

.....

Competitive Programming

A guide to start

by **Shikha Bhat** on August 13 2020

You will need to show motivation.

- Languages that should be used
 - C/C++/JAVA (your choice)
 - We will focus on C++, JAVA is slow (one big advantage of JAVA is Big Integers, we will see later)
 - C++ is like a superset of C with some additional tools. So, basically if you have knowledge of C, you are ready to code in C++ as well. Otherwise go back and learn how to write codes in C/C++
 - Sometimes knowledge of PYTHON is helpful when you really need big integers.
 - Python is easier to use and understand, maybe used for beginners.

PARTICIPATE PARTICIPATE PARTICIPATE (the only mantra)

- [SPOJ](#): Its a problem Archive (recommended for all beginners)
- Let's practice at least 3-5 problems per day.
 - Start with problems having maximum submissions. Solve first few problems (may be 20). Build some confidence. Then start following some good coders (check their initial submissions). Then start solving problems topic wise
 - Never get stuck for too long in the initial period. Google out your doubts and try to sort them out or you can discuss with someone (ONLY IN THE BEGINNING).
 - Before getting into live contests like codeforces or codechef, make sure that you have solved about 50-70 problems on SPOJ.
- [CodeChef](#): Do all the three contests every month. Do participate in CodeChef LunchTime for sure.

- Even if you are unable to solve a problem do always look at the editorials and then code it and get it accepted (this is the way you will learn).
- And even if you are able to do it, do look at the codes of some good coders. See how they have implemented. Again you will learn.
- Same point apply to TopCoder and Codeforces as well.

Analyze

Each problem has constraints:

Properly analyse the constraints before you start coding.

- **Time Limit** in seconds (gives you an insight of what is the order of solution it expects) -> **order analysis**(discussed later).
- The constraints on input (very imp): Most of the time you can correctly guess the order of the solution by analysing the input constraints and time limit .
- **Memory Limit** (You need not bother unless you are using insanely large amount of memory).

Errors

Run Time Error (Most Encountered)

- Segmentation fault (accessing an illegal memory address). You declared array of smaller size than required or you are trying to access negative indices .
- Declaration of an array of HUGE HUGE(more than 10^8 ints) size -_- .
- Dividing by Zero / Taking modulo with zero :O .
- USE gdb (will learn in coming lectures)

While solving the problems on an online Judge, many runtime errors can be faced, which are not clear by the message which comes with them. Lets try to understand these errors.

To get clear about the definition of run time error:

A runtime error means that the program was compiled successfully, but it exited with a runtime error or crashed. You will receive an additional error message, which is most commonly one of the following:

1) SIGSEGV

This is the most common error, i.e., a "segmentation fault". This may be caused e.g. by an out-of-scope array index causing a buffer overflow, an incorrectly initialized pointer, etc. This signal is generated when a program tries to read or write outside the memory that is allocated for it, or to write memory that can only be read. For example, you're accessing `a[-1]` in a language which does not support negative indices for an array.

2) SIGXFSZ

"output limit exceeded". Your program has printed too much data to output.

3) SIGFPE

"floating point error". This usually occurs when you're trying to divide a number by 0, or trying to take the square root of a negative number.

4) SIGABRT

These are raised by the program itself. This happens when the judge aborts your program in the middle of execution. Due to insufficient memory, this can be raised.

5) NZEC

(non-zero exit code) - this message means that the program exited returning a value different from 0 to the shell. For languages such as C/C++, this probably means you forgot to add "return 0" at the end of the program. It could happen if your program threw an exception which was not caught. Trying to allocate too much memory during code execution may also be one of the reasons.

For interpreted languages like Python, NZEC will usually mean that your program either crashed or raised an uncaught exception. Some of the reasons being in such cases would be: the above mentioned runtime errors. Or, for instance usage of an external library which is causing some error, or not being used by the judge.

6) MLE (Memory Limit Exceeded)

This error means that your program tried to allocate memory beyond the memory limit indicated. This can occur if you declare a very large array, or if a data structure in your program becomes too large.

7) OTHER

This type of error is sometimes generated if you use too much memory. Check for arrays that are too large, or other elements that could grow to a size too large to fit in memory. It can also sometimes be generated for similar reasons to the SIGSEGV error.

Unknown signal 6 (or 7, or 8, or 11, or some other). — This happens when your program crashes. It can be because of division by zero, accessing memory outside of the array

bounds, using uninitialized variables, too deep recursion that triggers stack overflow, sorting with contradictory comparator, removing elements from an empty data structure, trying to allocate too much memory, and many other reasons. Look at your code and think about all those possibilities. Make sure that you use the same compilers and the same compiler options as we [do](#). Try different testing techniques from this [reading](#).

Some ways to avoid runtime errors:

- 1) Make sure you aren't using variables that haven't been initialized. These may be set to 0 on your computer, but aren't guaranteed to be on the judge.
- 2) Check every single occurrence of accessing an array element and see if it could possibly be out of bounds.
- 3) Make sure you aren't declaring too much memory. 64 MB is guaranteed, but having an array of size $[100000] * [100000]$ will never work.
- 4) Make sure you aren't declaring too much stack memory. Any large arrays should be declared globally, outside of any functions, as putting an array of 100000 ints inside a function probably won't work.

Compilation Error

- You need to learn how to code in your language.
- USE GNU G++ compiler or [IDEONE](#) (be careful to make codes private).

Time Limit Exceed(TLE)

- You program failed to generate all output within given time limit.
- Input Files are not randomly generated , they are made such that wrong code does not pass.
- Always think of worst cases before you start coding .Always try to avoid TLE.
- Sometimes a little optimizations are required and sometimes you really need a totally new and efficient algorithm (this you will learn with time).
- So whenever you are in doubt that your code will pass or not .Most of the time it won't pass .
- Again do proper order analysis of your solution .

It gives you insight into the order of the correct solution. Consider the following situation. Time Limit = 1 Sec

Let us Assume 1 sec approximately can perform 10^8 operations.

If you write a program that is of order $O(N^2)$ and the problem has T test cases.

Then the total order of your program becomes $O(T \cdot N^2)$.

If $T \leq 1000$ and $N \leq 1000$, then (ignoring hidden constants in asymptotic notations) your code may not be accepted in the worst case as $1000 \cdot 1000 \cdot 1000$ is 10^9 operations which mean 10 seconds.

To avoid TLE, always think of the worst test cases that are possible for the problem and analyze your code in that situation.

Length of Input (N)	Worst Accepted Algorithm
$\leq [10..11]$	$O(N!), O(N^6)$
$\leq [15..18]$	$O(2^N * N^2)$
$\leq [18..22]$	$O(2^N * N)$
≤ 100	$O(N^4)$
≤ 400	$O(N^3)$
$\leq 2K$	$O(N^2 * \log N)$
$\leq 10K$	$O(N^2)$
$\leq 1M$	$O(N * \log N)$
$\leq 100M$	$O(N), O(\log N), O(1)$

Wrong Answer

Whenever you encounter WA, write a brute force code & make sure that it is perfect. Now generate test cases using random function in C++. Run your code on these test cases and match the output. Now think of the corner cases that will help you to find the problem in your algorithm.

IntOverflow

Many times unknowingly you will exceed the maximum value that can be stored in primitive type int. e.g. Constraints : $0 < \text{num1}, \text{num2} \leq 10^9$.

So you will have to use long long int or unsigned int primitive data type for the answer in such a case.

What Are the Possible Grading Outcomes?

There are only two outcomes: “pass” or “no pass.” To pass, your program must return a correct answer on

all the test cases we prepared for you, and do so under the time and memory constraints specified in the problem statement. If your solution passes, you get the corresponding feedback "Good job!" and get a point for the problem. Your solution fails if it either crashes, returns an incorrect answer, works for too long, or uses too much memory for some test case. The feedback will contain the index of the first test case on which your solution failed and the total number of test cases in the system. The tests for the problem are numbered

from 1 to the total number of test cases for the problem, and the program is always tested on all the tests in the order from the first test to the test with the largest number.

Here are the possible outcomes:

- Good job! Hurrah! Your solution passed, and you get a point!
- Wrong answer. Your solution outputs incorrect answer for some test case. Check that you consider all the cases correctly, avoid integer overflow, output the required white spaces, output the floating point numbers with the required precision, don't output anything in addition to what you are asked to output in the output specification of the problem statement.
- Time limit exceeded. Your solution worked longer than the allowed time limit for some test case.

Check again the running time of your implementation. Test your program locally on the test of maximum size specified in the problem statement and check how long it works. Check that your program doesn't wait for some input from the user which makes it to wait forever.

- Memory limit exceeded. Your solution used more than the allowed memory limit for some test case.

Estimate the amount of memory that your program is going to use in the worst case and check that it does not exceed the memory limit. Check that your data structures fit into the memory limit. Check that you don't create large arrays or lists or vectors consisting of empty arrays or empty strings, since those in some cases still eat up memory. Test your program locally on the tests of maximum size specified in the problem statement and look at its memory consumption in the system.

- Cannot check answer. Perhaps the output format is wrong. This happens when you output something different than expected. For example, when you are required to output either “Yes” or “No”, but instead output 1 or 0. Or your program has empty output. Or your program outputs not only the correct answer, but also some additional information (please

follow the exact output format specified in the problem statement). Maybe your program doesn't output anything, because it crashes.

- Unknown signal 6 (or 7, or 8, or 11, or some other). This happens when your program crashes. It can be because of a division by zero, accessing memory outside of the array bounds, using uninitialized variables, overly deep recursion that triggers a stack overflow, sorting with a contradictory

comparator, removing elements from an empty data structure, trying to allocate too much memory, and many other reasons. Look at your code and think about all those possibilities. Make sure that you use the same compiler and the same compiler flags as we do.

- Internal error: exception... Most probably, you submitted a compiled program instead of a source code.

Comparing Doubles:

float and double data types don't have infinite precision . Beware (6/15 digit precision for them respectively). So always use a margin of (~ 0.0000001) in comparing. Example –

```
if (abs(a -10) < (0.0000001))
{
cout << "YES";
}
```

Other Useful Points :

Sometimes when you are stuck. Check running time of other accepted code and analyze whether the order of solution is required and the amount of memory that is allowed.

1. 4 MB ~ integer array of size 10^6 (assuming int takes 4 bytes) or 2-d array of size $10^3 \times 10^3$

Standard Memory limits in most of the problems are of the order of 256MB.

If you have to allocate a large array, then it is NOT a good idea to do allocation inside a function as memory is allocated and released for every test case, and memory is allocated on the function call stack (stack size is limited at many places). Thus if you have to make an array of large size, make it global.

Steps to follow during a contest

1. Meditate before contest
2. Think of algorithm to solve problem
3. Consider ALL possibilities of inputs: negative numbers, real numbers, 0, characters, special symbols, whitespace, etc
4. Write on paper
5. Think of $O()$ if it's efficient or not
6. Think if you can avoid a for loop or something or break out of it earlier
7. Make sure you are taking correct data types
8. Try it on your compiler first before submitting
9. Check if all syntaxes are correct: == or and >= etc
10. Check if everything has been initialised
11. Check if you are incrementing the while loop
12. Check if you have put the right function names
13. Check if there might be pre existing functions for the task.
14. Check if Sorted Array will make it better
15. Greedy<divide and conquer<dynamic
16. Time can be converted into space
17. Check solutions after contests.
18. Okay, well while practicing problems, if you solve it on your own or even if you don't, read the editorial solution and complexity and try to justify the mentioned complexity to yourself

Standard Template Library:

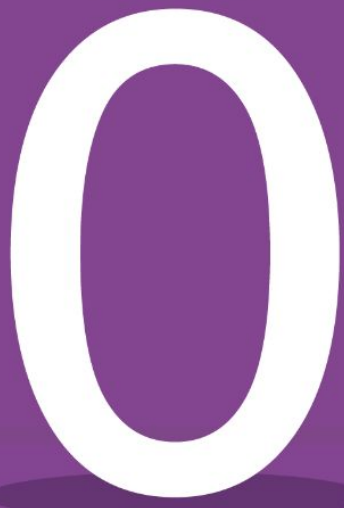
In your code sometimes you need some Data Structures(DS) and some functions which are used quite frequently. They already have lots of standard functions and data structures implemented within itself which we can use directly.

- Data Structures (To be discussed in later lectures)
 - Vectors
 - Stack
 - Queue
 - Priority Queue
 - Set
 - Map
- Functions
 - Sort
 - Reverse
 - GCD
 - Swap
 - next_permutation
 - binary_search (left+right)

- max, min
- pow, powl
- Memset

HackerEarth practice problems.

Level



Levels of our Journey

A walk in the Park

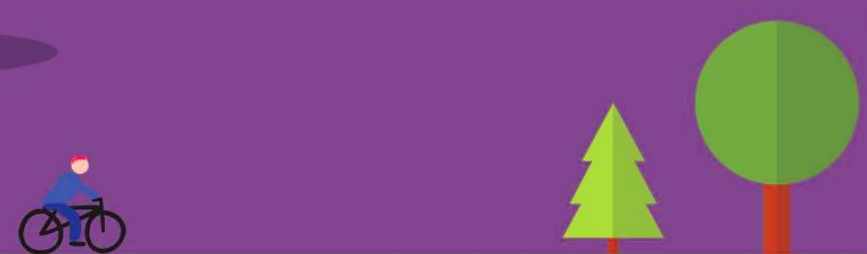
Alright, let's conquer the first 20% of programming problems out there.

You need to know:

- Intermediate hold on any one programming language
- English! Convert english to code!

Let's take an example problem of this level: [Terrible Chandu](#)

All you have to do is, read input line from STDIN and print reverse of that line to STDOUT. Go ahead, make a submission. Seek your first AC. Want more? We've got loads in our [practice](#) section. Look for the ones with thousands of correct submissions.



Welcome to the Jungle

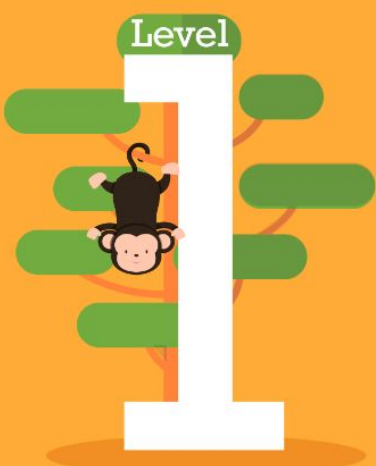
Okay, now you're ready to take on some real challenge. Hold tight, we are diving deeper.

You need to know:

- [Sort and Search algorithms](#)
- [Hashing](#)
- [Number Theory](#)
- [Greedy Technique](#)

More importantly, you have to figure out what, when and where to apply them. It gets really tricky and hence to help beginners gain a feeling of confidence we run a series of contests as [Code Monk](#). Before each contest, we release a tutorial on certain topic and later in the contest the problems are aimed only on that particular topic. I'd recommend you to go through the tutorials and solve a question or two on each topic.

Level



Level

2



Take the Fast Track

By now you've realised that the questions are framed to deceive the way we think. Sometimes, If you convert plain english to code, you'd end up with TLE (Time Limit Exceeded) verdict. You need to learn a set of new techniques and algorithms to cope up with the time limits. In certain cases, [Dynamic Programming \(DP\)](#) comes to the rescue. Infact, you might have already intuitively used this technique. There's always at least one question in any contest that can be solved by DP.

Also, you'd have noticed that there are questions that just can't solved by linear array data-structures.

- [Graph Theory](#)
- [Disjoint Set Union \(Union-find\)](#)
- [Minimum Spanning Tree](#)

These set of data-structures will get you quite far enough. Moreover, you'd have figured that the real art is to modify the techniques you know in order to solve a question. All Easy-Medium and Medium level questions can be tackled in this fashion.

Load up the Weapons

You are all set to top the leaderboards of Short Programming Challenges, just keep steady persistence. As I've already mentioned, it's a sport, you won't master it until you actually do it. Go ahead, participate in a short contest, know your strengths, weaknesses and see how you handle the adrenaline mode when the clock is ticking.

Stick to your own logic as long as possible, you'll eventually come up with something similar to the algorithm required to solve the question. You just need to brush it up. Several of these techniques will help you solve some of the toughest of the problems around.

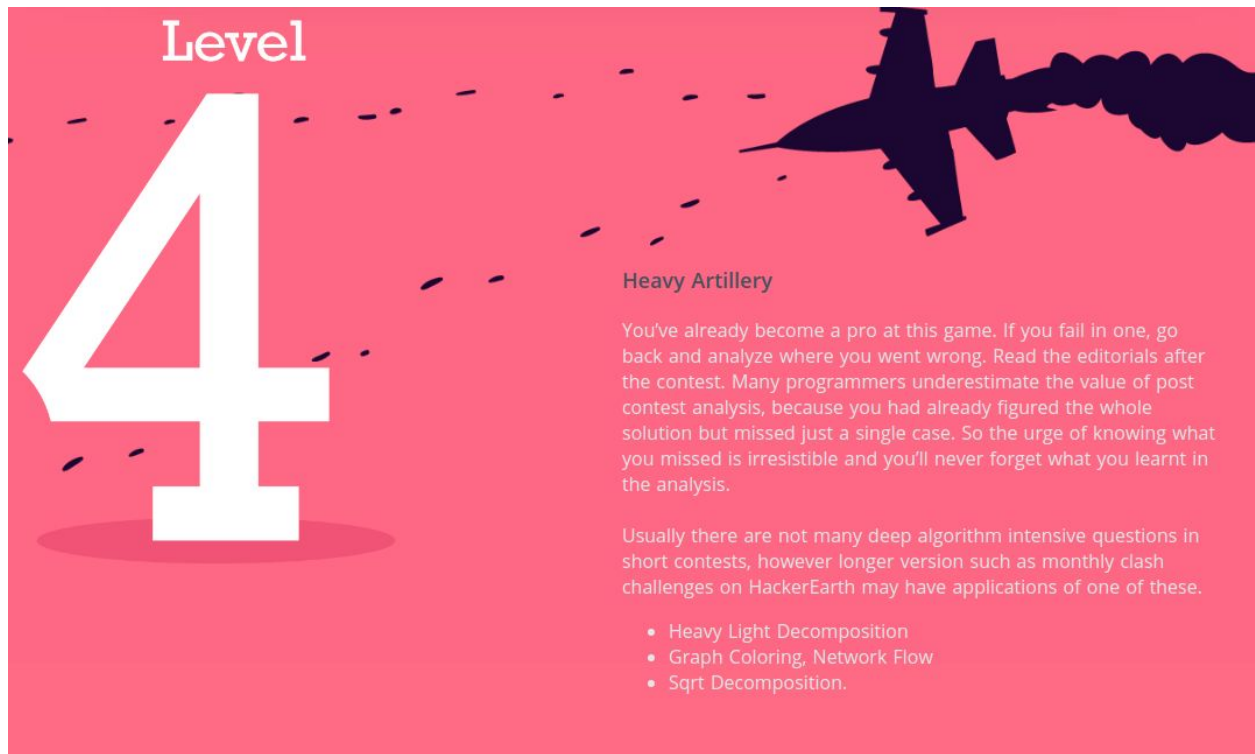
- [Segment Tree](#)
- [String Algorithms](#)
- [Tries, Suffix Tree, Suffix Array.](#)



Level

3





Interpreting Problem Statements

Introduction

The problem statement usually begins by motivating the problem. It gives a situation or context for the problem, before diving into the gory details. This is usually irrelevant to solving the problem, so ignore it if necessary.

The Definition

This is a very barebones description of what topcoder wants you to submit.

Notes and Constraints

Notes don't always appear. If they do, READ THEM! Typically they will highlight issues that may have come up during testing, or they may provide background information that you may not have known beforehand. The constraints section gives a list of constraints on the input variables. These include constraints on sizes of strings and arrays, or allowed characters, or values of numbers. These will be checked automatically, so there is no need to worry about writing code to check for these cases.

Be careful of the constraints. Sometimes they may rule out certain algorithms, or make it possible for simpler but less efficient algorithms to run in time. There can be a very big difference between an input of 50 numbers and an input of 5, both in terms of solutions that will end up passing, and in terms of ease of coding.

Examples

These are a list of sample test cases to test your program against. It gives the inputs (in the correct order) and then the expected return value, and sometimes an annotation below, to explain the case further if necessary.

It goes without saying that you should test your code against all of the examples, at the very least. There may be tricky cases, large cases, or corner cases that you have not considered when writing the solution; fixing issues before you submit is infinitely preferable to having your solution challenged or having it fail during system testing.

The examples are not always comprehensive! Be aware of this. For some problems, passing the examples is almost the same as passing every test case.

For others, however, they may intentionally (or not) leave out some test case that you should be aware of. If you are not completely sure that your code is correct, test extensively, and try to come up with your own test cases as well. You may even be able to use them in the challenge phase.