Code Club – JU

- Introduction
- Big O Notation
- Some Elementary Data structures
- Standard Template Library: C++

An example Problem

FCTRL2 - Small factorials

no tags

You are asked to calculate factorials of some small positive integers.

Input

An integer t, 1<=t<=100, denoting the number of testcases, followed by t lines, each containing a single integer n, 1<=n<=100.

Output

For each integer n given at input, display a line with the value of n!

Example

Sample input:

```
4
1
2
5
3
```

Sample output:

```
1
2
120
6
```

Date:	2004-05-28
Time limit:	1s
Source limit:	2000B
Memory limit:	1536MB
Cluster:	Cube (Intel Pentium G860 3GHz)
Languages:	All except: SCM chicken

Components

- The Problem Statement
- The input specifications
- A few I/O examples for test cases
- Time and memory limits

Solving Strategy

- Read and understand the problem
- Understand required algorithmic bounds (time,space)
- Identify Edge Cases
- Code and test.
- Debug and finally submit the Solution to the Judge

Now what is the judge?

- Cluster of computers that run the submitted solution and gives feedback
- The program is run with various input files and checked against output
- The memory usage, time limits are recorded
- After running the program, the judge gives it’s verdict

The Judge gives it’s verdict

You can get various messages from the judge.

Message	Meaning
AC	*accepted* - your program ran successfully and gave a correct answer. Congratulations, you have solved the problem!
WA	*wrong answer* - your program ran successfully, but gave an incorrect answer. Most probably your program contains a bug, or you are not interpreting the problem text correctly.
TLE	*time limit exceeded* - your program was compiled successfully, but it didn't stop before time limit. This may be because the algorithm is badly designed (too slow), or because it contains a bug, e.g. it goes into an infinite loop, or hangs up, expecting some input data.
CE	*compilation error* - your program couldn't be compiled; the details of the compiler error can be accessed from the www interface.
5. RE -	*runtime error* - your program was compiled successfully, but it exited with a runtime error or crashed. You will receive an additional error message, for example SIGSEV, SIGFPE,SIGABRT,NZEC, etc.

A few common websites

www.hackerrank.com

www.codechef.com

www.topcoder.com

www.spoj.com

Choice of Language

- Most Competitions/Coding exams for placements have a time limit.
- Not enough time to code all functions.
  - Example: Sort, find,
- Not enough time to code all data structures.
  - Example: Queues, stacks, lists, hash-maps, priority queues, etc
- Best choice is to use some language which provides these functions
- C, C++, Java, Python, etc. are some choices.

Time for a pros and cons list:

	C	C++	Java	Python
Speed	Fast	Fast	Slower than C,C++	Slow
Data Structures	Very Limited	Stack, queue, list, map, priority queue and a few others	Extensive. Much more than C++	List, dictionary, tuple can be manipulated easily for elementary data structures
I/O	Very Fast	Fast	Slow	Slow
Support for Large integers	None above Long Long	None above Long Long	BigInteger class deals with large integers	Integers can be arbitrarily large
Functions	Qsort	<algorithms> has lots of inbuilt algorithms required for most operations	Extensive	Almost everything is done

C++: A balanced choice

	C	C++	Java	Python
Speed	Fast	**Fast**	Slower than C,C++	Slow
Data Structures	Very Limited	**Stack, queue, list, map, priority queue and a some others**	Extensive. Much more than C++	List, dictionary, tuple can be manipulated easily for elementary data structures
I/O	Very Fast	**Fast**	Slow	Slow
Support for Large integers	None above Long Long	**None above Long Long**	BigInteger class deals with large integers	Integers can be arbitrarily large
Functions	Qsort	**<algorithms> has lots of inbuilt algorithms required for most operations**	Extensive	Almost everything is done

What is big O?

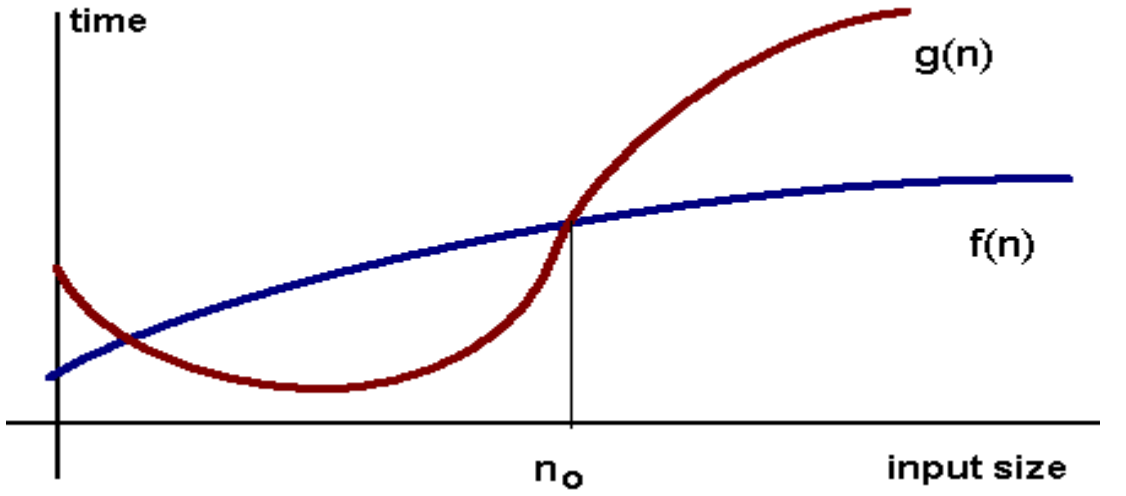
- Measures Complexity in terms of input size.
- Emphasis on worst case scenario.
- Can be measured in terms of memory requirements or time.
- Big O specifically describes the **worst-case** scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

A bit of Mathematics

- Mathematical definiton for the big O:

- we say that f(n) = O(g(n)) when there exist constants c > 0 and n0 > 0 such that

$$f(n) \leq c * g(n), \text{ for all } n \geq n0$$



A simple Question

What does it mean when we say that an algorithm X is asymptotically more efficient than Y?

A. X will be a better choice for all inputs

B. X will be a better choice for all inputs except small inputs

C. X will be a better choice for all inputs except large inputs

D. Y will be a better choice for small inputs

O(1)

- O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set

```
bool IsFirstElementNull(String[] strings)
{
        if(strings[0] == null)
        {
                return true;
        }
        return false;
}
```

Some More Examples

- array: accessing any element
- stack: push and pop methods
- queue: enqueue and dequeue methods

O(N)

- O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

```
bool ContainsValue(String[] strings, String value)
{
        for(int i = 0; i < strings.Length; i++)
        {
                if(strings[i] == value)
                {
                        return true;
                }
        }
        return false;
}
```

Other Examples

Operations on array like :

- Searching
- Traversing (Because why not?)
- Finding max/min

Lots of operations on Linked lists:

- Reversal
- Cycle Detection
- Merging 2 sorted linked-lists

$O(N^2)$

- $O(N^2)$ represents an algorithm whose performance is directly proportional to the square of the size of the input data set.
- This is common with algorithms that involve nested iterations over the data set.
- Deeper nested iterations will result in $O(N^3)$, $O(N^4)$ etc.

An Example:- Bubble Sort for array with n Elements

```
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n; i++)
       for (j = 0; j < n-i-1; j++) //Last i elements are already in place
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
```

Some Other Examples

- Insertion Sort
- Selection sort
- Operations on square matrices like addition, scalar multiplication

Time for another question.

```
int fun(int n)
{
  int count = 0;
  for (int i = n; i > 0; i /= 2)
     for (int j = 0; j < i; j++)
        count += 1;
  return count;
}
```

O(log (n))

```c
#include <stdio.h>
#include <stdlib.h>

/* Assuming arr to be already sorted */
int binary_search(int arr[], int len, int key)
{
  int low = 0, high = len - 1, mid;
/* In every iteration, the search space is reduced to half its previous size  */
  while(low <= high)
    {
      mid = (low + (high - low)/2;
      /* Notice how we are not doing mid = (high + low)/2;
         This is because high + low may exceed the range of
         an int and give erroneous results. */
      if(arr[mid] == key)       //key found at index mid
        return mid;
      else if(key < arr[mid])   //key is in the half with elements <= arr[mid]
        high = mid - 1;
      else
        low = mid + 1;          //key is in the half with elements >= arr[mid]
    }
  return -1;
}
/* Notice that we find only one occurrence of key here */
```

Input size and Complexity

- Let t be the number of test cases, n the maximum input size
- Let N=t*n
- Usually there are ~10^8 computations allowed per second

N	Probable order
~20-25	2^N, N!
10^4	N^2
10^6	N*log(N)
$10^{\wedge 8}$	N
Higher	log(N) or O(t)

- There are certain problems where the complexity seems really high, but due to some reason, there is a constant time solution. For such cases, O(t) time is needed

STL and some Elementary Data Structures

- Stack
- Queue
- List/Vector
- Heap/Priority Queue
- Map
- Set

Stack

- Last in, first out (LIFO)
- 2 Supports three constant-time operations
  - Push(x): inserts x into the stack
  - Pop(): removes the newest item
  - Top(): returns the newest item

Queue

- Last in, First out (LIFO)
- 2 Supports three constant-time operations
  - push(x): inserts x into the stack
  - pop(): removes the newest item
  - top(): returns the newest item
- Constructor:
  - priority_queue< T, vector <T>, greater/lesser<T> > q;
  - T is a builtin data type.

Priority Queue

- Each element in a PQ has a priority value
- 2 Three operations:
  - Insert(x, p): inserts x into the PQ, whose priority is p
  - RemoveTop(): removes the element with the highest priority
  - Top(): returns the element with the highest priority
- 3 All operations can be done quickly if implemented using a heap

List/Vector

- Vector is an array that resizes itself. Similar to ArrayList in Java.
- Accessing elements is O(1) , same syntax as array.
- Amortized Insertion complexity is O(N) where N is the size of the vector
- Maximum size of a vector is around 10^8
- All other Data Structures in STL discussed upto now is internally represented using Vectors

Priority Queue using Heap

- Complete binary tree with the heap property:
- value of a node values of its children
- 2 The root node has the maximum value
- 3 Constant-time: top()
- 4 Inserting/removing a node can be done in O(log n) time
- without breaking the heap property
- 5 May need rearrangement of some nodes

Map

- Insertion/Deletion/Access takes O(log n) time.
- Declaration:
  - map<string, int> count;
  - This declares a new map that stores integers addressed by string keys
  - Any pair of data types may be used as long as <= operator is defined for key datatype
  - Data ordered on key values.
- Insertion:
  - count[“Jadavpur”]=10
- Access:
  - count[“Jadavpur”]++
- Search:
  - count.find(str)

    If not found, count.end() is returned

Set

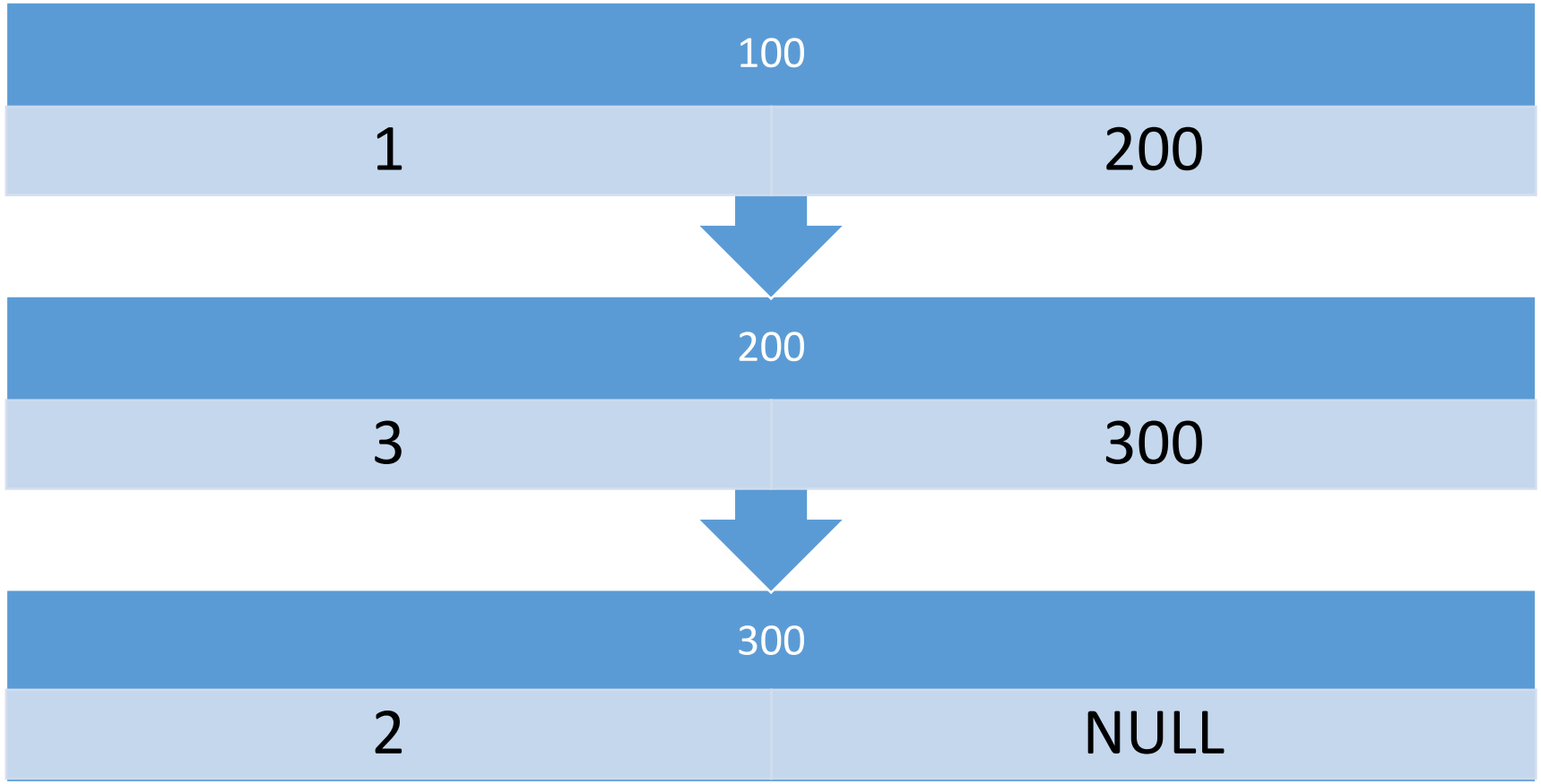
- Set stores a sorted sequence of unique data.
- Declaration:
  - set<int> s;
- Insertion:
  - set.insert(<pointer/iterator to start of list>,<pointer/iterator to end>)
- STL provides the following functions
  - set_union,set_intersection,set_difference, set_symmetric_difference
  - Parameters for them are (Pstart S1,Pend S1, Pstart S2, Pend S2,ResultP start)
  - Returns iterator to last element

Iterators:

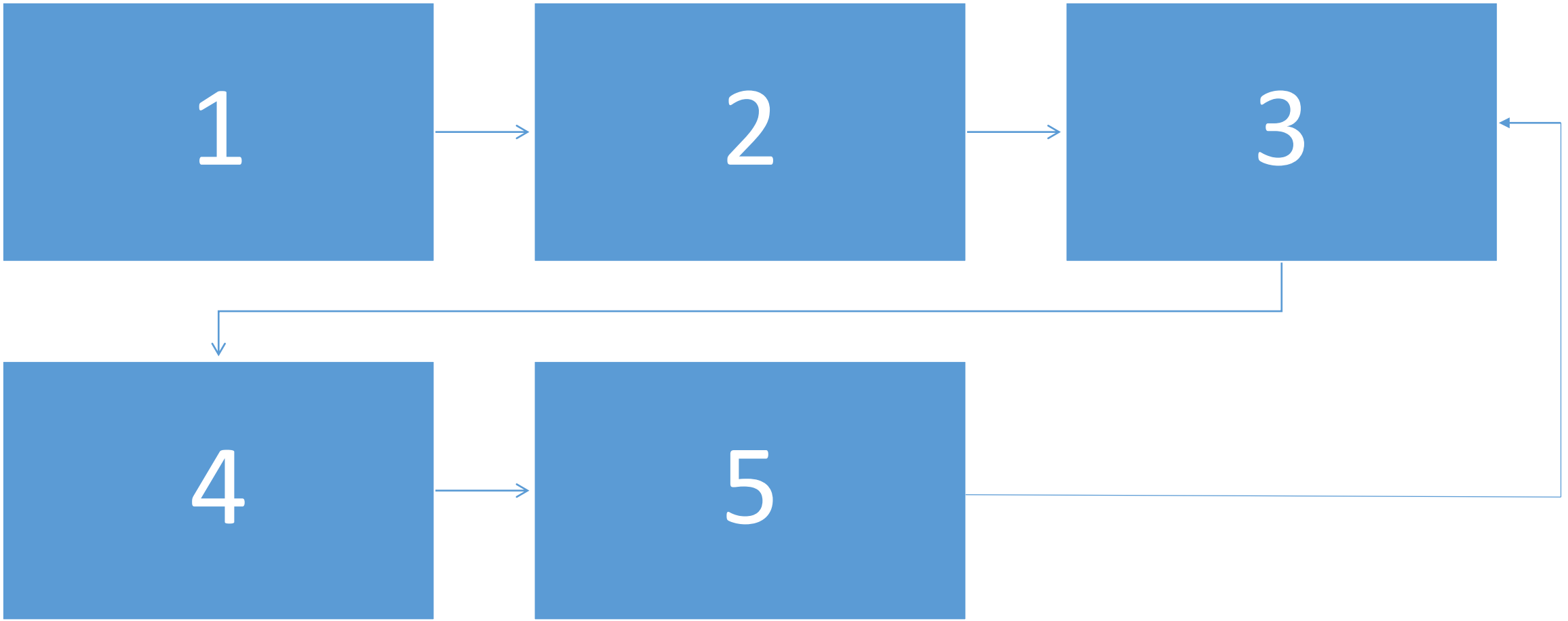
- Iterators are pointer wrappers that can be used to access elements in STL containers.
- Examples:
  - Printing elements in a map
    for(map<string,int>::iterator it=count.begin();it!=count.end;++it)
      printf("%s %d", it->first.c_str(), it->second);
  - Printing elements T where T=set/vector
    for(T<int>::iterator it=instanceT.begin();it!= instanceT.end();++it)
      printf("%d", *it);
- Can perform any activity inside for loop

Now for a few problems

- Detecting loops in linked lists.



Example of loop



Code snippet

- struct node{
  int data;
  struct node *next;
  }

  const struct node* head=create_node(0);/*create_node creates a new node*/
  …
  … //create random list starting at head

  bool has_cycle(struct node* ptr){
  //function to write
  }

  main(){
  …//initializing and calling functions to create list
  if (has_cycle(head)) printf(“Has Cycle”);
  }

Possible Approaches

- Naïve approach is to check all pairs of nodes: O(n^2)
- Possible in O(n) using O(n) space by using
  map<struct node *, bool> visited
- Optimal algorithm uses 2 pointers, fast, slow
- Fast traverses 2 nodes at a time
- Slow traverses 1 node at a time
- If fast reaches NULL at end, no cycle
- Else fast and slow pointers end up at same position. Thus loop

Turbo Sort : spoj.com/problems/TSORT

- Given the list of numbers, you are to sort them in non decreasing order.

- Input
  - *t* – the number of numbers in list, then *t* lines follow [*t* <= 10^6]. Each line contains one integer: *N* [0 <= *N* <= 10^6]
- Output
  - Output given numbers in non decreasing order.

Possible approaches

- Recall that map is ordered by key. Hence one approach can be to store the number of occurences in a map and then traverse the map printing keys n times where count[key]=n.
- STL has certain algorithms implemented including sort.
  - sort(PTRSTART,PTREND);
  - Here, if we have the numbers stored in array arr the code to solve the problem will be:
    sort(arr,arr+t) /*recall t is the number of integers, specified in the problem*/
  - Can also be done by inserting all numbers into a priority queue and popping and printing till pq.empty()

Online Minimum

- You have a stream of inputs as described below:
  - Single Integer n: Insert the element into the list.
  - d-> delete the last element entered.
- Input: One integer t which denotes the number of test cases is followed by t lines of the form given above.
- Output: After each line, output the minimum element
- Specs: Memory: 256MB, T=1s, $1<=t,n<=10^8$

Analyzing the problem

- The complexity required seems to be O(t) or better, since t=10^8, and time limit is 1s
- Can we use an array?
- No The maximum array you can declare is 4*10^4
- What other options do we have?

Analyzing the problem

- The complexity required seems to be O(t) or better, since t=10^8, and time limit is 1s
- Can we use an array?
- No The maximum array you can declare is 4*10^4
- What other options do we have?
- Vector? Linked list? Stack?
- How to handle Queries?
- If you search for minimum every time, complexity becomes O(t^2)

Optimal Solution

- Use Stack.
- Top deletion corresponds to pop()
- What about minimum?
- Let at some time, x is pushed on the stack, x is minimum. Let after m operations, y is pushed on to the stack and y is minimum
- If after that, y is popped off the stack the next minimum will again be x.

Optimal Solution

- Use Stack.
- Top deletion corresponds to pop()
- What about minimum?
- Let at some time, x is pushed on the stack, x is minimum. Let after m operations, y is pushed on to the stack and y is minimum
- If after that, y is popped off the stack the next minimum will again be x.
- In essence, we can say that the the minimum elements also behave like a stack.
- Define 2 stacks. One for data, one for minimum elements. If popped element == min_stack.top() min_stack.pop().

Optimal Solution

- Use Stack.
- Top deletion corresponds to pop()
- What about minimum?
- Let at some time, x is pushed on the stack, x is minimum. Let after m operations, y is pushed on to the stack and y is minimum
- If after that, y is popped off the stack the next minimum will again be x.
- In essence, we can say that the the minimum elements also behave like a stack.
- Define 2 stacks. One for data, one for minimum elements. If popped element == min_stack.top() min_stack.pop().
- Bet you didn’t notice that extra the.

Few Tips

- Use bitwise operations. This saves a lot time.]
  - Examples:
    - Division by 2: x>>1.
    - Getting upper case characters: c&95
- Integer Overflow
  - In many cases when the input is large and greater than the size of int, and you have to calculate the sum MOD some number, which is int, it may so happen that the sum becomes larger than 2^31-1. In such cases it's best to use long long
- Double Comparison
  - Due to implementation of doubles, often there is an error or 10^-7 or less between 2 values that are actually same. Hence, when comparing doubles, it's best to keep a small cushion EPS = 10e-7
- Last but not least, header files. <bits/stdc++.h> is a header file that includes all possible headers you might need. In a contest, it is wise to just include this header file instead of adding them as you need them.

And Concluding, some useful resources

- Books to try:
  - Introduction to Algorithms, CLRS
  - Algorithm Design Manual, Skiena
  - Programming Challenges, The Programming Contest Training Manual, Skiena
  - Competitive Programming in C++, Miguel A. Revilla
- Websites:
  - http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=alg_index
  - http://web.stanford.edu/class/cs97si/