CSE200A: Competitive Programming I (Summer 2019) Reading 1

Anmol Khurana, Kanika Saini, Madhav Sainanee

May 2019

1 Linked List

Refer to the **Section 3.1 - Contiguous vs. Linked Data Structures** from the textbook "The Algorithm Design Manual by Steven S. Skiena".

1.1 Problems

- Palindrome Linked List (Easy)
- Intersection of Two Linked Lists (Easy)
- Group Odd Even Nodes in Linked List (Medium)
- Linked List Components (Medium)
- Reorder Linked List (Medium)

2 Binary Search Tree

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

2.1 Problems

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

3 AVL Trees

Refer to the MIT 6.006 video on AVL trees.

3.1 Why AVL Trees ?

An AVL tree allows the following operations in O(logN) 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?

3.2 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 it's implementation. How will you implement an AVL tree? Give it a sincere thought.

4 Heaps

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.

5 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.

5.1 C++

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

5.1.1 Vector

Vectors are sequence containers representing arrays that can change in size.

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.

5.1.2 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.

5.1.3 Map

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

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.

5.1.4 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.

5.1.5 Set

Sets are containers that store unique elements following a specific order.

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

5.1.6 Multiset

Multisets are containers that store elements following a specific order, and where multiple elements can have equivalent values.

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

5.1.7 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.

Refer to the documentation at http://www.cplusplus.com/reference/unordered$_m$*ap*/*unordered*$_m$*ap*/.

5.1.8 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/priority$_q$*ueue*/.

More Useful Resources
Topcoder STL Tutorial
Hackerearth STL Tutorial

5.2 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

5.3 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.

6 Binary Search

6.1 Introduction and Monotonicity

Binary search is a commonly used yet extremely powerful technique in competitive programming.
Binary search can only be used for functions that are purely monotonic in the required range.

Let us remember what monotonicity means.
1. non-decreasing monotonicity: for all x, y such that $x > y$, $f(x) \geq f(y)$
2. non-increasing monotonicity: for all x, y such that $x > y$, $f(x) \leq f(y)$

Simply put, these are functions that don't change their direction.
For more information, check the Wiki page on monotonic functions.

6.2 Principle, Working and Time Complexity

Let us assume we have a function f ($Z -> Z$) that we know is monotonically non-decreasing between the range {A, B} (we shall look at an example soon)

Given a value **v**, our task is to find any **x** such that $f(x) = \mathbf{c}$.

An obvious solution is to check all possible values (A, A+1, A+2, ... B) and see if the function has value c at any of those. This solution finds O(B-A+1) = O(N) values of the function where N is size of the range.

This solution ignores the fact that the function is monotonic. Let us try to use the monotonicity to improve our running time.

Let $f(A) = a$ and $f(B) = b$. Consider the middle point of the range, mid = (A+B)/2.
Now, if a genie was to tell us that $f(mid) = d \geq c$, then we would have no need to check all values from {mid + 1, B} since all function values in that range will be greater than d which in turn in greater than or equal to than c. So we can simply reduce our range to {A, mid}.

On the other hand, if the genie were to tell us that $f(mid) = d < c$, then we would have no need to check all values {A, mid} since all function values in the range will be less than or equal to d which in turn is lesser than c. So we can simply reduce our range to {mid+ 1, B}.

Let us be the genie ourselves, since we can calculate f(mid) ourselves!

Every time we calculate function value, we can immediately reduce our range by **half**. Doing this repeatedly, we will reach a trivial range {L, L}. Then we can simply check L and comment on whether it is equal to c or not.
So we need only O(log2(B-A+1)) = O(log2(N)) function calls to find the value.

Let us take in how incredible that is. For N = 10^{18}
Our naïve solution requires 10^{18} function calls. Even assuming every function call can be computed in $O(1)$, it would still require approximately 10^{10} seconds or **316 years**!

In comparison, our binary search solution requires 60 function calls. That would take a **few milliseconds**!

6.3 Example

To get familiar with binary search, let's walk through a problem:

Problem Link : Problem

Solution : Let us consider the naïve solution first.
We take the prefix sums for the piles. What the means is that we will take an array p[n] such that
p[1] = a[1]
p[2] = a[1] + a[2]
.
.
.
p[i] = a[1] + a[2] + ... + a[i-1] + a[i]
.
.
.
p[n] = a[1] + a[2] + ... + a[n-1] + a[n]

We can see for all i > 1 p[i] = p[i - 1] + a[i]

So, we can calculate this array in O(N) time.

Then for every query we iterate through this array and we print the smallest i such that p[i] $\geq$ val, where val is the index of the worm to be found.

Time complexity = O(N + NM) = O(NM)
Since N = 10^5 and M = 10^5, this will clearly time out.

Let us now try binary search.

Firstly it is easy to see that the search function f(val) = p[val] is monotonically non-decreasing.
Since p[i] = p[i - 1] + a[i] and a[i] $\geq$ 1
$\implies$ p[i] > p[i - 1]

So, we can binary search to find minimum index of pile.
Pseudocode:

```
start = 1
end = n
result = n
while(s <= e):
    mid = (start + end)//2
    if(p[mid] >= val):
        result = min(result, mid)
        e = mid - 1
    else:
        s = mid + 1
print(result)
```

This runs in O(N + MlogN) time. This fits in our required TL!

6.4 Practice Problems

Problem 978C (Easy)
Problem 492B (Easy)
Problem 1010A (Medium)
Problem 760B (Medium)
Problem 348A (Hard)
Problem 1111C (Hard)

7 Number Theory

7.1 Modular Arithmetic

There are many problems which require printing the answer modulo some large prime number (for example: $10^9 + 7$). For such problems and many more, understanding Modular Arithmetic is a must.
Resource - Modular Arithmetic Properties

7.2 More

Refer to the following resources on various important topics from Number Theory:

- Codechef Basics of Number Theory
- Basic and Extended Euclidean Algorithms
- Exponential Squaring
- Sieve of Eratosthenes - Example Code

7.3 Problems

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

8 Topics in May Long Challenge

Following are some of the topics featured in this month's CodeChef Long Challenge. Feel free to look up these topics on the internet and read about them. However, Contest 1 will only be based on the topics covered till section 7.

- Implementation
- Basic Number Theory
- Game Theory
- Basic Geometry