ACM CP

Session 2:
Data Structures

ACM CP Leaderboard

- Leaderboard entries begin today.
- The scoring duration ends at the end of the CP session
- The leaderboard is updated and closed until the following week's session.
- Problems will still be available after the session but no points will be reflected.
- The sooner you solve them, the higher up in the leaderboard you will be, given your answers are right.
- The competition goes on until the end of the semester to crown the championship winner.

ACM CP Leaderboard Points Distribution

If you see this symbol next to a round/ question, then those set of questions will be counted for the leaderboard

Warmup Contest

www.hackerrank.com/acm-cp-contest-1

What are data structures?

- Data structures are ways in which we can organise information to make use of it for computation.

- Almost all code consists of some sort of algorithms operating on one or more data structures.

Basic data structures

- Static arrays – int arr[20]
- Dynamic arrays – vector<int>
- Linked lists – list<int>
- Stacks – stack<int>
- Queues - queue<int>

Applications of Stacks

- Processing events in LIFO order
- Reversing a sequence
- Processing prefix/postfix expressions

Applications of Queues

- Processing events in a FIFO order

More data structures

- Heaps
- Priority queues – priority_queue<int>
- Sets – set<int>
- Maps – map<int, int>

Applications of Priority Queues

- Processing events in order of priority
- Finding a shortest path in a graph (Dijkstra's algorithm)
- Creating a minimum spanning tree of a graph (Prim's algorithm)
- Some greedy algorithms

Applications of Sets

- Keep track of distinct items
- Check if an item has been seen before

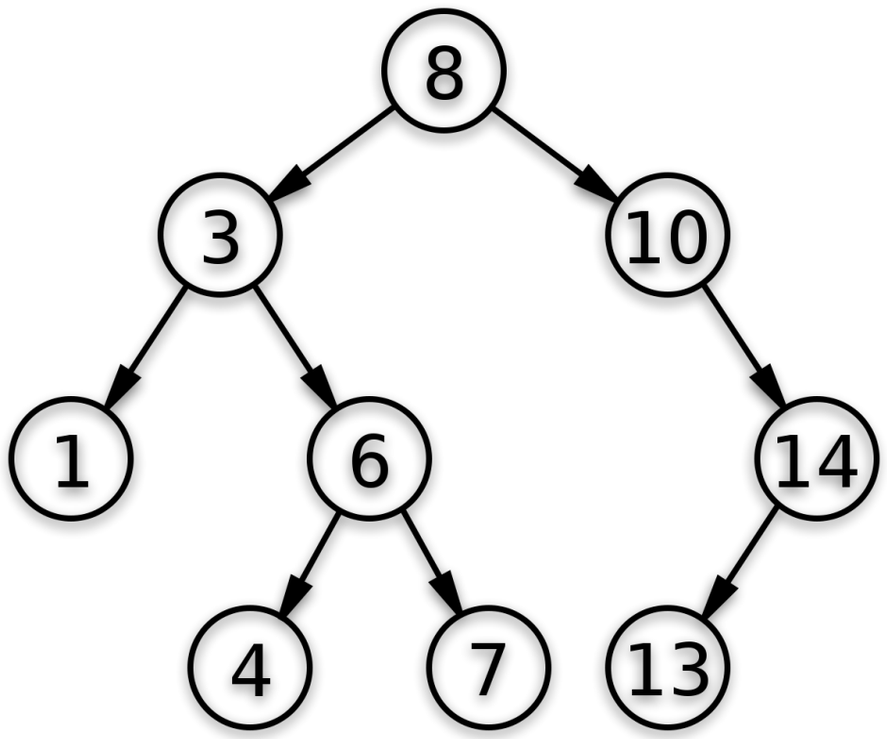
Applications of Maps

- Associating a value with a key
- As a frequency table

Binary Search Tree

A binary tree with the following property for each node *v*:

- value of v ≥ values in v's left subtree
- value of v ≤ values in v's right subtree



Sorting

- Bubble sort
- Insertion sort
- Merge sort
- Quick sort
- Counting sort
- Radix sort

Sorting

- For most purposes, standard library implementations will be sufficient for sorting. Let arr be the name of the array variable.

  - sort(arr, arr + size) in **C++**

  - arr.sort( ) in **Python**

  - Arrays.sort(arr) in **Java**

Searching

- *For unsorted arrays, use linear search*
- *For sorted arrays, use binary search*
- Standard library functions will work in most cases
  - Linear search – find(arr, arr + size, x)
  - Binary search – lower_bound(arr, arr + size, x)

    upper_bound(arr, arr + size, x)

Trunks is a third year student at elementary school. He is now learning the addition operation.

The teacher has written down the sum of multiple numbers. Pupils should calculate the sum. To make the calculation easier, the sum only contains numbers 1, 2 and 3. Still, that isn't enough for Trunks. He is only beginning to count, so he can calculate a sum only if the summands follow in non-decreasing order. For example, he can't calculate sum 1+3+2+1 but he can calculate sums 1+1+2 and 3+3.

You've got the sum that was written on the board. Rearrange the summands and print the sum in such a way that Trunks can calculate the sum.

Input

The first line contains a non-empty string *s* — the sum Trunks needs to count. String *s* contains no spaces. It only contains digits and characters "+". Besides, string *s* is a correct sum of numbers 1, 2 and 3. String *s* is at most 100 characters long.

Output

Print the new sum that Trunks can count.

Sample Input 1

3+2+1

Sample Output 1

1+2+3

Sample Input 2

1+1+3+1+3

Sample Output 2

1+1+1+3+3

Hint: Counting sort

Representing sets

- Small (n ≤ 30) number of items
- Label them using integers
- **Represent these sets as a 32-bit integer**

- [i] item is in the set if the [i] bit of the integer is 1
- Example:
  - We have the set {0,3,5}
  - `int x = (1<<0) | (1<<3) | (1<<5);`

Representing set operations

- Empty set: 0
- Universe set: (1<<n) – 1
- Union: x | y
- Intersection: x & y
- Complement: ~x && ((1<<n) - 1)

Sherlock and Array

Watson gives Sherlock an array of length *N*. Then he asks him to determine if there exists an element in the array such that the sum of the elements on its left is equal to the sum of the elements on its right. If there are no elements to the left/right, then the sum is considered to be zero.
Formally, find an *i*, such that, $A_0 + A_1 + \ldots + A_{i-1} = A_{i+1} + A_{i+2} + \ldots + A_{N-1}$.

Input Format
The first line contains *T*, the number of test cases. For each test case, the first line contains *N*, the number of elements in the array. The second line for each test case contains *N* space-separated integers, denoting the array *A*.

Constraints
$1 \leq T \leq 10$
$1 \leq N \leq 10^5$
$1 \leq A_i \leq 2*10^4$
$1 \leq i \leq N$

Output Format
For each test case print `YES` if there exists an element in the array, such that the sum of the elements on its left is equal to the sum of the elements on its right; otherwise print `NO`.

Sherlock and Array

Sample Input

```
2
3
1 2 3
4
1 2 3 3
```

Sample Output

```
NO
YES
```

Hint: Compute the sum only once and update it by adding or subtracting values for each *i*

Queries About Less or Equal Elements

You are given two arrays of integers a and b. For each element of the second array b_j you should find the number of elements in array a that are less than or equal to the value b_j.

Input
The first line contains two integers n, m ($1 \le n, m \le 2 \cdot 10^5$) — the sizes of arrays a and b.
The second line contains n integers — the elements of array a ($-10^9 \le a_i \le 10^9$).
The third line contains m integers — the elements of array b ($-10^9 \le b_j \le 10^9$).

Output
Print m integers, separated by spaces: the j-th of which is equal to the number of such elements in array a that are less than or equal to the value b_j.

Queries About Less or Equal Elements

Sample Input

```
5 5
1 2 1 2 5
3 1 4 1 5
```

Sample Output

```
4 2 4 2 5
```

Hint: Binary search

Hardwood Species

http://poj.org/problem?id=2418

Hint: Use a `map` to store frequency of each tree

End of Session

Session 2:
Data Structures