Tips and Tricks for Competitive Programming

Competitive programming is like a sport, you will need to write code under various restrictions like memory limits, execution time etc. to solve a particular problem. These challenges often involve the understanding of programming principals and how the computer interprets numbers and calculations.
To prepare for algorithmic competitions such as the CCC or USACO, you will need to learn do a lot of practice and below are some of my resources and information I gather over the years and through this 8 question learning session. There will also be general advice on how you should approach these questions.

Introduction

Before we start, you need to know that the questions mentioned below are not the most beginner friendly, they are from the Silver Level of the USACO competition. To get started, I recommend that you learn basic algorithms through platforms such as LeetCode or HackerRank and try the Junior CCC first. Nonetheless, the principals that I will cover later most definitely applies to all questions and competitions that you may encounter.
Depending on your language and preference, some points may not directly apply to you but I will always mention alternatives that you could learn from. I will be using `C++ 11` for all examples since thats the language and version I use for competitions.

1. A Well Designed Structure/Routine

I am a firm believer in routine and structure during rapid and mind consuming competitions such as competitive programming events, thus a refined routine is something that is definitely worth the time to set up.
Depending on language, your setup might be different, but when interacting with `C++` some boiler plate code set up for competitive programming is essential in my opinion. Take a look at a macro I defined.

```cpp
#include <bits/stdc++.h>

using namespace std;

void Input(){

}

void Solve(){

}

int main(){
    Input();
    Solve();
}
```

The first line is a tip I learnt from another competitive programmer as well, instead of importing separate libraries for different functions, data structures, the `bits/stdc++.h` library has most of the important elements that you need for such competitions. Then I define the name space as well as two functions `Input()` and `Solve()` as a base. Keep in mind that only standard libraries are allowed in these competitions.

No matter what language you use, I strongly recommend that you do write your solution in a separate function rather than in `main()` for reasons that I will explain later, but briefly, it makes debugging and troubleshooting way easier.

If you are a person like me who attends quite some competitions, you should definitely take the time and configure the text editor or IDE of your choice to follow your routine. This might be defining macros or shortcuts or even boiler templates when you create a new file. Personally, I just need to type `cp` on my keyboard, and all of this above is filled in for me.

2. Data Structures and Corresponding Methods

Using data structures and it's built in methods are extremely important, since I work with `C++` I have a wide range for me to choose from. Theoretically, you can create all these data structures from scratch but why not learn the syntax for a way easier time during contests?

In the 8 questions I investigated in, I made use of `sets`, `pairs`, `vectors`, `two-dimensional lists`, `queues` and obviously `arrays`. Look at these shorthands that I defined:

```
#define ll long long
#define mp make_pair
#define pb push_back
#define pi pair <int, int>
#define vi vector<int>
#define si set<int>

#define f first
#define s second
#define vf front
#define vb back
#define lb lower_bound
#define ub upper_bound

#define bg begin
#define end end
#define sz size
#define rv reverse
#define i insert
#define r erase
```

As you can see, the second argument for `#define` is the actual method it self. The diversity and versitility of these built in functions will be your best helper during competitions.
Why not learn them and keep them in your backpack?

3. Shorthands and `#define`

Look at `2.`, do you realize what I did there? Using `#define` statements in `C++` I organized common methods to shorter 1 or 2 letter combinations. You might think that this is unnecessary, but after a while of use, it will become second nature.

I first saw this method watching a YouTube video and decided to try it out myself and after a couple of days, it's been proven to be the best thing you can do to add you your routine. Although official documentation is often allowed in competitions, it's such a waste of time and effort to search through the lengthy documents. With these shorthands, It's way more custom and intuitive for me to operate upon these data structures.

Along side with method shorthands, you can also use shorthands for loops and functions, take a look at this:

```
#define FOR(i, a, b) for (int i=a; i<b; i++)
#define F0R(i, a) for (int i=0; i<a; i++)
#define FORd(i,a,b) for (int i = (b)-1; i >= a; i--)
#define F0Rd(i,a) for (int i = (a)-1; i >= 0; i--)
```

instead of writing for loop over and over, we all know that the most common forms are incrementing by 1 from 0 up to a number or starting from another number. By doing so my code can change from looking like

```
for(int i=0; i<=10; i=i+1){

}
```

into

```
FOR(i,10){

}
```

In conclusion, this is totally personal experience and preference but I highly recommend you try something like this and add it to your routine.

4. Time Complexity & Big O Notation

While doing competitive programming questions, you need to understand time and space complexity as well as Big O notation. The correctness of your solution is one thing to consider, the efficiency is also in the criteria of the judging.
Most competitions will give you the specific number for it's test cases such as `N<500,000` and you will need to make your solution efficient enough to complete all test cases, even the ones with the highest N.
Big O notation is a topic worth a whole article itself, but it essentially estimates how much time does your program need to complete the calculations. This is often found by how many times you need to iterate through the input and does your solution compound exponentially.
I recommend you look at the question and come up with the brute force solution first - the solution that

simulates what the question is asking. For easier levels of contests, this is often suffice but for higher levels, you will need to either optimize the brute force using various techniques or go a completely different route.

5. Code Splitting & The Debugger

Looking back at `1.` you can see that in the template for competitive programming, I split the code into 2 main sections `Input()` and `Solve()`. This is effectively code splitting.

There are many benefits to code splitting but one of the most obvious is code reusability and the ease to debug. Think about a situation where you need to `binary search` for a value in a list, instead of writing the algorithm many times, why not split it into a separate function?

Almost all languages that you use for Competitive programming has its debugger and incorporating that with code splitting, you will be able to step through your code chunk by chunk, function by function, line by line and locate the issues much more easily.

6. Edge Cases and Logic Charts

Last but not least, setting up flowcharts and identifying the edge cases are also crucial. I recommend you take out a piece of paper or in your text editor, try to draw blocks of operations that move you closer to the solution. **Try to draw the brute force solution first** then see where you can improve on or is there totally new approaches?

Edge cases are values or indices that lie on the edge of the test data and you have to consider them. In these 8 questions, I had to go back and modify the code numerous times to satisfy such cases. The easiest way is to highlight them when you are planning your solution and write them in your flow chart.