Template and Basics

What all should you include in a template? Should you even have a template in the first place? There are a lot of opinions on this, and everyone takes the one that they find the best. Although predefined templates are not allowed in coding competitions but they can speed up your problem solving a lot in competitions.

I have an extensive template with a lot of macros and predefined functions (don't use most of them), while Kunal's template is minimal with the essentials and a few macros (tourist himself does not use a template for CP).

A few things that I prefer to include are:-

Typedefs

typedef int64_t ll;

int64_t is pretty much another name for long long int. Usually when doing problems there is often an overflow from 32 bit numbers to 64 bit numbers, and if you use long long you basically avoid any problems from that. Most problems get accepted if you use ll everywhere even in the places that would work with int, and so I pretty much use ll by default unless I really need int for something or if time constraints are too tight.

typedef vector<ll> vll;
typedef vector<int> vi;

The basic data structure of a list of numbers, this is used in almost every CP question. Creates a vector of int64_t (or ints, depending on what you are using). Search on the constructors on vector to know how to create a vector of a specific length and fill it with a specific value.

Of course, you can add other structures here, such as 2 dimensional vectors, 3 dimensional vectors, vectors of pairs, vector of vectors of pairs and so on, but that is up to you.

I would suggest not to use arrays because vectors have a whole depth of features that we can cover later. The only place where arrays are useful is that they do give you a small time advantage. This is not often useful but if time limits are very tight, then this can be of real help.

We will cover sets and maps in detail later, but I do have a couple of macros for them as well.

Function Macros

I also use a couple of function shorthands for commonly used functions.

```
#define pb push_back
#define pob pop_back
#define ff first
#define ss second
#define sz size
```

And so on... Obviously, use the macros that you feel are more intuitive since the point of a template is to increase speed. If you feel that you are more comfortable with the original functions you don't need macros at all.

You can have macros for loops as well (present in my template to some capacity), I only use them while taking in input though, as while performing computations I prefer to take time to think through what I am doing.

Also there are a couple of numbers that you will be seeing a lot in CP problems, 998244353 and 10^9+7 (1000000007). A lot of problems will tell you to output the answer modulo these numbers (when the answer is too big to fit into a 64 bit number).

This is because first of all these numbers are big, and there is a very low probability that you would be able to exploit some property of the problem to output the result without the full computation. Also, these are primes which stops people from using number theory properties like the Chinese Remainder Theorem to cut down on computation. These numbers also have a few other interesting properties which you can look up if interested.

Print Statements

Often times during problems you would need to debug them. Preferably you would use GDB for this, but at times you need a quick fix which is why your macro can also have print statements that can print the value of a variable or just a string to let you know if control has passed to this block of code.

```
#define reach cout<<”Reached”<<endl;
#define o1(a) cout<<a<<”\n”;
```

You can also have other macros for printing values of 2 variables, 3 variables and other statements. You would notice that the reach macro has an endl statement instead of a normal newline. This is important because in case of Segfaults you won’t get the output on the console till the output is flushed to the screen. So it might be possible that the string was in the buffer when the program segfaulted and you thought that the block was not even executed. Endl forces the buffer to be flushed and hence avoids this problem.

You can also have functions for printing vectors, sets and maps by overloading the << operator. You can go through my template for this.

Fast I/O

These are the standard few lines of code that you will find almost everywhere :-

```
ios_base::sync_with_stdio(false);
cin.tie(NULL);
cout.tie(NULL);
```

This basically increases the speed of input and output of your code.

https://stackoverflow.com/questions/48367983/why-does-the-following-snippet-speed-up-the-code

Check out this link to understand what the code actually does (and also while sometimes it might be harder to debug with these lines written). I would highly recommend adding at least this to your template.

Tips

1. Don’t replace “\n” with endl everywhere. Flushing the output takes time and can lead to TLE on many testcases. Make it a habit of always using “\n” unless endl is actually required.

2. Get an idea of the time complexity of the algorithm needed from the constraints. A computer can do around 10^8 operations in a second, 10^6 is comfortable and 10^9 is a bit on the limit but doable in some special cases. Anything more than that simply cannot be done in 1 second. So, on an array of size 10^6, one can expect an O(n) solution or an O(nlogn) solution, certainly not an O(n^2) solution.

3. Look at the constraints of the values of the numbers as well. For example, if the question wants you to make the solution work no matter what number is entered, the limit will usually be 10^9. But if the question is looking for some preprocessing on the numbers itself (such as prime factorization, etc) you can expect a smaller limit around 10^6.

4. Avoid rewriting code for which a function actually exists. This sucks a lot of your energy and you basically keep reinventing the wheel. It is a good implementation exercise for sure, but once you are certain you can do it, use the functions instead. A few of these functions (for vectors) include :-

sort, reverse, fill, unique, distance, resize, push_back, pop_back

Also, learn how to use sort with custom comparator functions, for example using the sort function to sort strings on the basis of their length instead of normal lexographical order and so on.

You don’t need to remember these functions as such, feel free to look up their usage again and again from time to time, but you should know that a function exists for doing the exact same thing you want to do.

As a small exercise, consider you are given a vector of numbers (ints, let us say), and the aim is to transform it into a sorted version without any duplicates. Eg a=[2 ,5 ,1 , 6 , 1, 3 , 7 , 3 , 4 , 6, 1] will be changed to [1, 2, 3, 4, 5, 6, 7]. Don't use any loops, just a single line comprising of one or more of the functions mentioned above.

5. Don't do premature optimizations. If your algorithm is $O(n^2)$ for a solution that was intended to be O(n), there is no need to change all vectors to arrays, and other small changes for that extra burst in speed. You need a different algorithm altogether to pass, don't waste your time on this.

6. Often you will be stuck on a Codeforces test case with no way around it other than seeing the editorial. Before you do so though, there is a way to get the offending test case by modifying your solution to output only the offending test case (codeforces gives the line number of the test that failed) instead of the answer to all questions. Then you can simply see the test case in codeforces.

7. Also, be sure to look at other people's code as well while solving questions. You get to know new and better implementation approaches this way. It is a major part of how we progressed ourselves. Be sure to glance at the editorial even if you were able to solve it yourself. If you weren't try solving it after the editorial, and then have a look at other implementations as well. You will be surprised how efficiently some people are able to write code.

8. Also, try to use your own IDE/editor while doing CP, and not online ones. You can also use GDB on your computer or inbuilt VSCode Debugger to find the exact lines that are segfaulting, or that are not working as expected.

9. Finally, don't copy a template blindly. Templates are only useful if you are comfortable in using them. Create a template that you are comfortable with. It is fine to start out with no template, only when you do a few problems you realize what parts you are repeating again and again and can be sped up.

Happy CPing!