IOI Training 2017 - Week 0

Payton Yao

February 27, 2017

1 Introduction

Throughout this first week, we'll be delving into what happens when you run programs. We'll have a look at what happens when you compile your program, what it looks like when you run them, and reasons why that's important. We'll have some experiments and exercises to understand what is really happening.

2 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++. For those unaware, IDEs combine multiple tools in one, allowing you to edit source code, compile programs, debug them, and more, all in one convenient program. 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. 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 to too much magic and sometimes we miss all the small things that happen behind the scenes.

Note that this is only a short introduction into the command line. Mastering the use of the command line 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. For Mac users, you can follow along on the Linux path since the differences are small. Please read the appropriate section based on your operating system.

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/`.

2.1 Windows

On Windows, you have two main choices of terminals: Command Prompt (`cmd.exe`) and PowerShell (`PowerShell.exe`). PowerShell is the newer (and more powerful) one. Command Prompt has a width limit of 80 characters, so it's kind of annoying. Run them from the Start Menu or the Windows 8 search.

The important commands are as follows:

- `cd` - change directory, use as `cd .`

- `dir` - list files in directory, use as `dir C:\Users\Public`
- `copy` - copy a file, use as `copy a.txt b.txt`
- `move` - move a file from one place to another, use as move original.txt new.txt. Interestingly, this is also the command used to rename files.
- `del` - delete a file, use as `del a.txt`
- `mkdir` - make a new directory, use as`mkdir new_directory`

We encourage you to try these commands out. Your computer won't break if you type a command wrong. You can hit the tab key to auto-complete.

Alternatively, you can try out the new Linux subsystem. You can find out how to install it here: https://msdn.microsoft.com/en-us/commandline/wsl/install_guide

2.2 Linux and Mac

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. For Linux users, you'll most likely be using the Gnome Terminal if you're on Ubuntu. (And if you're not on Ubuntu, you probably already know what you're doing anyway.)

The important commands are as follows:

cd - change cirectory, use as `cd /Downloads`. ls - list files in current directory, use as `ls` cp - copy a file, use as `cp a.txt b.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` Again, we encourage you to try these commands out. Your computer won't break if you type a command wrong. You can hit the tab key to auto-complete.

You may have noticed that we did `cd /Downloads`. `/` is the Linux shorthand for `/home/user/` and is the Mac shorthand for `/Users/user/`. 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/.

3 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! It's different on every operating system, so we'll just link you to instructions written by others.

3.1 Installing

3.1.1 Windows

Please go to https://wiki.wxwidgets.org/Installing_MinGW_under_Windows. There are more detailed instructions on the link. Make sure you don't forget to add `C:\MinGW\bin` to your Path environment variable.

3.1.2 Mac

For Mac users, you have to install Xcode then download the command line tools from there.

3.1.3 Linux

For Linux users, g++ comes installed on the base operating system.

3.2 Using g++

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`.

On Windows, the default name of the executable is `a.exe`. 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 the command `g++ example.cpp -o executable`. Note that for Windows users, the resulting executable name should end with `.exe` otherwise Windows won't run it.

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.

3.3 Running Executable Files

Now that you've compiled your program, it's time to run it! For Linux and OSX users, 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`. For Windows users, you don't even need the `./`. So if your program is saved as `a.exe`, just type `a.exe` directly and it should run.

3.4 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 task manager, but there's a much easier way. Just press Control + C. This will work on any operating system and 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 Control + C.

4 Input redirection

4.1 Saving Output to a File

There are cases where 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 notepad, 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`.

On windows, the corresponding command is `executable.exe > 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!

4.2 Reading Input from a File

Another common case is when we have input from a file that we want our program to read in from instead of us having to type the values. 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`.

On windows, the command is `executable.exe < input.txt`.

This is extremely useful for debugging your solutions for contests. Especially for problems with a lot of input like graphs.

4.3 Combining the Two

There are cases where we have input from a file and we want to save the output to another file. During the 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 to a pre-determined input. The way you do that is just combining the two.

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.

I'm sure the Windows users among you can get the pattern by now, so I won't belabor the point.

4.4 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. For example: `./program1 | ./program2`

4.5 Extras: diff

There's a very useful tool available on Linux and Mac called `diff`. What `diff` does is it compares two files line by line and tells you which lines differ. The usage is `diff file1.txt file2.txt`.

It's useful to consider using if you have, for example, a slow solution and a faster but more complex one. You can use the slow solution for a lot of small cases then use diff to ensure that the slow solution and the faster one give the same results.

5 Random Access Memory

Modern computers have something called Random Access Memory (RAM). Most modern computers have anywhere between one gigabyte to thirty two gigabytes of this, and it's the topic of our discussion today. At its core, random access memory 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 a command saying "Store these 8 bits in slot X" and "Retrieve the 8 bits in slot X". The number of slots correspond to how many gigabytes the RAM is.

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.

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 **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.

6 Pointers

Pointers allow you to access the slot numbers themselves. The official term used is the **address**. Every variable has an address which is a 64-bit number. Some older computers from the early 2000s might still

have addresses as 32-bit numbers because RAM rarely exceeded four gigabytes back then, but for the most part addresses nowadays are 64 bits long.

The address of a variable can be gotten by using the `&` operator. One naive way to store it is as a `long long`, but as we mentioned earlier, does this number refer to a `char` or a `long long` or maybe something even bigger?

In C++, the 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:

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

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`. Both of these mean the same thing, but I find more meaning in the way I prefer to do it. 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 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 do this even more.

```
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 dereference operator is unary, similar to how the unary negative operator is different from the minus operator.

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

```
#include <iostream>
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 6 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. 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. The implementation of such a function would look like this:

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

void foo(int* addr1, int* addr2) {
  *addr1 = 5;
  *addr2 = 7;
}

int main() {
  int a, b;
  foo(&a, &b);
  cout << a << " " << b;
}
```

7 Arrays and Pointers

7.1 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).

7.2 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 == 100`, `p+1 == 1004`, `p+3 == 12` 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.

7.3 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! 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 the 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.

7.4 The [] Operator and the * Operator

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.

8 Dynamic Memory Allocation

8.1 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:

```
#include <stdio.h>

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)?

8.2 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:

```
#include <stdio.h>
#include <malloc.h>

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 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. :)

8.3 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.

8.4 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. Heap memory is practically all the memory available on your computer. Stack memory is used for local variables. Heap memory is only from 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.

8.5 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.

```
#include <malloc.h>

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);`.

8.6 The Null Pointer

malloc makes a guarantee that it will never return an address below 4096 for 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.

8.7 Memory Errors

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

8.7.1 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. 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.

8.7.2 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.

9 Strings, Character Arrays, and Encoding

9.1 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. 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, 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 20 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 <stdio.h>

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.

9.2 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 <iostream>
using namespace std;

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

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

    // equivalent to the code above
    // if (97 <= string[i] && string[i] <= 122) {
    //   string[i] = string[i] - 97 + 65;
    // }
  }
  cout << string << endl;
}
```

10 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 <stdio.h>
#include <algorithm>

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. 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!

10.1 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.

11 Homework

11.1 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.

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

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

linked_list* linked_list_create() {
   // Fill out
}
```

```
void linked_list_destroy(linked_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

   linked_list* list = linked_list_create();

   while(1) {
       string instr;
       cin >> instr;
       if (instr == "add") {
          int v;
          cin >> v;
          linked_list_add(list, value);
       } else if (instr == "remove) {
          int idx;
          cin >> idx;
          linked_list_remove(list, idx);
       } else if (instr == "get") {
          int idx;
          cin >> idx;
          cout << linked_list_get(list, idx) << endl;
       } else if (instr == "length") {
          cout << linked_list->length;
       } else if (instr == "exit") {
          break;
       }
   }
   linked_list_destroy(list);
}
```

11.2 Jagged Array

Subtask 1. Allocate a 2D array from the heap.

Subtask 2. Allocate a jagged array (i.e. a 2D array with each subarray not having equal size) from the heap. Row i should have space for (i+1) elements.

11.3 Problems

UVa 11278 - One-Handed Typist
UVa 554 - Caesar Cypher

IOI Training 2017 - Week 1
The C++ Standard Template Library

Vernon Gutierrez

March 6, 2017

1 Introduction

This week, 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 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 **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 (AKA you), so that these users (AKA 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 re-invent 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 done `#include <iostream>` and `using namespace std;` and `cin` and `cout`, 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.

"**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++`, 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 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.

"**Template**" here means the particular part of the C++ standard library that includes data structures and algorithms, which are most useful for programming contests. For all C++ programs actually, since data structures and algorithms are the bread and butter (or *kanin* and *toyo*) of programs. The nice thing about the C++ STL is that all of its features can be easily used through a common pattern called an *iterator*, which you will learn more about below. For a competitive programmer (AKA 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. These patterns are not the reason why STL is called "template." The real reason is a bit too technical for our purposes, and has something to do with the template feature of the C++ language. You can check this video (and the entire playlist) out if you want to delve deeper into templates (and the STL).

2 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.

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.

3 Pairs

3.1 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.

3.2 How to Use C++ Built-in Pairs

Using `pair` is easy. Just `#include <utility>` to make it available. 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.

Sample code:

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

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

    lattice_point.first = 2;
    cout << lattice_point.first << " " << lattice_point.second << endl; // 2 -1

    pair<string, int> name_age = make_pair("Aldrich", 20);
    cout << name_age.first << ", " << name_age.second << endl; // 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 << endl; // 1 2
    cout << nested_pair.second << endl; // 3
    return 0;
}
```

3.3 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 the most convenient choice. The benefits of using pairs will become clearer later with graph algorithms. Thinking about pairs will also help you understand the `map` data structure below.

3.4 Gotchas

Modify the code above to directly print `lattice_point`. That is, try `cout << lattice_point << endl;`. What happens? `std::pairs`'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 the currently most widely-used version of C++, 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 a `>>` (e.g. used for `cin`) operator. 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 C++14) by default. If you did, then you should either add the space, or explicitly tell your compiler to use C++11 instead:

```
g++ -std=c++11 program.cpp
```

Or, if you have an even newer compiler, use C++14: instead:

```
g++ -std=c++14 program.cpp
```

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++11. So, you might want to make it a habit to always compile with `-std=c++11`. C++14 is not yet as widely supported, but in a few years, you should probably change that habit to use C++14 instead.

4 Vectors

4.1 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, which you saw in Week

0. But a linked list has a serious disadvantage: retrieving elements from the middle of the list requires $O(n)$ time. In technical terms, a linked list does not have efficient *random access*.

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 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 $O(n)$ work per operation to copy items.

C++ has a built-in dynamic array, 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.

4.2 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 sequence containers. But in competitive programming settings, only `vector` and `deque` are widely used.

4.3 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 that it can be queried in $O(1)$.

4.4 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.

5 Stacks, Queues, and Doubly-Ended Queues

5.1 Theory

Stacks, queues, and doubly-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 doubly-ended queues work.

5.2 How to Use C++ Built-in Stacks, Queues, and Doubly-ended Queues

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

Doubly-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.

5.3 How They're Implemented in C++

Stacks, queues, and doubly-ended queues in C++ are generally implemented using dynamic arrays rather than linked lists.

5.4 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.

6 Priority Queues

6.1 Theory

Priority queues are a generalization of queues, where in each dequeue operation, instead of accessing or removing the 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 $O(n)$ time. But if we needed to do lots of operations, this is too slow. A heap data structure allows us to perform each operation in $O(\lg n)$ time. Watch this or this to learn how heaps achieve this.

6.2 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. Note that you must `#include <queue>`, and not `<priority_queue>`.

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

6.3 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.

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, competitive programmer from University of Indonesia, has a nice blog post about it.

6.4 How They're Implemented in C++

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

6.5 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 Sets and Maps

7.1 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 $O(n)$ space rather than $O(N)$ space. If $n << N$, 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 $O(\lg n)$ time per operation, sets and maps are typically implemented using binary search trees. Check this video out to learn about them.

7.2 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`

7.3 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 $O(\lg n)$ time. There are two variants for this: `upper_bound` and `lower_bound`. Here is the reference for set. Map works similarly.

7.4 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.

8 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.

9 Sorting

9.1 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 $O(n \lg n)$ time, but you're not sure why. If you're curious and want to know why, watch this video.

9.2 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.

10 Permutations

10.1 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.

10.2 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.

11 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.

12 Miscellaneous Tips

Be very careful in naming your variables when you are `using namespace std` and the C++ STL. If you include the above 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, if you `#include <algorithm>` and you are `using namespace std`, you never 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.

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.

13 Challenges (To Be Submitted)

Try out the problems below. Use the C++ STL to make your code as short as possible.

Codeforces 493B - Vasya and Wrestling

UVa 11995 - I Can Guess the Data Structure!

Codeforces 44A - Indian Summer

Codeforces 390A - Inna and Alarm Clock

UVa 11849 - CD

Codeforces 501B - Misha and Changing Handles

UVa 10226 - Hardwood Species

Codeforces 474B - Worms

Codeforces 405A - Gravity Flip

UVa 146 - ID Codes

Codeforces 431B - Shower Line

14 Extra Challenges (Optional)

UVa 732 - Anagrams by Stack

UVa 10901 - Ferry Loading III

UVa 11034 - Ferry Loading IV

UVa 1203 - Argus

UVa 11286 - Conformity

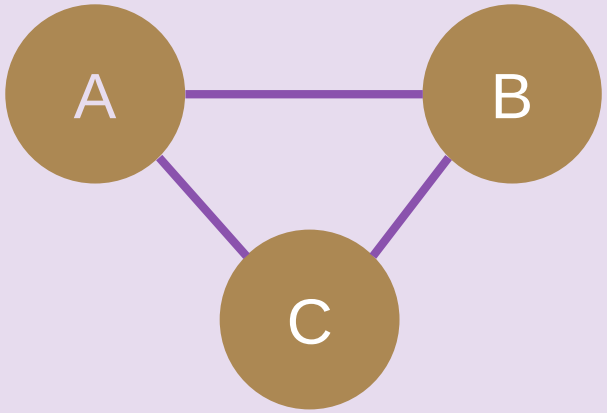
UVa 11136 - Hoax or what

GRAPH THEORY

Hadrian Ang, Kyle See, March 2017

What are graphs?

A graph G is a pair $G = (V, E)$ where V is a non-empty set of vertices and E is a set of edges e such that $e = \{a, b\}$ where a and b are vertices.



Why graphs?

Graphs are usually used to represent different elements that are somehow related to each other.

What are vertices?

Vertices, sometimes called **nodes**, are objects that form graphs. They are usually used to represent certain elements to be related with another.

Examples

Cities in a country
People in a social network

What are edges?

Edges are two element subsets of V (at least in the undirected case, but more on this later). They usually represent connections in a system.

Examples

Roads between cities
Friendships between people

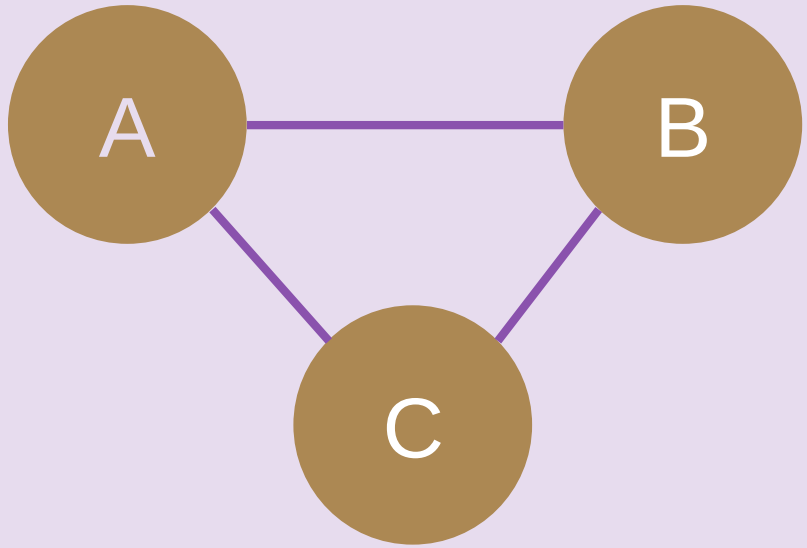
Just some terms…

Two vertices are said to be **adjacent** if they are joined by an edge. In Fig. A, the vertices A and B are adjacent.

An edge is said to be **incident** to the vertices it joins. In Fig. A, the edge $\{A, B\}$ is incident to vertex A and B**.**

The number of edges incident to a vertex is called the **degree** of that vertex. It is sometimes denoted as $\deg(v)$ for some vertex v. All vertices in Fig. A have a degree of 2.

Figure A



Graph Directedness

Graphs can be **directed**, sometimes called **digraphs**, or undirected.

A

In an **undirected** graph, edges go both ways. An edge from A to B is also an edge from B to A. Undirected edges are usually drawn as straight lines between vertices. Edges are subsets of the set V.

Example: $\{A, B\}$

B

In a **directed** graph, edges do not go both ways. In Figure B, there is an edge from A to B, but no edge from B to A. Edges are usually drawn with arrows to show directedness. Instead of being subsets, edges are ordered pairs with both elements from the set V.

Example: (A, B)

Figure A

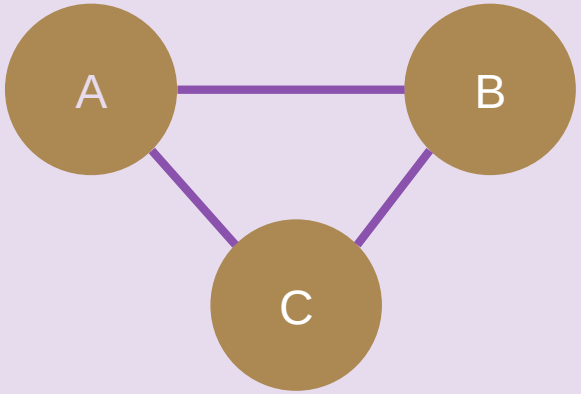
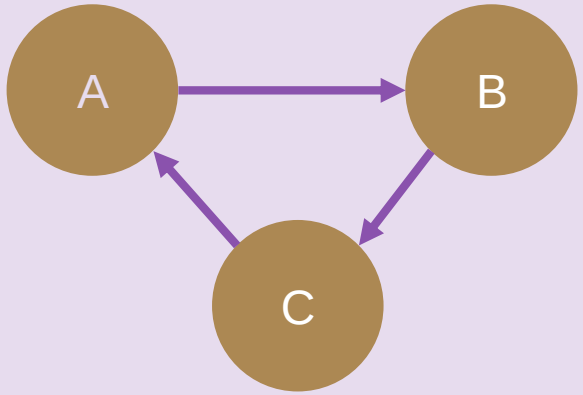


Figure B



Graph Weightedness

Graphs can be **weighted**, or **unweighted**.

A

In an unweighted graph, edges do not have any specific numeric value relative to other edges in the graph.

Example: Following on Instagram, a specific follower is not intrinsically more valuable than another one.

B

In a weighted graph, edges each have some associated value called the **weight** of the edge. Weights are usually drawn near the edge they are associated with.

Example: Roads (some roads are longer than others, thus have more weight).

Figure A

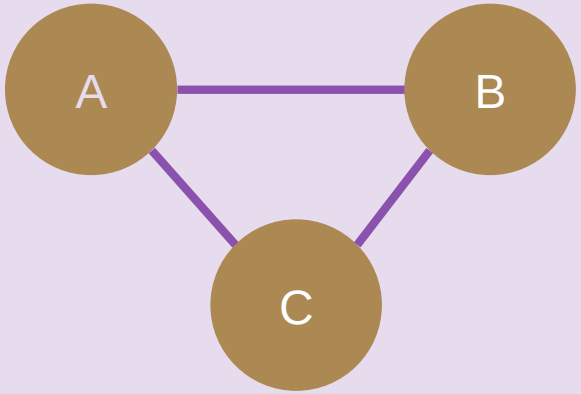
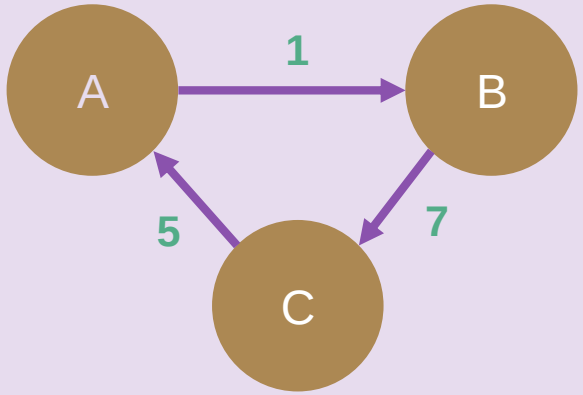


Figure B



Complex Graphs

A graph is called a complex graph when it has loops or multi-edges.

A

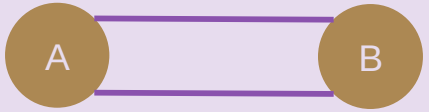
Loop

A **loop** is an edge that has the same source and destination vertex. In other words, it is an edge from a vertex to itself.

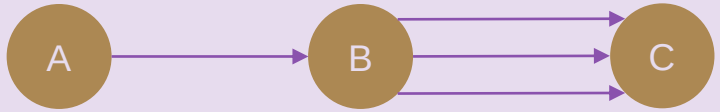
B

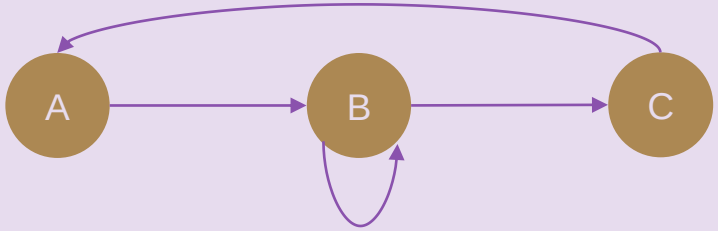
Multi-edge

Multi-edges or **multiple edges** are two or more edges between the same two vertices. Note that in a directed graph, edges from A to B and B to A are not multi-edges (they have to be going in the same direction to be considered so).



Below are some examples of complex graphs





A graph without multi-edges or loops is called a simple graph.

Walks and Paths

A **walk** is a sequence of vertices and edges

$$v_0, v_1, \dots, v_k$$
$$\{v_0, v_1\}, \{v_1, v_2\}, \dots, \{v_{k-1}, v_k\}$$

Such that $\{v_i, v_{i+1}\}$ is an edge in G for all i, $0 \leq i < k$. The walk starts at v_0 and ends at v_k. The **length** of the walk is k, its source is v_0 and its destination is v_k.

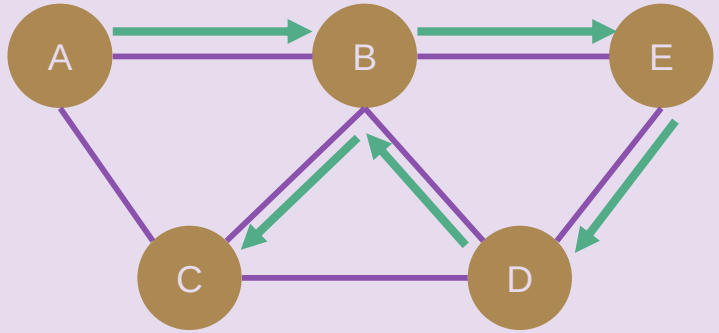
A **path**, similar to a walk, is also a sequence of vertices and edges.

$$v_0, v_1, \dots, v_k, v_i \neq v_j \ \forall i, j$$
$$\{v_0, v_1\}, \{v_1, v_2\}, \dots, \{v_{k-1}, v_k\}$$

A path cannot cross the same vertex twice, however.

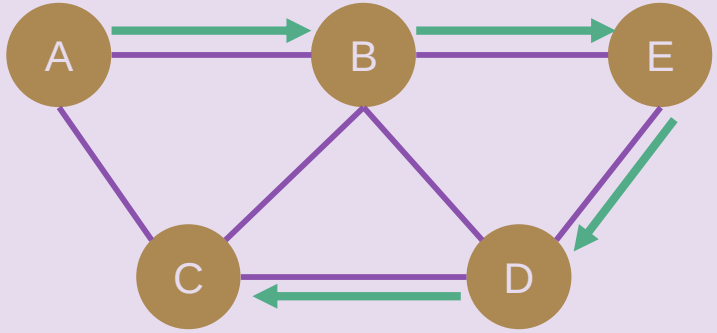
All paths are walks, but not all walks are paths.

Sample Walk



$$A, B, E, D, B, C$$
$$\{A, B\}, \{B, E\}, \{E, D\}, \{D, B\}, \{B, C\}$$

Sample Path



$$A, B, E, D, B, C$$
$$\{A, B\}, \{B, E\}, \{E, D\}, \{D, C\}$$

Closed Walks and Cycles

A **Closed Walk**, just like a walk, is a sequence of vertices and edges, but it starts and ends with the same vertex. In other words, the source of the walk is the same as the destination.

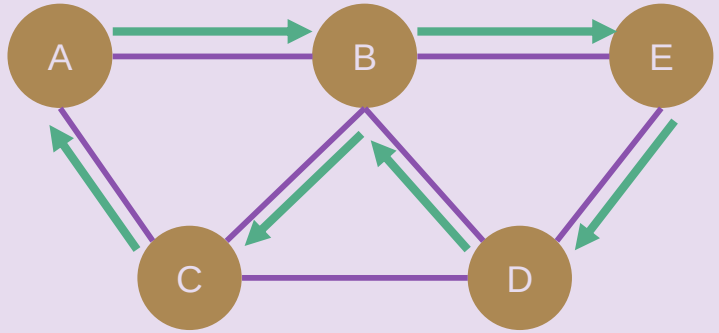
$$v_0, v_1, \ldots, v_k$$
$$\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$$

Such that $\{v_i, v_{i+1}\}$ is an edge in G for all i, $0 \leq i < k$ and $v_0 = v_k$.

A **Cycle** is to a closed walk as a path is to a walk. It is also a sequence of vertices and edges starting and ending on the same vertex, but all vertices are unique except for the start / end vertex.

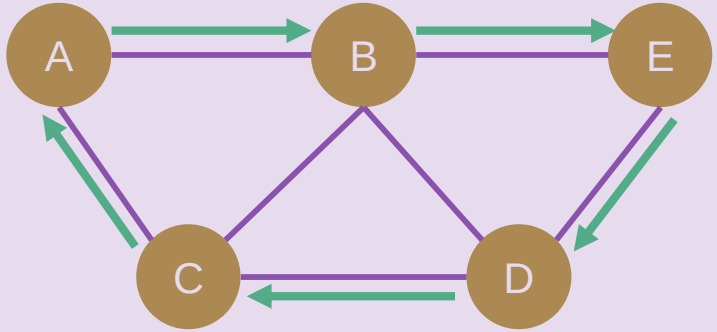
All cycles are closed walks, but not all closed walks are cycles.

Sample Closed Walk



$$A, B, E, D, B, C, A$$
$$\{A, B\}, \{B, E\}, \{E, D\}, \{D, B\}, \{B, C\}, \{C, A\}$$

Sample Cycle



$$A, B, E, D, B, C, A$$
$$\{A, B\}, \{B, E\}, \{E, D\}, \{D, C\}, \{C, A\}$$

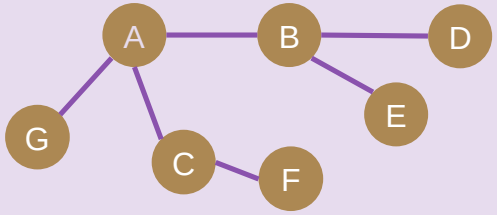
Special Graphs

Many special types of graphs have names to more uniquely define them. The following are a few of these graphs.

A

Tree

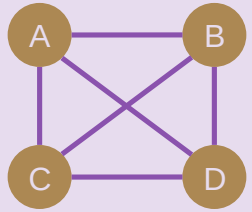
An undirected graph where there is exactly one path from any vertex to any other vertex



B

Complete Graph

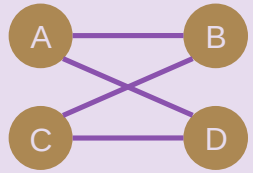
A graph where each vertex has an edge to every other vertex in the graph.



C

Bipartite Graph

A graph where the vertices can be divided into two sets such that no vertex in one set has an edge to another vertex in the same set. Called complete bipartite if all vertices in one set are connected to all vertices in the other set.



D

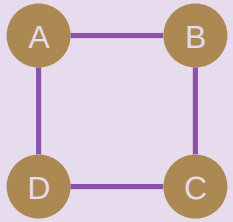
Path/Linked List

A tree that simply consists of one connected line of vertices.

E

Cycle

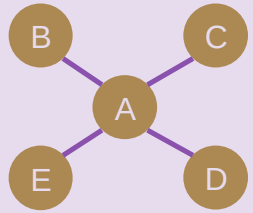
A graph that consists of one entire loop or cycle.



F

Star

A tree with one “center” vertex connected to all the other vertices, but all other vertices only connected to the center vertex.



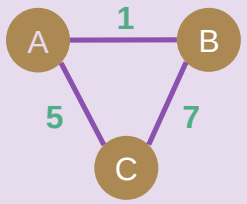
Computer Representation

There are three main ways of representing graphs using a programming language. Carefully selecting which representation to use is important when solving a problem.

A

Adjacency Matrix

Stores the graph in a matrix usually represented as a 2-D array such that `matrix[i][j]` contains the weight of the edge (i, j).



	A	B	C
A	0	1	5
B	1	0	7
C	5	7	0

B

Adjacency List

Each vertex is associated with a list (can be represented in C++ as a vector) populated by vertices each is adjacent to. One possible implementation of this is to use objects.

```
class Vertex{
  public:
    int index;
    vector<Vertex*> adj;
    Vertex(int ind){
      index = ind;
    }
};

int main(){
  Vertex* a = new Vertex(1);
  Vertex* b = new Vertex(2);
  //If there is an undirected
  //edge between a and b
  a->adj.push_back(b);
}
```

C

Edge List

Perhaps least common among the three approaches, a list is created to store all the edges in the graph. Edges can be stored as objects.

```
class Edge{
  public:
    int source, dest;
    Edge(int s, int d){
      source = s;
      dest = d;
    }
};

int main(){
  vector<Edge*> edges;
  //Assume 1 and 2 are indices
  //of adjacent vertices stored
  //in an array

  Edge* edge = new Edge(1,2);
  edges.push_back(edge);
}
```

Adjacency Matrix

Pros

Can easily retrieve the weight of the edge between two connected vertices or check if two vertices are connected (just check `matrix[A][B]`).

Cons

Looping through all neighbors of one vertex is expensive because you have to go through all of the empty cells.
Takes a lot of memory, a lot of it wasted on empty cells (especially for graphs with many vertices, but few edges).
Because of this, adjacency matrices cannot be used for problems with relatively large limits.
Each cell can only contain one value, so adjacency matrices do not support complex graphs (unless you use a 2-dimensional matrix of vectors).

Notes

A ***sentinel*** value must be used to “fill in” the empty spaces left by edges that do not exist in the graph. Typical values used include 0, -1, and INT_MAX. The choice of sentinel values depends on the problem and how you implement your solution. For example, some problems may require you to have negative weight edges, so it may be wiser to use INT_MAX.

Adjacency Matrix

Sample Implementation

```
int adj[N][N];

int main(){
    //start of test case
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            adj[i][j] = sentinel;
        }
    }
    //if a and b are connected
    adj[a][b] = weight;
    adj[b][a] = weight; //if undirected

    //check all neighbors of a
    for(int i=0; i<n; i++){
        if(adj[a][i] == sentinel) continue;
        //use adj[a][i]
    }
}
```

Adjacency List

Pros

Using objects can help in attaching specific information to each vertex.
Less space used than adjacency matrix.
It is easy to loop through all the neighbors of a vertex.

Cons

To check if two vertices are connected, one will have to loop through the whole adjacency list of one of the vertices.

Notes

Don’t forget to clear each vector when beginning a new test case if you declare them globally.

Adjacency List

Sample no objects implementation

```
vector<int> adj[N];

int main(){
    //start of test case

    for(int i=0; i<n; i++){
        //clear adj[i]
    }

    //if a and b are connected
    adj[a].push_back(b);
    adj[b].push_back(a); //if undirected

    //check all neighbors of a
    for(int i=0; i<adj[a].size(); i++){
        //a is adjacent to adj[a][i]
    }
}
```

```
/*if weighted, use pair<int, int> or create a
second vector<int> array and use the same indices
to correspond to the same edge, for example:

vector<int> adj[N], adjw[N];
adj[a].push_back(b);
adjw[a].push_back(weight);

use adj[a][i] and adjw[a][i] */
```

Edge List

Pros

Can easily iterate over all the edges in the graph (required for some algorithms).
Least space used since edges are not repeated for the two vertices they are incident to (unlike in adjacency lists)

Cons

Makes it difficult to get only the edges incident to a specific vertex (there are ways to get around this, but it requires more space and effort, more on this in the sample implementation).
Like adjacency lists, it is expensive to determine if two vertices are connected.

Notes

Don’t forget to clear the list of edges at the beginning of each test case.
Creating helper functions may make using edge lists easier for many problems.

Edge List

Sample no objects implementation

```
int n, e, last[N], prev[E], head[E];

void init(int n){
    e = 0;
    //set last[i] to -1 for I from 0 to n
}

void addEdge(int u, int v){
    head[e] = v; prev[e] = last[u];
    last[u] = e++;
}
```

```
int main(){
    //start of test case
    init(nodes);

    //if a has an edge to b
    addEdge(a, b);
    addEdge(b, a); //if undirected

    //check all neighbors of a
    for(int e=last[a]; e >= 0; e = prev[e]){
        //a is adjacent to head[e]
    }
}

//for weighted graphs, add an extra array similar
to adjacency list
```

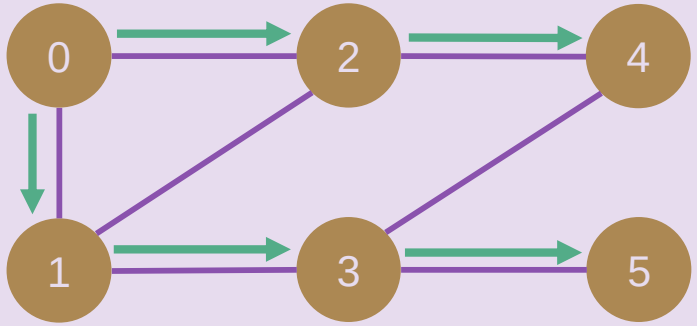
Graph Searching

There are two main ways of searching through or traversing a graph: breadth-first or depth-first.

A

Breadth-First Search (BFS)

Start at some vertex we call the root of the search. From this root, visit all adjacent vertices first before moving to the next level. This can be done with the use of the queue data structure.



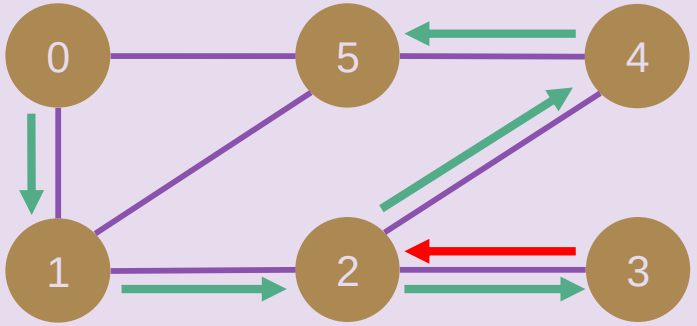
Applications

Using BFS, we can solve the shortest path problem for unweighted graphs, as it is ensured that we go from the source to the destination in the smallest number of vertex-vertex traversals.

B

Depth-First Search (DFS)

Start at some vertex we call the root of the search. From this root, keep going to the next level of vertices until a dead-end is reached. Once a dead-end is keep moving up levels until a new DFS can be started on an unvisited node. This can be done with the use of a stack data structure.



Applications

Has applications in finding strongly connected components in a graph (subgraphs where all vertices are reachable from every other vertex, but more on this in the future).

Graph Searching

Sample implementation (using the object version of adjacency list)

A

Breadth-First Search (BFS)

```
int main(){
    //setup graph

    queue<Vertex*> q;
    q.push(root);
    while(q.size()>0){
        Vertex* v = q.front(); q.pop();
        for(int i=0; i<v->adj.size(); i++){
            q.push(v->adj[i]);
        }
    }
}
```

B

Depth-First Search (DFS)

```
int main(){
    //setup graph

    stack<Vertex*> s;
    s.push(root);
    while(s.size()>0){
        Vertex* v = s.top(); s.pop();
        for(int i=0; i<v->adj.size(); i++){
            s.push(v->adj[i]);
        }
    }
}
```

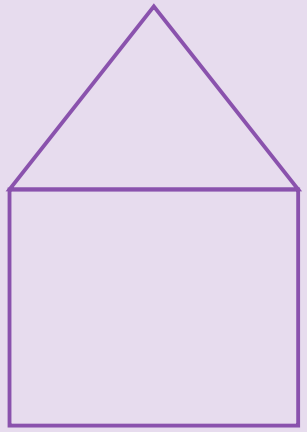
Note

To avoid looping infinitely, we keep track of which vertices have already been visited in the course of the search, typically using booleans. If a vertex has been visited before, we skip it because we would already be repeating a part of the search we did before.

Euler Walk

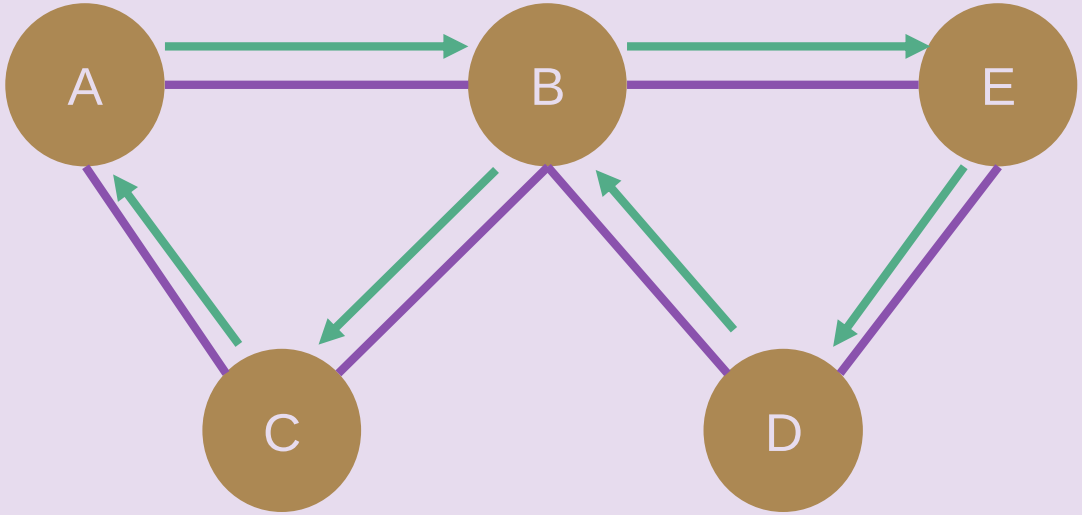
An **Euler Walk** (sometimes called an Euler Path) is a walk that goes through every edge in the graph exactly once.

Example: Drawing the house below without lifting your pen is an example of an Euler Walk.



Euler Tour

An **Euler Tour** (sometimes called an Euler Cycle) is an Euler Walk that starts and ends on the same vertex.



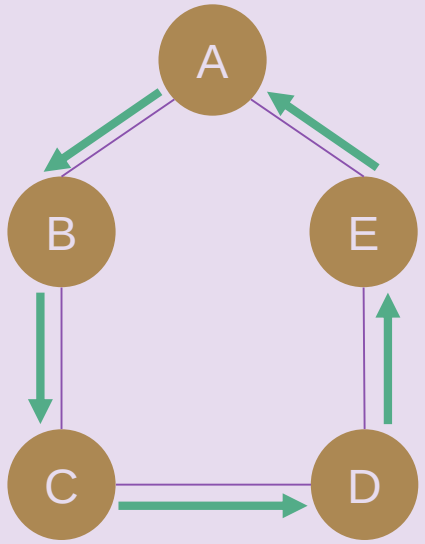
Eulerian Graphs

A

Euler Tour

Theorem: A connected graph (meaning there is a path from each node to every other node) has an Euler Tour if and only if every vertex in the graph has even degree.

Intuition: To traverse every edge exactly once and go back to the start vertex, the number of ways to enter a vertex must be equal to the number of ways to exit.

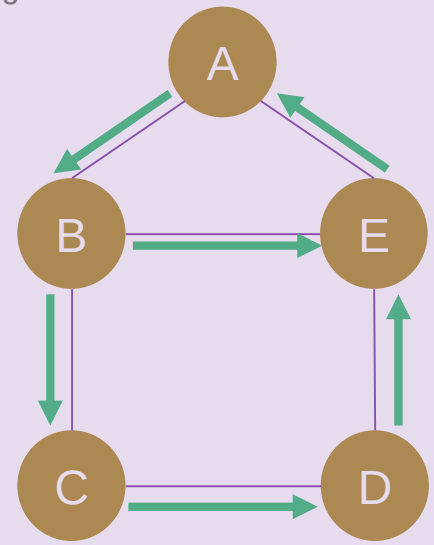


B

Euler Walk

Corollary: A connected graph has an Euler Walk if and only if there are at most two vertices with odd degree

Intuition: Start at one of the vertices with odd degree. If there is an edge that leads to the other vertex of odd degree, ignore it first. Find an Euler Tour starting from this initial vertex then traverse the previously ignored edge.



Note

For those interested in the algorithm to determine an Euler Tour/Walk, look up Hierholzer's Algorithm.

Directed Acyclic Graph

As the name implies, a **Directed Acyclic Graph** or DAG is simply a directed graph without any cycles.

DAGs are important because they allow us to capture the reality of dependencies (tasks that make prerequisites of other tasks).

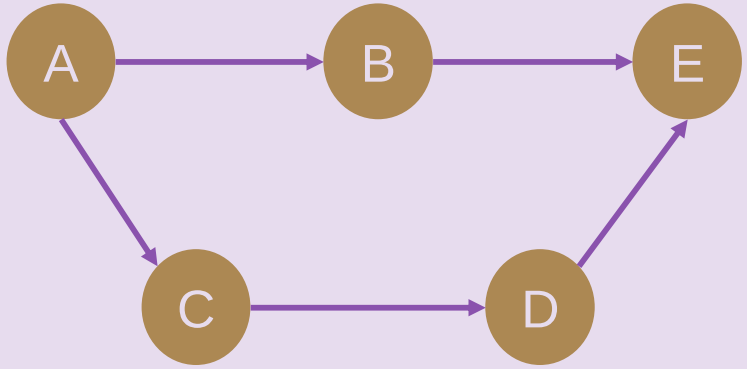
Example:

Having to take some subjects before others.

A certain program needing the output of another program before executing.

Topological Sorting

A topological sorting of a DAG is a list of all the vertices such that each vertex v appears before all other vertices reachable from v. Note that a single DAG may have multiple different topological orderings.

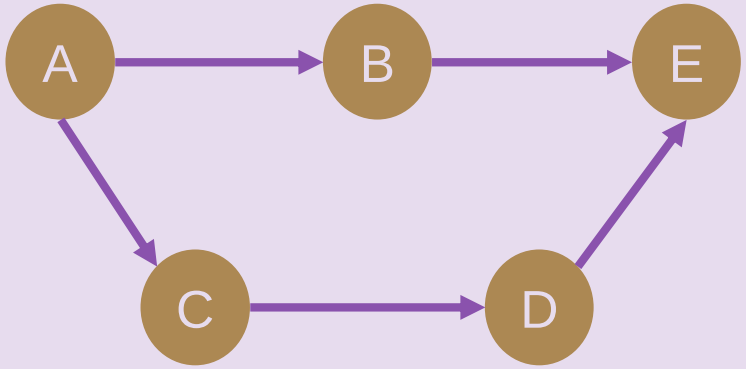


Sample Topological Sorting: A, B, C, D, E
Another possible sorting is A, C, D, B, E

Topological Sorting

Theorem: Every finite DAG has a topological ordering. We can show this by starting a search from the *minimal* elements.

A vertex v is *minimal* if and only if v is not reachable from any other vertex (has an in-degree of 0).



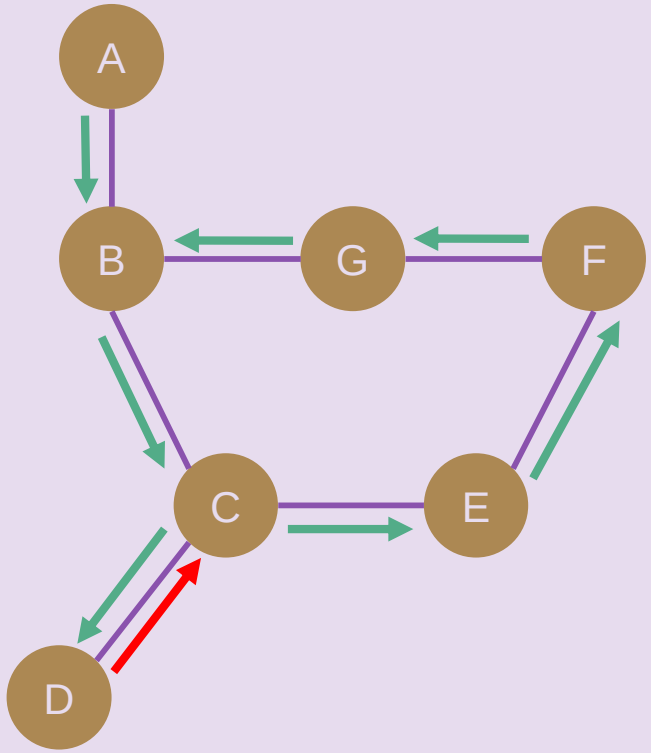
Vertex A is the only minimal vertex in the above graph

Towards an Algorithm

Generating a topological ordering can be done with a BFS-like approach. Starting a search from minimal elements, we can easily generate a topological ordering. Once a vertex has been visited, we can remove it from the graph (subsequently removing all of its edges). Assuming a valid DAG, deleting a vertex will result in new minimal elements. We then place these new minimal elements into our queue and continue our search from there. The order with which we visit vertices and delete them from the graph should be a topological ordering.

Cycle Finding

Finding the cycles within a graph can be done using a DFS-like approach. Start the search from some root vertex. When processing a vertex, we know that a path exists from the root to the current vertex using the vertices we passed through in the previous iterations, excluding those we have already backtracked from. Because of this, if there is an edge from the current vertex to some vertex already in this path, we know that a cycle exists from that vertex back to itself.



Cycle exists: B, C, E, F, G

Sample Implementation (using object version of adjacency list and recursion instead of a stack)

```
void DFS(Vertex* v, Vertex* p){
    v->inPath = true;
    for(int i=0; i<v->adj.size(); i++){
        if(v->adj[i] == p) continue;
        if(v->adj[i]->inPath) //cycle found
        DFS(v->adj[i], v);
    }
    v->inPath = false;
}

int main(){
    //setup graph

    DFS(root, NULL);
}

//Note that this only finds cycles reachable from root.
//You may have to repeat the process for other vertices.
```

Challenges (to be submitted)

- Codeforces 292B – Network Topology
- UVa 280 – Vertex
- UVa 10150 – Doublets
- UVa 459 – Graph Connectivity
- UVa 10203 – Snow Clearing
- UVa 11060 – Beverages
- UVa 10116 – Robot Motion
- UVa 12582 – Wedding of Sultan
- UVa 572 – Oil Deposits

Challenges (at least 2, the rest is optional)

- UVa 1103 – Ancient Messages
- Codeforces 510C – Fox and Names
- Codeforces 115A – Party
- Codeforces 475B – Strongly Connected City
- Codeforces 22C – System Administrator
- Codeforces 402E – Strictly Positive Matrix

IOI Training 2017 - Week 3
Dynamic Programming

Vernon Gutierrez

March 2017

1 Introduction

There is already plenty of excellent material on the internet about dynamic programming, and I believe many of our NOI participants are already familiar with DP. So instead, in this tutorial, I will simply give you some tips on implementation, and on how to make it easier for you to get divine inspiration to come up with a DP solution. If you still don't know what the fuss is all about, check out the following videos from MIT to get you started.

DP is seriously one of the coolest things in algorithms, so you have to learn it. What makes it so cool is that with just a little bit of effort, it is suddenly possible to transform exponential time algorithms into polynomial time algorithms. It is also an interesting combination of being formulaic while requiring creativity.

What DP is all about can be summarized in the following equation:

$$DP = brute force + memoization \tag{1}$$

That's all there is to it. First you need to see a few classic DP problems before what's written below will make sense. So here you go. Come back after you've read items 1-14, 27, and 55.

Finished? Ok, good. Now we can get to business.

2 The Brute Force Part

2.1 The right way to think about recursion for DP

Maybe when you think about recursion, you think about function calls getting pushed and popped off the stack. That is a correct picture of how function calls work when executed, but it is not necessarily the best picture to have when trying to come up with recursive algorithms.

Especially for brute force and DP problems, the way I like to think about recursion is to imagine myself having a super-parallel computer. If I write a recurrence like `F(n-1) + F(n-2)`, what I'm really doing is making the universe branch off into two. In one universe, I compute F_{n-1}. In another universe, I compute F_{n-2}. And then I am doing both computations in parallel and adding up the results when the two computations are done. Another example. For the knapsack problem, if I write `max(v[i] + OPT(i-1, W - w[i]), OPT(i-1, W))`, what I'm really doing is creating two universes. One in which I chose the ith item, another in which I did not. And then those two universes evolve in parallel, and may continue to split. I'm sure that in one of those universes, after n steps, I have the optimum set. The way I obtain this optimum set in order to print it out is by killing off the other universes using the `max` function. Of course, in reality, the computer doesn't really become super-parallel, and I'm not really splitting universes. The computer executes each branch of my recurrence in sequence, but when trying to come up with the DP solution, I don't need to think about that. It is more fruitful for me to imagine that I have a more powerful computer that can do an infinite number of things in parallel, and the language of recursion allows me to play that trick.

2.2 Brute Force vs. DP

Many classic DP problems are easy, because all that is required is brute force. And brute force requires almost no thinking. Right?

Wrong. Equation (1) above is not 100% accurate. In reality, the brute force required for DP needs to be a little bit smart so that subproblems can overlap. For example, if in the Knapsack problem, we used the set of items already chosen to represent our subproblem (shown in the code below), then even if we memoized our solution, the worst case running time would be $O(n^2 2^n)$ rather than $O(nW)$. (There would be $O(n2^n)$ subproblems, and each subproblem would require $O(n)$ time to copy the set.)

```
int OPT(int i, set<int> &s) {
    if(i == 0) {
        if(sum_weights(s) <= W) {
            return sum_values(s);
        } else {
            return -1;
        }
    } else {
        set<int> new_set;
        new_set.insert(s.begin(), s.end());
        new_set.insert(i);
        return max(OPT(i-1, new_set), OPT(i-1, s));
    }
}
```

The key conflict in designing DP solutions is this. You need your subproblem to contain enough information to be able to take into account all the constraints. But you also don't want to put so much information that subproblems don't overlap enough to make the running time go down to polynomial. In the knapsack problem, if the subproblem only has one parameter i, it is not enough to correctly capture the fact that the knapsack capacity is limited. On the other hand, all we really need to capture is the capacity we have left in order to solve our subproblems. We don't need to care about exactly what items have already been chosen.

2.3 DP patterns

One way to more easily come up with the DP recurrence is to recognize that most DP problems fall into certain patterns. This excellent TopCoder recipe explains what some of those patterns are.

I earlier stated that DP is formulaic. The reason is, when designing a DP solution, you often just need to follow these steps:

1. Represent the state or subproblem
2. Figure out the state transitions or "moves"
3. Analyze the running time
4. Implement the brute force recurrence
5. Memoize

Steps 3-5 are often easy and straightforward, once you've gotten a fair bit of practice with DP problems. Step 1 is usually the hardest. Step 2 sort of follows once you have Step 1, but it still requires some thinking. In some cases, Step 2 needs to be improved by clever means to reduce the degree of the running time of your solution by 1. But we will talk about that some other time.

2.4 Figuring out the state

The most common states for DP are DP on prefixes, DP on substrings, and DP on two (or more) pointers.

DP on prefixes means, that a subproblem looks something like $OPT(i, ...)$, where we are considering the prefix of the input that spans from item 1 to item i. The answer to a subproblem depends on subproblems for a "prefix" of the given input. An example of a problem that fits this pattern is weighted interval (or job) scheduling. To solve weighted interval scheduling, we needed to consider either the prefix of intervals without the last interval, or the prefix of intervals without all of the intervals which overlap with the last interval.

DP on substrings means, that a subproblem looks something like $OPT(i, j, ...)$, where we are considering the substring of the input that spans from item i to item j. An example of a problem that fits this pattern is matrix chain multiplication. To solve matrix chain multiplication, we guess a splitting point k, and then recursively solve the substrings of matrices $OPT(i, k)$ and $OPT(k + 1, j)$.

DP on two (or more) pointers is like DP on prefixes, but this time, there are multiple strings, and not all of the pointers have to move backward to constitute a prefix. An example of a problem that fits this pattern is longest common subsequence.

2.5 Figuring out the transitions

DP is usually about choice. Choose a subset or combination that maximizes this. Or minimizes that. It is too difficult to think about choosing an entire subset or combination. So what we do in DP solutions is to break up the choice into *stages*. To choose a subset, I break it down into smaller choices of either choosing one item or not. Once you think about choice in DP in this way, as making step-by-step choices, it becomes much clearer what you need to do to come up with the recurrence.

There are only two basic transition patterns for DP: binary choice or multiway choice.

Binary choice means, for each state of your problem, you have two *mutually exclusive* options on how to proceed. Usually, this is because your subproblems represent subsets, and you are choosing on whether or not to include the last item. Examples of binary choice DP problems are knapsack problem and weighted interval scheduling.

Multiway choice means, for each state of your problem, you have multiple *mutually exclusive* options on how to proceed. An example of this is the coin change problem, where starting from some amount, you have multiple possible single moves or choices on how to reduce the amount by using just one coin.

2.6 Analyze the running time

Analyzing the running time of DP solutions is usually straightforward. Just multiply the the number of subproblems with the amount of time spent on each subproblem, counting recursive calls as free due to memoization. The number of subproblems is just the product of the maximum sizes of the parameters to your recurrence. This is usually easy to compute from the appearance of your recursive function. If your recursive function is $OPT(i)$, then the number of subproblems is $O(n)$. If the recursive function is $OPT(i, j)$, then it is something like $O(n^2)$. $OPT(i, j, k)$? $O(n^3)$. Then you may need to multiply with an extra $O(n)$ factor to get your final running if you did a for-loop inside the recursive function. For example, there are $O(n)$ states for coin change, where n is the required amount. If there are d denominations, then the recurrence would be a multiway choice which requires a loop over d items (or d if statements). This means the running time is $O(nd)$. Don't forget to do this step before implementing, or you might waste time implementing a TLE solution.

3 The Memoization Part

It helps to have a mental template for implementing DP problems. That way, during contest time, all you have to focus on is finding the recurrence, and the implementation follows automatically. The personal template that I follow is the following (feel free to deviate from this):

```cpp
#include <iostream>
#include <cstring>
#define N 1001 // 1 + whatever the maximum size of the first parameter is.
#define M 1001 // 1 + whatever the maximum size of the second parameter is.
// .. and more if there are more parameters

using namespace std;

int memo[N][M]; // Change the type to long long or another type if needed.
                // Add more dimensions if needed.

int OPT(int n, int m) {
    if(memo[n][m] == -1) {
        int ans;
        // solve the recurrence here and save the answer to ans
        memo[n][m] = ans;
    }
    return memo[n][m];
}

int main() {
    for(int t = 0; t < n_testcases; t++) {
        memset(memo, -1, sizeof memo);
        int x, y;
        // read inputs
        cout << OPT(x, y) << endl;
    }
    return 0;
}
```

For example, here's how I would write a recursive, memoized Fibonacci function:

```cpp
#include <iostream>
#include <cstring>
#define N 1001
using namespace std;

long long memo[N];

long long F(int n) {
    if(memo[n] == -1) {
        int ans;
        if(n == 0) ans = 0;
        else if(n == 1) ans = 1;
        else ans = F(n-1) + F(n-2);
        memo[n] = ans;
    }
    return memo[n];
}

int main() {
    memset(memo, -1, sizeof memo);
    cout << F(1000) << endl;
    return 0;
}
```

One common mistake is to create an array of size one too small than you need. For example, if we had defined `N` to be 1000 above, the program wouldn't have worked. Or worse, because of the way C++ behaves, it would've only probabilistically worked. Or it might consistently work on your computer but always fail to work when you submit to UVa or Codeforces. That is a debugging nightmare! To avoid this, always remember to make your memo the right size. Usually, it is one larger than the maximum expected input size. Some competitive programmers prefer to just add a lot of allowance to the memo size to avoid the pain, so they would instead define `N` to be 1010 above. I personally have not found enough reason to resort to such a tactic, but if you find yourself wasting time dealing with off-by-one errors too often, this tactic might be of benefit to you.

Warning: the `memset` function does not really set each individual element to some specified number. Instead, it sets each individual byte of memory spanned by the specified addresses to the specified byte. It is too time consuming and annoying to explain how it really works. The bottom line is that it happens to work for setting all elements to 0 or to -1, because of how binary numbers work. It does not in general work for any arbitrary value. For these cases, you should write a for loop to set the elements manually. You will rarely need to do so because -1 is the most common valid dummy value we use for DP problems. But if -1 happens to be a legitimate answer for one of your DP subproblems, then you need to use a different dummy value, and you also need to use a for-loop to initialize your memo with this other dummy value. There is a safer, more C++-ish alternative for 1D arrays, `fill_n`, but it only really works well with 1D arrays. If you want to understand what can go wrong with memset, try to compile and run the code below:

```cpp
#include <iostream>
#include <algorithm>
#include <cstring>
#define N 10
using namespace std;

int memo1[N];
int memo2[N][N];

int main() {
    memset(memo1, 2, sizeof memo1);
    for(int i = 0; i < N; i++)
        cout << memo1[i] << " ";
    cout << endl;

    fill_n(memo1, sizeof memo1, 2);
    for(int i = 0; i < N; i++)
        cout << memo1[i] << " ";
    cout << endl;

    memset(memo2, 2, sizeof memo2);
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            cout << memo2[i][j] << " ";
        cout << endl;

    fill_n(*memo2, sizeof memo2, 2);
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            cout << memo2[i][j] << " ";
        cout << endl;

    return 0;
}
```

You also should not use `memset` for non-integer arrays. Try changing the types of the arrays above to hold `double` values instead. What happens when you run it?

Here's a bottom-up implementation for Fibonacci:

```cpp
#include <iostream>
#define N 1001
using namespace std;

long long memo[N];

int main() {
    memo[0] = 0;
    memo[1] = 1;
    for(int n = 2; n < N; n++) {
        memo[n] = memo[n-1] + memo[n-2];
    }
    cout << memo[1000] << endl;
    return 0;
}
```

The advantage of writing the solution this way is that we don't have function calls, leading to a slightly faster program. We also never need to deal with stack overflow errors if the recursion becomes too deep. We also don't need to initialize the array to hold some dummy value. It's also now much clearer that the running time of memoized Fibonacci is $O(n)$. There's also a DP optimization which you will see in the future, which requires that the DP has already been written in a bottom-up style.

The disadvantage of writing the solution this way is that we needed to carefully think about the order in which to put elements in the memo, so that by the time we need answers to the smaller subproblems, they are already in the memo. This is something we didn't have to think about when doing the top-down version. This may not be an obvious disadvantage for Fibonacci, but for some problems, like matrix chain multiplication, thinking about the order is wasted contest time.

4 A Worked Example

Let's try applying all of the ideas above to solve an actual problem, Codeforces 118D - Caesar's Legions. Before you read the explanation below, I recommend trying to solve it yourself first.

4.1 Figuring out the state, attempt 1

In this problem, we need to form sequences of n_1 footmen and n_2 horsemen, subject to certain restrictions. What makes this problem interesting is that there are several variables here. There are many possible ways to form a DP state/subproblem, and it's not immediately clear what our states/subproblems should be.

One very useful problem strategy that applies to many kinds of problems, and not just DP, is to simplify the problem. There are too many things we have to do in this problem: form a sequence of footmen and horsemen, make sure not too many footmen are consecutive to each other, and make sure not too many horsemen are consecutive to each other. Maybe that's a little bit to difficult to think about all at once. So what we will do is first to consider an easier version of the problem. Let's say we just need to count the total number of sequences of n_1 footmen and n_2 horsemen, without the additional restrictions. How would we solve this problem? Think about it for a bit before moving on to the next paragraph.

A non-brute force solution would be to realize that all we are really choosing is where to put the n_1 footmen among the $n_1 + n_2$ soliders. This is $\binom{n_1+n_2}{n_1}$. We could have also chosen where to put the n_2 horsemen instead, giving us $\binom{n_1+n_2}{n_2}$, which happens to be equal to $\binom{n_1+n_2}{n_1}$.

But this is too smart! It's so smart, that it's now quite difficult to add into our holy, pristine formula the constraints we initially decided to ignore. Remember, $DP = brute force + memoization$. So we need to think brute force. Let's try to be dumber. Thinking about producing entire sequences at once is too hard. So, we will break down the choice for the entire sequence into a series of smaller choices or *stages*, and consider building sequences step-by-step.

This naturally leads us into thinking about doing DP on prefixes. A first attempt at the state might be $C(i)$, where i is the number of soldiers we have yet to pick, $0 \leq i \leq n_1 + n_2$.

4.2 Figuring out the transitions, attempt 1

The number of sequences we can form with 0 soldiers is just 1, the empty sequence. Now, for some position $i > 0$, we have two choices: we can either let a footman stand there, or a horseman. This naturally leads to the following recurrence:

$$C(i) = \begin{cases} 1 & i = 0 \\ C(i-1) + C(i-1) & i > 0 \end{cases} \tag{2}$$

But woops, $C(n_1+n_2)$ is $2^{n_1+n_2}$, and does not agree with the non-brute force solution. This tells us something is wrong. Indeed, we failed to consider the restriction that we only have n_1 footmen and n_2 horsemen.

4.3 Figuring out the state, attempt 2

In order to take this restriction into account, our state representation is not enough. We have to keep track of how many footmen and horsemen we have remaining at our disposal. The most natural way to do that is to add two new parameters to our state: f representing the number of footmen remaining, and h representing the number of horsemen remaining. This should be reminiscent of the knapsack problem, where we added an extra parameter to keep track of the remaining capacity. Note that instead of doing this, one other obvious way to change our state would be to add s, the sequence of footmen and horsemen we have made so far, and to determine f and h from there. But note that this is too much information! We already saw how this led to a bad solution for knapsack, so let's not make that mistake again. What really matters for a subproblem is only the number of footmen and horsemen remaining, not the entire sequence of footmen and horsemen we have chosen so far. Summarizing, our state would then be $C(i, f, h)$. Stop and think about what the recurrence relation should look like if we have this kind of state representation.

4.4 Figuring out the transitions, attempt 2

Like before, the number of sequences we can form with 0 soldiers is just 1. Again, for some position $i > 0$, we have two choices: we can either let a footman stand there, or a horseman. But we can only do either move if we still have footmen (or respectively, horsemen) remaining:

$$C(i,f,h) = \begin{cases} 1 & i = 0 \\ C(i-1,f-1,h) + C(i-1,f,h-1) & i > 0, f > 0, h > 0 \\ C(i-1,f-1,h) & i > 0, f > 0, h = 0 \\ C(i-1,f,h-1) & i > 0, f = 0, h > 0 \\ 0 & i > 0, f = 0, h = 0 \end{cases} \tag{3}$$

The last case can't really happen (why?), but we put it up there just as a sanity check. The answer we want is $C(n_1 + n_2, n_1, n_2)$. This now correctly computes the number of sequences, without the restrictions about footmen or horsemen standing consecutively. But because we've now set up a brute force solution, we're ready to *extend* it to handle the additional constraints.

4.5 Figuring out the state, attempt 3

Let's first figure out how to handle the restriction that at most k_1 footmen may stand consecutive to each other. The most natural way to do this is to again add a parameter to the state, k_f denoting the number of consecutive footmen we have, thus $C(i, f, h, k_f)$. Again, stop and think about what the recurrence should look like given this state representation.

4.6 Figuring out the transitions, attempt 3

How do the moves we have available change with this extra restriction? Every time we choose to add a footman to the sequence, we have to increase k_f. If k_f is already k_1, we can no longer add a footman, or we will have too many consecutive footmen. We're forced to add a horseman. Adding a horseman resets the number of consecutive footmen to 0:

$$C(i, f, h, k_f) = \begin{cases} 1 & i = 0 \\ Foot(i, f, h, k_f) + Horse(i, f, h, k_f) & i > 0 \end{cases} \tag{4}$$

where:

$$Foot(i, f, h, k_f) = \begin{cases} C(i-1, f-1, h, k_f + 1) & f > 0, k_f < k_1 \\ 0 & otherwise \end{cases} \tag{5}$$

and:

$$Horse(i, f, h, k_f) = \begin{cases} C(i-1, f, h-1, 0) & h > 0 \\ 0 & otherwise \end{cases} \tag{6}$$

The answer we want is $C(n_1 + n_2, n_1, n_2, 0)$.

We can actually write this a little bit better by letting k_f denote the number of footmen we can *still* add consecutively, rather than the number of footmen that we have *already* added. Adding a footman to the sequence decreases this number by one. If k_f is zero, then we know we have already added k_1 footmen consecutively, and we shouldn't add another one. On the other hand, adding a horseman resets k_f to k_1, since we are free to add up to k_1 consecutive horsemen again.

$$C(i, f, h, k_f) = \begin{cases} 1 & i = 0 \\ Foot(i, f, h, k_f) + Horse(i, f, h, k_f) & i > 0 \end{cases} \tag{7}$$

where:

$$Foot(i, f, h, k_f) = \begin{cases} C(i-1, f-1, h, k_f - 1) & f > 0, k_f > 0 \\ 0 & otherwise \end{cases} \tag{8}$$

and:

$$Horse(i, f, h, k_f) = \begin{cases} C(i-1, f, h-1, k_1) & h > 0 \\ 0 & otherwise \end{cases} \tag{9}$$

This formulation is slightly better than the first one, because it more easily lends itself to a bottom-up implementation. The answer we want is $C(n_1 + n_2, n_1, n_2, k_1)$.

4.7 Figuring out the state and transitions, attempt 4

After we've figured this out, adding in the restriction for the horsemen is now easy, since it's just symmetric to the footmen case. Our new state will be $C(i, f, h, k_f, k_h)$. Before reading the recurrence below, again, try to figure it out on your own first.

Done? Ok. Here's the recurrence:

$$C(i,f,h,k_f,k_h) = \begin{cases} 1 & i = 0 \\ Foot(i,f,h,k_f,k_h) + Horse(i,f,h,k_f,k_h) & i > 0 \end{cases} \tag{10}$$

where:

$$Foot(i,f,h,k_f,k_h) = \begin{cases} C(i-1,f-1,h,k_f-1,k_2) & f > 0, k_f > 0 \\ 0 & otherwise \end{cases} \tag{11}$$

and:

$$Horse(i,f,h,k_f,k_h) = \begin{cases} C(i-1,f,h-1,k_1,k_h-1) & h > 0, k_h > 0 \\ 0 & otherwise \end{cases} \tag{12}$$

The answer we need is $C(n_1+n_2, n_1, n_2, k_1, k_2)$. Before reading the next section, try to figure out the running time of our algorithm, assuming we properly memoized it.

4.8 Analyzing the running time, attempt 4

The running time of our current solution is $O((n_1+n_2)n_1n_2k_1k_2)$. Depending on our implementation, this is a risky AC. It appears that we have too many parameters in our state here. Perhaps we can safely reduce them?

4.9 Attempt 5

Notice that we don't really need to keep track of how many soldiers we still need to put in line. That number can be easily deduced from two other parameters: the number of footmen and the number of horsemen. We can thus safely eliminate the first parameter from our DP state and get a faster solution. This technique works well for plenty of DP problems. This is one of the things you should try when your DP solution is too slow: try to eliminate parameters that are not really independent, but can be derived, from the others. Before reading the recurrence below, try to figure it out yourself first.

$$C(f,h,k_f,k_h) = \begin{cases} 1 & f + h = 0 \\ Foot(f,h,k_f,k_h) + Horse(f,h,k_f,k_h) & f + h > 0 \end{cases} \tag{13}$$

where:

$$Foot(f,h,k_f,k_h) = \begin{cases} C(f-1,h,k_f-1,k_2) & f > 0, k_f > 0 \\ 0 & otherwise \end{cases} \tag{14}$$

and:

$$Horse(f,h,k_f,k_h) = \begin{cases} C(f,h-1,k_1,k_h-1) & h > 0, k_h > 0 \\ 0 & otherwise \end{cases} \tag{15}$$

The answer we need is $C(n_1, n_2, k_1, k_2)$. The running time reduces to $O(n_1n_2k_1k_2)$, and our solution is now a sure AC solution.

4.10 Implement the brute force recurrence

Before reading the code below, I invite you once again to try it out yourself first. Don't forget to take answers modulo 10^8.

Finished? Good. See how your implementation compares with mine:

```cpp
#include <iostream>
#define MOD 100000000
using namespace std;

int n1, n2, k1, k2;
long long C(int f, int h, int kf, int kh) {
    if(f + h == 0) return 1;
    else {
        long long ans = 0;
        if(f > 0 && kf > 0) ans = (ans + C(f - 1, h, kf - 1, k2)) % MOD;
        if(h > 0 && kh > 0) ans = (ans + C(f, h - 1, k1, kh - 1)) % MOD;
        return ans;
    }
}

int main() {
    cin >> n1 >> n2 >> k1 >> k2;
    cout << C(n1, n2, k1, k2) << endl;
    return 0;
}
```

4.11 Memoize

Finally! This is the easiest step. I'm sure you can do it on your own, before comparing with mine. I deviated from my template a little bit, just to make the code fit nicely on this page, but it is a good idea to use `#define`'s for the sizes, just so you can more quickly spot an off-by-one-error, a bug due misread input constraints, or whatever.

```cpp
#include <iostream>
#include <cstring>
#define MOD 100000000
using namespace std;

int n1, n2, k1, k2;
long long memo[101][101][11][11];

long long C(int f, int h, int kf, int kh) {
    if(memo[f][h][kf][kh] == -1) {
        long long ans;
        if(f + h == 0) ans = 1;
        else {
            ans = 0;
            if(f > 0 && kf > 0) ans = (ans + C(f - 1, h, kf - 1, k2)) % MOD;
            if(h > 0 && kh > 0) ans = (ans + C(f, h - 1, k1, kh - 1)) % MOD;
        }
        memo[f][h][kf][kh] = ans;
    }
    return memo[f][h][kf][kh];
}

int main() {
    memset(memo, -1, sizeof memo);
    cin >> n1 >> n2 >> k1 >> k2;
    cout << C(n1, n2, k1, k2) << endl;
    return 0;
}
```

It is not too hard to convert this into bottom-up style:

```cpp
#include <iostream>
#include <cstring>
#define N1 101
#define N2 101
#define K1 11
#define K2 11
#define MOD 100000000
using namespace std;

int n1, n2, k1, k2;
long long C[N1][N2][K1][K2];

int main() {
    memset(C, 0, sizeof C);
    cin >> n1 >> n2 >> k1 >> k2;

    for(int f = 0; f <= n1; f++) {
        for(int h = 0; h <= n2; h++) {
            for(int kf = 0; kf <= k1; kf++) {
                for(int kh = 0; kh <= k2; kh++) {
                    long long ans;
                    if(f + h == 0) ans = 1;
                    else {
                        ans = 0;
                        if(f > 0 && kf > 0) ans = (ans + C[f - 1][h][kf - 1][k2]) % MOD;
                        if(h > 0 && kh > 0) ans = (ans + C[f][h - 1][k1][kh - 1]) % MOD;
                    }
                    C[f][h][kf][kh] = ans;
                }
            }
        }
    }
    cout << C[n1][n2][k1][k2] << endl;
    return 0;
}
```

4.12 Attempt 6?

This problem can be solved in $O(n_1 n_2 \max(k_1, k_2))$. I leave it as a challenge for you to figure out how.

5 Subtask 1

These are warmup problems. If you are having difficulty solving the subtask 2 problems, try these first. You are not required to submit your solutions to these.

Codeforces 313B - Ilya and Queries

UVa 10130 - SuperSale

UVa 10684 - The jackpot

UVa 357 - Let Me Count The Ways

UVa 108 - Maximum Sum

6 Subtask 2

These problems are required.

Codeforces 368B - Sereja and Suffixes

Codeforces 698A - Vacations

Codeforces 626B - Cards

Codeforces 706C - Hard Problem

Codeforces 711C - Coloring Trees

Codeforces 455A - Boredom

Codeforces 474D - Flowers

7 Subtask 3

Submitting 2 out of 4 of these exempts you from submitting the others. Warning: they are hard!

Codeforces 538E - Demiurges Play Again

Codeforces 427D - Match & Catch

Codeforces 677E - Vanya and Balloons

Codeforces 543C - Remembering Strings

NOI.PH Training: Week 4

Jared Guissmo Asuncion

Contents

Chapter 1

Divide and Conquer

The divide-and-conquer paradigm is a useful technique in solving certain problems. The idea of this paradigm is simple. Suppose we have a problem whose input size is N. Then, we must

- find a way to split the big problem of size N into two smaller subproblems of size $N/2$, and
- find a way to obtain a solution for the big problem using the solutions of two smaller subproblems.

1.1 Modular Exponentiation

A good first example is modular exponentiation, but first we need some facts about integer division.

Theorem 1.1 (division algorithm in $\mathbb{Z}$)**.** Let $a, b \in \mathbb{Z}$ with $b > 0$. Then, there exists unique integers q and r with $a = bq + r$, such that $0 \leq r < b$.

Remark 1.2. From theorem 1.1, we are sure that the remainder r when a is divided by m satisfies $0 \leq r < m$.

Notation 1.3. We write $a \pmod m$ to denote the remainder when a is divided by m.

We are now ready to state the problem state the problem.

Problem 1.4. Let b, x, m be integers with n non-negative and $m > 0$. Compute for $b^n \pmod m$.

The naïve approach to this problem is to solve

$$b, b^2, b^3, b^4, \ldots, b^n \pmod m.$$

However, this will take $O(n)$ multiplications[1]. However, we can do better. For simplicity, we first assume that n is a power of 2, say $n = 2^x$. We then make the trivial observation that

$$b^n \equiv b^{n/2} \cdot b^{n/2} \pmod m.$$

We have *divided* n into two parts. We can do it again. We have that $b^{n/2} \equiv b^{n/2^2} \cdot b^{n/2^2} mod n$. Continuing, we eventually reach $b^{n/2^{x-1}} = b^2 \equiv b \cdot b \pmod m$. This means, that to solve $b^n = b^{2^x} \pmod m$, we just need to solve

$$b, b^2, b^{2^2}, \ldots, b^{2^x} \pmod m.$$

[1] Very very informally speaking, $O(n)$ means that you will take around cn multiplications where c is some constant.

These can be solved by $O(x)$ squarings (or x multiplications). Take note that $x = \log_2 n$. And so, $O(\log n)$ multiplications is clearly an improvement over the naïve approach which neeeds $O(n)$ multiplications. However, we are not yet done. What if n is not a power of 2? What if it's odd? If we assume that n is odd, then we can write it as $n = 2k + 1$ where k is an integer. Thus, we have

$$b^n \equiv b^{2n+1} \equiv b^{2n} \cdot b \pmod{m}.$$

But take note that we can do the divide and conquer thingy with b^{2n} since it's equal to $b^n \cdot b^n$. Hence, just like the approach we just finished talking about, we will only ever need to compute for $b, b^2, b^{2^2}, \ldots \pmod{m}$. Hence, we will also need to make $O(\log n)$ multiplications. Finally, we state the algorithm in an organized manner:

Algorithm 1.5 (modular exponentiation)**.** `modpow(b, n, m)`
Input b, n, m integers with n non-negative and $m > 0$.
Output $b^n \pmod{m}$.

1. If n is 0, return $1 \pmod{m}$.
2. If n is 1, return $b \pmod{m}$.
3. If $n \geq 2$:
   (a). Solve for $x =$ `modpow(b, n/2, m)`. Here, / means integer division.
   (b). If n is even, return $x^2 \pmod{m}$.
   (c). If n is odd, return $x^2 \cdot b \pmod{m}$.

Implementation 1.6. Here is an implementation of algorithm 1.5.

```
modpow(b, n, m):
    if n == 0:
        return 1
    if n == 1:
        return b
    x = modpow(b, n/2, m);
    if n%2 == 0:
        return (x*x)%m
    else
        return (b*x*x)%m
```

1.2 Binary Search

Another generic problem-solving technique that uses the divide-and-conquer paradigm is the binary search method. We state the following problem:

Problem 1.7. Let $p(x) : \{0, 1, \ldots, n-1\} \to \{\texttt{true}, \texttt{false}\}$ be a function such that

$$p(i) = \begin{cases} \texttt{false} & \text{if } i < k \\ \texttt{true} & \text{if } i \geq k. \end{cases} \tag{1.8}$$

Given the array and a way to determine the answer to $p(j)$ for any j, determine k.

The naïve approach to this problem is to go through each element of the array from left to right and check if $p(x) =$`true`. This means that you will have to call p at most n times! Not only is this inefficient, but it does not utilize the fact that the special form of the function. Now, how do we abuse this special property of f? We make the simple observation that:

Observation 1.9. If $p(j)$ is `true`, then $k \leq j$ and if $p(j) =$`false`, then $j < k$.

This means that if we choose j such that x_j is the middle element of the array, we essentially cut our search space in half! And so, we have the following algorithm:

Algorithm 1.10 (binary search)**.** `binsearch(a, b, p)`
Input integers $a, b \in \{0, 1, \ldots, n-1\}$ and a function p satisfying 1.8.
Output the lowest index $a \leq k \leq b$ such that $p(k) =$`true`.

1. If $a > b$, then k does not exist.
2. If $a = b$, check if $p(a) =$ `true`.
   - If it is true, then return a.
   - If it is false, then k does not exist.
3. Determine an integer j such that $|j - a|$ and $|b - j|$ differ by at most 1.
4. Check if $p(x_j) =$`true`.
   - If it is true, then $a \leq k \leq j$. Return `binsearch(arr, a, j, p)`.
   - If it is false, then $j < k \leq b$. Return `binsearch(arr, j+1, b, p)`.

Now, let's turn this algorithm into code. Note that we can let `j = (a + b)/2`, this is the average of a and b. However, a nasty test case might force `a + b` to overflow. However, there is no need to worry because by the power of mathematics, we have that

$$j \quad = \quad \frac{a+b}{2} \quad = \quad \frac{2a}{2} + \frac{b-a}{2} \quad = \quad a + \frac{b-a}{2}.$$

Math has saved the day! We now conclude that the safer option is to take `j = a+(b-a)/2`. Indeed, if a and b are non-negative integer whose value is at most `MAX_INT`, then $b - a$ will also be at most `MAX_INT`. Unlike God's love, your computations will not be overflowing.

Remark 1.11. Algorithm 1.10 requires $O(\log n)$ evaluations of p.

Implementation 1.12. Here is an implementation of 1.10.

```
binsearch(lo, hi, p):
   while lo < hi:
      mid = lo + (hi - lo)/2
      if p(mid) == true:
         hi = mid
      else:
         lo = mid+1
   if p(lo) == true:
      return lo
   return no_answer
```

Example 1.13. As problem 1.7 is stated rather generally, here are a few example problems wherein you can apply this algorithm.

1. Given a sorted array of integers $x_0 \leq x_1 \leq \ldots \leq x_{n-1}$, find the smallest integer in the array whose value is at least k.

   Solution: Let $p(i) = (x_i \geq k)$. Note that this satisfies 1.8. Then the answer is $x_{\texttt{binsearch(0,n-1,p)}}$.

2. Given a sorted array of integers $x_0 \leq x_1 \leq \ldots \leq x_{n-1}$, determine if the integer k is on the list.

   Wrong solution: Let $p(i) = (x_i = k)$. This does not satify 1.8.

   Solution: Let $p(i) = (x_i \geq k)$. Note that this satisfies 1.8. Say yes if and only if $x_{\texttt{binsearch(0,n-1,p)}} = k$.

3. Given a sorted array of integers $x_0 \leq x_1 \leq \ldots \leq x_{n-1}$, find the largest index i such that $x_i = k$.

   Solution: Let $p(i) = (x_i > k)$. Note that this satisfies 1.8. Let $j = \texttt{binsearch(0,n-1,p)}$. If $j > 0$, return $j - 1$ if and only if $x_{j-1} = k$. Otherwise, k does not occur.

4. Given a dictionary with words listed in alphabetical order, determine if the word `banana` is there.

5. Given a list of (not necessarily sorted) list of integers $x_0, \ldots, x_{n-1}$ whose first k entries are composite numbers and the rest are prime, find the first prime on the list.

   Solution: Let $p(i) = (x_i$ is prime). Note that this satisfies 1.8. Then the answer is $x_{\texttt{binsearch(0,n-1,p)}}$. Note that evaluating p is costly in this case. If we used the naive approach, we would be doing it $O(n)$ times. Here, we do it $O(\log n)$ times.

6. Given a polygon X completely contained in the first quadrant of the 2D cartesian plane, find the line which divides X into 2 equal parts.

   Solution: Let $p(m) =$ (the area of X on the 'left side' of the line $y = mx$ is $\geq k$). Let a and b be such that X is completely on the right side of $y = ax$ and on the left side of $y = bx$. Return $m_{\texttt{binsearch(a,b,p)}}$. Note that the domain of p is not the finite set $\{0, 1, \ldots, n-1\}$ anymore but instead $[a, b] \subseteq \mathbb{R}$. Can you formulate a new version of 1.8 that adapts to this problem?

1.3 Merge Sort

One of the fastest sorting algorithms follows the divide-and-conquer paradigm. It is more commonly known as merge sort. But before we can state the problem of sorting in full generality, we define the notion of a totally-ordered set.

Definition 1.14. A totally ordered set $(S, \leq)$ is a set S equipped with a comparison function $\leq$ on X such that for any $a, b, c \in S$, the following conditions hold:

1. **reflexivity**: $a \leq a$
2. **antisymmetry**: if $a \leq b$ and $b \leq a$, then $a = b$
3. **transitivity**: if $a \leq b$ and $b \leq c$, then $a \leq c$
4. **trichotomy**: exactly one of the following is true: $a \leq b$ or $b \leq a$

Example 1.15. If $(X, \leq)$ is a totally-ordered set, then we can define a comparison function $\geq$ on X such that $x \geq y$ if and only if $y \leq x$. Then $(X, \geq)$ is a totally-ordered set.

Example 1.16. If $(X, \leq)$ and $(Y, \leq)$ are totally-ordered sets, then we can define a comparison function $\leq$ on $X \times Y$ as follows: For any $a = (x_0, y_0), b = (x_1, y_1) \in X \times Y$, we say that $a \leq b$ if one of the following conditions is satisfied:

1. $x_0 \leq x_1$, or

2. $x_0 = x_1$ and $y_0 \leq y_1$

Then $(X \times Y, \leq)$ is a totally-ordered set.

Example 1.17. Consider the set

$$C = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}, \mathtt{H}, \mathtt{I}, \mathtt{J}, \mathtt{K}, \mathtt{L}, \mathtt{M}, \mathtt{N}, \mathtt{O}, \mathtt{P}, \mathtt{Q}, \mathtt{R}, \mathtt{S}, \mathtt{T}, \mathtt{U}, \mathtt{V}, \mathtt{W}, \mathtt{X}, \mathtt{Y}, \mathtt{Z}\}.$$

Let $f : C \to \{65, \ldots, 90\}$ such that $f(\mathtt{A}) = 65$, $f(\mathtt{B}) = 66$, etc. Define $\leq$ to be a comparision function on C such that for each $a, b \in C$, $a \leq b$ if and only if $f(a) < f(b)$ (as integers).

Example 1.18. Let S be a set and consider $(\mathcal{P}(S), \subseteq)$, where $\mathcal{P}(S)$ is the set of subsets of S and $\subseteq$ is the usual subset comparison function. Note that this is **not** a totally-ordered set. To show this, consider the set of $S = \{0, 1\}$. Note that $\{0\}$ and $\{1\}$ are subsets of S (and thus elements of $\mathcal{P}(S)$). However, observe that $\{0\} \not\subseteq \{1\}$ and $\{1\} \not\subseteq \{0\}$ and this violates the trichotomy property in definition 1.14. Indeed, it is **not** a totally-ordered set. However, such sets which do not necessarily have the trichotomy property but satisfy the three other properties are called partially-ordered sets.

Definition 1.19. If $(X, \leq)$ is a totally ordered set in which X is finite, then there exists s and t such that for all $x \in X$, we have that

$$s \leq x \qquad \text{and} \qquad x \leq t.$$

We call s the smallest element of X and t to be the largest element of X. We denote s by $\min(X)$ and t by $\max(X)$.

Notation 1.20. When the context is clear, we usually drop the $\leq$ and write just the name X of the set when we talk about a totally-ordered set $(X, \leq)$.

Problem 1.21. Let $(X, \leq)$ be a totally-ordered set. Let $x_0, \ldots, x_{n-1} \in X$. Find a one-to-one function $s : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ such that

$$x_{s(0)} \leq x_{s(1)} \leq \cdots \leq x_{s(n-1)}.$$

The **divide** step of the merge sort algorithm, as usual, involves splitting the array into two parts. The much more interesting step is the **combine** step (or the merge step). Here, we assume that we already have two sorted arrays S_1 and S_2:

S_1	S_2

Suppose we want S to be the final sorted array. As both subarrays S_1 and S_2 are sorted, then $\min(S)$ will either be $\min(S_1)$ or $\min(S_2)$, depending on which one is smaller[2]. After figuring out the first element of S (i.e. the smallest), we remove it from its respective subarray. Now, we take the minimum between $\min(S_1)$ and $\min(S_2$ to find the second (smallest) element of S. And so on.

Algorithm 1.22 (merge sort)**.** `mergesort(a, b, f)` Fix $x_0, \ldots, x_{n-1} \in X$, where X is a totally-ordered set.
Input $a, b \in \{0, \ldots, \mathrm{n}-1\}$
Output a one-to-one function $s(x) : \{a, \ldots, b\} \to \{a, \ldots, b\}$ such that $x_{s(i)} \leq x_{s(j)}$ if and only if $i \leq j$.

1. If $a = b$, then let $s(a) = a$.
2. Determine an integer j such that $|j - a|$ and $|b - j|$ differ by at most 1.
3. Let s_1 = `mergesort(a, j)` and s_2 = `mergesort(j+1, b)`.

[2]We break ties by comparing the original index, which are both integers. In this case, we take $\min(S_1)$

4. Let $i_1 = a$, $i_2 = j + 1$.
5. For $k = a, a + 1, \ldots, b$:
   - If $i_2 > b$, set $s(k) = s(i_1)$ and increment i_1 by 1.
   - If $i_1 > j$, set $s(k) = s(i_2)$ and increment i_2 by 1.
   - If $x_{s_1(i_1)} \leq x_{s_2(i_2)}$, then set $s(k) = s(i_1)$ and increment i_1 by 1.
   - Otherwise, set $s(k) = s(i_1)$ and increment i_2 by 1.
6. Return s.

Remark 1.23. Algorithm 1.22 requires $O(n \log n)$ comparisons. This is easy to see if $n = 2^x$. because you will have to call the function with an array of size n one time, with an array of size $n/2$ two times, an array of $n/4$ four times, and so on until you arrive at an array of $n/2^x = 1$ and you have to deal with n elements at each level. This means that the depth of the recursion will be $\log n$. Moreover, at each depth you will need to compare n times[3]

Implementation 1.24. Here is an implementation of 1.22.

```
mergesort(lo, hi, s):
   if lo == hi:
      return [ lo ];
   mid = lo + (hi - lo)/2
   s1 = mergesort(lo, mid)
   s2 = mergesort(mid+1, hi)
   i1 = lo
   i2 = mid+1
   arr = [ ];
   for k = lo, ..., hi:
      if i2 > hi:
         arr.append( s1[i1] )
         i1++
      if i1 > mid:
         arr.append( s2[i2] )
         i2++
      if x[s1[i1]] <= x[s2[i2]]:
         arr.append( s1[i1] )
      else:
         arr.append( s2[i2] )
```

1.4 Longest Increasing Subseqence

Suppose $a_1, a_2, \ldots, a_n$ are from a totally-ordered set $(X, \leq)$. We call this a finite sequence S in X of length n. A subsequence of S is obtained by removing some (possibly none, possibly all) of the terms of S. For people who like formal definitions:

Definition 1.25. Let $S = (a_1, a_2, \ldots, a_n)$ be a sequence. Let ℓ be a non-negative integer such that $\ell \leq n$. Consider $i_1, \ldots, i_\ell$ such that

$$1 \leq i_1 < i_2 < \cdots < i_\ell \leq n.$$

[3]For example, since you will call the function 2^i times with an input array of size $n/2^i$, this batch of calls will require about $2^i \cdot n/2^i = n$ comparisons.

Then,

$$a_{i_1}, a_{i_2}, \ldots, a_{i_\ell}$$

is a subsequence of S. Note that if $\ell = 0$, we end up with an empty sequence. For our purposes, we consider this a subsequence of S.

Optional Exercise 1.26. How many subsequences does a sequence of length n have? Assume that all the terms are distinct for simplicity.

We also define the following types of sequences:

Definition 1.27. Let $S = (a_1, a_2, \ldots, a_n)$ be a sequence in a totally-ordered set $(X, \leq)$. We write $a < b$ to denote that $a \leq b$ but $a \neq b$.

- If
$$a_1 < a_2 < \cdots < a_n$$
then S is said to be a (strictly) increasing sequence.
- If
$$a_1 \leq a_2 \leq \cdots \leq a_n$$
then S is said to be a non-decreasing sequence.
- If
$$a_1 > a_2 > \cdots > a_n$$
then S is said to be a (strictly) decreasing sequence.
- If
$$a_1 \geq a_2 \geq \cdots \geq a_n$$
then S is said to be a non-increasing sequence.

We are now ready to pose the main problem of this section:

Problem 1.28. Given a sequence S in a totally-ordered set X, find the length of the longest increasing subsequence of S.

Optional Exercise 1.29. One naïve solution to problem 1.28 has complexity $O(2^n)$. Figure it out.

1.4.1 An $O(n^2)$ solution

One solution to problem 1.28 is by using dynamic programming. Here's the solution! Spoiler alert. Let $\ell(j)$ be the length of the longest increasing subsequence ending in a_j. Now, if we know $\ell(i)$ for each $i < j$, then we will be able to solve $\ell(j)$ as follows:

$$\ell(j) = \max(S_j) \tag{1.30}$$

where

$$S_j = \{\ell(i) + 1 : i < j \text{ and } a_i < a_j\} \cup \{1\}.$$

This is because if $a_k < a_m$, then we can add a_m to the longest increasing subsequence ending in a_k. With this idea, we are now ready to discuss the algorithm.

Algorithm 1.31 (longest increasing subsequence)**.** `lisdp(a, n)`

Input a sequence $S = (a_1, \ldots, a_n)$ in a totally-ordered set X of length n

Output a function ℓ such that $\ell(i)$ is the length of the longest increasing subsequence whose last term is a_i, a function $p(i)$ such that $a_{p(i)}$ is the penultimate term of the longest increasing subsequence whose last term is a_i, and a sequence $(s_1, \ldots, s_m)$, a longest increasing subsequence of S.

- Set $\ell(1) = 1$ and $p(i) = -1$.
- Set $m = 1$;
- For each $j = 2, \ldots, n$:
  - Set $\ell(j) = 1$.
  - For each $i = 1, 2, \ldots, j-1$:
    * If $a_i < a_j$ and $\ell(i) + 1 > \ell(j)$, then let $\ell(j) = \ell(i) + 1$ and set $p(j) = i$. This solves 1.30.
  - If $\ell(j) > \ell(m)$, then set $m = j$.
- Let $t = \ell(m)$.
- Let $k = m$.
- While $k \neq -1$:
  - Let $s_t = a_k$.
  - Decrease t by 1.
  - Let $k = p(k)$.

Implementation 1.32. Here is an implementation of 1.31.

```
lisdp(a, n):
   l[1] = 1
   p[1] = -1
   max = 1;
   for j = 2, ..., n:
      l[j] = 1
      for i = 1, ..., j-1:
         if a[i] < a[j] and l[i]+1 > l[j]:
            l[j] = l[i]+1
            p[j] = i
      if l[j] > l[max]:
         max = j
   t = l[m]
   k = m
   while k != -1:
      s[t] = a[k]
      t--
      k = p[k]
```

Note that there are two nested for loops. This means that whatever's done in the innermost loop will be repeated $n(n+1)/2 = O(n^2)$ times. Finally, constructing the sequence will take at most $n = O(n)$ operations. Hence, much of the computations in the algorithm are done in the nested for loops and hence the algorithm runs in $O(n^2)$ time.

1.4.2 A faster solution

While the $O(n^2)$ dynamic programming solution is already impressive, there is an even faster solution. In this new solution, we replace the inner for-loop by something that takes $O(\log n)$ time. In algorithm 1.31, we are essentially storing all longest increasing subsequences that end in each of the a_i. Hence, we may have been storing more than one longest increasing subsequences of length r, for example. Our new idea is that we instead store one longest increasing subsequence of length r – the one whose last term is minimal.

Definition 1.33. We define $S_{j,r}$ to be the longest increasing subsequence of $a_1, a_2, \ldots, a_j$ of length r whose last term is minimal. We denote by $L_{j,r}$ to be the last term of $S_{j,r}$.

Claim 1.34. For a fixed j, we have $L_{j,1} \leq L_{j,2} \leq \ldots \leq L_{j,r_j}$ (where $r_j = \max\{\ell(1), \ldots, \ell(j)\}$).

Proof. We prove by contradiction. Suppose there exists $a < b$ such that $L_{j,a} > L_{j,b}$. Exercise: Find a contradiction. □

Now, if we have the sequences $S_{j-1,1}, \ldots, S_{j-1,r}$, whose last terms are $L_{j-1,1}, \ldots, L_{j-1,r}$, then we know that we can replace with a_j the largest $L_{j-1,t}$ such that $a_j < L_{j-1,t}$. If this does not exist, then we append a_j to the longest increasing subsequence we have so far. We acknowledge that this is a bit too much to take in, and so here is an animation available on Wikipedia demonstrating what we have just said. And so, the new and faster algorithm goes as follows:

Algorithm 1.35 (longest increasing subsequence)**.** `lis(a, n)`
Input a sequence $S = (a_1, \ldots, a_n)$ in a totally-ordered set X of length n
Output a function ℓ such that $\ell(i)$ is the length of the longest increasing subsequence whose last term is a_i, a function $p(i)$ such that $a_{p(i)}$ is the penultimate term of the longest increasing subsequence whose last term is a_i, and a sequence $(s_1, \ldots, s_m)$, a longest increasing subsequence of S.

- Set ~~$\ell(1) = 1$ and~~ $p(i) = -1$.
- ~~Set $m = 1$~~;
- Let $\mathcal{L}$ be a list (of indices). Add 1 to this list.
- For each $j = 2, \ldots, n$:
  - Use binary search to find the largest index r such that $a_j \leq a_{\mathcal{L}(r)}$.
  - If r exists:
    * Set $p(j) = p(\mathcal{L}(r))$.
    * Set $\mathcal{L}(r) = j$.
  - If r does not exist:
    * Set $p(j) = \mathcal{L}(m)$, where m is the length of $\mathcal{L}$.
    * Set Append j to $\mathcal{L}$.
- ~~Let $t = \ell(m)$.~~
- Let $k = m$, where m is the length of $\mathcal{L}$.
- While $k \neq -1$:
  - Let $s_t = a_k$.

- Decrease t by 1.
- Let $k = p(k)$.

Implementation 1.36. Here is an implementation of 1.35.

```
lis(a, n):
   p[1] = -1
   L = [1]
   for j = 2, ..., n:
      m = length(L)
      r = binsearch(1, m, p)-1   // p(i) = ( L[i] < a[j] )  You can implement your own binary search if you want.
      if r > 0:
         p[j] = p[L[r]]
         L[r] = j
      else:
         p[j] = L[m]
         L.append(j)
   k = length(L)
   while k != -1:
      s[t] = a[k]
      t--
      k = p[k]
```

Chapter 2

Number Theory

2.1 Modulo

Definition 2.1 (divisibility)**.** We say that a non-zero integer b divides an integer a (written $b|a$) if there exists $q \in \mathbb{Z}$ such that $a = bq$. In this case, we say that b is a divisor of a.

Notation 2.2 (congruence modulo m)**.** We write

$$a \equiv b \pmod{m}$$

to mean $m|(a - b)$. We read this as 'a is equivalent to b modulo m'.

Definition 2.3. Let m be a positive integer. We denote by $\overline{a}$ the set

$$\overline{a} = \{b \in \mathbb{Z} : a \equiv b \,(\text{mod } m)\}.$$

We define $\mathbb{Z}_m$ to be the set

$$\mathbb{Z}_m := \{\overline{0}, \overline{1}, \ldots, \overline{m-1}\}.$$

Definition 2.4 (addition on $\mathbb{Z}_m$)**.** Let m be a positive integer. We define addition $+_m$ on the elements of $\mathbb{Z}_m$ as:

$$\overline{a} +_m \overline{b} = \overline{a + b}.$$

Theorem 2.5. Addition on $\mathbb{Z}_m$ shares some similar properties to the normal addition $+$ on integers.

1. It is closed.

   This means that if you add any two elements of $\mathbb{Z}_m$, the result will be in $\mathbb{Z}_m$. Indeed:

   $$\overline{a} +_m \overline{b} = \overline{a + b} = \overline{r}$$

   where r is an integer $0 \leq r < m$ such that $a + b = qm + r$, as in theorem 1.1.

2. It has an additive identity, $\overline{0}$. We sometimes call this the zero element.

   This means that $\overline{a} + \overline{0} = \overline{a}$ for any element $\overline{a} \in \mathbb{Z}_m$.

3. Any element $\overline{a}$ has an additive inverse $\overline{-a} = \overline{m - a}$.

   This means that $\overline{a} +_m \overline{-a}$ is equal to the additive identity for any element $\overline{a} \in \mathbb{Z}_m$.

Definition 2.6 (multiplication on $\mathbb{Z}_m$)**.** Let m be a positive integer. We define addition $\cdot_m$ on the elements of $\mathbb{Z}_m$ as:

$$\overline{a} \cdot_m \overline{b} = \overline{a \cdot b}.$$

Theorem 2.7. Multiplication on $\mathbb{Z}_m$ shares some similar properties to the normal multiplication $\cdot$ on integers.

1. It is closed.
2. It has an multiplicative identity, $\overline{1}$.

   This means that $\overline{a} \cdot_m \overline{1} = \overline{a}$ for any non-zero element $\overline{a}$.

Notation 2.8. Instead of writing $\overline{a} +_m \overline{b}$ and $\overline{a} \cdot_m \overline{b}$, we write

$$a + b \pmod{m} \qquad \text{and} \qquad ab \pmod{m}$$

respectively.

Example 2.9. We prove the divisibility rule by 9: an integer x is divisible by 9 if and only if the sum of its digits is divisible by 9.

Proof. Let $x = a_d \cdot 10^d + a_{d-1} \cdot 10^{d-1} + \ldots + a_1 \cdot 10 + a_0$. Note that $10 \equiv 1 \pmod 9$. Hence, $10^n \equiv 1 \pmod 9$ for any non-negative integer n. Thus,

$$x \quad = \quad a_d \cdot 10^d + a_{d-1} \cdot 10^{d-1} + \ldots + a_1 \cdot 10 + a_0 \quad \equiv \quad a_d + a_{d-1} + \ldots + a_1 + a_0 \pmod 9.$$

This proves the claim. □

2.2 Prime Factorization

There are many algorithms to determine prime factors of numbers. Most of these involve choosing random integers along the way, which makes them unappealing for programming contests. And so, for most problems in programming contests, only prime factorizations of small numbers are needed to be factored in order to complete the algorithm. As these algorithms are very much standard, they will probably be used in conjuction with other algorithms. Hence, it is vital to know these basic number-theoretical algorithms.

Definition 2.10. A prime number p is an integer greater than 1 that has no positive integer divisors other than 1 and p itself.

Algorithm 2.11 (Sieve of Eratosthenes)**.** `sieve(N)`

Input an integer N

Output an array `primeBa` of length N such that for any $n \leq N$, `primeBa[n] = 1` if n is prime and `primeBa[n] = 0`, otherwise.

1. Initialize the elements of `primeBa` to 1.
2. Let `primeBa[1] = 0`.
3. For each $p = 2, 3, 4, 5, \ldots$:
   - If `primeBa[p]` is equal to 0, continue to the next value for p.
   - For each integer $k > 1$ such that $kp \leq N$, set `primeBa[k*p] = 0`.
4. Return `primeBa`.

Remark 2.12. The running time for this algorithm is $O(N \log\log N)$. In the algorithm, we make around

$$\frac{N}{2} + \frac{N}{3} + \frac{N}{5} + \frac{N}{7} + \frac{N}{11} + \dots$$

operations. And we use the fact (which we will not prove) that

$$\sum_{p,\text{ prime},\, p \le N} \frac{1}{p}$$

is approximately equal to $\log\log N$ when N is very large.

Implementation 2.13. Here is an implementation of algorithm 2.11.

```
sieve(N):
   primeBa[1] = 0
   for k = 2, 3, 4, 5, ..., N
      primeBa[k] = 1
   for p = 2, 3, 4, 5, ..., N
      if primeBa[p] == 0:
         continue
      for i = 2, 3, 4, 5, ...
         if i*p > N:
            break
         primeBa[i*p] = 0
   return primeBa
```

Theorem 2.14 (fundamental theorem of arithmetic)**.** Any positive integer $n > 1$ can be expressed as

$$n = p_1^{e_1} p_2^{e_2} \cdots p_t^{e_t} \tag{2.15}$$

where the p_i are distinct prime factors and e_i are positive integers.

Remark 2.16. We call the expression in 2.15 the prime factorization of n.

Remark 2.17. Algorithm 2.11 can be modified to find the prime factorization of all integers below N.

Definition 2.18 (greatest common divisor)**.** The greatest common divisor of two integers a and b is the largest integer d such that $d|a$ and $d|b$.

Theorem 2.19. Let

$$a = p_1^{e_1} p_2^{e_2} \cdots p_t^{e_t} \qquad \text{and} \qquad b = p_1^{f_1} p_2^{f_2} \cdots p_t^{f_t}$$

be the prime factorization of two positive integers a and b (some of the e_i and f_i may be zero). The greatest common divisor, $\gcd(a, b)$, of a and b is:

$$\gcd(a, b) = p_1^{\max\{e_1, f_1\}} p_2^{\max\{e_2, f_2\}} \cdots p_t^{\max\{e_t, f_t\}}.$$

Theorem 2.20. Here are some properties concerning the greatest common divisor:

1. $\gcd(a, 0) = 0$.
2. If $b < a$, then $\gcd(a, b) = \gcd(b, a - b)$.
3. If $b < a$, then $\gcd(a, b) = \gcd(b, r)$ for r such that $r \equiv a \pmod{b}$ and $0 \le r < b$.

Optional Exercise 2.21. Prove theorem 2.20. Hint: Use definition 2.1 when appropriate.

From theorem 2.20, we come up with this simple algorithm to find the greatest common divisor of two integers a and b:

Algorithm 2.22 (Euclidean algorithm)**.** `gcd(a, b)`

Input two non-negative integers a and b

Output $d = \gcd(a, b)$

1. If $a < b$, return `gcd(b, a)`.
2. If $b = 0$, return a.
3. Return `gcd(a%b, b)`.

Remark 2.23. The complexity of algorithm 2.22 is $O(\log n)$.

Implementation 2.24. Here is an implementation of algorithm 2.22.

```
gcd(a, b):
   if a < b:
      return gcd(b, a)
   if b == 0:
      return a
   return gcd(a%b, b)
```

Chapter 3

Combinatorics

Problem 3.1. How many ways can one arrange n distinct objects in a row of n?

There are

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$$

ways to arrange n distinct objects in a row. You have n choices for the first one. Once you've set that, you will have $n-1$ choices for the next and then $n-2$ and so on until you have 1 object left for the nth slot.

Problem 3.2. How many ways can one arrange n distinct objects in a row of k where $k \leq n$?

You will have n choices for the first slot, $n-1$ for the second, all the way down to the kth slot which will have $n-k+1$. Hence, you will have

$$n \cdot (n-1) \cdot n-2 \cdot \cdots \cdot (n-k+1).$$

You can also get the same answer by first considering a solution for the first problem, and then taking note that the order of the last $n-k$ elements does not matter (because they will not be in the row anyway). So you can jumble them up in whatever order you like. And from the answer to the problem 3.1, there are $(n-k)!$ ways to jumble them up. So with the original answer $n!$, you treat the $(n-k)!$ arrangements of the last $n-k$ items as 1. And so, you get the same answer

$$\frac{n!}{(n-k)!} = n \cdot (n-1) \cdot n-2 \cdot \cdots \cdot (n-k+1).$$

Problem 3.3. How many ways can one choose k different objects from n distinct objects where $k \leq n$?

Starting from the answer to problem 3.2, we are now allowed to jumble the first k objects as well. That means the $k!$ different arrangements of the first k items will be treated as 1. Hence, we end up with the formula:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}.$$

Remark 3.4. Zero factorial is equal to the empty product. That is $0! = 1$.

While the solutions presented in the above problems are intuitive, we have another way to approach these kinds of problems. Take for example problem 3.3. Let's abbreviate problem 3.3 as $\mathcal{P}(n,k)$. This denotes the problem of choosing k different objects from n distinct objects. If we had a solution for $\mathcal{P}(n-1,k-1)$, then we can add an nth element as a chosen one. We could also take a solution for $\mathcal{P}(n-1,k)$. In this case, we can add an nth element as an element which is not chosen (since the solution for $\mathcal{P}(n-1,k)$ already has k chosen elements). Hence, we have the relation

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}. \tag{3.5}$$

This value is called the binomial coefficient. These numbers arise in different contexts. For example, the polynomial obtained by raising $x + 1$ to the nth power will be equal to

$$(x+1)^n = x^n + \binom{n}{n-1}x^{n-1} + \binom{n}{n-2}x^{n-2} + \ldots + \binom{n}{2}x^2 + \binom{n}{1}x + \binom{n}{0}.$$

Remark 3.6. Here are some properties of the binomial coefficient.

1. For $0 \leq k \leq n$, we have $\binom{n}{k} = \binom{n}{n-k}$.
2. For $0 < k < n$, we have $\binom{n}{k}$ is divisble by n.

Optional Exercise 3.7. What is the remainder when $42^{69} + 26^{69}$ is divided by 69?

Optional Exercise 3.8. There are also a lot of other variations of counting problems. Here are some of them. Try using the answers to the previous problems, or use a similar technique to arrive at the answer.

1. How many distinct strings can you obtain by rearranging the letters of the string `BEBEBIBIGURL`?
2. How many ways can you seat yourself and $N - 1$ other people in a row if you insist on sitting with your one true love?
3. How many ways can you seat yourself and $N - 1$ other people in a row if your one true love insists on sitting at least one seat away from you?
4. Let S be a positive integer. How many positive integer solutions does $x_1 + x_2 + \ldots + x_n = S$ have?
5. Let S be a positive integer. How many non-negative integer solutions does $x_1 + x_2 + \ldots + x_n = S$ have?
6. How many ways can you assign n different pigeons in k different holes?
7. How many ways can you rearrange n pairs of (and) in a string such that each (is paired up with a) somewhere on its right?

Remark 3.9. One can implement a function which returns the binomial coefficient by means of a recursive function. If memory allows, one can do better by remembering all the results of the computations to avoid solving for the same thing twice.

Chapter 4

Exercises

Send an email **on or before 11:59PM of 2 April 2017 (Sunday)** to training@noi.ph containing (at least) the following:

- Your Codeforces, UVa, HackerRank, and ProjectEuler usernames.
- Links to the source code you submitted for the required Codeforces problems.
- Code (attached) used for the required ProjectEuler and UVa problems.
- Text, PDF or photo (attached) of all required proofs.

4.1 Required Exercises

1. Proof of 1.34
2. CF 215A: Points on Line
3. Project Euler 133: Repunit nonfactors
4. UVa 612: DNA Sorting
5. UVa 10113: Exchange Rates
6. UVa 10534: Wavio Sequence
7. UVa 10665: Contemplation! Algebra
8. UVa 13083: Yet Another GCDSUM
9. At least one of:
   (a). UVa 10247: Complete Tree Labeling (BigInt)
   (b). HackerRank Mirror : Complete Tree Labeling (Mod)

4.2 Optional Exercises

- UVa 11378: Bey Battle
- CF 294C: Shaass and Lights
- CF 327C: Magic Five
- CF 577B: Modulo Sum
- CF 272D: Dima and Two Sequences
- Project Euler 147: Rectangles in cross-hatched grids
- UVa 10229: Modular Fibonacci
- UVa 12192: Grapevine

IOI Training Week 5
DP on Trees and DAGs

Tim Dumol

Contents

1 Trees and Directed Acyclic Graphs (DAGs)

Recall from the graphs section the following definitions (if any of them feel vague, then please ask the trainer or just search for the terms):

Definition 1 (Graph)**.** A graph, G, consists of a set of v vertices (or nodes) and a set of e edges, which connect pairs of vertices.

Definition 2 (Directed Graph)**.** A directed graph, G, is a graph whose edges have a start and an end (their edges have a direction, or orientation).

Definition 3 (Chain (or Walk))**.** A chain is an alternating sequence of vertices and edges, such that v_i and v_{i+1} are connected by edge e_i.

Definition 4 (Path)**.** A (simple) path is a chain that does not have repeated vertices.

Definition 5 ((Directed) Cycle)**.** A directed cycle is a chain of vertices and edges that start and end at the same vertex.

Definition 6 (Directed Acyclic Graph)**.** A directed acyclic graph (DAG) is a directed graph which does not have any cycles.

Definition 7 (Connected Graph)**.** A connected graph is a graph such that there exists a path between every pair of vertices.

Definition 8 ((Unrooted) Tree)**.** A (unrooted) tree is a connected (undirected) graph that has v vertices and $v - 1$ edges. Equivalently, a tree is a graph such that there exists exactly one path between every pair of vertices. Equivalently, a tree is a connected acyclic undirected graph.

The vertices of the tree with connectivity 1 are called the *leaves* of the tree.

Definition 9 (Orientation of a graph)**.** An orientation of an undirected graph G is a directed graph G' formed by assigning a direction to each edge of the graph G.

Definition 10 (Rooted Tree)**.** A rooted tree is an orientation of an unrooted tree, such that there exists a vertex r, called the *root* of the tree, such that there exists a path from the root to every other vertex of the tree.

Colloquially, you can imagine getting a vertex of an unrooted tree, and then pulling it up and dangling the rest of the tree from it.

2 Dynamic Programming (DP) on Trees and DAGs

The structure of DAGs and trees make them particularly amenable to dynamic programming. This can be seen most easily through an example. And since I'm a mathy guy, we'll start with a definition (actually, several definitions):

Definition 11 (Independent Set)**.** An independent set of a graph is a set of vertices in a graph, no two of which are independent.

Definition 12 (Maximum independent set)**.** A maximum independent set of a graph is an independent set of the largest possible size for the given graph.

(Note: this is distinct from a *maximal* independent set, which is an independent set to which you cannot add any more vertices. In general, a *maximal* thing is something to which you can't add to, while a *maximum* thing is the largest possible of that thing. All *maximum* sets are *maximal*, but not vice versa.)

Definition 13 (Maximum-weight independent set)**.** A maximum-weight independent set of a weighted graph (where weights are attached to each nvertex) is an independent set of the largest possible sum of weights in the given graph.

In general, finding a maximum independent set (and by extension, the maximum-weight independent set) of a graph is an *NP-hard* problem (there is no known sub-exponential algorithm to find it). However, the simple structure of a tree makes it simple to find the maximum independent set of a tree[1].

The trick of doing DP on a DAG (and a rooted tree is a DAG) is, as with all DP, to decompose the problem into subproblems, solve the subproblems, and then put the solutions to the subproblems together. The difference with DP on a tree/DAG is that the dependency structure of your solution has been handily been given to you.[2] In particular, your DP solution will often be of the form:

```
f(vertex) := g(f(child) for child in children(vertex))
```

(substituting `neighbors(v)` for DAGs). For example, to find the maximum-weight independent set of a tree, simply root the tree[3], and try to break it down into subproblems. In this case, we can exhaustively break it down into two cases: either we pick the current vertex (in which case we can't choose its children), or we don't (in which case we can). In code:

```
typedef vector<int> EdgeList;
typedef vector<EdgeList> AdjList;

constexpr int N = 1024;

AdjList graph;
int weights[N];
// assume graph and weights are initialized
int ans[N];
fill(ans, ans+N, -1);

int f(int v) {
  if (ans[v] != -1) return ans[v];
  // choose children to put into set, so you can't choose this vertex
  int childrenSum = 0;
  for (auto it = graph[v].cbegin(); it != graph[v].cend(); ++it) {
    childrenSum += f(*it);
  }
  // choose this vertex, so you can't choose children
```

[1] There are many other classes of graphs for which it is simple to find the maximum independent set of.

[2] In fact, you may find that the call tree of your solution (which is also a DAG), happens to be pretty similar to the DAG in your problem.

[3] in this case, we can take an arbitrary node and orient all other edges away from it–in other cases, you may want to minimize the height of the tree, which you can do by finding the *center* of the graph, and setting that as the root)

```
  int selfSum = weights[v];
  for (auto it = graph[v].cbegin(); it != graph[v].cend(); ++it) {
    int child = *it;
    for (auto it2 = graph[child].cbegin(); it2 != graph[child].cend(); ++it2) {
      selfSum += f(*it2);
    }
  }
  return (ans[v] = max(childrenSum, selfSum));
}

printf("%d\n", f(root));
```

The same reasoning applies to DAGs, of course, except that instead you may have multiple starting points.

Now that the theory's done–the best way to learn DP is by practicing. So without further ado, the problems.

3 Problems

1. Unidirectional TSP (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=52)
2. Spreadsheets (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=132)
3. Not So Mobile (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=780)
4. Walk Through the Forest (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1858)
5. The Queue (https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=391&page=show_problem&problem=3003)
6. Alternative Aborescence (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2282)
7. Sultan's Chandelier (https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=2535)
8. Matrix (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=1299)
9. Dragster (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=78&page=show_problem&problem=2727)
10. Optimal Cut (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=117&page=show_problem&problem=2882)

3.1 Bonus Problems

These problems are ungraded.

1. IT Restaurants (http://codeforces.com/problemset/problem/212/E)
2. Shaass the Great (http://codeforces.com/problemset/problem/294/E)
3. Ostap and Tree (http://codeforces.com/problemset/problem/735/E)

4 References

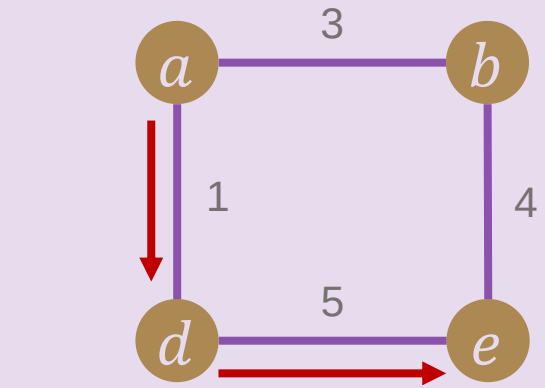
1. DP on Trees Tutorial — Codeforces (http://codeforces.com/blog/entry/20935)

GRAPH THEORY 2

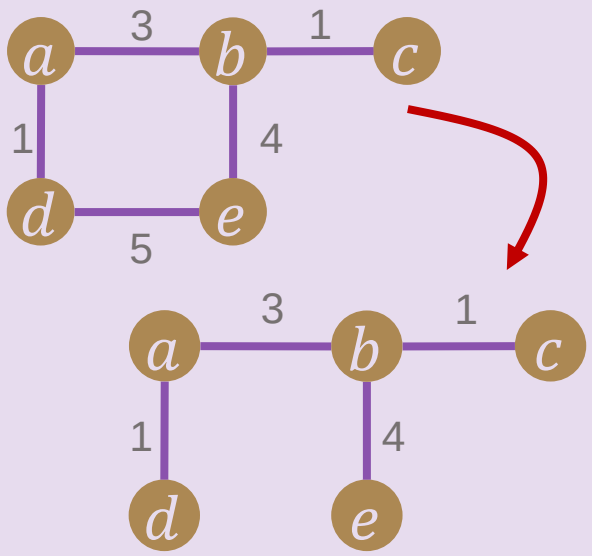
Hadrian Ang, Kyle See, April 2017

What is a shortest path?

Given a graph G, a source vertex u in G, and a destination vertex v in G, a shortest path from u to v is a path in G from u to v such that the total of the weights of all edges in the path is minimized. If G is unweighted, minimize the number of edges in the path (BFS).



$SP(a, e)$



What is a minimum cost spanning tree?

Given a graph G, a spanning tree of G is a connected subgraph of G that is a tree and contains all of its vertices. A minimum cost spanning tree of G is a spanning tree with the minimum possible total weight of all edges included in it. All spanning trees in an unweighted graph are considered minimum cost spanning trees.

Determining Shortest Paths and Minimum Cost Spanning Trees

There are standard algorithms used to determine the shortest paths and minimum cost spanning trees within graphs.

A

Single Source Shortest Path (SSSP)

- Dijkstra's Algorithm
- Bellman-Ford Algorithm

All Pairs Shortest Path (APSP)

- Floyd-Warshall Algorithm
- SSSP from each source

B

Minimum Cost Spanning Tree (MCST)

- Prim's Algorithm
- Kruskal's Algorithm
- Boruvka's Algorithm (not covered)

Notes on Shortest Path Problems

There are two typical types of shortest path problems:

- Single Source Shortest Path (SSSP) – look for the shortest path from one given vertex to any or all other vertices in the graph.
- All Pairs Shortest Path (APSP) – look for the shortest path between multiple pairs of vertices in the graph.

What algorithms to use depends on the nature of the given problem.

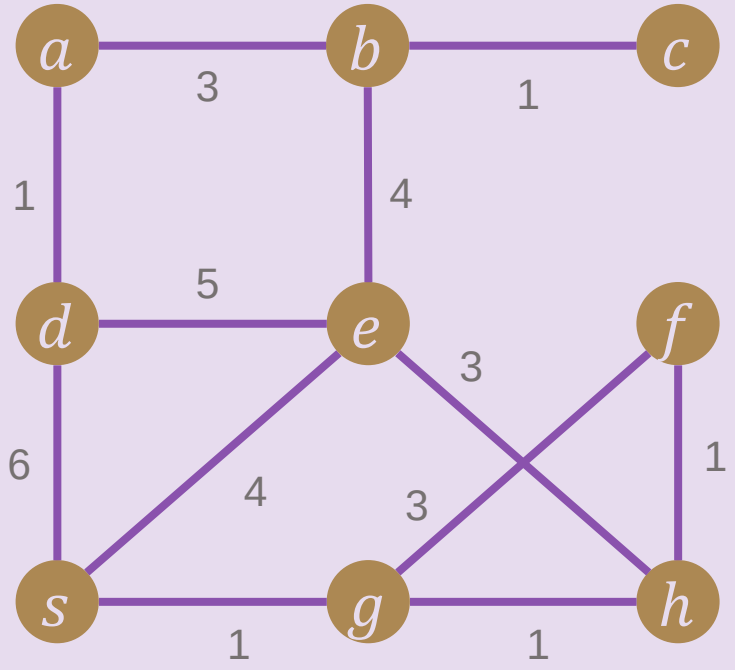
Most shortest path problems have cases where the end vertex is not reachable from the start vertex. In these cases, typically there is a default “not found” output like printing a distance of -1. All the algorithms we will discuss cover these cases by simply leaving the unreachable vertex unprocessed or never changing the initial sentinel value assigned to its distance.

Some shortest path problems require you to print the actual shortest path instead of just the distance between the vertices. All the algorithms we will discuss will have some way of “updating” current knowledge on the distances of each vertex. These are typically matched with updating a “parent” variable to allow us to trace back the path we actually took. More on this in the implementation of each individual algorithm.

Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from some given vertex to all other vertices in the graph. It is typically used for finding the shortest path between two vertices because it is the fastest of the standard shortest path algorithms.

Dijkstra's Algorithm begins with the single source vertex s having a known distance of 0 from itself and all other vertices having an unknown distance, typically labeled infinity, from s.

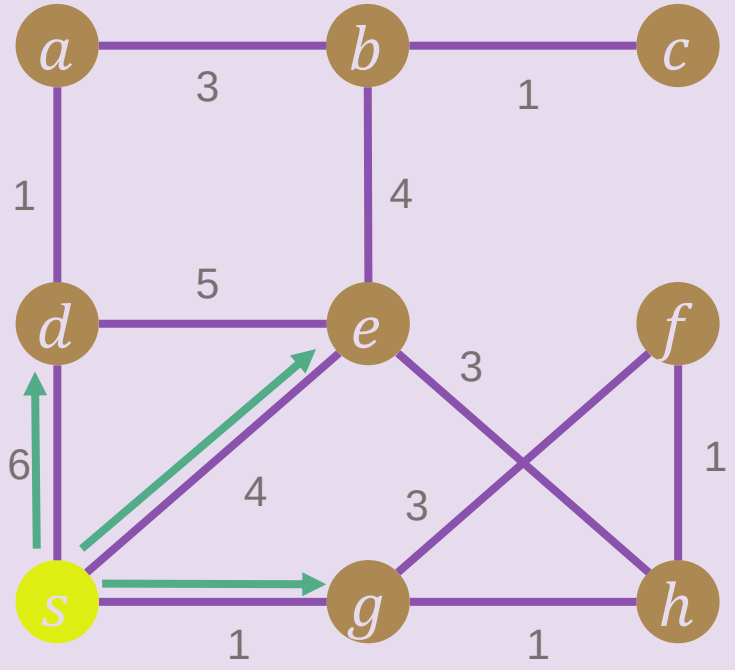


s	0	
a	∞	
b	∞	
c	∞	
d	∞	
e	∞	
f	∞	
g	∞	
h	∞	

Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from some given vertex to all other vertices in the graph. It is typically used for finding the shortest path between two vertices because it is the fastest of the standard shortest path algorithms.

The algorithm first "visits" or processes s. The algorithm determines the distance of each vertex v adjacent to s if the path consists of the edge (s, v).

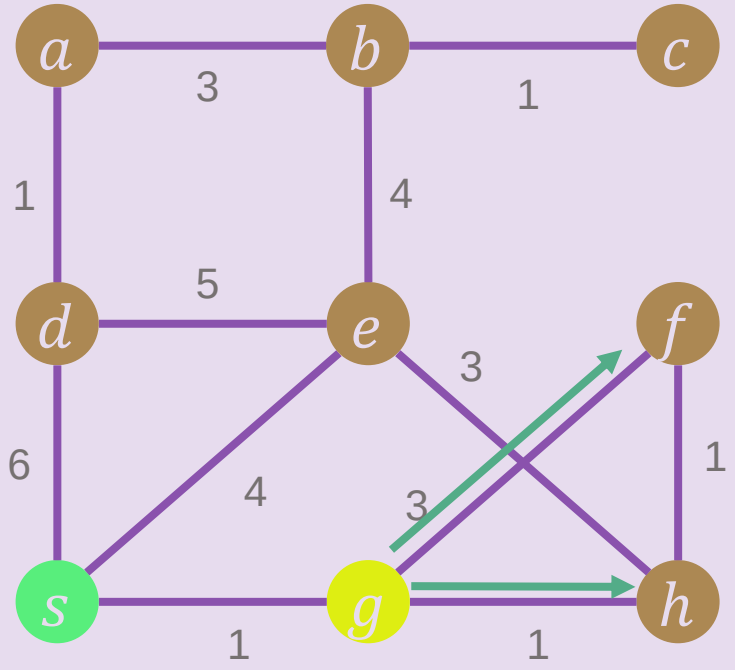


s	0	
a	∞	
b	∞	
c	∞	
d	6	
e	4	
f	∞	
g	1	
h	∞	

Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from some given vertex to all other vertices in the graph. It is typically used for finding the shortest path between two vertices because it is the fastest of the standard shortest path algorithms.

The algorithm then "visits" or processes the vertex closest to s. When visiting a vertex u, the algorithm determines the distance of each vertex v adjacent to u from s if the path includes the edge (u, v). If it is shorter than the currently known distance, replace it.

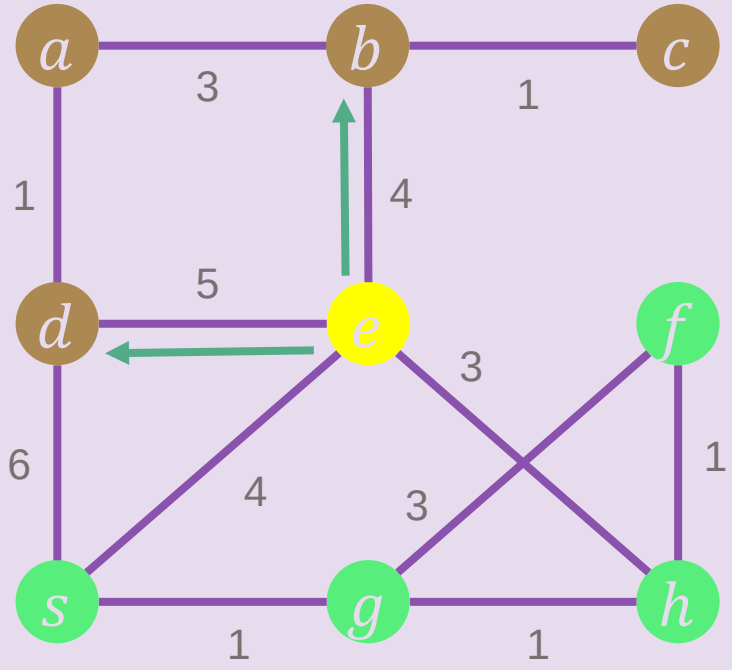


s	0	✓
a	∞	
b	∞	
c	∞	
d	6	
e	4	
f	4	
g	1	
h	2	

Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from some given vertex to all other vertices in the graph. It is typically used for finding the shortest path between two vertices because it is the fastest of the standard shortest path algorithms.

The algorithm continues visiting the vertices in order of their known distances from s, making sure not to repeat vertices already visited beforehand.

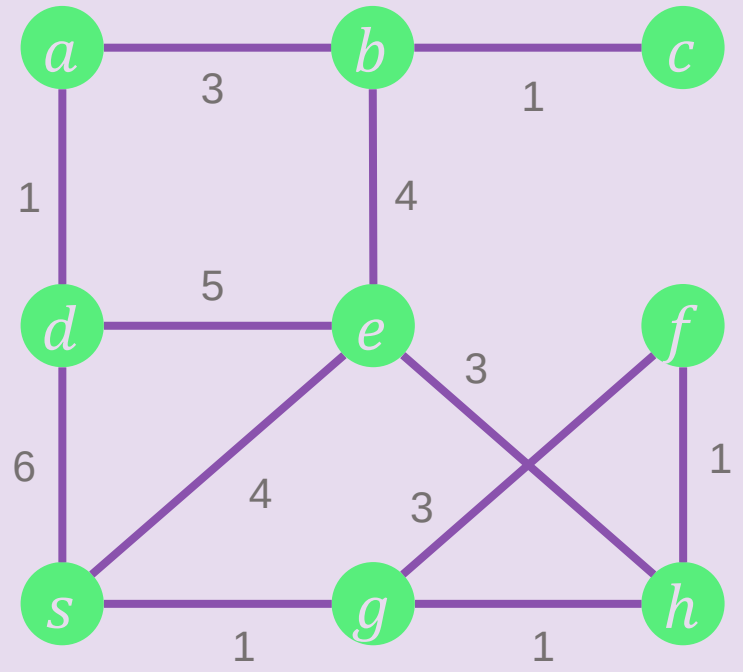


s	0	✓
a	∞	
b	8	
c	∞	
d	6	
e	4	
f	3	✓
g	1	✓
h	2	✓

Dijkstra's Algorithm

Dijkstra's Algorithm finds the shortest path from some given vertex to all other vertices in the graph. It is typically used for finding the shortest path between two vertices because it is the fastest of the standard shortest path algorithms.

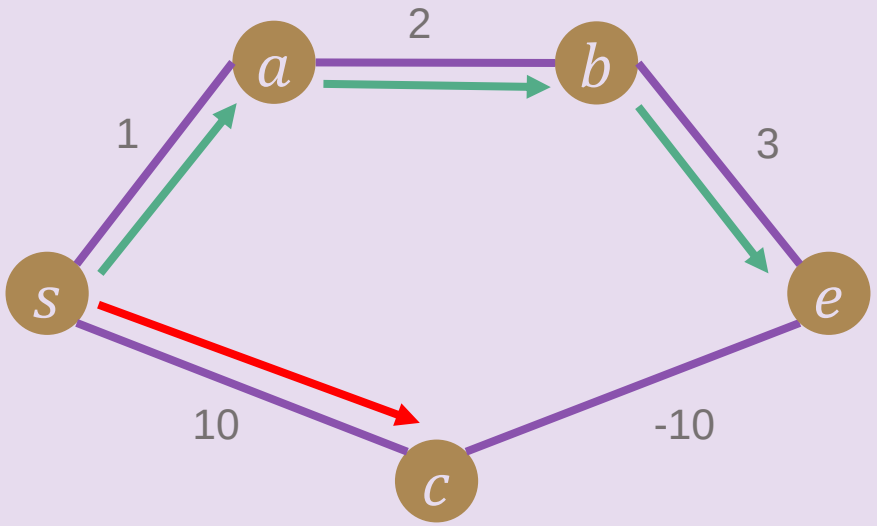
The algorithm ends when the destination vertex is reached or all vertices have been visited.



s	0	✓
a	7	✓
b	8	✓
c	9	✓
d	6	✓
e	4	✓
f	3	✓
g	1	✓
h	2	✓

Dijkstra's Algorithm

Dijkstra's Algorithm works on the idea that, since we visit vertices in order of their distance from the source vertex, it is impossible for us to find a shorter path to some vertex we have previously visited that visits the vertex we are currently visiting. Because of this, when we visit a vertex for the first time, we are guaranteed to have found a shortest path to it already. Note that because of this, Dijkstra's Algorithm does not work for graphs with negative edge weights.



Dijkstra's Algorithm Sample Implementation ($O(V^2 + E)$)

```
//N is the maximum possible number of vertices in the input.
//n is the number of vertices for that test case.
//In this sample, our source vertex is 0.
bool vis[N]; int dist[N]; vector<int> adj[N], adjw[N];

int main(){
    //read graph into adj, adjw
    //set vis[0]..vis[n-1] to false
    //set dist[1]..dist[n-1] to inf or -1 (sentinel value)
    dist[0] = 0;
    while(true){
        int next = -1;
        for(int i=0; i<n; i++){
            //add extra check if sentinel is -1
            if(!vis[i] && (next == -1 || dist[i] < dist[next]))
                next = i;
        }
        if(next == -1) break; //no more unvisited vertices
        vis[next] = true;
        for(int i=0; i<adj[next].size(); i++){
            if(vis[adj[next][i]]) continue;
            // or if dist[adj[next][i]] == -1 if sentinel is -1
            if(dist[next] + adjw[next][i] < dist[adj[next][i]]){
                dist[adj[next][i]] = dist[next] + adjw[next][i];
            }
        }
    }
    //dist[u] will contain the distance from 0 to u
}
```

```
//For constructing the path itself,
//we add a parent variable to each vertex.
//This acts like the "previous" vertex in the path
int parent[N];

//By default, the parents do not exist,
//so we set them to some sentinel value

//set parent[0]..parent[n-1] to -1

//Whenever we update the distance of a vertex,
//we know that its shortest path will contain that
//edge and the current vertex being processed is
//the previous vertex in that path.

parent[adj[next][i]] = next;

//Reconstruct the path by following each vertex's
//parent until we return to the source.

vector<int> path;
int cur = end;
while(cur != source){
    path.push_back(cur);
    cur = parent[cur];
}

//path will contain the actual path in reverse.
```

Dijkstra's Algorithm Sample Implementation ($O(V \log E + E)$)

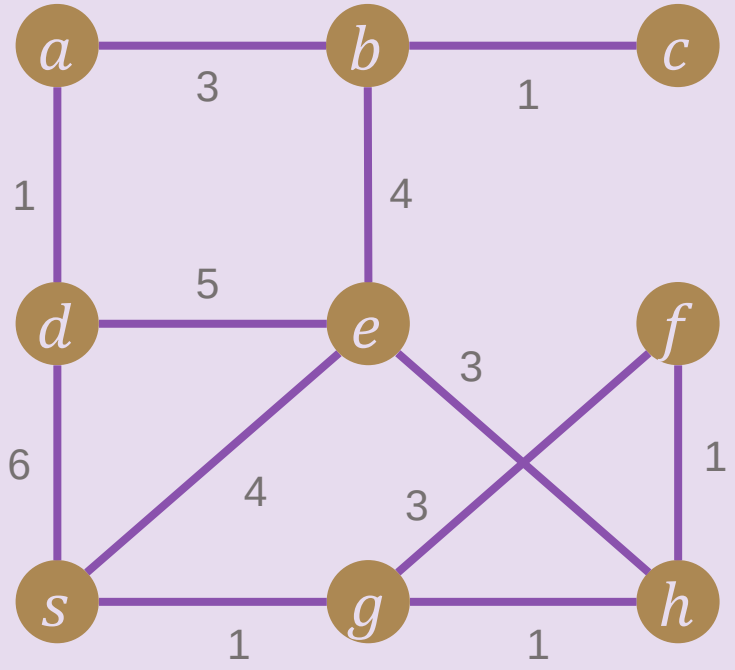
Dijkstra's Algorithm can be sped up by using a priority queue to find the next closest vertex to the source.

```
//define a new comparator for the priority queue
struct cmp{
    bool operator()(int a, int b){
        return dist[a] > dist[b]; //get the smallest distance first
    }
};

//after setting all the starting values
priority_queue<int, vector<int>, cmp> pq;
pq.push(0); //we start with the source vertex
while(pq.size() > 0){ //instead of while(true), we only need to check if the priority queue is nonempty
    int next = pq.top(); pq.pop(); //instead of searching, we can just get the next element in pq
    if(vis[next]) continue; //the same vertices will appear multiple times
    vis[next] = true;

    //process as before but push the new vertices into the priority queue
    for(int i=0; i<adj[next].size(); i++){
        if(vis[adj[next][i]]) continue;
        if(dist[next] + adjw[next][i] < dist[adj[next][i]]){
            dist[adj[next][i]] = dist[next] + adjw[next][i];
            pq.push(adj[next][i]);
        }
    }
}
```

Bellman-Ford Algorithm

The Bellman-Ford Algorithm also finds the shortest path from some given vertex to all other vertices. It is slower than Dijkstra's algorithm, making it less commonly used. However, it covers graphs with negative weight edges.

Like Dijkstra's Algorithm, the Bellman-Ford Algorithm also begins with the single source vertex s having a known distance of 0 from itself and all other vertices having an unknown distance, usually labeled infinity, from s.

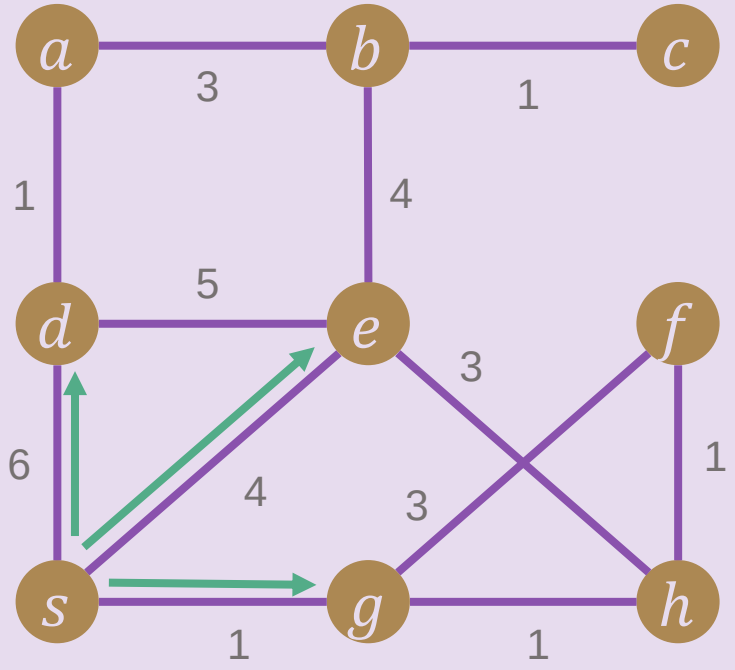


s	0
a	∞
b	∞
c	∞
d	∞
e	∞
f	∞
g	∞
h	∞

Bellman-Ford Algorithm

The Bellman-Ford Algorithm also finds the shortest path from some given vertex to all other vertices. It is slower than Dijkstra's algorithm, making it less commonly used. However, it covers graphs with negative weight edges.

It then iterates through every edge in the graph to determine if the known distance from s to the adjacent node can be "relaxed" or reduced.

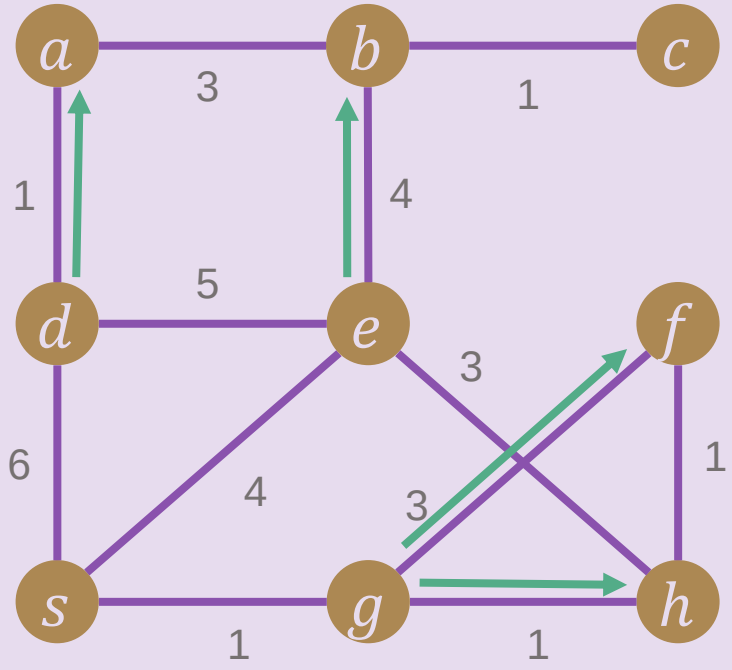


s	0	0
a	∞	∞
b	∞	∞
c	∞	∞
d	∞	6
e	∞	4
f	∞	∞
g	∞	1
h	∞	∞

Bellman-Ford Algorithm

The Bellman-Ford Algorithm also finds the shortest path from some given vertex to all other vertices. It is slower than Dijkstra's algorithm, making it less commonly used. However, it covers graphs with negative weight edges.

This is repeated multiple times. During each iteration, vertices have their distances from s reduced, and so are able to relax the vertices adjacent to them on the succeeding iterations.

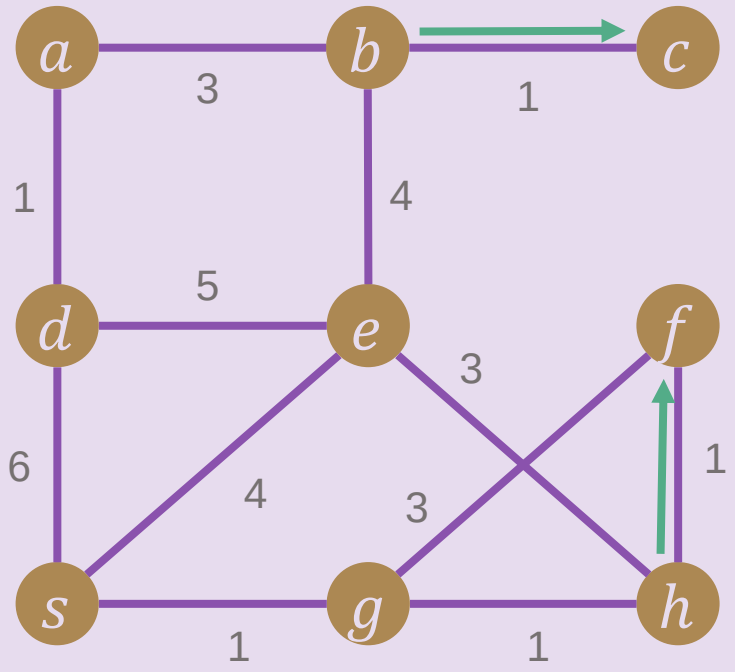


s	0	0
a	∞	7
b	∞	8
c	∞	∞
d	6	6
e	4	4
f	∞	4
g	1	1
h	∞	2

Bellman-Ford Algorithm

The Bellman-Ford Algorithm also finds the shortest path from some given vertex to all other vertices. It is slower than Dijkstra's algorithm, making it less commonly used. However, it covers graphs with negative weight edges.

If there are no negative weight cycles (which means no vertex will have a distance of $-\infty$), this will continue until an iteration where no more distances can relaxed. Here, the shortest paths for each vertex has been found.



s	0	0
a	7	7
b	8	8
c	∞	9
d	6	6
e	4	4
f	4	3
g	1	1
h	2	2

Bellman-Ford Algorithm

The Bellman-Ford Algorithm works on a similar idea as Dijkstra's Algorithm. That is, the shortest path to some vertex can be found by repeatedly finding better solutions until such a solution can no longer be found. However, Dijkstra's algorithm assumes that adding a new edge to a path can only increase its length, while the Bellman-Ford Algorithm makes no assumption and instead processes every vertex and every edge on each iteration. This allows it to handle negative edge weights.

Assuming the non-existence of negative weight cycles, it can be proven that the Bellman-Ford Algorithm will take no more than $|V| - 1$ iterations of relaxation, where $|V|$ is the number of vertices in the graph. This is because the shortest path from s to any other vertex can take no more than $|V| - 1$ jumps. The algorithm however, will continue indefinitely if a negative weight cycle exists. Because of this, if a relaxation still occurs on the $|V|$'th iteration, it is guaranteed that the graph has a negative weight cycle, and the algorithm can safely be terminated.

Bellman-Ford Algorithm Sample Implementation ($O(VE)$)

```
//N is the maximum possible number of vertices in the input.
//n is the number of vertices in that test case.
//E is the maximum possible number of edges in the input.
//e is the number of edges in that test case.
//In this sample, our source vertex is 0.
int dist[N], a[E], b[E], w[E];

int main(){
    //read graph into a, b, w
    //set dist[1]..dist[n-1] to inf or -1 (sentinel value)
    dist[0] = 0;

    //run the relaxation n times
    for(int i=0; i<n; i++){
        bool relaxed = false;
        for(int j=0; j<e; j++){
            if(dist[a[j]]+w[j] < dist[b[j]]){
                dist[b[j]] = dist[a[j]]+w[j];
                relaxed = true;
            }
            //repeat with reversed a, b for undirected graphs
        }
        if(!relaxed) break; //no more newly relaxed vertices
        else if(i == n-1){
            //negative weight cycle exists
        }
    }
    //dist[u] will contain the distance from 0 to u
}
```

```
//For constructing the path itself,
//we add a parent variable to each vertex.
//This acts like the "previous" vertex in the path
int parent[N];

//By default, the parents do not exist,
//so we set them to some sentinel value

//set parent[0]..parent[n-1] to -1

//Whenever we update the distance of a vertex,
//we know that its shortest path will contain that
//edge and the current vertex being processed is
//the previous vertex in that path.

parent[adj[next][i]] = next;

//Reconstruct the path by following each vertex's
//parent until we return to the source.

vector<int> path;
int cur = end;
while(cur != source){
    path.push_back(cur);
    cur = parent[cur];
}

//path will contain the actual path in reverse.
```

On Reconstructing Negative Weight Cycles in Directed Graphs

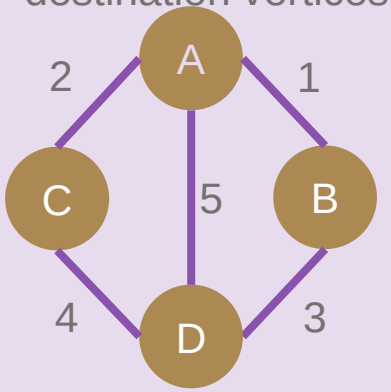
If a vertex u is relaxed during the nth iteration, it is necessarily part of some negative weight cycle. In undirected graphs, finding negative weight cycles is trivial because any negative weight edge necessarily forms a cycle with itself ($a \to b \to a$). However, this is not the case in directed graphs.

When using the method where we store the parents of each vertex, finding a cycle from u to itself may not necessarily work. The parent array will contain at least one negative weight cycle, but this is not guaranteed to be the negative weight cycle containing u. This is because parent entries may be rewritten for the same vertex multiple times by multiple different negative weight cycles in the same iteration.

When reconstructing a negative weight cycle, we have to check all the parent entries of all the vertices to find the cycle.

Floyd-Warshall Algorithm

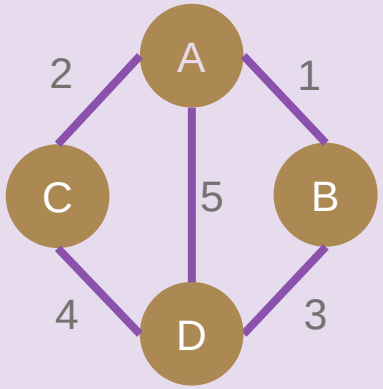
The **Floyd-Warshall Algorithm** solves the All-Pairs Shortest Path problem. In other words, it finds the shortest path between any two nodes in the graph. It does this by iterating through all vertices and checking if it can serve as an intermediate node to a shorter path between some other source and destination vertices.



	A	B	C	D
A	0	1	2	5
B	1	0	∞	3
C	2	∞	0	4
D	5	3	4	0

```
int dist[n][n];
for(int k=0; k<n; k++){
     for(int i=0; i<n; i++){
          for(int j=0; j<n; j++){
               if(dist[i][j] > dist[i][k] + dist[k][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
          }
     }
}
```

If the path passing through vertex k $(path(i,k) + path(k,j))$ is shorter than the direct path from i to j, then let us take the path through k instead of the direct one.



	A	B	C	D
A	0	1	2	4
B	1	0	3	3
C	2	3	0	4
D	4	3	4	0

Let $k = \mathrm{B}$, $i = A$, $j = D$. We know that $dist[A][B] = 1$ and $dist[B][D] = 3$. This means $dist[A][B] + dist[B][D] = 4$, which is shorter than $dist[A][D] = 5$. Thus the new value of $dist[A][D]$ is 4. We do this for all sources and destinations and then move on to the next intermediate node. When the algorithm ends, the adjacency matrix should contain the shortest path between any two nodes in the graph.

SSSP from each source

There are cases where the Floyd-Warshall Algorithm will not work for APSP. This is because either $O(V^3)$ is too slow or V^2 memory is too large.

To solve these problems, we can apply an SSSP algorithm from each starting vertex. Since there are V vertices in a given graph, Dijkstra's Algorithm will take $O(V^2 \log E + VE)$ time to complete. The actual time it takes for Dijkstra's Algorithm to complete will also be significantly reduced when the graph itself is sparse (doesn't have many edges) or disconnected, while the Floyd-Warshall Algorithm will take the same amount of time. It is also typically not necessary to find all pairs of shortest paths, but only a large number of them. If a vertex is never the starting vertex of any requested pair, then that vertex can be skipped. The same applies for the Bellman-Ford algorithm.

What's the point of using the Floyd-Warshall Algorithm then? It is much easier and faster to code than iterating Dijkstra's Algorithm V times and can save a lot of time in contests. Dijkstra's Algorithm also becomes very slow for very dense graphs (especially complete graphs) as E approaches V^2. The Floyd-Warshall Algorithm becomes faster than Dijkstra's Algorithm for this case.

Notes on Minimum Cost Spanning Tree Problems

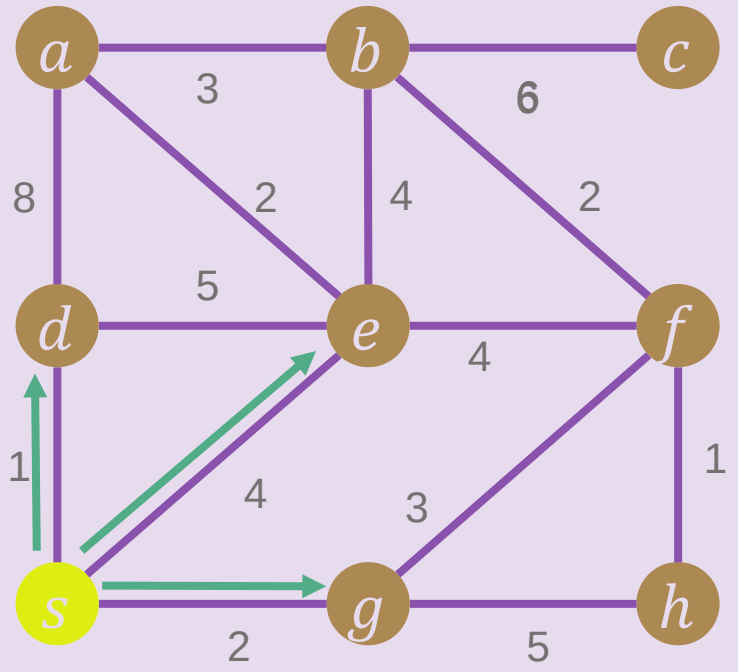
Minimum cost spanning tree problems always use undirected graphs. If used with directed graphs, the problem is called the minimum cost arborescence problem and requires a different algorithm to solve.

A variant of this problem, the minimum cost spanning forest problem, requires you to create multiple trees instead of one single tree. Both algorithms to be discussed can be used to solve these problems.

Prim's Algorithm

Prim's Algorithm finds a minimum cost spanning tree of a graph.

Prim's Algorithm begins with the single source vertex s being the only vertex in the spanning tree and determines the cost of adding each vertex u adjacent to s to the tree if we take the edge (s, u).

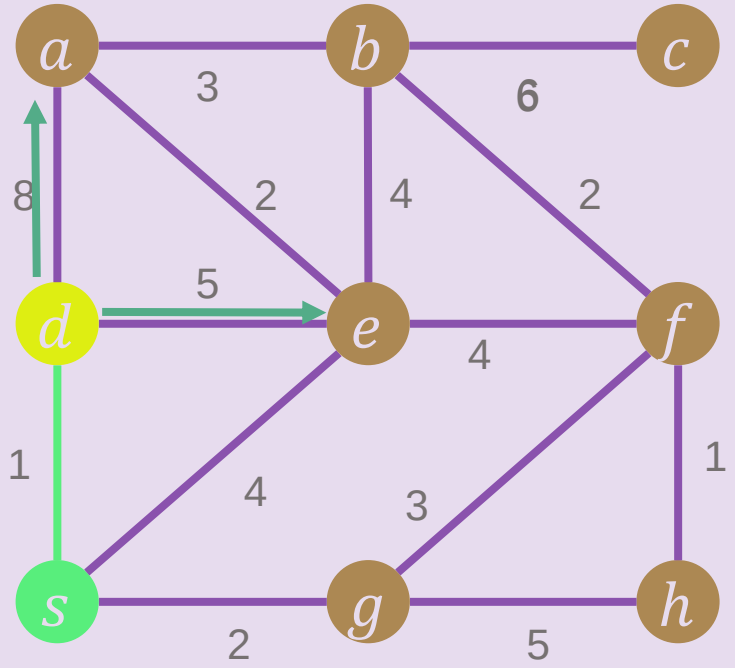


s	0	✓
a	∞	
b	∞	
c	∞	
d	1	
e	4	
f	∞	
g	2	
h	∞	

Prim's Algorithm

Prim's Algorithm finds a minimum cost spanning tree of a graph.

It takes the cheapest of these vertices and adds it to the spanning tree. It then, again, determines the cost of adding each vertex adjacent to the current vertex being added, always keeping track of the edge used in the cheapest way.

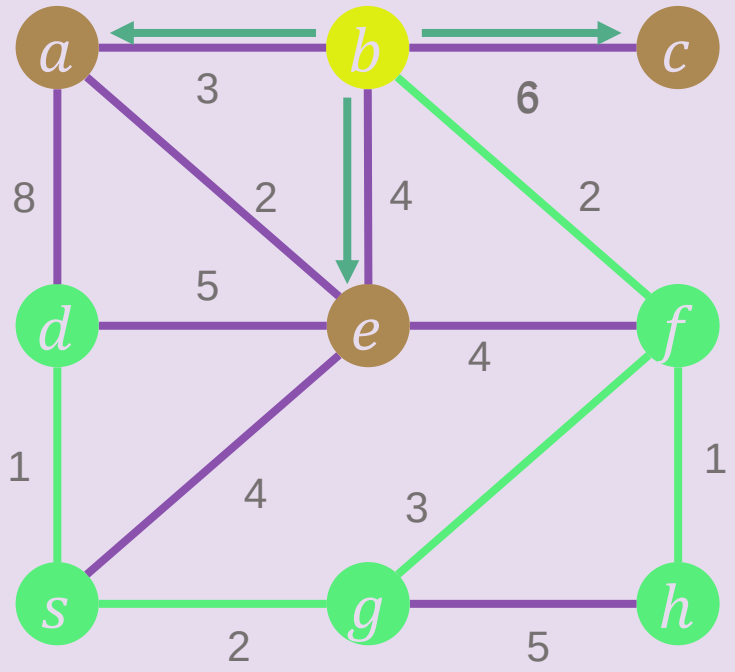


s	0	✓
a	8	
b	∞	
c	∞	
d	1	✓
e	4	
f	∞	
g	2	
h	∞	

Prim's Algorithm

Prim's Algorithm finds a minimum cost spanning tree of a graph.

This is repeated until all vertices have been added to the spanning tree.

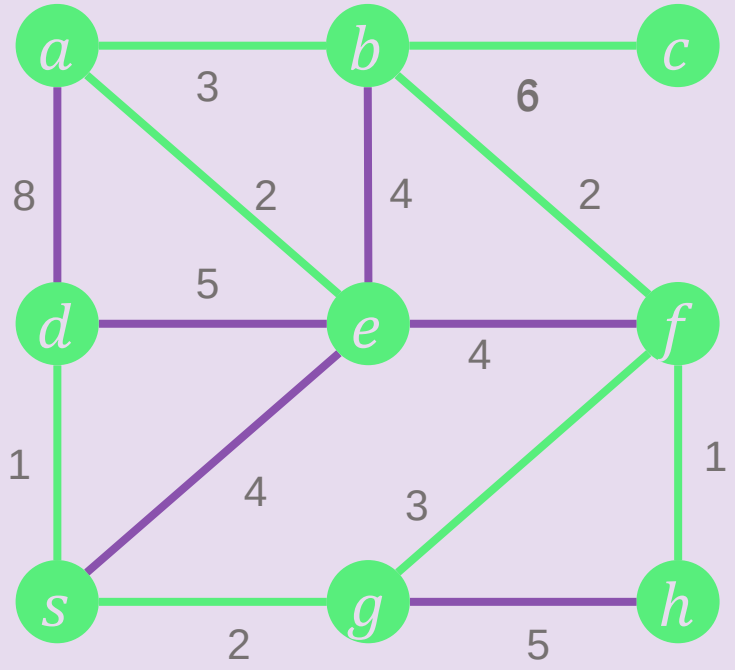


s	0	✓
a	3	
b	2	✓
c	6	
d	1	✓
e	4	
f	3	✓
g	2	✓
h	1	✓

Prim's Algorithm

Prim's Algorithm finds a minimum cost spanning tree of a graph.

Once all vertices have been added to the spanning tree. All edges in the minimum cost spanning tree of the graph, and the minimum cost spanning tree itself, have been found.



s	0	✓
a	3	✓
b	2	✓
c	6	✓
d	1	✓
e	4	✓
f	3	✓
g	2	✓
h	1	✓

Prim's Algorithm

Prim's Algorithm works very similarly to Dijkstra's Algorithm. In fact, the implementation is the same except we set the "distances" in Dijkstra's Algorithm to edge length instead of distance + edge length. This is because Prim's Algorithm works with the same idea – that selecting the next cheapest vertex and expanding from there will always give an optimal result.

Prim's Algorithm will still work on graphs with negative edge weights because, unlike Dijkstra's Algorithm, Prim's Algorithm only needs to take individual edges, instead of entire paths, into account.

Prim's Algorithm Sample Implementation ($O(V^2 + E)$)

```
//N is the maximum possible number of vertices in the input.
//n is the number of vertices for that test case.
//In this sample, our source vertex is 0.
bool vis[N]; int cost[N]; vector<int> adj[N], adjw[N];

int main(){
    //read graph into adj, adjw
    //set vis[0]..vis[n-1] to false
    //set cost[1]..cost[n-1] to inf or -1 (sentinel value)
    cost[0] = 0;
    int total = 0;
    while(true){
        int next = -1;
        for(int i=0; i<n; i++){
            //add extra check if sentinel is -1
            if(!vis[i] && (next == -1 || cost[i] < cost[next]))
                next = i;
        }
        if(next == -1) break; //no more unvisited vertices
        vis[next] = true;
        total += cost[next];
        for(int i=0; i<adj[next].size(); i++){
            if(vis[adj[next][i]]) continue;
            // or if cost[adj[next][i]] == -1 if sentinel is -1
            if(adjw[next][i] < cost[adj[next][i]]){
                cost[adj[next][i]] = adjw[next][i];
            }
        }
    } //total will contain the cost of the MCST
}
```

```
//For constructing the minimum cost spanning tree
//itself, we keep track of which edges are used to
//add vertices to the MCST.
vector<int> adjid[N]; //give edges ids
int edge[N]; //id of the edge used

//This may be implemented many ways.
//Giving edges ids is just one.

//set edge[0]..edge[n-1] to -1

//Whenever we update the cost of a vertex, we use
//the edge currently being processed.
cost[adj[next][i]] = adjid[next][i];

//Reconstruct the minimum cost spanning tree by
//finding all edges used. Most of the time, the
//problem will only ask for the ids of the edges
//used. In this case, we can simply print the ids
//stored in edge[1]..edge[n-1].

//Sometimes the problem asks for the actual edges
//(the vertices and the weight) or other
//information attached to these edges. In these
//cases, it could be easier to store the edges in
//an additional edge list or store them as objects.
```

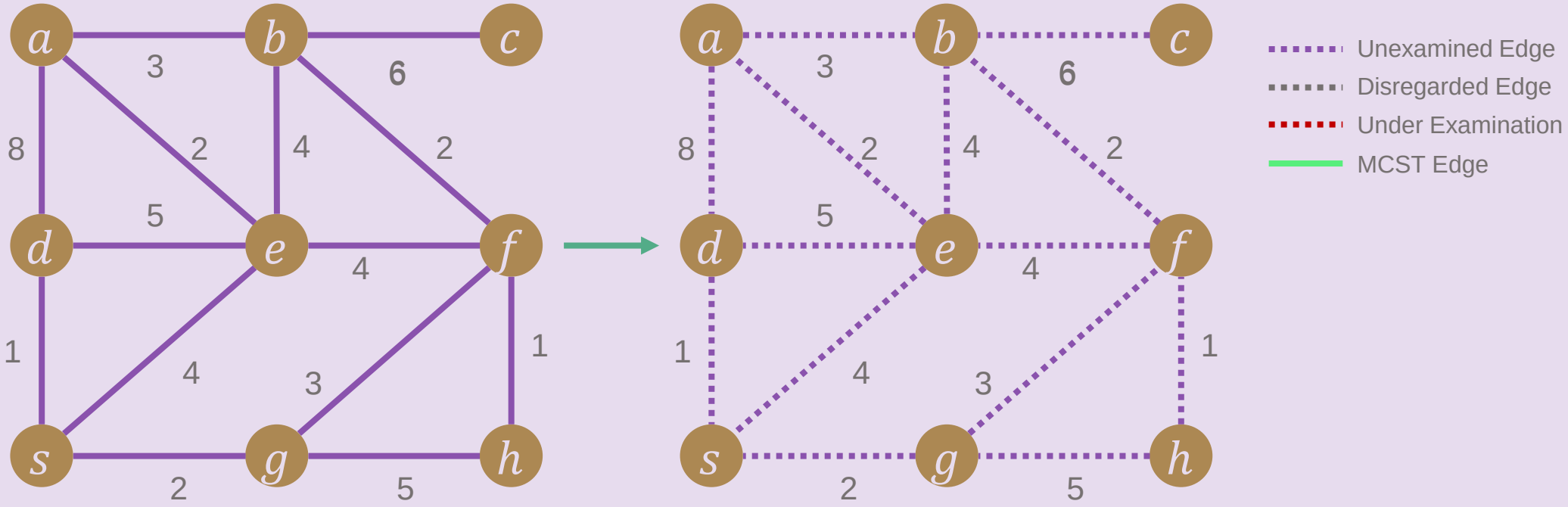
Prim's Algorithm Sample Implementation ($O(V \log E + E)$)

Like Dijkstra's Algorithm, Prim's Algorithm can be sped up by using a priority queue to find the next cheapest vertex to add.

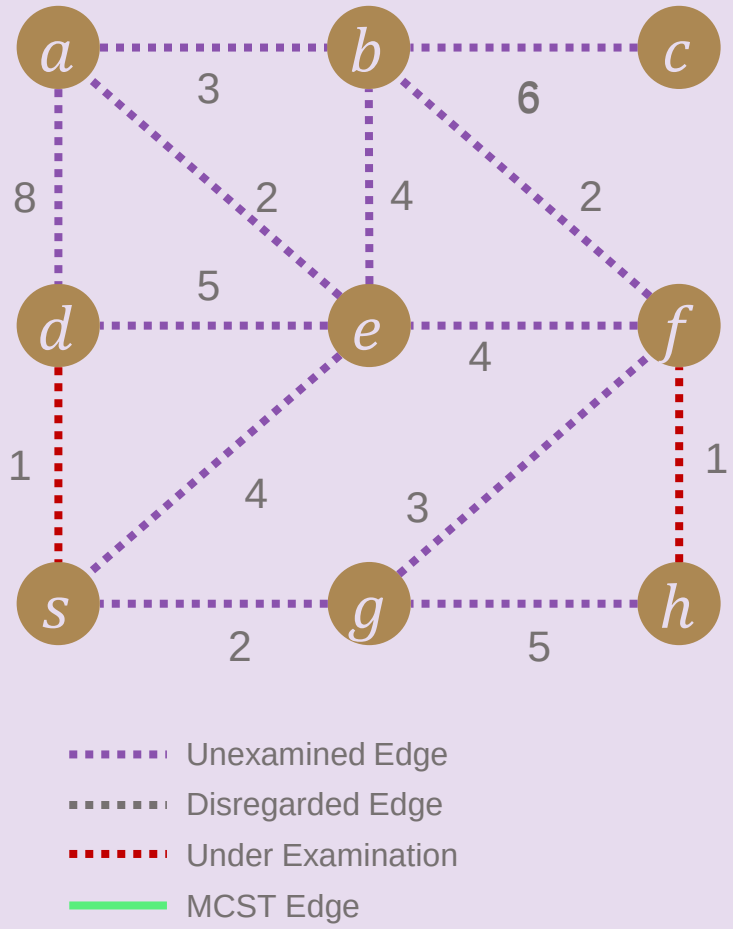
```
//define a new comparator for the priority queue
struct cmp{
    bool operator()(int a, int b){
        return cost[a] > cost[b]; //get the smallest cost first
    }
};

//after setting all the starting values
priority_queue<int, vector<int>, cmp> pq;
pq.push(0); //we start with the source vertex
while(pq.size() > 0){ //instead of while(true), we only need to check if the priority queue is nonempty
    int next = pq.top(); pq.pop(); //instead of searching, we can just get the next element in pq
    if(vis[next]) continue; //the same vertices will appear multiple times
    vis[next] = true;

    //process as before but push the new vertices into the priority queue
    for(int i=0; i<adj[next].size(); i++){
        if(vis[adj[next][i]]) continue;
        if(adjw[next][i] < cost[adj[next][i]]){
            cost[adj[next][i]] = adjw[next][i];
            pq.push(adj[next][i]);
        }
    }
}
```

Kruskal's Algorithm

Kruskal's Algorithm is another method for finding the minimum cost spanning tree given a graph. The concept is simple: we start with the the same input graph, but without any of the edges. We go through each of the edges, starting with the one with least weight. If adding the edge to the graph forms a cycle, we disregard it, otherwise we add it to our MCST. We do this until we have gone through all the edges or we have added $|v| - 1$ edges to our graph, forming the MCST.



Kruskal's Algorithm

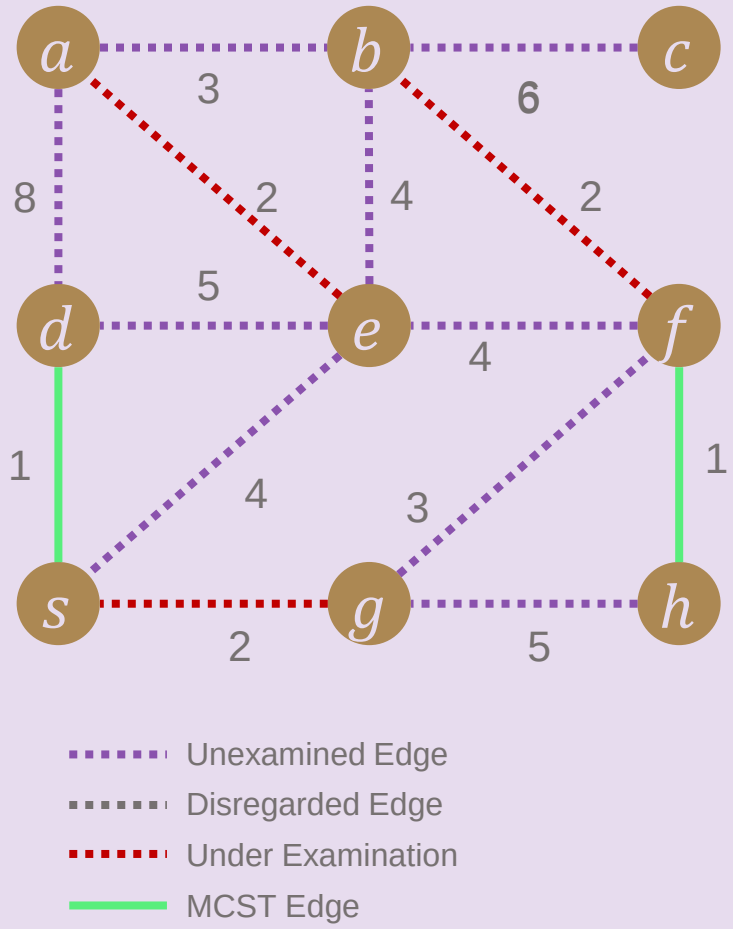


Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

First, edges are sorted in ascending order. Implementation wise, this sorting is usually done implicitly by placing all edges in the graph in a priority queue. We then poll from the queue each time we examine a new edge, guaranteeing that we go through edges with the lowest weight first.

The edges with the lowest weight in the graph are $\{f, h\}$ and $\{d, s\}$. Neither of these edges form a cycle when added to our graph.

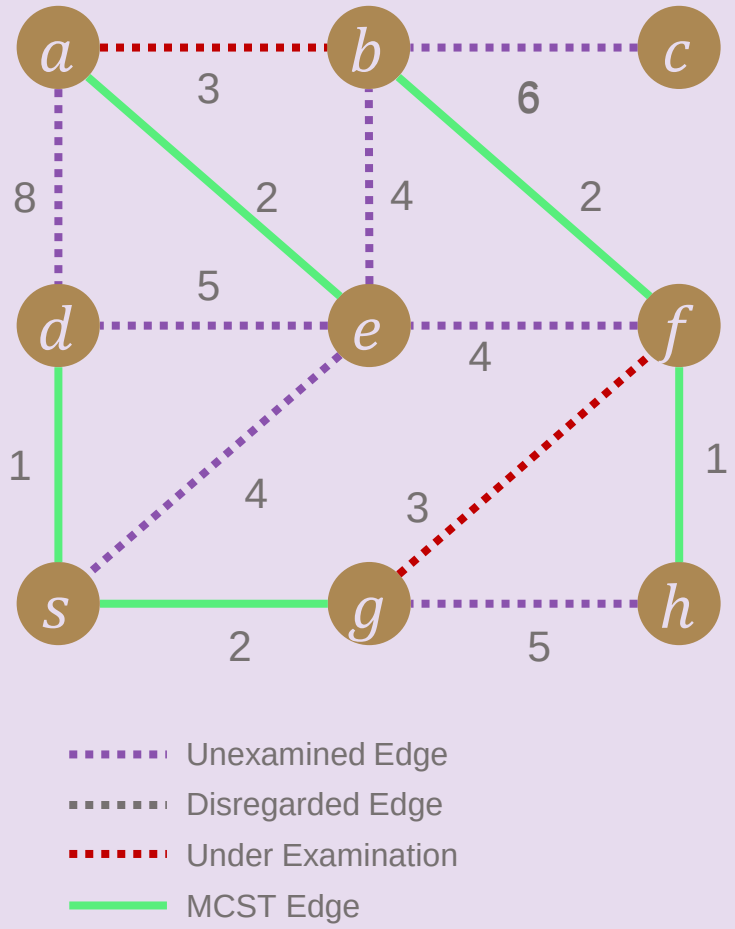
Kruskal's Algorithm



Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

We then proceed to examine more edges. $\{b, f\}$, $\{a, e\}$ and $\{s, g\}$ all have a weight of 2. None of these edges produce a cycle when added to the graph.

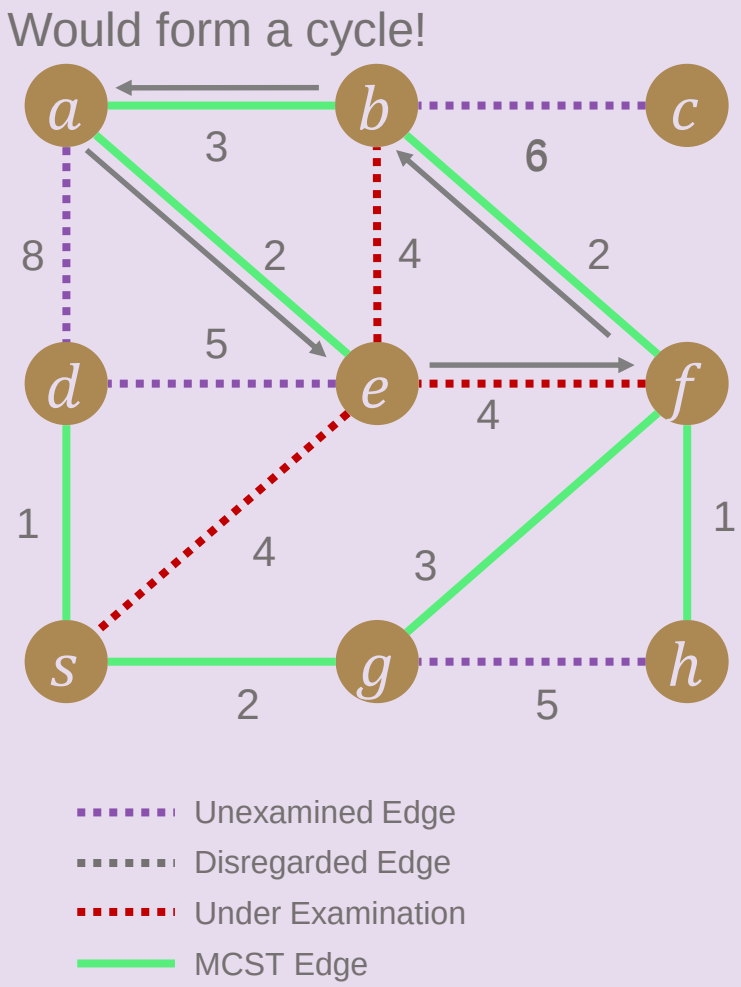
Kruskal's Algorithm



Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

$\{a, b\}$ and $\{g, f\}$ both have a weight of 3. Adding both these edges to the graph, there are still no cycles formed, so they are part of our MCST.

Kruskal's Algorithm



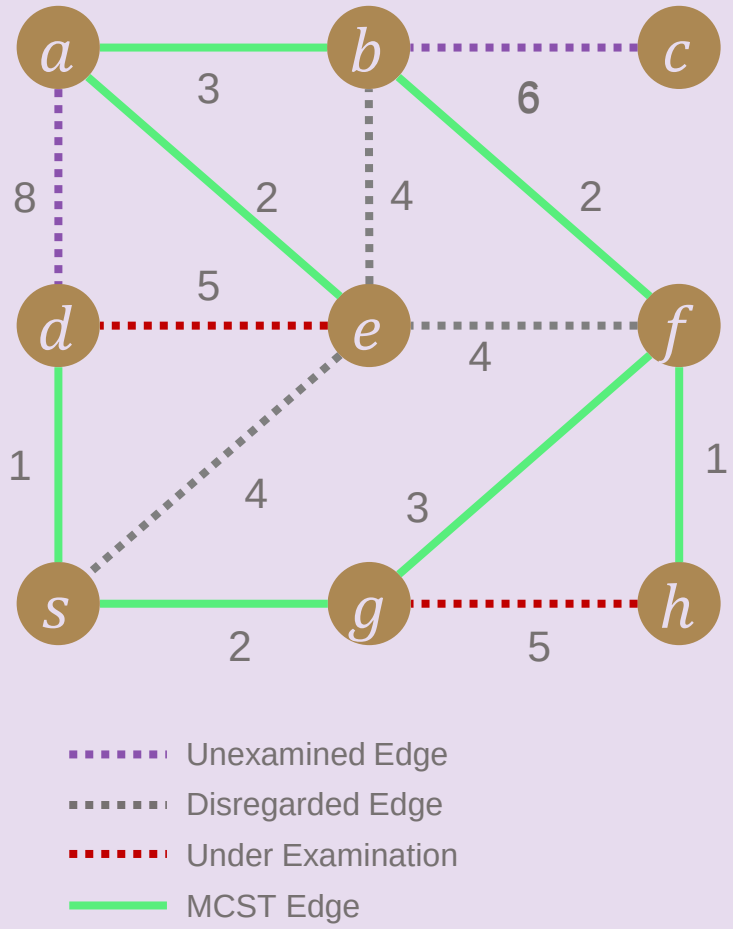
Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

$\{e, f\}$, $\{e, s\}$ and $\{b, e\}$ all have weights of 4. We first look at the edge $\{e, f\}$. Notice that if we add it to the graph, we form the cycle $a \to e \to f \to b \to a$. Thus we should disregard the edge $\{e, f\}$.

Similarly, adding the edge $\{e, s\}$ would form the cycle $a \to b \to f \to g \to s \to e \to a$, thus we should disregard it.

Adding the edge $\{b, e\}$ would also form a cycle, $a \to e \to b \to a$, thus we also disregard the edge $\{b, e\}$.

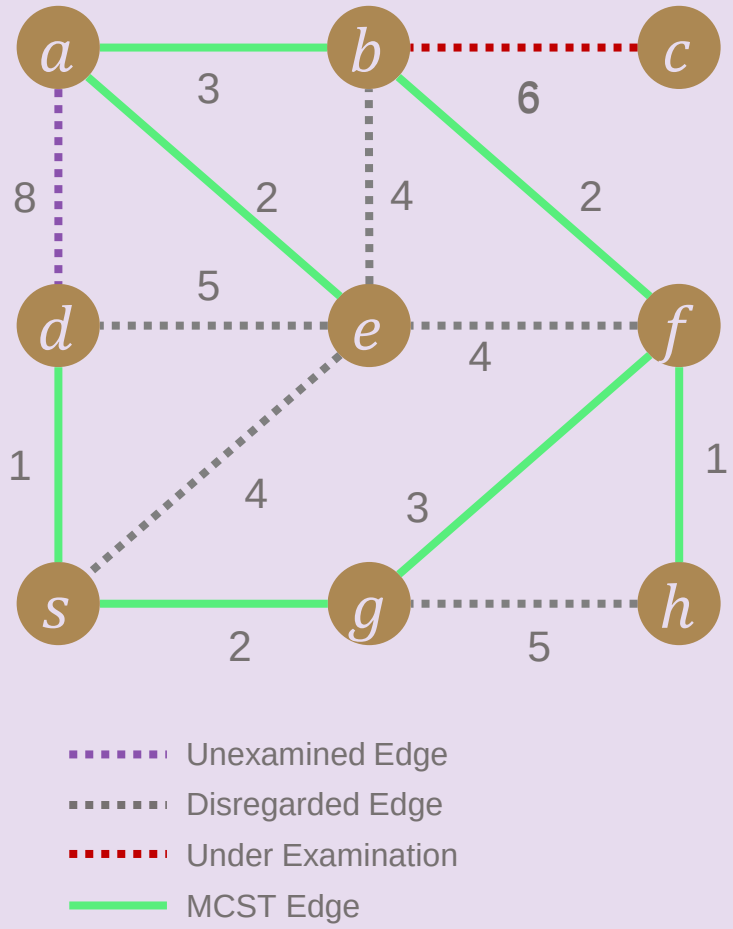
Kruskal's Algorithm



Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

The edges $\{d, e\}$ and $\{g, h\}$ both have a weight of 5. By inspection, we can see that adding either edge to the graph will result in the formation of cycles. We should thus disregard both edges.

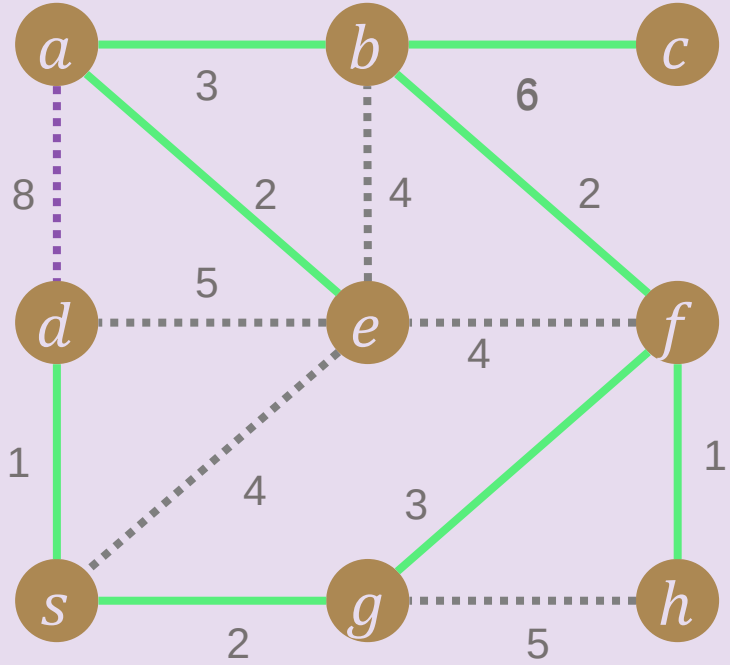
Kruskal's Algorithm



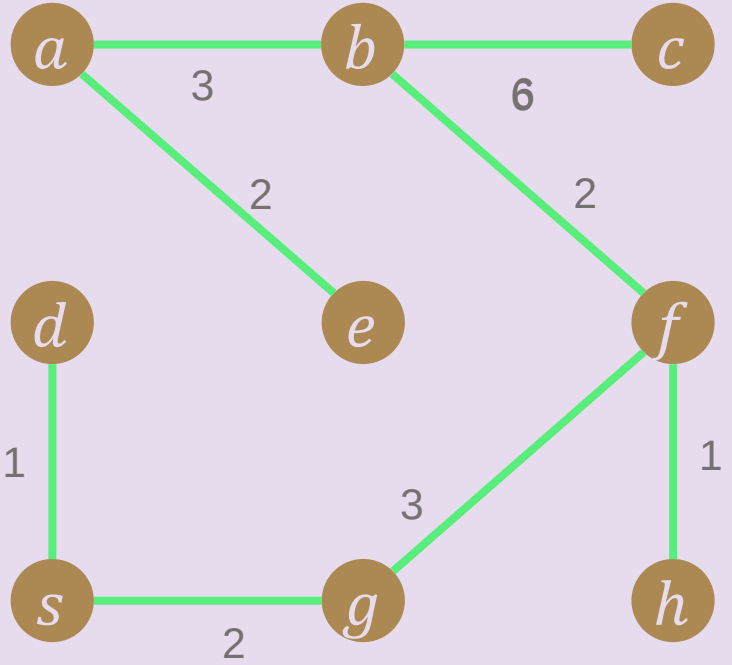
Vertex A	Vertex B	Weight
f	h	1
d	s	1
b	f	2
a	e	2
s	g	2
a	b	3
g	f	3
e	f	4
e	s	4
b	e	4
d	e	5
g	h	5
b	c	6
a	d	8

The next edge is $\{b, c\}$ with a weight of 6. Adding this to our graph does not form any cycles, thus we add it to our MCST. Note that after adding $\{b, c\}$, we have added $|v| - 1$ edges to our graph. This means we may now end the algorithm.

Kruskal's Algorithm



Unexamined Edge
Disregarded Edge
Under Examination
MCST Edge



Now that the algorithm has ended, the edges that were added to the graph will form our final MCST.

Kruskal's Algorithm Sample Implementation

```
//N is the maximum possible number of vertices in the input.
//n is the number of vertices for that test case.

struct edge{
    int u, v, w; //the two vertices and the weight
    edge(int u, int v, int w){
        this->u = u; this->v = v; this->w = w;
    }
};
struct cmp{
    bool operator()(const edge &a, const edge &b){
        return a->w > b->w; //priority queue in C++ is max heap
    }
};

int main(){
    priority_queue<edge, vector<edge>, cmp> pq;
    //insert all edges into pq
    int total = 0;
    int need = n-1;
    while(need > 0){
        edge cur = pq.top(); pq.pop();
        if(!formsCycle(cur)){
            //add cur to graph
            total += cur.w;
            need--;
        }
    }
    //total will contain the cost of the MCST
}
```

Kruskal's Algorithm

In order for Kruskal's Algorithm to work, we need a way to determine if adding an edge (u, v) to the current graph creates a cycle. Since the graph is undirected, this is the same as checking whether u and v are already connected and can be easily done using a Breadth-First or Depth-First Search. That however would take $O(E)$ time for each edge, making Kruskal's Algorithm run in $O(E^2 + E \log E)$ time, which is much slower than Prim's Algorithm.

There is a much faster way for us to determine whether two vertices are connected while constructing the MCST using Kruskal's Algorithm. This is called Union-Find or Disjoint Set Union.

Union-Find / Disjoint Set Union

Union-Find or Disjoint Set Union is a method to determine which items from a number of mutually disjoint sets (each item is in exactly one set) are part of the same set. It has two operations:

- Find – Determine the set an element is part of.
- Union – Combine two sets into one.

This is done by assigning a representative element to each set. Find then returns this representative element. To check whether two elements are part of the same set, we just check whether their representative elements are the same.

This is done by creating a tree using the elements with the root as the representative element. Union then simply sets the parent of one root as the other root, effectively combining the two trees.

In Kruskal's Algorithm, we assign each connected subgraph to a set and the vertices as the elements.

Kruskal's Algorithm Sample Implementation with Union-Find

```
//N is the maximum possible number of vertices in the input.
//n is the number of vertices for that test case.

struct edge{
    int u, v, w; //the two vertices and the weight
    edge(int u, int v, int w){
        this->u = u; this->v = v; this->w = w;
    }
};
struct cmp{
    bool operator()(const edge &a, const edge &b){
        return a->w > b->w; //priority queue in C++ is max heap
    }
};

int main(){
    priority_queue<edge, vector<edge>, cmp> pq;
    //insert all edges into pq
    int total = 0;
    int need = n-1;
    while(need > 0){
        edge cur = pq.top(); pq.pop();
        if(!formsCycle(cur)){
            //add cur to graph
            total += cur.w;
            need--;
        }
    }
    //total will contain the cost of the MCST
}
```

```
//Keep track of the parent of each vertex
int par[N];

//Follow the parent of the vertex being checked
//until you reach the root.
int find(int u){
    if(par[u] == u) return u;
    return find(par[u]);
}

//Set the parent of one root to the other. Merge
//is used here because union is a reserved keyword.
void merge(int a, int b){
    par[find(a)] = find(b);
}
```

```
//Since no vertices are connected at the start, each
//is in its own tree.

//set par[0]..par[n-1] to 0..n-1 (par[i] = i)
```

```
//formsCycle(cur) is changed to checking whether
//cur.u and cur.v are in the same tree.

if(find(cur.u) != find(cur.v))
```

```
//Adding cur to the graph is just taking the union.
merge(cur.u, cur.v);
```

Union-Find Optimizations

Find takes $O(V)$ time, while Union takes $O(V)$ time if we use Find again or $O(1)$ time if we keep track of what Find returned when we first checked the parents of u and v. This makes Kruskal's Algorithm run in $O(VE + E \log E)$ time.

Find can be further optimized by replacing the parent of each vertex passed in the recursive function with the representative element. This allows the vertex to go straight to its tree's root instead of having to go through each parent first.

```
int find(int u){
    if(par[u] == u) return u;
    return par[u] = find(par[u]); //reassign before returning
}
```

Union-Find Optimizations

Union can also be optimized by making the tree with the larger depth the new root instead of using either of the two roots. This reduces the height of each tree, reducing the number of function calls Find has to go through to reach the root of a set.

```
int depth[N];

int union(int a, int b){
    int roota = find(a);
    int rootb = find(b);

    if(depth[roota] < depth[rootb]){
        par[roota] = rootb;
    }else if(depth[rootb] < depth[roota]){
        par[rootb] = roota;
    }else{
        par[rootb] = roota;
        depth[roota]++;
    }
}

int main(){
    //set depth[0]..depth[n-1] to 0 before Kruskal's Algorithm
}
```

Union-Find Optimizations

It can be proven that using both of these optimizations reduces the running time of Find to $O(\alpha^{-1}(V))$ where $\alpha^{-1}(n)$ is the inverse Ackermann function (proof outside of scope), an extremely slowly growing function that, for most practical values of n, is less than 5. This effectively reduces the complexity of Union-Find to a small constant.

Using these optimizations, the complexity of Kruskal's Algorithm reduces to $O(E \log E)$.

On Prim's and Kruskal's Algorithms

For most cases, Prim's and Kruskal's algorithms are effectively the same in terms of running time. Use which one you are more comfortable with.

Any variant of the typical MCST problem that can be covered by one of these algorithms can be covered by the other as well. However, the modifications necessary to solve the problem may be more complicated for one of them, so it is still suggested to be familiarized with both algorithms.

Basic Implementation Problems for Practice (Optional)

- CodeForces 20C – Dijkstra?
- UVa 558 – Wormholes
- UVa 821 – Page Hopping
- UVa 11631 – Dark Roads
- UVa 10147 – Highways

Problems (Required)

- UVa 1235 – Anti Brute Force Lock
- UVa 11733 – Airports
- UVa 10600 – ACM Contest and Blackout
- UVa 10557 – XYZZY
- UVa 1250 – Robot Challenge
- CodeForces 229B – Planets
- CodeForces 329B – Biridian Forest
- CodeForces 676D – Theseus and Labyrinth

Challenges (At least 3 required)

- UVa 1202 – Finding Nemo
- UVa 1253 – Infected Land
- UVa 11329 – Curious Fleas
- CodeForces 295B – Greg and Graph
- CodeForces 295C – Greg and Friends
- CodeForces 472D – Design Tutorial: Inverse the Problem

- For those who have not gotten full points in “The Cheapest Reid” from NOI 2017 eliminations, we encourage you to try it again. This does not count towards the 3 problems.

IOI Training Week 7
Advanced Data Structures

Tim Dumol

Contents

1 Range Minimum Query

This section rotates around a common problem:

Definition 1 (Range Minimum Query (RMQ))**.** Given an integer array of fixed length, answer a set of queries of the form: "What is the minimum element in the range of the array from `i` to `j`?". The contents of the array may change between queries.

The naïve solution for RMQ has no setup time, and $O(n)$ query time. We can improve on this by adding some setup time, and using some additional memory. We will discuss two approaches: square-root decomposition, and segment trees.

1.1 Square-root (sqrt) Decomposition

The idea behind sqrt decomposition is simple: preprocess the array into $\sqrt{n}$ chunks of size $\sqrt{n}$ each (thus consuming $O(\sqrt{n})$ extra memory), so that we can perform the query in $O(\sqrt{n})$ time, by using the preprocessed chunks to compute the minimum for the parts of the range that have a full intersection with the chunks, and then traversing the remaining at most $2\left(\sqrt{n}-1\right)$ elements uncovered by the chunks[1]. To elaborate, in code[2]:

```
struct SqrtDecomp {
  vector<int>* arr;
  vector<int> chunks;
  int chunk_size;
  int n_chunks;
  SqrtDecomp(vector<int> const *arr) : arr(arr) {
    chunk_size = (int)sqrt(arr->size());
    n_chunks = (int)ceil(arr->size()/(double)n_chunks);
    chunks.resize(n_chunks);
```

[1] One can see that this can be extended to any associative operation.
[2] code is untested, if it's wrong, feel free to correct

```
    for (int i = 0; i < n_chunks; ++i) {
      // cap computed in advance to avoid recomputing
      const int cap = min(n_chunks * (i+1), arr->size());
      // assumption: all input values less than 1 << 30
      chunks[i] = 1 << 30;
      for (int j = i*chunk_size; j < cap; ++j) {
        chunks[i] = min(arr[j], chunks[i]);
      }
    }
  }

  // end is exclusive
  int query(int begin, int end) {
    int left = (int)ceil(begin/(double)chunk_size);
    int right = (int)floor(end/(double)chunk_size);
    int ans = 1 << 30;
    if (left <= right) {
      ans = min(ans, min_element(chunks.begin() + left, chunks.begin() + right));
    }
    if (start % chunk_size != 0) {
      ans = min(ans, min_element(arr->begin() + begin, arr->begin() + ↙
          ↳ chunk_size*left));
    }
    if (end % chunk_size != 0) {
      ans = min(ans, min_element(arr->begin() + chunk_size*right, arr->begin() + ↙
          ↳ end));
    }
    return ans;
  }
};
```

The code to update the sqrt decomposition is an exercise left to the reader (you don't need to submit it).

1.2 Segment Trees

But a query time $O(\sqrt{n})$ is still pretty slow. Can we do faster? The answer is yes. We can get $O(\log(n))$ query time with $O(n)$ extra memory, by using a segment tree.

Definition 2 (Segment Tree)**.** A segment tree over an array of length n (for simplicity, let's say it's a power of two – extending to a non-power of two is an exercise for the reader) is a balanced binary[3] tree with n leaves, each corresponding to an element in the array. Each internal vertex of the segment tree has a value corresponding to the minimum of all vertices under its subtree.

Since a segment tree is a complete binary tree, it can be represented similarly as a heap, using only a single-dimensional integer array. Furthermore, a segment tree has $2n-1$ vertices, and can be constructed in $O(n)$ time. For querying, we recursively traverse the segment tree, stopping when the segment covered by a vertex is wholly included in the query range. It can be shown that this results in $O(\log(n))$ time[4].

```
struct ST {
  typedef int el_type;
  vector<el_type> tree;
  int n;
  ST(vector<el_type> const& arr) {
    n = arr.size();
    tree.resize(2*arr.size());
```

[3] technically you can use any arity, but for simplicity let's say binary

[4] https://cs.stackexchange.com/questions/37669/time-complexity-proof-for-segment-tree-implementation-of-the-ranged-sum-proble

```
    build(0, arr, 0, arr.size());
  }

  inline int left(int idx) const { return 2*idx + 1; }
  inline int right(int idx) const { return 2*idx + 2; }

  // end is exclusivel, as usual
  int build(int idx, vector<el_type> const& arr, int start, int end) {
    if (start + 1 == end) {
      return (tree[idx] = arr[start]);
    }
    const int mid = (start + end)/2;
    return (tree[idx] = min(build(left(idx), arr, start, mid), build(right(idx), ↙
        ↳ arr, mid, end)));
  }

  int query(int start, int end) {
    return query(start, end, 0, 0, n);
  }

  int query(int start, int end, int idx, int tree_start, int tree_end) {
    if (start <= tree_start && tree_end <= end) {
      return tree[idx];
    } else {
      const int mid = (tree_start + tree_end)/2;
      return min(query(start, end, left(idx), tree_start, mid), query(start, ↙
          ↳ end, right(idx), mid, tree_end));
    }
  }
};
```

The code to update a single element is an exercise left to the reader (you don't need to submit it). Furthermore, through the usage of lazy propagation, you can make it so that you can update a range of elements in the segment tree (also an exercise left to the reader – hint: you need to only add a single piece of additional information, if your update is adding an integer to a given range).

Similar to sqrt decomposition, you can adapt the segment tree to any associative operation (e.g., range sum query).

1.3 Notes

Coincidentally, RMQ turns out to be deeply related to another (somewhat rarer, but also standard) problem, LCA:

Definition 3 (Lowest Common Ancestor (LCA))**.** Given a rooted tree, `T`, answer a set of queries of the form: "What is the vertex of `T` that is farthest from the root, that is an ancestor of both vertices `x` and `y`?".

The naïve solution for LCA has no setup time, and $O(n)$ query time, and it turns out you can preprocess LCA into an RMQ problem, and vice versa. You can also use square-root decomposition for LCA, although of a different form. To learn more about this (and asymptotically better solutions to RMQ), check out the RMQ and LCA tutorial on Topcoder (c.f. references).

2 Self-Balancing Binary Search Trees

Recall that most library implementations of the `map` and `set` datastructures use a self-balancing binary search tree (usually a Red-Black tree), allowing query, delete, and insert operations in amortized $O(\log(n))$ time. Usually, this is good enough for our purposes, but sometimes we need to augment each vertex with some additional information in order to enable a special kind of query not supported by library implementations.

For example, you may have a dynamically changing ordered list of integers, and have to find the index of an arbitrary integer in the list.

Now, the most commonly used trees in library code are AVL trees and Red-Black trees, because of their good asymptotic characteristics. However, they are a pain to code, and have a lot of edge cases. Thus, for competitive programming, we usually implement simpler trees: either scapegoat trees or treaps. In this discussion, we'll focus on treaps (because the author considers them easier to implement and understand)[5].

A treap is a binary search tree augmented with a randomized priority value on each of its vertices, and kept balanced by maintaining a heap (priority queue) structure on its vertices (i.e., each vertex should have a smaller priority than its children). Because a random heap is balanced, a treap is probabilistically balanced. The hard part in implementing a treap is in maintaining the heap structure, as one will have to rotate vertices that violate the heap structure, while maintaining the BST structure.

```
struct TreapNode {
  int start, end;
  int priority;
  TreapNode *kids[2];
  TreapNode *parent;
  TreapNode(){}
  void init(int start, int end, TreapNode *parent);
  void update_aug() {
    // whatever augmentation you need
  }
};

// we cache the objects so that we don't need to dynamically
// allocate objects in heap (this is also known as arena
// allocation)
TreapNode cache[3000024];
int cctr;
void TreapNode::init(int start, int end, TreapNode *parent) {
  kids[0] = kids[1] = NULL;
  this->parent = parent;
  this->start = start;
  this->end = end;
  priority = rand();
}

struct Treap {
  TreapNode *root;
  Treap() : root(NULL) {}
  // dir: 0 is left, 1 is right
  void rotate(TreapNode *node, int dir) {
    TreapNode *rkid = node->kids[dir^1];
    if (node->parent) {
      if (node == node->parent->kids[0]) {
        node->parent->kids[0] = rkid;
      } else {
        node->parent->kids[1] = rkid;
      }
    } else {
      root = rkid;
    }
    rkid->parent = node->parent;
    node->kids[dir^1] = rkid->kids[dir];
    if (node->kids[dir^1]) {
```

[5]But briefly: scapegoat trees remain balanced by maintaining some statistics on their subtrees, and if a subtree is sufficiently "imbalanced", it completely reconstructs that subtree. This rebalancing is done rarely enough that operations on a scapegoat tree remain $O(\log(n))$ time.

```
      node->kids[dir^1]->parent = node;
    }
    // finally, transplant orig to new parent
    rkid->kids[dir] = node;
    node->parent = rkid;

    // update augmentation
    node->update_aug();
    rkid->update_aug();
  }
  void add(int start, int end) {
    if (root == NULL) {
      root = &cache[cctr++];
      root->init(start, end, NULL);
      return;
    }
    // look for node to insert at
    TreapNode *p = root;
    TreapNode *node = NULL;
    while (true) {
      if (start < p->start) {
        if (!p->kids[0]) {
          node = p->kids[0] = &cache[cctr++];
          break;
        } else p = p->kids[0];
      } else {
        if (!p->kids[1]) {
          node = p->kids[1] = &cache[cctr++];
          break;
        } else p = p->kids[1];
      }
    }
    node->init(start, end, p);
    // maintain augmentation
    TreapNode *p2 = p;
    while (p2 != NULL) {
      p2->update_aug();
      p2 = p2->parent;
    }

    // rotate
    while (node->parent && node->priority > node->parent->priority) {
      if (node == node->parent->kids[0]) {
        rotate(node->parent, 1);
      } else {
        rotate(node->parent, 0);
      }
    }
  }
  int search(TreapNode *p, int start, int end) {
    // problem-dependent
  }
  int search(int start, int end) {
    return search(root, start, end);
  }
};
```

3 Bonus: More interesting data structures

This is just a listing of interesting data structures that you may want to look into:

- Binary Index Tree (BIT) (aka Fenwick Tree) https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/ – used to get prefix sums; functionality is a subset of Segment Tree, but has faster to type implementation
- k-d tree https://en.wikipedia.org/wiki/K-d_tree – allows you to query the nearest neighbor of a point among a dynamic set of points in k-dimensional space, in $O(\log(n))$ time. (out of IOI scope)
- Skip list https://en.wikipedia.org/wiki/Skip_list – randomized data structure with performance characteristics similar to a balanced binary search tree (very rare usage in competitive programming)

4 Problems

1. Reverse Prime (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2657)
2. Frequent Values (https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=2176)
3. Census (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2272)
4. Can you answer these queries VIII (http://www.spoj.com/problems/GSS8/)
5. Permutations (https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=2520)
6. Ahoy, Pirates! (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2397)
7. XOR on Segment (http://codeforces.com/problemset/problem/242/E)
8. Robotic Sort http://www.spoj.com/problems/CERC07S/

4.1 Bonus Problems

These problems are ungraded.

1. Alien Abduction (https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=559&page=show_problem&problem=4056)
2. Genetics (https://www.codechef.com/problems/GENETICS)
3. Little Elephant and Tree (http://codeforces.com/problemset/problem/258/E)

5 References

1. RMQ and LCA tutorial https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/

NOI.PH 2017 Training Week 8

Kevin Charles Atienza

April 2017

Contents

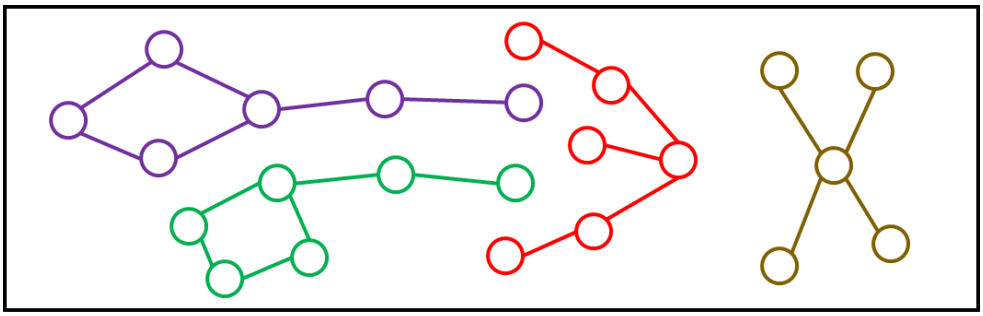
This week, we'll connect ourselves with various topics related to connectivity in graphs.

1 Connectivity in Undirected Graphs

It turns out that connectivity is easier to understand with undirected graphs. That's because the edges are *bidirectional*, meaning that if there's a path from a to b, then there's also a path from b to a.

A **connected component** is a maximal[1] set of nodes that is pairwise connected.

Here are the connected components in an example graph, where each connected component is colored by a single color:



connected components

More formally, if we let $a \sim b$ mean "a is connected to b", then:

- $\sim$ is reflective, i.e., $a \sim a$.
- $\sim$ is symmetric, i.e., $a \sim b$ implies $b \sim a$.
- $\sim$ is transitive, i.e., $a \sim b$ and $b \sim c$ implies $a \sim c$.

Hence, $\sim$ is what you would call an **equivalence relation**, and the **equivalence classes** under $\sim$ are the connected components of the graph.

Finding connected components can be done with BFS or DFS, which you probably already knew. Both algorithms work fine.[2]

A **cycle** is a nontrivial path from a to itself. We say a graph is **acyclic** if it has no cycles. An undirected acyclic graph is called a **forest**. A connected forest is a **tree**.

The natural question is how to detect if there's a cycle in an undirected graph. For this, you

[1] "Maximal" means you can't add any more nodes without violating the requirement.
[2] though DFS usually gets deeper in recursion than BFS.

can use BFS or DFS again to detect if a cycle exists (how?) and find such a cycle (how?) in $O(n+e)$ time. A fancier (and perhaps easier-to-implement) approach is **union-find**, although it runs in $O(n+e\cdot\alpha(n))$, where α is the slow-growing *inverse Ackermann function*. Due to this factor, union-find is noticeably slower when $n \geq 2^{2^{2^{65536}}}$, so this is unacceptable. Just kidding! $\alpha(2^{2^{2^{65536}}}) \leq 5$, so in practice this is fine. (Though if the time limits are particularly tight, that factor of 5 sometimes bites, so it could be better to use a BFS/DFS instead. You have to judge when to use union-find or not.)[3]

1.1 Depth-first search revisited (undirected version)

Now it's time to take a closer look at one of the simplest graph traversal methods: DFS. The DFS will be central in the algorithms we'll discuss later on, so now is a good time to revisit DFS with more detail.

DFS, in its simplest form, is just a particular order of traversal of the graph determined by the following recursive procedure: (in pseudocode)[4]

```
1  function DFS(i):
2     // perform a DFS starting at node i
3
4     visited[i] = true // mark it as visited
5     for j in adj[i]:
6        if not visited[j]:
7           DFS(j)
8
9  function DFS_all():
10    // perform a DFS on the whole graph
11    for i = 0..n-1:
12       visited[i] = false
13
14    for i = 0..n-1:
15       if not visited[i]:
16          DFS(i)
```

On its own, it's not very interesting, since all this does is *visit* all nodes (and mark "`visited[i]`" as true). But we can actually extract more information from this DFS procedure. The first (useful) thing we can do is generalize:

[3] Also, note that this only detects a cycle; it's also harder to find the cycle with union-find.

[4] I expect you can easily convert pseudocode into real code by now!

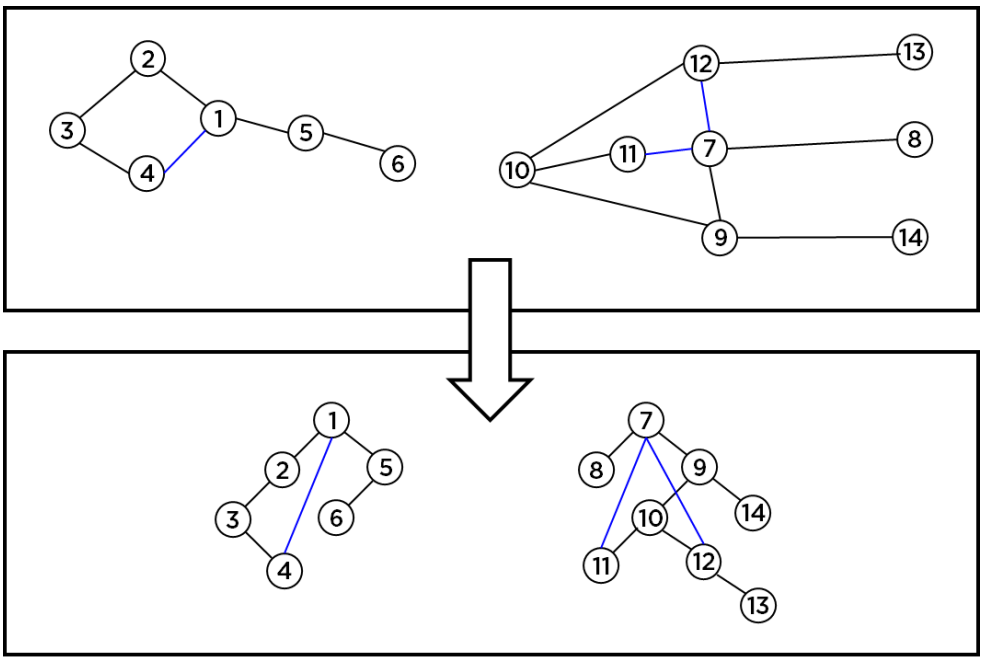
```
function DFS(i, p):
   visited[i] = true
   parent[i] = p

   visit_start(i) // do something once a new node is visited

   for j in adj[i]:
      if not visited[j]:
         DFS(j, i)

   visit_end(i) // do something once a node has finished expanding

function DFS_all():
   for i = 0..n-1:
      visited[i] = false
      parent[i] = -1

   for i = 0..n-1:
      if not visited[i]:
         DFS(i, -1)
```

Here, the function "visit_start" and "visit_end" are whatever you wanted to do with the nodes. They will be called once for each node, in order of the starting and ending times of the nodes, respectively. This generalized version is quite useful in many cases.

Notice also that we're computing the "parent" array, denoting the parent of the nodes in the traversal. This could be useful in some cases.

But actually, DFS is more interesting and useful than that; there are still other bits of information we can extract from this traversal that can help us solve some problems.

To find this hidden information, let's try consider a particular DFS traversal, and then let's draw the nodes on paper so that each node appears "beneath" its parent, and the "children" of a node are ordered from left to right according to the order of visitation. For instance, it might look like this:



tree edges, back edges

The black edges represent the edges that are traversed by the DFS. We call such edges **tree edges**. The remaining edges, colored in blue, are the ones ignored by the DFS, and are called **back edges**. Thus, we can clearly see that DFS classifies the edges into one of two types, depending on how they were handled by the DFS traversal! We'll see later on that this classification will be useful for us.

Note that, due to the depth-first nature of DFS, other types of edges in the DFS forest can't appear, such as nontree edges that don't point to an ancestor. Please convince yourself of this.

If we consider only the black edges, then we get a forest. For this reason, this is sometimes called the **DFS forest** of the graph.

To be more specific, the classification of the edges is done by the following procedure:

```
1  function DFS(i, p):
2     start_time[i] = time++
3     parent[i] = p
4
5     for j in adj[i]:
6        if start_time[j] == -1:
7           mark (i, j) as a tree edge
8           DFS(j, i)
9        else if j != p:
10          mark (i, j) as a back edge
11
12    finish_time[i] = time++
13
14 function DFS_all():
15    time = 0
16    for i = 0..n-1:
17       start_time[i] = -1
18       finish_time[i] = -1
19       parent[i] = -1
20
21    for i = 0..n-1:
22       if start_time[i] == -1:
23          DFS(i, -1)
```

An important thing to note here is that the condition `j != p` is checked before marking some edge as a back edge; this is very important, otherwise we will be marking all edges as back edges! (Why?) [5]

Notice that we've also replaced the `visited` array with two new arrays `start_time` and `finish_time`, which will contain the starting and finishing times of each node's visitation. For now, there aren't many uses to them, but they will be more useful for us later on.

The running time is still $O(n + e)$, but along the way, we've gained more information about the DFS traversal, namely the edge classifications, the starting and ending times of each visitation, and the parents of the nodes! These pieces of information will prove valuable in the upcoming algorithms.

By the way, note that the implementation above is written *recursively*. In some large trees, stack overflow might be a concern, especially if the stack size is limited. In those cases, you might want to implement an *iterative* version.[6]

[5] An even further complication in this part of the code is when there are **parallel edges**, that is, multiple edges that connect the same pair of nodes. In this discussion, we'll assume that there are no parallel edges. But if you want to learn how to deal with parallel edges, you can try to figure it out yourself, or just ask :)

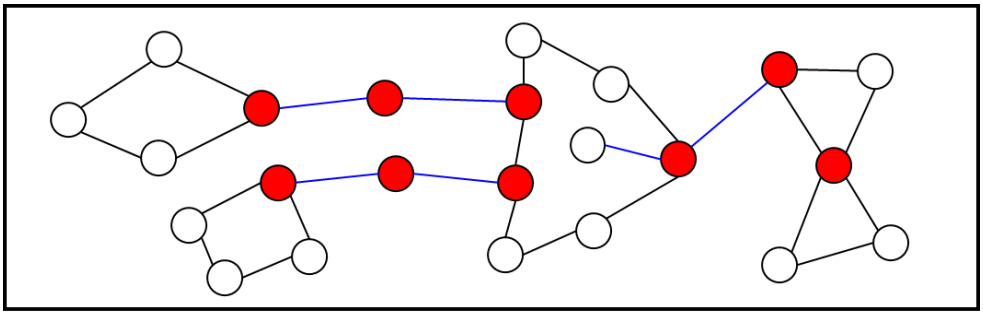
[6] A generic way to do that conversion is to simulate the call stack with an actual stack, and the stack entries describe the whole state of the function at that point. In this case, you only need to push "`i`" and the index of "`j`" in "`adj[i]`".

1.2 Bridges and articulation points

A **bridge** is an edge whose removal increases the number of connected components. An **articulation point** is a node whose removal increases the number of connected components. Note that when you remove a node you also remove the edges adjacent to it.

For simplicity, let's assume here that the graph is connected; if not, we can consider each connected component separately. Thus, we will use the following specialized definitions: In a connected graph, a **bridge** is an edge whose removal disconnects the graph, and an **articulation point** is a node whose removal disconnects the graph.

In the following picture, the blue edges are the bridges, and the red nodes are the articulation points:



bridges, articulation points

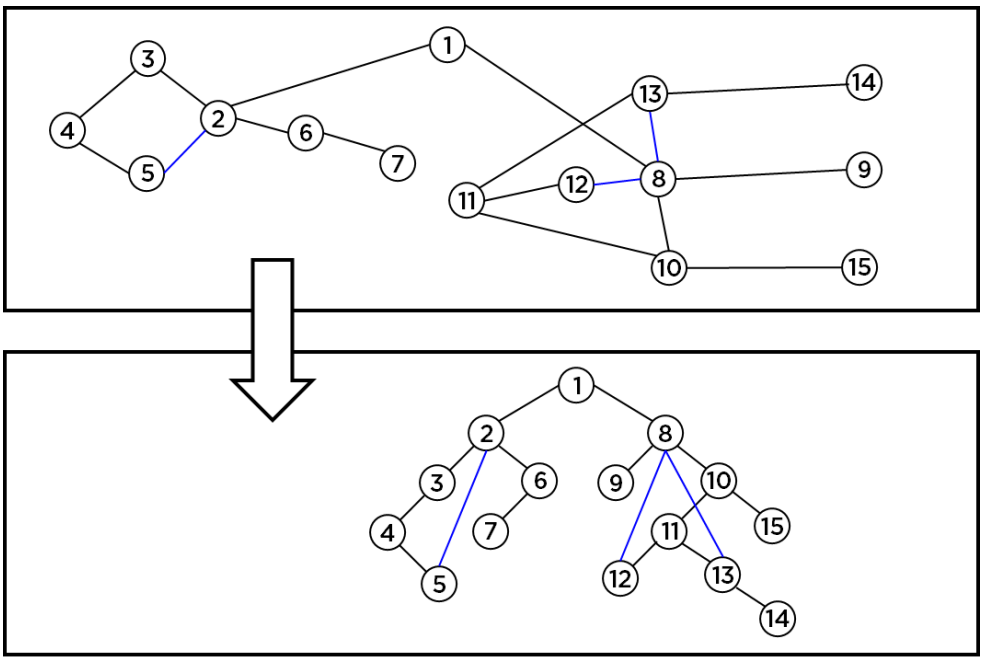
It's easy to see why one would identify and study these edges/nodes. Roughly speaking, these edges and nodes are the "weakest links" of your network; if you're modelling a computer network, then a bridge could represent a critical connection, and an articulation point could represent a critical computer.

Bridges and articulation points are also sometimes called "cut edges" and "cut vertices", respectively, for obvious reasons.

1.2.1 Finding bridges and articulation points

Given a (connected) undirected graph, how do you find all its bridges and articulation points? If the graph is small enough, you can just individually check each edge/node if they are a bridge/articulation point by removing it and checking if the resulting graph is disconnected. Since it takes $O(n+e)$ time to traverse a graph, it takes $O(e(n+e))$ and $O(n(n+e))$ time to find the bridges and articulation points this way.

But it turns out that we can compute both in $O(n+e)$ time using DFS! To see how, let's say we performed DFS on our graph. The following is a picture of the resulting "DFS forest" (which is really just a "DFS tree" since the graph is connected):



DFS tree

Stare at this picture for a while and think about exactly when an edge is a bridge, or when a node is an articulation point.

After a bit of pondering, we can state the conditions precisely. Note that the terms "**ancestor**" and "**descendant**" refer to the nodes' relationships in the DFS tree, and a node is considered an ancestor and a descendant of itself.

- Back edges can never be bridges.
- Let (a, b) be a tree edge, where a is the parent. Then (a, b) is a bridge if and only if there's no back edge from any descendant of b to any ancestor of a.
- Let a be a node. Then a is an articulation point iff either of the following is true:
  - a is not the root of a DFS tree and a has a child b such that there's no back edge from any descendant of b to any *proper* ancestor[7] of a.
  - a is the root of a DFS tree and a has at least two children.

Take note of the second case regarding articulation points; it's easy to miss, but important.

With these observations, we can now compute the bridges and articulation points by augmenting DFS with additional data:

- Let $\texttt{disc}[i]$ be the discovery time of i. (Identical to $\texttt{start\_time}[i]$ above.)

[7] A proper ancestor of a is an ancestor distinct from a.

- Let $\texttt{low}[i]$ be the lowest discovery time of any ancestor of i that is reachable from any descendant of i with a single back edge. If there are no such back edges, we say $\texttt{low}[i] = \texttt{disc}[i]$.

Using $\texttt{disc}[i]$ and $\texttt{low}[i]$, we can now state precisely when something is a bridge or an articulation point:

- Let (a, b) be a tree edge, where a is the parent. Then (a, b) is a bridge iff $\texttt{low}[b] > \texttt{disc}[a]$.
- Let a be a node. Then a is an articulation point iff either of the following is true:
  - a is not the root of a DFS tree and a has a child b such that $\texttt{low}[b] \geq \texttt{disc}[a]$.
  - a is the root of a DFS tree and a has at least two children.

Thus, the only remaining task is to compute $\texttt{disc}[i]$ and $\texttt{low}[i]$ for each i. But we can compute these values *during the DFS*, like so:

```
function DFS(i, p):
   disc[i] = low[i] = time++

   children = 0
   has_low_child = false
   for j in adj[i]:
      if disc[j] == -1:
         // this means (i, j) is a tree edge
         DFS(j, i)

         // update low[i] and other data
         low[i] = min(low[i], low[j])
         children++

         if low[j] > disc[i]:
            mark edge (i, j) as a bridge

         if low[j] >= disc[i]:
            has_low_child = true

      else if j != p:
         // this means (i, j) is a back edge
         low[i] = min(low[i], disc[j])

   if (p == -1 and children >= 2) or (p != -1 and has_low_child):
      mark i as an articulation point

function bridges_and_articulation_points():
   time = 0
   for i = 0..n-1:
      disc[i] = -1

   for i = 0..n-1:
      if disc[i] == -1:
         DFS(i, -1)
```

This procedure now correctly identifies all bridges and articulation points in the graph!

An important thing to understand here is how $\texttt{low}[i]$ is being computed. Please ensure that you understand it.

2 Connectivity in Directed Graphs

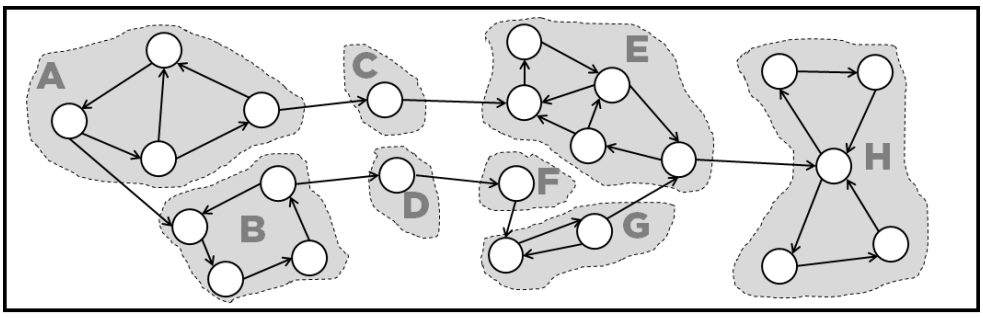
Since connectivity with undirected graphs is so boring, let's consider directed graphs instead. The important difference is that paths are not *reversible* anymore, so the nice picture of "connected components" above does not apply any more.

More formally, if we let $a \rightsquigarrow b$ mean "there is a path from a to b", then $\rightsquigarrow$ is still reflexive and transitive, but is not necessarily symmetric any more. Thus, "equivalence classes" are not well-defined any more.

But we can fix this: If we let $a \sim b$ mean "$a \rightsquigarrow b$ and $b \rightsquigarrow a$", i.e., "there is a path from a to b and vice-versa",[8] then it's easy to verify that $\sim$ is reflexive, symmetric and transitive, hence it's an equivalence relation!

If $a \sim b$, then we say a **and** b **are strongly connected**. A **strongly connected component**, or **SCC**, is a maximal set of nodes that is pairwise strongly connected. In other words, the SCCs are the equivalence classes under $\sim$. SCCs are the best analogue of connected components in undirected graphs.

The following offers a picture of the strongly connected components of an example directed graph:



strongly connected components (SCC)

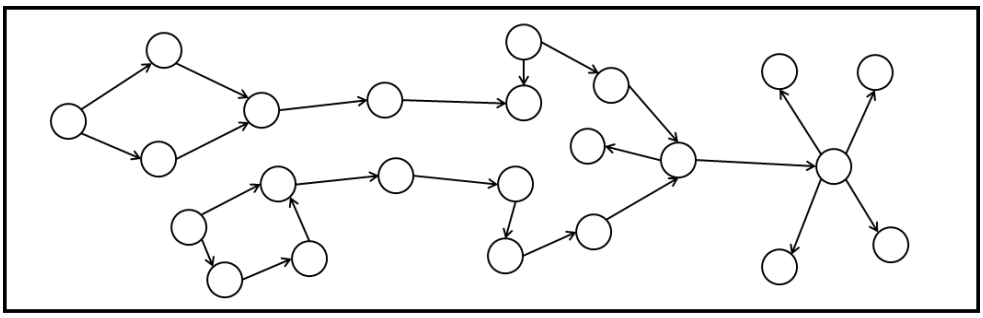
Note that this time, there could be edges from one SCC to another. This is more exciting than before!

2.1 Cycles and DAGs

A **cycle** is a nontrivial path from a to itself. We say a graph is **acyclic** if it has no cycles. A directed acyclic graph is called, well, a **directed acyclic graph**, or **DAG**.

Here's an example of a DAG:

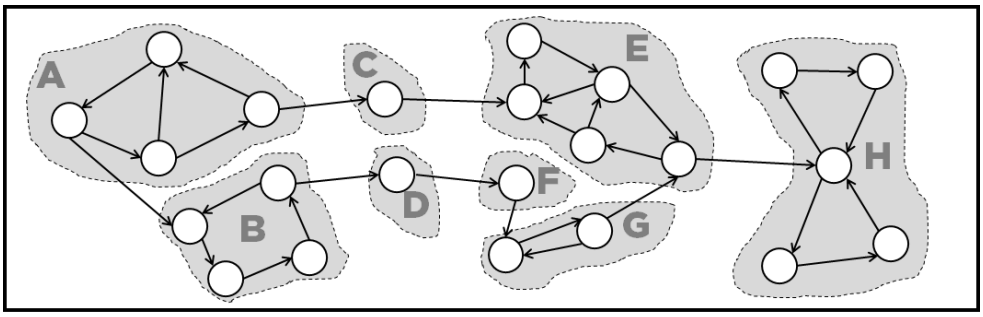
[8] obviously these are two different paths.



directed acyclic graph (DAG)

One nice thing about DAGs is that you can *serialize* the nodes, i.e., find a total order of the nodes such that every edge connects a node to a further node in the total order. This is called a **topological sort**, or *toposort*, of a graph. You probably already learned how to compute the topological sort in linear time.

Now, let's go back to our previous example:



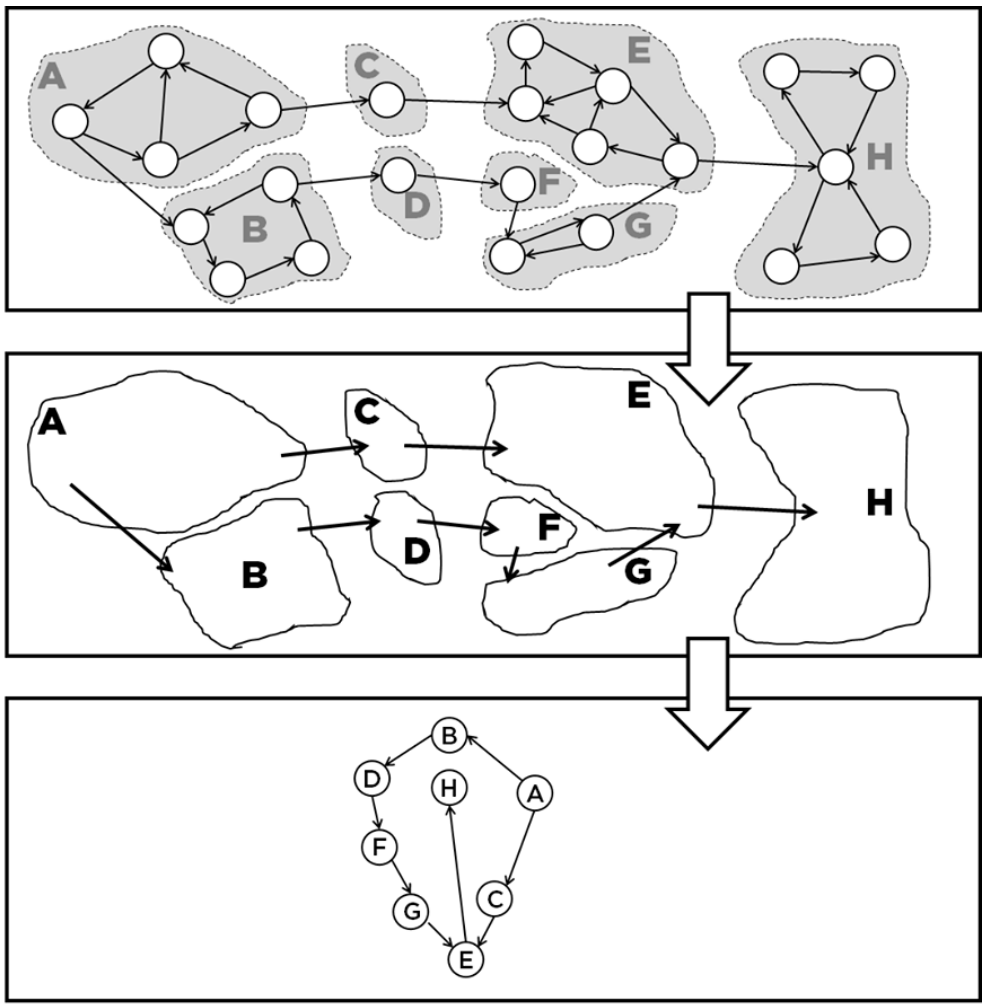
strongly connected components (SCC)

suppose we "shrink" each SCC into a single node, and for every edge $a \to b$ connecting two nodes from different SCCs, we add an edge $\mathrm{SCC}_a \to \mathrm{SCC}_b$, where SCC_x is the node of the SCC containing x. We then obtain a new, smaller graph. My question is: can there be a cycle in a graph formed this way?

It turns out that there can't be any cycle in such a graph! The simple reason is that if there were a cycle, then the SCCs in that cycle could be compressed further into a single SCC, contradicting the fact that they're maximal. Thus, a cycle cannot exist.

Hence, if we shrink the SCCs into a single node, then we get a DAG, which we'll just call the **DAG of SCCs**.[9]

[9] You'll also see this called the **condensation** of the graph. Since $\sim$ is an equivalence relation, you may even see it called as "**the graph modulo $\sim$**".



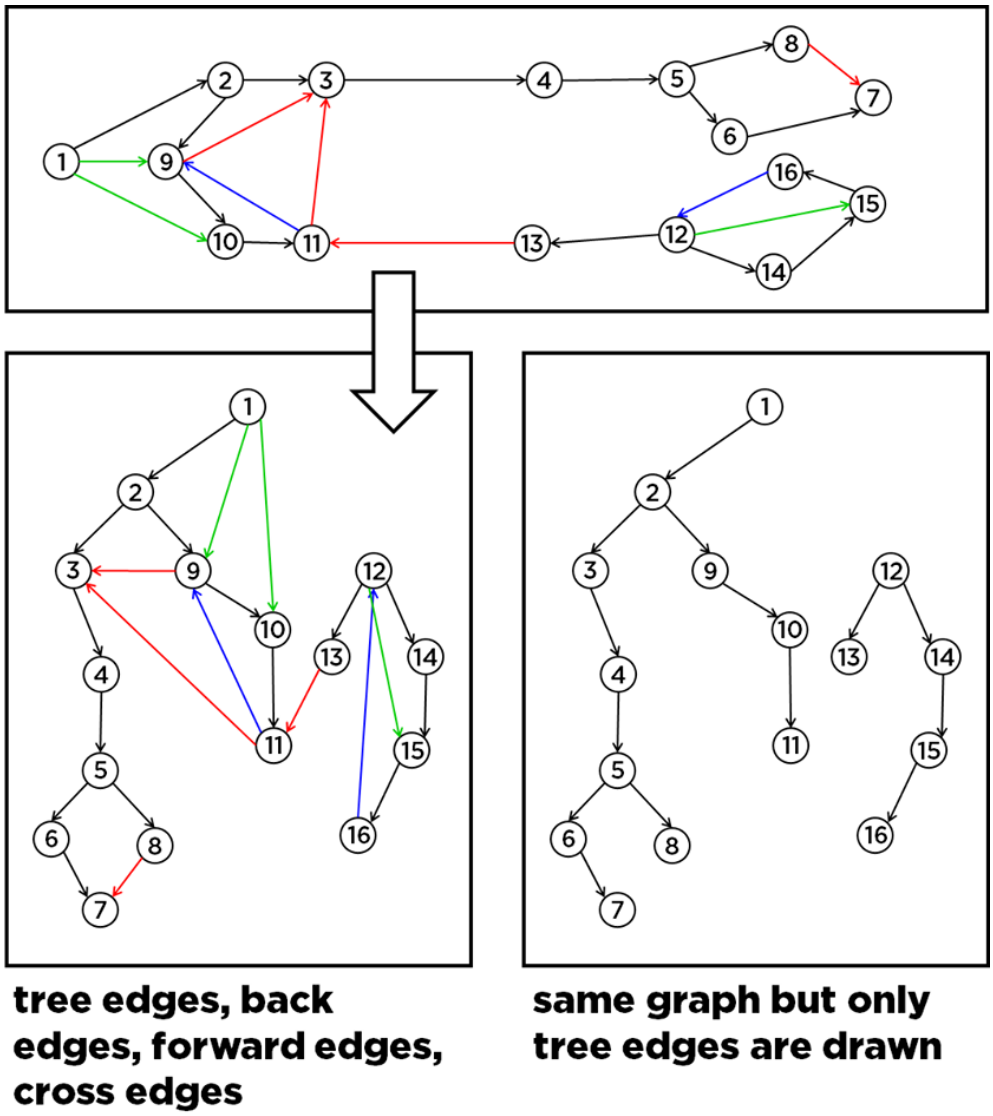
DAG of SCCs

Note that this is already more interesting and has more structure than in the undirected case, where shrinking the connected components results in just a graph with no edges!

2.2 Depth-first search revisited (directed version)

Let's discuss how DFS works in the case of directed graphs. It turns out that the DFS forest in a directed graph provides a similar set of useful information as in the undirected case.

For instance, let's look at how the "DFS forest" might look like in a directed graph:



Note that there are still black and blue edges, representing tree and back edges. However, it looks like there are two new types of edges! It seems that the DFS on a directed graph classifies the edges into one of *four* types this time:

1. The black edges are the **tree edges**, which are the edges genuinely traversed by the DFS, i.e., $i \to j$ is a tree edge if the first time j is visited is through i.

2. The blue edges are the **back edges**, which are edges that point to an ancestor of a node in the DFS forest.

3. The green edges are the **forward edges**, which are edges that point to a descendant of a node in the DFS forest.

4. The red edges are the **cross edges**, which are edges that point to neither an ancestor nor a descendant of a node in the DFS forest.

It's sometimes convenient to consider a tree edge as a type of forward edge as well, although there are forward edges that are not tree edges (unlike in the undirected case).

Now, let's look at how a DFS procedure could identify these edges:

```
1  function DFS(i):
2     // perform a DFS starting at node i
3
4     start_time[i] = time++
5
6     for j in adj[i]:
7        if start_time[j] == -1:
8           mark (i, j) as a tree edge
9           DFS(j)
10       else if finish_time[j] == -1:
11          mark (i, j) as a back edge
12       else if finish_time[j] > start_time[i]:
13          mark (i, j) as a forward edge
14       else:
15          mark (i, j) as a cross edge
16
17    finish_time[i] = time++
18
19 function DFS_all():
20    time = 0
21    for i = 0..n-1:
22       start_time[i] = -1
23       finish_time[i] = -1
24
25    for i = 0..n-1:
26       if start_time[i] == -1:
27          DFS(i)
```

Notice how we made use of the values `start_time`$[i]$ and `finish_time`$[i]$ to distinguish between back, forward and cross edges. In particular, assume `start_time[j] != -1`. This means that we have already started visiting node j. Thus, in the inner `for` loop above,

- If `finish_time[j] == -1`, then j is being visited while we're on i, which means j must be an ancestor of i in the DFS forest.
- If `finish_time[j] != -1` and `finish_time[j] > start_time[i]`, then j's visitation is already finished, but only after i's visitation began, so it means j must be a descendant of i.
- If `finish_time[j] != -1` and `finish_time[j] < start_time[i]`, then j's visitation is already finished even before we started visiting i, hence j is neither an ancestor

or descendant of i.

The running time is still $O(n + e)$, but along the way, we've again obtained useful information about our directed graph!

As with the undirected case, note that the implementation above is written *recursively*. In some large trees, stack overflow might be a concern, especially if the stack size is limited. In those cases, you might want to implement an *iterative* version.[10]

2.3 Cycle finding

Now, let's go back to discussing cycles. Given a directed graph, how do we detect if it has a cycle? Note that it already looks more interesting than the undirected case, and in fact there are many interesting approaches.

You probably already knew one approach, which is to run a toposort algorithm on the graph, and check if it failed. The toposort algorithm fails if and only if there is a cycle in the graph, hence this correctly solves our problem! But if you were asked to actually find a cycle, then it could get tricky depending on the toposort algorithm used.

But it's still worthwhile to discuss additional approaches to this problem. For instance, a simple algorithm arises from the following insight: *there is a cycle if and only if there is a back edge in the DFS forest*. (Why?) Thus, we can detect a cycle by performing a DFS (like above) and stopping once we find a back edge! Another advantage of this is that it's easy to actually find a cycle if one is detected. (How?)[11]

One can also detect (and find) a cycle using BFS, although we will leave it to you to discover.

2.4 Floyd's cycle finding algorithm

Let's restrict ourselves to a special kind of graph. Specifically, let's only consider graphs where each node has exactly one outgoing edge. Let's call such graphs **function graphs** because on such a graph, we can define a function $f : V \to V$ where $f(x) = y$ if and only if $x \to y$ is an edge.[12] Since there is exactly one outgoing edge from each node, f is well-defined. Conversely, every function $f : S \to S$ corresponds to a function graph whose node set is S and whose edges are $\{(x, f(x)) : x \in S\}$ (we're implicitly allowing self-loops here, but that's fine).

[10] Please see the footnote on the undirected case version for one way of dealing with this.

[11] In fact, by pursuing this idea further, you can use a DFS to actually extract a toposort of a DAG: Just order the nodes by decreasing finishing time! Think about why that works.

[12] We can also consider more general graphs with *at most one* outgoing edge per node. We can convert such graphs into function graphs by adding a new node, say trash, and pointing all nodes without an outgoing edge to trash (including trash itself).

Now, starting at any node, there's only one path we can follow. In terms of f, starting at some node x and following the (only) path corresponds to iteratively applying f on x, thus, the sequence of nodes we visit is:[13]

$$(x, f(x), f(f(x)), \ldots, f^{(n)}(x), \ldots)$$

Now, if we assume that our node set is finite, then (by the pigeonhole principle) this will eventually repeat, i.e., there will be indices $0 \leq i < j$ such that $f^{(i)}(x) = f^{(j)}(x)$.[14] In fact, once this happens, the subsequence $(f^{(i)}(x), f^{(i+1)}(x), \ldots, f^{(j-1)}(x))$ will repeat forever. This always happens regardless of which node you start at.

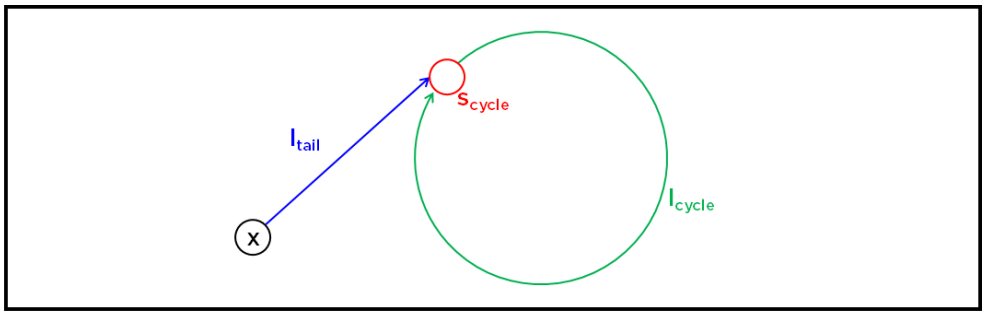
This gives rise to a natural question: Given a starting point x, when is the first time that a repeat happens? Furthermore, how long is the cycle? To make the question more interesting, suppose we don't know anything about f apart from:

1. f's domain and codomain are the same and are finite.
2. We can evaluate $f(x)$ at any x. (For this discussion, we'll assume we can do it in $O(1)$.)

We can formalize the **cycle-finding problem** as: Given a function f with the above properties, and a starting point x, compute the following:

1. s_{cycle}, defined as the first node that repeats in the sequence.
2. l_{cycle}, defined as the length of the (repeating) cycle.
3. l_{tail}, defined as the distance from x to s_{cycle}.

We can visualize these values with the following:



cycle finding

A simple approach is to use BFS or DFS, which is equivalent to just following the (only) path and storing the visited nodes until we encounter a node we've already visited.

[13] Here, $f^{(n)}(x)$ is f applied to x a total of n times.
[14] Note that this is no longer true if the graph is infinite. Why?

```
// Just−walk algorithm
function cycle_find(x):
   visit_time = new empty map
   time = 0
   s = x
   while not visit_time.has_key(s):
      visit_time[s] = time++
      s = f(s)

   s_cycle = s
   l_tail = visit_time[s]
   l_cycle = time - l_tail
   return (s_cycle, l_cycle, l_tail)
```

Assuming no preprocessing and no specialized knowledge on f, this is probably close to the fastest we can do. It needs $O(l_{\text{tail}} + l_{\text{cycle}})$ time.

It also needs $O(l_{\text{tail}}+l_{\text{cycle}})$ memory, but one might wonder if it can be improved upon. Amazingly, there's actually a way to compute it using $O(1)$ memory, called **Floyd's cycle-finding algorithm**!

Now, you might ask, why the need for a fancy algorithm? Surely it's trivial to find an $O(1)$-memory solution. Here's one:

```
// Bogo−cycle−finding algorithm
function cycle_find(x):
   for time in 1,2,3,4...
      s = x
      for i = 1..time:
         s = f(s)

      s_cycle = s

      s = x
      l_tail = 0
      while s_cycle != s:
         s = f(s)
         l_tail++

      if l_tail < time:
         l_cycle = time - l_tail
         return (s_cycle, l_cycle, l_tail)
```

Let's call this the **bogo-cycle-finding algorithm**. Although it might not be obvious why this works, clearly this is $O(1)$ memory! Well, that's certainly true, but this is an incredibly slow solution! The idea is to use only $O(1)$ memory without sacrificing running time.

Let's discuss Floyd's algorithm. This is also sometimes called the **tortoise and the hare algorithm**, since we will only use two pointers, called the *tortoise* and the *hare*, respectively.

The idea is that both the tortoise and the hare begin walking at the starting point, but the hare is twice as fast. This means that at the beginning, the hare might be quite ahead of the tortoise, but once they both enter the cycle, they will eventually meet. Once we they meet, they stop, and the hare *teleports* back to the starting point. They then proceed walking *at the same speed* and stop once they meet. This meeting point will be s_{cycle}!

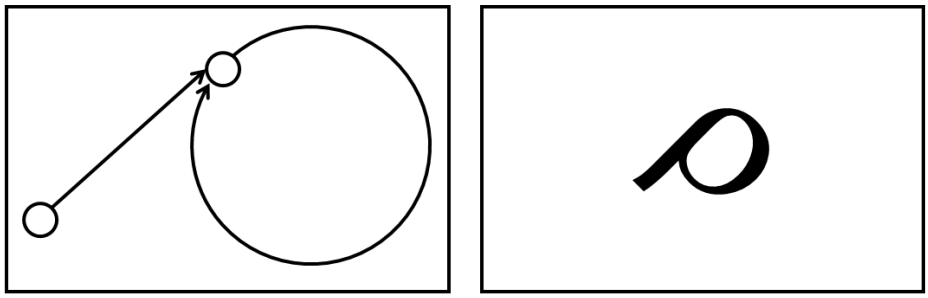
Once we get s_{cycle}, l_{tail} and l_{cycle} can easily be computed, e.g., l_{cycle} can be computed by going around the cycle once. Here's the pseudocode:

```
1  // Floyd's cycle-finding algorithm
2  function cycle_find(x):
3     tortoise = hare = x
4     do:
5        tortoise = f(tortoise)
6        hare = f(f(hare))
7     while tortoise != hare
8
9     // teleport, and walk at the same speed
10    hare = x
11    l_tail = 0
12    while tortoise != hare:
13       tortoise = f(tortoise)
14       hare = f(hare)
15       l_tail++
16
17    s_cycle = hare
18
19    // compute l_cycle by walking around once.
20    l_cycle = 0
21    do:
22       hare = f(hare)
23       l_cycle++
24    while tortoise != hare
25
26    return (s_cycle, l_cycle, l_tail)
```

This clearly uses $O(1)$ memory. We've also mentioned that this runs in $O(l_{\text{tail}}+l_{\text{cycle}})$, but I didn't provide a complete convincing proof. It's not even clear why this correctly computes "s_{cycle}". We leave it to you as an exercise to prove the running time and correctness of this algorithm!

2.4.1 Cycle-finding and factorization: Pollard's ρ algorithm

An interesting application of Floyd's algorithm (or at least its idea) is with integer factorization. **Pollard's ρ algorithm**[15] is a factorization algorithm that can sometimes factorize numbers faster than trial-and-error division. The name comes from the shape of the path when starting at some value:



side-by-side comparison of pollard-rho diagram (left) and a tilted greek letter rho (right)

The algorithm accepts N, the number to be factorized, along with two additional parameters, a starting value s, and a function $f : \{0, 1, \ldots, N-1\} \to \{0, 1, \ldots, N-1\}$, which must be a polynomial modulo N. The algorithm then attempts to find a divisor of N. One issue with this algorithm is that *it's not guaranteed to succeed*, so you may want to run the algorithm multiple times, with differing s (and possibly f).

Suppose we want to factorize a large number N. Also, suppose $s = 2$ and $f(x) = (x^2+1) \bmod N$. Here is the pseudocode of Pollard's ρ-algorithm.

```
1  function try_factorize(N):
2     x = y = 2 // starting point
3     do:
4        x = f(x)
5        y = f(f(y))
6        d = gcd(|x - y|, N)
7     while d == 1
8
9     if d == N:
10       failure
11    else:
12       return d
```

[15] ρ is pronounced "rho".

One can clearly see f being used as the iteration function, and x and y are assuming roles that are similar to the tortoise and hare, respectively. If this fails, you could possibly try again with a different starting point, or perhaps a different f. (Of course, this will always fail if N is prime.)

A more thorough explanation of why this works, and why cycle-finding appears, can be seen on the Wikipedia page: https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm.

2.5 Computing strongly connected components

So far we've discussed what SCCs are, along with some of their properties. But we haven't explained how to compute them yet. Unlike in the undirected case, a naïve search won't work here. (Why?)

We will discuss two algorithms. It's instructive to learn both, and then possibly choose your preferred algorithm later on.

Note that these sections go into quite some detail in proving the correctness of the algorithms, so unless you're comfortable, you might want to skip the proofs on first reading and just learn how the algorithms work first.

2.5.1 Kosaraju's algorithm

Here, we describe **Kosaraju's algorithm** which computes the strongly connected components of a directed graph.

Here are the steps of Kosaraju's algorithm:

1. Mark all vertices as not visited.
2. Perform a DFS on the whole graph (in an arbitrary order). Take note of the *finishing times* of the nodes.
3. Reverse the directions of the edges.
4. Again, mark all vertices as not visited.
5. Perform another DFS traversal, this time in decreasing order of finishing time (which were calculated earlier). Every time we start a new top-level DFS traversal `DFS(i)`, all the nodes visited in that run constitutes a strongly connected component.

Since we already know how to perform a DFS, this is easy to implement! Also, this runs in $O(n + e)$ time. You might wonder why not $O(n \log n + e)$ since we need to sort the nodes in decreasing order of finishing time, but there's actually no need to do that, since we can simply push every node that we just finished visiting onto a *stack*. Then, in the next phase, we simply

pop from the stack to determine the order. This works because the nodes that are finished last will be the ones popped first from the stack! In fact, using this stack, we don't really even need to store the finishing times.

Here's a pseudocode of the algorithm:

```
function DFS(i, adj, group):
   // DFS starting at i, using the adjacency list 'adj'
   // then push all visited nodes to the vector 'group'
   visited[i] = true
   for j in adj[i]:
      if not visited[j]:
         DFS(j, adj, group)

   group.push(i)

function SCC_Kosaraju():
   // mark all nodes as unvisited
   for i = 0..n-1:
      visited[i] = false

   stack = new empty vector
   for i = 0..n-1:
      if not visited[i]:
         DFS(i, adj, stack)

   // create the reversal graph
   jda = new adjacency list
   for i = 0..n-1:
      for j in adj[i]:
         jda[j].push(i)

   // reinitialize visited
   for i = 0..n-1:
      visited[i] = false

   // do DFS again on jda, in decreasing order of finishing time
   sccs = new empty vector of vectors
   while not stack.empty():
      i = stack.pop()
      if not visited[i]:
         scc = new empty vector
         DFS(i, jda, scc)
         sccs.push(scc)
   return sccs
```

Now, why does it work? We'll provide a proof here, but you may want to skip it if you want

spend some time thinking about it yourself.

The following is a rough proof of why this algorithm is correct. It relies on the fact that the SCCs of a graph is the same as the SCCs of its reversal graph (this can be proved very easily). Let us denote by G^R the graph G with all its edges reversed.[16]

Claim 2.1. *Kosaraju's algorithm correctly computes the strongly connected components of a graph G.*

Proof. In order to prove that the algorithm is correct, we only need to ensure that in phase two, whenever we start a top-level DFS in G^R, we do it in such an order that *all the reachable nodes that belong to a different SCC have already been visited*. This is sufficient because if this is true, then the first time we visit a node in some SCC B is when we actually start the DFS on a node in B, not on a node in a different SCC that can reach B (otherwise, this would contradict the above statement), and once we start a DFS in B, all nodes in B will be visited by this DFS (because they are reachable from each other), and only those in B will be visited, because all the nodes in the other SCCs reachable from B have already been visited (again according to the statement). Therefore, whenever we start a new DFS, we visit exactly those nodes that belong to an SCC.

Now, consider two distinct SCCs of G, say A and B, and suppose there is a path from some node in A to some node in B. Since A and B are SCCs, it follows that there is no path from any node in B to any node in A. Now, during the first phase, where we are performing the DFS in an arbitrary order, there are two cases:

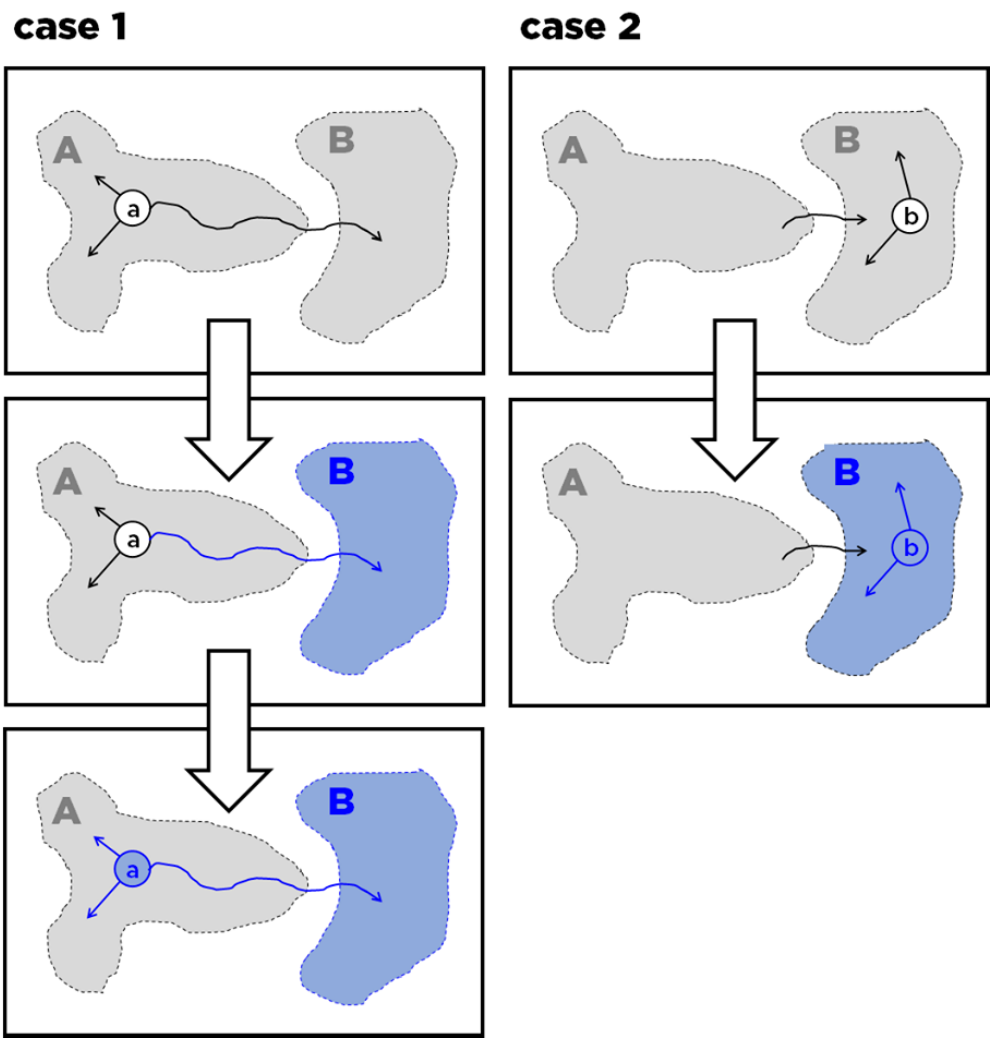
1. *A node in A, say a, is discovered before any node in B.* In this case, the DFS will be able to traverse the path from A to B and visit all nodes in B before a itself finishes expanding.[17] Therefore, the finishing time of a is greater than the finishing time of any node in B.

2. *A node in B, say b, is discovered before any node in A.* In this case, the DFS will finish visiting all nodes in B before it ever reaches any node in A, because there is no path from B to A. Therefore, all nodes in A have a greater finishing time than all nodes in B.

The two cases are illustrated below:

[16] G^R is also called the **transpose** of G.

[17] By "finishing expanding" I mean finishing the DFS run on the node and returning from the call. In the DFS pseudocode, it's precisely the time when we set a node's `finish_time`.



What this shows is that regardless of the order we visit the nodes for the DFS, as long as there exists a path from component A to B, there will always be a node in A that has a greater finishing time than all nodes in B. More generally, if $A_1, A_2, \ldots, A_k$ are all the SCCs that can reach B, then there exists a node in each one of those components that have a greater finishing time than all nodes in B.

In the reversal graph G^R, $A_1 \ldots A_k$ are precisely the SCCs that B can reach, and therefore none of the A_i can reach B. And since in the second phase we are performing the DFS in decreasing order of finishing times, it follows that we will have done a DFS on each A_i before we visit any on B, and thus, all nodes in $A_1 \ldots A_k$ will have been visited before any node in B. Therefore, once we start visiting a node in B, all the nodes reachable from it that belong to a different SCC have already been visited. This is exactly what we wanted to prove. □

As a side note, the main idea in this proof can be repurposed to prove that we can get a topological sort of the DAG by ordering the nodes by decreasing finishing time:

Claim 2.2. *A topological sort of a DAG is obtained by performing a DFS on it and ordering the nodes in decreasing finishing time.*

Proof. Note that the main idea in the previous proof is to show that *for two SCCs A and B, if there is a path from some node in A to some node in B, then there is a node in A with a greater finishing time than all nodes in B.*

But the SCCs of a DAG consist of single nodes, thus A has exactly one element, say a, and B has exactly one element, say b, so it simply says that *if there is a path from a to b, then a has a greater finishing time than b.* In particular, paths of length 1, i.e., single edges, point from a node to another node with a lower finishing time, hence ordering the nodes in decreasing finishing time results in a valid topological sort! □

As in Kosaraju's algorithm, you can construct the toposort without computing the finishing times explicitly by pushing the just-finished nodes onto a stack, and then reversing the stack in the end.

2.5.2 Tarjan's SCC algorithm

Here, we describe another algorithm called *Tarjan's SCC algorithm*. Tarjan's SCC algorithm also uses a DFS, but unlike Kosaraju's algorithm, it needs only one DFS traversal. It works by augmenting the DFS procedure with additional bookkeeping data that's enough to identify the SCCs.

This algorithm uses the `disc` and `low` arrays, just like in our algorithm for bridges and articulation points!

Here's the pseudocode:

```
function DFS(i):
   disc[i] = low[i] = time++

   stack.push(i)
   instack[i] = true
   for j in adj[i]:
      if disc[j] == -1:
         DFS(j)
         low[i] = min(low[i], low[j])
      else if instack[j]:
         low[i] = min(low[i], disc[j])

   if low[i] == disc[i]:
      get_scc(i)

function get_scc(i):
   // pop the stack until you pop i, and collect those as an SCC
   scc = new empty vector
   do:
      j = stack.pop()
      instack[j] = false
      scc.push(j)
   while j != i
   SCCs.push(scc)

function SCC_Tarjan():
   stack = new empty vector
   sccs = new empty vector of vectors
   time = 0
   for i = 0..n-1:
      disc[i] = -1
      instack[i] = false

   for i = 0..n-1:
      if disc[i] == 0:
         DFS(i)
   return sccs
```

Let's try to explain how this works. Let's first describe the following property of the DFS forest.

Claim 2.3. *The nodes of any SCC form a rooted subtree in the DFS forest.*[18]

A simple proof sketch is as follows. Here, we define the *head* of an SCC as the node that's

[18]Note that "subtree" means slightly different here. This doesn't mean that all nodes *down to the leaves* are part of the SCC. It means that, if you consider only the nodes of some SCC and ignore the rest (including possibly some nodes below them), then you get a rooted tree.

discovered first among all nodes in the SCC:

Proof. Consider two nodes a and b from a single SCC such that a is an ancestor of b in the DFS forest. Then every node from the path between them must belong to the same SCC. This is because for every node c in the path, a reaches c, c reaches b and b reaches a (since a and b are in an SCC), so c must also belong to the same SCC as a and b.

Next, let h be the head of an SCC. Then every other node in the SCC is a descendant of h in the forest, because they are all connected, and h is visited earliest. Therefore, combining the above with this, it follows that the SCC forms a rooted tree in the forest, with h as the root. □

Note that this also proves that the head is the root of that tree, and that the head is also the last to finish expanding among all nodes in an SCC.

Let's now describe $\texttt{disc}[i]$ and $\texttt{low}[i]$. They are defined similarly as before:

- Let $\texttt{disc}[i]$ be the discovery time of i.
- Let $\texttt{low}[i]$ be the lowest discovery time of any ancestor of i that is reachable from any descendant of i with a single back edge. If there are no such back edges, we say $\texttt{low}[i] = \texttt{disc}[i]$.

It's worth mentioning that $\texttt{low}[i]$ ignores forward edges or cross edges.

Using the ever-useful $\texttt{low}[i]$ and $\texttt{disc}[i]$, we can identify whether a node is a *head* or not.

Claim 2.4. *A node i is a head of some SCC if and only if $\texttt{low}[i] = \texttt{disc}[i]$.*

Here's a rough proof:

Proof. By the definition of $\texttt{low}$, we find that $\texttt{low}[i] \leq \texttt{disc}[i]$.

Thus, the claim is equivalent to saying that a node i is *not* a head of some SCC if and only if $\texttt{low}[i] < \texttt{disc}[i]$.

Now, note that $\texttt{low}[i] < \texttt{disc}[i]$ happens if and only if there is a node j reachable from i that is an ancestor of i in the tree (using only tree edges and back edges). But for such a j, i reaches j and j reaches i, so j is another member of the SCC containing i that has been discovered earlier. Therefore, i is not a head if and only if such a j exists, if and only if $\texttt{low}[i] < \texttt{disc}[i]$, which is what we wanted to prove. □

Now, we're calculating $\texttt{low}[i]$ and $\texttt{disc}[i]$ on the fly as we perform the DFS traversal, and since the head is the last to finish expanding, we can collect all the members of the SCC containing that head right at that moment. Conveniently enough, it turns out that the members of this SCC are all on top of the stack!

To see this, note that whenever we visit a node, we simply push it onto the stack. But when a node finishes expanding, we don't necessarily pop it from the stack. We only pop the stack whenever we finish expanding a head h, and we keep popping until h is popped.

Now, due to Claim 2.3 above, and the fact that we only pop when we finish expanding a head, we are guaranteed that for every two nodes i and j in the stack belonging to a single SCC, all nodes between them in the stack also belong to the same SCC, therefore all nodes in a single SCC in the stack are found in contiguous locations. Also, when we finish expanding a head h, all other nodes in the SCC of h are still in the stack. Therefore, whenever we pop from the stack, all the nodes popped belong to a single SCC!

After the traversal, we would have computed all the SCCs of the graph.

Clearly, the time complexity is $O(n + e)$. However, although the time complexity is the same with Kosaraju's algorithm, Tarjan's algorithm can still be seen as somewhat of an improvement over Kosaraju's algorithm in a few ways:

- Only one DFS is required.
- We don't have to build the reversal graph (which could be memory-intensive).
- The correctness of the algorithm is more easily seen.[19]
- It's usually faster in practice.

2.5.3 Which algorithm to use?

Now that you know both algorithms, which one should you now use? Well, it's really up to you. For me, the choice here is essentially whether to choose an easy-to-implement solution or a slightly faster solution. I usually choose Kosaraju's algorithm since it's easier to understand (and remember), although I know a lot of others who prefer Tarjan. In fact, it seems I'm in the minority. So it's up to you if you want to go mainstream or hipster.

2.6 DAG of SCCs

Finally, it's at least worth mentioning how we can construct the DAG of SCCs. Once we can compute the SCCs using any of the algorithms above, we can now construct the DAG of SCCs. In high level, the steps are:

1. Compute the SCCs of the graph.
2. "Shrink" the SCCs into single nodes.

[19] at least to some; honestly it took me quite some time to understand it myself.

3. Remove the self-loops and duplicate edges.

4. The resulting graph is the DAG of SCCs.

The "shrinking" part might be too vague, but a simple "engineer" approach can be used. Let's restate the steps above, but also expand that part:

1. Compute the SCCs of the graph. Suppose there are k SCCs.

2. Compute the array f, where $f[i]$ represents the index of the SCC containing i.

3. Construct a new graph with k nodes and initially 0 edges.

4. For every edge $a \to b$ in the original graph, if $f[a] \neq f[b]$, then add the edge $f[a] \to f[b]$ in the new graph (if it doesn't exist already).

5. The new graph is now the DAG of SCCs.

Congratulations! You may now use the DAG of SCCs (if you need it).

Note that this still runs in $O(n + e)$ time.

3 Biconnectivity in Undirected Graphs

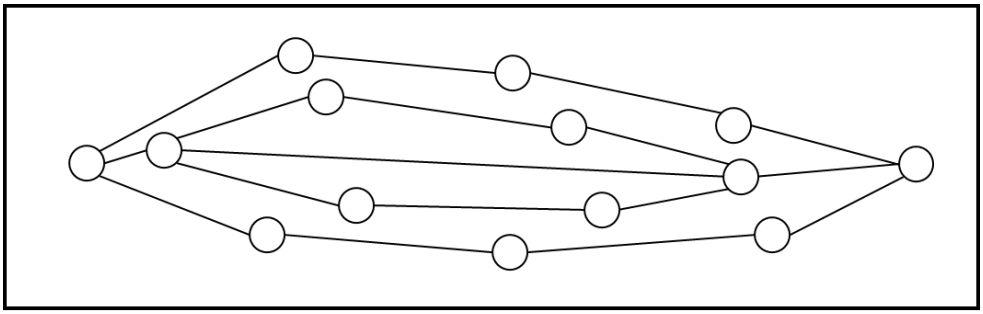
Let's return to undirected graphs. We mentioned that the directed case is more interesting, but that's not entirely true! Here we'll describe another aspect of connectivity in undirected graphs.

We say that a connected undirected graph is 2-**edge-connected** if removing any edge doesn't disconnect the graph. Alternatively, an undirected graph is 2-edge-connected if it is connected and doesn't have any bridges.

We say that a connected undirected graph is **biconnected** if removing any vertex doesn't disconnect the graph. Alternatively, an undirected graph is biconnected if it is connected and doesn't have any articulation points.

Note that these are stronger notions than mere connectivity. Having these properties tells us that the graph is more interconnected than usual.

Here's an example of a graph that's both 2-edge-connected and biconnected:



biconnected graph

The two notions are similar, but be careful not to confuse the two! They're not exactly the same. (Why?)

We can generalize the definitions above naturally:

- An undirected graph is k-**edge-connected** if removing less than k edges doesn't disconnect the graph.
- An undirected graph is k-**vertex-connected**, or k-**connected**, if it has more than k nodes and removing less than k vertices doesn't disconnect the graph.

Being biconnected and 2-connected are the same except that connected graphs of ≤ 2 nodes are considered biconnected but not 2-connected. Also, note that being 1-connected is the same as being connected (except when the graph has a single node).

3.1 Menger's theorem

There's a nice result concerning k-edge-connectivity and k-vertex-connectivity called **Menger's theorem**. The theorem provides two results:

1. Let x and y two *distinct* vertices. Then the minimum number of *edges* whose removal disconnects x and y equals the maximum number of pairwise *edge*-independent paths from x to y.

2. Let x and y two *distinct, nonadjacent* vertices. Then the minimum number of *vertices* whose removal disconnects x and y equals the maximum number of pairwise *vertex*-independent paths from x to y.

Aren't these results nice? Even nicer, they hold for both directed and undirected graphs!

In particular, they imply that:

1. An undirected graph is k-edge-connected if and only if for every pair of vertices x and y, it is possible to find k edge-independent paths from x to y.

2. An undirected graph is k-vertex-connected if and only if for every pair of vertices x and y, it is possible to find k vertex-independent paths from x to y.

We won't be proving these properties for now, but at least you know them; they could be useful.

There are other similar results like that all throughout graph theory, such as the **min-cut max-flow theorem** (the minimum cut equals the maximum flow) and **König's theorem** (the size of the minimum vertex cover equals the size of the maximum matching in bipartite graphs).[20]

3.2 Robbins' theorem

Another interesting fact about 2-edge-connected graphs is that *you can orient*[21] *the edges of the graph such that it becomes strongly connected*. Such graphs are called **strongly orientable**. In fact, the converse is true as well: every strongly orientable graph is 2-edge-connected. This equivalence is called **Robbins' theorem** and is not that difficult to prove.

If a graph has a bridge, then obviously a strong orientation is impossible. But if a graph has no bridges, then let's perform a DFS and orient the edges according to the way we traversed it; i.e., tree edges point away from the root, and back edges point up the tree. (Remember that there are no forward and cross edges since this is an undirected graph.) Since there are no bridges, it means that for every tree edge (a, b) in the DFS forest, $\texttt{low}[b] \leq \texttt{disc}[a]$. This means that from any node b, one can always climb to an ancestor of b (using at least one back edge). By

[20] Unsurprisingly, these results are all related.

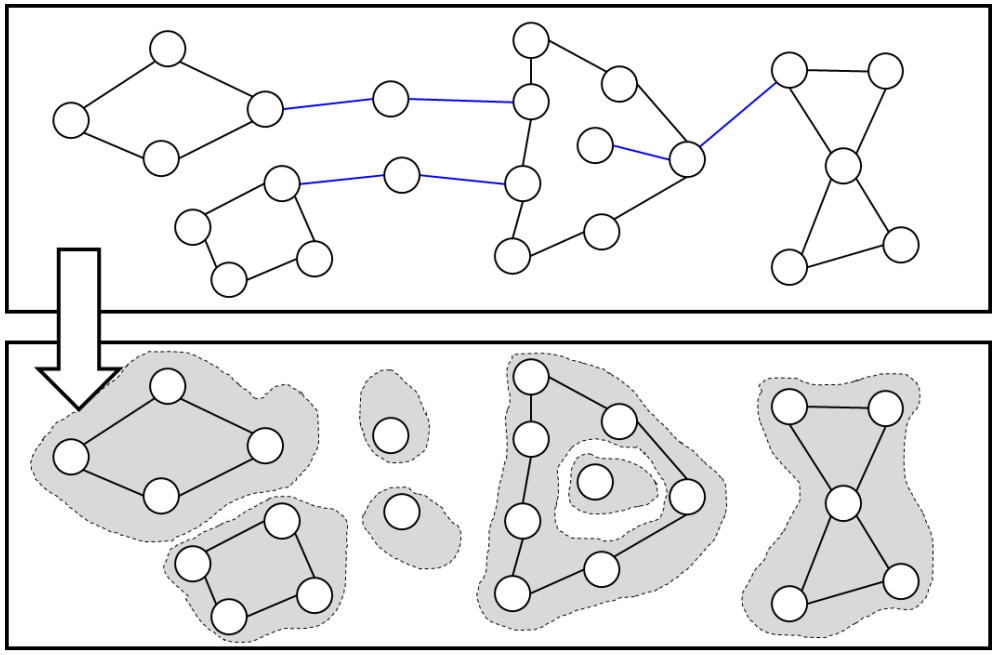
[21] To "orient" an edge means to choose its direction, hence the edge becomes directed.

repeating this process, one can reach the root from any node b. Since the root can reach all the other nodes as well (just using tree edges), it means the graph is strongly connected! As a bonus: this DFS-based proof can easily be converted into a DFS-based algorithm to compute a strong orientation of the graph.

3.3 2-edge-connected components

A 2-**edge-connected component** is a maximal subgraph that is 2-edge-connected. Given a graph, a natural question is: How can we find the 2-edge-connected components, and what can we say about their structure? Let's assume the graph is connected for simplicity; we can simply do the same algorithm for each connected component otherwise.

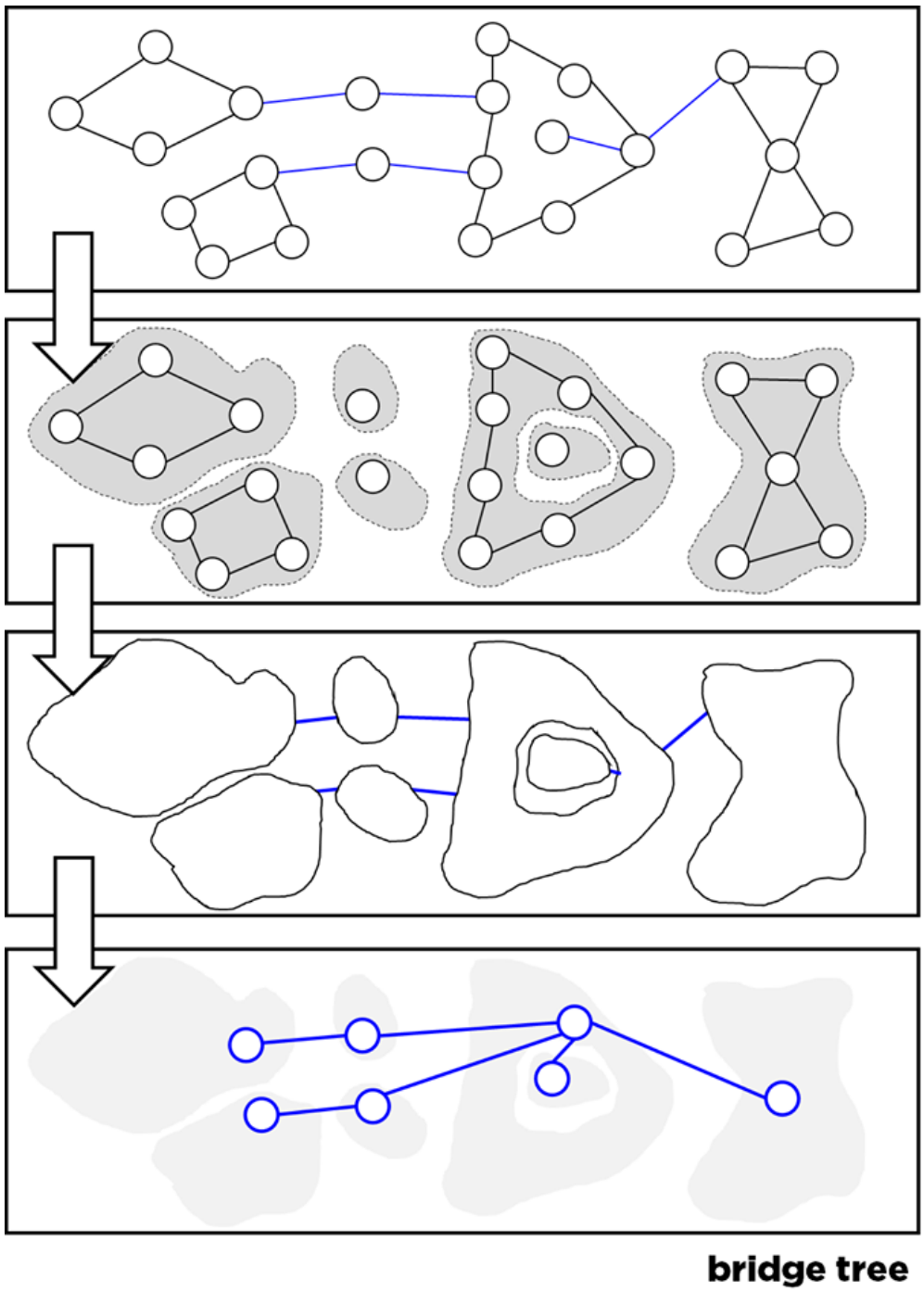
Well, by definition, a 2-edge-connected component cannot have bridges, so let's say we remove the bridges in the original graph first. (We can find those with DFS.) Then what we're left with are subgraphs that don't contain any bridges, hence are 2-edge-connected![22]



Furthermore, suppose we "shrink" each 2-edge-connected component into a single node. Then, observing that a bridge is not a part of any cycle (otherwise it couldn't have been a bridge at all), we find that the resulting graph is actually a *tree*!

[22] Actually, it doesn't follow from the definition that removing bridges doesn't introduce new bridges, but it shouldn't be hard to convince oneself of this.

We can call this resulting tree the **bridge tree** of the graph.



As just described, constructing this tree is straightforward:

1. Detect all the bridges.
2. Remove (burn) all the bridges

3. "Shrink" the remaining connected components into single nodes.

4. Put the bridges back.

5. The resulting graph is the bridge tree.

Like before, the "shrinking" process might be too vague, but an engineer approach will work just as well:

1. Detect all the bridges.

2. Remove all the bridges temporarily; store them in an array for now.

3. Collect all connected components. Assume there are k connected components.

4. Construct an array f, where $f[i]$ denotes the index of the connected component containing i.

5. Construct a graph with k nodes and initially 0 edges.

6. For every bridge (a, b), add the edge $(f[a], f[b])$ to the graph.

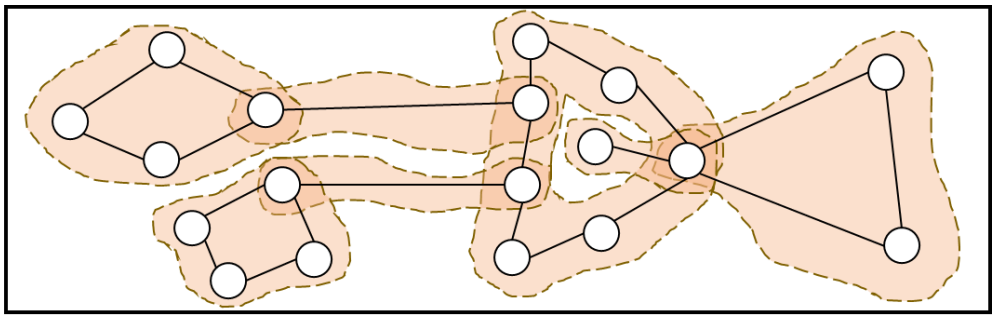
7. The resulting graph is the bridge tree.

Congratulations! You have now constructed the bridge tree and you may now use it to solve some problems.

3.4 Biconnected components

A **biconnected component**, or **BCC**, is a maximal subgraph that is biconnected. The natural question is: How can we find the BCCs, and what can we say about their structure? Again, assume the graph is connected for simplicity.

The structure of the BCCs of a graph is quite different from the structure of the 2-edge-connected components. In particular, BCCs can overlap! See the following picture:



overlapping BCCs

So given this complication, how can we compute the BCCs?

The natural first step is to compute all the articulation points. After that, notice that pairs of BCCs can only overlap on at most one node, and that node must be an articulation point.

However, it looks like we're stuck. Even given that information, there doesn't seem to be a simple way to compute the BCCs.

Fortunately, we can actually modify DFS (again!) to compute the BCCs *alongside* the articulation points! The key here is to think of a BCC as a set of *edges*, rather than a set of nodes; this way, each edge belongs to exactly one BCC, which is very helpful. Furthermore, similar to Tarjan's SCC algorithm, we can again collect the *edges* that belong to the same BCC on a stack, and pop whenever we detect an articulation point!

```
function DFS(i, p):
   disc[i] = low[i] = time++

   children = 0
   has_low_child = false
   for j in adj[i]:
      if disc[j] == -1:
         stack.push(edge(i, j))
         DFS(j, i)

         low[i] = min(low[i], low[j])
         children++

         if low[j] >= disc[i]:
            has_low_child = true
            get_bcc(edge(i, j))

      else if j != p:
         low[i] = min(low[i], disc[j])

   if (p == -1 and children >= 2) or (p != -1 and has_low_child):
      mark i as an articulation point

function get_bcc(e):
   // pop the stack until you pop e, and collect those as a BCC
   bcc = new empty vector
   do:
      E = stack.pop()
      bcc.push(E)
   while E != e
   bccs.push(bcc)

function articulation_points_and_BCCs():
   stack = new empty vector
   bccs = new empty vector of vectors
   time = 0
   for i = 0..n-1:
      disc[i] = -1

   for i = 0..n-1:
      if disc[i] == -1:
         DFS(i, -1)
   return bccs
```

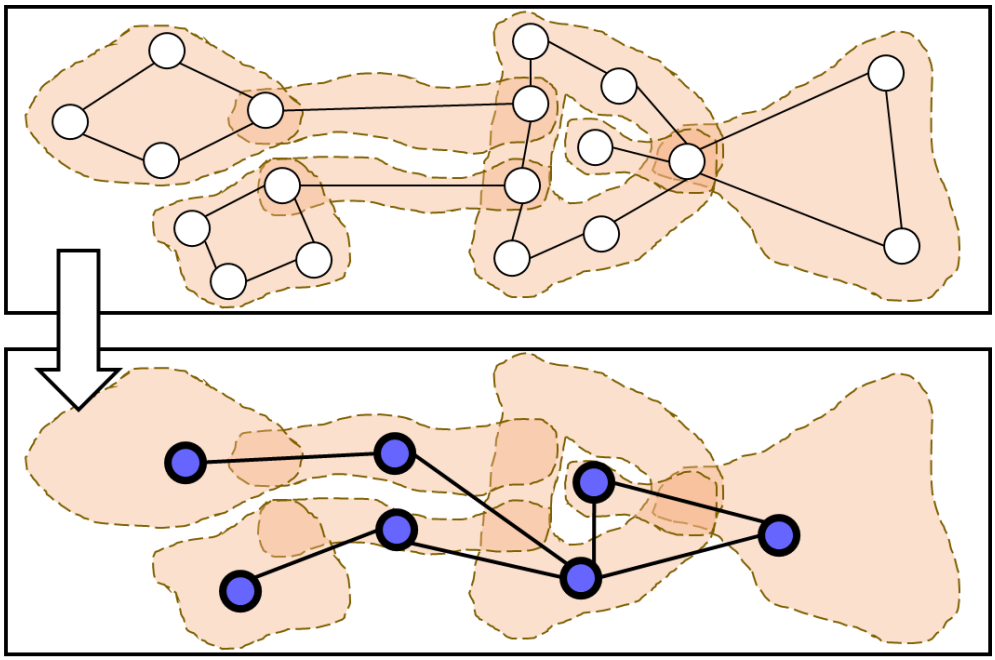
After this $O(n+e)$ process, we now have the articulation points and the BCCs of the graph!

3.4.1 Block graph

The next natural question is: what structure do the BCCs have? Remember that they can only overlap in at most one node, and this must be an articulation point. Therefore, the articulation points somehow serve the role of "edges" in the same way the bridges were the "edges" in the bridge tree.

The natural graph structure we can form, then, is to compress each BCC into a single node, and declare that two BCCs are adjacent iff they share an articulation point in common. This is called the **block graph**, and it's easy to see that this forms a valid connected graph structure on the BCCs, and that it encodes a bit of information about the connectivity of the graph as a whole. Constructing the block graph from this definition is straightforward.[23]

Here's an example of a block graph:



block graph

Unfortunately, as you can see above, the block graph is not (always) a tree! That's sad :(And that's the reason it's not called a "block tree".

In fact, what's even sadder is that this can fail very badly. Specifically, a block graph formed from a graph with n nodes and $O(n)$ edges can have up to $\Omega(n^2)$ edges![24] So that's really sad.

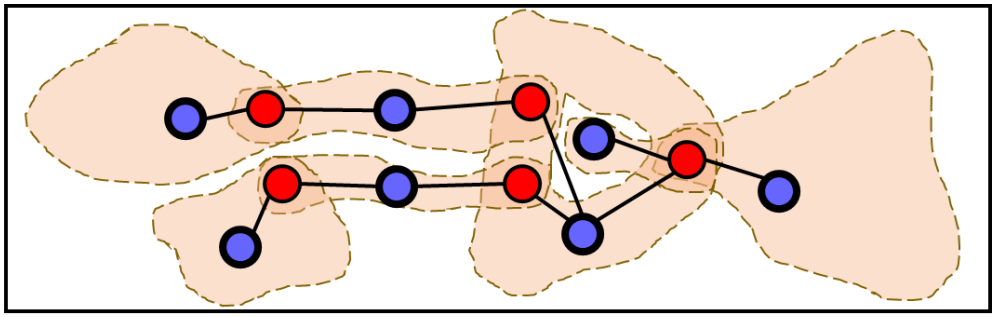
[23] Use the engineer approach.

[24] $f(n) = \Omega(g(n))$ is the same as $g(n) = O(f(n))$. Informally, you can think of O as "asymptotically at most" and Ω as "asymptotically at least".

3.4.2 Block-cut tree

Fortunately, there's an alternative structure on the BCCs where the number of edges doesn't explode. The key is to *represent the articulation points* as nodes in their own right. Thus, in our compressed graph, we have a node a_x for each articulation point x and a node b_Y for each BCC Y.[25] We say there is an edge between a_x and b_Y if the BCC Y contains the articulation point x. It can be seen that this structure is a tree (why?), and this is more commonly known as the **block-cut tree**. ("Block" stands for BCC and "cut" stands for cut vertex.)

Here's an example of a block-cut tree:



block-cut tree

Thankfully, the block-cut tree is indeed a tree, thus it doesn't have that many more nodes and edges than the original one. And it still encodes a good amount of information about the connectivity between the BCCs. As an added bonus, the articulation points are represented as well!

Constructing a block-cut tree from definition is straightforward.[26]

[25] Note that this "compressed" graph can actually be larger than the original graph!
[26] Use the engineer approach.

4 Problems

I realize the problems here are tougher than usual; the general rule here is to solve as many as you can!

4.1 Warmup coding problems

Most of these are straightforward applications of some of the algorithms discussed. Not required, but you may want to use these problems to test your implementations of the algorithms above.

1. **Test**: UVa 10731
2. **Tourist Guide**: UVa 10199 (implement a linear-time solution)
3. **Network**: UVa 315 (implement a linear-time solution)
4. **Lonely People in the World**: https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/lonely-people-in-the-world
5. **Putogethers**: https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/putogethers

4.2 Coding problems

Here are the required problems:

1. **Checkposts**: http://codeforces.com/problemset/problem/427/C
2. **Dominos**: UVa 11504
3. **Cutting Figure**: http://codeforces.com/problemset/problem/193/A ($O(nm)$ required)
4. **Chef Land**: https://www.codechef.com/problems/CHEFLAND
5. **Sub-dictionary**: UVa 1229 ($O(n+e)$ required)
6. **Proving Equivalences**: UVa 12167
7. **Sherlock and Queries on the Graph**: https://www.hackerrank.com/contests/101hack26/challenges/sherlock-and-queries-on-the-graph
8. **Mr. Kitayuta's Technology**: http://codeforces.com/problemset/problem/505/D

9. **Case of Computer Network**: http://codeforces.com/contest/555/problem/E
10. **Tourists**: http://codeforces.com/contest/487/problem/E
11. **Chef and Sad Pairs**: https://www.codechef.com/problems/SADPAIRS

Please feel free to ask if some of the problems are unclear.

4.3 Non-coding problems

No need to be overly formal in your answers; as long as you're able to convince me, it's fine!

If any definition is unclear, please ask.

1. In an undirected graph, why is a back edge never a bridge?
2. Prove or disprove: an undirected graph is acyclic if and only if every edge is a bridge.
3. Prove or disprove: an undirected graph is acyclic if and only if every node of degree > 1 is an articulation point.
4. Prove or disprove: each endpoint of a bridge is either of degree 1 or an articulation point.
5. Prove or disprove: every articulation point is an endpoint of some bridge.
6. An **arborescence** is a directed graph that is formed by starting with an undirected tree, choosing some node to be a root, and *orienting* the edges to point away from the root. A **polytree** is a directed graph such that if you ignore the directions of the edges, you get an undirected tree. Prove or disprove: every arborescence is a polytree.
7. Prove or disprove: every polytree is an arborescence.
8. A **multitree** is a directed graph such that if you pick any node t, and remove all nodes and edges not reachable from t, you get an arborescence. Prove or disprove: every polytree is a multitree.
9. Prove or disprove: every multitree is a polytree.
10. Prove that the bogo-cycle-finding algorithm correctly computes s_{cycle}, l_{cycle} and l_{tail}. Determine its running time.
11. Prove that Floyd's cycle-finding algorithm correctly computes s_{cycle}, l_{cycle} and l_{tail}.
12. Prove that Floyd's cycle-finding algorithm takes $O(l_{\text{tail}} + l_{\text{cycle}})$ time.
13. Prove or disprove: Every 2-edge-connected graph is biconnected.

14. Prove or disprove: Every biconnected graph is 2-edge-connected.

15. Show that the block graph formed from a graph with n nodes and $O(n)$ edges can have $\Omega(n^2)$ edges.

16. The k-**SAT** problem, or k-**satisfiability**, is the problem of determining whether one can assign truth values to a set of boolean variables $x_1, x_2, \ldots, x_n$ so that a given boolean formula of the form[27]

$$\underbrace{(\mathbf{x} \vee \mathbf{x} \vee \ldots \vee \mathbf{x})}_{k} \wedge \underbrace{(\mathbf{x} \vee \mathbf{x} \vee \ldots \vee \mathbf{x})}_{k} \wedge \ldots \wedge \underbrace{(\mathbf{x} \vee \mathbf{x} \vee \ldots \vee \mathbf{x})}_{k},$$

    where each $\mathbf{x}$ is either x_i or $\neg x_i$ for some i, evaluates to true. The "k" denotes the length of each term, i.e., the number of $\mathbf{x}$'s.

    For any k, a straightforward $O(2^n m)$-time algorithm for k-SAT exists, where m is the length of the formula. (What's that algorithm?)

    3-SAT (and above) is known to be **NP-complete**, thus there's no known solution for them that runs in polynomial time[28] in the worst case. On the other hand, show that 1-SAT and 2-SAT are solvable in $O(n + m)$ time by providing an algorithm. Also, show that you can determine one such assignment in $O(n + m)$ time. Bonus: Implement them!

17. A **tournament graph** is a directed graph that is obtained by *orienting* the edges of an undirected *complete* graph. In other words, if you ignore the directions of the edges, you get a graph where each pair of nodes is connected by exactly one edge.

    You can think of a tournament graph as the results of all rounds in a round-robin tournament with n participants, and an edge $a \to b$ means "a won in a round over b". Such a graph is not necessarily transitive; sometimes a lower-skilled participant beats a higher-skilled participant.

    A **Hamiltonian path** is a path that passes through all vertices exactly once. In the context of a tournament, a Hamiltonian path is a possible "ranking" of the participants.[29]

    Prove that any tournament graph has a Hamiltonian path. (Hint: Use induction, and notice that removing a single node v yields a smaller tournament graph. Where do you insert v in the Hamiltonian path of the smaller graph?)

18. Describe an algorithm that finds a Hamiltonian path in a tournament graph in $O(n^2)$ time. (Hint: use your proof above.) Bonus: Implement it!

19. Suppose you have a tournament graph, but you don't (initially) have access to its edges. However, you can *query* for the edges, i.e., you are provided with a function $f(a, b)$ which

[27] $(a \vee b)$ means "a *or* b", $(a \wedge b)$ means "a *and* b" and $\neg a$ means "not a".
[28] To be more precise, *deterministic polynomial time*.
[29] There are other (perhaps better) ways to rank participants, such as ranking by the number of wins.

returns true if $a \to b$ is an edge and false otherwise.[30] Assume a single call to $f(a, b)$ runs in $O(1)$ time. Describe an algorithm that finds a Hamiltonian path in the tournament graph that uses only $O(n \log n)$ calls to f. (An $O(n^2)$ *running time* is acceptable.) Bonus: Implement it!

20. Improve the algorithm above to $O(n \log n)$ time. Bonus: Implement it!

21. A **Hamiltonian cycle** is a cycle that passes through all vertices exactly once. It's easy to see that any graph with a Hamiltonian cycle is strongly connected. Prove that a strongly connected tournament graph always has a Hamiltonian cycle. (Hint: Start with a cycle and try growing it.)

22. Describe a polynomial-time algorithm that finds a Hamiltonian cycle in a strongly connected tournament graph.

23. Given some tournament graph G, prove that the following things are equivalent:

    (a) G is transitive, i.e. if $a \to b$ and $b \to c$ are edges, then $a \to c$ is an edge as well.

    (b) G is acyclic.

    (c) The outdegrees of G's nodes are distinct. (In fact, they form the set $\{0, 1, \ldots, n-1\}$)

    (d) G has a *unique* Hamiltonian path.

    (e) G has no cycles of length 3.[31]

[30] Obviously, if $a \neq b$, exactly one of $f(a, b)$ and $f(b, a)$ is true.

[31] In general graphs, this is a weaker condition than being acyclic, but in tournament graphs it turns out that they're equivalent, i.e., if there are no cycles of length 3, then there are no cycles at all.

4.4 Bonus problems

Here are bonus problems. Solve them for extra credit.

1. In the DFS pseudocode above for undirected graphs, why is it that if we don't check the condition `j != p`, all edges will be marked back edges?

2. Show that there are no forward and cross edges in a DFS forest of any undirected graph.

3. Show that there can't be any two nodes i and j such that

   $$\texttt{start\_time}[i] < \texttt{start\_time}[j] < \texttt{finish\_time}[i] < \texttt{finish\_time}[j]$$

   in the DFS for both directed and undirected graphs. (In other words, if the visiting intervals of i and j intersect, then one must contain the other.)

4. Exactly how many calls to f are done by the bogo-cycle-finding algorithm as written above, in terms of l_{tail} and l_{cycle}?

5. Exactly how many calls to f are done by Floyd's cycle-finding algorithm as written above, in terms of l_{tail} and l_{cycle}?[32]

6. In the pseudocode for Tarjan's SCC algorithm, is the line `low[i] = min(low[i], disc[j])` being run only when $i \to j$ is a back edge? Is it being run on forward edges as well? What about cross edges? If yes on either, does that mean Tarjan's algorithm is incorrect? Why or why not?

7. Implement an iterative version of Tarjan's SCC algorithm.[33]

8. Show that removing a bridge doesn't turn any non-bridge edge into a bridge.

9. Show that the block-cut tree is indeed a tree.

10. Show that the block-cut tree can have both more nodes and edges than the original graph.

11. Let G be a graph with $n > 0$ nodes and $e > 0$ edges. Suppose its block-cut tree has n' nodes and e' edges. Find an upper bound on the ratios n'/n and e'/e. Also, what are the tightest possible upper bounds? Show why your upper bounds and least upper bounds are correct.[34]

[32] This will verify that Floyd's algorithm indeed runs in $O(l_{\text{tail}} + l_{\text{cycle}})$, and will in fact show the constant hidden in the O notation. On the other hand, minimizing this constant is important in other applications, e.g., if computing f isn't $O(1)$ anymore; one example that improves upon Floyd's algorithm's constant is **Brent's algorithm**.

[33] Once you have this, it's easy to write the other `low-disc` algorithms iteratively as well. Save your code; it'll be useful if the online judge has a small stack.

[34] Hint for those who aren't familiar with proving least upper bounds: Suppose you want to show that U is a least upper bound to n'/n. Then you have to show that (a) U is an upper bound, and (b) n'/n can get *arbitrarily close* to U. You can do the latter by exhibiting an infinite family/sequence of graphs where n'/n approaches U as n gets large.

12. Show that for two nodes x and y in a tournament graph, the following are equivalent:

    (a) x and y are strongly connected.

    (b) There is a cycle containing x and y.[35]

    (c) There is a Hamiltonian path where x comes before y, and another Hamiltonian path where y comes before x.

    Hint: use your result from 21.

13. Does the previous statement hold for general directed graphs? If not, show which implications are true and which implications are false, and show why they are so. (There are 6 implications in total.)

14. Given two nodes x and y in a tournament graph such that there is a path from x to y, describe an $O(n^2)$ algorithm that finds a Hamiltonian path where x comes before y.

15. A **clique** of an undirected graph is a subset of its nodes where each pair is connected by an edge. In other words, a clique is a subset of the nodes that induces a complete subgraph.

    Consider the problem of finding the maximum clique in a graph. For general graphs, we don't (yet) know of any polynomial-time solution. On the other hand, for a given graph G with n nodes and e edges, show that one can find the maximum clique in the *block graph of* G in $O(n+e)$ time. (Be careful: even if $e = O(n)$, the block graph can have $\Omega(n^2)$ edges!)

16. Let G be an undirected graph with n nodes and e edges. Suppose G is the block graph of some graph. Show how to find the maximum clique in G in $O(n+e)$ time. (Note: you don't know which graph has G as its block graph.)

17. Find a tournament graph that has ≤ 75 nodes and *exactly* $2^{64}-1$ Hamiltonian paths.[36]

Acknowledgment

Thanks to Jared Asuncion for the images and the initial feedback!

[35] Note: a cycle cannot contain duplicate nodes.
[36] Of course, show that your answer is correct.

NOI.PH 2017 Training Week 9

Kevin Charles Atienza

April 2017

Contents

1 Introduction

The in-house training and the selection exam are fast approaching! It would be best to focus the last few days consolidating and applying what you've learned. This week, we will prepare you for the in-house training which will consist mostly of practice/real contests.

I painstakingly compiled 8 four-problem sets, requiring the topics we've covered, for you to practice on this week. I tried to evenly distribute the difficulty and topics, although don't expect it to be *perfect*; some problem sets are probably more data-structure-heavy, others more trick-based, and some sets are probably easier than others.

Although you may solve the problems in any order you want, it is recommended to solve them *per problem set*, and to solve each problem set as a group.

1.1 Scoring

Each problem is worth 1 point, regardless of difficulty. Completing a problem set nets you an additional 1 point. Completing a problem within its target date nets you some bonus.

The score for this week is determined by the number of points over $\max(30, \text{maxscore})$, where maxscore is the maximum score of any participant, not counting the target bonus.

Please don't waste these prepared problem sets; use them as real practice. Your goal is to maximize your chances to get a medal in the IOI!

1.2 Recommendation

Each set is roughly equivalent to a short contest: 3-5 hours. Although you don't have to follow that strictly, I recommend trying out each set as a single round on its own and timing yourself. (If you do this, please let us know, and also please tell us how you did!) This will give you a rough estimate on how well you could do at the IOI: note that the problems in the IOI will be harder on average. But if you don't meet your set target time, please continue trying to solve them anyway to get points; remember that a full set gives you an extra point.

I *highly* recommend being ahead by 2 or 3 problem sets every day, so if you don't finish a problem set, you'll have time to spare to (maybe) try to get that 3rd or 4th problem (and thus the extra

point) by the target date.

Feel free to ask for advice or hints. (Yes, I'll give hints for some problems.)

2 The problem sets

2.1 Problem set HA

Target date: **May 4**

1. Edges in MST: http://codeforces.com/contest/160/problem/D
2. Chef and Prime Divisors: https://www.codechef.com/problems/CHAPD
3. Kth Max Subarray: https://www.codechef.com/problems/KTHMAX
4. Across the River: https://www.codechef.com/problems/RIVPILE

2.2 Problem set JA

Target date: **May 5**

1. Lighthouses: https://www.codechef.com/problems/LIGHTHSE
2. Xor-tree: http://codeforces.com/contest/429/problem/A
3. Product of Diameters: https://www.codechef.com/problems/TREEDIAM
4. Range Minimum Queries: https://www.codechef.com/problems/ANDMIN

2.3 Problem set KA

Target date: **May 6**

1. Oracle Devu and Longest Common Subsequence: https://www.codechef.com/problems/ORACLCS
2. BerSU Ball: http://codeforces.com/contest/489/problem/B
3. Yet Another SubSegment Sum Problem: https://www.codechef.com/problems/SEGSUMQ
4. Möbius function and intervals:[1] https://projecteuler.net/problem=464

[1] Don't worry, this is more algorithmic than mathematical! We've covered everything you need.

2.4 Problem set KS

Target date: **May 7**

1. Nice SubSegments: https://www.codechef.com/problems/SUBSGM
2. Sebi and the corrupt goverment:[2] https://www.codechef.com/problems/SETELE
3. Guessing Game: https://projecteuler.net/problem=406
4. Island Puzzle: http://codeforces.com/problemset/problem/627/F

2.5 Problem set PY

Target date: **May 8**

1. Uphill paths: https://projecteuler.net/problem=411
2. Chef and Reversing: https://www.codechef.com/problems/REVERSE
3. Time Travelling Monster: https://www.codechef.com/problems/TIMETRAV
4. Merciless Chef: https://www.codechef.com/problems/MLCHEF

2.6 Problem set TD

Target date: **May 9**

1. Superinteger: https://projecteuler.net/problem=467
2. Niceness of a tree: https://www.codechef.com/problems/NICENESS
3. Binary Circles: https://projecteuler.net/problem=265
4. Perfect Subarrays: https://www.codechef.com/problems/SUBARR

[2] "Expected value", in this case, is simply the average. For example, the expected value of throwing a dice is the *average across all events*, i.e., $\frac{1+2+3+4+5+6}{6} = 3.5$. This works because all events are equally likely; in other cases, you should weigh the events by their probability.

2.7 Problem set TR

Target date: **May 10**

1. Forest Gathering: https://www.codechef.com/problems/FORESTGA
2. New Year Permutation: http://codeforces.com/contest/500/problem/B
3. Almost Union-Find: https://uva.onlinejudge.org/external/119/11987.pdf (UVa 11987)
4. Line Intersections: https://www.codechef.com/problems/CHN15D

2.8 Problem set VG

Target date: **May 11**

1. Roses for Alexey: https://www.codechef.com/problems/ALEXROSE
2. Parity tree: https://www.codechef.com/problems/PARITREE
3. Devu and Manhattan Distance: https://www.codechef.com/problems/MDIST
4. Tree: https://www.codechef.com/problems/RRTREE

2.9 Notes

For Project Euler problems, consider your code accepted if it can *compute* the answer in less than 10 seconds, and by that, I mean actually computing the answer, not just hardcoding $+$ printing it (or something of similar "cheaty" flavor).

For problems with subtasks, partial solutions will get partial points, although you'll only get the extra 1 point per set if you get full points for all 4.