NOI.PH Training: DIST

Debugging, Implementation, Strategy, Tactics

Carl Joshua Quines

Contents

1 Introduction

First, some required reading. Uploaded in the Discord channel for this week is the file `IOI 2015 Session 1.pdf`. It's some slides from the Singapore team's IOI training back in 2015. Some of the information is a bit dated (e.g. the IOI is now a decade older than you), but most of it is still relevant. **Read it in its entirety before proceeding.**

Hopefully those slides give you an idea of what this module is about. This is the module for competition strategy and techniques, for dirty tricks and important tips, for classic stories and street advice. This is the distillation of many years of competitive programming wisdom from the NOI.PH team.

Here's an exercise. Read this group of complaints:

- I wish I tried problem X instead of spending all of my time on problem Y.
- I should have aimed for partials instead of going for full points.
- If I only spent a little more time debugging, I would have found the bug that cost me X points.

Now read this group of complaints:

- I wish I knew about segment trees, so I could've solved problem X.
- I should have practiced more divide-and-conquer problems so I would be prepared for problem X.
- If I was only better at math, I would have found the trick to this problem that cost me X points.

Which complaints resonate with you more? Which complaints are more common among contestants? Which complaints do people regret the most? For many of us, **our bottleneck isn't algorithms skill, but strategy.**

No matter how good you are at algorithms, no matter how good you are with the theory, nothing will come out if you can't implement it. Whatever algorithms skill you have is *multiplied* by your implementation skill: if you can't implement, your score will be zero.

The IOI is not a math contest. You do not get points for ideas. You do not get points for algorithms. You do not get points for tricks. You get points if your code works. **The IOI is an engineering contest.** It rewards engineers. Treat the IOI like an engineering contest and you will perform better.

> **Story 1.1.** In the Google Code Jam 2016 Finals, Kevin did *not* aim for first place. In particular, he did *not* aim to beat Gennady, who at the time, won Code Jams 2014 and 2015. Instead, he was aiming for second place, by trying to get as many points as he can. For example, this means prioritizing partial solves over full points.
>
> He considered many of his opponents to have better algorithms skill than he did. But when the contest ended, Kevin was second place. Not because he solved any problems that no one else did, or because he was better at finding algorithms. Kevin won that year because he had a solid strategy.

This module is divided into four parts:

1. **Strategy** refers to overall aspects of the competition. It refers to time management, which problems to try in which order, what to do when you don't know what to do, and so on.

2. **Debugging** refers to fixing your code. A *bug* refers to program error: freezing up a certain test case, getting a wrong answer in a special case, giving output different from what you've expected. Debugging refers to removing bugs by fixing your code.

3. **Implementation** refers to implementation aspects that aren't debugging. It refers to knowing your language well and writing good code.

4. **Tactics** refers to problem-specific tips. It refers to trying small or special cases, sorting the input, processing queries backwards, and so on.

Each part is divided into several sections, each referring to a single piece of advice. The parts are roughly sorted from most to least important. Within each part, the sections are also roughly sorted from most to least important.

Something like the Pareto principle[1] applies here: the first one or two tips in each section will probably cause the biggest effect. Of course, all the tips are useful, but it just happens that some tips are so much helpful than others.

We'll be focusing a lot on specific stories and anecdotes about me or the other authors, which will function as our examples. If people learn from mistakes, then we better learn from the mistakes of others. Plus, stories are easy to remember and transmit.

Stories will often describe the solutions to problems, so consider this your spoiler warning if you want to try the problems on your own beforehand. I'll try to stick to problems from NOI contests to minimize spoilers.

[1] which states that, for many events, about 80% of the effects come from 20% of the causes

2 Strategy

Again, **strategy** refers to overall aspects of the competition. It refers to time management, which problems to try in which order, what to do when you don't know what to do, and so on.

2.1 Aim for subtasks

If you only remember and apply one piece of advice from the entire module, let it be this.

Advice 2.1. Always aim for subtasks.

It means that you should always **solve the subtasks in order, going from the easiest to the most difficult.** There are several good reasons to do this.

A harder subtask may be worth more points, but they also take more time, are often harder to implement, and thus often harder to debug. In contrast, the easy subtasks are easier to solve, and therefore easier to get points in.

In terms of points earned over time spent, **solving easy subtasks is the logical first choice:** they only take a few minutes to think of, a few minutes to program, and give you guaranteed points.

Story 2.2. The points assigned to easier subtasks are small, but they add up. In 2015, Robin won the Philippines's first IOI medal through easy subtasks:

Robin Christopher Yu	Philippines	20	55.55	34	54	36	0	199.55	33.26%	151/322	53.42%	Bronze

A bronze medal is *not* "get all points on a few problems". It is often "get a few points on all problems".

Harder subtasks are risky, and are best saved for later. This is why guaranteed points are good, because they are a safety net. If you don't solve the harder subtasks, you'll still have points.

Story 2.3. In IOI 2018, I spent a lot of my time on Mechanical Doll. I quickly got the idea behind each of the first four subtasks, but decided not to implement them because it was worth "only 26 points." I constantly felt like I was *so close* to solving the last two subtasks to get full points.

Eventually, one hour left in the contest, the idea comes to me on how to get full points. So I program it, and it gets Wrong Answer: my code doesn't work. I debug it for the next hour, getting no progress. In the end, I get 0 points for the problem, when I could have gotten 26.

Any points are better than no points.

The easier subtasks are often a guide on how to solve harder subtasks. This is why editorials always explain the problem, going from the easier to the harder subtasks, because the subtasks build upon each other. Sometimes, the subtasks *directly* build upon each other. Even if they don't, often, the intuition gained from easier subtasks helps solve the harder subtasks.

Psychologically, having guaranteed points from the easy subtasks also helps a lot for stress.

Story 2.4. One of the reasons I was so stressed out during the second day of IOI 2018 was because I didn't have any points. This was really bad for me when I was debugging Mechanical Doll, because I was under a lot of pressure to "get it right or get 0 points". In contrast, if I had solved the easy subtask earlier, I would be less stressed out, which would have made debugging much easier.

A corollary of this advice is to **always write the brute force**, which often solves the easiest subtask. We'll talk more about this in the debugging section, because the brute force is a useful tool when it comes to debugging.

Caution: It's also possible to spend *too much* time on subtasks, rather than stopping to think about a problem. While it's definitely possible to make this mistake, I think that many NOI.PH contestants make the opposite mistake instead, which is why I put so much emphasis on doing subtasks. The best way to find out the correct balance is to do practice contests and find out yourself.

2.2 Avoid tunnel vision

We're all familiar with this feeling. You spend too long on a single problem, always feeling that you're on the verge of solving it. You just have to fix this one bug, or resolve this one special case, and you'd be done. But you end up spending an hour of your time on the problem, without making any progress.

What happened? This is a classic case of **tunnel vision**. Think of a driving through a tunnel, where you only see a narrow circle of light at the end. The phrase is a metaphor for this: being too focused on a single thing, unable to look at the whole. We begin to become blind to the bigger picture. Hence the advice:

Advice 2.5. Never spend too long doing a single thing.

Story 2.6. Again, in IOI 2018, I spent too much time debugging my solution to Mechanical Doll. I got caught up in the thinking that, "if I debug this, I'll get 100 points." This was true, but if you keep thinking the same thing *for an hour*, you know that something's wrong and you should move on.

Story 2.7. It's day two of the NOI 2018 Finals and there's an hour left. I have 356 points, with three full solves, 56 points from People Power, and 0 points on Backbreaking Breakout. My options were either aiming for full solve on People and get 44 points, or to get the easiest subtask of Breakout and get 15 points.

Here's what I was thinking. Dan got full points in People, and I felt like I could definitely get full points too with a little more effort. I ended up spending a whopping fifty minutes on People making no progress, because I convinced myself that I was *so close* to solving it fully.

When I realized what was going wrong, it was too late. I rushed the first subtask of Breakout in the last ten minutes, desperately trying to get the 15 points I needed for first place, but the pressure was too much. If only I realized a little sooner, I could've won gold.

Here's some advice that may work well for you. **Spend at most thirty minutes doing any single thing.** One way to keep track of this is to set up an alarm. On Ubuntu, the

command `sleep 30m && zenity --error --text="Hi!"` on the terminal will make a pop-up window appear after thirty minutes. Time management made easy!

Another good rule of thumb: **if you've made five consecutive wrong submissions, move to another problem**. Of course, you're likely to find that the "sweet spot" of thirty minutes or five submissions is different for you: experiment and find out which works best.

2.3 Read all problems

What's the first thing you should do when the contest starts?

Advice 2.8. Read all problems before doing anything else.

In the IOI, there are only three problems, and you have no excuse not to read all of them fully *before* you do anything else. It's easier to solve the problems if you read all of them first; that way you already have an idea what the different problems are. Reading all of the problems increases your options of what to do at any given moment:

Story 2.9. In the NOI 2017 finals, I took a break to eat some food while debugging The Chenes of the Chorva. My mind naturally went to think about Kapuluan ng Kalayaan, and I had the main idea for the solution while eating.

You also don't want to miss an easy problem! So as an extension, this means giving all problems a serious try, maybe at least ten minutes of effort.

Story 2.10. In the team round for IOI 2017, I read all of the problems quickly, but didn't try City Map because it "looked hard", and didn't bother trying to understand the statement. I read it after the contest, and my first and immediate idea got something like 14 points.

I make this mistake *again* in the team round for IOI 2018, where I read Mood Swings but didn't give it a serious try. When the contest ended, and I tried to upsolve[a] it, I got full points within fifteen minutes.

[a] To upsolve is to attempt solving a problem after the contest

This is more important in ICPC-style rounds, where it's critical to identify the easiest problem as soon as possible. Hence the strategy of distributing the problems to the team members in order to read faster. This is not as big of an issue for IOI-style rounds.

2.4 Read carefully

I'm sure we all have a lot of experience with this:

Advice 2.11. Take your time to read the problem statement very carefully.

Story 2.12. Problemsetters may be working against you. Remember the Week 3 mock, with modulo 1234657890? Try reading the statement of **Confusing Date Format:** ICPC Live

Archive 7711, the easiest problem in the contest it came out in. Near the end of the statement, you have "To punish teams who did not read the problem statement carefully…"

Exercise 2.13. Solve **Dissecting a Hexagon:** UVa 11298. Be very, very careful with the constraints.

You might not think this is a problem for the IOI. Indeed, in the IOI, the problemsetters are trying their best to make the statements as clear and unambiguous as possible. But committee decisions and translation errors can get in the way of this. And sometimes, even if a problem is unambiguous, people will still misread it:

Story 2.14. Read the statement of **The Chenes of the Chorva:** https://www.hackerrank.com/contests/noi-2017-finals-1/challenges/chenes-chorva. Did you notice that the array is bitonic, or that it always decreases, and then increases? Very few people did. The statement is perfectly clear and unambiguous, but even Robin missed that the array was bitonic. For an unsorted array, the problem is much harder; once you add in the bitonic condition, then the problem becomes much easier.

Story 2.15. From Kevin:

> Based on my experience, whenever I try to read faster than my normal reading speed, I too often miss some important detail. So we have an important legit lesson here even if the problemsetter isn't working against you. Never just skim the statement.

2.5 Don't guess, ask

Do not attempt to read the minds of the problemsetters. In other words:

Advice 2.16. If something is unclear after reading the statement carefully, ask for clarification.

You don't lose anything by doing so. You're not going to get any less points. It's only a couple seconds of typing, in exchange for potentially minutes of wasted effort.

If it turns out the answer is "read the problem statement again", then you don't lose anything; at least you're assured that the answer is somewhere in the problem statement. If the answer is something else, then you got something clarified, hooray!

Story 2.17. It's the team round for IOI 2018, and I'm solving Socrates and Plateau. It asks to modify a grid to get the minimum and maximum of a certain property, and you can choose to answer only the minimums, or only the maximums. The output format said to output a grid for *both* the minimum and the maximum, even if you only choose to answer one of them.

I wrote a solution answering all the minimums, which would get me 50 points. For all the maximums, I just output a blank grid. But then I submit, and get 0 points, even though I was sure (through brute force testing) that my code was right. I reread the problem statement, and after thirty minutes of experimentation I finally figure out what's wrong: you need to output a *valid* grid, and the *correct* answer for that grid, even if it's not the correct maximum.

This would have been resolved if I simply asked for clarification. That extra thirty minutes could have been spent getting more points.

2.6 Practice and upsolve

It may seem obvious, but:

Advice 2.18. Practice: do practice contests that are just above your level, write implementations of algorithms and data structures until you know them cold, and upsolve problems after a competition.

It's especially important to listen to obvious advice. The reason it's obvious is because it's way more likely to be true and helpful, and the reason we should pay more attention is because we're more likely to dismiss it. **Never underestimate the power of practice!**

Story 2.19. Check out Kevin's uHunt: https://uhunt.onlinejudge.org/id/95830. In particular, look at the "Progress over the Years" graph. See that huge spike in the beginning? That's May 2011. He made 521 accepted submissions that month. Let me do the math for you: that's an average of 17 problems per day.

He then made 248 accepted submissions in June, 136 accepted submissions in September, 148 accepted submissions in October, and 135 accepted submissions in December. By the end of the year, he's solved more than a thousand UVa problems. Kevin really took practice seriously when he started out.

Although I'm not sure if I can recommend solving a thousand UVa problems, I can definitely recommend spending some time each day or each week practicing, throughout the year.

In the math competition discipline, there's a lot of advice about how to practice. Evan Chen goes over most of the advice here: https://usamo.wordpress.com/2019/01/31/math-contest-platitudes-v3/. In particular,

> For some people, the easiest first step to getting better is to double the amount of time you spend practicing.

And also,

> The common wisdom is that you should consistently do problems just above your level so that you gradually increase the difficulty of problems you are able to solve.

The full post is very good and I strongly recommend you read it. One specific reason to practice is knowing your implementations cold:

Story 2.20. In the NOI 2017 finals, I had to figure out how to program Kruskal's *from scratch* for Kapuluan ng Kalayaan! I knew how the algorithm worked, but I've never implemented it before. Needless to say this did not go well, because I did not know about union–find at the time.

In the NOI 2018 finals, I had to program a sparse table for a range minimum query. Thankfully, I *knew* how to do this, but I spent twenty more minutes doing it than I should have

because I couldn't get the right formulas for combining intervals, and I had to debug a silly error.

Both of these would have been avoided if I practiced more. Even now, I still admit to needing more practice when it comes to implementing data structures.

Finally, it's also important to **upsolve**. This means solving a problem after the contest, possibly after some more thinking, a hint, or reading the editorial. We strongly encourage upsolving because, like in math, in competitive programming the same themes *do* come up again and again.

2.7 Learn from your mistakes

We make mistakes. It happens, even to the best of us. As much as we'd like to learn from the mistakes of others, we often make mistakes ourselves. These mistakes are all the more important for us to learn from, because they're also mistakes that we're more likely to make in the future.

Advice 2.21. Reflect on problems you haven't solved, and reflect after each contest.

Let me make it clear what *reflection* entails. For a specific problem, you don't want to just read the editorial and upsolve. The question you want to ask yourself is, **how could I come up with this solution?**

The competitive programming tradition is luckier when it comes to editorials, because more often than not they emphasize main ideas rather than specific details. In practice, the solution to a problem can be summarized in a phrase, or a few sentences. What's the shortest thing you can tell someone who's struggled with the problem, such that after you tell them, they can figure out the rest on their own?

For example, here's this year's finals, summarized very quickly: "root the tree and do DP", "use merge sort", "use a sieve", "place the partitions one-by-one", "greedily assign people to love", "group the partitions by minimum element", "scale the map". It's a useful exercise to figure out what the main idea is, because it helps you understand a problem more.

After extracting the main idea, you're ready to ask yourself the question of how to come up with the solution. Here are some helpful questions:

- *Does it involve a technique you don't know?* Then read about the technique, learn it, and find problems to practice it on until you know it.
- *Does it involve a technique you know, but didn't recognize?* Ask yourself: what about the problem signals the use of this technique? If a problem that can be solved with this technique comes up, how will I be able to recognize it?
- *Is it similar to a problem you've solved before?* Then compare the problems and ask yourself: what's similar with these two problems? What are the similar ideas, and what are the signals for me to use this idea?

If you can't find the answer to these questions, it may be helpful to ask other people who've solved the problem how they came up with the solution.

Story 2.22. During the NOI.PH 2018 month-long Christmas practice round, I solved Crazy Planets, which involved computing the GCDs of subarrays. One of the ideas behind the solution was that computing GCDs of subarrays had a lot of overlap: $\gcd(a, b, c) = \gcd(\gcd(a, b), c)$.

When I think about how I came up with this, it's because I've seen this pattern so many times before in similar problems. If you want to answer a query about subarrays, think about how the answer the changes when you add a new element at the end.

This is the idea behind one of the subtasks in The Chenes of the Chorva, and one of the subtasks of Sorting the Planets, and it's a super common idea in general. How else could I have come up with this idea, if I haven't spent time reflecting on The Chenes of the Chorva after the 2017 Finals?

It's also a good idea to reflect about contests as a whole. Consider something like http://cjquines.com/files/noi2017team.pdf. I'm not referring to the whole document here; instead, focus on the section Hindsight. Here, I talk about what I thought I did well, and what I thought I did badly.

Story 2.23. Here's a quote from that document:

> The tactical error I can assign a real cost to, however, is Lapida. I have missed 23 points because I forgot to reset my variables in between cases.

Guess what? I've never made that mistake again.

Reflect about your mistakes after each contest. What went wrong? Just as important a question, but often neglected, is: what went well?

Here's some advice that may work well for you; it certainly worked for me. **Write down what you've learned**, and store them somewhere you can refer to. Aside from the contest reflections I post on my website, I also have this document where I group problems according to the technique I've used to solve them. Over time, I started to see patterns between problems that I wouldn't have noticed otherwise.

Story 2.24. Two problems that look different at first but feel similar to me are Jabahw from the 2016 Team round and Agadoo from the 2017 Team round. The solutions involve very different techniques: Jabahw needs a segment tree while Agadoo is a sliding window minimum. But instead, I felt they were similar because of the idea to "take the perspective of the queries, and not the contents."

This isn't an idea I can easily articulate, and it's not a perspective that's useful to everyone. But the important part was that this perspective that's useful to *me*, and spotting this helped me apply it to solve a lot of other problems in the future.

Evan Chen again wrote about this on https://usamo.wordpress.com/2017/03/06/on-reading-solutions/, if you're interested on learning more. It's a good read, but some of the points aren't directly applicable to competitive programming.

2.8 Take care of yourself

Finally, **remember you're human,** and you're affected by human factors. So,

Advice 2.25. Take care of yourself: get good sleep and take breaks.

My biggest piece of advice here is **get enough sleep the night before a contest.** This is very important, and people underestimate the importance of good sleep. Even if you *think* you can perform well without much sleep, like I used to, it's a fact that getting enough sleep can mean the difference between winning or not.

The facts: investigators ruled sleep deprivation as playing a critical role in the Challenger disaster. By reducing a doctors' work shifts to 16 hours, and thus allowing them more sleep, medical errors were reduced by as much as 36%.[2] Drivers who get only five to six hours of sleep have 1.9 times the crash risk.[3] So even reducing the amount of sleep you get by an hour can impair you *a lot*.

You are also not expected to have the stamina to write code for five hours straight! Stand up, walk around, and eat food. It's hard to think on an empty stomach; I probably don't need to cite anything for this.

Story 2.26. Personally, I find it much more difficult to think when I'm hungry. One of the reasons I did so poorly in PMO 2016 Nationals was because I didn't have breakfast. As soon as I ate lunch, I quickly found the solutions to the problems I didn't solve. Since then, I've made sure that I start contests on a full stomach.

Taking breaks is also helpful, especially if you've been working on the same thing for a while. Whether it's an indirect break, like doing something else for a couple minutes, or a direct break like walking to the restroom and back, I find it easier to approach a task after a break.

The science also backs us up. See Smith (2002), "Getting into and out of mental ruts: A theory of fixation, incubation, and insight", or Dijksterhuis and Meurs (2006), "Where creativity resides: The generative power of unconscious thought". Both studies support that taking breaks leads to better results when it comes to problem-solving.

Story 2.27. I was debugging my solution to Agadoo in the 2017 Team round. It got 8 points, when I was expecting 55 points. After spending some time trying to solve Spreadsheet Game, I still couldn't debug it, so I decided to take a break to eat some food. I came back to it and I found the error pretty quickly, getting 47 more beautiful points. I think that the reason I found the error was because I took a break.

[2] http://healthysleep.med.harvard.edu/healthy/matters/consequences/sleep-performance-and-public-safety

[3] https://www.forbes.com/sites/tanyamohn/2019/03/10/sleeping-less-due-to-time-change-missing-just-1-2-hours-nearly-doubles-crash-risk/

3 Debugging

A **bug** refers to program error: freezing up a certain test case, getting a wrong answer in a special case, giving output different from what you've expected.[4] **Debugging** refers to removing bugs by fixing your code, and is often the bane of many competitive programmers.

3.1 Avoid silly mistakes

The worst kind of bugs are the silly ones. The stupid mistakes. Not using `long long`, not resetting your variables, out of bound indexing, off by one errors, edge cases…the list goes on.

Advice 3.1. Stop making silly mistakes by developing good habits and being organized.

In the math contest tradition, we called these mistakes *sillies*, ranging from arithmetic errors, shading the wrong choice, misreading the question, and not answering in the right units. Richard Rusczyk recounts the typical advice in Stop Making Stupid Mistakes: https://artofproblemsolving.com/news/articles/stop-making-stupid-mistakes. While many of the tips there don't apply to competitive programming, one of them does: **develop good habits**.

Story 3.2. In the 2017 Team round, I was incredibly confused when none of my submissions got any points for Lito Lapida's Lost Lapida, when I was expecting 23 points. When the contest ended, Kevin asked me if I realized my mistake, and I told him I didn't. He revealed that I wasn't resetting my variables. That was *23 points gone* because I didn't reset my variables!

I've never made that mistake since, because every time I declare a global variable, I immediately write the code that resets it at the beginning of each test case. Good habits!

It might help to **use a checklist to avoid making silly mistakes**. Benjamin Qi (username Benq), who topped IOI 2018, ends his source code template with the following lines:

```
/* stuff you should look for
  * int overflow, array bounds
  * special cases (n=1?), set tle
  * do smth instead of nothing and stay organized
*/
```

It only goes to show that yes, even people at the highest level of competition are susceptible to these kinds of mistakes. But we don't see them make these mistakes because they've developed the habits to avoid them, even if that habit is checking things off from a checklist. Something like:

- Did I reset all my variables?
- Did I use `long long` if I need to?
- Are my array bounds set correctly?
- Am I not making out-of-bounds indexing?
- Am I not making off-by-one errors? (Are all my intervals right?)
- Does my code work for $n = 0$, $n = 1$, the maximum n?

[4] The etymology is interesting, see https://upload.wikimedia.org/wikipedia/commons/8/8a/H96566k.jpg.

Story 3.3. Again, in the 2017 Team round, I was puzzled when I was debugging Agadoo: I was getting 8 points instead of an expected 55. The 8 points was from passing the first subtask, which only dealt with small updates. So I decided to check if my code would pass if I gave a test case with the largest possible update, and it was taking too long.

Aha! I was getting Time Limit Exceeded because my code was failing for large updates! I quickly fixed this (by *commenting out* code rather than adding code, oddly enough), and resubmitted, just before the time ended, and got the 55 points I was expecting.

Still not convinced by the power of checklists? Preparing problems for NOI is basically filling out a bunch of checklists. For each problem, we need to check if the constraints are followed, if there's a test case for the minimum and maximum constraints, if the subtasks are graded correctly, if the scores add up to 100, and so on. NOI problems are consistently high-quality because of good testing, which is much easier with a checklist.

Testing document	chocolates	robots	mystery	city-map	bato	arnibal
Is the story complete?	yes (CJ)	yes (CJ)	yes (Kevin)	yes (CJ)	yes (Jared)	yes (Jared)
Is the test data ready?	yes (CJ)	yes (CJ)	yes (CJ)	yes (CJ)	yes (Kevin)	yes (Vernon)
Is the test data correct?	yes (Kevin)	yes (Kevin)	yes (Kevin)	yes (Kevin)	yes (CJ)	yes (Kevin)
Do intended solutions in different languages pass?	yes	yes	yes	yes	yes	yes
Are the constraints followed?	yes	yes	yes	yes	yes	yes
Are the constraints tight?	Ensure that there are data for every extreme case. E.g. if the constraints say 1 <= n <= 10000, there should be a case with n = 10000 and n = 1.		yes	yes	yes	yes
Is the statement clear and unambiguous?			yes	yes	yes	yes
Are **all** the slow solutions failing?	yes	yes	yes	yes	yes	yes
Are **all** the weak solutions/heuristics failling?	yes	yes	yes	yes	yes	yes

And if you're *still* not convinced that checklists are good, you may want to read Atul Gawande's *The Checklist Manifesto*. Did you know that in Johns Hopkins ICU, a checklist as simple as "wash hands with soap, clean the patient's skin, put drapes, wear sterile clothes, put dressing" prevented 43 infections and 8 deaths over the span of a whole year? You'll learn about this and dozens more checklist facts if you read this book.

It may look silly, but checklists save lives—and checklists save points. However you do it, checking against silly mistakes is the crucial first step to debugging anything.

3.2 Write the brute force

I've said it before, and I'll say it again. When dealing with a hard problem,

Advice 3.4. Always write the brute force!

This is in line with aiming for subtasks. Often, the easiest subtask can be solved with a simple brute force program. So it's free points! But more than that, **the brute force is an**

incredibly useful debugging tool because it's useful as a source of counterexamples.

In particular, what you want to do is to write a solution using brute force, and write the solution you want to this. You then generate thousands, or better, millions of small random cases, and run both solutions using these as input. Then you compare. If they're the same, you can be reasonably confident your solution works. If they're different, you know the solution is wrong and you also have a counterexample.

Story 3.5. While I was the one who initially came up with the idea behind Sorting the Planets in the 2019 Finals, it was Kevin who proposed taking the sum over all subarrays. I had several ideas for the full solution, and Kevin gave me counterexample after counterexample for why they wouldn't work.

He showed me how he was just comparing my attempts against a brute force, so I decided to do the same myself. After all, if the logic of a solution is wrong, it's pretty likely that there's a small counterexample. Since then, my problem-solving life has been changed forever: it's *so* much easier to fix a Wrong Answer when you have an actual counterexample.

Note that you want to generate random tests automatically using a computer. This is strictly better than testing programs by typing "random" inputs in by hand, which is much slower, and much worse than actual random inputs.

Story 3.6. In DISCS PRO 2018, we were debugging some prime factorization code. It took us several minutes before we realized what the mistake was: it wasn't including any prime factors that are larger than 10^6, because of a bug in our code. If we simply tested using a random number generator, we would have found this mistake much more quickly.

This is called **stress testing**. Stress testing, coupled with a brute force solution, again helps tremendously with debugging. When wrong submissions have penalties, like in the ICPC, it's also better to stress test rather than attempting a "proof by AC". The term is borrowed from engineering, because you're placing your solution under a lot of "stress", by checking millions of random cases, and seeing if it doesn't break.

3.3 Write clean code

In line with "prevention is better than cure", it's often better to write bug-free code in the first place.

Advice 3.7. Write clean, organized code!

It is much easier to debug clean code than messy code. I shudder when I look at solutions with incomprehensible variable names, poor indentation, copy-pasted code, and single lines of code that try to do too much.

Story 3.8. My code for Spreadsheet Game in 2017 Team was horrible. If I remember correctly, I copy-pasted the same dozen lines of code four times. So every time I had to edit one of the copies, I had to make the same edit to the other three copies too. This is *extremely* error-prone, and if I instead cleaned up my code to avoid this, I could have saved a precious thirty minutes debugging.

More on this in the Implementation section, which is a whole section dedicated to writing good code.

3.4 Print the state

The first debugging tool that everyone learns is the print statement: it's simple, fast, and versatile.

> **Advice 3.9.** If you're not getting the output you want, print the involved variables at various stages in the program, so you can isolate which part of the code has the error.

An effective way to use print statements is to **isolate through binary search**. Let's say `x` should have the value 10 at the end of the program, but it's actually 9. So you find where `x` is introduced, and find the end of the program, and add a print statement somewhere in the middle. If it has the value you want, then the error has to be in the second half, and if it doesn't, then the error has to be in the first half. Rinse and repeat.

Be careful with print statements in C++. Due to something called *buffered output*, it's possible that your program stops running before everything in the output is printed. So this can cause trouble if you're debugging a runtime error. To avoid this, make sure your debug statements are *flushed*; an easy way to assure this is to output `endl`.

In line with printing variables, you may want to **learn how to use a debugger**. Debuggers are much more powerful than print statements, because you can step through the code *as it runs* and examine the values of variables live. Some examples of debuggers that you can use with C++ are `gdb`, `valgrind`, and `ddd`.

I tried to learn `gdb`, but in the end, I still preferred lots of print statements. On the other hand, Tim has had great success with `gdb` and `valgrind`, so it's worth a try. Oh, and `ddd` is like `gdb`, but with a graphics interface. It's *much* easier to use than `gdb`, so I'm going to give it the strongest recommendation: look at the tutorial on https://www.gnu.org/software/ddd/manual/html_mono/ddd.html.

3.5 Rubber duck debug

Some people find **rubber duck debugging** useful.

> **Advice 3.10.** Try debugging your code by forcing yourself to explain it line-by-line.

The name comes from the story of a programmer carrying around a rubber duck to apply this method to. The programmer would tell the rubber duck what each line fo code was doing. Often, they would notice that a line of code was doing something different from what they were expecting, which led to fixing the code.

Supposedly, the reason the IOI allows contestants to bring small mascots (often a small, stuffed animal), is to help them rubber duck debug. Personally, I only turn to rubber duck debugging as one of my last resorts—if the checklist, brute force comparison, and copious print statements fail, then I might force myself to rubber duck debug to try to figure out what's wrong.

3.6 Print your code

Some people find **printing code** and debugging without a computer useful. By *printing* here, I mean *physically printing onto paper*.

Advice 3.11. Try printing your code and debugging it on paper.

In both the IOI and the ICPC, you're allowed to print documents. You can use this to print your actual code, print debugging output, or print output to look for patterns. This is much more important in ICPC-style contests because you're in a team of three working on a single computer, and sometimes you have to just print your code and debug offline while another member is typing.

I personally haven't tried doing this, but it's worth considering.

3.7 Rewrite

As a last resort, and *only* as a last resort:

Advice 3.12. Try rewriting your entire program from scratch.

When all else fails, sometimes you just have to give up and rewrite. I've never had to do this during a contest, and if you find yourself having to do this too often, you might want to consider developing better coding habits.

4 Implementation

This section, **implementation**, refers to implementation aspects that aren't debugging. It refers to knowing your language well and writing good code. As mentioned in the debugging section, good implementation prevents having to debug in the first place, hence why it's *very* important to write good code.

4.1 Know your language

How well do you know C++? How do you access the integer in front of a `stack`? (Is it `top` or `front`?) Which integers are converted to true? What's the result of `6 + 10 >> 1`? How do you initialize a `vector` to have 100 `int`s, all of them -1? What's the best way to sort an array, and what if you want custom comparison? What is `auto`, and does it make your code run slower?

Advice 4.1. Know your language, or at least know where to look things up.

I don't think I know C++ very well. There are features like shared pointers, thread safety, error handling, dynamic memory management, polymorphic function wrappers, and so on, that I have no clue how to use. But I'm confident of the parts that I do know:

- `stack` uses `top` to access the top element. It's `queue` that uses `front`.
- The integer `0` is converted to `false`, everything else is converted to `true`.
- It's parsed as `(6 + 10) >> 1`, which gives `8`. In particular, it's not `6 + (10 >> 1)`. This is because addition has higher precedence than bitwise shifts.

Story 4.2. This has tripped me more than once. I was debugging some code for a sparse table, which used `1 << n` to compute powers of two, and it kept throwing errors. It took me twenty minutes before I realized that the mistake was the bitwise shifts.

- You write `vector<int> v(100, -1);`. Sure, you can set the elements one-by-one, but this is much more concise.
- Often, the best way is the `sort` function: something like `sort(begin(a), end(a));`. This works even if `a` is a C-style array. I'd implement custom comparison with a lambda expression.
- `auto` is a keyword that asks the compiler to figure out the type of a variable, which is often helpful for iterators. Since types in C++ are checked during compilation, it doesn't make runtime slower at all.

And even if you *don't* know these, you should at least know where to look for these. The IOI allows access to the C++ reference, which you can access on https://en.cppreference.com/. If you want offline access, you can download the program Zeal, which also uses the same reference. With the C++ reference, here's what I'd search to find the answers to the above questions.

- "stack". This directs me to the page https://en.cppreference.com/w/cpp/container/stack. Here, I can look down the member functions, and see that `top` is one of them. If I want, I can even click the link and get an example of how to use it.

- "types". This directs me to the page https://en.cppreference.com/w/cpp/language/type, where I click the link *fundamental types*. Skimming the page shows that it doesn't have the right information, but it does have a link to *implicit conversions*. That page does: it tells me that the value zero becomes `false` and everything else becomes `true`.

  You may have noticed how roundabout this is. The inbuilt search on the cppreference website is really bad, it doesn't show the implicit conversions page when I search "conversion". Hence why you often want to make your search as general as possible. (Zeal does not have this problem, and searching "conversion" in Zeal gives me the page I want.)

- "precedence". On the website, this redirects me precisely to the page that I want. In particular, note that bitwise shifts have lower precedence than any arithmetic operation; this is something that has caused me at least two bugs. The easiest way to avoid this is to always use parentheses when working with bitwise shifts.

- "vector". I click the link for *std::vector*, and then I click the link for *(constructor)*. I then completely ignore *everything* on the page, and then move all the way to the example. The last example gives me exactly the format I want to use.

- "sort". Again, I click the link for *std::sort* under the C++ heading, ignoring the result under the C heading. Then I ignore everything and go straight to the example. This tells me how to sort using `begin` and `end`, and I combine that with my knowledge to figure out how to use it for arrays.

  For custom comparison, the last two examples give me a format I can use. I prefer copying the latter example, which uses a lambda expression. If I forget how to write lambda expressions, I can search "lambda", ignore all the text, and look at the examples until I find what to use.

- "auto", then the *placeholder type specifiers* link. Of course, I would need to know that a thing like `auto` even exists in the first place.

The last example above shows a general problem. While I can look up how to use a feature I know *exists*, it doesn't tell me what features exist in the first place.

Is there a function that sorts? (Yes.) Is there a function that reverses vectors? (Yes, `reverse`.) Is there a function that does binary search? (Yes, `lower_bound` or `upper_bound`.) What about a function that lists all subsets? (No.) These ones are pretty easy to answer: just look up the algorithms library.

But then you have questions about syntax, which are harder to answer. Is there some nice syntax for iterating over vectors? (Yes, range-based for loops.) Is there nice syntax for swapping two integers without a temporary variable? (Yes, `swap`.) What about Python-style multiple assignment? (Kinda, `tie`.)

There's not really a centralized reference for these, but I will direct you to two good Codeforces posts: https://codeforces.com/blog/entry/10124, and https://codeforces.com/blog/entry/15643. These go over some of C++ features you might not know about.

4.2 Know your environment

While you can look up C++ features, there are some things that you just *have* to know in advance. In particular, I'm talking about how to use the terminal and the compiler, and knowing the specifics of the software on IOI laptops.

Advice 4.3. Know your competition environment.

While IOI 2019 has not released the specifics of the competition environment yet, we can be sure that it'll be similar to last year's, which you can view on https://ioi2018.jp/competition/competition-environment/. Take note of the following information:

- The laptops are running on Ubuntu, which you should be familiar with from the NOI Finals. You should at least know how to navigate the terminal on a Linux system, `cd`, `ls`, `mkdir`, `rmdir`, `mv`, `cp`, `rm`, on the bare minimum.
- Read which text editors the laptops have. The Sublime Text used is Sublime Text 3. There was no Visual Studio Code during the IOI, and JetBrains was eventually provided but buggy, if my memory serves right. Best to stick to editors that are neither.
- Submissions are compiled with C++14. So yes, you can use C++11 and C++14 features. I'm not sure if this will change in IOI 2019, which *might* use C++17, but I'm doubtful. The exact commands used to compile are given in each task, so you don't need to worry *too much* about the compile commands.
- There is both Python 2 and Python 3. Although you're not allowed to *submit* in these languages (yet), you can definitely use their interpreters and run scripts. Also note that the debuggers `gdb`, `ddd`, and `valgrind` exist.

When compiling, I always use the flag `-Wall`, which gives several useful warnings. Vernon is an advocate of using `-g` and `-fsanitize=address,undefined` so that runtime errors produce more useful debugging output; I have it turned on and it helps quite a bit.

In line with that, get to know your text editor well! I've found much use out of my text editor, while other people, not as much. In Sublime Text 3, to compile and run my code, all I have to do is press F7 after setting up the build system: https://www.sublimetext.com/docs/3/build_systems.html.

The most important piece of advice here is to **use the practice round to test everything**. The practice round is there for you to practice using the laptops. Practice going around the terminal, practice compiling code locally, practice using your text editor.

And even practice printing, practice asking questions on the system, practice submitting code, wrong code, and check how fast you get a response. You do *not* want to figure out how to request for food midway through the contest when you're starving and need to desperately eat. Figure everything out during the practice round, and if you don't know how to do something, *ask*.

4.3 Don't repeat yourself

We now go to the details of actually writing good code. While I can't give you a whole lecture, there's a single piece of advice that will make your code much, much better:

Advice 4.4. Don't repeat yourself.

Let me use the verb **factor out**. In math, $xy + xz$ turns to $x(y + z)$ when we factor out the x. The x, instead of being written twice, is written only once. Similarly, in code, to factor

something out is to take something that's written multiple times, and *factor it out*, so it's only written once. Hence the term **refactoring**.

For example, constants should only be written once throughout the whole program. You don't want to write `% 1000000007` every time you need to take modulo. Write `const ll MOD = 1000000007;` in the beginning of your program, and use `% MOD` instead.

Generally, the red flag should be copying and pasting. If you ever find yourself copying code and changing the variables, think about whether it's really necessary. Can you change it to a function instead? I talk more about this in the editorial for Exchange Gift, on https://www.hackerrank.com/contests/noi-ph-2019/challenges/exchange-gift/editorial, so I'm going to practice not repeating myself by linking you there instead.

4.4 Avoid floating point

There's the usual spiel about avoiding floating point calculations that you hear when you first start learning about computational geometry. Floating point is bad, it is evil, using it will mess your code up, oh the horror. This, thankfully, will *not* be a big factor in the IOI, as the syllabus explicitly recommends that everything be doable using strictly integers. Hence the recommendation to:

Advice 4.5. Stick to integer calculations as much as possible.

The common case you want to do this is with comparing ratios. If we want to minimize a quantity like a/r, then instead of comparing if $a/r < a'/r'$, we instead want to ask if $ar' < a'r$ after cross multiplying. That way, we can compare fractions using strictly integers, without having to resort to floats.

Sometimes it's unavoidable to use floats, especially if the answer format asks us for them, such as in a probability question. We can again avoid this by sticking to integers for the entire calculation, and converting to floats and doing division only at the end of the computation.

This is a different issue when joining ICPC-style contests, however, where you can almost always expect one or two geometry problems with every set. There are some general rules: stick to `long double` as much as possible, never use `==` to check for equality, beware of catastrophic cancellation, and so on. If you're interested, you can read more about this in any discussion on floating point for competitive programming.

4.5 Optimize constants

It happens. The problem has $n \leq 100\,000$, your $\mathcal{O}(n \log n)$ code *should* work, but it runs a tad above the time limit—local testing on the worst-case shows it runs in 1.5 seconds rather than 1 second. What do you do?

Advice 4.6. Know how to optimize constants.

To *constant optimize* is to make a solution faster without making it better asymptotically.

Story 4.7. Your code fails, you add `ios_base::sync_with_stdio(0);` at the beginning, and then all of a sudden it passes. We've all heard this story; it's happened to me too many times

before I learned my lesson. This is an example of constant optimization. Your solution still has the same complexity, it just runs faster.

Not handling large input and output properly is probably the most common case of needing to constant optimize. If you make it a habit to add this line, or if you use `scanf` and `printf`, then you're saving yourself from a lot of trouble down the road.

Big O notation hides the constant from us, and sometimes we can't neglect the constant when it is large. For example, we can treat $n \times n$ matrix multiplication, which is $\mathcal{O}(n^3)$, as constant when n is small. But when n gets larger, we have to start paying attention to it. It's important to pay attention to hidden constants.

Story 4.8. Sometimes, `map` is faster than `unordered_map`. You may be puzzled: `map` does things with $\mathcal{O}(\log n)$ complexity, but `unordered_map` does things with $\mathcal{O}(1)$ complexity. This is true, but sometimes the constant in `unordered_map` is large enough that `map` is faster.

And when I was starting out, I have made several solutions pass by simply changing `unordered_map` to `map`. This is particularly important to remember: your default dictionary should be `map` and not `unordered_map`.

Story 4.9. Kevin shares: "I've had submissions where quick sort passes but counting sort doesn't. A similar thing happens sometimes with tree sets and hash sets."

If you're unfamiliar, quick sort is $\mathcal{O}(n \log n)$ while counting sort is $\mathcal{O}(n)$, and `map` and `unordered_map` is an example of the tree set/hash set distinction. There are good reasons why quick sort and `map` have much smaller constants, which we'll discuss a bit later.

For many problems that are set *well*, you will not need to optimize constants. Authors take great strides to prevent this from needing to happen. If the goal is to separate $\mathcal{O}(n \log n)$ from $\mathcal{O}(n^2)$, then it's fine to be a little lax with the data and set $n \leq 100\,000$, even if the setter's code works with larger n.

It takes work to find the right data, so that the right solutions, even with somewhat bad constants, will win over heavily optimized but asymptotically worse solutions. So if you're getting a TLE, don't turn to constant optimization immediately. It's possible you have a bug that's making your code slower than it should be, or you calculated the asymptotics wrong.

But sometimes it doesn't work. Sometimes, authors have to separate $\mathcal{O}(n)$ from $\mathcal{O}(n \log n)$. Sometimes, the constants involved are just really bad, like in DP with matrix multiplication. Sometimes, the setters have to make compromises. Sometimes, constant optimization is even part of the problem:

Story 4.10. From Kevin again:

> A certain problem, I think from one of the JOIs, needed constant optimization. One natural solution made use of sets, but in fact, we could have just used sorting and binary search. Both are $\mathcal{O}(n \log n)$, but the latter one had a much smaller constant. In fact, the ratio of the constants is large enough to be distinguished with the time limit.

Whatever the case, it happens that we need to constant optimize. While there are several tricks to doing this, sometimes the most effective ways to optimize are "obvious":

Story 4.11. Remember Lots of Cookies from 2019 Eliminations? While my solution passed, it was running dangerously close to the time limit. We want tester solutions to run well under the time limit. For example, for the default 1 second time limit, solutions should run under 0.7 seconds.

So I was trying to figure out what made my solution slow. I had the function $f(n) = an^2 \oplus bn$, and the main loop in my code had `ans[j] += f(i) - f(i - 1)`, where j looped over a lot of numbers. Kevin pointed out that the function f is actually expensive to compute, and that by simply factoring it out, the solution could be made much faster.

So that's the first thing to try: if you're computing something over and over, even if it won't affect the complexity, try factoring it out.

There is a *lot* to be said about constant optimization, so here's a sampler:

- Why is quick sort often faster than counting sort? The answer is something called the *cache memory*, something that stores previously accessed data so that future requests can be made faster. If a processor only has to do operations with data in the cache, it's much faster. This principle is called *locality*.

  Because quick sort changes the array in place, it only access the same, contiguous block of memory: the array it's working on. In the later stages of the algorithm, when the subarrays are small, they can fit entirely within the cache, making it work much faster. In contrast, counting sort does *not* have good locality.

- From personal experience, dynamic memory allocation is a big one. When using custom classes, it's often faster to allocate the memory all at once in the beginning of the program, rather than allocating it when it's needed. This is called *arena allocation*, which again tries to use locality.

- Unrolling recursions, as discussed in Backtracking 1, is also sometimes a big one. Because function calls write and read more data than raw loops, converting recursion into iteration often makes code faster.

- When it comes to picking the block size in square root decomposition, it's often better if you don't pick block sizes that are near $\sqrt{n}$. Sometimes you want to pick sizes closer to $2\sqrt{n}$ or $\frac{1}{2}\sqrt{n}$, or maybe a different constant altogether, because of the hidden constants in $\mathcal{O}(\sqrt{n})$.

Take careful note of me using the word *often* throughout this section. **Constant optimization is never a guarantee**: it's not something that always happens. To really find out which one is faster, the best way is to test both yourself, for the specific application, and see which runs better. You have to **benchmark** your code to find out which is best.

Here's an example of why benchmarking is important. Consider these two snippets of code, both of which calculate the first n Fibonacci numbers. Which do you expect to be faster?

```
for (int i = 0; i <= n; i++) {
  if (i <= 1) fib[i] = i;
  else fib[i] = fib[i-1] + fib[i-2];
}
// versus
fib[0] = 0; fib[1] = 1;
```

```
for (int i = 2; i <= n; i++) {
  fib[i] = fib[i-1] + fib[i-2].
}
```

You might expect that the second one will be faster, because it doesn't have the conditional. But, when compiled with `-O2` under my system, they both run in the same time! This is because of *branch prediction*, which you can read about on https://stackoverflow.com/questions/11227809/.

Compilers are smart, and sometimes an optimization you thought of, the compiler has already thought of as well. If you're interested, the loop optimization page on Wikipedia gives a lot of examples: https://en.wikipedia.org/wiki/Loop_optimization.

5 Tactics

This final section, **tactics**, refers to problem-specific tips. It refers to general problem-solving strategies like trying small or special cases, sorting the input, processing queries backwards, and so on. Some of these have already been discussed in their respective modules, but it makes sense to repeat them here.

5.1 Get your hands dirty

If you only remember one tactic, remember this one:

Advice 5.1. Experiment! Try small or special cases, look at edge or extreme cases, enumerate all possible answers and look for patterns. If it's faster to do by hand, then do it by hand: use the scratch paper provided to you.

Story 5.2. I solved Sorting the Planets from 2019 Finals through lots and lots of experimentation. You should see how much scratch paper I used to solve it!

I programmed the brute force to output the actual partitions, generated lots of random examples, and copied the output to my text editor to look for patterns. And I would come up with guess after guess about how to solve it. So I drew out diagrams on paper, tried more examples to verify if my guesses made sense.

When I had a guess that looked like it would work, I programmed it and tested it against the brute force. There was a counterexample, so I modified it a little, and then it finally worked.

Trying small cases is an important skill. Trying special cases is another important skill, because sometimes special cases reveal things about the problem that can be used to solve the general case. This is why subtasks are almost always special cases of the general problem. And this is also why I'm heavily encouraging you to solve subtasks, because they give the insight needed to solve the full problem.

Note how I used a computer in the above example. Some things are better to be done by hand, but some other things are better to be done with your computer. If I find myself spending a lot of time writing out examples, I'll definitely get a computer to generate them for me, along with whatever output I need.

Story 5.3. One time I *did not* use a computer was with Betty's Bitter Batter Bother from 2017 Finals. I was listing down a lot of things by hand, and making wrong guesses because I had wrong data. I eventually got enough data to make the correct guess, but I had already wasted thirty minutes. If I programmed the brute force to give me the data I want instead, I would have saved that time.

In general, you can tell how it's helpful to have strong pattern recognition skills. Combinatorics problems, for example, sometimes boil down to recognizing a sequence in order to compute it quickly.

For the above problem, the solutions to $B = 1, 2, 3, \ldots$ are $1, 2, 4, 10, 26, 76, 232, \ldots$. The first thing to do with any sequence is examine the common differences: here it's $1, 2, 6, 16, 50, 156, \ldots$. What's odd is that the ith term here is divisible by i, so we divide it to get $1, 1, 2, 4, 10, 26, \ldots$. But this is the original sequence! So, magically, we get a recurrence for it without having to do

any actual proving.

Story 5.4. Again, here's another story about Kevin during Code Jam 2016 Finals. Problem B, Family Hotel, is a combinatorics problem with only two integers as input. So the natural thing to do is to consider small cases and make a table, and then try to look for patterns. This strategy actually worked, and Kevin ended up being the third to solve the problem!

You can watch it happen live on https://youtu.be/jIYzkvpzk9M?t=1277. The link starts at the time Kevin codes the brute force, which he runs at 23:30. A few seconds later, he has a guess for the pattern, and checks it by programming it. It's interesting to watch his process, and I strongly recommend watching it.

In general, guessing a polynomial or a linear recurrence given small cases is doable if you know how to look for it. Kevin talks more about this in `#random` on the Discord: try searching for `recurrence` to find it.

Exercise 5.5. Here are some sequences to try recognizing. Yes, you can use the OEIS, but that would be boring. Don't take these too seriously:

(a) 2, 4, 7, 11, 16, ...
(b) 4, 10, 22, 46, 94, ...
(c) 1, 5, 13, 29, 61, 125, ...
(d) 3, 5, 11, 21, 43, 85, ...
(e) 1, 1, 2, 3, 6, 10, 20, ...
(f) 1, 2, 6, 20, 70, 252, ...
(g) 1, 3, 15, 84, 495, ...
(h) 1, 10, 126, 1716, ...
(i) 1, 3, 11, 43, 171, ...

Also under getting your hands dirty is examining the data yourself!

Story 5.6. For Tower Bloxx in 2019 Finals, examining the data reveals that many of the test cases are randomly generated. So that means we don't have to worry about the worst-case scenario of whatever sort we're using, and we can use heuristics to make it go faster. Examining the data also reveals patterns like long, increasing subsequences, that encourage us to treat these cases separately.

Often "machine learning" problems like IOI 2013 Art Class are actually about pattern recognition. The problem asks us to classify paintings into several categories, and we're given some examples of paintings. The trick, then, is to get our hands dirty: look at the given examples and try to find a pattern. For example, impressionist landscapes have a lot of green, so one thing we can try is classify a painting as impressionist if its average color is greenish.

5.2 Change the order

Another useful trick is to change the order we do things, which is especially helpful for data structure problems:

Advice 5.7. Check if order matters. Is the problem solved easier if you do things backwards? Is it easier to answer queries if you answer them out of order?

Story 5.8. Consider Global Warming from 2018 Finals. The main idea for solving this problem is realizing that it's easier to answer it backward: instead of visualizing the water as rising, instead imagine that the water is sinking. We can then use union–find to answer this problem. My mistake, at the time, was that I didn't try to solve this problem because I thought it would be hard to implement.

By "doing this backwards", this doesn't necessarily mean answering the queries backwards. It can be running the whole process backwards, as in the above example. Similarly, if a process on a graph only involves removing edges, it's often easier to run it backwards and turn it to adding edges instead.

Exercise 5.9. Solve **Frog Pushers:** ICPC Live Archive 7856. (This is a problem from ICPC Manila 2016, by the way.)

Doing queries out of order is what **offline algorithms** do, in contrast to an online algorithm that answers queries as they are given. The classic example of an offline algorithm is **sqrt decomposition on queries**, as discussed in DS 2. Here, we sort the queries in a certain order and use the solution for one query to solve the next one. By sorting in a way that adjacent queries have a lot of "overlap", we get a faster solution.

5.3 Add structure

It's often easier to work with structured data than unstructured data.

Advice 5.10. Add structure or transform the input. Sort the input, root or flatten trees, compress coordinates, and so on.

Story 5.11. The first thing to try with any tree problem is to root it, if it's not already rooted. Baby Come Bak from 2019 Finals can be described as "DP on a tree". But even if you haven't seen a tree DP problem before, if you root the tree, the idea comes very naturally. Indeed, I didn't recognize it as a DP problem until *after* I rooted the tree, which gave the structure needed to solve the problem.

The first step in a greedy algorithm is often sorting the input. Rooting trees is the first step in tree DP. Flattening trees converts the least common ancestor problem to a range minimum query. These are ideas and algorithms that wouldn't be possible without the additional structure in the data. This has been talked about a lot in the previous modules, so I'll refer you to those instead.

5.4 Precompute

The idea of **precomputation** is typically described as computing all possible answers beforehand, and then simply outputting the answer later on. For example, the sieve of Eratosthenes precomputes which numbers are prime or not. But the term refers to much more than just precomputing the answer in your solution: the concept is much wider.

Advice 5.12. Know how to apply precomputation!

Story 5.13. One of my favorite problems is **BF Calculator:** https://codeforces.com/problemset/problem/784/G from the Codeforces 2017 April Fools round. The key to solving the problem is realizing we don't have to actually write a calculator. We can *precompute* the answer to the question beforehand, and instead output a program that prints the answer.

This isn't what we traditionally think of when we say "precomputation", but it's the right mindset. You want to do as much of the solving as possible beforehand.

This is why a common kind of precomputation is **offline precomputation**, where you solve all instances of the problem offline, copy the answers in the source code, and then print the answers when needed. For example, for Sugar Roll in the 2018 Team round, we can just use an array of strings to store the answer for each of the n, and since $1 \leq n \leq 45$, this easily fits in the source code.

The great thing about offline precomputation is that *we're not restricted to usual time limits*. That means we can run our code for much longer on our computer, even if it would normally be TLE. Another great thing is that we're not restricted to using C++: if we wanted to, we could use Python to compute the answer.

Problems like Sugar Roll, which are offline precomputation problems under the NOI system, are turned to **output only** problems in the IOI. This means we directly submit a text file containing only the answer to each problem, which is great because we don't have to worry about the source code limit for most cases. So although Tower Bloxx from 2019 Finals needed compression to fit the output in the source code, if this was the IOI, we wouldn't need to compress it.

But there are some problems where we want to do precomputation that *aren't* output only. In these kinds of problems, we *do* have to worry about whether it will fit the source code:

Story 5.14. From the first 2018 Practice Contest we have **Snowball and Snowballing:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/tslop-3. The final subtask boils down to computing the values of $n!$ modulo $p = 10^9 + 7$ for all $n \leq 10^9 + 6$. Precomputing all of it in the solution itself would be too slow. But if we precompute all of it offline, even if we compress it, it won't fit in the source code.

The trick, then, was to find a happy medium: somewhere in between computing everything and precomputing everything. Why don't we just precompute *some* of it, then, and use that to compute the rest? For example, if you know what $100!$ is modulo p, then you can find $101!$ modulo p by multiplying by 101, and so on.

So let's precompute $(10^6k)!$ modulo p for each $k = 0, 1, 2, \ldots, 1000$, and store these values in an array in the source code. To compute any $n!$ modulo p, we start from the largest $10^6k \leq n$, and then multiply by $10^6k + 1$, $10^k + 2$, and so on, until we reach $n!$. This takes 10^6 steps at worst. We can also adjust 10^6 lower for a faster runtime, but more numbers to include in the source code.

Sometimes, you really do have to fall back to compression. For example, in the first 2019 Practice Contest the problem **Banana Bonding Primes:** https://www.hackerrank.com/contests/noi-ph-2019-practice-1/challenges/banana-bond required compressing a sequence of 10^4 integers in the source code, each less than 10^6. There are some useful ideas when compressing sequences:

- If the precomputed values are increasing, only store differences.
- If it grows quadratically, store differences of differences.
- If they have a common factor of 24, divide by 24.

For the Banana Bonding Primes problem, we have to fall back to general compression methods, though. The typical trick is to store it in another base, like Base64 or Ascii85. It's relatively easy to write an encoder and a decoder for either one.

5.5 Dealing with unconventional tasks

You probably know what I mean when I say "unconventional tasks". I'm referring to tasks like Tower Bloxx in 2019 Finals, or Sugar Roll in 2018 Team. These may be output only problems, or ones that are NP-complete, or ones that involve finding heuristics, or ones that involve constructions.

The IOI really likes unconventional tasks. In 2018, Mechanical Dolls; in 2017, Nowruz; in 2016, Unscrambling Messy Bug; in 2015, Scales; in 2014, Game; in 2013, Art Class; in 2012, Pebbling Odometer; in 2011, Parrots; in 2010, Languages, and I could go on. I only listed what I thought was the most unconventional problem per year; many years had two or three of them.

I don't really think there's a single piece of advice I can give here. These problems can easily be timewasters, since they're unconventional, so **be careful to avoid tunnel vision** through good time management. Don't be afraid to use heuristics or randomness, or to run code on your computer. The solution to Nowruz, for example, uses a little randomness and a simple idea.

> **Story 5.15.** For Sugar Roll in 2018 Team, my generator "uses a little randomness and a simple idea". The solution for n was done by taking the best solution for $n-1$, and then randomly modifying it to get better and better solutions. I ran the code on my laptop for the whole contest, and I submitted it at the end to get around 47 points. Not bad for a solution that was very low-effort.

You can read some more about this in Steven Halim's article Expecting the Unexpected: https://ioinformatics.org/journal/INFOL113.pdf. And don't think we've exhausted all of the unconventional tasks. There's way more that we haven't seen yet: cryptography, compression, strict memory limits, finite automata, deconstructing algorithms, CTF-style problems…

Who knows what kind of things will happen?

Who knows what kind of problems will come out?

Who knows what kind of skills we'll need to solve them?

We don't. But that doesn't mean we can't prepare for it.

Acknowledgments

Thanks to Kevin for contributing stories and ideas.