NOI.PH Training: Technical Stuff 1

Technical Stuff (and C++ Garbage)

Payton Yao, Kevin Atienza & Vernon Gutierrez

Contents

1 Introduction

Throughout this module, we'll be delving into:

- **Technical stuff**: The command line, some details about how a computer works, and what happens when you compile and run programs.
- **C++**: Features you might not know about, its standard library (which contains some useful data structures and algorithms that you can use), and some of its technical inner workings that might be helpful for you to know.

Note: Some of the things said here will not be strictly correct, or somewhat simplified compared to actual reality. This is done to make the explanations simpler.

2 The command line

Many of you already know how to code. Some way or another, you've managed to solve some problems from the NOI allowing you to get this far. For some of you, you might have programmed directly into HackerRank's web interface, compiling and debugging your programs through that platform. For others, you might have used an Integrated Development Environment (IDE) like CodeBlocks or DevC++.[1] From our understanding, very few of you do everything through the **command line** which is what we'll introduce here.

You might be asking: "Why?" If everything can be done on an IDE, why bother learning to use the command line? There are two answers, both equally valid. The first and more practical one is that you may not have the exact IDE you're comfortable with available at all times. Especially at the IOI, HackerRank definitely won't be available. However, every computer will have some form of command line available, and every computer meant for programmers will have the command line tools set up. The second reason is about understanding. IDEs give way too much magic and sometimes we miss all the small things that happen behind the scenes.

Note that this is only a short introduction to the command line. Mastering its use takes years of experience to do extremely intricate things, but basic usage allowing you to navigate the file system, compile code, and run programs you can learn in the space of an hour or so.

At this point, it's important to note that there are some major differences between using the command line on Windows and on Linux. Since *NOI.PH onsite rounds and IOI all use Linux*, we will only describe the **Linux** terminal. For Mac or Windows users, you need to take the following extra steps:

- For **Mac** users, you can mostly follow along since the differences between Mac and Linux are small. However, note that the built-in C++ compiler is Clang. You may want to install GCC to access GCC-specific features[2] (there are a few handy ones for competitive programming), but this is not required. Note that the IOI uses GCC.
- If you are on **Windows**, I strongly recommend installing WSL.[3] WSL allows you to pretend that your computer is running Linux even if you're on Windows. Then, C++ already comes installed with the Linux distribution. Then set up WSL for VS Code.[4] (VS Code is a text editor. We recommend using it.) WSL has a separate file system from the Windows file system. To locate your WSL files (e.g., for submitting to online judges), look for the "Linux" folder at the bottom of the navigation pane in File Explorer. If you use the terminal within VS Code connected to WSL, you are effectively using Linux and can pretend that your OS is Linux, so follow the commands for Linux when compiling, running, listing files, etc.

As a historical note, there was once an operating system called Unix developed at Bell Labs in the 1970s. It's this original operating system whose design philosophy Linux and Mac copied. Because they come from similar roots, much of the core command line tools are the same with some small differences in some of the options they can be given. Advanced users will also note a striking similarity in the way system files are organized, for example Linux has `/home/` where Mac has `/Users/`.

[1] IDEs are these huge pieces of software that combine multiple tools in one, allowing you to edit source code, compile programs, debug them, and more, all in one convenient program.

[2] http://cs.millersville.edu/~gzoppetti/InstallingGccMac.html

[3] https://learn.microsoft.com/en-us/windows/wsl/install

[4] https://code.visualstudio.com/docs/remote/wsl

2.1 Basic commands

The first step is to learn how to open the command line terminal. If you're on Ubuntu, open the Gnome Terminal. (And if you're not on Ubuntu, you probably already know what you're doing anyway.) For Mac users, you can use the built-in Terminal, just search for it using Spotlight. You can also opt to download something called iTerm2 which is a much-improved version of Terminal with lots of useful new features.

The important commands are as follows:

- `cd` - change directory. Use as `cd ~/Downloads`.
- `ls` - list files in current directory. Use as `ls`
- `cp` - copy a file. Use as `cp source.txt destination.txt`
- `mv` - move a file from one place to another. Use as `mv original.txt new.txt`. Interestingly, this is also the command used to rename files.
- `rm` - delete a file. Use as `rm a.txt`
- `mkdir` - make a new directory. Use as `mkdir new_directory`
- `rmdir` - delete an empty directory. Use as `rmdir new_directory`

We encourage you to try these commands out. Your computer won't break if you type a command wrong. (Just make sure not to delete or overwrite your important files!) You can hit the tab key to auto-complete.

You may have noticed that we did `cd ~/Downloads`. The symbol `~` is a shorthand for `/home/user` in Linux (`/Users/user` in Mac) where `user` is the username of your computer account.[5] (For example, it's `/home/kevin` in my computer.) We'll have a bigger discussion on Unix commands and the environment in the future, but this is enough for now.

Read more about the Unix command line at:
http://www.ee.surrey.ac.uk/Teaching/Unix/

2.2 Compiling and executing programs

For this training, we will focus on using C++. And to run C++ programs, we first have to learn to compile them. And to compile them, you first need to install a compiler! We will assume that the GCC compiler is installed. For Linux users, it comes installed on the base operating system. For Mac users, earlier we've provided instructions on how to install GCC.

2.2.1 Compiling programs

Now that you're all done setting up, it's time to start compiling! If you save your source code file as `source.cpp`, you can compile it using the command `g++ source.cpp`.

By default it saves the resulting executable as "`a.out`". You can change the name for the resulting output file by adding `-o filename` to the command. For example, if my source code is "`example.cpp`" and I wanted to save the output executable as "`executable`" I should then type

[5] To check that this is true, enter `echo ~` in the terminal. (`echo` just prints something, e.g., `echo "Hello World!"`)

the command `g++ example.cpp -o executable`. Note that unlike Windows, you don't need to add an extension such as `.exe`; Linux knows that a file is executable using other means.

We recommend that you try compiling a sample Hello World program this way before moving on.

If you're having an error like `g++: error: source.cpp: No such file or directory`, please ensure you're in the right directory first.

2.2.2 Running executable files

Now that you've compiled your program, it's time to run it! The command to run a program is `./program_name`. So if your program is saved as `a.out`, you should type "`./a.out`", and if it's saved as `executable`, you should type "`./executable`". Note that you have to be on the folder containing the compiled executable.

2.2.3 Killing running programs

In case you accidentally run code that never ends or you just want it to die for some other reason, what can you do? It's gonna block your command line and stop you from typing other commands, so you will want to kill it. You can opt to kill it using the System Monitor[6], but there's a much easier way: Just press `Ctrl + C`. This will kill the program that's hogging up the command line.

Please write a program with an infinite loop, compile it, run it, then kill it just to try out `Ctrl + C`.

2.3 Input redirection

2.3.1 Saving output to a file

There are cases when we want to save the output of our program to a file. The simple way is to highlight and copy things from the command line, paste it in your text editor, and save the file. But there's an easier and more precise way to do it: Use the `>` symbol.

To be precise, if we want to save the output of the program executable to a file called `output.txt`, we type the command `./executable > output.txt`

Try it yourself! Of course, you can change the output file name to anything you want and not just `output.txt`

It's important to note at this point that **this will completely erase the contents of the file you're saving into**, even if your program hasn't printed anything out yet. So be careful!

2.3.2 Reading input from a file

Another common case is when we want our program to read input from some file instead of us having to type the values manually. Again, there's a simple way to do that! Use the `<` symbol.

If we want to make the program executable read from a file called `input.txt`, we type the command `./executable < input.txt`

[6] the Linux equivalent of Windows' Task Manager

This is extremely useful for debugging your solutions for contests, especially for problems with large-ish input like graph problems.

2.3.3 Combining the two

There are cases where we have input from a file and we want to save the output to another file. For example, in old versions of Google Code Jam, input files are downloaded from their website and you have to upload the correct output file. Or maybe you just want to store the output for a pre-determined input file into a different file, presumably to inspect later. It turns out that you can do this by combining `<` and `>`.

An example command that combines them is `./executable < input.txt > output.txt`

Note that the command `./executable > output.txt < input.txt` is just as valid and does exactly the same thing.

2.3.4 Extras: pipe

On the topic of redirection, sometimes you will find that you have two programs and you want the output of one file to feed into the input of the other. The simple solution is to redirect the output to a file then run the second one with the output of the first. You can save yourself the temporary file by using the `|` symbol, also called *pipe*. For example, the command `./program1 | ./program2` feeds the output of `./program1` directly as input of `./program2`. In fact, this command simultaneously runs both programs; it doesn't wait for `./program1` to finish first. If `./program2` tries to read an input that `./program1` hasn't given yet, it simply waits.

2.4 Extras: `diff`

There's a very useful tool called `diff`. What `diff` does is it compares two files line by line and tells you which lines differ. For example, running `diff file1.txt file2.txt` will compare the two files. If they are the same, then `diff` outputs nothing. Otherwise, it outputs the lines where they differ.

This tool is useful for debugging. For example, suppose you have an accepted solution for Subtask 1, say `sol1.cpp`, and you've written a solution attempt for Subtask 2, say `sol2.cpp`. Then to check whether `sol2.cpp` is correct, one thing you can do is to run both solutions on a bunch of input files and then check that they give the same results. Instead of manually checking that the outputs are the same, you can use the `diff` tool to automate large parts of this process. It saves time, and is also less error-prone.

The process of comparing two programs across lots of input files is called **stress testing** and is a very powerful debugging technique, for a couple of reasons:

- If `sol1.cpp` and `sol2.cpp` differ in some input, then not only do you infer that at least one of them is wrong, but you also get access to a test case that breaks it.
- Doing stress testing is a purely mechanical process; very little thinking is required.
- The whole process can be automated! (See the *bash scripting* portion below for more details.)

For these reasons, I strongly recommend learning how to stress test. With enough experience, this process can be done in a few minutes, making it very much worth it.

2.5 Extras: bash scripting

The lines you enter in the terminal are actually valid commands in a scripting language called **bash**. Since there's a good chance that you're already in the middle of learning a new language (C++), I won't describe the full syntax and semantics of bash. Instead, I will just teach you a few basic things about it, hopefully enough for you to at least know how to automate the stress testing process.

Suppose you want to stress test `sol2.cpp` against `sol1.cpp`, and you have a program that generates valid random test cases, say `gen.cpp`. Here's a sample bash script that performs basic stress testing by first compiling the programs and then running them on three input files generated by `gen.cpp`.

```
# this line is explained below
set -e

# compile the programs
g++ sol1.cpp -o sol1
g++ sol2.cpp -o sol2
g++ gen.cpp -o gen

# test once
./gen > input.txt
./sol1 < input.txt > output1.txt
./sol2 < input.txt > output2.txt
diff output1.txt output2.txt

# test again
./gen > input.txt
./sol1 < input.txt > output1.txt
./sol2 < input.txt > output2.txt
diff output1.txt output2.txt

# test again
./gen > input.txt
./sol1 < input.txt > output1.txt
./sol2 < input.txt > output2.txt
diff output1.txt output2.txt

echo "DONE"
```

To run this script, first save it into a file, say "`test.sh`". Then, enter the command `chmod +x test.sh` to tell Linux that `test.sh` is an executable. Finally, enter `./test.sh` to run it.

Notice that a bash script is basically a collection of commands that the terminal accepts! Thus, at its most basic, a bash script at the very least allows you to run several commands in sequence while only typing one line in the terminal.

Now, let me explain a couple of details in this script.

- Comments begin with the `#` character.
- `echo` is just a command that prints something to the terminal. You can also use it directly in the terminal—enter the command `echo "Hello World!"`
- `set -e` is a command that tells the script to exit immediately as soon as any command

issues an error. This is useful for a couple of reasons:

- First, as it turns out, `diff` issues an error if the two files it's comparing are different, which means the script stops as soon as the stress testing fails!
- Also, `g++` issues an error if it fails to compile the program, which is also nice since that means the script won't even bother doing the stress test if the compilation step failed!
- Importantly, if your stress test fails and finds a test case where `sol1.cpp` and `sol2.cpp` differ, then the breaking test case will be `input.txt`. In other words, you get access to a failing test case which you can use for debugging!
- Finally, since `echo "DONE"` is only ever run if all commands succeeded, we can easily tell whether the stress test succeeded or not by checking whether the string "`DONE`" was printed in the terminal or not.

Very convenient!

We can actually do even better! It turns out that bash is pretty much a complete programming language: it has **`if`** statements, **`for`** loops, variables, functions, arrays, etc. You don't need to learn all of these, but maybe it's at least useful to know how to perform a loop, since this will allow us to do stress testing across hundreds or even thousands of inputs!

There are many ways to loop in bash. Luckily, one of them is pretty close to the C++-style **`for`** loop:

```
# exit as soon as any command fails
set -e

# compile the programs
g++ sol1.cpp -o sol1
g++ sol2.cpp -o sol2
g++ gen.cpp -o gen

# test many many times
for ((x = 0; x < 100000; x++)); do
    echo "DOING TEST $x"
    ./gen > input.txt
    ./sol1 < input.txt > output1.txt
    ./sol2 < input.txt > output2.txt
    diff output1.txt output2.txt
done

echo "DONE STRESS TEST!"
```

This performs 100000 tests! Also, remember that if this takes too long, you can always kill the script with `Ctrl + C`.

You can also comment out the `echo` line inside the loop if you don't want to print 100000 lines in the terminal. It's up to you!

3 Structs in C++

In C++, **struct**s are a way to create new data types. The data types produced by **struct** are *compound* data types, that is, they are composed of other, simpler data types.

A prototypical example would be if you want to, say, create a data type called `Person` with two attributes, `name` and `age`, of types `string` and `int`, respectively. This is how you do it:

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Person {
    string name;
    int age;
};

// sample usage:
int main() {
    Person p;
    p.name = "Jared";
    p.age = 15;

    cout << "The person named " << p.name << " is of age " << p.age << ".\n";
}
```

Notice that we use the dot character `.` to access the attributes of a person. Thus, a **struct** is pretty simple: conceptually, it's just a way to pack together a collection of values into a single data type. Of course, you may just as well use several variables, but sometimes, using a **struct** is conceptually simpler. For example, if your problem involves queries, and each query consists of two integers, then it may make sense to create a **struct** called `Query` containing two integers. In some instances, using **struct**s may even make your code run faster, depending on what you're doing. Although sometimes it may make it slower too.[7]

You can also include a *constructor* to make it easier to initialize a value of your new data type:

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Person {
    string name;
    int age;

    // constructor
    Person(string n, int a) {
        name = n;
        age = a;
    }
};

int main() {
    Person p("Jared", 15);
```

[7] If you want to learn more about why, feel free to ask in Discord.

```
    p.age = 16;  // you can still modify the attributes, of course

    cout << "The person named " << p.name << " is of age " << p.age << ".\n";
}
```

Note that **struct**s can contain other **struct**s as attributes:

```
#include <bits/stdc++.h>
using namespace std;

struct Dog {
    string name;
    int age;

    Dog() {}  // empty constructor

    Dog(string n, int a) {
        name = n;
        a = a;
    }
};

struct Person {
    string name;
    vector<string> name_of_children;
    Dog pet;

    Person() {}  // empty constructor

    Person(string n, vector<string> noc, Dog p) {
        name = n;
        name_of_children = noc;
        pet = p;
    }
};

int main() {
    Dog dog("Fluffy", 100);
    vector<string> children = {"Jay", "Red"};
    Person p("Jared", children, dog);
    p.name = "Jerrold";  // you can still modify the attributes, of course
}
```

This also shows that there may be more than one constructor.

3.1 Some behind-the-scenes `struct` details

I mentioned that conceptually, **struct**s are a way to pack together a collection of values into a single data type. Thinking about **struct**s this way is beneficial from the perspective of the programmer who will use it a lot.

Actually, as it turns out, it is how it's implemented by the compiler in C++ as well! That is, internally, to store a struct as a sequence of bytes, the program simply *packs together the bytes* comprising the structure.

We can see this happen by running the following VERY HACKY code, which forcefully converts, to an **int**, a value that's *not* of type **int** but nonetheless of the same size as an **int**. **Warning: This is very dangerous code: don't do this!** This is for illustration purposes only.

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Dog {
    char a, b;
};
struct Person {
    Dog pet;
    char c, d;
};
int main() {
    cout << "The size of a Person is " << sizeof(Person) << " bytes.\n";

    Person p;
    p.pet.a = 'U';
    p.pet.b = 'U';
    p.c = 'U';
    p.d = 'U';

    // crazy and dangerous hack to forcefully convert 'p' into an int
    int p_rep = *(int*)&p;

    cout << "The internal representation of Person p is " << p_rep << ".\n";
}
```

Running this program gives the following output:

```
The size of a Person is 4 bytes.
The internal representation of Person p is 1431655765.
```

Note that a **char** is 1 byte, while an **int** is 4 bytes,[8] so a `Person` and an **int** are of the same size. Indeed, the **sizeof** operator returns the size of a data type, and indeed, that's what the output says.

Also, we initialized all four bytes comprising `p` to be the byte representing the ASCII character `'U'`, that is, the byte `85`. And as it turns out, if you jam four of these `85`s together, you get the representation of the **int** value `1431655765`, which is indeed what the output shows.

Again, let me repeat: **Don't do this!** (You don't need to know what the scary expression `*(int*)&p` means since you won't be using it!)

3.2 Recursive `struct`s?

Above, we saw that a **struct** can contain other **struct**s as attributes. Can you make a *recursive* struct? For example, can you define something like the following?

```cpp
struct Person {
    int age;
    Person best_friend;
};
```

[8] In some weird computers or setups, **int** may have a different number of bytes. Stick to regular computers, please!

The answer is no! (You can even easily check this by trying to compile it and noticing that GCC refuses.) Indeed, given what we know about how a `struct` is handled internally, we can see why this is not possible in two ways:

- The size of a `Person` is the sum of the sizes of its constituents: an `int` and a `Person`. An `int` has a size of 4 bytes. What's the size of a `Person`? Well, it's the sum of the sizes of its constituents: an `int` and a `Person`. An `int` has a size of 4 bytes. What's the size of a `Person`? Well, it's the sum of the sizes of its constituents: an `int` and a `Person`. An `int` has a size of 4 bytes. What's the size of a `Person`? Well, it's …

  As you can see, this chain doesn't stop. We can only conclude that the size of a `Person` is $4 + 4 + 4 + \ldots = \infty$ bytes, which is impossible!

- Let p be the size of a `Person`, in bytes. The size of a `Person` is the sum of the sizes of its constituents: an `int` and a `Person`. An `int` has a size of 4 bytes. A `Person` has a size of p bytes. Therefore, we have the equation $p = 4 + p$, which has no real solution!

Overall, a `struct` definition only makes sense if its size is well-defined.

4 Computer memory

At some point in your training, you'll learn about things called **data structures**, which are ways to organize and manipulate data. Data structures are essential in algorithms; the right data structure can improve the running time of the same algorithm, which may mean the difference between "Time Limit Exceeded" and "Accepted".

To understand data structures, it's essential to understand how data is even stored and represented in the computer in the first place.

In this section, we'll delve a bit into how *computer memory* works.

The next few sections will describe quite a bit of detail into how a computer works at a somewhat low level. On your first reading, it is perfectly fine to skip them and jump over to the part titled "Summary".

4.1 What is a computer?

One of the things you quickly learn in your computer courses is that computers are dumb. This is unsurprising; a computer is basically just a machine that manipulates data (in the form of bits) into other data, as dictated by the code that some programmer wrote. Indeed, this is why informatics is called "informatics" in the first place—an algorithm is just a procedure that transforms some information (the Input) into some other information (the Output), and information can always be encoded in bits. At its heart, informatics is about manipulation of data.

Computers nowadays are very complex beasts, so in order to understand computer memory more easily, let's take a look at a simplified model of what a computer is. At its core, a computer is basically just *a device that can run exactly one program*. The program has access to a temporary storage area called the **Random Access Memory**, or **RAM**, which it can use as a sort of "scratch paper" to do its calculations. We call this *memory* for short.

The RAM is basically just a glorified array! A b-byte RAM is basically just an array of b "bytes". Each cell of this RAM array is indexed by the integers 0 and $b-1$. However, no one really calls it the "index". For some reason, they call it **address** instead.

Remember that a "byte" is defined as 8 "bits". Since 8 bits can represent any integer in the interval $[0, 255]$, this means we can view the RAM as just an array of b integers, each in the interval $[0, 255]$.

Notice that I said "exactly one program." You might object to this and say, "how come I can run multiple programs at once?" The answer is that, in fact, there's actually only one "master" program that's running in your computer right now. It's called the **operating system**, and it's a complex program that's responsible for *simultaneously simulating/running several programs* at once! It's also responsible for chopping up the RAM and assigning it to the programs, and also ensuring that the programs can only access their assigned portion of the RAM, never interfering with the others. Finally, even if a program's assigned portion of the RAM doesn't start at address 0, the operating system is also responsible for doing a bit of address "translation" so that the program can simply pretend that its assigned portion of the RAM begins at address 0. All in all, an operating system has a lot of responsibilities, and it's not a trivial task to write one.

Thankfully, you don't need to worry about that! From your perspective, as a programmer, you may simply pretend that your program is the only program running, that it has all the RAM to itself, and the addresses begin at 0.

4.1.1 What does compilation really do?

So a computer runs exactly one program. What programming language does it understand? Does it understand C++? Python? JavaScript? No! A computer only understands one language, which we call the **machine language**.

It's very hard to program directly into this machine language! The language itself consists of a few low-level instructions that do very simple things.[9] Among the difficulties with working in this language is the fact that the computer usually only has a small fixed number of predefined variables, called **registers**. To use more memory, you'll have to use the RAM, which means you have to directly work with RAM addresses using your limited registers. You cannot simply "declare a variable"! To add to this, the language itself is in binary, so you couldn't even read a program in machine language easily. As you can imagine, programming in an unreadable language while directly working with addresses with a limited number of variables is...cumbersome, to say the least!

Thankfully, you don't need to worry about all that. The **compiler** is the program that does the job of translating a human-readable program, such as a C++ program, into a program written in machine language. So when you run something like `g++ sol.cpp -o sol`, the "`sol`" program that the compiler produces is actually a program written in machine language. If you try to open that file in a text editor, you can't, because it's in binary![10]

As you can imagine, the job of a compiler is hard. It has the job of translating a line like `int ans = 42;` into something like

- Allocate an address a for `ans` in the RAM that's still unused.[11]
- Write the integer 42 at RAM address a.
- Whenever `ans` is read from later in the program, return the contents at RAM address a.
- Whenever `ans` is updated later in the program, update the contents at RAM address a.
- After exiting the function (or scope) that `ans` lives in, free up the address a for later use.

On top of this, the compiler also has to write instructions for managing the RAM (e.g., which parts are used and which are unused), and for remembering which variable is assigned to which address!

It's even more complicated in the case of more complicated C++ features. For example, for something like an array declaration like `int arr[25];`, the program has to allocate 25 contiguous addresses in the RAM![12] For a struct like

```
1 struct Person {
2     int age;
```

[9] In fact, these instructions depend on who made the processor in your computer, so there isn't even one unique "machine language"! Anyway, you don't need to learn about all these instructions, but if you're curious, the following link describes the instruction set that's available to most processors that Intel manufactures: https://en.wikipedia.org/wiki/X86_instruction_listings . There's a good chance that your computer runs on one of these Intel processors.

[10] Some text editors may be able to open it anyway, but they will only display the raw binary contents, or perhaps convert them to hexadecimal for a slight convenience. But it will still be unreadable.

[11] Well, technically, `int` is usually 4 bytes, so in fact, four consecutive bytes need to be allocated.

[12] Again, technically, it should be $25 \cdot 4 = 100$.

```
    int favorite_number;
} p;
```

it must allocate just enough memory to store two `int`s worth of data, and whenever, say, `p.age` is accessed, it must remember that it's the first `int`, not the second, etc.

But again, as a programmer, you don't need to worry about this, since it's the compiler's job. Instead, the more important takeaway for you here, as a C++ programmer, is that **every variable corresponds to some address in the RAM.**

Since addresses are 64-bit integers (this is why we say that a computer is a 64-bit computer, or a 32-bit computer, etc.), you can actually print the RAM address that a variable is located in, as a 64-bit integer! This can be done with the `&` operator, which returns the *address* of a variable. Well, it doesn't actually do exactly that—we'll go into it in a bit more detail later—which is why we have to typecast it into a `long long int` before printing:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int ans = 42;
    cout << (long long int) &ans << '\n';
}
```

I ran this program three times and this is what it printed:

```
140727576848500
140721580026836
140731721424820
```

As you can see, it doesn't always assign a variable to the same location every time. Also, recall that the operating system does a bit of address translation, so this isn't necessarily the actual physical location of `ans` in the actual RAM of your computer.

4.2 All memory in the RAM is the same

Most modern computers have anywhere between one gigabyte to thirty two gigabytes of RAM.[13] At its core, RAM is hardware that stores data. We won't go into details of how it works on a physical level, but at an abstract level it does two things: It responds to commands saying "Store these 8 bits in slot X" and "Retrieve the 8 bits in slot X". (But while programming, you don't have to think about these low-level commands—the compiler does it all.) The number of slots corresponds to how many gigabytes the RAM has.

The important thing to note is that "Store these 8 bits in slot X" makes no difference whether your 8 bits are supposed to represent integers or floats or strings. RAM simply does not care. It takes those bits and shoves them into the buckets. If you wanted to store more than 8 bits, it's going to take more instructions.

Okay, this is an oversimplification. There are actually 8 instructions, two of which are "Store these 8 bits in slot X" and "Store these 32 bits in slots X, $X+1$, $X+2$, $X+3$" and the rest follow that simple pattern, with the RAM supporting 8-bit, 16-bit, 32-bit, and 64-bit operations.

[13] Recall that a **gigabyte** is a billion bytes.

It's interesting to note that RAM is the reason why "try turning it on and off again" works. If RAM stored water in buckets, you can think of the buckets as having holes in them. If the power is turned on, someone is always refilling the filled buckets. But when the power goes off, all the buckets drain to empty, and whatever messed up state your computer was in is completely wiped out. And RAM is manufactured this way because of economic and technological reasons.

Again, we emphasize that **all memory is the same**. There is no separate memory for integers or floats or strings or arrays. The only difference is how we interpret regions of memory. Eight bytes can be taken to mean a string of eight `char`s, a single `long long`, two `int`s, or one `double`. What matters is what your program thinks it is.

4.3 Summary

Okay, it's time to move on. I'll reiterate the main takeaways:

- The RAM is a memory storage device, and is the only memory available to your program.
- The RAM is basically just one big array of bytes. (The RAM is pretty big; modern computers have gigabytes of RAM.)
- Each cell of the RAM is indexed by a 64-bit integer called an **address**.
- Every variable in your C++ code corresponds to some address in the RAM. Its contents are stored in that RAM location.
- An array or struct corresponds to a contiguous subarray of the RAM. As in the above, the variable itself that represents the array or struct corresponds to some address in the RAM, typically the address of the first byte.

These will be essential when you study pointers, which will itself be essential when you study data structures.

4.4 Pointers

Note that a RAM address is just a 64-bit integer. In other words, it's also just *data*, which means you can do with it pretty much anything you can do with data. For example, you can do arithmetic with it, square it, take its prime factorization, store it in another variable (as long as your variable is 64 bits long, such as a `long long`), etc. Of course, most of these operations are pretty useless when done on addresses. But the last one in particular is more useful; it allows our program to *point* to different parts of the memory!

To illustrate this, we first explain the unary[14] operators `&` and `*`.

- The expression `&x` returns the *address* of the variable `x`.
- The operator `*` is basically the inverse of `&`. It *undoes* an application of `&`.

We can test this.

[14] A *unary operator* takes one argument, e.g., negation (the $-$ in $-x$) is a unary operator. A *binary operator* takes two arguments, e.g., subtraction (the $-$ in $x - y$) is a binary operator. There can also be ternary operators (three arguments), quaternary operators, etc.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int ans = 42;
    cout << ans << '\n';
    cout << &ans << '\n';
    cout << *&ans << '\n';
}
```

If * is indeed the inverse of &, then we'd expect the first and third lines to output the same thing: 42. And indeed, they do:

```
42
0x7ffd2f857ef4
42
```

Now, you may have noticed the weird output in the second line. This is because the "definitions" I gave above for & and * are not quite the full picture. For example, you know that an address is a 64-bit integer, so you'd expect the type of &x to be, say, a **long long**. But that can't be it, because 0x never appears when you print a **long long**!

In fact, &x isn't a **long long**, and its type actually depends on the type of x itself. For example:

- If the type of x is **int**, then the type of &x is **int***.
- If the type of x is **double**, then the type of &x is **double***.
- If the type of x is **char**, then the type of &x is **char***.
- If the type of x is **int***, then the type of &x is **int****.
- etc.

Note that the '*' appearing here has nothing to do with the *unary* operator '*' I just mentioned earlier; this * is just part of the data type's name. The creators of C++ merely reused the asterisk character for two different purposes.

These **int***, **double***, **char***, **int**** things are called **pointer** types. Why are they called that? Do they point? To explain this, recall that behind the scenes, these are all just 64-bit addresses, so actually, all of these are the same *size*. It doesn't matter if it's **int***, or **double***, or **char***, or **int****...they're all 64 bits. The only purpose of calling them **int*** or **double*** or **char*** is so that you know that the 64-bit integer is supposed to mean something:

- If the variable y has type **int***, then the 64-bit integer represents a RAM address whose contents are supposed to be interpreted as an **int**. In other words, **y points to an int**.
- If the variable y has type **double***, then the 64-bit integer represents a RAM address whose contents are supposed to be interpreted as a **double**. In other words, **y points to a double**.
- etc.

In other words, the designation **int*** or **double*** or **char*** is basically just a tool to help the programmer remember that the variable points to an address whose contents are a certain data

type. That's why they're called "pointers". Behind the scenes, the computer doesn't care; as far as it's concerned, they're all 64 bits.

We can verify this by running the following program and checking that it outputs the same thing as our earlier program (ignoring the fact that the second line may be different).

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int ans = 42;
    int* pointer_to_ans = &ans;
    cout << ans << '\n';
    cout << pointer_to_ans << '\n';
    cout << *pointer_to_ans << '\n';
}
```

Now you understand what a *pointer* is, and how they work behind the scenes. Note that this is not the same as learning how to use them. We'll learn about that later.

4.5 Dynamic memory

So far, the only way we can obtain pointer is via the `&` operator. This is all well and good, but the problem is that the `&` operator only lets us point towards addresses occupied by some named variable! What if you want to point to an address that isn't occupied by any of your variables? Can you do that?

The answer is yes...Easily! Just choose a random 64-bit number, interpret that as an address, and convert it to a pointer. Chances are that no variable will occupy that address! For example:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int* pointer_to_ans = (int*)314159265358979LL; // some random address
    *pointer_to_ans = 42; // store 42 in that address
    cout << *pointer_to_ans << '\n';
}
```

Unfortunately, **this is a mistake**. Although address `314159265358979` is probably unoccupied by any variable, that doesn't mean you can use that address for your purposes! In fact, if you attempt to run this program, then you'll likely get a **Segmentation Fault** error, which means that your program is accessing parts of the memory it isn't supposed to access!

What's going on here is that behind the scenes, there's actually a part of the program that *manages* the memory. It is wholly responsible for the whole RAM, and knows which parts are being used, which parts are vacant, and which parts are off-limits to your program. You can think of this as a *librarian*. By accessing the address `314159265358979`, you're effectively bypassing the librarian, analogous to taking home a book without telling the librarian!

As it turns out, there's indeed a way to get access to unoccupied parts of the memory. However, you have to go through the librarian—you can't bypass them. If you want to obtain

access to some vacant chunk of memory, say 100 bytes, you have to request it from the librarian. The librarian will always grant you access (as long as there's enough available vacant memory), and will return the address of these 100 bytes.

However, although the librarian will always grant your request, it's the librarian who gets to choose which 100 bytes of memory to give, not you. In other words, you can't request for a *specific* section of 100 bytes that you like.

Okay, enough with metaphors. How does this translate to C++? It's simple: you just use the **new** keyword to request for memory![15]

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int *pointer_to_ans = new int[25]; // request 25 ints from the librarian
    *pointer_to_ans = 42; // store 42 in that address
    cout << *pointer_to_ans << '\n';
}
```

You can now verify that this works!

Here's an example that uses a **struct**:

```
#include <bits/stdc++.h>
using namespace std;
struct Person {
    int age;
    int favorite_number;
};
int main() {
    Person *p = new Person;
    (*p).age = 20;
    (*p).favorite_number = 6174;
    cout << "This " << (*p).age << "-year old likes the number " <<
    ↪  (*p).favorite_number << ".\n";
}
```

Since the expression `(*x).y` comes up so often when working with pointers, they added a convenient shortcut for it: `x->y`. Thus, we can also write it as follows:

```
int main() {
    Person *p = new Person;
    p->age = 20;
    p->favorite_number = 6174;
```

[15] Actually, there's a second keyword, **delete**, which does the opposite of **new**: it frees up memory. (You "return the books you borrowed".) Note that you can only use **delete** on chunks of memory that you obtained via **new**. In competitive programming, we don't need **delete** all that much since all memory will be automatically freed up when your program exits. However, for long-running program such as games, freeing up memory becomes important. If you don't, and you continuously request for more and more memory via **new** without freeing them via **delete**, then your program will run out of memory at some point, and it may crash, or it may slow down your computer. This bug is called a **memory leak**, and is notoriously difficult to debug since it doesn't make the program wrong; it just consumes more and more memory over time before anything goes awry.

```
    cout << "This " << p->age << "-year old likes the number " <<
    ↪  p->favorite_number << ".\n";
}
```

One other question you may ask is "who put the librarian there??" In fact, the compiler put them there; the librarian always exists in any compiled C++ code, whether you like it or not. (Though if you don't use `new`, then it doesn't really matter if they're there.)

4.6 A glimpse of linked lists

There's something interesting we can do with pointers and structs.

```
struct Chain {
    int val;
    Chain* bruh;
    Chain(int v, Chain* b) {
        val = v;
        bruh = b;
    }
};
```

To begin with, notice anything strange? We're defining a `struct` called `Chain`, but there also seems to be a `Chain` attribute, which means this can't be right; a recursive `struct` is invalid because it doesn't have a well-defined size!

However, looking closer, you notice that the attribute actually has type `Chain*`, not `Chain`. In other words, it's a pointer type, which means behind the scenes, it is just 64 bits in size. So our `struct` has a well-defined size after all: it's just $4 + 8 = 12$ bytes!

Okay that's good, but what can you do with it? Nice things! For example, you can do something like this:

```
int main() {
    Chain* x = new Chain(31, new Chain(4, new Chain(15, nullptr)));

    cout << x->val << '\n';
    cout << x->bruh->val << '\n';
    cout << x->bruh->bruh->val << '\n';
}
```

Note that `nullptr` is just a special pointer value which "points to nothing". As a 64-bit integer, it is represented by 0, but you should use `nullptr` in your code rather than 0 as a good practice.

The output of this program is:

```
31
4
15
```

Notice that we accessed the `bruh` attribute in sequence multiple times! The point of this example is to show that we can form chains of data pointing to each other via pointers. (The example above is an example of a *linked list*.)

We can even do something wackier like this:

```
int main() {
    Chain* x = new Chain(31, new Chain(4, new Chain(15, nullptr)));
    x->bruh->bruh->bruh = x; // point the last Chain back to the front!

    cout << x->val << '\n';
    cout << x->bruh->val << '\n';
    cout << x->bruh->bruh->val << '\n';
    cout << x->bruh->bruh->bruh->val << '\n';
    cout << x->bruh->bruh->bruh->bruh->val << '\n';
    cout << x->bruh->bruh->bruh->bruh->bruh->val << '\n';
    cout << x->bruh->bruh->bruh->bruh->bruh->bruh->val << '\n';
}
```

The output is as follows:

```
31
4
15
31
4
15
31
```

We've successfully made a circular chain of data pointing to each other!

4.7 A more birds-eye view of the situation

Let's take a step back. So far, we've learned the following:

- In addition to all the basic data types such as **int**, **double**, **char**, etc., there are also these things called *pointer* data types, obtained by appending the symbol *, e.g., **int***, **double***, **char***, etc.
- While regular data types vary in size (in terms of number of bytes), every pointer type is of size 64 bits (or 8 bytes).
- Of course, you can also have pointers to pointer types, such as **int****, or **double*****, or **char*******.[16] Each of them is also 64 bits in size, just like any other pointer.
- Every pointer variable *points* to a memory location containing some data. For example, a variable of type **int*** *points* to a memory location containing an **int** value.
- You can use the operator * on a pointer, say `*y`, to obtain the variable that it's pointing to. This operator has nothing to do with the * symbol appearing in the names of pointer types; they just happen to be the same symbol.
- You can use the operator `&` on a variable, say `&x`, to obtain the pointer to that variable.

[16] Although I don't know if that last one in particular has any good use, haha

Again, I think understanding all these details about pointers and addresses are fairly easy, but that's not the same as being able to naturally use them to solve actual problems. I think this is the part where a lot of students struggle. (So if you're struggling, don't worry! It doesn't mean you're bad at it.)

Anyway, if you're just learning about these things, then it's indeed hard to suddenly juggle these new tools and use them to good effect. Here are some of my thoughts on why this is hard:

- I think that juggling between `*` and `&` is hard for many beginners, especially if you're still imagining variables as being assigned to cells in the "RAM".
- Another pain point is ensuring that the declared variables and values you're assigning have the same type. If you make a mistake, and you're still just starting out, then it can be very tricky and confusing to fix it, especially with the unhelpful ~~garbage~~ error messages that GCC usually spits out.

So here's my advice: *only use a subset of these features for now.* That is, consciously forbid yourself from using certain features, until you've mastered the ones you're using. In fact, if I'm bold, I might go further and say that *you should forget some of the low-level details such as the RAM and addresses*, and just think of them in terms of high-level metaphors such as *pointers*, *librarian*, etc.

For a beginner, I recommend simplifying the picture a bit. Here are some of my recommendations:

- Forget about the fact that a pointer "contains the 64-bit address of a location in RAM". Think of a pointer as an *abstract* data type, whose only purpose is to *point* to another value.
- In fact, forget about the RAM altogether. Just think of it as an infinite void of nothingness (that you can use later).
- Forget about the operator `&` completely.
- Disallow pointer-to-pointer types, such as `int**`. In other words, only allow pointers to regular types (primitive types or structs) are allowed. Note that this also means it is now pointless to apply the unary `*` operator twice in a row.
- Actually, just disallow the operator `*` altogether.[17]

  Note that `x->y` is still allowed, even if it's merely an abbreviation of `(*x).y`; conceptually, it's somewhat different.

Note that we have completely forgotten about the operators `&` and `*`. Therefore, our **simplified C++ language** is now:

- The unary operators `&` and `*` **don't exist**.
- In addition to basic data types such as `int`, `char`, `MyStruct`, etc., there are also pointer types such as `int*`, `char*`, `MyStruct*`. Also, **there are no types with two asterisks**. For example, `int**` is not allowed.
- The RAM doesn't exist. There is only an infinite *void of unlimited memory*, guarded by a **librarian**.

[17] Actually there's one instance when we'll allow `*`: when your variable's type allows indexing. For example, if `x` is of type `string`, then `x[i]` is called "indexing", so if `y` is of type `string*`, then you can write `(*y)[i]` to index the string `y` is pointing to. **This is the only instance in which we'll allow the operator `*`.**

- You can request the librarian to grant you access to some portions of this void of memory. The librarian will always grant your request. There are two types of requests:
  - Write "`new S;`" to allocate space for a single value of type `S`, where `S` is some struct. This returns something of type `S*`, i.e., a pointer to an `S`.
  - Write "`new T[n];`" to allocate space for an array of `n` values of type `T`. It returns something that's written as "`T*`"; **but it's not a pointer**. Rather, it's a weird way of writing it as **an array of length `n` whose elements are of type `T`**.

    The only allowed operation with a variable of such a type is indexing, that is, `y[i]`. You may read to or write from this array as long as the index is between `0` and `n-1`.

- If `x` is of type `S` where `S` is a struct, then there's the "dot" syntax to access its attributes, such as `x.something`. The equivalent syntax if `y` is of type `S*` is `y->something`. **Think of `->` as the equivalent of `.` for pointers.**

- If `x` is of type `T` where `T` supports indexing, that is, `x[i]`, then the equivalent syntax if `y` is of type `T*` is `(*y)[i]`. **The `*` in this syntax doesn't mean anything; think of the syntax `(*y)[i]` as the equivalent of the syntax `x[i]` for pointers.**

- The main advantage to using a pointer over a regular data type is that it's **always** 64 **bits in size**, which means passing them around is cheap, unlike passing around heavier data types such as `string` or `vector` or `set`.

- However, the downside is that passing a pointer means you're...well...passing a pointer to the same piece of data. It doesn't pass a copy of the data itself. Thus, if you modify the piece of data is pointing to, everyone will see it. (On the upside, it lets different parts of your code share access to the same piece of data.)

I have chosen these to be sufficiently simple and at the same time powerful enough to let you do basically anything you need to do, at least at this point in your training/contest career.

Now, you may wonder, "why did you even bother explaining all those technical stuff when in the end you're just going to ask us to forget about them?" There are a few reasons:

- Learning about how it all works behind the scenes can reduce the "magic factor" of the computer. It's good to realize that the computer isn't a magic machine, but a very complex piece of engineering built from many layers of abstraction, and a lot of engineers' hard work.

- At some point in your programming career, you'll probably actually going to need knowledge of these more technical things.

- There's an important lesson here that's more directly applicable to competitive programming: Seeing the behind-the-scenes details exposes you to the very important idea of **abstraction**: the idea that you can separate a complicated program into multiple parts, each having separate roles, and all working together to achieve a goal.

  One of the key traits that differentiate a novice from an experienced person is the latter's ability to naturally decompose a problem into multiple subproblems, solve them separately, and combine them into a single whole. While solving a single subproblem, experienced competitors abstract away all other subproblems—they simply assume that the other subproblems have been solved already. Of course, it is still their responsibility to actually solve them, but the benefit is that splitting a problem into smaller, more manageable subproblems and then solving those is much easier than solving the full problem as a whole.

The fact that we can even think about the RAM and dynamic memory as an "infinite void guarded by a librarian" is an example of the power of abstraction: I don't need to think about a large-but-finite array consisting of bytes: I can just think about a virtually unlimited void. It makes things simpler from a programmer's point of view.

4.8 (Optional) Pass by reference

Okay, I lied. There's actually one more thing you might need, and it applies to a very specific situation.

However, this section is optional. There are ways to get around this specific use case using only what you've learned so far (pointers) so feel free to skip this part on first reading.

Consider the following relatively easy problem:

Problem 4.1. There is a string S, initially empty. You need to process a sequence of n operations. There are two types of operations:

- **Query**: Given i, find the ith character of S. It is guaranteed that $1 \leq i \leq length(S)$.
- **Update**: Given a character c, append c at the end of S.

The straightforward solution should pass:

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    int n;
    cin >> n;
    string s; // initially empty
    for (int op = 0; op < n; op++) {
        string typ;
        cin >> typ;
        if (typ[0] == 'U') { // update
            char c;
            cin >> c;
            s.push_back(c);
        } else { // query
            int i;
            cin >> i;
            cout << s[i-1] << '\n';
        }
    }
}
```

On my computer, this finishes in < 0.2sec on a random input with $n = 10^6$.

However, what if you decided to make a function instead? For example, suppose you updated your program into the following:

```cpp
...
char get_char_at(string t, int i) {
```

```
    return t[i-1];
}

int main() {
    ...
        if (typ[0] == 'U') { // update
            ...
        } else { // query
            int i;
            cin >> i;
            cout << get_char_at(s, i) << '\n';
        }
    ...
}
```

Basically the same, right? Unfortunately, when I run this modified code on the same (large) input, all of a sudden it takes > 6 seconds! What gives?

Unfortunately, although they look similar, there's actually a crucial difference. The `string s` can get rather large, but passing it as an argument actually *forces* the program to make a full copy of it, which is pretty expensive! This is because passing arguments to a function is "*pass by value*" by default. The variable `t` inside the function `get_char_at` is different from the variable `s` inside `main`, so they are assigned different locations in the RAM. Therefore, the program is forced to copy the whole string over from the location of `s` to the location of `t`, which is expensive!

To fix this, instead of pass by value, you want to *pass by reference* instead. The change is simple. We simply add the character `&` to the argument `string t` to indicate that it's supposed to be *pass by reference*!

```
...
char get_char_at(string& t, int i) {
    return t[i-1];
}
...
```

Passing arguments by reference is much faster; internally, only a 64-bit value is passed around. In other words, no copying of the full string is needed, and it becomes fast again!

Of course, the downside of passing by reference is now that *the two variables `s` and `t`* now refer to exactly the same piece of data, which means if you modify `t`, then `s` is also automatically modified, and vice versa. Sometimes, it isn't what you want. But sometimes, it is. For example, with pass-by-reference, we can now turn updates into its own function, as follows:

```
...

char get_char_at(string& t, int i) {
    return t[i-1];
}

void append_char(string& t, char c) {
    t.push_back(c);
}

```

```
int main() {
    ...
        if (typ[0] == 'U') { // update
            ...
            append_char(s, c);
        } else { // query
            ...
            cout << get_char_at(s, i) << '\n';
        }
    ...
}
```

Since the parameter `t` of `append_char` is passed by reference, it means any modification to it, such as `.push_back`, is also visible to `s`.

Now, you may notice that this feels suspiciously similar to pointers, in that only 64-bit values get passed around, and they refer (or point) to the same pieces of data. And indeed, you can alternatively use pointers to do basically the same thing.[18] However, the syntax if you do this with pointers can be rather cumbersome. For example, (at least with the "simplified C++" we discussed above) you must instantiate `s` via "`string *s = new string;`". Also, instead of `s[i-1]` and `s.push_back(c)`, you need to write `(*s)[i-1]` and `s->push_back(c)`. Both will work in roughly the same way, but the latter is harder and more awkward to type! Using references simplifies things a bit; the key thing to remember is that references are still *regular types, not pointer types*; they just happen to share the same location as some other variables, elsewhere.

Anyway, as I've mentioned above, learning about references is not needed if you're still new to pointers, so feel free to return to this section some other time for a second reading.

[18] In fact, internally, pointers and references are handled pretty similarly by the compiler.

5 Strings and characters

Conceptually, a "string" is just a sequence of characters. A character is a sort of symbol. If data is represented as bits, how are characters and strings represented on a computer?

5.1 Character encoding

The simplest idea to encode characters as bits is to assign a unique sequence of bits to each unique character that exists in the world. There are many characters that exist in the world. For example, in Chinese writing alone, there are already thousands of characters! However, at the time when computers were being developed and standards were being set, most programmers were still mostly in the English-speaking world, which means they only cared about a few characters: Latin letters (uppercase and lowercase), digits, and some symbols (most of the characters you see on your keyboard).

And so, they decided to assign bit sequences to these characters they care about; there are around a hundred of them. Since $100 < 2^8$, a single byte is enough to represent everything. And so they did! They assigned these characters to bytes (mostly arbitrarily) and declared this assignment to be a **standard code** that everyone will use when dealing with characters. They called it the **American Standard Code for Information Interchange**, or **ASCII**.

And so, due to historical reasons, ASCII became a universal standard. Nowadays, there are many more character encoding standards. The newer standards attempt to include the characters that weren't included in ASCII, such as the characters in other alphabets or writing systems, and even Chinese characters! But since ASCII is so ubiquitous, these newer standards are usually defined to be (mostly) compatible with ASCII.

As a competitor, you don't need to care about these modern character encodings; it's not the focus of algorithm contests after all. In competitive programming, essentially all input and output data will be ASCII-encodable and ASCII-printable, so for our purposes, we can assume everything is in ASCII.

In fact, many programming languages, such as C++, encode characters by their ASCII values by default. For example, `'U'` corresponds to ASCII value 85, so the following code succeeds.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    // assert simply checks that the condition is true, and raises an error
    ↪ otherwise
    assert('U' == (char)85);
    assert(85 == (int)'U');
    int v = 'U';
    assert(v == 85);
    char c = 85;
    assert(c == 'U');
    assert('U' + 'U' == 170);
    assert(40 + 45 == 'U');
    assert('S' + 2 == 'U'); // 'S' has ASCII value 83
```

```
}
```

Table 1 shows the complete ASCII table. However, you don't need to memorize it, not even a single one! As we've seen above, you can simply directly use the character literals such as `'U'` and do arithmetic on them as if they were regular integers.

For example, to check whether a character is a letter (either uppercase or lowercase), you can simply do the following:

```
bool is_letter(char c) {
    return ('A' <= c && c <= 'Z') || ('a' <= c && c <= 'z');
}
```

Since `char`s are represented by their ASCII values, we can use comparison operators such as `<=` on them. Well actually, this function assumes that the letters `A` to `Z` are found sequentially in that order somewhere on the ASCII table, as well as `a` to `z`. Luckily, this is true in ASCII. It's also true for the digits `0` to `9`. This is about the only thing you need to know about ASCII. You don't need to know their exact values; you just need to know that they're there, somewhere, on the table.

Exercise 5.1. Without looking at the ASCII table (no cheating by memorizing!), write a function that takes a single `char` and:

- if it's an uppercase letter, converts it to its corresponding lowercase version;
- if it's an lowercase letter, converts it to its corresponding uppercase version;
- otherwise, leaves it unchanged.

5.2 Strings

It should now be clear that the reason that a `char` is only a single byte in size is that all ASCII values fit in a byte. Now, how do we represent strings? In this section, we'll describe some ways to do so.

Now, before we proceed, let me say something important: As a competitive programmer, **you should use the builtin `string` data type from C++'s standard library!**[19] Thus, the discussion here is probably not directly important to contests, but learning about it is still somewhat useful, for similar reasons as those about pointers.

Recall that a string is fundamentally just a sequence of `char`s. So the most natural way to represent a string is to just use an array of `char`s. However, this runs into a subtle issue: As it turns out, C++ most basic *array* type—that is, the one it inherited from C—is pretty bare-bones. It doesn't even store its length! This means that you don't even know where it ends.

There are a few ways around this:

- We can use a special character to delimit the end of the string. Now, as it turns out, the creators of ASCII foresaw this need, so they invented a special character just for this task: the *null* character, with ASCII value 0. (As a `char` literal, you can write it as `'\0'`.) Thus,

[19] If you don't know about it or even what a "library" even is, you'll learn more about how to a bit later in this module (and training camp). Feel free to ask in Discord as well.

code	char	code	char	code	char	code	char	code	char
32	space	51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	”	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	’	58	:	77	M	96	‘	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

Table 1: Table of characters and their ASCII values. Characters below value 32 are special characters (e.g., the new line character) and are omitted. You can find the full table online. These ASCII values are $< 2^7$, but they're still always represented by 8 bits—the last bit will simply always be 0. That bit was left unused to allow future standards to be developed *on top* of ASCII—a smart decision on the part of the ASCII inventors.

to represent a finite string, you just have to use an array that's one byte longer than your string, and put `'\0'` at the end.

There are a couple of downsides to this.

- First, getting the length of a string becomes an expensive operation, since you have no choice but to iterate through the whole string to look for the null character. Worse, if the null character isn't there (perhaps due to some sort of mistake or bug), then you may even incur out-of-bounds errors since the program will just keep reading on and on, beyond the boundaries of the array!
- Next, you can't use the null character as a character in your string, because if you place it somewhere in the middle, then you're effectively cutting off the string at that point; your only indication of the end of the string is the null character, so it doesn't know that the characters after it (and indeed, the null character itself) still belong to the string.

Nonetheless, since it only costs an extra byte, and it's conceptually simple, this is the standard way strings in C are represented.

But C++ is not C. C++ has better ways to represent strings, described below.

- Another way would be to also store the length of the string separately. Now, the length of the string can be large, not just 255, so we may need an `int` or a `long long` to represent the length. But the upside is that we've avoided the problems above! We can now get the length of the string fairly quickly, and we can even include null characters in our string!

  To make this a clean implementation, you can even consider creating a `struct` that contains two attributes: the length of the string, and a `char` array containing the actual characters.

  Alternatively, you can just use a `vector<char>` since a `vector` automatically stores its length.

- Finally, although the previous solution is better than the first one, it's still not the best C++ approach. The best one is to simply use the builtin `string` type in the standard library! It has all the advantages of the previous solution, plus more, since you don't have to code anything, and it has other convenient builtin functions and methods to do certain tasks more easily![20]

[20] C++'s string-processing facilities are still worse than most other languages (like Python) though.

6 Recursion

In a general context, *recursion* is simply when something refers to itself. For example, in mathematics, the *factorial* of n, denoted by $n!$, is defined as the product of all positive integers from 1 to n, for example, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$, but can also be defined via recursion:

$$n! = \begin{cases} 1 & \text{if } n = 0 \quad \text{[let's call this "case 0"]} \\ n \cdot (n-1)! & \text{if } n > 0 \quad \text{[let's call this "case 1"]} \end{cases}$$

Note that in case 1, $n!$ is described in terms of $(n-1)!$. Also, case 0 is called the *base case*. In this case, this completely and unambiguously describes the factorial. For example,

$$\begin{aligned} 5! &= 5 \cdot 4! \\ &= 5 \cdot 4 \cdot 3! && \text{case 1} \\ &= 5 \cdot 4 \cdot 3 \cdot 2! && \text{case 1} \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! && \text{case 1} \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! && \text{case 1} \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 && \text{case 0} \\ &= 120 && \text{arithmetic} \end{aligned}$$

In programming, recursion is when a function calls itself. For example, we can implement a factorial with a loop:

```
long long factorial(int n) {
    long long res = 1;
    for (int i = 1; i <= n; i++) {
        res *= i;
    }
    return res;
}
```

But we can also implement it using recursion, by calling the same function inside itself, like so:

```
long long factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Note how closely it resembles the recursive definition!

By the way always ensure that there is a base case in your recursive function, otherwise, the function will just keep calling itself over and over until your program crashes via a Segmentation Fault error (though this is not guaranteed). This error is called *stack overflow* because every time you call a function, some data (mainly function arguments and local variables) is stored in the stack memory. (If you don't know what the "stack memory" is, don't worry for now.)

6.1 Application: Enumeration and brute-force

In C++, there isn't a big difference between ordinary functions and recursive functions; functions can definitely call other functions inside them, and there's no reason why a function should not be able to call itself. Despite this, some beginners find recursion confusing. Other beginners understand them but have trouble implementing them.

That's all fine! It takes a bit of practice, but it's totally worth learning it because it's very useful.

For instance, suppose you are given the following problem: *Given an integer n, print all the permutations of $0, 1, \ldots, n-1$.* This task can be accomplished in many ways, but it's instructive to learn how to perform this task using recursive functions.

But first, how would we go about implementing this task without functions? Well, for example, for $n = 4$, we might want to write something like this:

```
for (int a = 0; a < 4; a++) {
  for (int b = 0; b < 4; b++) {
    if (b != a) {
      for (int c = 0; c < 4; c++) {
        if (c != a && c != b) {
          for (int d = 0; d < 4; d++) {
            if (d != a && d != b && d != c) {
              cout << a << ' ' << b << ' ' << c << ' ' << d << '\n';
            }
          }
        }
      }
    }
  }
}
```

There are four loops, each one enumerating one of the values of the permutation, and the `if` statements ensure that we're making a permutation. However, there are several issues with this code:

1. For larger n, the number of nested loops will increase, and the whole thing becomes harder and harder to read (and debug!).
2. More importantly, this does not *scale* well; we will have to write brand new code for every distinct n, and clearly we can only write so much. What we want is to write code that can handle *all n at once*.

But this task is easy if we can use recursive functions! The following is a recursive implementation. I highly encourage you to understand it:

```
const int N = 100;
int seq[N];
bool taken[N];

void enumerate_permutations(int n, int i) {  // i is the 'current index'
    if (i < n) {
        // enumerate all possibilities for seq[i]
        for (seq[i] = 0; seq[i] < n; seq[i]++) {
            if (!taken[seq[i]]) {
                taken[seq[i]] = true; // mark seq[i] as taken
```

```
                enumerate_permutations(n, i + 1);
                taken[seq[i]] = false; // unmark seq[i]
            }
        }
    } else {
        // we've enumerated everything. now print the created permutation.
        for (int j = 0; j < n; j++) {
            cout << seq[j] << ' ';
        }
        cout << '\n';
    }
}
```

The initial call would be `enumerate_permutations(n, 0)`, i.e., we start enumerating at index 0.

6.2 Mutual Recursion

Another thing: there's also this thing called *mutual recursion*. This occurs when you define a function `f` that calls another function `g`, which also somehow calls `f` again. This is useful in some cases, and the idea is the same. However, when implementing this in C++, you first have to declare one of the functions so that it's visible to the other. (This is similar to how you can't write `int x = y; int y = x;` but you can write `int y; int x = y; y = x;`)

For example, here's a (slow) solution that attempts to solve the problem UVa 847 - A Multiplication Game:

```
// declare the function 'ollie_wins'
bool ollie_wins(long long n, long long v);

// define the function 'stan_wins'
bool stan_wins(long long n, long long v) {
    for (int m = 2; m <= 9; m++) {
        if (m * v >= n or (not ollie_wins(n, m * v))) {
            return true;
        }
    }
    return false;
}

// define the function 'ollie_wins'
bool ollie_wins(long long n, long long v) {
    for (int m = 2; m <= 9; m++) {
        if (m * v >= n or (not stan_wins(n, m * v))) {
            return true;
        }
    }
    return false;
}
```

Note that `ollie_wins` is declared first so that it is visible inside `stan_wins`.

By the way, note that there are better ways to so this problem. This is just an illustration of mutual recursion. At some point, you'll encounter situations/tasks where the simplest and most intuitive implementation is mutual recursion.

Longer chains of mutual recursion are possible. Complex webs of functions mutually calling each other are also possible. You only need to ensure that all functions have been declared beforehand.

7 The C++ Standard Template Library

Now, we will learn about the **C++ Standard Template Library**, or STL for short. The STL contains some useful common data structures and algorithms that you can use in your programs, so that you don't have to spend time implementing these yourself, and therefore make your code shorter, and therefore solve your problems faster. The STL is technically not part of the plain "C++ language", but as they are available almost everywhere C++ is available, for the most part you can think of them as if they were part of the language.

According to Wikipedia, a real-world **library** is "a collection of sources of information and similar resources, made accessible to a defined community for reference or borrowing." You have probably borrowed books from a library sometime in your life. In programming, you can think of a **library** as a collection of code that is made available to all users of a particular programming language (a.k.a. you), so that these users (a.k.a. you) can borrow or *include* these pieces of code in their programs. So that you don't have to write them yourself. So that you don't need to reinvent the wheel. Normally, these collections are code for common functionality that is common enough to be useful in *lots* of programs, but not common enough to be useful in *all* programs, so they are not made part of the language, but have to be included separately. Like toys with no batteries included, C++ is! (You can't even read and print stuff without using the C++ standard library `iostream`.)

Speaking of which, you've already used the C++ standard library without even realizing it. If you've ever done `#include <iostream>` and used `cin` and `cout`, then you were using the C++ standard library. In general, to include functions from a C++ library, you write `#include <name-of-library>` at the beginning of your C++ file.

Note that this means you'll be including potentially many libraries, which can be cumbersome (and frustrating if you don't remember the library name). Fortunately, there is a shortcut for including *the whole standard library*:[21]

```
1 #include <bits/stdc++.h>
2 using namespace std;
```

For contests, this is more convenient since you don't need to remember the exact libraries that contain the features you need. However, it slows down the compilation process a bit. Fortunately, it doesn't really affect the runtime of the actual program all that much, so for the purposes of competitive programming, it's always a good idea to just do it all the time.[22] In what follows, we'll assume that this has been done, and never bother telling the exact library name to include. (You can always look it up.)

"**Standard**" just means that a bunch of guys met together, wearing suits, and decided that some library features are so useful, they mandated that all implementers of C++ who wish to be respected must include them with every installation of C++. What is a programming language implementer, you ask? They are, for example, people who write `g++` (the compiler is, after all, just another program, albeit a complicated one), and in general, everyone who writes programs that compile C++ programs. *Standard* means you can count on these features to be already installed when you install a good C++ compiler on your computer.

There are also *third-party* libraries, which are also quite useful but not as widely useful, so they are not included with every C++ installation, and you need to install these manually if you want to use them. One example is the GNU Multiple Precision Arithmetic Library, which

[21] The `using namespace std;` part has something to do with "namespaces"; you don't need to worry much about it, though you can ask in Discord if you're curious.

[22] For (moderately) obvious reasons, it is discouraged in real-world C++ applications.

lets you perform arithmetic on numbers larger than 2^{64}, among other things. But we don't have third-party libraries in most programming contest settings, especially the IOI, so we will not talk about or use them.

If you want to delve deeper into what a programming language really is and why we do this weird business of separating the language itself from libraries, check this YouTube playlist out.

The "**template**" part has to do with the `template` feature of C++, which is a bit too technical at this point, so we won't explain it yet. It will be useful later on, though.[23] For now, let's just go with a fake explanation. Let's just say that "template" means a particular part of the C++ standard library that includes data structures and algorithms. Since data structures and algorithms are the ~~bread and butter~~ *kanin and toyo* of competitive programming (and programming in general), it's very useful. Or we can also say that "template" means that C++'s STL features can be used through a common pattern, which makes it easy to use. We will learn more about this pattern (called "iterator pattern") later. For a competitive programmer (a.k.a. you), practically, this means that code using the STL look very similar to each other, even if they are using different features of the STL. Hence, less memorizing.

You can check this video (and the entire playlist) out if you want to delve deeper into C++ templates (and the STL).

7.1 Mathematical Functions

The `cmath` library contains a bunch of useful mathematical functions, as commonly found in a scientific calculator, and more. When you `#include <cmath>`, it is like magically converting your standard C++ calculator into a scientific C++ calculator, so that you can do more fun stuff. The `cmath` library is technically a C library and not part of the C++ STL, but that's not too important. (Also, remember that you don't need to `#include` it explicitly since we've already included the whole STL.)

You will almost never need `cmath` for the IOI, but it's good to check it out and see what it contains, just so you know it's there in case you need it. Also, you might use it in the future.

7.2 Pairs

7.2.1 Theory

There are many situations wherein you would want to define and use ordered pairs. Rather than creating arrays with two elements, it is sometimes cleaner to use a `pair` object. Also, there are cases where you might want to group together two items of *different types*. An array, which is all about grouping together items of the *same type*, is not quite the right concept for such a grouping. The C++ Standard Library contains a built-in `pair` object for your convenience.

7.2.2 How to Use C++ Built-in Pairs

Using `pair` is easy. The type of a pair variable is `pair<T1, T2>`, where `T1` is the type of the first member of the pair, and `T2` is the type of the second member of the pair. To make a pair, just use the function `make_pair` (you don't say). To get the first member of a pair, apply `.first` on the pair. You can probably guess how to get the second member of a pair.

[23] As usual, you can ask in Discord if you're curious.

Sample code:

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    pair<int, int> lattice_point = make_pair(1, -1);
    cout << lattice_point.first << " " << lattice_point.second << '\n'; // 1 -1

    lattice_point.first = 2;
    cout << lattice_point.first << " " << lattice_point.second << '\n'; // 2 -1

    pair<string, int> name_age = make_pair("Aldrich", 20);
    cout << name_age.first << ", " << name_age.second << '\n'; // Aldrich, 20

    pair<pair<int, int>, int> nested_pair = make_pair(make_pair(1, 2), 3);
    cout << nested_pair.first.first << " " << nested_pair.first.second << '\n'; // 1 2
    cout << nested_pair.second << '\n'; // 3
    return 0;
}
```

There's a slightly more convenient syntax to create pairs (though there are cases when the compiler rejects it, and you have to use `make_pair`), and something called "unpacking", demonstrated below:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    // special syntax for creating a pair
    pair<int, int> lattice_point = {1, -1};
    cout << lattice_point.first << " " << lattice_point.second << '\n'; // 1 -1

    // unpacking assignment: assigns .first and .second to variables x and y
    // x and y will be ints (automatically inferred by the compiler)
    auto [x, y] = lattice_point;
    cout << x << " " << y << '\n'; // 1 -1

    return 0;
}
```

7.2.3 How They're Implemented in C++

Under the hood, C++ pairs are actually **structs**. If you want to group items into a pair and give meaningful names to each member (not just `first` and `second`), you would create your own **struct** instead. Also, if you wanted to group together three or more items, nesting pairs, as we did above, gets messy and hard to read and debug. For these cases, you would use a **struct** or `tuple` instead. But, for many cases where you need pairs of items, using `pair` is a convenient choice. The benefits of using pairs will become clearer later when we talk about graph algorithms. Thinking about pairs will also help you understand the `map` data structure below.

7.2.4 Gotchas

Modify the code above to directly print `lattice_point`. That is, try `cout << lattice_point << '\n';`. What happens? You notice that `pair`'s can't be directly fed into `cout` or read from

`cin`.

Create a nested pair where the second element is a pair, instead of the first one:

```
pair<int, pair<int, int>> nested_pair;
```

What happens? In version C++03, this throws a compile error. This is due to a design flaw in C++03 that treats all instances of `>>` in the code as the operator `>>` (e.g. the one you use with `cin`). If you are using C++03, to properly disambiguate the angle brackets used for types and the angle brackets used for the `>>` operator, you must put a space between the two closing brackets:

```
pair<int, pair<int, int> > nested_pair;
```

This is fixed in newer versions of C++ starting from C++11. If you didn't encounter this error, then great! Your compiler uses C++11 (or later) by default. If you did, then you should either add the space, or explicitly tell your compiler to use, say, C++17 instead:

```
g++ -std=c++17 program.cpp
```

Or, if you have an even newer compiler, use `c++20` (or `c++23`).

The same gotcha applies to all the different data structures below. Most modern programming contest environments, including the IOI, have compilers that fully support C++17 (or later). So, you might want to make it a habit to always compile with `-std=c++17`.

If you're reading this into the far future, the newest version might have already updated, so adjust this lesson accordingly!

7.3 Vectors

7.3.1 Theory

Arrays are great, but they require you to specify the maximum size in advance. In cases where you cannot predict this maximum size, you would need a list that allows you to keep adding as many elements as you need, by changing its size on demand.

One way this is done is through a linked list. But a linked list has a serious disadvantage: retrieving elements from the middle of the list requires $\mathcal{O}(n)$ time. In technical terms, a linked list does not have efficient *random access*.[24]

Another way is through dynamic arrays. The idea is, just initially allocate space for some small number of elements. Whenever you need more space, create a new array that is bigger than your old array, and copy all the elements from the old array to the new array. To save space, after removing a lot of elements, create a new array that is smaller than the old array, and again copy all the elements from the old array to the new array. If you want to see this in greater detail, check this YouTube playlist out.

After going through this entire lesson, you may want to check out this video and this video, to really understand why this is fast even though it seems like we are doing $\mathcal{O}(n)$ work per operation to copy items.

[24] And RAM is called *random access* memory because accessing any part of it is (roughly) as fast as accessing any other part.

Luckily, C++ has a built-in dynamic array. It's called `vector`. Like arrays, you can have vectors of any type. You can have vectors of integers, vectors of pairs, even vectors of vectors and vectors of vectors of vectors of vectors. Though in those last two cases, it's probably easier to work with multi-dimensional arrays instead.

7.3.2 How to Use C++ Built-in Vectors

Most of you are probably already familiar with vectors. (If not, check this out.) Aside from `vector`, the video also discusses a bunch of other "container" types for sequences. But in competitive programming settings, only `vector` and `deque` are widely used.

7.3.3 How They're Implemented in C++

Behind the scenes, a `vector` grows and shrinks by dynamically allocating new arrays of a new size, and copying the old array into the new array. For this reason, they are slower than regular arrays, but the speed difference is usually not significant enough to cause performance problems and TLE's. A `vector` also keeps track of its size so you can easily get the size in $\mathcal{O}(1)$.

7.3.4 Gotchas

Because `vectors` grow and shrink through dynamic allocation, there are annoying rare cases when this causes memory problems. Especially if you are solving a problem that requires processing multiple test cases, take care to cleanup the vector after you are done using it. No, you don't use `myvector.clear()`. Instead you need to do `vector<type>().swap(myvector)`. The reason is too technical. Just trust us for now.

7.4 Stacks, Queues, and Double-Ended Queues

7.4.1 Theory

Stacks, queues, and double-ended queues are limited versions of linked lists and vectors, where you can only access, insert, and delete at one or both ends of an array. Why would you ever want a more limited version of a data structure? One reason is that it is conceptually clearer that your intention is to just operate on the ends of the list, rather than randomly accessing elements in the middle. You also avoid bugs that may occur if accidentally do access elements in the middle, when your intention is to only access elements from the ends. There is also a very beautiful connection between these restricted data structures and graph algorithms, which you will see later.

Check this video out to see how stacks, queues, and double-ended queues work.

7.4.2 How to Use C++ Built-in Stacks, Queues, and Double-ended Queues

If you watched the video about `vectors`, `deques`, and other sequence containers from the previous section, it should be very easy to understand the following example programs for `stack` and for `queue`.

Double-ended queues can be used in two ways. This and this should make that clear. These two ways are not mutually exclusive. You can insert, access, and delete from either end at any time.

7.4.3 How They're Implemented in C++

Stacks, queues, and double-ended queues in C++ are generally implemented using dynamic arrays rather than linked lists. (Of course, they can also be implemented with linked lists, and with basically the same big $\mathcal{O}$ running time, but it's still a bit slower.)

7.4.4 Gotchas

The same gotcha for `vector` applies to `stack`, `queue`, and `deque`.

Don't forget to check first if the data structure actually has elements in it before popping.

7.5 Priority Queues

7.5.1 Theory

Priority queues are a generalization of queues, where in each dequeue operation, instead of accessing or removing the element that has been in the queue for the longest time, we access or remove the element with the highest "priority". The priority rule can be anything we like. It can be highest value first, or lowest value first, or some other rule. It is quite easy to implement this with arrays or vectors, so that either insertion or access and deletion takes $\mathcal{O}(n)$ time. But if we needed to do lots of operations, this is too slow. C++'s `priority_queue` performs each operation in $\mathcal{O}(\lg n)$ time, which is faster.

By default, `priority_queue` prioritizes elements by highest value first.

Priority queues are usually implemented with a data structure called a *heap*, which can perform each operation in $\mathcal{O}(\lg n)$ time. Watch this or this to learn how it achieves this.

7.5.2 How to Use C++ Built-in Priority Queues

If you understand how to use stacks and queues, then using priority queues is fairly straightforward. This sample program should be easy to understand.

7.5.3 Defining Priority

"Highest value" makes sense for numbers, but what does it mean for strings and other kinds of data? For strings and vectors, lexicographic order is the default rule. For pairs, the first elements are compared first. If they are tied, then the second elements are compared. And so on. (If one of them runs out before the other without finding a tiebreaker, the shorter one is deemed the "smaller" one.)

What about for `struct`'s and objects that you define yourself? C++ knows nothing about your custom objects and how they should be ordered, so it asks you to specify the rules yourself. There are several ways to specify these rules, and Ashar Fuadi (a competitive programmer from University of Indonesia) has a nice blog post about it.

7.5.4 How They're Implemented in C++

Under the hood, `priority_queue`s are implemented using binary heaps.

7.5.5 Gotchas

The order in which two equal elements are retrieved from a priority queue is not specified. Any one of them can come before the other.

Don't forget to check first if the data structure actually has elements in it before popping.

7.6 Sets and Maps

7.6.1 Theory

Vectors and lists are *indexed* collections of items. If you didn't care about the positions of items, but instead would like to check for the existence of items or *keys* really quickly, then you would use a set instead. A set allows you to store a collection of items, and quickly determine if your collection contains a certain key or not.

Maps are similar to sets, but in addition to storing just keys, you can also store a *value* associated with each key. When you go look for a key, the map will also tell you what the value associated with the key is, if the key exists in the map. You can think of a map as a generalization of an array, where instead of associating integers from 0 to $n-1$ to objects, you are associating characters, strings, or any arbitrary member of the key type to objects. Like sets, maps are able to quickly check for the existence of a certain key and retrieve values associated with that certain key, though not quite as quickly as an array can retrieve values associated with certain integers.

Maps can also be used as *sparse arrays*, where you can store n items in "positions" 0 to $N-1$, using only $\mathcal{O}(n)$ space rather than $\mathcal{O}(N)$ space. If $n \ll N$, this is a significant saving and can spell the difference between feasible and infeasible solutions.

In order to support insertion, deletion, and lookup of keys in $\mathcal{O}(\lg n)$ time per operation, sets and maps are typically implemented using binary search trees. Check this video out to learn about them.

7.6.2 How to Use C++ Built-in Sets and Maps

See this video for an overview of sets and maps. For more details, check out TopCoder tutorials for `set` and `map`.

7.6.3 How They're Implemented in C++

Under the hood, a `set` is a binary search tree of keys of the specified type, while a `map` is a binary search tree of pairs, where the first element of each pair is a member of the key type, and the second element is a member of the value type.

Since `set`s and `map`s are implemented using binary search trees, you can actually do another interesting thing with them: finding the key nearest to a given query key, in $\mathcal{O}(\lg n)$ time. There are two variants for this: `upper_bound` and `lower_bound`. Here is the reference for set. Map works similarly.

7.6.4 Gotchas

If you use custom `struct`'s or objects as keys to your sets and maps, you need to specify a custom comparison function, like you would for `priority_queue`. Only very rarely would you actually need this. But precisely because you only do it very rarely, it's very easy to forget. Maybe the lesson on graphs and graph search will remind you about this again.

7.7 Iterators

As we mentioned earlier in this document, the nice thing about the C++ STL is that there is a common way to use all the data structures above, and to use them with all the algorithms below. That way is through what is called an *iterator*. An iterator is like a pointer, but fancier.

See this video to learn how they work.

7.8 Sorting

7.8.1 Theory

This is probably not new to you. You're already probably convinced of the usefulness of sorting, and may have in fact already used the C++ STL `sort` function before. You've probably also heard that sorting takes $\mathcal{O}(n \lg n)$ time, but you're not sure why. If you're curious and want to know why, watch this video.

7.8.2 How to Use C++ Built-in Sorting Methods

Check this video out to see all the various ways you can sort using the C++ STL. You can again define your own sorting rule, like you would for priority queues, sets, and maps, if you wanted to override the default ordering, or if you were using custom `struct`'s or objects.

7.9 Permutations

7.9.1 Theory

When you want to do a brute-force solution, you sometimes need to check all possible permutations of a list of items. While it is possible to write your own short function that does this, it is not such a good idea, especially under contest pressure. It is better to use a built-in function so that you don't spend time implementing and debugging your own permutation generator. (However, this doesn't mean you shouldn't learn recursion!)

7.9.2 How to Use C++ Built-in Permutation Generators

The sample program here should be easy enough to understand. Again, if you are permuting custom-defined objects, you also need to specify a custom comparison function.

7.10 Miscellaneous Helper Functions in the C++ STL

For programming contests, `sort` and `next_permutation` are the most useful functions of the algorithms library. Another set of important functions are `binary_search`, `lower_bound`, and `upper_bound`, which implement binary search on sorted sequences. Binary search is deceptively simple to implement on your own, but under contest pressure, even the most experienced coders can sometimes implement them incorrectly and waste a few minutes trying to debug their binary search implementation. I therefore recommend reading the C++ reference for them and learning how to use them, to give you a slight advantage in programming contests. They are covered in the first few minutes of this video. There are also a bunch of other functions that are not as useful, in the sense that you will still be able to write solutions quickly without them, but they are good to know and they can spell the difference between writing code within 5 minutes versus writing code within 4 minutes. They are the following, in order of usefulness: `min`, `max`, `fill_n`, `fill`, `copy`, `reverse`, `count`, `count_if`, `find`, `find_if`, `for_each`. Check the complete list of available functions in the `algorithms` library.

7.11 Miscellaneous tips

Be careful in naming your variables when you are `using namespace std` and the C++ STL. Since you'll be including the STL libraries in your program, there is a chance that you might name your variable using one of the names already used by the above libraries. For example, you don't want to name your variables `min`, `max`, or `count`, because functions with these names exist in the algorithms library. Your variable name and `std`'s name will clash, and you will get a weird compile error. To avoid having to deal with this, you can either stop `using namespace std`, or stick to a naming convention that is weird enough to ensure that your names never clash with `std`'s names.[25]

TopCoder has an excellent tutorial on using the C++ STL for competitive programming, here and here. If you need more information and examples, I recommend checking them out.

[25] It's important to note that `using namespace std` is not recommended when writing real-life code, but is highly recommended when writing competitive programming code. Just make sure to not mix up the two!

8 Homework

8.1 Instructions

- Items labelled with A, B, C, D are to be prioritized.
- For items labelled with D: use the C++ STL to make your code as short as possible.

8.2 Tasks

A1: Linked List

Implement a linked list of **int** from scratch. In particular, fill out the commented sections of the code below to make it work. You are free to add any structs you want, but don't modify the function signatures.

```cpp
#include <bits/stdc++.h>
using namespace std;

struct linked_list {
    // Fill out
    int length;
};

linked_list* linked_list_create() {
    // Fill out
}

void linked_list_destroy(linked_list* list) {
    // Fill out
    // Make sure you don't have memory leaks!
}

void linked_list_add(linked_list* list, int value) {
    // Add something to the end of the linked list
    // Fill out
}

void linked_list_remove(linked_list* list, int index) {
    // Fill out
}

int linked_list_get(linked_list* list, int index) {
    // Fill out
}

int main() {
    // Input is lines of the form:
    // add v
    // remove idx
    // get idx
    // exit

    linked_list* list = linked_list_create();

    while (true) {
        string instr;
        cin >> instr;
        if (instr == "add") {
            int v;
            cin >> v;
            linked_list_add(list, value);
```

```
47          } else if (instr == "remove") {
48              int idx;
49              cin >> idx;
50              linked_list_remove(list, idx);
51          } else if (instr == "get") {
52              int idx;
53              cin >> idx;
54              cout << linked_list_get(list, idx) << '\n';
55          } else if (instr == "length") {
56              cout << linked_list->length;
57          } else if (instr == "exit") {
58              break;
59          }
60      }
61      linked_list_destroy(list);
62  }
```

Bonus: Refactor this code to use *methods* inside the `struct` instead of global functions. (Rename them, of course.)

A2: Jagged Array

- Allocate a 2D array using `new`.
- Allocate a jagged array (i.e., a 2D array with each subarray not having equal size) using `new`, where row i has space for exactly $(i + 1)$ elements.

A3: Array Manipulation

The main goal of this task is to give you practice, so don't use the STL to perform the following operations.

Write some functions that take in an array of `int`s and performs some manipulations.

- **draw.** Print the contents of the array.
- **fill.** Given a value `v`, set all values of the array to `v`.
- **reverse.** Reverse the array.
- **resize.** Given a value `k`, create a copy of the array but with `k` extra slots at the end, and return it. Initialize the values to `0`. Keep the original array intact.
- **rotate.** Right-rotate the array, i.e., move each element to the right. The last element goes to the first element.
- **rotate k.** Given a value `k`, right-rotate the array `k` times. The running time must be independent of `k`, so for instance, `rotate(1)` must run in almost the same amount of time as `rotate(100)` or `rotate(2000000000)`.
- **insert.** Given a value `v` and an index `i`, return a new array with the value `v` inserted at index `i` and the elements after that being moved one index to the right (thus increasing the array length by 1). Keep the original array intact.
- **histogram.** Assume the array values are nonnegative, and let m be the largest value in the array. Return an array of length $m + 1$ such that the ith value represents the number of times i appears in the original array. Keep the original array intact.

A4: 2D Array Manipulation

The main goal of this task is to give you practice, so don't use the STL to perform the following operations.

Write some functions that take in a 2D array of `int`s and performs some manipulations. Note that `A[i][j]` denotes the element at the ith row and jth column of `A`. The 0th row is written at the top and the 0th column is written at the left.

- **draw.** Print the contents of the 2D array, as described in the previous paragraph.
- **rotate clockwise.** Return the 2D array formed by rotating the array clockwise. Keep the original array intact.
- **rotate counterclockwise.** Return the 2D array formed by rotating the array counterclockwise. Keep the original array intact.
- **flip horizontal.** Return the 2D array formed by flipping the array along a horizontal line. Keep the original array intact.
- **flip vertical.** Return the 2D array formed by flipping the array along a vertical line. Keep the original array intact.
- **flip diagonal.** Return the 2D array formed by flipping the array along the diagonal line pointing down-right. Keep the original array intact.

B1 Basic Tower of Hanoi: https://www.hackerrank.com/contests/noi-ph-2018-preselection/challenges/basic-tower-of-hanoi

B2 Basic Subset Enumeration: https://www.hackerrank.com/contests/noi-ph-2018-preselection/challenges/basic-subset-enumeration

B3 Basic String Enumeration: https://www.hackerrank.com/contests/noi-ph-2018-preselection/challenges/basic-string-enumeration

---

C1 A Multiplication Game: (UVa) Online Judge 847

C2 One-Handed Typist: (UVa) Online Judge 11278

C3 Caesar Cypher: (UVa) Online Judge 554

C4 Traveling Supermodel Problem: https://www.hackerrank.com/contests/noi-ph-2018-preselection/challenges/traveling-supermodel-problem

---

D1 Vasya and Wrestling: https://codeforces.com/problemset/problem/493/B

D2 I Can Guess the Data Structure!: (UVa) Online Judge 11995

D3 Indian Summer: https://codeforces.com/problemset/problem/44/A

D4 Inna and Alarm Clock: https://codeforces.com/problemset/problem/390/A

D5 CD: (UVa) Online Judge 11849

D6 Misha and Changing Handles: https://codeforces.com/problemset/problem/501/B

D7 Hardwood Species: (UVa) Online Judge 10226

D8 Worms: https://codeforces.com/problemset/problem/474/B

D9 Gravity Flip: https://codeforces.com/problemset/problem/405/A

D10 ID Codes: (UVa) Online Judge 146

D11 Shower Line: https://codeforces.com/problemset/problem/431/B

E1 Anagrams by Stack: (UVa) Online Judge 732

E2 Ferry Loading III: (UVa) Online Judge 10901

E3 Ferry Loading IV: (UVa) Online Judge 11034

E4 Argus: (UVa) Online Judge 1203

E5 Conformity: (UVa) Online Judge 11286

E6 Hoax or what: (UVa) Online Judge 11136

E7 An express train to reveries: https://codeforces.com/problemset/problem/814/B

E8 Sagheer, the Hausmeister: https://codeforces.com/problemset/problem/812/B

E9 Dynamic Problem Scoring: https://codeforces.com/problemset/problem/806/B

E10 Godsend: https://codeforces.com/problemset/problem/841/B

E11 String Reconstruction: https://codeforces.com/problemset/problem/827/A

E12 Okabe and Banana Trees: https://codeforces.com/problemset/problem/821/B

A C++: A low-level approach

C++ is a language that is mostly a superset of a lower-level language called C. This section attempts to discuss the lower-level aspects of C++, mostly those it inherited from C.

Note: This section is not required. You may skip it. If you're still starting out, I recommend skipping it for now.

A.1 Memory

Computers have something called **memory**. This is where data in your variables are stored. The amount of memory you use is dictated by your program. In programming contests, it is limited by the amount specified either in the contest or in the problem itself. Time and memory are the resources that you need to manage well in order for your solutions to pass.

One important thing to note is that all variables, regardless of type, are stored as bytes in the same memory pool available to your program. There is no separate memory for integers or floats or strings or arrays. The only difference is how regions of memory are interpreted. Eight bytes can be taken to mean a string of eight `char`s, a single `long long`, two `int`s, or one `double`. What matters is what your program thinks it is.

A.1.1 Addresses

Each variable in your program is assigned a location in memory[26], represented by an integer address. When a C++ program is compiled, those integer addresses are being used (explicitly or implicitly) to refer to the data in memory instead of "variables" (variables don't exist anymore in the actual running of the program).

For the most part, you don't have to deal with addresses directly as it's all managed by the C++ compiler for you. But there are times when having knowledge about them is useful (and sometimes necessary) in order to program your intended algorithm or data structure.

The address of a variable can be obtained by using the `&` operator. One naive way to store it is as a `long long`, but as we mentioned earlier, you (and the compiler) lose some information if you do it this way: does this number point to a `char` or a `long long` or maybe something even bigger?

In C++, a pointer type is what is used to tell the computer how we want to interpret these 64-bit addresses. To declare a pointer variable, we declare a variable with a `*`. For example, we can do:

```
1  void main() {
2      int x;
3      int* pointer_to_x = &x;
4  }
```

In case any of you become confused with examples on the internet, most other sources will put the `*` declaration beside the variable name as in `int *pointer_to_x`. It means exactly the same thing as `int* pointer_to_x`. I prefer the latter. Unfortunately, the designers of the original C prefer the second because `int *ptr, x;` makes a pointer to an `int` and then a normal `int`. If you wanted two pointers, you'd have to declare `int *ptr, *x`. I will not be doing this in any of our examples because I believe that the variable type is a "pointer to an `int`" and not

[26] With compiler optimizations, this isn't exactly correct.

an "`int` that happens to be a pointer". This is a stylistic preference, so do what you want. The C++ compiler doesn't care how you put the spaces.

Of course, you can have pointers to pointers, pointers to pointers to pointers, etc.

```
void main() {
    int x;
    int* pointer_to_x = &x;
    int** pointer_to_pointer_to_x = &pointer_to_x;
    int*** pointer_to_pointer_to_pointer_to_x = &pointer_to_pointer_to_x;
}
```

At some point it gets a bit silly. But we will find a use for this kind of thing later when we talk about arrays.

To access the memory at that address, we can use the * operator, known as the **dereference operator**. Don't get confused by the multiplication operator with the same symbol.

The type of the pointer tells the computer how many bytes to interpret starting from that address. For example, look at the following:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    unsigned int x = (3 << 24) + (5 << 16) + (7 << 8) + 11;

    char* ptr = (char*)&x;

    cout << (int)*ptr << " ";
    ptr++;
    cout << (int)*ptr << " ";
    ptr++;
    cout << (int)*ptr << " ";
    ptr++;
    cout << (int)*ptr << " ";
}
```

This code prints out "`11 7 5 3 `" because of something called "endianness". But the important point is that we can abuse different pointer types to access the same data in different ways. **All memory is the same.**

Now you may be wondering why we would even want to mess with the memory address of variables instead of the variables directly. The above example is clearly really contrived. So let's give a simple one for the sake of example. Suppose we wanted a function to return two variables. But C++ only allows one variable at a time. The more clever ones among you might consider packing multiple values in a single 64-bit number. The ones who know a bit more C++ might think of using structs or the STL. But there's something done in the standard C library that uses pointers, and I think it's pretty great. The function in question is called `scanf`. You use it like: `scanf("%d %d", &a, &b)`. Note that it asks for the address of the variables it should store the data in. An implementation of such a function may look like this:

```
#include <bits/stdc++.h>
using namespace std;

void foo(int* addr1, int* addr2) {
    *addr1 = 5;
```

```
6       *addr2 = 7;
7   }
8
9   int main() {
10      int a, b;
11      foo(&a, &b);
12      cout << a << " " << b;
13  }
```

A.2 Arrays and pointers

A.2.1 Arrays as blocks of memory

An **array** is a big continuous chunk of memory. To get the first item in an array of 1-byte objects, we just look at the first byte of that chunk of memory. The second item is the second byte. And in general the nth item is the nth byte in that chunk of memory.

If we have an array of 4-byte objects like `int`s, then the first item is the first four bytes of memory. The second item is bytes five to eight. And so on.

Another way to view it is if we have a pointer to the first byte of memory in that array, then we can find a simple formula that makes the pointer point to the nth byte of that array. If we have 1-byte objects, we only need to look at `ptr + (n-1)`.

A.2.2 Pointer arithmetic

Extending this idea, you would think that if we have 4-byte objects, to get the nth item in the array, we want to look at bytes `ptr + 4*(n-1)` until `ptr + 4*(n-1) + 3`. But it's not!

If you have a pointer and you add an integer to it, it automatically multiplies the added integer by the size of type of the pointer. So an `int` pointer, if you add 1 to it, will actually get incremented by 4. If you add 3 to it, will get incremented by 12. In some way, this makes pointers easier to work with if you think about them as arrays. Some people might find this strange, but that's just the way C++ was designed. This is called pointer arithmetic, by the way. Most of the time, you just want to increment and decrement pointers, and it generally works out really well. In case you really need to mess with memory addresses (which you almost never should), you can just typecast your pointers into `long long` before doing the operations, then typecast them back to your prefered pointer type.

To repeat that previous paragraph, if we have a variable `int* p` and we know `p` is 100, then `p+1` is 104, and `p+3` is 112, because of the quirks of pointer arithmetic. Explicitly turn them into `long long` if you want to do some black magic with your memory addresses.

A.2.3 Zero-based indexing

Earlier we mentioned that to get the nth item in an array, we look at `ptr + (n-1)`. But if we just started counting from 0, then we can save a subtraction by 1! The 0th item is at address `ptr`, the 1st item is at address `ptr + 1`, the 2nd item is at address `ptr + 2`. And the kth item is at address `ptr + k`. This trick is one reason why programmers like to count from 0.

As a side note, there are some programming languages that are not zero-indexed and their compilers automatically add -1 when arrays are accessed. That's not C++ though, so no need to worry about it.

A.2.4 The [] and * operators

By now you might be thinking: "Wait a minute! I've been doing array access without all this pointer stuff. It's just `arr[x]` right?" And I'm here to tell you that array access by doing `arr[x]` is completely the same as `*(arr+x)`!! The compiler turns them into the exact same processor instructions. So if you have a pointer that points to a single variable and not an array, instead of doing `*ptr` you can just as well do `ptr[0]`. It might not make intuitive sense when you're reading some code that's not yours though. Most people will expect `[]` for arrays and `*` for single variable pointers.

A.3 Dynamic memory allocation

A.3.1 Stack memory

So far we've been playing around with using pointers to point to individual bytes of an `int` or to access indices of an array. We also discussed how pointers can be used to implement a function like `scanf`. Let's look at another application of pointers: dynamic memory allocation. Normally, C++ code will only be able to access a few megabytes of memory depending on the operating system and the specs of the computer. This memory, called the **stack**, is used for all local and global variables. We can verify this by testing the following code:[27]

```
#include <bits/stdc++.h>
using namespace std;

int total_allocated = 0;
void go_deeper() {
    char arr[10000];
    total_allocated += 10000;
    printf("I have allocated about %d bytes of memory.\n", total_allocated);
    go_deeper();
}

int main() {
    go_deeper();
}
```

On my computer, this code goes deep enough to allocate about 8 megabytes before it crashes. But my computer has several gigabytes of memory. So what happened? And how do we access the rest of our several gigabytes of memory (or hundreds of megabytes in contests)?

A.3.2 The `malloc` function

Let's first discuss the question of how to access the rest of our memory. There's a function called `malloc` that asks the operating system for some number of bytes, and the function will return a pointer to the start of that chunk of memory. It's up to you to decide what you want that memory region to mean. The operating system only cares that you asked for N bytes so it'll give you N bytes. If you point an `int` pointer to it, your program will think of it as an array of integers. If you turn it into a `char` pointer, it'll think it's an array of characters. Let's have a look at some sample code:

[27] The `go_deeper` function is an example of a recursive function since it calls itself.

```cpp
#include <bits/stdc++.h>
using namespace std;

int total_allocated = 0;
void go_deeper() {
        char* arr = (char*)malloc(10000);
        // This malloc call gives you 10000 bytes to play with.
        // It doesn't have any idea of the meaning of this chunk of memory,
        // so it's your job to convert it to a pointer of the correct type.
        total_allocated += 10000;
        printf("I have allocated %d bytes of memory.\n", total_allocated);
        go_deeper();
}

int main() {
    go_deeper();
}
```

If you open Task Manager (or its equivalent in Linux) while this program is running, you'll notice that it's slowly eating up more and more of your memory. You might want to kill it before it goes too far and your computer starts lagging. Or you could also wait for your computer to crash or the program to crash because of memory issues. :)

A.3.3 The `free` function

Now that we know how to allocate memory, we have to know how to de-allocate it. This is done through the `free` function. Just call it and pass the memory address given by `malloc` and it'll return the memory back to the operating system.

A.3.4 Heap memory

Memory allocated and freed through the use of `malloc` and `free` is memory from the **heap**. In contrast to stack memory, heap memory is shared by all running programs. If we have time in the future, we can go into how stack and heap memory are used by the compiler, but it's not really necessary. The important thing to know is that **stack memory is limited to a few megabytes by default. Heap memory is practically all the memory available on your computer. Stack memory is used for local variables. Heap memory is memory gotten by `malloc` and `free`.** This means your pointers (the addresses) are in stack memory, but they're free to point to any memory region whether that memory region is in the stack or on the heap.

A.3.5 The `sizeof` operator

Many times, you're too lazy to manually compute the number of bytes you need. For example, you know that you want enough space for an array of `int` with 12345 elements. Doing the math yourself is such a pain. Of course you can type `malloc(4 * 12345);` since you know that `int` takes 4 bytes. But in case you forget or you want to make the code more readable, we'll introduce a new operator called `sizeof`. Using this operator actually looks like a function call, but the compiler evaluates it while your code is compiling to compute sizes. Here is some sample code.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int* arr = (int*)malloc(sizeof(int) * 10000);
}
```

`sizeof` is also really useful for when you define our own custom data types. One day you might want to make our data type take 16 bytes, but then you update it so it needs 24 bytes. In that case, you don't want to go through all our code to replace all the references of `malloc(16 * x);` with `malloc(24 * x);`. Instead, we can just do `malloc(sizeof(custom_data_type) * x);`.

A.3.6 The null pointer

The function `malloc` guarantees that it will never return an address below 4096 in modern operating systems. The value of 4096 changes based on computer specs and operating system, but it's guaranteed that it will never allocate you the memory address 0. So if you want to say that a pointer points to nowhere, set its value to 0. If you want to check if a pointer is uninitialized, check if it points to 0. This is a convention for everybody and is the universally accepted way of knowing whether a pointer is uninitialized or not.

A.4 Memory errors

Lastly, here are some common errors you might encounter when playing around with memory.

A.4.1 Segmentation fault

When you access a memory location that does not belong to you, you get something called a segmentation fault. Usually this happens in three scenarios. One, you allocate 100 bytes, but try to access the 101th byte. (If we're using zero-indexing, the 101th byte is the byte at index "100".) Two, you access memory that you have not even allocated yet. Three, you access memory that you have already freed. Sometimes, it's possible that your code continues on. But this kind of scenario is very bad.

A.4.2 Memory leak

When you allocate memory, you are given a pointer to the address of the memory you requested. If you update your pointer to something else, say you request a new chunk of memory, and you forget to free the first block of memory, you will never again be able to recover that memory address. Since you can't access that memory address, that memory is lost forever until your program stops. This situation is called a memory leak. Some software and video games are notorious for this and they will gradually eat up more memory until you kill them. For programming contests, if you allocate, say, 100MB per test case but fail to free your memory, you will likely hit a memory limit exceeded error after a few cases.

A.5 Strings, character arrays, and encoding

A.6 C strings

Before this new `string` data type from C++, C only had character arrays. In C, a string is only a `char` array with the last `char` equal to 0.

To get the length of a string, you would loop through all characters until you hit 0 (the null character). To add a character to the string, you just write one more character at the end, making sure that the new end is 0 again. It's all very simple. And if you have a C++ string, you can get a pointer to the original C string using `s.c_str()`.

The reason we bring this up is because we want to talk about how characters are represented in computers.

Computers only work with numbers. They know how to copy, add, subtract, and so on. But characters are not numbers. What we really need to do is to decide a mapping from each number to a character. Thankfully, other people have decided that mapping for us. The most important mapping of numbers to characters to learn is called ASCII. There are tables online for ASCII (and in Linux/Mac, you can use `man ascii`), but memorizing the table doesn't have very much use. It's enough to know that the mapping exists and that's computers will look up characters from the table to decide what to display on your screen.

By looking up some values on the table, we can learn that 32 maps to the space character. 10 maps to the new line character (the "enter" character). 65 maps to `A`. 97 maps to `a`. In general, ASCII maps numbers from the range 0 to 127 into characters, and this is enough to fit in one byte. The numbers from 128 to 255 vary wildly, so we won't go into that. The following code demonstrates my point about the equivalence of characters and numbers:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    if ('A' == 65) {
        printf("'A' is equal to 65\n");
    }
    if ('z' == 122) {
        printf("'z' is equal to 122\n");
    }
    printf("Let's print the %c character\n", 65);
}
```

You might notice from this that the `'x'` notation for characters is just convenience for letting the compiler look up the number for you during the compilation stage. There's not very much reason to look up the character yourself when the compiler can do it. The important thing is that the equivalence of letters and numbers allows you to do operations on characters. The ASCII table was designed for a purpose, so it's not just completely jumbled garbage. Digits are consecutive on that table, and so are capital letters and small letters. For example, if you wanted to capitalize letters, you just need to do some arithmetic to convert the range 97-122 ('`a`' to '`z`') to 65-90 ('`A`' to '`Z`').

This is practically all you'll need to know about character encoding for programming contests.

A.6.1 Sample string upper case

To demonstrate what I mean about using ASCII with arithmetic, here is some example code that capitalizes one line from the input.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    char s[1000];
    cin.getline(s, 1000);
    for (int i = 0; i < 1000; i++) {
        // Terminating character
        if (s[i] == 0) {
            break;
        }

        // if within the range of 'a' to 'z'
        if ('a' <= s[i] && s[i] <= 'z') {
            // Subtract 'a' to make it 0 to 25, then add 'A' to capitalize.
            s[i] = s[i] - 'a' + 'A';
        }

        // equivalent to the code above (but don't use this)
        // if (97 <= s[i] && s[i] <= 122) {
        //   s[i] = s[i] - 97 + 65;
        // }
    }
    cout << s << '\n';
}
```

You can also use the `toupper` function for this.

A.7 C++ `structs`

We have been mentioning defining our own data types for a while now. It's time we learn to do that now.

Let's start with a simple problem. We want to write a function that computes the simplest form of a fraction.

If you actually try to think of how to do this, you'll quickly run into a problem: Fractions are represented by a numerator and a denominator. But functions can only return one value. If you're willing to mess around, I'm sure some of you will be able to think of solutions to this question.

Using what you already know by now, I can imagine three tricks to get around this problem. The first way is to store return values in global variables and copy them over as soon as the function is done. A second way is to accept pointers and store the answers in the given addresses. And third, allocate some new memory, store the answer there, then return a pointer to that memory address. Any of these solutions will work, but they don't fit into the contrived story for this section! Not to mention it's going to be difficult to work with these things.

Wouldn't it be nice if we could just bundle variables together in a simple and elegant way? Then we could return those bundles so we'll be able to return two variables. And taking that idea further, we can pass those bundles as arguments to functions, saving us lots of typing and error-prone code. What a wonderful world that would be if only we could bundle variables

together.

Well I have good news for you! C++ supports exactly that through a feature called a **struct**! Another name you might read on the internet for a struct is a "record", which is the term used for the same thing in a few older programming languages (and even some newer ones like the Starcraft II modding language called Galaxy). To understand what a struct is, let's look at some sample code.

```
struct fraction {
    int numerator;
    int denominator;
};
```

The sample code above defines a struct called `fraction`. In this case, fraction is a bundle of two `int` variables. The variables that it groups together are called its members. In this case, our fraction struct has two members called numerator and denominator. Let's look at some sample code to see how we can use our new data type.

```
#include <bits/stdc++.h>
using namespace std;

struct fraction {
    int numerator;
    int denominator;
};

int gcd(int a, int b) {
    // The GCD is the largest number both numbers are divisible by.
    int smaller = std::min(a, b);
    for (int i = smaller; i > 0; i--) {
        if (a % i == 0 && b % i == 0) {
            return i;
        }
    }
    return 1;
}

fraction simplify_fraction(fraction f) {
    // We simplify a fraction by dividing its numerator and denominator by their gcd.
    int g = gcd(f.numerator, f.denominator);
    fraction answer;
    answer.numerator = f.numerator / g;
    answer.denominator = f.denominator / g;
    return answer;
}

int main() {
    // create a new fraction called `unsimp`
    fraction unsimp;
    unsimp.numerator = 15;
    unsimp.denominator = 20;

    // create a new fraction called `simp`
    fraction simp = simplify_fraction(unsimp);

    printf("The unsimplified version is %d / %d\n", unsimp.numerator, unsimp.denominator);
    printf("The simplified version is %d / %d\n", simp.numerator, simp.denominator);
}
```

Looking at the sample code, you'll see that members are accessed via the `.` operator. You will also notice that we passed fraction objects around as if they were any other data type. We will eventually learn how to do operations on our custom operations.

In the sample code above, `f`, `unsimp`, `answer`, and `simp` are called **objects** of our new data type `fraction`. So a struct is a definition of a data type, and an object is an occurrence of that data type. Sometimes you might also encounter the word **instance** which means the same as object.

One common error: You need a semi-colon `;` after the closing bracket `}` of the struct. Also you need a semi-colon ; at the end of each member declaration. So when you're defining structs and you suddenly get a syntax error, that's one of the most likely places to look.

One more thing I would like to note at this point is that for those of you who may have heard the term "data structure" before, I would like to distinguish a struct from a data structure. Some online references might confuse the two because of a similar-sounding name, but they are different. Note that **`struct`** is only a keyword in C++ that allows you to group variables together. We will be covering actual data structures later on, so hold on to your hats!

A.7.1 The arrow operator

C++ has a special operator for pointers to structs. If you have a pointer to a struct, you can use `->` to access its members instead of having to dereference the pointer and use the `.` operator. In other words, `(*ptr).member` means the same thing as `ptr->member`. The arrow notation was introduced simply for convenience, and it's recommended to use because it looks nicer.

A.8 C vs C++

A lot of the things discussed in this section are a mix of C-style and C++-style techniques of implementation. Here are a few notes on how C++ manages to build on top of C:

- The **`new`** and **`delete`** statements in C++ are implemented behind the scenes via C's lower-level `malloc` and `free`.[28] In other words, the "infinite void of unlimited memory" we talked about earlier is just the heap!
- **`new`** and **`delete`** are higher-level than `malloc` and `free`; that's why it has more features and safer in general.
- It is worth mentioning that you cannot mix **`new`**/`free` and `malloc`/**`delete`**. I recommend only using **`new`** (and **`delete`**).
- Mucking about with the equivalence between arrays and pointers to contiguous chunks of memory is mostly low-level C. In C++, the recommended ways are via the higher-level stuff like `vector` and `array`.
- The keyword **`struct`** is from C; the "proper" C++ keyword is **`class`**, which includes more features related to so-called **object oriented programming (OOP)**. For the most part, you don't need OOP in competitive programming, which is why we taught **`struct`**s instead of **`class`**es. (If we did it via **`class`**es, then you'd have to worry about something called "visibility"; but for small programs written for contests, it's better to just make everything **`public`** anyway, so it's mostly a nuisance.)

[28] at least, this is true for many implementations of C++. There's no requirement in the C++ standard for **`new`** and **`delete`** to be implemented via `malloc` and `free`.

- The `printf`, `scanf`, and `fgets` functions are from C; the recommended C++ ones are `cin`, `cout` and `getline`.
- In general, if a library starts with `c` (e.g., `cstdio`, `cmath`), then it's from C, otherwise (e.g., `iostream`, `algorithm`), it's from C++.
- C++ introduces a sort of *range-based* philosophy, which is the reason why most STL functions require you to pass the start and end of a range of values, e.g., `sort(a.begin(), a.end())` or `lower_bound(begin(a), end(a), v)`.

Anyway, one of the reasons C++ was built on top of C, and designed the way it is, has a lot to do with speed/efficiency. It's the reason why it's relatively trickier to code in C++ than in other high-level languages like Java or Python.[29]

All in all, the C++ stuff are higher-level than those of C, so we recommend *always using the C++ stuff over the C stuff* (unless you have a good reason otherwise).

[29] There are other, more modern languages that are nice to code in but are nonetheless competitive with C++ in terms of speed/efficiency, but since they haven't caught with the competitive programming community yet, we won't be covering them here.