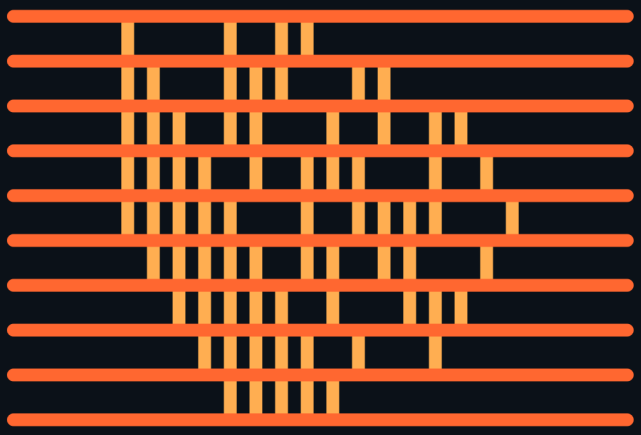
COMPETITIVE PROGRAMMING

START TO FINISH



Preface

For a Beginner Programmer

For those who are relatively new to programming, this book can serve as an excellent resource. There are plenty of exercises that a new programmer can practice on, and there are many different concepts they will be exposed to throughout this book.

That said, this would not do well as your only resource. The book expects that you either know one of C++, Java, or Python, or that you are able to learn what you need from other books or websites when you need to. While this is meant to be usable by someone relatively new to programming, it isn't meant to cover everything needed to learn programming from scratch. This serves well as a "second step" for new programmers by providing tasks and concepts beyond an introduction to programming (where many beginner programmers can find plenty of resources for starting out but little beyond that), without being unaccessible to someone with relatively little experience.

For a Beginner Competitive Programmer

If you're fairly comfortable with the languages you're programming in, but want to learn how to program competitively, this book is ideal. We focus on many simple applications of common data structures and algorithms, and offer many more creative applications that are intended to further develop a budding competitive programmer's skills.

That isn't to say that people interested in things other than competitive programming won't find value in this book. Technical interviews will commonly have similar tasks to what is presented in this book and involve the same methods and techniques to solve. Many data structures and algorithms are presented that are not as easy to find as normal within other books or resources, but have practical use that can be worthwhile to know. Often, concepts taught and utilized for competitive programming is applicable in other domains as well. You don't necessarily need to participate in competitive programming (though this book certainly recommends it) in order to benefit from learning about it.

For an Experienced Competitive Programmer

A competitive programmer who has their share of experience will be able to skip large portions of this book, since much of it is foundational knowledge that they would already be familiar with. That said, there are many less well-known algorithms and data structures that should be of interest, and many problems that can be good to solve.

For a Teacher

If you teach a course on data structures or algorithms, you will likely see a benefit to this book. There's a focus on the applications of data structures and algorithms, rather than

the approach many textbooks take where they focus on the design and implementation details of data structures and algorithms. Competitive programming tasks are an excellent form of learning and developing intuition about these data structures and algorithms, because they:

- are very well-defined tasks, backed with testing
- range from very standard to incredibly creative applications of data structures and algorithms
- are usually not implementation-heavy and won't bog down students with unnecessary details, they're generally meant to be quick to implement, as long as you're quick to figure out a valid solution.

For a Coach

For those that coach students and competitors in competitions, this book can provide a good list of topics to cover as well as various example problems to cover. The focus is, first and foremost, to provide a resource for potential competitors to become familiar with competitive programming and get up to speed on many of the necessary (or additional) concepts. For coaches, you can do anything from getting students to read this book directly, to covering the exact topics and problems here, to using this as a guideline and suggestions as you prepare your own training material.

Acknowledgements

This would not be possible without a base of existing literature to use as reference and as building blocks. The authors specifically thank the following:

Algorithm References

- *The Art of Computer Programming* - Donald Knuth
- *Introduction to Algorithms* - Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein (CLRS)

Competitive Programming Textbooks

- *Competitive Programming 3* - Steven Halim, Felix Halim
- *Guide to Competitive Programming* - Antii Laaksonen

Additional References

- *Optimizing Software in C++* - Agner Fog
- *Matters Computational* - Jorg Arndt

Notation

A brief description of some of the mathematical notation used in this book might be useful, because not everyone reading this may be overly familiar with some of them:

- when describing a list or set, we will usually denote it as $\{1, 2, 3, 4, 5\}$. Multiple sets or lists have the notation $\{1, 2\}, \{3, 4\}, \{5, 6\}$.
- inclusive ranges are given in the form of $[x, y]$ that represents every value beginning with x (including x) and ending with y (including y). For example, the range $[1, 5]$ includes the values $\{1, 2, 3, 4, 5\}$.
- ranges may also exclude a value, by using (or) instead of [and]. The one most often encountered would be $[x, y)$, where we have a range from x (including x) to y (not including y). As a result, $[1, 5)$ is the values $\{1, 2, 3, 4\}$.
- modular arithmetic follows code-like conventions, where we use $\%m$ to denote numbers $mod\ m$. For example, adding 2 and 3 under modulo 7 would look like $(2 + 3)\%7 = 5$ rather than the traditional $2 + 3 \equiv 5 \mod 7$.

There is some specific notation we will use in this book as well, that we will cover here.

- individual elements of an array are addressed, 0-indexed, as $arr[i]$ where arr is the array and i is the index, or with constants such as $arr[0]$.
- subarrays use the notion $arr[i : j]$ for elements in the range $[i, j)$, similar to how Python handles subarrays.
- strings are surrounded by double-quotes like `"string"`.
- individual characters are surrounded by single-quotes like `'c'`. Characters that use escape codes such as newlines are given as `'\n'`.

Contents

1 Introduction

1.1 Recreational Programming

Programming can be very productive – making something with practical value, that people can use in their life to simplify tasks. Programming can also be done for the sake of programming itself, and only because it's fun. The latter is called "recreational programming", where the act of programming is done for fun rather than to create something useful.

There is a wide variety of forms of programming that people do recreationally. Many of them involve playing with how the sourcecode of a program appears, such as:

- Codegolf – solving problems in as few characters of sourcecode as possible.
- Polyglot Programming – writing programs where the same sourcecode can successfully run in multiple different languages (either by giving the same output, or intentionally giving different outputs depending on the language).
- Quine Programming – programs that output their own sourcecode.
- Obfuscation – programs made to be as difficult to read as possible, and are very hard to understand what they do by looking at the sourcecode (see IOCCC).
- Competitive Programming - solving well-defined problems under specific constraints (runtime, memory) often in a tournament setting with an emphasis on solving quickly.

This book focuses on competitive programming. It should stand out as the one most similar to raw problem solving – in fact the only things it adds on top of basic problem solving is well-defined constraints. This makes it very useful in general programming, which is also essentially problem solving.

To some, competitive programming is very focused on the *competitive* aspect. They will practice for the sake of getting better results than others. For others, the *programming* part of competitive programming is what they focus on. They will practice for the sake of improving their own abilities. For many, it will be some mixture of both. In any case, both groups of people should find competitive programming fun in general.

1.2 Competitive Programming

The concept behind competitive programming is simple: you are given a problem with specific inputs and required outputs, and you have to submit code that solves it under well-defined constraints. These constraints are almost always in the form of a runtime limit and a memory limit. These are tested by an external "judge" that verifies your program works on hidden testcases (so that you can't hardcode the solution for every testcase) and that it works under the constraints (the runtime would depend on the computer that the judge is running on, but it should be relatively consistent for everyone submitting so no one gets an advantage or disadvantage).

There are several skills tested in competitive programming. Often, they rely on knowledge

and creative applications of algorithms. Other times the solution does not need any particularly advanced techniques or prior knowledge, but tests your problem solving skills in an interesting way. Many times, it's easy to come up with a valid solution that, given enough time and memory, will get you the proper answer – but the time constraints mean you need to develop a much more efficient approach to the problem. Some have a strong basis in math, while others are solely determined by computer science, and many combine mathematical and computational aspects together.

All of the above problem types take time to improve at. Sometimes they will be problems you are familiar with in your education and you can comfortably solve them, sometimes they will be advanced applications of techniques you've learned that are significantly more difficult than what you've likely encountered before, and sometimes they will involve techniques most people would never hear of outside of competitive programming.

1.3 Value of Competitive Programming

Competitive programming is very helpful to a programmer. For students, many homework tasks will be of a similar style to competitive programming tasks, and being skilled with competitive programming will give you a huge advantage in this sort of work. For those interested in academia, competitive programming is a great way to apply various algorithmic or computational techniques and to explore a wide variety of interesting problems to research. And for those looking to get hired by a company, not only does competitive programming experience look good on a resume, but many technical interviews involving algorithmic problems are essentially competitive programming problems. Someone with experience in competitive programming will have significant experience in all of the above that can be difficult to practice otherwise.

Not to mention that practicing competitive programming is a great way to practice programming in general. When learning a new language, it is useful to solve many smaller problems with it to get more comfortable. Fortunately, competitive programming practice involves many small problems to solve. When learning a new technique or algorithm, it's useful to find many problems that require it of varying difficulty. Often, competitive programming practice will list out problems in terms of topic and difficulty for this reason.

All that said, it's important to preface this book by saying that competitive programming won't help all your skills programming. Competitive programming tasks are a niche within programming, and as a result have special considerations. The biggest being that code maintainability doesn't matter at all for competitive programming, and as a consequence clean code practices can be pretty much entirely ignored (and often are, for the sake of typing faster) as long as you're still able to debug your solution in case of an error. Many practices recommended in competitive programming are actively discouraged in general software development, and vice versa. If you are looking to get better at writing readable and maintainable code, you have to be careful not to let competitive programming teach you bad habits.

Usually you don't care about things like memory management much either – this is both a blessing and a cause for concern in a language like C++. It is generally much easier to learn and use C++ if you never deal with manual memory management, as long as

the solution runs through the testcases without passing the memory limit. But it is also missing out on a major part of the language that you will need if you work on larger projects. Issues from badly management memory are a relatively rare problem in competitive programming (they happen all the time, but if your solution still gets accepted then it doesn't matter) but are a very frequent problem in general software development.

Don't let that dissuade you though. These sorts of things develop through practicing or working with larger projects, that you absolutely can do alongside competitive programming.

1.4 Programming Contests

There are a handful of prominent contests that regularly get held:

- ICPC (International Collegiate Programming Contest) is the main programming contest for university students.
- IOI (International Olympiad in Informatics) is the main programming contest for highschool students.

1.5 Programming Sites

Outside of contests themselves, there are many sites that can be used for practice. A non-exhaustive list of several different competitive programming sites are:

- Kattis
- Codeforces
- Atcoder
- Codechef
- UVa

1.6 Purpose of This Book

The key purpose of this book is to familiarize students with many topics found in competitive programming.

There is a focus on beginner-level problems in this book that are ideal for learning a new programming language and growing proficiency in it. While this book doesn't detail the basics of a language or describe how to initially learn that language very much, it should serve as an excellent companion in practicing with that language and comparing it to other languages.

Specifically, the languages used in this book are Python, Java, and C++.

1.7 Tips

Before we begin really looking at competitive programming, it's important to go over some general tips.

It's never too late to begin competitive programming. You can begin practicing competitive programming at any time, and that there are others who began before you should not prevent you from doing so. Someone who begins practicing later will be far better than someone who never practices at all.

It's never too early to begin competitive programming either. Intentionally waiting until later to start programming competitively won't help you. A lot of people will put off practicing competitive programming while they develop their skills elsewhere thinking that it will ultimately help them more to have a stronger foundation, but in practice it results in far less time improving at competitive programming.

In other words, there's no better time to start than right now. There's a saying that goes "the best time to plant a tree was 20 years ago. The second best time is now." This applies to lots of things, but is true for competitive programming as well. As nice as it would be to have begun years ago, there's no sense procrastinating it now.

Getting involved is a great way to continue your competitive programming journey if you're involved with a community that focuses on it. It can be hard to practice more when you're doing it alone, and it is far more productive to practice alongside others who are improving with you.

Practicing consistently is how you best grow your skills. Often people are extremely motivated to practice, go through lots of problems in a weekend or so, then get burnt out and don't touch another competitive programming problem for months. This is not a good way to practice – it is far better to practice consistently and solve a maintainable amount of problems regularly. Enough so that you learn new things often and can develop your skills, but not so much that you get mentally exhausted and can't keep it up for long.

Practice problems slightly above your level rather than only practicing problems you find very easy (where you don't get much out of it) or problems that are very hard and far beyond your current level (where you don't get much out of it). You'll practice best doing problems outside of your comfort zone, but still doable for you. Solving easy problems can increase your typing speed and attempting hard problems can introduce you to new techniques, but in general you will develop your skills as a whole by practicing problems of a proper difficulty.

Improve weak areas so that you are more capable of solving problems of a certain type that you recognize you're not as strong at. It is far easier and faster to improve in a particular skill if you focus it specifically. If you're practicing a specific topic, you should prioritize the ones you're less comfortable with rather than the ones you're already proficient in. That isn't to say you should only work on specific topics – trying whatever problems of a slightly higher difficulty than you can comfortably solve is good, but this tip is for if you intend to practice something specific rather than practice your general problem solving skills.

Don't spend all your time thinking about practice. A common pitfall that many new competitive programmers get into is spending a lot of their time trying to figure out *how* to practice more optimally. The truth is, the most optimal way to practice is to not worry about it and just solve more problems. There are no get-rich-quick-schemes for getting good at competitive programming, and strong competitive programmers got where they were by solving lots of problems.

1.8 Example Problems

We've discussed competitive programming a fair bit by this point, but haven't seen any actual problems yet. What does a competitive programming task look like? We'll cover a few here.

After listing out some problems, we will go over the solutions for them and explain the problem itself. You've been warned in case you'd like to solve these problems on your own first (it is recommended).

Hello World!

When we begin programming, one of the first tasks we'll do is write out "Hello World!" or some variation. The same applies here – likely the easiest competitive programming task you'll encounter is a simple "Hello World!" program. That's what we want in this program.

Input

There is no Input

Output

Output is a single string, Hello World!

Constraints

Time Limit: 1 second
Memory Limit: 1024 mb

Samples

`Input:`

```
No Input
```

`Output:`

```
Hello World!
```

In the above problem we can notice several sections. We have the title of the problem, **Hello World!**. Following this is a simple description. In this problem we have a pretty simple description.

After that, we have our input format. Hello world is a relatively unique problem in that it has no input – these types of problems are rare. We do still have output, which we see in the next section, asking us to output "Hello World!". We also have our constraints section, where we list out the time limit and memory limit of our solutions. These constraints shouldn't be an issue for this problem specifically, but will come up in other problems.

We also have a sample input and output. When the program runs, it should produce the given output. These are basic test cases that are readily available to you.

Apples and Oranges

Alice has many apples, and Owen has many oranges. The two of them usually get along, as long as Alice doesn't accidentally eat any oranges and Owen doesn't accidentally eat any apples.

As a friendly competition, Alice suggests that they compare how many of each fruit they have. Owen agrees, sure that he has more oranges than Alice has apples. Alice believes that she has more apples though. In true competition fashion, they get a judge to count so that they can be sure.

Unfortunately, while the judge knows how to count, they don't know how to tell what number is bigger than the other. It's up to you to help finish the contest and properly declare the winner.

Input

You are given two integers, a and b where $0 \leq a, b \leq 10^9$. a represents the number of apples that Alice has, and b represents the number of oranges that Owen has.

Output

If Alice wins, print out "Alice". If Owen wins, print out "Owen". If it's a tie and neither wins, print out "Tie".

Constraints

Time Limit: 1 second
Memory Limit: 1024 mb

Samples

`Input:`

```
50 1
```

`Output:`

```
Alice
```

Input:

```
0 1000000000
```

Output:

```
Owen
```

Input:

```
5 5
```

Output:

```
Tie
```

This problem is slightly more involved than **Hello World!**. The actual problem itself isn't complicated, but there is now something inside the input section that we have to consider. For someone new to competitive programming or someone unfamiliar with math, this can be a bit intimidating. It simply means that you have two input numbers whose values are between 0 and 10^9 (inclusive). It then describes what these numbers actually mean, where a and b are used in the rest of the problem.

The output section is also more involved than before. Our output depends on what we got for inputs. It is fairly easy to determine, but we do have to deal with 3 different cases for our output.

The samples section goes over a bit more detail too. Specifically, there are multiple samples to look at. For each given input, the program should output what is seen in the samples.

Factorial Digit Product

The factorial of a number n is denoted as $n!$. The math behind a factorial is $n! = 1 * 2 * 3 * ... * (n-1) * n$. In other words, the factorial of n is the product of every number between 1 and n inclusive. The exception to this rule is when $n = 0$, where $0! = 1$. This is the same as 1! where $1! = 1$.

What we want to do is find the product of each individual digit of a factorial. For example, with 4! we get $1 * 2 * 3 * 4 = 24$, and the product of the digits of 24 is $2 * 4 = 8$.

Input

Input first consists of a number t where $1 <= t <= 100000$. t denotes how many test cases there are.

For each test case there is a number n where $0 <= n <= 10^9$.

Output

For each test case, print out the factorial digit product of n.

Constraints

Time Limit: 1 second
Memory Limit: 1024 mb

Samples

Input:

```
5
0
1
2
3
4
```

Output:

```
1
1
2
6
8
```

This problem is relatively difficult compared to the others. At first glance, we can notice that we have to handle multiple tests within a single testcase. This is not much more logic to handle, but does mean we have to be careful not to have too slow of a solution.

Without giving away the answer, runtime is likely something to be concerned for with this problem. The input section mentions that we can have up to 100000 tests, where each number can be as large as 10^9. We can assume that one of the hidden testcases will be something along the lines of requesting the factorial digit sum of 10^9 100000 times. If we calculate the factorial naively, this will not be able to pass runtime constraints (not to mention that $10^9!$ is a huge number and multiplication slows down for larger numbers, and would require a custom class in C++ since it can't handle arbitrarily large numbers natively) – we have to think of a smarter solution.

The solutions to **Hello World!** look like:

```cpp
#include <iostream>

using namespace std;

int main() {
    cout << "Hello World!";
}
```

```java
class Driver {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

```
print('Hello World!')
```

There is not much to say for the solution in general – they look like you'd find from any other resource.

For C++ specifically, you can notice that we use `using namespace std;`. This just simplifies some of the rest of the program – we can use `cout` instead of `std::cout`. This applies to nearly everything we obtain via `#include`. While there are arguments against using it in larger projects (namely, when you are dealing with many different namespaces that may or may not have conflicting function names) those same arguments don't apply too much to competitive programming (where you almost exclusively use the `std` namespace and programs only consist of a relatively small amount of functions) so you will find most people use `using namespace std;` when programming competitively.

It's not uncommon to see people also use `#include <bits/stdc++.h>` as our include statement instead of `#include <iostream>`. That is a technique specific to GCC compilers that imports everything in the standard library, so that you don't need to have multiple `#include`s for each different library used. We avoid doing so in this book, both because it makes the origin of certain standard library functions or classes more clear to not, and also to maintain compatibility with other compilers other than GCC (which does not apply when competitively programming, it only needs to work for whichever compiler you're using).

The solutions to **Apples and Oranges** look like:

```
#include <iostream>

using namespace std;

int main() {
    // read in our input variables
    int a, b;
    cin >> a >> b;

    // handle each individual case
    if (a > b) {
        cout << "Alice";
    }
    else if (a < b) {
        cout << "Owen";
    }
    else {
        cout << "Tie";
    }
}
```

```python
# read in integers on same line
a, b = [int(x) for x in input().split()]

# print the right answer
if (a > b):
    print('Alice')
elif (a < b):
    print('Owen')
else:
    print('Tie')
```

The solutions to **Factorial Digit Product** look like:

```cpp
#include <iostream>

using namespace std;

int main() {
    // get the number of testcases
    int testcases;
    cin >> testcases;

    // for every testcase
    for (int i = 0; i < testcases; i++) {
        int n;
        cin >> n;

        // handle all possibilities below 5
        // any number above 4 will always be 0
        if (n == 0 || n == 1) {
            cout << "1\n";
        }
        else if (n == 2) {
            cout << "2\n";
        }
        else if (n == 3) {
            cout << "6\n";
        }
        else if (n == 4) {
            cout << "8\n";
        }
        else {
            cout << "0\n";
        }
    }
}
```

You will notice that the solution is quite simple, and not necessarily immediately obvious. The problem description gives no indication that numbers above 4 are trivial to calculate.

The fact that 5! = 120 whose digit product is 0, and that increasing the factorial and multiplying it more will always leave a 0 as the last digit (6! = 720, 7! = 5040, 8! = 40320, 9! = 362880, 10! = 3628800 and so on) which means the digit product is always 0 is an observation that needs to be made by the problem solver.

Because of this, we just have to have special cases for each number from 0 to 4 that we can figure out on paper easily (in fact the sample data already gives us the answer to all of these), and then print out 0 if it's any other number.

—

This book will recommend other problems as well, related to the techniques explored in the chapter or subsection.

1. `https://open.kattis.com/problems/hello`
2. `https://open.kattis.com/problems/quadrant`
3. `https://open.kattis.com/problems/timeloop`
4. `https://open.kattis.com/problems/fizzbuzz`
5. `https://open.kattis.com/problems/lastfactorialdigit`

2 Programming Environment

2.1 C++

2.1.1 Setup

There are a few different possible compilers that C++ can use. While options like the clang compiler are occasionally used, the most common compiler in competitive programming is GCC. Because of this, we will focus primarily on setting up the GCC compiler. The C++ compiler specifically is the `g++` command, most often.

On a Linux system the installation varies between different distros, but is generally relatively easy for someone used to the operating system. On Ubuntu for example, you should be able to run `sudo apt install build-essential` which will install various necessary packages, including `g++`.

On Mac OSX...

On windows, the setup requires `http://mingw.org/` in order to get the GCC. You can download the installation manager that will simplify the process.

To verify that the install worked, you should be able to run `g++ -version` to see if it's a missing command or if it tells you the version number.

2.1.2 Strengths and Weaknesses

The most clear benefit of C++ is that it's among the fastest languages there are, and is usually the fastest language available in a contest setting. While solutions in other languages will usually be able to pass a problem's time constraints (assuming the algorithm is the intended solution, and potentially with some optimization) C++ is the only language that is guaranteed to be capable of passing time constraints.

While not an actual feature of the language itself, C++ is also the most common language in competitive programming in general. As such, you will find most resources use C++.

Unfortunately, there are some notable issues in C++'s standard library. For example, the standard library offers no way to represent arbitrarily large integers, and you would have to implement a class for this purpose on your own.

2.2 Java

2.2.1 Setup

Under Construction

2.2.2 Strengths and Weaknesses

Compared to the other languages listed here, Java's strength is that it enforces much more safeguards at compile-time than either C++ (that often only warns about things if you enable those warnings in the compiler, and they are not necessarily the most clear warnings) or python (that doesn't even have a proper compile step, and outside or erroneous syntax will throw its errors at runtime).

Java also comes with some very useful bits inside its standard library. An example would be `BigInteger` and `BigDecimal` that...

Unfortunately, it does come with a fair amount of boilerplate, so solutions in Java tend to be relatively verbose compared to other languages.

While Java's JIT is very impressive and its programs are quite fast once it's warmed up, this startup time is not an insignificant concern.

2.3 Python

For the purposes of this book, we assume Python 3 is used. Python 2 can offer better speeds on some platforms due to more optimized interpreters and compilers, but in general we aren't too concerned about that as far as this book is concerned. If you would prefer to use Python 2, most of the code examples given should be fairly simple to translate.

2.3.1 Setup

Under Construction

2.3.2 Strengths and Weaknesses

Python's main strength from a competitive programming standpoint is that it has very little boilerplate code, and in general it is very short. Many of its abstractions can lead to very short code, and in fact lots of relatively simple (at least, simple to code) problems can be solved in a single line.

The main weakness of Python is that it's fairly slow, and is certainly slower than C++ or Java. Some contests will specifically point out that they cannot guarantee Python is able to pass the time constraints (though this only means the contest testers didn't write a solution in Python, not that they actually know it can't be done).

2.4 Other

While we won't go in-depth here describing them, there are a few other languages that are important to mention. You can explore them but this book isn't going to describe

setting them up or cover anything about them in any sort of detail, in favor of the other three languages C++, Java, and Python.

2.4.1 C

C has some historical note as one of the languages offered in many competitions.

In practice, C++ is generally preferred – there isn't any major speed differences between the two, and for the most part C++ is a superset of C with many more useful utilities and parts to its standard library. While there is valid reasoning why you would prefer C for some general software development (usually relating to what platforms it can run on or the team that you develop with), none applies for competitive programming.

2.4.2 Pascal

Like C, Pascal has some historical note as well. It was a fairly popular language in IOI competitions and was one of the few allowed, though in recent years it was removed from the list of valid languages.

2.4.3 Kotlin

Kotlin has started to see some use and is being offered in some contests recently.

2.5 Text Editors

Most text editors are comparable to each other in terms of quality and features, and usage is largely based on preference. That said, there are some recommendations to be made.

In Windows, Notepad++ is a common recommendation. Gedit and Geany are often options available on Linux systems (and have the benefit that they are usually available on contest computers). Sublime, Atom, Visual Studio Code, or many others are viable options as well on multiple platforms.

IDEs (integrated development environments) are very nice and convenient, but if you intend on entering competitions like the ICPC where IDEs are usually not available, it is recommended to avoid (or at least be comfortable without) using an IDE. They are also often relatively bulky, with large menus and slowdowns due to features that are only really relevant in larger-scale projects.

Usually, text editors like vim and emacs are offered in contest settings. However, it's important to consider that you would not have much time to set up your editor to your preferences, and a complicated setup would be hard to replicate or take some time to prepare. For that reason, if you do decide to use an editor like vim or emacs, it's worthwhile to practice with them either being completely default, or with very little customization that you can quickly apply in a contest.

3 Algorithm Analysis

3.1 Big-O Notation

Under Construction

3.2 Runtime Complexity

The main application of Big-O notation is to describe runtimes of algorithms.

While we will say things like "$O(n)$ algorithms will take n operations" this is not entirely true, and is a generalization. If you were to calculate out how many operations an algorithm takes, Big-O only concerns itself with the largest term. For example, if you had an algorithm that took $n^2 + 100n$ operations, you would have $O(n^2)$ even though the $100n$ is significant. $O(n^2)$ could be $999n^2$ operations or it could be $0.1n^2 + 10^9n + 10^{100}$, since Big-O is only a measure of the highest growing term without any coefficients.

There are many frequent examples of algorithm complexities:

- $O(1)$ means that an algorithm is done in constant time, no matter how big the input is. A trivial example would be "for a list of n elements, print the first element" because the algorithm is not concerned with how many elements there are after the first one.
- $O(logn)$ means that an algorithm does take longer to execute with larger inputs, but it doesn't increase very significantly. In log base 2, doubling the input size will only make it take one more unit of time – if $log_2(n) = 1$ then $log_2(2n) = 2$, $log_2(4n) = 3$, $log_2(8n) = 4$, and so on. It is very uncommon that a $O(logn)$ algorithm will not pass a problem's time limits.
- $O(n)$ is an extremely common runtime, where you have to go through every element of the input.
- $O(nlogn)$ is a common runtime when we perform a $O(logn)$ algorithm on every element of the input.
- $O(n^2)$ is a very common runtime that can be best described as "for every input, go through every input."
- $O(n^3)$ is much like $O(n^2)$ except that it is "for every pair of input, go through every input" where every pair of input is $O(n^2)$ itself.

To elaborate a bit on what these actually mean, we should look at some example programs. For a $O(1)$ algorithm, we can look at:

```
# read in n
n = int(input())

# O(1) algorithm
result = 0
if (n == 99):
    result = 1

# output the result from our algorithm
print(result)
```

Notice that it doesn't matter if n is extremely large or extremely small, the program will take the same amount of time to run in any case. Compare this to a $O(n)$ algorithm:

```
# read in n
n = int(input())

# O(n) algorithm
result = 0;
for i in range(n):
    result = result + i

# output the result from our algorithm
print(result)
```

Where a small n will run very fast, but a large n might take a long time to run. This can then be compared to a $O(n^2)$ algorithm:

```
# read in n
n = int(input())

# O(n) algorithm
result = 0;
for i in range(n):
    for j in range(n):
        result = result + i * j

# output the result from our algorithm
print(result)
```

If a $O(n)$ algorithm took a long time with a large n, this algorithm will take an extremely long time. Specifically, if n is $10,000$, then a $O(n)$ algorithm will take $10,000$ operations while a $O(n^2)$ algorithm will take $10,000^2 = 100,000,000$ operations. Fortunately, $10,000$ is not a huge amount of operations for a computer, and usually (at least for most online judges) it's around 10^8 that a $O(n)$ algorithm is still relatively safe to use for a time limit of 1 second. When n is 10^8 though, a $O(n^2)$ algorithm would perform 10^{16} operations which is far beyond 1 second of runtime.

You might be able to notice a pattern, where every nested for-loop from 0 to n multiplies our Big-O by n as well. Indeed, if we had 3 nested for-loops we would have $O(n^3)$. 4 nested for-loops from $[0, n)$ would be $O(n^4)$. And so on.

There are several less common time complexities as well, but some of the uncommon (but far from unheardof) runtimes would be:

- $O(\sqrt{n})$
- $O(nloglogn)$
- $O(2^n)$
- $O(n!)$
- $O(n^n)$

As well, there are sometimes time complexities that rely on multiple variables. For example, you will see:

- $O(v + e)$
- $O(v * e)$

3.3 Space Complexity

Space complexity is comparable to runtime complexity. But instead of measuring how much time an algorithm takes to run, it measures how much additional data the algorithm needs to store.

If you have a problem with an input of size n, and you need to store elements into an array of size n as well, we are talking about $O(n)$ space complexity. If you have a 2 dimensional array where both dimensions are size n (in other words, we're storing n^2 individual elements), our space complexity is $O(n^2)$. If we never have to store any additional data (or more accurately, the amount of data we store doesn't change depending on the problem size) we're dealing with $O(1)$.

In general, all the same complexities that exist with runtime complexities are possible, but we are less likely to run into the less common ones, and likewise we are less likely to concern ourselves with the memory complexity (more often than not, runtime constraints are what limit you in a problem, memory constraints are not uncommon to be a problem but are less often than runtime).

3.4 Precision

Frequently, we will need to deal with precision in solving problems. Specifically, often we have problems where larger inputs do not fit into smaller types (for integers), or repeated arithmetic makes us require a certain level of precision that smaller types don't offer (for floats).

3.4.1 Integer Types

- **16-bit signed** (`short` in C++ and Java) supports $[-2^{15}, 2^{15})$ and can hold integers up to over 10^4.
- **16-bit unsigned** (`unsigned short` in C++) supports $[0, 2^{16})$ and can hold integers up to over 10^4.
- **32-bit signed** (`int` in C++ and Java) supports $[-2^{31}, 2^{31})$ and can hold integers up to over 10^9.
- **32-bit unsigned** (`unsigned int` in C++) supports $[0, 2^{32})$ and can hold integers up to over 10^9.
- **64-bit signed** (`long long` in C++ and just `long` Java) supports $[-2^{63}, 2^{63})$ and can hold integers up to over 10^{18}.
- **64-bit unsigned** (`unsigned long long` in C++) supports $[0, 2^{64})$ and can hold integers up to over 10^{19}.
- **128-bit signed** (`__int128` in GCC C++) supports $[-2^{127}, 2^{127})$ and can hold integers up to over 10^{38}.
- **128-bit unsigned** (`unsigned __int128` in GCC C++) supports $[0, 2^{128})$ and can hold integers up to over 10^{38}.
- Python's **int** will internally convert into an arbitrary-precision data type when it needs to in order to store integers of any size (restricted only by memory).
- Java's **BigInteger** is much like Python's **int**. It does have its own features, such as supporting a method `isProbablePrime(10)` that can tell you (to a likelihood of $1 - \frac{1}{1^{10}}$) whether the number is prime or not (though it will not give false negatives, if it returns `false` is is definitely not prime, but returning `true` only means that it is probably prime).

It's notable to mention that C++ does not have its own arbitrary-precision integer type. There are popular third-party libraries for this, but those aren't normally available in contests or on online judges. It's not uncommon for competitive programmers who use C++ to write their own arbitrary-precision integer type (you can find examples in many teams' ICPC notebooks) but it is also common for problems to not require arbitrary precision types.

3.4.2 Floating Point Types

- **32-bit floating point** (`float` in C++ and Java) supports up to $\pm 3.40282 * 10^{38}$.
- **64-bit floating point** (`double` in C++ and Java or `float` in Python, the normal floating point representation) supports up to $\pm 1.79769 * 10^{308}$.
- **80-bit floating point** (`long double` in C++) supports up to $1.18973 * 10^{4932}$.
- **128-bit floating point** (`__float128` in GCC C++) supports up to $\pm 1.18973 * 10^{4932}$ but with greater precision than **80-bit** floats.
- Python's **decimal** is a library class that allows you to set precision as you need, via setting `getcontext().prec` (used as `getcontext().prec = 50` for 50 significant places).
- Java's **BigDecimal** is like Python's **decimal** in that you can set your own precision, by calling `.setScale(50)` on the **BigDecimal** object.

Like with arbitrary-precision integers, C++ doesn't offer any standard library solution for arbitrary-precision floats. This is generally less of an issue because problems that can't be solved with `long double` or `__float128` and actually require arbitrary-precision are quite rare (unlike integers, it is extremely rare to find arbitrary-precision floats in ICPC team notebooks).

Floating point precision mostly comes into play when we're dealing with huge amounts of repeated operations. To demonstrate, we can simulate the difference between a few of these types (in C++) with something like:

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

// what type to use
using precision_type = long double;

int main() {
    // set up our variables
    precision_type val = 1e10, max = 1e3, step = 1e-4, start = 1.0;

    // repeatedly do nothing
    // if precision were perfect this would not change `val`
    for (precision_type i = start; i <= max; i += step) {
        val *= i;
        val += i;
        val /= i;
        val -= 1;
    }

    // tell us what `val` is after all those iterations
    cout << fixed << setprecision(10) << val << '\n';
}
```

Which, when we change what floating point type we're using, gets us:

- `float` is 10020363264.0000000000 which is $\sim$ 20363264 off the correct answer.
- `double` is 9999999999.8307151794 which is $\sim$ 0.1692848206 off the correct answer.
- `long double` is 9999999999.9999834169 which is $\sim$ 0.0000165831 off the correct answer.
- `__float128` is 10000000000.0000000000 which is accurate to the precision we're using.

In general, `float` isn't very accurate and has trouble handling repeated arithmetic like this – in a problem that requires lots of repeated arithmetic a simple `float` might not be sufficient. `double` is a lot better and has much greater precision, but is still not perfect and may occasionally have precision issues. `long double` is better and reduces the imprecision from doubles even further, but it still is not entirely precise with repeated

arithmetic.

`__float128` gives us the best precision out of all of them, but (as mentioned) is not guaranteed to work in all compilers since it's a GCC extension. In addition, `cout << val` doesn't work with `__float128` and it is necessary to cast it to another type, such as `cout << (double)val`. Because the issue is the precision when doing lots of arithmetic that eventually results in significant precision losses, it's fine to cast it to a smaller type after performing that repeated arithmetic.

In terms of speed, `float` is the fastest, `double` is slightly slower (expect a slowdown of around 20-30%), `long double` is slightly slower than that, and then `__float128` can be significantly slower (quick benchmarks indicate that $\sim$ 15 times slower than `double` is reasonable to expect).

Some considerations for Java and Python are that you have less options for your types than C++, but one of the options is an arbitrary precision type that can be as large as need be (that C++'s standard library doesn't offer). Note that arbitrary precision data types are relatively slow, so if possible it is recommended to use the primitive types.

4 Problem Analysis

4.1 IO Formats

For the vast majority of problems, the IO is some variant on "here is the inputs for one or more testcases, for each testcase print the proper output".

At its simplest form, it will be the inputs for a single testcase.

Input:

```
input
```

Output:

```
output
```

Often, there will be multiple testcases, given by a number t.

Input:

```
3
input 1
input 2
input 3
```

Output:

```
output 1
output 2
output 3
```

Other times, multiple testcases continue until you read a 0. For example:

Input:

```
input 1
input 2
input 3
0
```

Output:

```
output 1
output 2
output 3
```

Or instead will be terminated by no more input at all:

Input:

```
input 1
input 2
input 3
```

Output:

```
output 1
output 2
output 3
```

4.1.1 Interactive IO

There are some problems where you don't have all the input initially, and instead your solution communicates back and forth with the judge. These problems are called 'interactive problems' and are relatively uncommon but can throw someone off when they haven't encountered one before.

Consider a simple interactive problem, where we have to guess a number from 1 to 10. When we output our guess, we will get back a response that's either "correct" or "wrong".

Then, if it's "wrong", we guess again. Usually there will be some limit on the number of guesses (often called queries) we can make, and in this problem we can limit the total number of guesses to be 10 (we should never need more than that, even for a very naive approach).

A naive solution can be:

```cpp
#include <iostream>

using namespace std;

int main() {
    int guess = 1;

    // make initial guess
    // we use `endl` to ensure we flush output
    cout << guess << endl;

    // if we're wrong, we guess again
    string s = input;
    cin >> s;
    while (input == "wrong") {
        // make new guess
        guess++;
        cout << guess << endl;

        // read new response
        cin >> input;
    }
}
```

Note that we use `std::endl` rather than `'\n'` for our endlines. This is because `'\n'` doesn't immediately print our outputs and will instead buffer them to print later (and is faster for that reason) which doesn't work in interactive problems.

This problem is not very representative of actual interactive problems, because it's so basic and you don't need to do any interesting logic based off of the interactivity (it's a simple loop . This is meant more as a "hello world" problem for interactive problems.

4.2 Problem Solving

Problem solving is not a well defined, simple to follow process. There is no algorithm to follow that will solve problems with success. Problem solving involves creativity and intuition. Competitive programming problems especially, where they are carefully designed with the intention of being interesting problems that usually require a creative solution.

That said, there is a general process that can make it easier to figure out a problem. This is not strictly defined, and serves as a rough method that can help.

1. Read the problem. It can sometimes be easier to read the inputs, outputs, and samples first and then going back to read the full description so you can understand the description in context of the exact problem.
2. Solve the sample cases on your own. Don't dwell on this if one of the samples is especially gnarly, but try to figure out how the problem is actually solved. As you go, try to keep note of any assumptions you make that are used in solving it, or could be used to make it easier to solve.
3. Identify the problem type. Is the problem a brute-force problem? Maybe it's a simulation problem, where you have to implement what it asks for and find the solution that way. Perhaps the solution requires a relatively advanced technique, and that technique is the problem type. Maybe there is no technique, and the solution requires some creative thinking without being tied to any common type (an "ad hoc" problem).
   This is usually the hardest step, and the only way to get better at it is through practicing those problem types. The more problems of a specific type you solve, the better you will be at identifying it in a new problem you haven't seen before.
4. Consider counter-cases, if possible. The assumptions you made when solving the sample problems is what you used to determine the problem type. Do the samples handle all relevant cases, or are there cases you have to consider that aeen't given? Are your assumptions correct, or is there an error with your logic that means it requires an entirely different technique?

These steps are usually done, to some degree, in parallel. You'll usually be thinking of the problem type when reading the problem and solving the samples, and you'll usually be thinking of potential counter-cases when determining the problem type. It is very common that you will redo these steps as well, because of things you didn't consider initially or figured out later. In practice, these are more things to keep in mind than a strict step-by-step process.

It's recommended to experiment with this on your own, both to familiarize yourself with it in practice and to tune it for yourself. Are there other steps you perform better with? Perhaps you have your own order to approaching things. Often, you might approach an easy problem completely differently than a complex one, because you can recognize the necessary techniques fairly quickly.

4.3 Reading Comprehension

Under Construction

4.4 Debugging

Under Construction

Some example problems that involve techniques covered in this chapter (specifically, var-

ious IO formats, interactive IO, and reading comprehension).

Interactive:

1. `https://open.kattis.com/problems/guess`
2. `https://open.kattis.com/problems/askmarilyn` (harder, use good randomization)

Reading Comprehension:

1. `https://open.kattis.com/problems/carrots`

5 Foundational Data Structures

5.1 Arrays

Arrays are the most basic data structure in most programming languages. It is very simple in its functionality: instead of storing a single value like a regular variable does, it stores some amount of values.

5.1.1 Dynamic Arrays

Arrays don't have to be a fixed size – dynamic arrays allow you to dynamically resize the array when you need more space to put new elements.

In C++ we have the `std::vector` class that works as our dynamic array. In Java, it's `ArrayList`. And in Python, lists are already dynamic arrays.

5.2 Sets

In mathematics, a set is a collection of distinct elements. Pratically speaking, this differs from a traditional array in that every element is unique (in practice, trying to insert a duplicate element will not insert it) and that there is no guarantee of any specific order (in practice, the order depends on implementation).

In C++, the basic type for a set is `std::set`. Internally, this is implemented as a self-balancing tree, so that insertions, deletions, or queries all take $O(logn)$ time complexity.

C++ also offers the `std::unordered_set` that is based on a hash table. This offers $O(1)$ average case insertion, deletion, and queries, but is generally much less performant for iterating through the entire `std::unordered_set` than using an `std::set`.

5.3 Maps

Maps are very similar to sets, except that every element has an associated value with it as well.

C++ calls its map structure a `std::map` as the tree-based structure, with `std::unordered_map` serving as the hash-based variant. In Java, we have a `Map` interface, of which `TreeMap` and `HashMap` (among several other variants) are available. In Python, we have dict's as a language feature that serves as its map.

5.4 Sequential Structures

We define sequential structures as data structures that only let you see access one or two of its elements at any given time. Unlike an array that can access any arbitrary element

it stores, our sequential structures act as a container for elements that has very strong rules about what we're allowed to access.

5.4.1 Stacks

A stack is a first-in-last-out structure – you can add or remove elements from the back (or top) of the stack.

5.4.2 Queues

A queue is a structure that's the opposite of a stack in that it's first-in-first-out. Specifically, you can add elements to the back of a queue and remove elements from the front of the queue.

5.4.3 Deques

A Deque is a double-ended queue. Practically speaking, this structure supports both operations of stacks at queues at once – you can add or remove elements from either the front or the back.

5.4.4 Priority Queues

A priority queue, despite its name, does not behave exactly like a queue. While accessing and removing elements does happen at the front of a priority queue, a priority queue is always sorted and by consequence insertion can put an element anywhere in the priority queue (as long as it keeps the priority queue sorted).

5.5 Trees

While we have mentioned the possibility of trees in sets and maps, we often have to deal with trees as a general structure.

5.6 Binary Search Tree

One of the most common trees is the binary search tree.

6 Foundational Algorithms

There are a few algorithms that should be covered and discussed before going into much further explorations.

6.1 Sorting

Sorting is likely the most fundamental algorithm you will use in competitive programming. It appears in many problems, to the point where problems that just require sorting to solve them are usually considered relatively easy. More often you'll see sorting be involved as a necessary component to more difficult problems. In many cases, sorting is required in order to perform efficient algorithms (as we will see soon, sorting is required to perform a **binary search** which is generally much faster than a **sequential search**).

The goal of a sorting algorithm is simple, to transform an array of data to be sorted. A basic example would be given $1, 5, 3, 4, 2$ as an array, sorting would yield $1, 2, 3, 4, 5$. Or, $1000, 50, 100, 1$ becomes $1, 50, 100, 1000$.

6.1.1 Sorting Concepts

A **sorted** array refers to one where elements are ordered according to some criteria. In practice, this is usually in ascending order (so that each element is larger than the previous) and less occasionally in descending order (where each element is smaller than the previous, which sometimes is called "sorted in reverse").

In comparison, an **unsorted** array refers to one where this criteria is not met.

There are two main types of sorting algorithms: **comparison sorts** and **integer sorts**. The comparison variety relies on exactly what it says, comparisons.

What determines ascending or descending order can depend on what specifically is being sorted. For integers it's easy, one is clearly larger or smaller (or equal) to another. Same with floating point numbers. For strings, it's usually lexicographical order (alphabetical order). For more complex types it's not necessarily clear. To sort an array of queues stacks, it's less obvious what you should do, and it may depend entirely on your use case. In some problems you'll even find yourself with a `pair<int,int>` where you want to sort by the first integer at one part of your program, and then later sort by the second integer. In a comparison sort, you will have some **comparison function** that determines how to compare different elements with each other.

An integer sort on the other hand doesn't make use of any comparison function, and instead relies on properties about integers themselves. As a result, it doesn't work on arbitrary types, and you can't make it have its own sorting criteria by changing how it should compare elements. Generally, these sorting algorithms are much more specialized. That said a few integer sorting algorithms can offer very good runtime complexity, and depending on the data can offer significant speedups over any comparison sort alternative.

Another classification for sorting algorithms is whether they're **stable** or not. A stable sorting algorithm will ensure that different elements that compare equally to eachother (for custom classes with their own comparators for example) will have keep the relative order that they appeared in originally. For example, with an array of pairs $\{1,1\},\{2,2\},\{1,3\},\{2,4\},\{1,5\}$ where the comparison function only compares the first element of each pair, a stably sorted array would look like $\{1,1\},\{1,3\},\{1,5\},\{2,2\},\{2,4\}$.

In contrast, an **unstable** algorithm makes no guarantees. It could pretty easily still end up sorted the exact same as a stable sorting algorithm, but it isn't necessarily going to.

If every element of the array is unique, the difference between stable and unstable doesn't matter. If there are duplicate elements in the array, but you don't have any other data associated with them (two duplicate elements will be identical to eachother) then it doesn't matter either. It also wouldn't matter if duplicate elements with different data exist, but you don't really care what order they appear in as long as they're sorted relative to other elements. It's only in the (relatively narrow) circumstance that all these properties hold and the original order of duplicate elements is relevant that the difference between stable and unstable matters.

6.1.2 Insertion Sort

Insertion sort is a stable comparison-based sorting algorithm that's among the simplest to implement. The algorithm consists of iterating through the unsorted array, and for each element we check if the element before it is larger and swap if so, then check if the element before that is larger and swap if so, etc. In other words, we keep shifting an element earlier until elements to the left are all smaller and elements to the right are all larger.

```
// loop through each element
for (int i = 0; i < n; i++) {
    // propogate element to as early as possible
    // at each step, the range [0,i) should be sorted
    int j = i;
    while (j > 0 && arr[j] < arr[j-1]) {
        swap(arr[j], arr[j-1]);
        j-;
    }
}
```

What is the run-time of this algorithm? Unfortunately it is a fairly naive algorithm, and runs in $O(n^2)$. The worst-case is pretty simple to create, when the array is reversed we will have to do the maximum number of comparisons and swaps. Insertion sort isn't recommended as a general-use algorithm for these reasons – its runtime isn't ideal and its worst-case is very common to encounter (some test cases for a problem might be in reverse order just to cover all bases, not even necessarily to specifically exclude insertion sort).

While insertion-sort is generally not common because its $O(n^2)$ time complexity is too

slow for large arrays, it does have very little overhead when dealing with small arrays to the point of often being one of the fastest sorting algorithms. Specifically, when we're dealing with elements in the CPU cache, swapping elements is extremely fast. For this reason it's not uncommon to see it beind used as part of a more complicated hybrid sort. Timsort, the standard library sorting algorithm of Python, is one such hybrid sort that is based on a combination of insertion sort and merge sort, for example.

6.1.3 Quicksort

Under Construction

6.1.4 Mergesort

Mergesort can also be used to simplify certain problems, such as counting inversions in an array. An inversion is defined as a pair $i < j$ where $a[i] > a[j]$. In plain terms, an inversion is a pair of elements where the larger element appears earlier in the array. We can count this quickly using mergesort. Specifically when we do the merge step of mergesort, any time an element in the left half of the subarray is greater than the right half of the subarray, we know every other element remaining in the left half of the subarray is also greater. This means we can count the number of inversions as a part of mergesort, which takes $O(nlogn)$, instead of the naive approach of checking every pair in $O(n^2)$. This is not the only $O(nlogn)$ approach to this problem, but it should indicate that creatively using mergesort for other tasks related to sorting is an option.

6.1.5 Counting Sort

Consider the case where you have 10^9 1-digit numbers that you want to sort. It is certainly possible to perform a comparison-based sort that will run in $O(nlogn)$, there is actually an integer sort that's perfectly suited for this sort of problem that runs in $O(n)$.

Counting sort is based around having an array where each element is the number of times that index appeared in the original unsorted list. For example, with the array $0, 5, 2, 2, 2, 1, 5, 9$ we would have the counts $1, 1, 3, 0, 0, 2, 0, 0, 0, 1$ because 0 appears once, 1 appears once, 2 appears three times, 4 never appears, and so on. Generating this is a simple process – simply iterate through the unsorted list and increase the necessary count by 1 for every element.

The next step is to create a new sorted array, once we know the counts of each value. This time we iterate through our counts and insert them into our new array. When a value was counted multiple times, we insert it that many times. The result of this will be an array containing $0, 1, 2, 2, 2, 5, 5, 9$, the sorted version of the original array.

```
// generate the counts array from `arr`
for (int i = 0; i < n; i++)
    counts[arr[i]]++;

// sort `arr` using the counts array
int p = 0;
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < counts[i]; j++) {
        arr[p] = i;
        p++;
    }
}
```

Looking at the concepts of sorting, we can note that counting sort is:

- An integer sort
- $O(n + m)$ runtime, where n is the array size and m is the maximum element size (or size of the count array)
- $O(m)$ auxillary memory is used

Note that our count array has to be as large as the maximum element. Because of this (and that we have to traverse the entire count array), this means counting sort is best suited for when the range of valid elements is very small. A huge amount of one-digit numbers is a great example of where counting sort does well (and was chosen as the example here for that reason). When our elements are potentially very large, counting sort is not very well suited.

6.1.6 Cycle Sort

Cycle sort is a sorting algorithm designed to minimize the number of times the array is modified. It has a slow runtime of $O(n^2)$ so it's not used for general purpose sorting (outside of on hardware where memory writes are extremely expensive compared to memory reads) but it does have the neat property that it will only ever move an element into its correct position. Whereas in quicksort or mergesort, a single element may end up getting swapped many times before it ends up in its sorted position, cycle sort will only ever move an element at most one time (zero times if it's already in the right spot, once if it needs to be moved). This makes it potentially useful for some tasks where we want to know *how many elements need to be moved* or something along those lines, rather than just needing to sort it.

The core concept behind this algorithm is that moving an element from its unsorted position into a sorted position will also displace another element (for one element to be unsorted, there has to be at least one other element that's unsorted). Sometimes moving that second element into its correct position will displace a third element, which may itself displace another, and so on. Eventually we will end up trying to move an element into the spot our first element used to take up. We can consider these as cycles – moving elements into their correct positions will form a cycle of some length where we have to move other

elements afterwards. The cycle length is how many elements need to be moved before we end up at the position we started.

For example, if we try to sort $\{2, 1\}$, we will find we have a cycle of length 2 – when we put 2 into its correct position, we displace 1 and have to move that into its correct position (which used to belong to 2 but is now open). Similarly with $\{2, 3, 1\}$, we have a cycle of length 3, and moving 2 into its correct position displaces 3 which would then displace 1 before we finish the cycle. For simplicity's sake, we can consider a single element in its sorted position already to have a cycle length of 1, but we don't need to move it because it would simply return to the position it's already at.

The logic of the algorithm itself is to iterate through the array, and for each element, count the number of elements smaller than it (so that we know the correct position of that element in the sorted array). When we find an element that's out of place, we track its position as the start of the cycle, then attempt to place it in its proper position. We then try the same with the element we just displaced, counting the number of elements smaller than it and finding its correct position in the array, and putting it in its proper spot. If we bring it to the start of the cycle, we simply move our displaced element there. If it's not the start of the array, then we're displacing another element and we repeat the process. We continue until the cycle is done, and then iterate to the next element in the array.

6.1.7 Other Sorting Algorithms

It's likely that you've heard about or seen other sorting algorithms that we haven't covered in the above subsections. The truth is, we usually aren't too concerned with the details of various sorting algorithms in competitive programming. We listed some sorting algorithms because they're relevant (the details for merge-sort sometimes is useful to know for solving some problems), useful (counting sort is something that will occasionally get used for competitive programming), or more likely that they teach some fundamental concepts that we use elsewhere (quicksort's partitioning sees use in other algorithms).

If we're not too concerned with sorting algorithm details, what do we do instead? In the vast majority of cases, using the standard library to sort things is sufficient (and definitely far easier).

- C++ supports `std::sort` for general purpose sorting, and `std::stable_sort` for when a stable sort is required (because `std::sort` is not guaranteed to be stable). Both of these require `#include <algorithm>`.
- Java supports `Arrays.sort()` via `import java.util.Arrays;` for arrays specifically, and `Collections.sort()` via `import java.util.Collections;` for general data structure containers (including arrays). These sorts are stable by default.
- Python lets you either create a sorted copy of a list or iterable via `sorted(arr)` or lets you sort a list directly via `arr.sort()` (which is more efficient, but loses the original ordering and only works for lists). These sorts are guaranteed to be stable.

For interest, some other sorting algorithms are described here:

- **Bubble sort** is a commonly taught algorithm as a very simple (but very naive

and slow) sorting algorithm. You iterate through the list multiple times, and if two adjacent elements are out of order, they get swapped. It doesn't see much practical use because other sorting algorithms are generally better.
- **Selection sort** is a naive sorting algorithm that for each element, it finds whatever element is smallest out of all elements to the right and swaps it into its right place. You can be guaranteed that, at the ith iteration, the range $[0, i)$ contains the right elements that need to be sorted.
- **Heapsort** is an algorithm that inserts all elements of an array into a heap, and then removes all those elements from the heap.
- **Radix Sort** is an integer sorting algorithm that recursively sorts by individual digits – you take either the most-significant-digit or least-significant-digit of a number, distribute elements into different buckets based on what that digit was, then within each bucket you repeat with the next digit, and so on.
- **Bogosort** an extremely naive algorithm that creates a random permutation of an array, checks if it's sorted, and if not repeats. It's notable that it has an expected runtime of $O(n!)$ and in the worst case (if you are extremely unlucky) will never finish, but in the best case is $O(n)$.
- **Dropsort** is an extremely fast algorithm that sorts an entire array in a single pass, by simply removing any elements that are smaller than the one before it. It is usually mentioned as a joke algorithm for this reason – but the output is a sorted array in any case (even if it's potentially much smaller than the original array).

That isn't to say that we have a complete list of all sorting algorithms – not at all, there are lots of them. Some fairly well-known but not listed here, some that are variants on existing algorithms, and some that are nearly unheard of but still a valid sorting algorithm nonetheless.

Large and Small Pairs

The absolute difference between two integers, $|a - b|$, can be quickly calculated as the positive value of $a - b$. That is to say, if $a - b$ is negative, we turn it into a positive value instead.

In this problem, we have an array of integers and want to find the maximum and minimum absolute difference between any two pairs in the array.

Input

Input consists of an integer n where $1 \leq n \leq 10^7$, followed by n integers a_i where $0 \leq a_i \leq 10^9$.

Output

Output the maximum absolute difference between any two pairs in a, followed by the minimum absolute difference between any two pairs in a.

Constraints

Time Limit: 1 second
Memory Limit: 1024 mb

Samples

Input:

```
6
10 1 20 11 100 50
```

Output:

```
99 1
```

Input:

```
5
20 20 0 20 20
```

Output:

```
20 0
```

6.2 Searching

Often, you need to check whether an element exists within an array. Other times, you have to find the index of that element. In any case, given an array (or other list of elements), searching for an element is a common task. Like sorting, this is often fairly simple when done on its own, and more often than not it's a subtask of a larger problem or a part of a larger algorithm entirely.

There are various algorithms for searching for an element within an array.

6.2.1 Sequential Search

Sequential search (also called linear search) is the simplest form of searching for an element within an array. It consists of a simple for-loop that checks each individual element

It's useful because it doesn't rely on the array being sorted or anything like that. While it is often quite slow, it can be useful when runtime is not a concern.

6.2.2 Binary Search

Binary search is an extremely common search algorithm that uses the structure of a sorted array to offer great speedups.

6.2.3 Exponential Search

Exponential Search is a variant on binary search that both performs slightly better than binary search when the search key is near the beginning of the list, and also works for unbounded lists (not generally an issue with arrays, but if you were to binary search a nonstrictly increasing function without a clear maximum, this would apply).

The idea is to first obtain a valid range to binary search. Exponential search specifically tries to check if the search key potentially exists within the ranges $[0, 1)$, $[1, 2)$, $[2, 4)$, $[4, 8)$, $[8, 16)$, and so on. The patterns for the ranges isn't too complicated – we double our upper bound each time, and our lower bound is just our upper bound divided by 2. Checking whether the key is potentially within that range is simple – as long as the key is between the upper and lower bounds of the range, it could exist.

Once we have the valid range, we simply perform a normal binary search on that range. The code as a result looks like:

```
// get range to binary search
// remove "i < n" if unbounded
int i = 1;
while (i < n && arr[i] <= k)
    i *= 2;

// binary search the range
// replace "min(i,n)" with "i" if unbounded
return binary_search(arr + (i/2), arr + min(i, n), k);
```

6.2.4 Quickselect

Let's consider the case where you want to find the kth smallest element in an unsorted array. One of the easiest options is to sort that array and then just look at the kth element there.

You will notice however, that we don't actually need most of the array to be sorted for this to be the case. If we're looking for the smallest element, it's no concern of ours if the majority of the array is unsorted. If we're looking for the median element, we don't care that what's to the left and what's to the right are fully sorted, just as long as we're sure the median element is where it's supposed to be (which we can verify by every element to the left being smaller, and every element to the right being larger).

If you can notice the wording of that is fairly similar to quicksort, you're correct. **Quickselect** is a selection algorithm that uses similar concepts like partitioning that quicksort uses, in order to find the kth smallest element by only partially sorting the array.

6.2.5 Unimodal Searches

Less commonly, you may want to search the outputs of a function rather than an array. There are some search algorithms designed to search for maximums or minimums in a unimodal function (a function that has only one unique maximum or minimum).

Ternary Search is...

Golden Section Search is...

Under Construction

7 Iterative Problems

7.1 Brute-Force

Brute force problems are best described as problems where the intended (or at least, a valid) solution involves trying every possibility. These are usually marked by very small problem bounds, because of how computationally complex this process can be.

What exactly brute-force looks like depends on the problem itself. In some cases, it's "try all pairs" which is $O(n^2)$. It can be "try all triplets" at $O(n^3)$. Sometimes it's all combinations, in $O(2^n)$. In some cases, it's try all permutations which is $O(n!)$.

There is a reason we distinguish brute-force *problems* from brute-force *solutions*. For most problems, it's possible to find a brute-force solution that is guaranteed to give you the right answer. Usually it's pretty easy to come up with a technically valid brute-force solution, in fact. Most problems can't be solved with this approach because it tends to be extremely slow. A brute-force problem is a problem where the brute-force solution is valid and can work.

7.2 Simulation

Simulation problems are problems where the problem description describes some step-by-step algorithm (and the algorithm may be a single step repeated some number of times), and the intended solution is to simply perform that algorithm.

Generally, the difficulty from simulation problems comes from how difficult their implementation is, not from figuring out the solution. They tend to be relatively straightforward to understand a solution for (the problem statement tells you the solution, in a sense) but can be difficult to convert into code. Sometimes, the implementation details matter and while you're still just implementing what the problem statement tells you to do, you have to consider what data structures you're using in order to pass the time constraints on the problem.

We'll look at a simulation problem that involves a mathematical concept "Happy Numbers". Despite looking math-related, a simulation approach is the recommended solution.

Very Happy

If you take a positive integer and sum the square of all its digits, you end up with an interesting transformation. For example, applying this to 123 gives us $1^2 + 2^2 + 3^2 = 14$.

We can repeat this process multiple times, such as taking 14 and getting $1^2 + 4^2 = 17$, 17 becoming $1^2 + 7^2 = 50$, 50 becoming 25, 25 becoming 29, becoming 85, then 89, then 145, then 42, 20, then 4. This will continue to loop forever, since it turns out that 4 enters a cycle $4, 16, 37, 58, 89, 145, 42, 20, 4$.

Not all numbers will enter into a long cycle when you do this – some numbers will turn into 1 when you repeat this process enough times (and applying this process on 1 gives us $1^2 = 1$). These are called happy numbers.

The task here is to determine whether a number is happy or not. And if it is happy, figure out how many iterations it takes to do so.

Input

Input consists of an integer $1 \leq t \leq 10^5$. Then follows t integers $1 \leq n \leq 10^9$.

Output

For every integer n, output either the number of iterations it takes to become 1, or "impossible" if it never does.

Constraints

Time Limit: 1 second
Memory Limit: 1024 mb

Samples

Input:

```
3
1
4
19
```

Output:

```
0
impossible
4
```

7.3 Two-Pointers

Two-pointers problems, generally speaking, are problems that can be solved by keeping track of two locations on a sorted array. One pointer starts at the start of the array (the smallest element) and the other starts at the end of the array (the largest element). You are asked to find a pair of elements for which a certain property holds.

A classic example is "given a sorted array of integers, determine whether a pair of elements sums up to 100".

It is simple to solve this problem using a brute-force approach in $O(n^2)$:

```
// for each element in the array we will search for a pair
for (int i = 0; i < n; i++) {
    // we can start j at i+1 because earlier pairs were already checked
    for (int j = i+1; j < n; j++) {
        if (arr[i] + arr[j] == 100)
            return true;
    }
}
return false;
```

It can be improved further by using binary search instead our nested for loop, which turns it $O(nlogn)$ instead:

```
// for each element in the array we will search for a pair
for (int i = 0; i < n; i++) {
    // binary search before the current element
    // looking for what will add up to 100
    if (binary_search(arr, arr + i, 100 - arr[i]))
        return true;
    // do the same for after the element
    if (binary_search(arr + i + 1, arr + n, 100 - arr[i]))
        return true;
}
return false;
```

But with a two-pointers approach we can perform this in $O(n)$:

```
// set up our two pointers
int left = 0, right = n-1;
// loop as long our pointers haven't met eachother yet
while (left < right) {
    int sum = arr[left] + arr[right];
    // if we found a matching pair, end early
    if (sum == 100)
        return true;
    // otherwise we adjust our pointers
    if (sum > 100)
        right-;
    else
        left++;
}
return false;
```

To explain how this works internally, let's consider an array $1, 2, 7, 49, 50, 51, 95, 100, 110$. We start with considering the first and last elements, and then we will repeatedly perform this:

1. $1 + 110 = 111$, which is greater than 100 so we decrease right

2. $1 + 100 = 101$, which is greater than 100 so we decrease right
3. $1 + 95 = 96$, which is less than 100 so we increase left
4. $2 + 95 = 97$, which is less than 100 so we increase left
5. $7 + 95 = 102$, which is greater than 100 so we decrease right
6. $7 + 51 = 58$, which is less than 100 so we increase left
7. $49 + 51 = 100$, which is our target value, so we return true

It's useful to compare with an array that doesn't have a correct answer, like $1, 5, 97, 100$:

1. $1 + 100 = 101$, which is greater than 100 so we decrease right
2. $1 + 97 = 98$, which is less than 100 so we increase left
3. $5 + 97 = 102$, which is greater than 100 so we decrease right
4. both pointers are at the second element, so we return false

There's two things to note about this approach: it relies on the array being sorted (if it isn't sorted, you must sort it first), and that you at every step of the way, you either:

- found a valid pair
- know that your left pointer is too small (because your sum is too low, and changing the right pointer will only decrease it further)
- know that your right pointer is too high (because your sum is too high, and changing your left pointer will only increase it further)

7.4 Sliding Window

In a sliding window problem, we have a list of elements, and we want to find a range of a certain length with some property.

Consider a problem like "given an array of integers, determine whether a subarray of length k sums to 100". This isn't too hard to think of conceptually, and we can solve it with a straightforward brute-force approach – sum all elements from $[0, k)$ and see if they equal 100, then sum from $[1, k + 1)$ and see if they equal 100, and so on.

You'll notice that we redo a lot of work with that brute-force approach. Both $[0, k)$ and $[1, k + 1)$ involve the sum $[1, k)$, so we can reduce the amount of operations we need to do if we can stop re-adding elements together unnecessarily.

With a sliding window, we can move from $[0, k)$ to $[1, k + 1)$ by subtracting $arr[0]$ and adding $arr[k]$, so that we now have our next range in two operations instead of k operations.

```cpp
bool bruteforce(vector<int> arr, int k) {
    // sum all ranges [i,i+k) and check
    for (int i = 0; i <= arr.size() - k; i++) {
        int sum = 0;
        for (int j = 0; j < k; j++)
            sum += arr[i+j];
        if (sum == 100)
            return true;
    }
    return false;
}

bool slidingwindow(vector<int> arr, int k) {
    // sum range [0,k)
    int sum = 0;
    for (int i = 0; i < k; i++)
        sum += arr[i];
    if (sum == 100)
        return true;

    // check all other ranges
    for (int i = k; i < arr.size(); i++) {
        sum += arr[i];
        sum -= arr[i-k];
        if (sum == 100)
            return true;
    }

    return false;
}
```

7.4.1 Resizing Window

You can combine concepts with two-pointers and sliding windows together, where you keep left and right pointers like with two-pointers, but instead of considering pairs we consider ranges like with sliding windows.

A classical example for this is "given an array of non-negative integers, determine whether any subarray sums to 100". We approach this by starting with a range that's only the first element, and then if it's too small we add the element to the right of our window, and if it's too large we subtract the element at the left.

For example, let's look at the array $20, 30, 5, 5, 50, 30, 5, 5, 80, 20$. If we want to find a subarray that sums to 100, we can look:

20	20	30	5	5	50	30	5	5	80	20
50	20	30	5	5	50	30	5	5	80	20
55	20	30	5	5	50	30	5	5	80	20
60	20	30	5	5	50	30	5	5	80	20
90	20	30	5	5	50	30	5	5	80	20
90	20	30	5	5	50	30	5	5	80	20
95	20	30	5	5	50	30	5	5	80	20
100	20	30	5	5	50	30	5	5	80	20

Where the cells shaded blue are the current subarray we're considering, and the number to the left represents the current sum of the entire range. We find our final answer to be the subarray $5, 5, 50, 30, 5, 5$.

Note that in this example problem we're just looking for whether or not there is a subarray that sums to 100. If we want to find all subarrays that sum to 100, or the smallest such subarray (in our above example, the last two elements $80, 20$ sum to 100 in a mich smaller subarray), or some other criteria, we can still use the resizing window approach but will have to prevent it from finishing as soon as we find a valid range.

There are various ways to implement this. Here's an example implementation that used a for-loop for the left pointer, and a nested while-loop for the right pointer.

```
int right = 0, sum = 0;
for (int left = 0; left < n; left++) {
    // get rid of the last left value
    if (left != 0)
        sum -= arr[left-1];

    // ensure we always have a range
    if (right <= left) {
        sum = arr[left];
        right = left + 1;
    }

    // try to expand right as far as we can
    while (sum < 100 && right < n) {
        sum += arr[right];
        right++;
    }

    // check if the sum is equal to our target
    if (sum == 100)
        return true;
}
return false;
```

Something a keen eye might've noticed is that our example problem was "non-negative integers". This was carefully chosen, because this approach doesn't work when we include the possibility of negative numbers. Specifically, the sliding window technique relies on increasing the window size to always either increase our sum or keep it the same, and likewise that decreasing the window size will always decrease our sum or keep it the same. This assumption doesn't hold with negatives numbers.

To demonstrate the problem with negative numbers, let's look at the counter-example $50, -1, 50, 50, -99$. It should be obvious that there is a subarray that sums to 100, specially the $50, 50$ in the middle. If we attempt to find this via a resizing window, we will instead get:

1. our range is 50 with sum 50, which is less than 100 so we increase our window size
2. our range is $50, -1$ with sum 49, which is less than 100 so we increase our window size
3. our range is $50, -1, 50$ with sum 99, which is less than 100 so we increase our window size
4. our range is $50, -1, 50, 50$ with sum 149, which is greater than 100 so we decrease our window size
5. our range is $-1, 50, 50$ with sum 99, which is less than 100 so we increase our window size
6. our range is $-1, 50, 50, -99$ with sum 0, which is less than 100 so we increase our window size, except that we have traversed the entire array and have no more elements to consider

7.5 Constructive

Sometimes you will find problems along the lines of "here is a list of restrictions, construct an array that meets those restrictions" or "create the optimum answer under a list of restrictions". Usually, they also have some variant on "if there are multiple solutions, print any of them."

Generally, constructive problems require creative thinking to figure out a valid pattern that will satisfy the requirements. Usually, the implementation of a valid program itself is not very complicated, and it's being able to determine a valid pattern that is the challenging and interesting part of the problem.

Constructive problems are not necessarily iterative, but it's reasonable to introduce constructive problems here because many of them are.

We'll look at an example of a constructive problem:

Big Difference

"Big Difference!" Ivan said to his teacher, pointing at a list of numbers the teacher wrote on the board.

"That's right Ivan, there is a big difference here", the teacher said pointing to how the difference between every two elements in the list is fairly large. "In fact, I had carefully ensured that the difference between all elements was as large as possible, so that no elements next to each other was small." Specifically, the teacher made a list where the smallest (absolute) difference between every adjacent element was as large as possible.

Ivan wanted to figure out how to do this, but was not very fast at it. He couldn't figure out a way to solve it other than trying every different arrangement of the list, which was far too slow when he had a large list to deal with.

Input

Input consists of a single integer n where $1 \leq n \leq 10^8$.

Output

Output an list of numbers of size n with values $[1, 2, 3, ..., n]$ where the smallest absolute difference between adjacent elements is maximized. More formally, create a permutation of an array a with values $[1, n]$ with the maximum value of $min(|a_1 - a_2|, |a_2 - a_3|, ...|a_{n-1} - a_n|)$. If there are multiple answers, print any.

Constraints

Time Limit: 1 second
Memory Limit: 1024 mb

Samples

Input:

```
3
```

Output:

```
2 1 3
```

Input:

```
2
```

Output:

```
1 2
```

There's a few things about this problem that stands out. The restriction is fairly basic – we just have to make sure the minimum difference between adjacent elements is as large as possible. Our output is just a list of numbers between 1 and n that's ordered in a way that meets this restriction.

There can be multiple variants on this problem. It could also provide an input k and ask that no two adjacent elements have an absolute difference less than k. It could ask for the lexicographically smallest answer if there are multiple (if 213 and 312 as both valid answers, the lexicographically smallest answer would be 213 since 2 is less than 3).

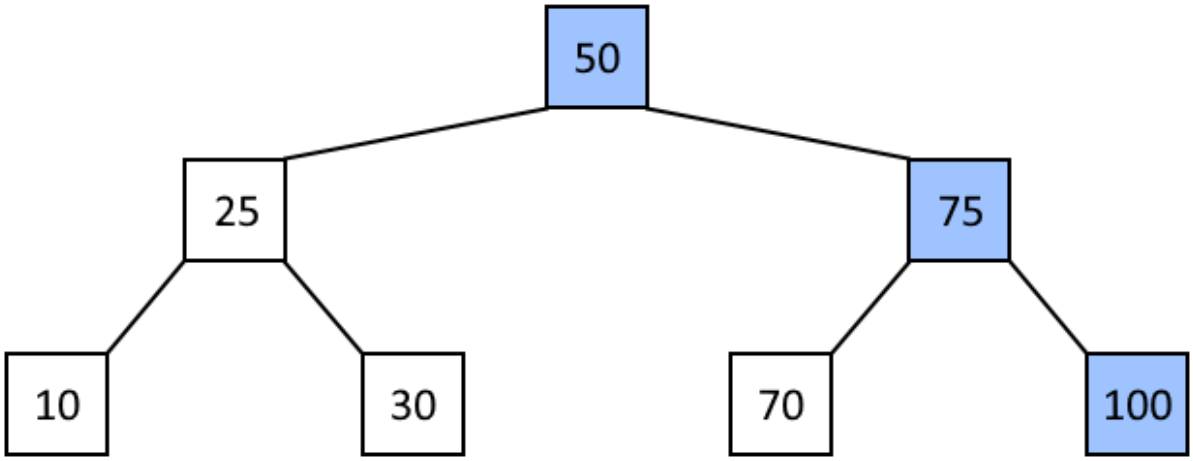
It's worth mentioning that it's rare for the sample inputs/outputs to actually describe each possible different valid answer for a given input. It's also unlikely that the test cases will show particularly large test cases. The reason for both of these is that a valid pattern may become obvious with this information. The sample cases are usually carefully chosen so that it's clear what the IO expects and can clarify what the problem expects, but doesn't give you any strong leads into what a solution might look like.

8 Greedy Problems

Greedy problems are problems where the optimal solution can always be developed by, at each opportunity, making the choices that provide the most immediate benefit. In other words, the locally optimal choice is also (part of) the globally optimal choice, in all circumstances. Practically, this just means that at any given step of an algorithm, we can reasonably and reliably make the best choice we can.

For example, a simple greedy problem might look like: we are given a binary search tree (remember that elements larger than a given node are always to the right, as a property of binary search trees), and we must print what the maximum element of this tree is.

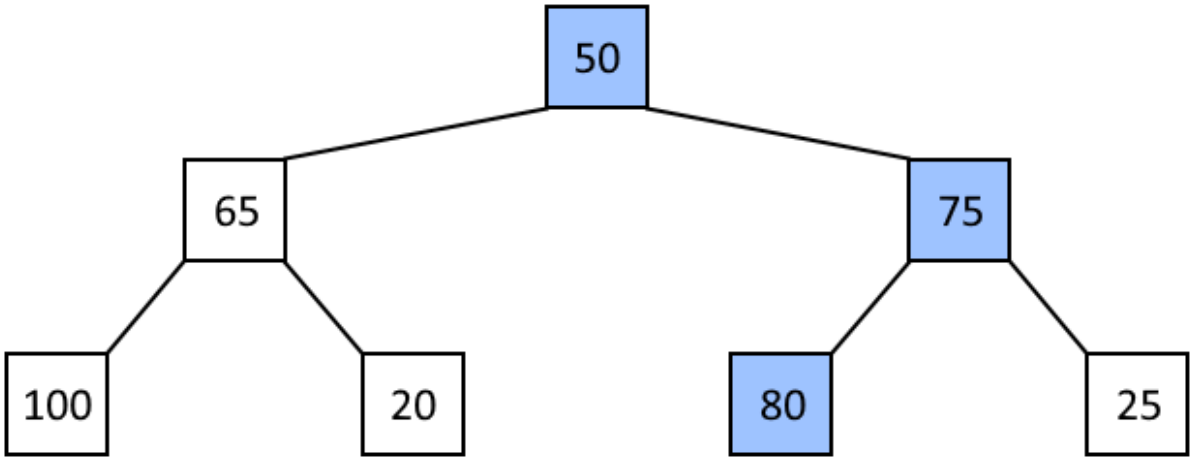
The greedy solution to this problem is to always traverse to the right of the tree until we end up at a leaf node. This leaf node is the largest element of the tree, guaranteed by the properties of a binary search tree. In greedy terms, starting from the root of the tree, we will greedy move to the largest child node.



In the above example, we can easily verify that 100 is the largest element in the binary search tree.

We can show a similar but slightly modified problem that stops being greedy: we are given a binary tree (that isn't necessarily a binary search tree), and we must print what the maximum element of this tree is.

Here, this is problem is no longer greedy because we can't reliably say that the largest element is always a child of the larger node at each step. Our greedy solution only worked because we could be guaranteed, at each decision, that our locally optimal choice would lead us to the global optimal.



While in this case we do greedily take the locally optimal choice at each step, the globally optimal answer involves us taking the locally sub-optimal choice of 65 instead of 75. Because we're not dealing with a binary search tree, we don't have the guarantees we need to greedily approach this problem.

8.1 Knapsack

Knapsack problems are those that involve having a knapsack with a maximum capacity c, and n items with values v and weights w. There are many variants on this problem, but the goal is to put items into the knapsack so that our total value is as high as possible without the total weight exceeding the maximum capacity.

Some knapsack variants are greedy, which we will explore here. Some are not, which won't be covered in depth here, but we will make mention of a few that can't be solved greedily and give reasons why.

In an unbounded knapsack problem, we can put any amount of an item into the knapsack – both multiple copies of the same item, and fractional amounts of that item.

The solution in this case is to determine what item has the highest $\frac{v}{w}$ to maximize the value we get for its weight, and then fill the entire knapsack with that.

In a fractional knapsack problem, we are allowed to put fractional amounts into the knapsack, but we can't put multiple of the same item.

The general concept is the same as before, we greedily use whatever item has the highest $\frac{v}{w}$ because it gives us to most optimal usage of our capacity. This time though, we can't fill our entire knapsack with that one item, and likely have to start using the item with the next highest $\frac{v}{w}$ assuming we still have capacity to spare. We do this until eventually we either run out of items, or we run out of capacity (potentially needing to use a fractional amount of the last item we insert).

One of the more well known knapsack problems is the 0-1 knapsack problem, where we can only insert an item exactly once, or not at all. In other words, we have to choose for each item whether to include or exclude it.

If we try a greedy approach to this problem, we will quickly find cases where it doesn't work. For example, if our capacity c is 10, and we're given items with value-weight pairs $\{15, 6\}, \{8, 5\}, \{8, 5\}$, our greedy approach will take $\{15, 6\}$ because $\frac{15}{6} > \frac{8}{5}$, but we can't fill our knapsack with anything more meaning our total value is 15. If we instead go with $\{8, 5\}$ we can fill the rest of the knapsack with the other $\{8, 5\}$ and end up with a total value of 16.

The issue here being that we can only assume the greedy approach will work if, no matter what choices we make, we can always fill the entire knapsack (assuming we have enough items to fill the entire knapsack). The unbounded knapsack problem and the fractional knapsack problem do this by letting us use fractional amounts of an item. In the 0-1 knapsack problem, we can't make this guarantee, and the locally optimal choice with the highest $\frac{v}{w}$ may not be the globally optimal choice, because another choice may lead to us filling more of the knapsack and ending up with a higher total value.

8.2 Coin Change

Under Construction

8.3 Interval Cover

In this problem you're given a range $[L, R]$ and a list of intervals $[l, r]$. The task is to determine whether these intervals fully cover the range $[L, R]$.

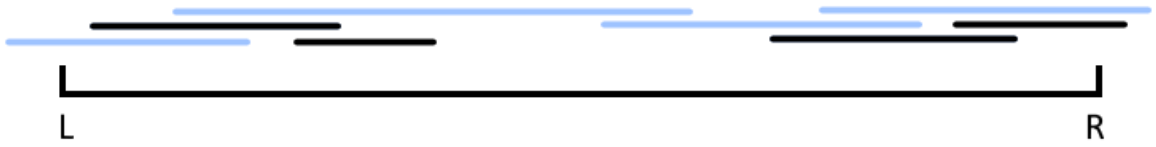
Let's first consider a non-greedy, recursive approach (formally, a depth-first search approach, that we will describe in a later section). Starting at the leftmost-interval (the interval with the smallest value l, even if r is large) that contains L, we find every interval that starts within the range $[l, r]$ (in other words, every overlapping interval that's to the right) and then for each of those intervals we recursively try the same. We will either exhaust all overlapping intervals before we can reach R and know that we can't cover the entire range $[L, R]$, or one of the intervals contains R and know that we do cover the entire range (at which point our algorithm is done).

With recursively considering the leftmost overlapping intervals, we can see in blue what gets considered:

It's nearly all intervals that get considered, with the only exception being that we find an interval containing R at the end. The above diagram doesn't account for, depending on implementation, that you may end up considering the same interval multiple times as well.

The greedy optimization of this is recognizing that we don't need to check every interval that overlaps, we only need to check the rightmost interval (the overlapping interval with the largest value r, **not** the overlapping interval with the largest value l) because it every other interval we might consider is suboptimal to that. This means that we can consider far less intervals.

We can see with the same example as above that we have to check less intervals in total:



9 Dynamic Programming Problems

Dynamic programming (often shortened to "dp") is a method to break down certain problems into smaller sub-problems, very similar to recursion. It can be thought of as an optimization over regular recursion, where multiple calls with the same input only have to be evaluated once instead of multiple times.

Consider the problem of calculating the Fibonacci sequence. The Fibonacci sequence is the sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$ where each value (other than the initial $0, 1$) is the sum of the previous two (so $89 = 55 + 34$, $55 = 34 + 21$, $34 = 21 + 13$, and so on). This is a pretty intuitive example of recursion, like in:

```
// calculate the nth fibonacci value
int fibonacci(int n) {
    // f(0) and f(1) base cases
    if (n < 2) return n;

    // general recursive case
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Unfortunately, this results in a lot of repeated work. A call to `fibonacci(10)` will call `fibonacci(9)` and `fibonacci(8)`, where `fibonacci(9)` will also call `fibonacci(8)`, `fibonacci(7)` ends up getting called 3 times, `fibonacci(6)` gets called 5 times, and so on until `fibonacci(1)` ultimately gets called 56 times. There is a huge amount of work being done with these recursive calls *that we already obtained the result from earlier*.

Dynamic programming fixes this by storing the results of these recursive subcalls in a process called memoization. We can simply store a value when we calculate it, so that we can reduce the number of recursive calls done.

```
// store our base cases f(0) and f(1)
map<int, int> f = {{0,0}, {1,1}};

// calculate the nth fibonacci value
int fibonacci(int n) {
    // if there is a stored result, use it
    if (f.count(n)) return f[n];

    // store our result
    f[n] = fibonacci(n-1) + fibonacci(n-2);

    return f[n];
}
```

That is called "top-down" dynamic programming because we still make recursive calls, but if we've already gotten the required result from one of our earlier recursive calls we'll store it for any times it appears later.

The alternative is "bottom-up" dynamic programming where we start with the smallest cases and gradually work our way up:

```
// calculate the nth fibonacci value
int fibonacci(int n) {
    int f[100];

    // f(0) and f(1) base cases
    f[0] = 0;
    f[1] = 1;

    // general "recursive" case
    for (int i = 2; i < 100; i++)
        f[i] = f[i-1] + f[i-2];

    return f[n];
}
```

Here, we don't make any recursive calls at all – a call of `fibonacci(10)` will not result in recalculating earlier values in the sequence repeatedly. We start from the base cases, and work our way up from the bottom, hence the term "bottom-up".

You may note that in this problem we can store this permanently, by precalculating entirely:

```
int f[100];

// precalculate fibonacci sequence
void genFibonacci() {
    // f(0) and f(1) base cases
    f[0] = 0;
    f[1] = 1;

    // general "recursive" case
    for (int i = 2; i < 100; i++)
        f[i] = f[i-1] + f[i-2];
}

// calculate the nth fibonacci value
int fibonacci(int n) {
    return f[n];
}
```

Where we call `genFibonacci()` at the start of the program, and then obtain results from `fibonacci(n)` or even directly with `f[n]`.

This works in this specific instance because our stored results are always the same: we always begin with 0,1 and always add the previous two terms. If this might change, whether it be by having different starting numbers or a different number of previous

terms (or even coefficients on those terms), we couldn't precalculate beforehand.

In theory, any problem that can be solved recursively can also be solved with dynamic programming. We can simply apply top-down dynamic programming to an existing recursive solution, as long as our recursive function doesn't have any side-effects like printing or modifying a global state. But we don't necessarily see any benefit from doing this – if there's no overlap in our recursive calls, there's no benefit to storing results.

9.1 Coin Change

Something of interest is that we can also use dynamic programming to efficiently count the number of valid solutions for a given problem.

An example of this will be in the coin change problem. Let's say we have a target value 100 and a list of elements $10, 25, 90$, and we want to count how many different ways we can make change from 100 evenly using the elements of our array. In this variant, we have an infinite amount of each element (so that we can reuse the same values repeatedly). It should be pretty easy to determine the valid solutions, $\{10, 90\}$, $\{10, 10, 10, 10, 10, 25, 25\}$, $\{25, 25, 25, 25\}$, and $\{10, 10, 10, 10, 10, 10, 10, 10, 10, 10\}$. Therefore, the count of different ways to make change from 100 evenly is 4.

What do we do when our list of valid coins is $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 17, 25, 90\}$? It should be clear that this would be difficult to do naively – 100 1s is a valid solution, 98 1s and a 2 is a valid solution, 97 1s and a 3 is a valid solution, 96 1s and a 4 or 2 2s are both valid solutions, something like 90 1s and any of $\{10\}, \{8, 2\}, \{7, 3\}, \{6, 4\}, \{5, 5\}, \{6, 2, 2\}, \{5, 3, 2\}, \{4, 4, 2\}, \{4, 2, 2, 2\}, \{3, 3, 2, 2\}, \{2, 2, 2, 2, 2\}$ are all valid solutions. It is far too difficult to simply look at the problem and be able to accurately determine the proper number of valid solutions (in this case, that number is 9505074).

```
// our dp table
int counts[n+1] = {0};

// our base case
// there is 1 way to solve 0
counts[0] = 1;

// for every coin, we go through the array
// and add the solutions from our subproblems
for (auto e : coins)
    for (int i = 0; i <= n - e; i++)
        counts[i+e] += counts[i];

// return the solutions with n as the target
return counts[n];
```

9.2 Longest Common Subsequence

The Longest Common Subsequence problem (commonly called the LCS problem) is one where you're given two strings (or some other array of data) and, as the name suggests, you have to find the longest subsequence they have in common.

Recall that subsequences refer to a sequence with some elements deleted from the original (including no elements or all elements), so that there is (potentially) less elements but they are all from the original sequence and keep the original ordering. For example, with the string "abcdef", some valid subsequences would be "abc" or "ace", or even the original string "abcdef" or an empty string "".

A subsequence in common, therefore, would be any subsequence that exists in two different sequences at once. Given two strings "abcdef" and "defghi", they might contain the common subsequences {"", "d", "e", "f", "de", "ef", "def"}. As another example, the common subsequences of "abcdef" and "aceg" would be {"", "a", "c", "e", "ac", "ce", "ace"}. It should be clear what the longest common subsequence is with these examples, "def" and "ace" respectively.

A potential problem that we may have to deal with is that the first possible subsequence we can take isn't necessarily the best one. In the above examples, you might notice that simply adding all letters they have in common would give you the right answers in those testcases. This is not true for all possible strings, for example with "abcde" and "abdbce" if we took a greedy approach we would likely find "abd" when the longest common subsequence is actually "abce".

We can solve this with dynamic programming:

```
int LCS(string l, string r) {
    // get sizes for convenience
    int m = l.size();
    int n = r.size();

    // our 2d dp state table
    int table[m+1][n+1];

    // initialize base states
    for (int i = 0; i <= m; i++)
        table[i][0] = 0;
    for (int i = 0; i <= n; i++)
        table[0][i] = 0;

    // perform dp
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (l[i-1] == r[j-1])
                table[i][j] = table[i-1][j-1]+1;
            else
                table[i][j] = max(table[i-1][j], table[i][j-1]);
        }
    }

    return table[m][n];
}
```

9.3 Longest Increasing Subsequence

A similar (but also standard) problem as above, we want to deal with the longest subsequence of a single array where all elements are increasing. The main variants of this problem are whether the elements are strictly increasing or not – for this description we will go with strictly increasing.

As a simple example, if we're given the array $\{1, 5, 6, 2, 3, 4, 7\}$ then the longest increasing subsequence is $\{1, 2, 3, 4, 7\}$.

There's a few important observations we can make:

- Every element always exists within some increasing subsequence, even if it's just itself. For example, in a decreasing array like $\{5, 4, 3, 2, 1\}$, every increasing subsequence is $\{5\}, \{4\}, \{3\}, \{2\}, \{1\}$. As such, our longest increasing subsequence will always exist and it'll always be at least 1.
- The next element in the longest increasing subsequence isn't necessarily the next increasing value. For example, consider the array $\{1, 4, 2, 3, 4, 5\}$. If we greedily take the first element that's increasing, we will get $\{1, 4, 5\}$ as our subsequence. This is suboptimal, since $\{1, 2, 3, 4, 5\}$ is the actual longest subsequence in this case.
- The next element in the longest increasing subsequence isn't necessarily the next

smallest value, either. Consider $\{1, 3, 4, 5, 2\}$, greedily taking the next smallest element will get us the subsequence $\{1, 2\}$. The correct answer in this case is $\{1, 3, 4, 5\}$.

Considering the above, our solution can't be greedy – we would need to account for the optimal solution involving elements that may not be immediately obvious to a greedy solution.

As for how to actually solve this, we can explore a bottom-up approach. In our memoization table, every element $memo[i]$ contains the length of the longest increasing subsequence in the subarray in the range $[0, i]$ that includes $arr[i]$. The approach to finding this requires us to look at the longest increasing subsequences of our earlier elements: we iterate through the range $[0, i)$ with j, and then find the maximum value of $memo[j]$ where $arr[i] > arr[j]$, and set $memo[i] = memo[j] + 1$. In other words, we want to find the longest increasing subsequence that we can append $arr[i]$ to. If there are no existing longest increasing subsequences that $arr[i]$ can be appended to, we simply set $memo[i] = 1$ as a base case.

When we finish our entire array in this fashion, our answer for the longest increasing subsequence is the maximum value of our memoization table.

```
int LIS(vector<int> arr) {
    // setup memoization table
    int n = arr.size();
    vector<int> memo(n);

    // for each element of arr
    for (int i = 0; i < n; i++) {
        // set base case
        memo[i] = 1;

        // get maximum subsequence size with this element
        for (int j = 0; j < i; j++) {
            if (arr[i] > arr[j] && memo[i] <= memo[j])
                memo[i] = memo[j] + 1;
        }
    }

    // get the longest subsequence size
    int best = 1;
    for (auto e : memo)
        if (e > best)
            best = e;

    return best;
}
```

9.4 Problems

Dynamic Fitness

Welcome to Dynamic Fitness Gym Equipment Emporium! Here, we sell all sorts of gym equipment for all your fantastic gymnastic needs!

The special sale for today is on dumbbell weights. Name a target total weight and we can give you the right combination of weights to hit that target. We can do this for any weight in kilograms up to a whopping 2000kg!

Actually it'd be nice if someone could verify that for us. We think we have enough unique types of dumbbell weights in stock to combine them to match any target total weight, but we're not sure. Can someone count how many possible weights there are from combinations of weights?

Input

Input consists of an integer $1 \leq n \leq 10^5$ for how many weights are given. What follows is n integers w_i where $1 \leq w_i < 2000$ which represents the weight of the ith dumbbell weight. Weights are not necessarily unique.

Output

Output the number of target total weights in the range $[0kg, 2000kg)$ that can be obtained through some combination of the dumbbell weights given. A single dumbbell weight can only be used one or zero times.

Constraints

Time Limit: 1 second
Memory Limit: 1024 mb

Samples

Input:

```
5
1 2 3 4 5
```

Output:

```
16
```

Input:

```
5
1 2 4 8 16
```

Output:

```
32
```

Input:

```
1
100
```

Output:

```
2
```

Input:

```
8
3 4 5 7 10 20 21 40
```

Output:

```
105
```

Common Combo

Consider an array $\{1, 2, 3\}$. There is exactly one subsequence that gives us the sum 1, where we take the subsequence $\{1\}$. Similar,y there is exactly one subsequence that gives us the sum 5, where we take the subsequence $\{2, 3\}$. But there are two different subsequences that give us the sum 3, where we take the subsequences $\{3\}$ and $\{1, 2\}$.

This problem involves finding what sum has the most distinct subsequences that add to it.

Input

Input consists of an integer t where $1 \leq t \leq 100$. For every testcase t, you are given an integer n where $1 \leq n \leq 100$, and n positive integers in the form of an array v where $\sum_{i=0}^{n-1} v_i \leq 5000$

Output

For every testcase, output value that appears most frequent as a sum of subsequences of v. If there are multiple equally frequent sums, output the smallest of the most frequent sums.

Constraints

Time Limit: 1 second
Memory Limit: 1024 mb

Samples

Input:

Output:

```
5
2
1 1
5
1 2 3 4 5
5
1 2 4 8 16
7
1 2 3 3 3 3 3
10
2 3 5 7 11 13 17 19 23 29
```

```
1
5
0
9
58
```

---

10 Advanced Data Structures

"Advanced" is possibly a misnomer. Some of these data structures aren't very complicated, in either their implementation or their usage. We call them advanced here because they relate to much more specific and particular applications. Things like arrays or sets are necessary to solve many basic problems – while things like range-based structures may be necessary to efficiently implement an algorithm or solution, the problems they appear in are never as simple.

10.1 Concepts

10.1.1 Persistent Data Structures

Persistent data structures are ones that allow for not just querying an element, but also being able to query it at a specific time, keeping track of the modifications that have been done to it. For example, if you had a queue with 10 elements and pushed and popped several times, you wouldn't be able to query what the front of the queue is at its initial state with 10 elements. A persistent queue would not only allow you to query it at that state, or after any arbitrary amount of pushes and pops for that matter.

A trivial example of a persistent data structure is simply a list of copies of a complete object for every modification that happens to it. If you add an element to a persistent queue, you keep a copy of the old queue and create a new copy of the list with that element added. Likewise with popping the first element in the queue. You do this 100 times, you now have 100 more copies of the queue at different points in history than you started with. And if you want to query the fron of the queue at some point in history, you can easily refer to the proper state and use that.

Conceptually, this is a persistent data structure – it managed to keep track of its history, and allows us to query this at any time. It should be noted that this is relatively slow, however – copying the queue is $O(n)$ and is required each time a modification is made to the queue, which makes it too slow for most problems that may require a persistent data structure. Similarly, it takes a large amount of memory, and in problems with fairly large numbers of elements or operations this has a strong likelihood of using too much memory.

Ideally, persistent data structures involve representing the data in such a way that the history is implicit to the structure itself – that copying the entire structure is unneeded, and it's preferable to always add to the structure in some way instead.

In our persistent queue's case, a better approach would be to recognize that we only ever can represent a queue as a subarray of all the elements added in order – if we have an array that contains all the elements that will be added to our queue, we can keep track of the index of the element we consider at the front of our queue, and the total size of our queue from that starting position (alternatively, the end position of our queue, or one more than the index of our last element in the queue). Pushing is a simple matter of incrementing our size by 1, while popping involves incrementing our start position by 1 and decrementing our size by 1 as well. For every push or pop we perform on our

persistent queue, we can keep track of the state of the queue as a pair of integers for the start position and the size of the queue:

```
struct PersistentQueue {
    // storage of all elements added in order
    vector<int> elements;
    // history of subarray window
    vector<pair<int,int>> sizes;

    // by default, have an empty queue in the history
    PersistentQueue() {
        sizes.push_back({0,0});
    }

    // put an element into the back of the queue
    void push(int value) {
        pair<int,int> last = sizes.back();

        elements.push_back(value);
        sizes.push_back({last.first, last.second + 1});
    }

    // remove an element from the front of the queue
    void pop() {
        pair<int,int> last = sizes.back();

        // error if the queue is empty
        assert(last.second > 0);

        sizes.push_back({last.first + 1, last.second - 1});
    }

    // check what the front of the queue is
    // state is what point in history it is
    int front(int state) {
        pair<int,int> size = sizes[state];

        // error if the queue is empty
        assert(size.second > 0);

        return elements[size.first];
    }
};
```

10.1.2 Lazy Evaluation

In some cases, we are far more likely to update a data structure than we are to query it. In other cases, we may have many updates in one go before we make any queries. Perhaps

an update involves modifying a huge range of elements in our data structure, when we only perform queries on a small handful of them, and most of the updates are ignored and unused.

For data structures where updating is $O(1)$ it's most likely fine to update as normal. If our updates happen in $O(logn)$ or $O(n)$ or some other non-constant time complexity, we may begin to be concerned with updates. Specifically, we would like to batch our updates together until we actually perform a query that's affected by that update.

Consider a data structure where...

We can make these updates lazy as...

10.2 Range-based Structures

Under Construction

10.2.1 Prefix Sum Arrays

Prefix sums are not a particularly complicated data structure. However, they are useful as the most basic form of range-based data structures. As the name suggests, they deal with sums of integers.

If we are given an array of integers, we can generate a prefix sum array by making each element be the cumulative sum of all elements before it in the original array. For example, given an array $arr = \{1, 2, 3, 5, 10, 3, 2, 5\}$, our prefix sum is $ps = \{1, 3, 6, 11, 21, 24, 26, 31\}$ because each ps_i is equal to $arr_0 + ... + arr_i$.

Generating the prefix sum is a simple process...

Normally, the prefix sum gives us the sum of the array from $[0, i)$. We can obtain the sum of an arbitrary subarray $[l, r)$ by obtaining $[0, r)$ and subtracting $[0, l)$.

While for some problems prefix sums are sufficient (construction in $O(n)$ and queries in $O(1)$), they aren't able to be updated efficiently (you would have to recalculate all sums after the element you're updating, which is $O(n)$).

10.2.2 Sparse Table

Under Construction

10.2.3 Fenwick Tree / Binary Indexed Tree

Fenwick Trees, also commonly known as Binary Indexed Trees (which is also commonly abbreviated to BITs) is a structure that allows calculates the prefix sum that can be

queried in $O(logn)$ and allows for updates in $O(logn)$ as well, compared to the prefix sum array that allows $O(1)$ queries but $O(n)$ updates.

10.2.4 Segment Tree

Segment Trees are conceptually similar to Fenwick Trees, in that they support range queries in $O(logn)$ and updates in $O(logn)$ as well. Segment Trees have the notable benefit of supporting more functions than just summation, though – sums, products, maximum, minimum, GCD, LCM, or xor of an arbitrary range are some examples that appear with some frequency.

The cost of this is some slight overhead – there is more code to them than Fenwick Trees, and they require some additional memory as well, but generally most problems that can be solved with Fenwick Trees can also be solved with Segment Trees.

10.2.5 Wavelet Tree

Wavelet Trees are range based structures that lend themselves to several different operations – namely, counting the number of occurrences of a value in a range, the number of occurrences of between two values within a range, and getting the kth smallest value in a range.

Construction of the Wavelet Tree can be done as...

Counting the occurrences of a value in a range can be done as...

The number of occurrences of any number between two values can be done as...

The kth smallest value in a range can be implemented as...

10.3 Ropes

Ropes are a data structure that can be thought of as an alternative to strings, with some operations that have better time complexity and others that are worse.

Inserting or deleting a character at the end of a string is generally a $O(1)$ operation (amortized), but insertion or deletion at an arbitrary point in the string is $O(n)$. For ropes, the time complexity of insertion or deletion is $O(logn)$ at any arbitrary point, but is also $(logn)$ when inserting or deleting at the end of the string. One of the general costs with this is that querying or modifying a character at an arbitrary index is $O(1)$ in strings, but is an $O(logn)$ operation in ropes.

Some Sums

Sometimes we want to sum a range of an array. Some call this a "range sum". Sometimes, we find some amount of range sums, and sum them together. Some would consider this a sum of some range sums problem.

Input

Input starts with two numbers, n and m where $1 \leq n \leq 10^5$ and $1 \leq m \leq 10^5$. For every n there is a value a_i where $-5000 \leq a_i \leq 5000$. For every m there is a two integers x_i, y_i representing the range $[x_i, y_i)$ where $0 \leq x_i < y_i \leq n$.

Output

Output the total sum of all range sums. In other words, calculate and print $\sum_{i=0}^{m-1} \sum_{j=x_i}^{y_i-1} a_j$.

Constraints

Time Limit: 1 second
Memory Limit: 1024 mb

Samples

Input:

```
5 5
1 2 3 4 5
0 1
1 2
2 3
3 4
4 5
```

Output:

```
15
```

Input:

```
3 6
-1 0 1
0 2
0 2
0 3
1 3
2 3
0 3
```

Output:

```
0
```

11 String Problems

11.1 String Concepts

11.1.1 Anagrams

Anagrams are the string equivalent to a permutation of an array. Two strings are anagrams of eachother if they contain the same elements, but their ordering isn't necessarily the same. For example, "abc" and "bac" are anagrams. Similarly, all anagrams of "abc" are {"abc","acb","bac","bca","cab","cba"}.

We can check whether two strings are anagrams by comparing the count of each character they have in their string:

```
bool isAnagram(string a, string b) {
    // store the number of each character
    int count[26] = {0};

    // increment the count of each character
    for (auto c : a)
        count[(c-'a')]++;

    // decrement the count of each character
    for (auto c : b)
        count[(c-'a')]-;

    // check to make sure they are all 0
    // otherwise, our two strings have a different character count
    for (int i = 0; i < 26; i++)
        if (count[i] != 0)
            return false;

    return true;
}
```

That said, we can make our code smaller (though somewhat less efficient) by simply sorting the strings (so that they are both the lexicographically lowest anagram) and then comparing them.

```
bool isAnagram(string a, string b) {
    // make sure both strings are ordered alphabetically
    sort(a.begin(), a.end());
    sort(b.begin(), b.end());

    // check if the two strings are the same
    // in other words, they have the same amount of each character
    return a == b;
}
```

11.1.2 Lexicographic Order

Lexicographic order of strings is one where strings are sorted based on the alphabetical ordering of their first characters. When strings have the same first character, the alphabetical order of their second characters are considered, and so on. A similar tie-breaker has to appear with shorter strings, where the end of the string is considered first alphabetically.

For example, {"abc", "abd", "ada", "ae", "aeiou", "b"} is a lexicographically ordered array of strings, following the rules above.

Most languages provide some way to check whether a string is lexicographically greater or lesser than another. In C++ and Python, you can simply use `"abc" < "zzz"` which will return `true` since "abc" is lexicographically lesser than "zzz". In Java, we would want to use `"abc".compareTo("zzz")` which would return a negative value to indicate that it's lexicographically lesser (a positive value means lexicographically greater, and a value of 0 means they're the same string).

11.1.3 Palindromes

Palindromes are strings that are the same when they're reversed. For example, "abccba" or "abcba" are both palindromes, whereas "abcde" isn't.

It can be pretty easy to check if a string is a palindrome or not. We can simply check whether the first half of characters are equal to the second half of characters.

```
bool isPalindrome(string str) {
    int n = str.length();

    // for every character in the first half of the string
    // make sure it's the same as its inverse
    for (int i = 0; i < n/2; i++)
        if (str[i] != str[n-1-i])
            return false;
    return true;
}
```

Something to watch out for is that you can have palindromes of even length (such as "abba") and palindromes of odd length (such as "abcba"). Even length palindromes can be thought of as a string concatenated by its reverse ("ab" + "ba"), while odd length palindromes are a string concatenated by its reverse with another character in the middle ("ab" + "c" + "ba"). Often when dealing with palindromes, we may have to generate two different cases to deal with both types of palindromes.

It is also worth noting that a palindrome of length > 2 always contains at least one other palindrome as a substring, which we can obtain by removing the first and last character. "abba" contains "bb" as a palindrome substring, while "abcba" contains "bcb" as a palindrome substring, which itself contains "c" as a palindrome substring (of length 1). It turns out that we can divide the length of a palindrome by 2 rounded down, and get the number of palindrome substrings that are centered on the same point ("abcba"'s length divided by 2 is 2, which is the number of strict substring palindromes with "c" as the center, "bcb" and "c". The same concept applies to even length palindromes). Note that these aren't necessarily the only palindrome substrings possible – "aabbaa" contains {"a", "b", "aa",""bb","abba"} as its unique palindrome substrings, as something to watch out for.

11.1.4 Prefixes

Prefixes are substrings that contain the start of a string. For example, the prefixes of "abcde" are {"a","ab","abc","abcd","abcde"}.

A strict or proper prefix is a prefix that isn't the same length as the full string (equivalently, a prefix that doesn't contain the end). Using the above example, proper prefixes would be {"a","ab","abc","abcd"} which notably excludes the original string "abcde".

11.1.5 Substrings

Substrings are the string equivalent to subarray – it is the original string with some amount from the start and end removed. "abcd" has the substrings {"a", "b", "c", "d", "ab", "bc", "cd", "abc", "bcd", "abcd"}. Note that the original string "abcd" is a valid substring of itself, it just removes 0 characters from the start or the end.

Sometimes you will see a "strict" substring. This is the same as a regular substring except the original string isn't counted. This makes sure that substrings will always remove at least one character from either the start or the end, and that the length of each strict substring will be less than the original string.

11.1.6 Suffixes

Opposite to prefixes, suffixes are substrings that contain the end of a string. For example, the suffixes of "abcde" are {"e","de","cde","bcde","abcde"}.

Strict or proper suffixes are also similar to prefixes, in that they're suffixes that aren't

the same length as the original string (or are suffixes that don't contain the start of the string).

11.2 Pattern Matching

One of the common tasks needed for string problems involves pattern matching – determining whether or not a substring exists in the larger string altogether.

There are a fair amount of possible variants and alternatives to pattern matching. The simplest is simply "does a given substring exist in a given larger string" with no other complications. In this case, if we have some text "abcdef" and we're looking for the pattern "def", we can confidently say the pattern does exist. A common variant is to find the specific location of a matched pattern, where the previous example would start its match at position 3 in the original string. Sometimes it's sufficient to find one match even if there may be many, sometimes it's necessary to find every valid match.

11.2.1 Regex

Note: the term 'regex' is a common shorthand for 'regular expressions', but there is a large difference between modern regex engines and true regular expressions. Briefly, that regular expressions are regular in a formal languages sense, while regex engines include many features that make them not regular expressions. Because of this, when we say "regex" we mean the language used in modern regex engines, while "regular expressions" is more akin to theoretical computer science. Confusingly, we might also say 'regex expression' when we want to refer to a specific pattern of regex, while we don't use 'regular expression expression'.

Pattern matching can be solved easily using regex, which is a domain-specific language that defines patterns of text. This can easily be used to search for a specific pattern, which can be as simple as a raw string such as `"pattern"`. For example, consider in python:

```
import re

if (re.search("pattern", "string to do pattern matching on")):
    print('yes')
else:
    print('no')
```

The main advantages of regex are that they can be very quick to code, and they can support more advanced patterns. For example, if you wanted to match either "Hello" or "World", you could use `"(Hello)|(World)"` as the pattern. You could search for any consecutive pair of vowels with `"[aeiou]{2}"`, or any word that ends in s with `"\b.+s\b"`. Regex can define fairly complex patterns to search for in only one short line. We won't go into detail here about regex's syntax or how to write complicated patterns with it, but it can be a useful read.

This comes with some hefty downsides though: namely that regex matching is significantly slower than other methods of pattern matching (to the point of needing to be very careful, because exceeding the timelimit with regex is very common), that it requires an entire different syntax to write and is generally harder to understand once written, and that if the regex engine doesn't do exactly what you need then it can't be easily modified to work.

11.2.2 Knuth-Morris-Pratt

Consider the naive approach: for every position i in the original string s with the search pattern p, we check if $s[i] = p[0]$, then if $s[i+1] = p[1]$, and so on until we go through the either the full pattern (in which case we found a match) or the full original string (in which case we didn't find a match).

That approach is not ideal, because we can filter out an invalid match early in many cases. For example, consider trying to match "abcabd" into "abcabcabd". When we fail to match at position 0, we've already made comparisons for positions 1 and 2 and can know that our pattern can't exist there either. This results in an unnecessary (and potentially large) slowdown.

Knuth-Morris-Pratt (commonly abbreviated as KMP) is an algorithm that can apply the necessary information to skip starting positions we may already know are invalid. We do have to do some pre-processing on the pattern string first though.

To begin, we generate an array 'l' where $l[i]$ is the length of the longest proper prefix of the substring $p[0:i]$ that's also a proper suffix. In other words, for every i, $l[i]$ is the largest value m where $m < i$ and $p[0:m] = p[i-m:i]$. The l-array of "aaa" will be $[0, 1, 2]$ because at $i = 0$ there is no common prefix/suffix, at $i = 1$ we have the substring "aa" and the longest proper prefix that's also a suffix is "a", and at $i = 2$ our substring "aaa" has the longest proper prefix/suffix of "aa". Meanwhile, the l-array of "abc" is $[0, 0, 0]$ because it never shares a common proper prefix and suffix.

Now that we have the l-array, we can actually perform our pattern matching. Let's consider the example of trying to pattern match "abcabd" into "abcabcabd" again. We begin naively, comparing $s[0] = p[0]$ through to $s[5] = p[5]$ and notice we have a mismatch. Rather than shifting the string by 1 and recomparing the full pattern, we instead shift the string by 3 and begin comparing the pattern starting at $s[3+2] = p[2]$.

11.2.3 Boyer Moore

11.2.4 Z-Algorithm

The Z-Algorithm generates an array Z where $Z[i]$ is the length of the longest common prefix between a string and its substring beginning at position i. Explained practically, $Z[i]$ is the number of characters that are the same starting from $s[0]$ and $s[i]$.

While this is a general string algorithm with several applications, one application is in

string matching. Given text s and a pattern p (and some character not present in either, which we'll call x which can be something like `'\n'` that is uncommon to be needed in a problem), we generate the z-array of $p + x + s$ so that they're all concatenated into one string.

Now, any time the length of p appears inside the z-array, we know that pattern has a match. We can convince ourselves of this pretty easily – because of x we should never have a value that's greater than the length of p, and any value in the z-array that appears after x is how many leading characters are the same in p.

11.2.5 Aho Corasick

Aho Corasick is an algorithm for pattern matching that's designed to search for multiple patterns simultaneously. Aho Corasick is based on construction of a finite state machine that represents all patterns simultaneously, and allows you to iterate over the original string with each character being used for the state transitions, and noting every accepted state as a successful match.

11.2.6 Pattern Matching Applications

Sometimes, we may want to do some creative applications using our pattern matching algorithms.

Consider the regex expression `"abc.*xyz"`, which essentially matches any substring that starts with "abc" and ends with "xyz", including "abcxyz". We don't actually need regex to solve this.

11.3 Subsequence Matching

11.4 Hashing

Comparing strings with eachother to see if they're the same or not is a $O(n)$ operation – you can only tell if two strings of the same length are different by checking if any characters are different, and as a result you can only know tht two strings are the same if you check each character. For problems that involve lots of string equality checks, this might be too slow.

Hashing is an option that allows us to treat strings as integers, where the comparison between them is a $O(1)$ operation.

11.4.1 Basic Polynomial Hashing

11.4.2 Operations on Polynomial Hashes

11.4.3 Collisions

When we try to represent strings as integers, we run the risk of representing different strings with the same integer.

A simple example of this happening is that, if our integer is 32 bit, we can store 2^{32} possible values. Let's say we want to try hashing $2^{32}+1$ different strings. Logically, there is going to be at least one value that gets repeated twice.

A more practical example, let's consider a case where we have a polynomial hash with the function `v = ((s[i] - 'a' + 1) + v * 27) % 31` that loops through the string s. With this function, the strings `"a"` and `"zl"` both have the hash value 1. In practice we'll rarely want to perform the hash modulo a small number like 31, but the same concept applies with larger moduluses or when relying on buffer overflows.

11.5 Prefix/Suffix Arrays

Under Construction

11.6 Distance

Under Construction

11.6.1 Diff

Under Construction

11.6.2 Levenshtein

Under Construction

Pattern Matching:

1. https://open.kattis.com/problems/avion
2. https://open.kattis.com/problems/fiftyshades
3. https://open.kattis.com/problems/deathknight
4. https://open.kattis.com/problems/ostgotska
5. https://open.kattis.com/problems/geneticsearch
6. https://open.kattis.com/problems/scrollingsign

7. https://open.kattis.com/problems/stringmatching
8. https://open.kattis.com/problems/stringmultimatching
9. https://open.kattis.com/problems/messages

Suffix Arrays:

1. https://open.kattis.com/problems/suffixsorting
2. https://open.kattis.com/problems/substrings
3. https://open.kattis.com/problems/dvaput
4. https://open.kattis.com/problems/burrowswheeler

Hashing:

1. https://open.kattis.com/problems/typo
2. https://open.kattis.com/problems/rimstyrka (proper solution isn't hash based, but a hashing solution works)

12 Graph Problems

12.1 Concepts

12.1.1 Nodes and Edges

Graphs are a construct of two basic concepts: **nodes** (or vertices, or points) represent some object or position, and **edges** which represent some direct connection or relation between nodes.

A common example is cities and roads. You have some number of cities represented by nodes, that have roads connecting them represented by edges. We can do some interesting things with a graph of cities and roads, such as finding the distance between two given cities and how many other cities we need to pass along the way (which we will cover in the pathfinding section), or finding the minimum number of roads we need in order to get from any one citity to any other city (which is the concept of spanning trees, also covered later).

12.1.2 Representations

There are multiple ways to represent graphs when we talk about actually implementing them in code.

Edge lists are likely the most simple representation of a graph, in which we have an array store each edge in the graph. Nodes are implicit by their value – an edge is represented as a pair of integers a and b (possibly with additional information like weight) that defines a connection between the ath node and the bth node.

```
struct Edge {
    int a, b;
};

struct EdgeList {
    // storing edges
    vector<Edge> edges;

    // add edge
    void add(int a, int b) {
        edges.push_back({a, b});
    }
};
```

Unfortunately, edge lists are relative uncommon and aren't the ideal graph representation for the majority of graph algorithms. This is because one of the most common things we want to do with graph algorithms is operate on edges connected to a specific node. With edge lists, this is $O(n)$ – we have to traverse through the entire list to find what edges

are connected to a specific node. We can optimize this somewhat (sort the edge list so all nodes appear in order in $O(nlogn)$, then binary search for a specific node in $O(logn)$, for example) but ultimately we can do better with other graph representations for most use cases.

Adjacency lists are a graph representation in which we have an array of each node, where each node stores its edges. This fixes our issue with edge lists, because now we can index directly into a node and operate directly on all of its incident edges. This can either be as an array of `Node` objects where each `Node` contains an edge list, or as an array of edge lists. Even simpler, we can have an array where each index is the start node of the edge, and store a list of end node values:

```
struct AdjList {
    // storing edges
    vector<vector<int>> edges;

    // constructor, n is number of nodes
    AdjList(int n) : edges(n) {}

    // add edge, a and b are 0-indexed
    void add(int a, int b) {
        edges[a].push_back(b);
    }
};
```

Now, we may notice that we improved our accesses for our start node from $O(n)$ to $O(1)$ by using an array instead of a list, with the index equal to the start node of a given edge. Unfortunately, since adjacency lists still use lists, trying determine whether an edge between a specific start node and a specific end node exists is potentially expensive as well – we would have to iterate through every edge indicent with the start node to find if that specific end node exists. So you may ask, why not use an array again, so that we have a 2D array where end nodes are our index for the inner array?

As it turns out, this is called an **adjacency matrix**. We maintain a 2D array of boolean values, so that we can quickly check whether an edge between a and b exists by checking the value of `edges[a][b]`.

```
struct AdjMatrix {
    // storing edges
    vector<vector<bool>> edges;

    // constructor, n is number of nodes
    AdjMatrix(int n) : edges(n) {
        for (int i = 0; i < n; i++)
            edges[i] = vector<bool>(n);
    }

    // add edge, a and b are 0-indexed
    void add(int a, int b) {
        edges[a][b] = true;
    }
};
```

Now, is any one of these representations strictly better than the others? Not at all! They all have their strengths and weaknesses, namely:

- Only adjacency matrices can check if an edge between a and b exists in $O(1)$. Edge lists need to iterate through the entire list to find if that edge exists, while adjacency lists have to iterate through the list of edges incident to a.
- If there are only a couple of edges incident to node a, only adjacency lists can iterate through those incident edges quickly. Edge lists don't have easy access to find node a and have to search through the entire list, while adjacency matrices have to look through every value of `edges[a][0..n]` to find incident edges.
- If there are only a couple of edges in general and you need to iterate through every edge, only edge lists can iterate through them quickly. Adjacency lists require you to iterate through its whole array of size n, and adjacency matrices require you to iterate through its entire 2D array – to go through n^2 different values.

To generalize, using a list is good if your graph doesn't have a large amount of edges compared to the number of nodes, and you want to iterate through edges rather than access specific ones. Comparatively, using an array is good if your graph has lots of edges relative to the number of nodes, and/or you don't need to iterate through edges and would prefer the $O(1)$ lookup time.

In practice, edge lists are usually not the preferred method – there are relatively few algorithms or problems that are best implemented by iterating through every edge in arbitrary order. Adjacency lists and adjacency matrices appear fairly frequently, so it's important to know both of them well.

12.1.3 Directed and Undirected

One trait of graphs is whether they're directed or undirected.

In a directed graph, edges are one way – an edge from a to b can be traversed in that order, but you could not go from b to a using that edge.

In contrast, an undirected graph has edges that go in either direction. The edge a, b also represents an edge b, a in that traversal can be done with either a to b or b to a.

Our graph representations that we developed above only supported directed graphs. However, when we add an edge a, b, we can also add the edge b, a in order to make our representations be equivalent to undirected graphs.

12.1.4 Cyclic and Acyclic

A **cycle** is a path where the start and end node are the same, but no other edges are traversed more than once (a path that does repeat a non-start node is called a **circuit** instead). In plainer terms, without traversing the same edges or any nodes multiple times (except the start node exactly twice), cycles are what you find when you end up at the same node as you started.

We call a graph that contains cycles **cyclic**. In comparison, a graph with no cycles is **acyclic**. This is relevant because sometimes algorithms work well on acyclic graphs but not on cyclic ones, or alternatively fundamentally only matter for cyclic graphs (such as cycle finding algorithms).

12.1.5 Connected and Disconnected

Two nodes are considered **connected** when there exists some path between them. Opposite to this, two nodes are **disconnected** if there isn't a path between them, and by definition no amount of graph traversing from one node will lead you to the other.

When we're talking about the graph as a whole, a **connected graph** means that every node is connected to every other node. Likewise, a **disconnected graph** means that at least one pair of nodes is not connected.

In directed graphs, the concept of just being connected isn't always enough. While there might be enough edges in the graph to fully connect every node, the direction of the edges matters and may mean there isn't a valid path between every node. We have to introduce the concept of a **strongly connected graph** where every node has a valid path to every other node when abiding by direction of edges as well.

In practice, a strongly connected graph not only needs to be cyclic, there needs to be a circuit that traverses every node (while we haven't described Hamiltonian cycles yet keep in mind that this *is not* a Hamiltonian cycle, because we do allow traversing nodes multiple times here).

An informal proof is relatively simple: in order to be strongly connected, we have to be able to pick an arbitrary node and have a valid path to every other node as well, and if we had chosen a different starting node then we would also have to have a valid path from every other node back to our originally chosen arbitrary node. In other words, for every pair of nodes a and b in the graph, there needs to be a valid path from a to b and from b to a, which is a circuit by definition.

The **transitive closure** of a graph is a transformation in which edges are created between any connected nodes. For example, if a graph normally contains the edges $\{0, 1\}, \{1, 2\}$, there is no edge for $\{0, 2\}$ despite there being a valid path between them. In the transitive closure, we would create the edge $\{0, 2\}$ so that every node has an edge directly to every other node it's connected to.

Often, the transitive closure of a graph will be stored as an adjacency matrix, because it generally results in a large amount of edges (with a lot of redundancy).

12.1.6 Complete

A graph is considered **complete** if every node contains an edge to every other node. As a result, the minimum path length from one node to another is always 1 in a complete graph.

A complete graph with n nodes will contain $\frac{n(n-1)}{2}$ edges. While often assumptions can be made about complete graphs that simplify problems immensely, the large amount of edges may also make common algorithms expensive and ineffective.

12.1.7 Weighted

Edges can also have **weights**. These are numerical values associated with edges, that usually represent a "cost" for traversing that edge.

A graph that does not include weights (or equivalently, one where every edge has a weight of 1) is considered an unweighted graph. A graph with edges that include different weights is considered a weighted graph.

12.1.8 Grids

Grids are a specific case of graphs, but are extremely common. You are given a (usually 2D) array where. In graph form, each element represents a node, has an edge between each of its neighbors.

in Euclidean geometry the distance between two points will be the length of the line segment directly connecting them (in other words, $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Often, the concept of **Manhattan distance** or **taxicab distance** will be used instead, in which the path cannot be diagonal and must consist of lines going either horizontally or vertically (and as a result, $d = |x_2 - x_1| + |y_2 = y_1|$).

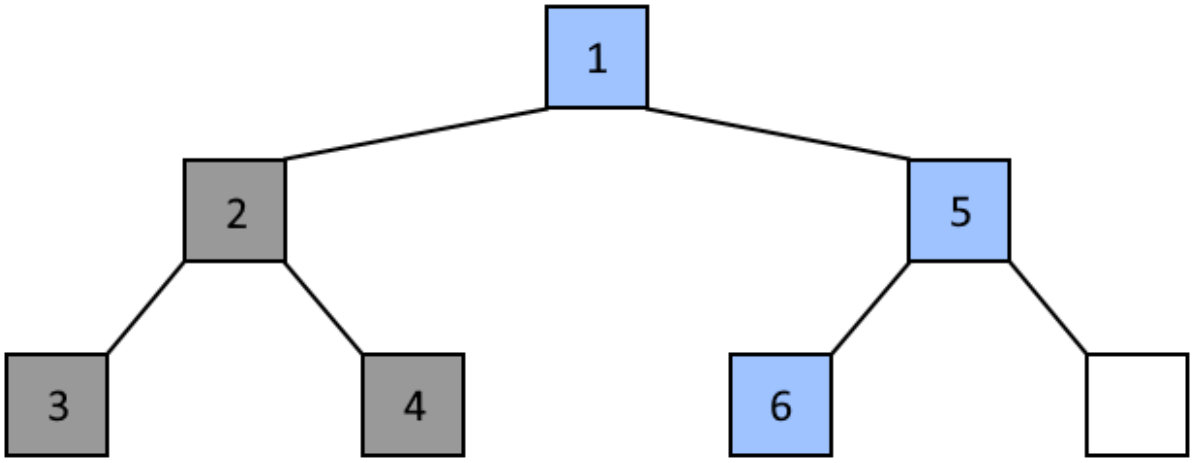
12.2 Pathfinding

One of the most common tasks with graphs is to find a path between two nodes, or the distance between them. As we will see, traversing the graph in this manner is a common subtask in several other, more complicated graph algorithms as well.

It's important to know that, in general, there isn't any single pathfinding algorithm that you should always use. Each have their own strengths and weaknesses that are only relevant in certain circumstances, or come with specific restrictions that make them only worthwhile on specific types of graphs.

12.2.1 Depth First Search

Out of all the pathfinding algorithms, the depth first search is the easiest conceptually. Starting at a specific node, we explore a path as long as possible without revisiting the same node twice (until we either find our target end node, or we reach a point where we have no more unvisited neighbors to explore). If we don't find the end node, we try again with a slightly different path – the last time we had multiple unvisited neighbors to a node (and we ended up going down the first path), let's no try the next neighbor and attempt that path.



In the above image, blue is the proper path from 1 *to* 6 *while grey is explored nodes that are not involved in the proper path.*

In the above example, we start at node 1 and want to find whether a path exists to node 6. A DFS will begin by checking $\{1\}$, $\{1, 2\}$, $\{1, 2, 3\}$, $\{1, 2, 4\}$, $\{1, 5\}$, $\{1, 5, 6\}$, and at that point can say there is a path between 1 and 6.

We can model this using a stack data structure, where we add all unvisited neighbors of a node into the stack. If at any point we encounter the end node, we know there's a valid path and can return `true` then. If our stack is empty, we know we've checked every node connected to our starting position without finding the end node.

```
bool dfs(AdjList& graph, int pos, int end) {
    // set up data structures
    vector<bool> visited(graph.edges.size());
    stack<int> s;

    // add our start to the stack and mark as visited
    s.push(pos);
    visited[pos] = true;
    while (s.size() > 0) {
        // get the current node
        int cur = s.top(); s.pop();

        // for every edge from the current node
        // we check if it's already visited
        for (auto v : graph.edges[cur]) {
            if (!visited[v]) {
                // if the node is the end, return
                if (v == end)
                    return true;

                // mark as visited and add to stack
                visited[v];
                s.push(v);
            }
        }
    }

    // we were unable to reach the end from pos
    return false;
}
```

A keen eye can notice that using a stack like that is very similar to recursion – in fact the code for a DFS becomes significantly shorter when we convert it to use recursion instead:

```
// we need to pass in visited from outside the method now
bool dfs(AdjList& graph, vector<bool>& visited, int pos, int end) {
    // base case
    if (pos == end)
        return true;

    for (auto v : graph.edges[pos]) {
        if (!visited[v]) {
            visited[v] = true;
            // recursive check, using v as our new starting pos
            if (dfs(graph, visited, v, end))
                return true;
        }
    }

    // we were unable to reach the end from pos
    return false;
}
```

12.2.2 Breadth First Search

Breadth first search is similar to depth first search, except instead of trying to search through the first path possible (prioritizing *depth*) we try to search through every neighbouring node (priorizing *breadth*).

The simplest approach to implementing this is to take our stack-based implementation for DFS and use a queue instead. This means that, instead of trying the most recently added node for the next iteration as we would in a DFS, we will use the oldest node for the next iteration.

12.2.3 Dijkstra's Algorithm

DFS and BFS, as can be noted, doesn't consider weighted graphs – if we exit at the first time we encounter the target end node, DFS will give us some valid path and BFS will give us a path with the minimum number of edges. In neither case are these edges necessarily with the minimum total weight (in BFS's case, we may find a solution in one edge with weight of 100, when the minimum total weight would be two edges each with weight 1).

Dijkstra's algorithm is a solution to this. It's also implemented like DFS or BFS except we use a priority queue, where the weight of our current path is how we base the priority (we consider lower weights before higher weights). Additionally, rather than storing whether a node is visited or not, we store the current best (minimum) weight of a path we've found (defaulting to infinity, or some huge number that won't be encountered normally). Using this, when we consider the neighbors of an node, rather than checking if that node is already visited we check if our current node's best weight + the weight of our next

edge, and add that to our priority queue if so.

12.2.4 A* Algorithm

The A* algorithm can be considered a generalization of Dijkstra's, in which we use a heuristic function to (hopefully) improve performance.

Under Construction

12.2.5 Bellman-Ford Algorithm

A restriction with Dijkstra's is that it can't handle negative weights properly – there are specific cases where a negative weight can break the assumptions that Dijkstra's makes.

The Bellman-Ford algorithm can work with this, both by being able to handle negative weights (when there are no negative cycles) and being able to detect negative cycles (in which case there is no shortest path).

Under Construction

12.2.6 Floyd-Warshall Algorithm

So far the algorithms we've considered have been single-source shortest path algorithms – namely that we have a specific starting node. What if we want to determine the shortest path distance between any two arbitrary points? We would need an all-pairs shortest path algorithm instead.

Floyd-Warshall meets these requirements. It works with negative weights as well, assuming that there are no negative weight cycles.

Under Construction

12.2.7 Path Reconstruction

For DFS, BFS, Dijkstra's, etc. we can include an auxillary array that represents some data about each node. In order for the algorithms to work, we already include an array that determines whether a node has been visited for DFS and BFS, and for Dijkstra's we include an array that determines the minimum weight of a path we've needed so far.

One of the most common ones is a *parent* array, in which every node stores what node came before it in the path. This is useful for path reconstruction, where we want to determine the path taken to get to the end node.

When we set our node as visited (for DFS and BFS) or the weight required (for Dijkstra's) we also set the parent node. Then, after our algorithm is done, we check the parent of

the end node, then its parent, then the parent of that node, and so on until we reach the source node. The path is this list of parents in reverse.

12.3 Flood Fill

Flood fill is an algorithm in which, given a graph (often a grid) and a specific node (in a grid, often a coordinate) and we set an entire region of alike elements (or any specific criteria that we need) as filled.

A standard example is in image editing software, the idea of a "fill" tool. Specifically, where we click on a part of an image and it changes the color of that pixel and any connected pixels of the same color.

Here is an example recursive implementation on a grid of `'.'` and `'#'` characters:

```
// grid helper
const int dx[] = {1, 0, -1, 0};
const int dy[] = {0, 1, 0, -1};

// given a 2D char array "grid" of '.' and '#'
// fill any continuous region of '.' containing grid[y][x] with '!'
void floodfill(vector<string>& grid, int w, int h, int x, int y) {
    // set the current coordinate as visited
    grid[y][x] = '!';

    // iterate through the neighbour in each direction
    for (int d = 0; d < 4; d++) {
        int x2 = x + dx[d];
        int y2 = y + dy[d];

        // if that neighbour exists and is a '.', floodfill that too
        if (x2 >= 0 && x2 < w && y2 >= 0 && y2 < h && grid[y2][x2] == '.')
            floodfill(grid, w, h, x2, y2);
    }
}
```

There are many additional variants to this. If we need to preserve the original grid, instead of setting nodes as visited by turning them from `'.'` to `'!'`, we can use a 2D boolean array to mark elements as visited or unvisited.

You can notice how similar this is to a depth first search. One way to view this implementation is as DFS'ing through every possible path from the source node, in order to determine every node it's connected to. While we normally set nodes as visited in a DFS simply to avoid operating on nodes we've already considered, the intention of flood fill is solely to find these visited nodes.

Following this logic, we can use alternatives like BFS as well, which should function near equivalently. A recursive DFS-like solution was chosen for simplicity because the code is

relatively short.

A common application of floodfill is to count how many distinct regions there are. For example, using the above code, we can count how many regions of connected `'.'` characters exist in a grid with a function such as:

```
int countRegions(vector<string>& grid, int w, int h) {
    int count = 0;

    // iterate over every point
    for (int i = 0; i < h; i++) {
        for (int j = 0; j < w; j++) {
            // if the point is a '.' we increment count and floodfill
            // so that anything else in the region is no longer a '.'
            if (grid[i][j] == '.') {
                count++;
                floodfill(grid, w, h, j, i);
            }
        }
    }
    return count;
}
```

12.4 Union-Find

Union-Find, which also goes by the name of *disjoint-set*, is a data structure that groups connected nodes together into different subsets through operations called **union** (which connects two nodes together, akin to creating an edge) and **find** (which determines which subset a node belongs to). As a result, Union-Find is well suited for any problem in which you have to determine if two different elements belong to the same set – this sees applications both by itself in many problems, and as an important part of different graph algorithms.

12.5 Minimum Spanning Trees

Many graphs have redundant edges – between nodes A and B there may be several different paths. A spanning tree of a connected graph is one in which we use a subset of edges (whether we think of it as removing redundant edges or building up only the necessary edges depends on the algorithm we use) so that every pair of nodes has exactly one unique path between them.

A minimum spanning tree (commonly going by the acronym MST) is a variant of a general spanning tree, where we want to obtain the minimum total weight among all spanning trees. This is not necessarily unique either, as there easily could be multiple possible minimum spanning trees.

12.5.1 Kruskal's Algorithm

In order to build a minimum spanning tree, we greedily take the edge from the graph with the minimum weight that does not form a cycle within our work-in-progress MST.

Algorithmically, we can do this as:

1. sort all edges in the graph by weight
2. iterate through the sorted edges
3. if the start and end node of that edge are not connected (which we can determine with Union-Find), we connect those nodes and add the edge to our MST.

When this process finishes, we will obtain a minimum spanning tree.

We can exit early from the above process if we already have $n - 1$ edges in the MST, where n is the number of nodes. At that point, we would have every node belong to the same set, and as a result any more edges we consider will not be added because they inherently will add a cycle.

12.5.2 Prim's Algorithm

Prim's algorithm, rather than dealing with every edge immediately and choosing between them, will instead keep track of what nodes are currently represented in the MST and only deal with edges connected to them. While Kruskal's can add edges fairly randomly and won't necessarily have them be connected, Prim's will consistently grow from a single node.

Algorithmically, we:

1. choose an arbitrary node as our start node
2. add that node into a set A and all edges connected to that node into a set B
3. find the minimum weight edge in B that involves a node not in A, and add that edge to our MST
4. repeat from step 2 until we have every node included

Conceptually, this is very similar to Dijkstra's algorithm for pathfinding. They aren't identical: Prim's operates on the minimum weight edge only, while Dijkstra's uses the minimum total weight (minimum sum of all weights along a path from the source node). As a result, despite very similar looking code, the outputs of these two algorithms can vary significantly.

12.6 Strongly Connected Components

Earlier in this chapter we defined the concept of strongly connected graphs: directed graphs where every node can reach every other node. **Strongly connected components** (or SCCs) are subgraphs in a directed graph that are strongly connected. We can decompose a graph into its strongly connected components, where each node of the graph represents a strongly connected component of the original.

This concept sees some use in various problems. For example, if we want to consider whether a path exists between two nodes in a directed graph with many cycles, then it may be useful to check on the graph of strongly connected components instead. Not only can we immediately know two nodes are connected if they belong to the same strongly connected component, but also the graph we need to check connectivity on is significantly smaller when we're only considering strongly connected components.

12.6.1 Kosaraju's Algorithm

Kosaraju's Algorithm is a conceptually simple algorithm for finding strongly connected components. The process is to:

1. DFS from each node that hasn't already been considered, storing the order that nodes are finished DFS'ing
2. DFS again for each node that hasn't already been considered, in reverse of the order found above, using the transpose of the graph (the same graph, but with all directions flipped). The reachable nodes from each DFS is a strongly connected component.

Let's start by defining the DFS we're going to use for this.

```
// DFS method used for Kosaraju's
void DFS(AdjList& graph, int i, vector<bool>& visited,
  vector<int>& accesses, bool isPreorder) {
    visited[i] = true;

    if (isPreorder)
        accesses.push_back(i);

    for (auto e : graph.edges[i])
        if (!visited[e])
            DFS(graph, e, visited, accesses, isPreorder);

    if (!isPreorder)
        accesses.push_back(i);
}
```

As a note about its design: often you will see implementations of Kosaraju's algorithm define two different DFSes, one for the first iteration (that adds elements to the `order` list in a postorder traversal) and another for the second iteration (that adds elements to the `ans.edges[count]` list in a preorder traversal). We combine the code here into one method to save a small amount of space, and to make it more clear how similar the two steps of the algorithm are.

With that made, we now just need to do the setup required in order to apply it:

```cpp
// return each component as a 2D array
AdjList Kosaraju(AdjList& graph) {
    int n = graph.edges.size();
    AdjList transpose(n), ans(0);

    // build transpose by reversing directions
    for (int i = 0; i < n; i++) {
        for (auto e : graph.edges[i]) {
            transpose.edges[e].push_back(i);
        }
    }

    // DFS from each unvisited node
    vector<bool> visited(n);
    vector<int> order(0);
    for (int i = 0; i < n; i++)
        if (!visited[i])
            DFS(graph, i, visited, order, false);

    // DFS in the opposite way
    fill(visited.begin(), visited.end(), false);
    int count = 0;
    for (int i = n-1; i >= 0; i-) {
        if (!visited[order[i]]) {
            ans.edges.push_back(vector<int>());
            DFS(graph, i, visited, ans.edges[count], true);
            count++;
        }
    }

    return ans;
}
```

Some of this work can be done outside of the method – for example, when we first create the graph, it can be useful to generate the reverse of the graph there too, instead of having it done within our algorithm.

12.6.2 Tarjan's Algorithm

Under Construction

12.7 Maximum Flow

Under Construction

12.8 Bipartite Graphs

A bipartite graph is one in which nodes belong to two mutually exclusive sets (usually these are unnamed, but we'll call them A and B to easily distinguish them from each other). Edges in a bipartite graph can only exist between two nodes if one belongs to set A and the other belongs to set B – a bipartite graph does not allow for an edge between two nodes of the same set. More than that, every node in A has to have an edge that connects it to a node in B, and vice versa.

12.8.1 Determining Bipartite Graphs

As it turns out, determining whether a bipartite graph is equivalent to whether or not there is a valid 2-coloring of the graph. That is, can we color nodes in a graph using only two colors without any same colors being adjacent to eachother?

To begin, we pick an arbitrary node. We set its color to red (our choice in color is arbitrary, and in truth we'll likely implement it with a boolean or an integer). We then set all its neighbouring edges to blue. Then we set all their neighbouring edges to red, as long as they haven't already been colored blue.

This coloring neighbours process continues until we either:

- successfully iterate through every edge in the graph and colored every node without issue, in which case we have found a valid 2-coloring (there is one other valid 2-coloring, which is simply the direct inversion).
- attempted to color a node that was already colored with the opposite color, in which case there is no such thing as a valid 2-coloring.

12.8.2 Maximum Bipartite Matching

Under Construction

12.8.3 Maximum Weight Bipartite Matching

Under Construction

12.9 Eulerian Paths

A Eulerian Path is a traversal of a graph in which every edge is visited exactly once. By consequence each node will also be visited one or more times.

Similarly, a Eulerian Cycle is a traversal of a graph that visits every edge exactly once with the added restriction that it's a cycle and has the same start and end node.

It is relatively easy to verify whether or not a Eulerian Path or Cycle exists. The following conditions must be satisfied:

- The graph must be connected
- The degree of each node must be even

12.9.1 BEST Algorithm

Under Construction

12.9.2 Hierholzer's Algorithm

Under Construction

12.10 Hamiltonian Paths

A Hamiltonian Path is similar to a Eulerian Path except that it deals with nodes rather than edges – it's a traversal in which each node is visited exactly once, and by consequence each edge is either visited once or not visited at all.

A Hamiltonian Cycle follows the same convention where we add the restriction that the end node of our traversal also has to be our source node.

Unlike Eulerian Paths and Cycles, verifying whether a Hamiltonian Path or Cycle exists or not is an NP-complete problem for the general case. There are special cases however:

- a complete graph with n nodes has $(n-1)!/2$ different Hamiltonian Cycles.
- By Dirac's Theorem, a simple graph with at least 3 nodes contains a Hamiltonian Cycle if every node has a degree of at least $\frac{n}{2}$ where n is the number of nodes in the graph.
- By Ore's Theorem (which generalizes Dirac's Theorem), a simple graph with at least 3 nodes contains a Hamiltonian Cycle if, for every pair of nodes in the graph, the sum of their degrees is at least n where n is the number of nodes in the graph.

13 Number Theory Problems

13.1 Binary Exponentiation

Consider the naive way to calculate a^n (where n is an integer), we simply get the product of n as. In terms of code, we can simply initialize a variable to 1 and then have a for-loop that multiplies our variable by a a total of n times. This works fine for 0^n where our answer should be 0, and a^0 where our answer should be 1, but doesn't work for 0^0 which is undefined (for simplicity's sake, we'll ignore 0^0).

```
int power(int a, int n) {
    int value = 1;
    for (int i = 0; i < n; i++)
        value *= a;

    return value;
}
```

Unfortunately, this is relatively slow. In many problems or algorithms where you need to use exponentiation, calculating it in $O(n)$ is far too slow.

Binary exponentiation is the idea of calculating an arbitrary integer power as a product of different values of a^{2^n}. Specifically, we rely on two properties:

- We can represent a number as a sum of powers of 2 based on its binary representation . Consider the decimal number 10, which has a binary representation of 1010_2. We can tell that $10 = 2^1 + 2^3$ from its binary representation.
- $a^{n+m} = a^n * a^m$. Rather than calculate the larger a^{n+m} we can instead calculate it as the product of smaller powers that add to $n + m$.

When we combine these together, we can notice that $a^{10} = a^{2^1+2^3} = a^{2^1} * a^{2^3}$. What makes calculating this very fast is noticing that $a^{2^{i+1}} = (a^{2^i})^2$, so that we can quickly iterate through values of a^{2^i}:

```
int power(int a, int n) {
    int value = 1;
    // as long as there's more bits set
    while (n > 0) {
        // if bit i is set, multiply value by a^2^i
        if (n % 2 == 1)
            value *= a;
        // we store the next a^2^i in a
        a *= a;
        // divide by 2 to handle the next bit
        n /= 2;
    }
    return value;
}
```

The value in this is that we have significantly reduced our runtime. Rather than calculate in $O(n)$, we calculate in $O(log_2 n)$. Now, rather than $2^{2^{31}-1}$ requiring over 2 billion iterations of a for-loop in the naive method, it takes a mere 31 iterations of a while-loop in our binary exponentiation method.

13.2 Primes

Prime numbers are a recurring concept inside number theory, and is foundational to a lot of more advanced applications.

In short, prime numbers are integers > 1 that cannot be evenly divided by any other integer > 1. In other words, a prime number is a number that can't be represented as the product of more than one prime number. For example, $2, 3, 5, 7, 11, 13, 17, 19, 23, ...$ are prime numbers, because none of the following can be representated as the product of other prime numbers.

The opposite of a prime number is a composite number, that is evenly dividable by some number of prime numbers. $4 = 2^2, 6 = 2 * 3, 8 = 2^3, 9 = 3^2, 10 = 2 * 5, 12 = 2^2 * 3, ...$ are examples of composite numbers. Every composite number has exactly one representation as the product of prime numbers.

13.2.1 Prime Sieve

A prime sieve, also known as the sieve of Eratosthenes, is a common way to find all prime numbers up to a specific size. It relies on two basic observations:

- if a number is prime, all of its multiples are not
- if a number is composite, it must be a multiple of at least one prime smaller than it. By looking at the inverse of this statement, if a number is prime then there must not be equal to any multiple of a prime number smaller than it.

A prime sieve is a pretty logical extension of the above observations – we say 2 is prime,

then we mark every multiple of 2 as composite. We check the next number, 3, which wasn't marked as composite and therefore is prime, and we mark every multiple of 3 as composite as well. The next number, 4, was marked as composite, so we know that it's not prime (and don't need to do anything extra, since every multiple of 4 is already marked as composite). 5 isn't marked, so we treat it as prime and mark all its multiples. 6 is composite, 7 is prime, 8 is composite, 9 is composite, etc.

In terms of code, we can express this as an array:

```
// generate the prime sieve
// if n is prime, sieve[n] = 0
// if n is composite, sieve[n] = 1
vector<int> primeSieve(int maximum) {
    vector<int> sieve(maximum+1);

    // 0 and 1 are not prime
    sieve[0] = 1;
    sieve[1] = 1;

    // iterate through each value
    for (int i = 2; i * i <= maximum; i++)
        // if i is prime, we mark every multiple as composite
        if (sieve[i] == 0)
            for (int j = i * 2; j <= maximum; j += i)
                sieve[j] = 1;

    return sieve;
}
```

This approach does have its limitations. Namely, that if we try to generate primes up to very large numbers, we have to worry about running out of memory or taking too long to generate primes. In practice, if we have to deal with numbers larger than 10^7 we have to be concerned about not being able to use a sieve anymore (this is not a hard limit, and 10^8 or occasionally 10^9 are possible given the problem constraints, but generally at these sizes the intended approach would be to use a different method).

13.2.2 Miller-Rabin Primality Test

Miller-Rabin is a primality test that can determine whether or not a number is *probably* prime (but with enough certainty that we don't have to worry in practice) or definitely composite.

The code for this requires some modular arithmetic functions that we will cover later in this chapter. While we normally would like to include all necessary code chronologically, there are modular arithmetic functions we cover that themselves rely on prime numbers, so we include everything necessary to prime numbers before that. If you wish to implement this code directly, know that you need to read some of the modular arithmetic section later as well.

```cpp
// for consistency with sieve, return 0 if prime and 1 if composite
bool miller_rabin(int n) {
    // base cases
    if (n < 2) return true;
    if (n == 2) return false;
    if (n % 2 == 0) return true;

    // factor out powers of 2 from n-1, stored as s
    // also keep track of how many factors were removed
    int s = n - 1;
    int c = 0;
    while (s % 2 == 0) {
        s /= 2;
        c++;
    }

    // repeat this process for more precision
    for (int i = 0; i < 10; i++) {
        // random number in range [1,n)
        int a = rand() % (n - 1) + 1;

        // a^s mod n
        int m = modulo_pow(a, s, n);

        // if a^s mod n == 1, we skip the rest
        if (m == 1) continue;

        // loop through, ending early if m == 1 or n-1
        for (int j = 0; j < c && m != n-1; j++) {
            m = modulo_mul(m, m, n);

            // when the loop ends and m isn't n-1, we are composite
            if (j == c-1 && m != n-1) return true;
        }
    }

    // n is probably prime
    return false;
}
```

We can adjust this algorithm to deterministically verify that a number is prime or not (up to a known maximum value) by setting specific values of a that we've previously verified works for a specific range. For any number below 2^{32} we only need to use $\{2, 7, 61\}$. Using the first 12 primes, $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37\}$ works for any number below 2^{64}. Adjusting the code can be done as so:

```cpp
// values of a that will work for any value < 2^32
vector<int> miller_rabin_primes = {2, 7, 61};

// for consistency with sieve, return 0 if prime and 1 if composite
bool miller_rabin_deterministic(int n) {
    // base cases
    if (n < 2) return true;
    if (n == 2) return false;
    if (n % 2 == 0) return true;

    // factor out powers of 2 from n-1, stored as s
    // also keep track of how many factors were removed
    int s = n - 1;
    int c = 0;
    while (s % 2 == 0) {
        s /= 2;
        c++;
    }

    // repeat this process for more precision
    for (auto a : miller_rabin_primes) {
        // a is prime, so we can check if prime here
        if (a == n) return false;

        // a^s mod n
        int m = modulo_pow(a, s, n);

        // if a^s mod n == 1, we skip the rest
        if (m == 1) continue;

        // loop through, ending early if m == 1 or n-1
        for (int j = 0; j < c && m != n-1; j++) {
            m = modulo_mul(m, m, n);

            // when the loop ends and m isn't n-1, we are composite
            if (j == c-1 && m != n-1) return true;
        }
    }

    // n is prime
    return false;
}
```

13.2.3 Prime Factoring

We can factorize a number into its prime factors by simply iterating over every possible divisor and attempting to divide it. As long as we start with the smallest possible divisors (with 2 being the smallest nontrivial divisor) and work our way up to larger divisors, we

are guaranteed to only find prime numbers (since any composite number would've already been handled by its prime factors).

A very important optimization is to note that we only need to actually iterate up to $\sqrt{n}$ inclusive. We can realize that if a number n is composite, then it must have at least one prime factor $\leq \sqrt{n}$. In contrast, if a number has no prime factors $\leq \sqrt{n}$, we can tell that it's prime. We can use this for our factoring algorithm – we can find all prime factors $\leq \sqrt{n}$ through attempting division, and then if what's left over of n is not equal to 1 we know that is a prime as well.

```
vector<int> factorize(int n) {
    // vector to store our factors
    vector<int> ans;

    // attempt all factors until sqrt(n)
    for (int i = 2; i <=sqrt(n); i++) {
        // divide n by i as much as possible
        while (n % i == 0) {
            ans.push_back(i);
            n /= i;
        }
    }
    // if there's anything left over, add it too
    if (n > 1)
        ans.push_back(n);

    return ans;
}
```

13.2.4 Pollard's Rho Algorithm

Under Construction

13.3 Greatest Common Divisor

The greatest common divisor between two non-zero integers is as its name suggest – the largest number that can divide both of two given integers. For example the divisors of 12 are $\{1, 2, 3, 4, 6, 12\}$ and the divisors of 16 are $\{1, 2, 4, 8, 16\}$, and as a result the GCD between the two is 4 which is the largest shared divisor between the two. Similarly, the GCD of 5 and 6, which share no divisors other than 1, is 1. As well, it can be observed that the GCD of two equal numbers is that number itself again, since a number contains itself as a divisor (and as such, the GCD of 10 and 10 is 10).

We also define the GCD of any number and 0 to be that number: $GCD(x, 0) = x$.

GCDs have many applications. One of the simplest being in reducing fractions – we know by looking at it that $\frac{2}{6}$ can be simplified into $\frac{1}{3}$. We can easily obtain this simplified fraction

by dividing both the numerator and the denominator by the GCD of the numberator and denominator. For example, we can simplify the fraction $\frac{1800}{2100}$ by finding the GCD between both parts of the fraction, which is 300, and dividing both 1800 and 2100 by it. We get $\frac{6}{7}$ which is the most reduced form that fraction can take.

13.3.1 Basic GCD

Calculating the GCD of two numbers can be done through a variety of methods. We'll explore the *Euclidean Algorithm* approach to the problem.

It's based on noticing that the GCD of two numbers doesn't change when you subtract the smaller number from the larger. For example, $GCD(100, 12) = 4$, as is $GCD(100 - 12, 12) = 4$. This can be performed repeatedly as well. For example, $GCD(100, 12) = GCD(88, 12) = GCD(76, 12) = ... = GCD(16, 12) = GCD(4, 12)$. We can continue the process even further by swapping 4 and 12 (since 12 is now larger), and find that $GCD(12, 4) = GCD(8, 4) = GCD(4, 4) = GCD(0, 4)$ before we can't go any further (we can keep subtracting 0 from 4 but it doesn't change the values we have, so when one parameter is 0 we have our base case).

As it turns out, if you repeat this process enough you will always be left with $GCD(x, 0)$. And calculating the GCD of any value and 0 is easy, it's that value unchanged. So we have our GCD now.

We can convert this into a form that's easier to calculate. Repeated subtraction is relatively slow when we're dealing with large values, but we may notice that repeatedly replacing a with $a - b$ until $a < b$ is equivalent to finding $a\%b$ which we can calculate in a single operation. We may also notice that if $a < b$ then $a\%b = a$, which we can use to avoid swapping values if one is larger than the other and instead keep swapping each iteration. This prevents us from having to deal with our second parameter being larger than the first, or our first parameter being 0 to start with.

This is fairly easy to express recursively. All together, our code may look like:

```
int gcd(int a, int b) {
    // base case
    if (b == 0)
        return a;

    // recursively calculate, swapping each time
    int val = gcd(b, a % b);
    return val;
}
```

However, you generally don't need to code this yourself. GCC comes with `__gcd(a,b)` as a function, C++17 added `std::gcd(a,b)` to the standard library, Python includes `math.gcd(a,b)` through its math module, and Java does not offer a GCD function for primitives but does support `a.gcd(b)` for its `BigInteger` class.

13.3.2 Extended GCD

Sometimes we want to know more than just the GCD of two numbers – namely we might want to know what values of x and y could satisfy the equation $ax + by = GCD(a, b)$ where a and b are known.

The values of x and y that can satisfy that equation aren't necessarily unique. For example, with $a = b = 1$ which has a GCD of 1, we can find that $x = 1, y = 0$ works, or $x = 2, y = -1$ works, or $x = -10, y = 11$, or infinitely many different other options. For our purposes, we're interested in finding one valid solution.

```
int extended_gcd(int a, int b, int& x, int& y) {
    // base case
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }

    // recursively calculate
    int val = extended_gcd(b, a % b, x, y);

    // modify x and y
    x -= a / b * y;
    swap(x, y);

    return val;
}
```

13.3.3 Primal Representation

Given a prime factorization of two numbers, the GCD is the product of factors in common. For example, the prime factorization of 60 is $\{2, 2, 3, 5\}$ while the prime factorization of 24 is $\{2, 2, 2, 3\}$. The factors in common are $\{2, 2, 3\}$ whose product is 12, which is the GCD of 60 and 24.

13.3.4 Least Common Multiple

The least common multiple between two non-zero integers can be calculated quickly: $\frac{a*b}{gcd(a,b)}$. Since $gcd(a, b)$ is a divisor of both a and b (and by extension $a * b$) this will always be an integer as well.

The primal representation of the LCM is similar to the primal representation of their GCD, except it's the product of factors they don't share multiplied by the product of factors in common (or GCD). For example, with 60 and 24 again, we have $\{2, 5\}$ as unshared factors, along with $\{2, 2, 3\}$ as common factors, which is $2 * 2 * 2 * 3 * 5 = 120$

which is the LCM.

13.3.5 Coprime Numbers

Numbers are considered coprime (or relatively prime) if their GCD is 1. That is to say, they have no prime factors in common.

Prime numbers are, by necessity, coprime to other primes. This is because they would share no prime factors (for both, their sole prime factor being themselves) and as such have a GCD of 1. Following similar logic, any number is either coprime to a prime, or it's a multiple of that prime (because the only scenario that the GCD wouldn't be 1 is when the prime is a factor).

13.4 Modular Arithmetic

13.4.1 Addition

Addition can be fairly simple – we add our numbers together and then perform the modulo operation on it. For example:

```
int modulo_add(int a, int b, int m) {
    return (a + b) % m;
}
```

The modulo operation is relatively expensive for a basic arithmetic operation, however. If we can guarantee that a and b are both less than m, we can avoid doing any modulo operations with:

```
int modulo_add(int a, int b, int m) {
    // subtract m if a+b>m, subtract 0 otherwise
    return (a + b) - ((a + b >= m) ? m : 0);
}
```

13.4.2 Subtraction

Subtraction is similar to addition, but we need to guarantee that we don't go negative. We can fix this by simply adding m before performing our modulo operation:

```
int modulo_sub(int a, int b, int m) {
    return (a - b + m) % m;
}
```

Likewise with addition, we can also optimize our subtraction function to not perform any

modulo operation:

```
int modulo_sub(int a, int b, int m) {
    // add m if a<b, add 0 otherwise
    return (a - b) + ((a < b) ? m : 0);
}
```

13.4.3 Multiplication

Multiplication can (mostly) be performed similar to the non-optimized method of addition and subtraction – by simply doing the multiplication operation and then performing the modulo operation on that. However, we may be concerned with overflows.

Specifically, consider $m = 10^9 + 7$. The largest numbers we will multiply will be when both a and b are equal to $10^9 + 6$. $a * b$ is somewhat above 10^{18} which is outside of the range of what a 32-bit integer can hold. In a language like C++ or Java, we will have to ensure that we use 64-bit integers when performing our multiplication:

```
int modulo_mul(int a, int b, int m) {
    // cast a to 64-bits to prevent overflows
    return ((long long)(a) * b) % m;
}
```

13.4.4 Exponentiation

Exponentiation follows a very similar methodology to the binary exponentiation method we explored earlier, except now at each multiplication we perform, we need to instead perform the multiplication modulo m. We can avoid our `modulo_mul` method's repeated casting to 64-bit integers by simply keeping every value we multiply in 64-bit integers always.

```
int modulo_pow(long long b, long long e, long long m) {
    // handle base case where every value is 0
    if (m == 1)
        return 0;

    // perform binary exponentiation
    long long value = 1;
    while (e > 0) {
        // our multiplication is modulo m
        if (e % 2 == 1)
            value = (value * b) % m;
        // our squaring is modulo m as well
        b = (b * b) % m;
        e /= 2;
    }
    return value;
}
```

The above code also handles a special edge case where $m = 1$, and by consequence our answer will always be 0.

13.4.5 Division (Prime Modulo)

So far we've explored and implemented operations that seem fairly similar to how they'd be implemented without modulus – addition, subtraction, and multiplication are the same as normal but with the modulus operation at the end of its computation. Exponentiation is similar, except with modulus operations performed at intermediary steps as well.

Division is more complicated. While some operations, such as $\frac{6}{2}\%7$ can be expressed as the regular division operation (since 2 evenly divides 6, and equals the integer 3) modulo 7, the same is not true of $\frac{2}{6}\%7$. $\frac{2}{6} = \frac{1}{3} = 0.333...$ but modular arithmetic is based on integers, so this won't do.

Let's consider what division really is: if multiplication is finding $x * y = z$ with known values of x and y, division is finding $x * z = y$. Modular division follows the same principle, where we're trying to find $(x * z)\%m = y\%m$ given known values of x, y, and m.

If we work it out by hand, we can discover that a solution for $\frac{2}{6}\%7$ is 5 – by rearranging it into our multiplication equation, we find that $(6 * 5)\%7 = 2\%7$ to be valid.

A naive method of calculating modular division would be to try every possible value $[0, m)$ and see if it satisfies the above equation. Unfortunately, when our m is large as it often is, this can be far too slow to reliably use.

This section is about division with a prime m because it's easier to do (we'll cover arbitrary values of m later). Another way of representing division is as $x * \frac{1}{y} = z$ with known x and y, and the same applies under modulus assuming that we can calculate $\frac{1}{y}\%m$.

As it turns out, it's not very difficult to calculate $\frac{1}{y}\%m$ when m is prime. Fermat's little

theorem states that $a^m\%m = a\%m$ when m is prime, which is equivalent to $a^{m-1}\%m = 1\%m$, and a can be divided out from both sides to obtain $a^{m-2}\%m = \frac{1}{a}\%m$. As such, as long as we know m is prime, we can quickly calculate $\frac{1}{y}$ by raising it to the (modular) power of $m-2$. All together, we can calculate $\frac{x}{y}\%m$ as $x * y^{m-2}\%m$.

13.4.6 Modular Inverse

Adding on from the section on division (prime modulo), when we try to find $\frac{1}{a}$ (equivalently a^{-1}), it turns out there's a name for this. It's the *modular multiplicative inverse* or just *modular inverse*. And it isn't limited to just primes either, there can be

That said, we aren't guaranteed for a multiplicative inverse of a number to exist under every modulo. Namely, for there to be a multiplicative inverse of a under modulo m, a and m need to be coprime. This will always be the case when m is prime (since $a\%m$ is smaller than m and cannot be a multiple of m, and as a prime number m is coprime to anything that isn't a multiple of it), but may not be the case if m is composite. For example, $a = 2, m = 4$ has no multiplicative inverse, since 2 and 4 aren't coprime. We can also verify this by trying to find an integer solution for $(2 * x)\%4 = 1\%4$ which we can quickly find does not exist (for all values of x we will either obtain $2\%4$ or $0\%4$, never $1\%4$).

As for calculating the modular inverse, recall the motivation for the extended GCD, finding x, y, and $GCD(a, b)$ in the equation $ax + by = GCD(a, b)$ given a and b. Now consider finding the GCD of a and m, knowing that they are coprime: $ax + my = 1$ since the GCD of any coprime numbers is 1. We can quickly rearrange this to find $ax = 1 - my$, and modulo m we find $ax\%m = 1\%m$. If we consider that this can be further rearranged into $x\%m = \frac{1}{a}\%m$, we've successfully calculated our modular inverse.

Conveniently, our function to determine whether a and m are coprime and our function to calculate the value of x are actually the same function. This makes the code for the modular inverse relatively short:

```
int modulo_inverse(int a, int m) {
    int x = 0, y = 0;
    // calculate x and GCD together
    // if GCD is not 1, there is no modular inverse
    if (extended_gcd(a, m, x, y) != 1)
        return -1;

    // handle negatives and return
    return (x + m) % m;
}
```

13.4.7 Division (Any Modulo)

Division under any modulo can now be implemented relatively easily: replace the prime mod specific method of finding the modular inverse $a^{m-2}\%m$ and use the general modular inverse method instead. We also have to take care that the value of this modular inverse isn't -1 (or whatever other value we have set as invalid), but the core idea of the function remains the same.

13.4.8 Discrete Log

Under Construction

13.4.9 Square Root

Under Construction

Simple Fractions

There are multiple ways to represent the same number. For example, 0.5 can also be represented as $\frac{1}{2}$ or alternatively $\frac{2}{4}$ or even $\frac{50}{100}$. The simplified fraction is what we call the one with the smallest denominator: in this case $\frac{1}{2}/$.

Input

Input consists of some number of decimal numbers n where $0 < n < 1$ and n has at most 6 significant figures.

Output

For every number n, output that number in simplified fraction form.

Constraints

Time Limit: 1 second
Memory Limit: 1024 mb

Samples

`Input:`

```
0.5
0.25
0.125
0.123456
```

`Output:`

```
1/2
1/4
1/8
1929/15625
```

14 Combinatorics Problems

Under Construction

15 Geometry Problems

15.1 Primitives

There are various simple constructs in geometry that we have to use frequently. These are called "primitives" here, and can be thought of as the basic building blocks of any geometry problem.

15.1.1 Points

Points are the most basic construct in geometry – they simply indicate a specific location.

In terms of code, a simple point structure can be created by simply storing a variable for each dimension. For example, with a 2D point:

```
struct point {
    // variables for each dimension
    double x, y;

    // constructors
    point() : x(0), y(0) {}
    point(double x, double y) : x(x), y(y) {}
};
```

15.1.2 Lines

Lines are a logical extension of points – they represent a point and a direction.

There are a variety of ways to define a line. One is through two points. Another is through a point and an angle, which determines the direction.

15.1.3 Segments

Segments are akin to lines, except they have a fixed distance instead of an infinite length.

Like lines, there are more than one way to define segments. Two points is another option, except in segments they are the start and end points rather than two arbitrary points along the line. A point, an angle, and a magnitude can be used to define a segment as well.

15.1.4 Circles

Circles are a slightly more complicated structure than the previous ones explored so far. They can be defined simply as a point and a radius, where the circle itself is all the

(infinite) points that are exactly the radius away from the point (which acts as the center of the circle).

While this definition itself may not be overly complicated, we have various other things to consider with this structure, and various circle-specific terms to define. The diameter is the width of the circle, or the distance between one point on a circle and the opposite point – this is easily calculated as twice the radius. The circumference, which in plain terms is the length of the circle if it was unrolled into a straight segment, is calculated as $2\pi r$. The area of a circle is defined as πr^2.

We can also define a circle by three points. We won't cover how to derive this, and that is left as an exercise for anyone interested. The code is included:

```
struct circle {
    // center point and radius
    point c;
    double r;

    // constructors
    circle() : r(0) {}
    circle(point c, double r) : c(c), r(r) {}
    circle(point i, point j, point k) {
        double sq1 = (i.x*i.x+i.y*i.y);
        double sq2 = (j.x*j.x+j.y*j.y);
        double sq3 = (k.x*k.x+k.y*k.y);

        double A = i.x*(j.y-k.y) - i.y*(j.x-k.x) + j.x*k.y - j.y*k.x;
        double B = sq1*(k.y-j.y) + sq2*(i.y-k.y) + sq3*(j.y-i.y);
        double C = sq1*(j.x-k.x) + sq2*(k.x-i.x) + sq3*(i.x-j.x);
        double D = sq1*(k.x*j.y - j.x*k.y) + sq2*(i.x*k.y-k.x*i.y) +
                   sq3*(j.x*i.y-i.x*j.y);

        c = point(-B / (2*A), -C / (2*A));
        r = sqrt((B*B + C*C - 4*A*D) / (4*A*A));
    }

    // methods
    double diameter() {
        return r * 2;
    }
    double circumference() {
        return 2 * M_PI * r;
    }
    double area() {
        return M_PI * r * r;
    }
};
```

Note that these aren't the only ways to define a circle. The equation $(x-h)^2+(y-k)^2 = r^2$

where $\{h, k\}$ is point is commonly used to define a circle and can express it on the coordinate plane. This definition is useful to keep in mind for some problems, but generally very easy to convert into a point and a radius, so it is not included here.

15.1.5 Triangles

Under Construction

15.1.6 Rectangles

Rectangles are a simple form of polygon with 4 points and all right angles. Defined another way, rectangles are a polygon with 4 points and at most two different side lengths (a necessary condition given that every angle is a right angle). They are a very common primitive that sees wide use in geometry problems.

For simplicity's sake, we don't define squares as a distinct primitive. They are simply rectangles with each side length equal.

The area of a rectangle is very quick to calculate, it is simply wh where w is the width and h is the height.

Axis-aligned rectangles refer to a rectangle with no rotation. Specifically, every side of the rectangle is on either the x-axis or on the y-axis. This is an important distinction because axis-aligned rectangles are a special case that leads much more simplified math in many situations.

15.1.7 Polygons

Under Construction

15.1.8 Convex Polygons

Under Construction

Basic:

1. https://open.kattis.com/problems/beavergnaw
2. https://open.kattis.com/problems/movingday
3. https://open.kattis.com/problems/sanic

16 Statistics Problems

Under Construction

17 Game Theory Problems

Under Construction

17.1 Nim

Under Construction

17.2 Sprague-Grundy

Under Construction

17.2.1 Nimber Addition

Under Construction

17.2.2 Nimber Multiplication

Under Construction

17.3 Josephus Problem

The Josephus problem is a specific task that can be formulated as:

n people stand in a circle. We begin at the first person, who then excludes the person who is $k-1$ positions to the right of themselves. We then repeat the same process, starting at the person to the right of the one who was just removed. We continue until there is only one person who hasn't been excluded. Our task is to find out who that last person is.

As an example with $n = 7$ and $k = 3$, we begin with a list $\{0, 1, 2, 3, 4, 5, 6\}$. The first person removes the position 2 to the right, so we adjust our list and have $\{0, 1, 3, 4, 5, 6\}$. 3 then removes 5, giving us $\{0, 1, 3, 4, 6\}$. 6 removes 1, for $\{0, 3, 4, 6\}$. 3 removes 6, 0 removes 4, then 0 removes 0. We arrive at a list of size 1, where our answer then is 3.

The problem has some grim origins, originating from a story of a historian Flavius Josephus living in the first century, wherein himself and 40 soldiers decided to avoid capture by standing in a circle and committing suicide in a manner not unlike what was described above. The original story has some differences from the formulated problem, namely that there were two survivors rather than just one, and that Josephus claimed it luck or will of God rather that he be spared rather than intentionally calculating the safest spot, but broadly speaking it's a strong example of the Josephus problem.

```
int josephus(int n, int k) {
    // handle base cases
    if (n == 1)
        return 0;
    if (k == 1)
        return n-1;

    // recursive special case
    if (k > n)
        return (josephus(n-1,k)+k)%n;

    // recursive general case
    int res = josephus(n-n/k,k)-n%k;
    if (res < 0)
        return res + n;
    return res + res/(k-1);
}
```

```
// case where k = 2
int josephus(int n) {
    // p = 2^(log2(n))
    int p = 1 << (32 - __builtin_clz(n)-1);
    return 2*(n-p);
}
```

Under Construction

A Optimizations

It's not uncommon for runtime to be the main concern with a problem. Sometimes, you may get a time limit exceeded issue with an approach you believe to be just barely too slow. Knowing how you can better optimize your code can be useful in getting the fractions of a second necessary to pass a problem like that.

A.a Concepts

There are various fundamentals we need to consider when dealing with optimizations.

A.a.i Cache Efficiency

A.a.ii Register Usage

A.a.iii Branch Prediction

A.a.iv Type Conversions

A.a.v Data Alignment

A.a.vi Run-Time and Compile-Time

A.b General

A lot of advice for optimization applies across the board, and are generally worth keeping in mind whenever you come across them.

A.b.i Avoid Additional Allocations

A.b.ii Bitwise Operations

A.b.iii Boolean Operand Order

A.b.iv Multidimensional Array Element Access Order

A.b.v Lazy Evaluation

A.c Specific Techniques

Sometimes we have to deal with very specific optimizations. Generally, these involve some way of doing less work.

A.c.i Reverse Access of Loops

A.c.ii Favor Simple Loop Counters

A.c.iii Recursion Alternatives

A.c.iv Distance as Distance Squared

A.c.v Counting Arrays as Boolean Arrays

A.c.vi Multiple Divisions as Multiplication by Reciprocal

A.d C++

There are some specific options that C++ allows for much more optimized code.

A.d.i Fast IO

A.d.ii Pragmas

A.d.iii Inlined and Macro Functions

A.d.iv Const Correctness

A.d.v Floating Point Exceptions as NAN/INF

A.e Java

Java has some optimizations that are unique to it.

A.e.i Fast IO

A.f Python

Python is generally the slowest language we regularly use in competitive programming, and as such it's important that we can.

A.f.i Fast IO

B Templates

Many competitive programmers, rather than remember the details for complicated data structures and algorithms, will save a copy of the necessary code for later reuse.

B.a Snippets

Code that you intend to use for templates can differ widely from other code you write – it's often far more generalized than what you normally write in competitive programming, which makes it like general software development in that sense. But it's also usually far more compressed with little concern about architectural decisions, much more like competitive programming than general development.

B.b Team Notebooks

Currently in ICPC competitions, and many smaller-scale competitions, you are allowed a 25-page single-sided notebook containing code snippets. This section contains some recommendations for creating this.

If you look for different ICPC notebooks, you will find many of them are written in latex. While alternatives (like word documents) are possible, latex is generally recommended because there are many options and packages that can include code snippets and fine-tune formatting easily. It tends to be easier to maintain and modify, and generally produces more lean or compressed final documents as well (with the proper settings) with far more customization options available to you than the majority of alternatives.

Some good examples of notebooks would be:

- `https://github.com/kth-competitive-programming/kactl/blob/master/kactl.pdf`
- `https://github.com/SuprDewd/CompetitiveProgramming/blob/master/comprog.pdf`

Index

Q

R

S

T

U

V