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.

It's never too early to begin competitive programming either. Intentionally waiting until later to start programming competitively won't help you.

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.

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 small 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.

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:	Output:
`No Input`	`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!");
    }
}
```

```python
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:

```cpp
#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

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 a significant issue.

2.3 Python

2.3.1 Setup

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.

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

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.
- $O(logn)$
- $O(n)$ is an extremely common runtime, where you have to go through every element of the input.
- $O(nlogn)$
- $O(n^2)$ 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:

```python
# 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:

```python
# 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
```

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

Input:

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

Output:

```
output
```

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

Input:

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

Output:

```
output
```

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

4.4 Debugging

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

tion)

Reading Comprehension:

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

5 Foundational Data Structures

5.1 Arrays

5.1.1 Dynamic Arrays

5.2 Sets

5.3 Maps

5.4 Sequential Structures

5.4.1 Stack

5.4.2 Queue

5.4.3 Deque

5.4.4 Priority Queue

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.

6.2 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 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 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...

7 Iterative Problems

7.1 Brute-Force

7.2 Simulation

7.3 Two-Pointers

7.4 Sliding Window

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

9 Dynamic Programming Problems

10 Advanced Data Structures

10.1 Range-based Structures

10.1.1 Sparse Table

10.1.2 Fenwick Tree / BIT

10.1.3 Segment Tree

10.1.4 Wavelet Tree

10.2 Ropes

11 String Problems

11.1 String Concepts

11.1.1 Anagrams

11.1.2 Palindromes

11.1.3 Prefixes

11.1.4 Suffixes

11.2 Pattern Matching

11.2.1 Regex

11.2.2 Z-Algorithm

11.2.3 Knuth-Morris-Pratt

11.2.4 Boyer Moore

11.2.5 Aho Corasick

11.3 Subsequence Matching

11.4 Hashing

11.5 Distance

11.5.1 Diff

11.5.2 Levenshtein

12 Graph Problems

13 Number Theory Problems

14 Combinatorics Problems

15 Geometry Problems

16 Statistics Problems

17 Game Theory Problems