
(A - B) mod C = (A mod C - B mod C) mod C

1.4 Modular Multiplication

Below are some interesting properties of Modular Multiplication:
(a x b) mod m = ((a mod m) x (b mod m)) mod m

1.5 Modular Division

However, note that the above rule does NOT apply for division. That is, $(a/b)\ mod\ P$ is not necessarily equal to $((a\ mod\ P)/(b\ mod\ P))\ mod\ P$
Resource - Modular Division

1.6 Binary Exponentiation

Given integers a, n, and m, calculate a^n mod m in $O(log(n))$ time.

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

Resource - Modular Exponentiation

1.7 Sieve of Eratosthenes

An $O(n * log(log(n)))$ time algorithm, that determines all prime numbers from 1 to n. Useful when you need to print all prime numbers in a range, or need to check primality of several numbers.
Resource - Sieve of Eratosthenes

1.8 Extended Euclidean Algorithm

Given integers a and b, find integer coefficients such that $ax + by = gcd(a, b)$.
Resource - Euclidean Algorithm (Basic and Extended)

1.9 Practice Problems

T-Primes (Primes)
Remainders Game (Modular Arithmetic)
Diophantine Equations
Totient
Count Primes (Sieve of Eratosthenes)
Power(x,n)
Alice, Bob, Oranges and Apples (Hard)
Reducing Fractions (Hard)

2 Data Structures

2.1 Arrays

Collection of similar data types.

Works with static memory.

This is means size of array can't be changed at run time.

Elements are independent of each other.

Element access time is less as compared to linked list.

2.2 Linked List

Collection of objects known as a node where node consists of two parts, i.e., data and address.

Works with dynamic memory. This means memory can be changed at run time.

Elements are dependent on each other.

Element access time is high as compared to array.

Refer to this for differences between linked list and array.

2.3 Stacks

It is a linear data structure that follows LIFO(last in first out) principle.

It has only one end.

It has only one top pointer pointing to the topmost element of the stack.

Elements can be deleted or added from the top of the stack only.

Time complexities of basic operations:

- push: Insert an element in the stack. Time complexity = $O(1)$
- pop :Delete an element from the stack. Time complexity = $O(1)$'
- isEmpty : Check if stack is empty or not. Time complexity = $O(1)$
- isFull : Check if stack is full or not. Time complexity = $O(1)$
- peek : Returns the element at top of the stack. Time complexity = $O(1)$

Access time of any element = O(n).

Stacks can be implemented by either array or linked list.

For more information you can refer to here.

2.4 Queues

It is an ordered list that follow FIFO(first in first out) principle.

It has two ends: FRONT and REAR.

It has two pointers pointing on both the ends.

Time complexities of basic operations:

- Enqueue: add element in the queue Time complexity = $O(1)$
- Dequeue: remove element of the queue Time complexity = $O(1)$
- isEmpty : Check if stack is empty or not. Time complexity = $O(1)$
- isFull : Check if stack is full or not. Time complexity = $O(1)$
- peek : Returns the element at top of the stack. Time complexity = $O(1)$

Access time of any element = O(n).

Queues can be implemented by either array or linked list.

For more information you can refer to here.

2.5 Binary Search Tree

A binary search tree is the one in which all the nodes follow the following properties:

- The left sub-tree of a node has a key less than or equal to it's parent node's key.
- The right sub-tree of a node has a key greater than or equal to the it's parents node's key.

Time complexities of basic operations:

- Insertion and Deletion: $O(h)$ where "h" is height of the tree. In case of balanced bst, the time complexity of insertion becomes $O(log(n))$ where n is number of nodes.
- Searching: $O(h)$

Refer to the Section 3.4 - Binary Search Tree from the textbook "The Algorithm Design-Manual by Steven S. Skiena".

2.6 AVL Trees

An AVL Tree allows the following operations in $O(log(n))$ time.

- find(x): Check if the value x exists in the AVL tree
- insert(x): Insert the value x in the AVL tree.
- delete(x): Erase the value x from the AVL tree

The C++ **set** and Java **TreeSet** are implemented as balanced binary search trees, however, the balancing technique used is different from that used in AVL trees. They also support the 3 operations mentioned above in O(logN) time, as you will see in section 5. Consider the problem of finding the kth largest number number in an AVL tree. Can this be done in logarithmic time? How?

Implementation: Even though you almost never need to implement your own AVL Tree (or any other variant of Balanced Binary Search Tree), it is good to at least think about its implementation. How will you implement an AVL tree? Give it a sincere thought.

2.7 Heaps

A heap is a tree-based data structure in which all the nodes of the tree are in a specific order. In binary heap, if a heap is a complete binary tree with n nodes, then smallest possible height of the tree is $log(n)$. You can refer to heap concepts here.

Refer to this page and the MIT 6.006 video on Heaps to get a good understanding of them. During programming competitions you can always use the inbuilt priority queue implementation found in various programming languages.

2.8 Practice Problems

Linked Lists:
Palindrome Linked List (Easy)
Intersection of Two Linked Lists (Easy)
Odd Even Linked List (Medium)
Linked List Components (Medium)
Reorder Linked List (Medium)

Binary Search Trees:
Lowest Common Ancestor (Easy)
Increasing Order Search Tree (Easy)
Binary Tree Tilt (Easy)
Distribute Coins in Binary Tree (Medium)
Greater Sum Tree (Medium)

3 Inbuilt Data Structures

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.

3.1 C++

In C++ these data structures are provided in the **Standard Template Library (STL)**. It also provides various useful algorithms for sorting, searching, etc.

3.1.1 Vector

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.

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

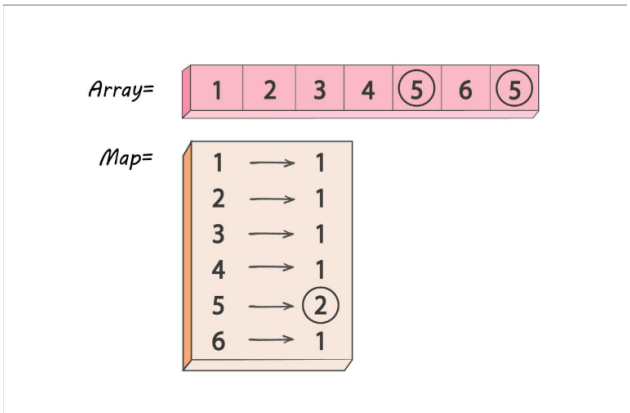
3.1.2 List

Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.

Refer to the documentation at http://www.cplusplus.com/reference/list/list/?kw=list. Learn about the complexity of various member functions (insert, pushfront, etc.) by clicking them on the above page. You can also see the examples for each of the above functions on their corresponding pages.

3.1.3 Map

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



Refer to the documentation at http://www.cplusplus.com/reference/map/map/?kw=map . 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.

3.1.4 MultiMap

Multimaps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order, and where multiple elements can have equivalent keys.

Refer to the documentation at http://www.cplusplus.com/reference/map/multimap/ . 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.

3.1.5 Set

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.

Refer to the documentation at http://www.cplusplus.com/reference/set/set/?kw=set .

3.1.6 Multiset

Multisets are containers that store elements following a specific order, and where multiple elements can have equivalent values. It is Set which can store duplicates.

Refer to the documentation at http://www.cplusplus.com/reference/set/multiset/?kw=multiset .

3.1.7 Unordered Map

Unordered maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys. Unordered Map is implemented using a hash table which takes $O(1)$ time in the average case to perform operations like insert, search, etc. However, in the worst case scenario it takes $O(N)$ time.

Refer to the documentation at https://www.cplusplus.com/reference/unordered_map/unordered_map/.

3.1.8 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/.

More Useful Resources
Topcoder STL Tutorial
Hackerearth STL Tutorial

Practice Problems
B. Fox Dividing Cheese
Monk and the Magical Candy Bags
C. Registration system
B. Trees in a Row

3.2 Java

Java Collections Framework provides these essential data structure implementations. Java also contains a special BigInteger Class to handle extremely large integers (the integers which cannot be stored in a 64 bit memory). There is no such provision in C++. Frequently used equivalents are - ArrayList, Queue, HashSet, TreeSet, HashMap, TreeMap, BigInteger.

Useful Resource - TutorialsPoint Java Collections

3.3 Python

Python provides lists, sets and dictionaries as inbuilt data types. Big numbers (extremely big numbers) are automatically handled in python without the need to use some special BigInteger Class. Frequently used data structures are Lists, Sets, Dictionaries. Python, although simpler to use, is slower than C++ and Java, and can result in TLE in some cases.

4 Binary Search

Binary Search is an algorithm used to search for a specific value in a sorted array efficiently. It operates by repeatedly dividing the search interval in half. The key idea behind Binary Search is to utilize a condition that allows us to effectively narrow down the search area.

Let's illustrate this with a simple example:

Suppose we have a sorted array of integers with a size of n, and we want to find the index of a given integer, X. The brute force approach would involve iteratively checking each element, resulting in a time complexity of O(n).

However, we can improve upon this solution using Binary Search.

Can we establish a condition that helps us divide the search area efficiently?
Yes, we can utilize the fact that the array is sorted.
Let's consider an arbitrary index, i. When comparing arr[i] with X, three different cases can arise:

1. $arr[i] < X$
2. $arr[i] > X$
3. $arr[i] = X$

Let's examine each case individually:

If $arr[i] > X$, we can infer that if X exists in the array, it must be located on the left side of index i. Similarly, if $arr[i] < X$, we can infer that if X exists in the array, it must be on the right side of index i.
This condition allows us to effectively reduce the search area, as in each case, we only consider one side of the current index.
Now, you might wonder how we determine the starting point. The answer is straightforward: we begin at the middle index. **Think of the reason behind this!!**

Now, as we have understood the algorithm, let's write the pseudo code for it:

Can you think of the time complexity of this algorithm?

```
fun (array A, int X):
        int left = 0, right = A.size - 1 // Here left and right define our current search area
        while(left<=right):  // We will continue our search until the size of search area > 0
                int mid = left + (right - left) / 2 // Try to find the reason for not using (left + right) / 2

                if A[mid] = X:
                        return mid // index found!!
                else if A[mid] < X:
                        left = mid + 1 // Case 2
                else:
                        right = mid - 1 // Case 1
```

Here are some questions for your practice:

Level - 1: Basic

1. https://www.interviewbit.com/problems/search-in-bitonic-array/
2. https://www.interviewbit.com/problems/smaller-or-equal-elements/
3. https://leetcode.com/problems/search-in-rotated-sorted-array/

Binary search on answer

Now, let's explore cases where the array is not sorted, yet Binary Search can still be applied. This may seem counterintuitive, but let's understand it using an example.

Consider an array, A, which represents the heights of N chocolate bars. The task is to find the minimum height at which a horizontal cut can be made to all the chocolates, such that the sum of the remaining amount (the lower part) is at least K.

Initially, this problem may not appear to be suited for Binary Search since the array is not sorted, and we cannot establish a condition based on its structure. However, there is no limitation preventing us from creating a condition based on some other criteria.

Let's try to identify a pattern that may be present but is not immediately apparent.
Suppose we know that if we make a cut at height H, the sum of the remaining chocolate parts is greater than K. Can we infer anything about heights greater than H based on this statement? Yes, we can.

In fact, we can conclude that if the condition is satisfied at height H, it will also be satisfied for all heights greater than H.

By leveraging this observation, we can devise a Binary Search approach to efficiently find

the minimum height for the horizontal cut. This technique allows us to locate the desired value by repeatedly dividing the search interval, even in cases where the array is not sorted.

Now, we can say that we can use Binary Search here because we have found the condition for reducing the search area. Let's assume we start from an arbitrary height, H. Two conditions can arise from this:

1. The sum of the remaining amount of chocolate is at least K. From this condition, we can deduce two things:
   (a) H is a possible answer since making a cut at height H satisfies the condition.
   (b) The answer may also exist on the left side of H, as the case might exist that for a value $h < H$, the condition is also satisfied.
2. The sum of the remaining amount of chocolate is less than K.
   From this condition, we can conclude that: for all values less than H, we won't find our answer. Therefore, our only option is to increase H and search in the higher height range.

By utilizing Binary Search with these conditions, we can efficiently locate the minimum height at which the horizontal cut should be made to meet the requirement of having a sum of remaining chocolate equal to or greater than K.

Try to think of the pseudo code for this algorithm.

Here are some more questions for your practice:

Level - 2: Easy

1. https://www.interviewbit.com/problems/square-root-of-integer/
2. https://leetcode.com/problems/search-a-2d-matrix/

Level - 3: Medium

1. https://www.interviewbit.com/problems/median-of-array/
2. https://www.interviewbit.com/problems/ median-of-array/
3. https://codeforces.com/problemset/problem/1809/B
4. https://codeforces.com/problemset/problem/1794/C

Level - 4: Hard

1. https://codeforces.com/problemset/problem/1811/E
2. https://codeforces.com/problemset/problem/1730/B
3. https://codeforces.com/problemset/problem/1731/D