Lecture Notes

Michael Levet

June 27, 2020

Contents

1 Hello, World

1.1 Compiling and Running Your First Java Program

In this section, students will compile and execute their first Java program. The goals are the following:

- To provide students practice using the command prompt to navigate the file directory, as well as compile and run Java programs.
- To ensure that the Java Development Kit (JDK) is correctly installed and configured for use with the command line.

In order to run your first Java program, do the following. **Please ask if anything is unclear.**

(a) Open a new, blank document in Notepad++. Save the file as `HelloWorld.java` (make sure to select the `.java` file extension); the name of your class will be `HelloWorld`. **Warning:** Java is case-sensitive.

In any `.java` file, there can be at most one `public` class, interface, or enum. Furthermore, the name of the `public` class, interface, or enum in the file **must** match the name of the file (excluding the .java extension).

(b) In your file, type the following Java program. This is the `Hello, World` program. We will discuss the code sample in more detail later.

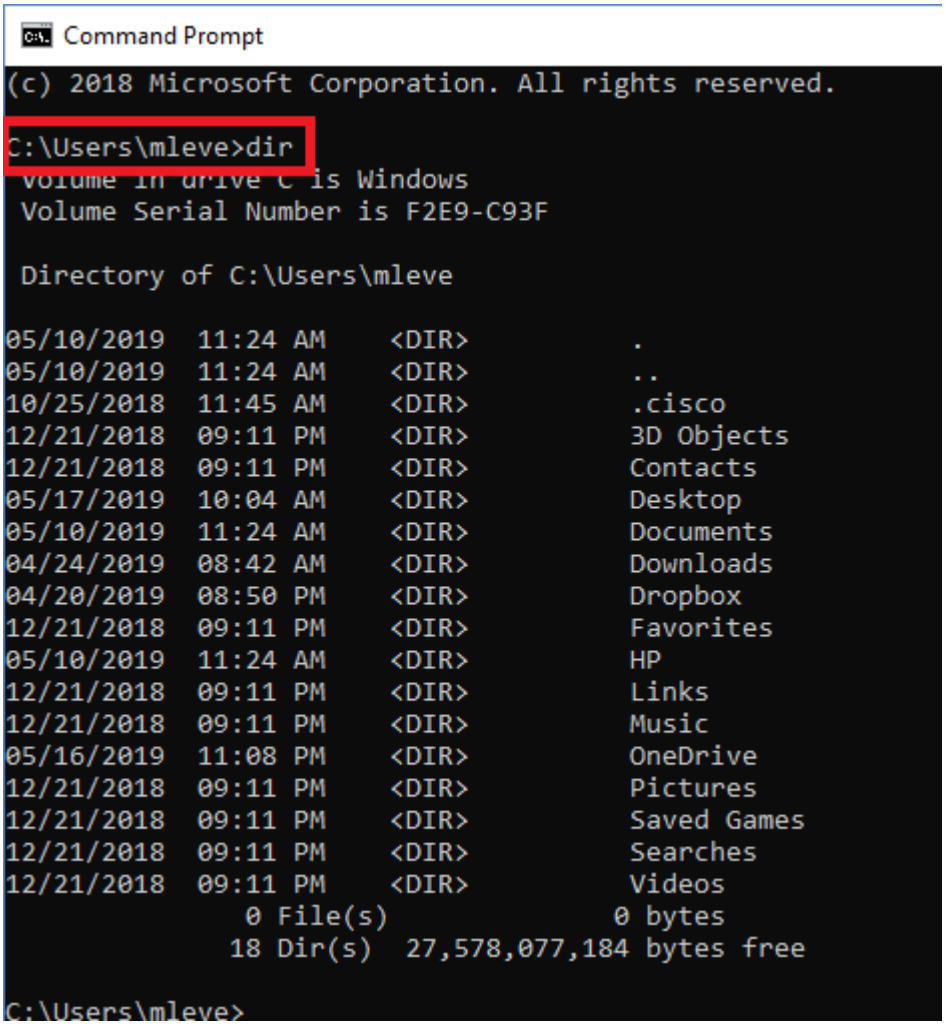
```
public class HelloWorld{

    public static void main(String[] args){
        System.out.println("Hello, world!");
    }
}
```

(c) Open up the command line (from the Start Menu, select `cmd.exe`). Using the command line, you will need to navigate to the folder where your `HelloWorld.java` file is stored. The `dir` and `cd` commands will be particularly helpful.

- The `dir` command lists all the files and folders in the current directory. To use the `dir` command, type

  `dir`

  into the command prompt and then hit Enter.



- The `cd` command is used to change directories. In order to enter a folder contained in the current directory, use the command `cd FolderName`. In the image above, observe the `Documents` folder is contained in my current directory. So the command

  ```
  cd Documents
  ```

  will move me into the documents folder. To return to the previous directory, type

  ```
  cd ..
  ```

  including the two periods.

(d) Once you have accessed the directory containing your `HelloWorld.java` file, it is time to compile and run your program.

- You first need to compile your program. To do this, type

  ```
  javac HelloWorld.java
  ```

- Once you compile your program, you may run it using the command:

  ```
  java HelloWorld
  ```

If you are successfull, then you should see `Hello, World!` displayed on your command prompt window.

1.2 Hello, World- Unpacking the Pieces

In this section, we will more closely examine the `Hello, World` program. For reference, recall the `Hello, World` program listed below.

```
public class HelloWorld{

    public static void main(String[] args){
        System.out.println("Hello, world!");
    }
}
```

First, observe that there are several words in blue: `public` , `class` , `static` , and `void` . These are special keywords in the Java language, known as *reserved words*. Each of these keywords has significance in the Java programming language. Each of these reserved words will be discussed in more detail later in the course. There are some key points worth discussing.

- A class is the primary container used to organize blocks of code. Note that any code belonging to a class is contained within the curly braces after its name. Later in the course, we will discuss *how* to effectively utilize classes to organize our code with Object-Oriented Programming, in which classes are used to model real-world objects.
- Next we have the main method:

  `public static void main(String[] args).`

  The main method is the starting point of any Java program. The code inside of the main method is executed immediately at the start of the Java program. Each statement inside the main method is executed, one at a time, in the order they are listed.
- Lastly, we examine the following statement more closely:

  `System.out.println("Hello, world!");`

  The `System` class is a part of the standard Java library, which contains a variable named `out` that is capable of writing to the console. The `println` method writes text to the console and then moves to the next line. So any text written after a `System.out.println` statement begins on the next line.

  Now `"Hello, world!"` is a String, which we can think of as plain text or words. The double quotes surrounding the text tell Java to treat everything in-between as just plain text with no special significance to the program.

2 Variables

In this section, we will introduce the notion of a variable. In the context of math classes, variables are frequently used to represent *solutions* when solving equations or as *placeholders* for functions (e.g., for $f(x) = x^2$, the variable x is a placeholder for the input). In the context of computer programming, variables act as *buckets* rather than *solutions*. That is, a variable is a container which stores a sheet of paper on which a value is written. A key difference between math and computer programming is that when programming, the value in a given variable can be changed. In fact, we will frequently update the values in our variables, the more code we write. In this manner, variables become instrumental in even the most basic tasks, such as tracking values that have been counted or storing the sum of student grades so that we may compute the average.

2.1 Declaring and Using Variables

We next discuss how to declare and use variables. Each variable has three components: a datatype, a name, and a value. In Java, there are two sets of datatypes: primitives and Objects[1]. There are eight primitive datatypes, which we shall introduce later. For now, we restrict attention to the `int`datatype to focus on utilizing variables. We note that `int`is a primitive datatype used to store integers.

The format for declaring and initializing a variable is as follows:

```
type name = value;
```

So to declare an int with value 0, the following syntax would be used.

```
int x = 0;
```

Here, `x` is the name of our variable, which is of type `int` and is initialized to store the value 0.

Recall that the value stored in a variable can be changed. Consider the following code sample.

```
int x = 0;
x = 5;
```

Here, the first line declares a variable named `x` and assigns it the value 0. The second line replaces the value 0 initially stored in `x` with the new value 5.

So far, we have only seen examples where there is an actual number to the right of the `=` operator. The right-hand side can contain any expression, involving numbers or other variables, so long as the resulting value is of the same type as the variable. We consider an example.

```
int y = 3;
int z = 5;
z = y + 7;
y = y + 5;
```

At the third line, `y` has the value 3, so `y+7` evaluates to 10. Hence, `z` is assigned the value 10. On the fourth line, the expression `y+5` is first evaluated. As `y` holds the value 3 at this point, `y+5` evaluates to 8. So `y` is assigned the value 8.

Aside from addition, Java also provides operators for subtraction, multiplication and division. We discuss these operations. Moving forward, we assume that we have a variable named `y` of type `int`.

- **Addition:** We have already seen that the `+` operator used for addition. There is also the `+=` operator, which is used to update a variable. In particular:

  ```
  y += 5;
  ```

  is equivalent to:

  ```
  y = y + 5;
  ```

- **Subtraction:** The `-` operator is used for subtraction. Java also provides the `-=` operator, which is analogous to the `+=` operator. In particular:

  ```
  y -= 5;
  ```

  is equivalent to:

  ```
  y = y-5;
  ```

[1]Note that Object has a capital *O* in Java, as Object is the class from which all other classes inherit. We will discuss this more later in the course.

- **Multiplication:** The `*` operator is used for multiplication. Java also provides the `*=` operator, which is analogous to the `+=` operator. In particular:

  ```
  y *= 5;
  ```

  is equivalent to:

  ```
  y = y * 5;
  ```

- **Division:** The `/` operator is used for division. Java also provides the `/=` operator, which is analogous to the `+=` operator. In paritcular:

  ```
  y /= 5;
  ```

  is equivalent to:

  ```
  y = y / 5;
  ```

  Recall that, as `y` is of type `int`, the expression `y/5` must evaluate to an integer. For this reason, `y/5` always rounds down if `y` is not a multiple of 5. So for example, `7/5` evaluates to 1. Similarly, `5/3` evaluates to 1.

Provided below is a runnable sample of code (`VariablesDemo.java`).

```
public class VariablesDemo{

    public static void main(String[] args){
        int x = 3;
        int y = 4;

        x = x + y;
        System.out.printf("x = %d, y = %d\n", x, y);

        y *= 5;
        System.out.printf("x = %d, y = %d\n", x, y);

        x -= 1;
        System.out.printf("x = %d, y = %d\n", x, y);

        y /= x;
        System.out.printf("x = %d, y = %d\n", x, y);
    }
}
```

The output of this program is as follows:

```
x = 7, y = 4
x = 7, y = 20
x = 6, y = 20
x = 6, y = 3
```

2.1.1 Exercises

(Required) Problem 1. Suppose a customer purchases \$18.83 in groceries and hands the cashier a \$20 bill. The cashier does not have any \$1 bills and so must make change using only quarters (worth \$0.25), dimes (worth \$0.10), nickels (worth \$0.05), and pennies (worth \$0.01). Write a Java program to determine the minimum number of coins needed in order for the cashier to provide change.

- Your program should still work, regardless of the amount for which change is needed (e.g., even if the amount is changed from \$1.17).
- You may assume the denominations (quarters, dimes, nickels, and pennies) remain the same.

Hint: Can you determine how many quarters are needed?

(Required) Problem 2. The *greedy* approach to solving the change problem examines the coin denominations from most valuable to least valuable, in an attempt to use as many of the most valuable coins as possible without exceeding the remaining amount. Suppose for instance, if our denominations are quarters, dimes, nickels, and pennies (worth \$0.25, \$0.10, \$0.05, and \$0.01, respectively) the algorithm would start by using as many quarters as possible without exceeding the amount owed to the customer. Next, the algorithm would use as many dimes as possible, followed by nickels, and then pennies.

We say that a coin system is *canonical* if the greedy algorithm will always make change using the fewest number of coins possible. In such a case, we say that the greedy algorithm is *optimal*. The United States coin system of quarters, nickels, dimes, and pennies is canonical. However, not every coin system is canonical.

Your job is as follows. For each of the following coin systems, find an amount of money for which the greedy algorithm is not optimal. Then provide a construction making change for your chosen amount using fewer coins than the greedy algorithm.

(a) Make change for \$0.06, using coins worth \$0.04, \$0.03, and \$0.01.

(b) Make change for \$0.10, using coins worth \$0.07, \$0.05, and \$0.01.

(c) Make change for \$0.31, using coins worth \$0.25, \$0.15, and \$0.01.

(d) Make change for \$0.33, using coins worth \$0.25 and \$0.03. In this case, will the greedy algorithm even be able to make change? Justify your answer.

You are welcome and encouraged to write code to aid you in solving this problem. Should you choose to adapt your code from Problem 1, please **create a new Java class rather than modifying your existing .java file for Problem 1.**

Remark. The change-making problem is a known difficult problem, which has been the subject of significant research. In 1975, G.S. Lueker [12] proved that the change-making problem is NP-Complete; which means that for arbitrary coin systems, it is easy to verify a correct solution to the change-making problem but difficult to efficiently construct such a solution. In particular, an efficient algorithm to solve the change-making problem is unlikely to exist[2]. In 1994, David Pearson constructed an efficient algorithm that examines the coin denominations to determine whether they form a canonical coin system. That is, Pearson's algorithm determines whether the greedy algorithm is optimal for a given coin system [9].

[2]More precisely, unless P = NP, a polynomial time algorithm to solve the change-making problem does not exist. It is unknown whether P = NP; this remains the most important open problem in Computer Science.

2.2 Primitive Datatypes

In this section, we expand on the primitive datatypes, as well as provide look at some examples. The point of this section is to provide a reference for the remainder of the course.

- The `boolean` datatype, which allows for values of `true` and `false`.

  Consider the following example. The first line declares a `boolean` variable, initialized to hold the value `true`. The second line then updates the value in `x` to `false`.

  ```
  boolean x = true;
  x = false;
  ```

- The `char` datatype, which allows for a variable to store a single character, like a letter, number, or punctuation. The value of a `char` is surrounded by *single quotes*, whereas String literals are surrounded by *double quotes*. So for example, 'a' is a `char`, while "a" is a String.

  Consider the following example. The first line declares a `char` variable named `x` and assigns it the value `'a'`. The second line then updates `x` to hold the value `'A'`. Recall that Java is case-sensitive, so `'a'` and `'A'` are treated as different values.

  ```
  char x = 'a';
  x = 'A';
  ```

- The `byte`, `short`, `int`, and `long` datatypes all store integers. The difference between these four datatypes is their range.
  - A `byte` has 8 bits (or one byte) allotted and can store integers between and including -128 to 127.
  - A `short` has 16 bits (or two bytes) allotted and can store integers between and including -2^{15} to $2^{15} - 1$.
  - An `int` has 32 bits (or four bytes) allotted and can store integers between and including -2^{31} to $2^{31} - 1$.
  - A `long` has 64 bits (or eight bytes) allotted and can store integers between and including -2^{63} to $2^{63} - 1$.

  We remark that the -1 term in the upper bounds (e.g., $2^{31} - 1$ for `int`) comes from the fact that we have to count 0 as a possible value. By convention, the `int` datatype should be used unless one has a compelling reason. The `byte`, `short`, and `long` datatypes are all used in the same way as `int`. For this reason, we omit code samples.

- The `float` and `double` datatypes store 32-bit and 64-bit floating-point numbers, respectively. That is, `float` and `double` are used to model real numbers such as 3.14 and 2.5. Note that as we only have finite memory, it is not possible to perfectly represent every real number. In particular, it is not possible to perfectly represent irrational numbers like π, $\sqrt{2}$, and Euler's constant e. However, floating-point numbers do provide a means to approximate these constants reasonably well. As the `double` datatype has 64-bits, it has a larger range and more precision than `float`. For this reason, it is convention to use `double` by default.

2.2.1 Exercises

(Required) Problem 3. Recall that the largest value a variable of type `byte` can hold is `127`. Consider the following code sample.

```
byte x = 127;
byte y = (byte)(x + 1);
```

(a) **Before testing this code sample on your computer**, what do you expect to happen when running this code sample. Will Java compile? If so, what value do you expect to be stored in `y`? Justify your reasoning.

(b) Write a Java program to test the above code sample. What value is actually stored in `y`?

(c) Modify your program, replacing the line

```
byte y = (byte)(x + 1);
```

with

```
byte y = (byte)(x + 2);
```

What value is stored in `y` when you run your program? What if we store `x+3` in `y`, rather than `x+2`?

(d) Based on your observations, explain how Java is handling operations on `byte` that exceed `127`, the largest value that can be represented as a `byte`.

2.3 Converting Between Primitives

In this section, we discuss how to convert a value of one primitive datatype to another. It is often advantageous to perform calculations using one datatype, and then convert the result back to another. For example, one may wish to perform numerical computations using floating-point math. We may know in advance that the result will of such a calculation will be an integer[3], and so the program should output the result as such. For this reason, it may be desirable to store the value in a variable of type `int`. As a variable of type `int` cannot store arbitrary real numbers, Java prohibits assigning floating-point numbers (such as `2.5`) to a variable of type `int`. In order to handle such conversions, Java has functionality called *casting*.

There are two types of casting: implicit and explicit. We delve into the details of casting primitive datatypes to understand how to use this tool, as well as understand how Java behaves. Casting objects will not be discussed in this section. We direct the reader to the Java Language Specification [1] for a more comprehensive overview of casting.

2.3.1 Implicit Casting

Implicit casting from a type `type1` to `type2` when every value of `type1` can be represented as `type2` without losing any information. So Java handles these casts implicitly. For example, a value of type `byte` can safely be upgraded to a `short`, `int`, or `long` without any danger of losing information. In such instances, the value of type `byte` is being *widened*.

Example 1. Consider the following example. Here, `x` stores the value `3`, represented as a `byte`. In the second line, the value stored in `x` is assigned to `y`. Because the `short` datatype is *wider* than the `byte`datatype, Java copies the value `3` stored in `x` and then implicitly converts this copied value to type `short` before assigning it to `y`.

```
byte x = 3;
short y = x;
```

There are 19 widening primitive conversions that Java handles implicitly, such as in Example 1, including the following. [1]

- `byte` to: `short`, `int`, `long`, `float`, or `double`.
- `short` to: `int`, `long`, `float`, or `double`.
- `char` to: `int`, `long`, `float`, or `double`.
- `int` to: `long`, `float`, or `double`.
- `long` to: `float` or `double`.
- `float` to `double`.

[3] We will, in fact, be performing such computations later in the course to quickly compute the nth Fibonacci number for large values of n.

Aside from assignments, Java handles implicit casting when performing arithmetic operations on primitives. If the two primitive values are compatible, but of different types, then the value of lower precedence is implicitly widened to the type of higher precedence. We illustrate this with the following examples.

Example 2. Recall that (in Java) `9/5` evaluates to `1`, as `9` and `5` are both of type `int`. Suppose instead we consider `9.0/5`. Here, `9.0` is of type `double`, so `5` is promoted to a `double`. Thus, `9.0/5` evaluates to `1.8`. Similarly, if we have `9/5.0`, the `9` is promoted to a `double`. So `9/5.0` also evaluates to `1.8`.

Example 3. We emphasize that the two values must be *compatible*. For instance, Java does not permit numerical operations with `boolean` numeric values. In particular, the following expression will trigger an error when compiling your Java program: `3 + true`.

2.3.2 Explicit Casting

Recall that Java implicitly casts primitive valuels when every value of the first type can safely be represented in the second type without losing information. Explicit casting is required when converting from `type1` to `type2`, where there exists a value of `type1` that cannot safely be represented in `type2` without losing information. For example, a value of type `long` must be explicitly converted to the type `int`. Note that if the value of type `long` exceeds the range of `int`, the value of type `long` can still be converted though at the cost of some information. We illustrate this with some examples.

Example 4. Consider the following code sample. Even though `0` safely fits into the `int` datatype, not every value of type `long` can safely be represented using the `int` type. For this reason, Java will fail to compile because of the second line.

```
long x = 0;
int y = x;
```

In order to convert the value stored in `x` to type `int`, it is necessary to explicitly cast the value in `x` to an `int`. The cast syntax syntax is as follows: `(int)`; Java attempts to convert the value after `(int)` to type `int`. In this case, `long` stores a number, so Java is successfully able to handle this conversion.

```
long x = 0;
int y = (int)x;
```

Example 5. Consider the following code sample. Just as in Example 4, the following code sample will fail to compile, due to the second line. Namely, a value of type `double` cannot be safely represented using an `int`.

```
double x = 2.5;
int y = x;
```

Explicitly casting the value stored in `x` to type `int` will resolve the compilation error. However, precision will be lost; namely, anything after the decimal point will be truncated. So `y` stores the value `2`, and not `2.0` or `2.5` which are both of type `double`.

2.3.3 Exercises

(Required) Problem 4. Consider the following section of code. Determine the first line that causes an error. If no line causes an error, clearly state this. In both cases, clearly justify your answer.

```
char c ='A';
int i = c;
c = i+1;
c = c+1;
```

(Required) Problem 5. Consider the following section of code. Determine the first line that causes an error. If no line causes an error, clearly state this. In both cases, clearly justify your answer.

```
byte b = 5;
char c = '5';
short s = 55;
int i = 555;
b = s;
i = c;
f = i;
```

(Required) Problem 6. Write a Java program that does the following:

- Prints out the character `'h'`.
- Prints out the corresponding `int` value of `'h'`, which we refer to as the ASCII value of `'h'`.
- Adds 3 to the ASCII value of `'h'` and prints out the corresponding character.

2.4 Converting Strings to Primitives

It is quite common to obtain input as a String, which we recall represents the values as text. Even if a string stores the value `"123"`, this value is treated as text and not a numeric primitive like `int`. In practice, one may wish to use the value `123` as an `int` or `double` to perform numerical calculations. For example, the payroll manager at a company may enter a 5% tax rate into the computer (as 0.05), for money to be withheld from employee paychecks. Java would interpret this entry as a String: `"0.05"`. In practice, we would want the program to multiply the employee's gross income for the given pay period by `0.05` to determine the withholding amount. Now it is not possible to multiply a number by text; for instance, we cannot multiply an `int` or `double` by a String. Thankfully, Java provides built-in library methods to convert Strings to primitive types. The two most common conversions are from String to `int`, and String to `double`.

- In order to convert a String to an `int`, we use the `Integer.parseInt()` method, which takes as input a String. Consider the following example. Here, `x` stores the text `"123"`. Now `Integer.parseInt(x)` takes the value stored in `x` and converts it to the `int` value `123`. So now we can use `y` in performing numeric calculations.

```
String x = "123";
int y = Integer.parseInt(x);
```

Caution: Note that the value in `x` **must** be an actual integer that can safely be represented as an `int`. If `x` stores `"123.5"` for example, `Integer.parseInt(x);` will fail to convert `"123.5"` to an `int`. In particular, your Java program will crash at runtime and generate a `NumberFormatException` and point to the offending line. We illustrate this with the following example.

```
public class ParseDemo{

        public static void main(String[] args){
                String x = "123.5";
                int y = Integer.parseInt(x);

        }
}
```

While this Java program successfully compiles, it crashes at runtime. Java generates the following error message.

```
Exception in thread "main" java.lang.NumberFormatException:  For input string:  "123.5"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at ParseDemo.main(ParseDemo.java:5)
```

This error message may seem daunting at first, but it actually provides a lot of useful information. The first line tells us that the offending String is `"123.5"`. The remaining four lines provide a trace as to where the error occurred. The last line in the error message points tells us the line of code that started this unfortunate chain of events: line 5 in `ParseDemo.java`, which is our code. In particular line 5 of our code is the following:

```
int y = Integer.parseInt(x);
```

Lines 3-4 of the error message point to certain lines of code within the `Integer.parseInt()` method. Because `Integer.parseInt()` is a Java method, we cannot do anything with this code. So it is safe to ignore those parts of the error message. It is usually a safe assumption that the built-in Java library functions work as intended, and that any issues are with code you wrote.

- The Java `Double.parseDouble()` method works analogously as the `Integer.parseInt()` method, converting a String to a `double`. We illustrate this with the following code sample. Note that `y` stores the value `123.5` represented as a `double` after the second line of code is executed.

```
String x = "123.5";
double y = Double.parseDouble(x);
```

Just as with the `Integer.parseInt()` method, the String provided to `Double.parseDouble()` **must** actually contain a value that can be safely represented as a `double`. Otherwise, Java will crash at runtime, with a `NumberFormatException`.

2.5 User Input via JOptionPane

In this section, we discuss a means to interact with the user. The JOptionPane window is a highly customizable popup window, which provides functionality to obtain user input via textboxes and buttons, as well as provide messages to the user. We discuss the basics of the JOptionPane window, so that we may begin writing Java programs that interact with the user. There are two key JOptionPane methods we will discuss, including `JOptionPane.showInputDialog()` and `JOptionPane.showMessageDialog()`.

- **Input Dialogs:** The `JOptionPane.showInputDialog()` method provides a basic popup window with a textfield, in which the user provides input. Once the user submits their input, a String is returned. We may provide a basic JOptionPane input dialog using the following method signature:

```
JOPtionPane.showInputDialog(Component parent, Object message)
```

We will provide `null` as the value for `parent` parameter. Since we are not developing a more extensive Java GUI application, there is no parent window to which the JOptionPane dialog should be associated. For the `message` parameter, we provide the String (text) to prompt the user. For example, the following line invokes an input dialog popup window prompting the user for their name, which is then stored in the variable `input`.

```
String input = JOptionPane.showInputDialog(null, "What is your name?");
```

- **Message Dialogs:** The `JOptionPane.showMessageDialog()` method provides a basic popup window, for the sole purpose of providing a message to the user. The `JOptionPane.showMessageDialog()` method has the following signature, which is similar to the `JOptionPane.showInputDialog()` method.

```
JOptionPane.showMessageDialog(Component parent, Object message)
```

Consider the following program which prompts the user for their name, then greets the user based on the name they entered. Note that in order to use the `JOptionPane` window, we must import it from the `javax.swing` package within the standard Java library. All `import` statements belong prior to the class definition.

```
import javax.swing.JOptionPane;

public class JOptionPaneDemo{

    public static void main(String[] args){
        String input = JOptionPane.showInputDialog(null, "What is your name?");
        JOptionPane.showMessageDialog(null, "Hello, " + input);
    }
}
```

2.5.1 Exercises

(Required) Problem 7. Familiarize yourself with Mad Libs. Then write your own Mad Libs template. Implement a Java program that prompts the user for words to fill in to your Mad Libs template, and then outputs the completed template.

(Required) Problem 8. When money is invested in a savings account, interest is earned on the money. Suppose the account pays 1% interest each year. So on a $100 initial deposit, the owner earns $1 in interest after the first year. Now after the second year, interest is paid on both the $100 initial deposit and the $1 interest payment from the previous year. So after two years, there is $102.01 in the account. This is referred to as *compount interest*.

In the discrete compount interest model, we have the following components:

- P: This represents the principal, or initial deposit.
- r: This is the interest rate.
- n: This is the number of times each year that interest is paid. For example, if interest is paid monthly, then $n = 12$.
- t: This is the duration from the initial deposit, in years. Note that t need not be an integer.

The amount of money in the account after t years is:

$$A(t) = P\left(1 + \frac{r}{n}\right)^{nt}$$

Your job is to write a program that prompts the user for principal, the interest rate, the number of times each year that interest is paid, and the number of years from the initial deposit. The program should then output the amount in the account.

You may find the `Math.pow()` method helpful to compute exponential expressions. The `Math.pow()` method has the following signature:

```
Math.pow(double a, double b)
```

Note that `Math.pow()` returns a `double`. As an example, the following computes $3.5^{1.5}$:

```
double x = Math.pow(3.5, 1.5);
```

(Required) Problem 9. Write a program to prompt the user to enter a file size in megabytes (MB). Convert the supplied file size to bits, bytes, kilobytes (KB), and gigabytes (GB) and output the result to the user. Note the following conversion factors:

- 8 bits = 1 byte.
- 1024 bytes = 1 KB.
- 1024 KB = 1 MB.
- 1024 MB = 1 GB.

3 Conditional Statements

Conditional reasoning is part of our everyday life. Here are some common examples.

- If a student earns at least a 90% in a class, then they receive a grade of `A`. If instead that student earns at least an 80% but less than a 90%, then that student earns a grade of `B`.
- In the NBA March Madness tournament, whether a team advances depends on whether they won the given round.
- In a game of Blackjack, a player immediately loses if the value of their hand is greater than 21.

In this section, we introduce the syntax for implementing conditional reasoning in Java.

3.1 If and Else Statements

The `if` statement is the most basic conditional flow statement. Code contained in an `if` statement is only executed if a particular condition evaluates to `true`. The syntax for an `if` statement is as follows:

```
if(Boolean-condition){
   ⟨code⟩
}
```

Here, the `Boolean-condition` represents any Java expression that evaluates to `true` or `false`. Any code inside the curly braces is only executed should the Boolean condition evaluate to `true`. We provide an example.

Example 6. Note that as `grade` has the value 90, `grade >= 90` evaluates to `true`. So the program outputs the following statement: `You earned an A`.

```
int grade = 90;
if(grade >= 90){
   System.out.println("You earned an A");
}
```

Note that if the value stored in `grade` falls below `90`, then the program does not output: `You earned an A`. This is illustrated in the following modification.

```
int grade = 89;
if(grade >= 90){
   System.out.println("You earned an A");
}
```

We next discuss the `else` statement, which provides allows us to execute code should the condition in the `if` clause evaluates to `false`. Consider the following example.

Example 7. As the value stored in `grade` is 90, the first if statement is evaluated and the program outputs: `You earned an A`. The `else if` statement is **not** evaluated because the `if(grade >= 90)` clause is evaluated.

```
int grade = 90;
if(grade >= 90){
   System.out.println("You earned an A");
}
else if(grade >= 80){
   System.out.println("You earned a B");
}
```

Suppose instead the value of `grade` is 89 instead of 90. As we saw in Example 6, the first `if` is not executed. However, as $89 \geq 80$, the `else if` statement is executed. So the program outputs: `You earned a B`.

Now an `else` need not be attached to an if statement, as in Example 7. Consider the following example.

Example 8. Note that if the value in `grade` is smaller than 90, then the program outputs `You earned a C.`

```
int grade = 70;
if(grade >= 90){
  System.out.println("You earned an A");
}
else{
  System.out.println("You earned a C");
}
```

We provide a runnable sample of code demonstrating the use of `if else` statements.

```
public class IfDemo{

    public static void main(String[] args){
        int grade = 64;
        if(grade >= 90){
                System.out.println("You earned an A");
        }
        else if(grade >= 80){
                System.out.println("You earned a B");
        }
        else if(grade >= 70){
                System.out.println("You earned a C");
        }
        else if(grade >= 60){
                System.out.println("You earned a D");
        }
        else{
                System.out.println("You earned an F");
        }
    }
}
```

3.2 Switch Statements

The `switch` statement provides another means for implementing conditional logic, in which a variable or expression can take on a set of values pre-determined by the developer. A `default` case is used to handle values that fall outside this pre-determined set. The `switch` statement supports the following datatypes: `byte`, `short`, `char`, `int`, String[4], and enumerated types defined using the `enum` keyword.

A `switch` statement accepts as input an expression. The `case` keyword is used to compare the expression to given values. Consider the example below.

[4]Support for the String datatype with the `switch` statement was introduced with the release of Java 7. Earlier versions of Java do not support use of the String datatype with the `switch` statement.

Example 9. Here, the value stored in `x` is compared to 0. If the value in `x` is 0, the program prints `Rock`. The `break` statement is used to terminate the `switch` statement. Now if instead the value in `x` is instead 1, the program prints out `Paper`. Similarly, if instead the value in `x` is 2, the program prints out `Scissors`. If none of the above cases are met, then the program prints out `Invalid option`. Note that for this sample, the output is precisely: `Rock`.

```
int x = 0;
switch(x){
        case 0:
                System.out.println("Rock");
                break;
        case 1:
                System.out.println("Paper");
                break;
        case 2:
                System.out.println("Scissors");
                break;
        default:
                System.out.println("Invalid option");
}
```

The `break` statements in Example 9 are key, to terminate a given case should it be executed. Without the `break`statement, the subsequent `case` blocks are contained within the previous `case`. This is illustrated in the next example.

Example 10. Suppose the `break` statements in Example 9 are removed.

```
int x = 0;
switch(x){
        case 0:
                System.out.println("Rock");
        case 1:
                System.out.println("Paper");
        case 2:
                System.out.println("Scissors");
        default:
                System.out.println("Invalid option");
}
```

As `x` contains the value 0, the first `case` is executed. In the absence of a `break`statement, the `case 1:` block is also executed. Similarly, the `case 2:` and `default` blocks are also executed. So the output of this code sample is:

```
Rock
Paper
Scissors
Invalid option
```

In situations when we wish to respond in the same manner to multiple cases, omitting the break statements allows us to group such cases together. We illustrate this with the following example.

Example 11. Suppose that `month` is a variable of type `int`. Here, if `month` holds the value $1, 3, 5, 7, 8, 10$, or 12, the program outputs: `There are 31 days in the month`. Analogously, if `month` holds the value 2, the program outputs: `There are 28 days in the month`.

```
switch(month){
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
                System.out.println("There are 31 days in the month");
                break;
        case 2:
                System.out.println("There are 28 days in the month");
                break;
        case 4:
        case 6:
        case 9:
        case 11:
                System.out.println("There are 30 days in the month");
                break;
        default:
                System.out.println("Invalid option");
        }
}
```

3.3 Exercises

(Required) Problem 10. Implement a two-player game of Rock-Paper-Scissors.

(Required) Problem 11. Implement a program that takes as input the month number, between 1 and 12 inclusive, and outputs the name of that month. If the month number is invalid, the program should let the user know this.

(Required) Problem 12. A company charges different rates depending on the number of T-shirts a customer orders, as well as the number of colors that will be used. The rates are as follows.

- **One Color:**
  - $1 - 99$ shirts: \$7 per shirt.
  - $100 - 249$ shirts: \$6 per shirt.
  - 250 or more shirts: \$5 per shirt.
- **Two Colors:**
  - $1 - 99$ shirts: \$8 per shirt.
  - $100 - 249$ shirts: \$7 per shirt.
  - 250 or more shirts: \$6 per shirt.
- **Three Colors:**
  - $1 - 99$ shirts: \$9 per shirt.
  - $100 - 249$ shirts: \$8 per shirt.
  - 250 or more shirts: \$7 per shirt.

Write a program to prompt the user for the number of T-shirts they would like to order, as well as the number of colors $(1, 2,$ or $3)$ they would like to use. Your program should calculate the total cost for the customer's order, including a 5% sales tax.

4 Logic

In this section, we introduce propositional logic. Propositions are the fundamental building blocks of mathematics, including mathematical statements (propositions), operations on these statements, and the rules for using these operations. The propositional logic framework provides the basis for constructing more complicated mathematical statements from simpler propositions, as well as for formulating mathematical proofs.

George Boole observed that, by representing simple propositions as symbols, propositions could be viewed as algebraic expressions. Hence, propositional logic is a common example of a *Boolean algebra.* Manipulating such algebraic expressions is of key importance in constructing conditional expressions in computer programming. Boolean algebra is also closely related to digital logic, which studies the combinatorial circuit model of computation. Combinatorial circuits are used to compute Boolean functions. So simplifying Boolean expressions allows us to construct simpler and more efficient circuits.

The exposition in this section has benefitted from several resources, including [2], [5], and [11].

4.1 Propositions

We begin with the definition of a proposition.

Definition 1 (Proposition)**.** A *proposition* is a declarative sentence that is either true (denoted T) or false (denoted F).

Example 12. The following statements are propositions, as they are either universally true or universally false. There are no special cases or conditions for the statements to be true.

- All cows are brown.
- $2 + 2 = 4$.
- There is life on Mars.

Example 13. We note that $x + 2 = 4$ is **not** a proposition, as the truth value of this statement depends on the value of x.

Example 14. *How are is it to the next town?* is **not** a proposition, as this is a question. We note that questions are neither true nor false.

Definition 2 (Logical Equivalence)**.** Let p and q be two propositions that take on the same truth value. We denote this as $p \equiv q$.

Example 15. Let p be the proposition that $2 + 2 = 4$, which is a true proposition. So we write $p \equiv T$.

We next discuss how to construct more complicated mathematical statements from simpler propositions, namely using the `and`, `or`, and `not` operations.

Definition 3 (Logical And)**.** Suppose that p and q are propositions. The proposition representing p `and` q, dentoed $p \wedge q$, is true precisely when both p and q are true. Otherwise, $p \wedge q$ is false. This is formalized with the following truth table.

p	q	$p \wedge q$
F	F	F
F	T	F
T	F	F
T	T	T

If we view F as 0 and T as 1, then $\wedge$ can be viewed as multiplication. So $p \wedge q$ evaluates to 1 precisely when $p = 1$ and $q = 1$.

Definition 4 (Logical Or)**.** Suppose that p and q are propositions. The proposition representing p `or` q, dentoed $p \vee q$, is true precisely when p, q, or both are true. Otherwise, $p \vee q$ is false. This is formalized with the following truth table.

p	q	$p \vee q$
F	F	F
F	T	T
T	F	T
T	T	T

Example 16. Let p denote the proposition that $2 + 2 = 4$, and let q denote the proposition that the sky is green. Clearly, p is true, while q is false. So $p \wedge q$ is false. However, $p \vee q$ is true.

Definition 5 (Logical Not)**.** Suppose that p is a proposition. The negation of p, denoted $\neg p$ or $\overline{p}$, is true precisely when p is false. This is formalized with the following truth table.

p	$\neg p$
F	T
T	F

In a propsitional expression, the $\neg$ operator takes the highest precedence, followed by the $\wedge$ operator, and lastly the $\vee$ operator.

Example 17. Suppose $p \equiv F, q \equiv T$, and $r \equiv T$. Consider the expression $\neg p \vee q \wedge r$, which is evaluated as follows.

- We first evaluate $\neg p \equiv F$.
- Next, we evaluate $q \wedge r \equiv T \wedge T \equiv T$.
- Finally, we evaluate $(\neg p) \vee (q \wedge r) \equiv T \vee T \equiv T$.

4.1.1 Exercises

(Required) Problem 13. Let p and q be propositions. The *exclusive or* operator is a binary operator, where $p \oplus q$ evaluates to true when exactly one of p, q is true. If both p and q are true, or both p and q are false, then $p \oplus q$ evaluates to false.

(a) Construct a truth table for $p \oplus q$.

(b) Construct a truth table for $(p \vee q) \wedge (\neg p \vee \neg q)$. Compare your truth table here to your truth table for part (a).

(Required) Problem 14. Let p and q be propositions.

(a) Evaluate $p \oplus p$.

(b) Evaluate $p \oplus \neg p$.

(c) Consider the following segment of Java code, where $\wedge$ is the exclusive-or operator in Java. You may assume that p and q are variables of type `boolean`. Explain what the following segment of code does in plain English.

```
p = p ∧ q;
q = p ∧ q;
p = p ∧ q;
```

4.2 Algebraic Laws

In this section, we introduce some useful algebraic laws of propositional logic. These algebraic laws will be useful later in verifying logical equivalences, as well as simplifying Boolean expressions. We note that these laws are theorems, which can be proven. These laws are not simply axioms which we impose.

We begin with the commutative, associative, identity, complement, idempotent, and annihilator laws.

- **Commutative Law:** Both the `and` ($\wedge$) and `or` ($\vee$) operators commute. This means that for any propositions p, q, we have that: $p \wedge q \equiv q \wedge p$, and $p \vee q \equiv q \vee p$.

- **Associative Law:** Both the `and` ($\wedge$) and `or` ($\vee$) operators are associative. This means that for any propositions p, q, r, we have that:

$$p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r, \text{ and}$$
$$p \vee (q \vee r) \equiv (p \vee q) \vee r.$$

- **Identity Laws:**

  - **And:** Recall that we may view the `and` ($\wedge$) operator as multiplication on $\{0, 1\}$. With that in mind, 1 is the multiplicative identity. So in the language of propositional logic, we have that for any proposition p:

$$p \wedge T \equiv p.$$

  - **Or:** In much the same way that the `and` ($\wedge$) operator can be viewed as multiplication, the `or` ($\vee$) operator bears striking resemblance to addition[5] over $\{0, 1\}$. Recall that the additive identity on the integers is 0. So F is the identity for $\vee$. That is, for any proposition p, we have that:

$$p \vee F \equiv p.$$

- **Complement Laws:** Suppose that p is a proposition. So regardless of whether p is true, we have that exactly one of p and $\neg p$ is true, and the other is false. It follows immediately that:

$$p \vee \neg p \equiv T, \text{ and}$$
$$p \wedge \neg p \equiv F.$$

- **Idempotent Laws:** Let p be a proposition. We have that:

$$p \vee p \equiv p, \text{ and}$$
$$p \wedge p \equiv p.$$

- **Annihilator Laws:** Let p be a proposition. We have that:

$$p \vee T \equiv T, \text{ and}$$
$$p \wedge F \equiv F.$$

- **Distributive Laws:**

  - $\wedge$ **distributes over** $\vee$: For any propositions p, q, r, we have that $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$.
  - $\vee$ **distributes over** $\wedge$: For any propositions p, q, r, we have that $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$.

  We prove the first distribution law later and leave the second as an exercise for the reader.

- **DeMorgan's Law:** Informally, DeMorgan's Law states that a negation distributes over both the `and` ($\wedge$) and `or` ($\vee$) operators, negating both the propositions and flipping the operator from $\wedge$ to $\vee$ or vice-versa. Formally, for any propositions p, q, we have the following.

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$
$$\neg(p \vee q) \equiv \neg p \wedge \neg q$$

  We prove one form of DeMorgan's Law later. The proof of the second form is left as an exercise for the reader.

[5] If we want subtraction, the `exclusive or` ($\oplus$) operator better represents addition over $\{0, 1\}$, as $\oplus$ is invertible while $\vee$ is not. More precisely, the integers modulo 2, denoted $\mathbb{Z}_2$ consists of the set $\{0, 1\}$ together with $\oplus$ as the addition operation and $\wedge$ as the multiplication operation. We will discuss this more during our Number Theory unit.

- **Absorption Laws:** Let p, q be propositions. The following hold:

$$p \wedge (p \vee q) \equiv p, \text{ and}$$
$$p \vee (p \wedge q) \equiv p.$$

Again, we prove the first Absorption Law and leave the proof of the second as an exercise for the reader.

We now prove the first Distributive Law.

Theorem 4.1. *Let p, q, r be propositions. We have that $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$.*

Proof. We construct a truth table to verify that $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$.

p	q	r	$(q \vee r)$	$p \wedge (q \vee r)$	$(p \wedge q)$	$(p \wedge r)$	$(p \wedge q) \vee (p \wedge r)$
F	F	F	F	F	F	F	F
F	F	T	T	F	F	F	F
F	T	F	T	F	F	F	F
F	T	T	T	F	F	F	F
T	F	F	F	F	F	F	F
T	F	T	T	T	F	T	T
T	T	F	T	T	T	F	T
T	T	T	T	T	T	T	T

We note that as the fifth and last columns agree on all possible configurations of (p, q, r), that $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$. □

Theorem 4.2 (DeMorgan's Law)**.** *For any propositions p, q, we have that:*

$$\neg(p \wedge q) \equiv \neg p \vee \neg q.$$

Proof. We construct a truth table to verify that: $\neg(p \wedge q) \equiv \neg p \vee \neg q$.

p	q	$p \wedge q$	$\neg(p \wedge q)$	$\neg p$	$\neg q$	$\neg p \vee \neg q$
F	F	F	T	T	T	T
F	T	F	T	T	F	T
T	F	F	T	F	T	T
T	T	T	F	F	F	F

As the fourth and last columns agree on all configurations of (p, q), we have that $\neg(p \wedge q) \equiv \neg p \vee \neg q$. □

We conclude this section with a proof of the first Absorption Law. While it is possible to use a truth-table as our proof, it is not necessary. Rather, we apply algebraic laws to manipulate the expression. It is these manipulations that will be particularly useful moving forward.

Theorem 4.3 (Absorption Law)**.** *For any propositions p, q, we have that:*

$$p \vee (p \wedge q) \equiv p.$$

Proof. Recall that T is the multiplicative identity. So $p \equiv (p \wedge T)$. It follows that:

$$p \vee (p \wedge q) \equiv (p \wedge T) \vee (p \wedge q).$$

We may now use the distributive law to factor out p from the right hand side of the above expression. Thus:

$$\begin{aligned}(p \wedge T) \vee (p \wedge q) &\equiv p \wedge (T \vee q) \\ &\equiv p \wedge T \\ &\equiv p.\end{aligned}$$

Now by the annihilator law, $(T \vee q) \equiv T$, which yields the second equivalence. Finally, we note that T is the identity for $\wedge$, so $p \wedge T \equiv p$. The result follows. □

4.2.1 Exercises

(Required) Problem 15. Using a truth table, prove the following: for any propositions p, q, r, we have that:

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r).$$

(Required) Problem 16. Using a truth table, prove the following: for any proposition p, q, we have that:

$$\neg(p \vee q) \equiv \neg p \wedge \neg q.$$

(Required) Problem 17. The *double negation law* states that for any proposition p, the following holds:

$$\neg(\neg p) \equiv p.$$

Using a truth table, prove the double negation law.

(Required) Problem 18. Using algebraic manipulations, prove the second Absorption Law: for any propositions p, q, we have that:

$$p \wedge (p \vee q) \equiv p.$$

Clearly justify each manipulation with the corresponding law.

4.3 Implications

An implication is the most important logical constructive for proofs. Intuitively, an implication captures the notion of an *if ..., then* statement.

Definition 6 (Implication)**.** Let p, q be propositions. The *implication* $p \implies q$ is read as p *implies* q or *if* p*, then* q. We note that $p \implies q \equiv F$ only when $p \equiv T$ and $q \equiv F$. That is, a true proposition cannot imply a false proposition. In light of this observation, $p \implies q \equiv \neg p \vee q$.

Example 18. Let p be the proposition that the time is 10:30 AM, and let q be the proposition that we take our break. The implication $p \implies q$ can be read as: *if the time is 10:30 AM, then we take our break.*

Example 19. Let p be the proposition that Mars is made of chocolate, and let q be the proposition that the world will end on midnight. The implication $p \implies q$ can be read as: *if Mars is made of chocolate, then the world will end at midnight.* Note that I am not appealing to special knowledge about Mars. Rather, as we know Mars certainly is *not* made of chocolate, the hypothesis is not satisfied. So the implication *if Mars is made of chocolate, then the world will end at midnight* is trivially true because the hypothesis is false.

Using the definition of an implication and our algebraic laws from the previous section, we can prove some helpful identities regarding implications.

Theorem 4.4. *Let p, q, r be propositions. Suppose that $p \implies q$ and $p \implies r$. This is equivalent to $p \implies (q \wedge r)$. Symbolically, we may write this as:*

$$(p \implies q) \wedge (p \implies r) \equiv p \implies (q \wedge r).$$

Proof. We note that $p \implies q \equiv \neg p \vee q$ and $p \implies r \equiv \neg p \vee r$. Thus:

$$(p \implies q) \wedge (p \implies r) \equiv (\neg p \vee q) \wedge (\neg p \vee r).$$

By the distributive law, we have that:

$$(\neg p \vee q) \wedge (\neg p \vee r) \equiv [\neg p \wedge (\neg p \vee r)] \vee [q \wedge (\neg p \vee r)].$$

By the absorption law, $\neg p \wedge (\neg p \vee r) \equiv \neg p$. Applying the distributive law again, we have that: $q \wedge (\neg p \vee r) \equiv (q \wedge \neg p) \vee (q \wedge r)$. So we have that:

$$[\neg p \wedge (\neg p \vee r)] \vee [q \wedge (\neg p \vee r)] \equiv \neg p \vee (q \wedge \neg p) \vee (q \wedge r).$$

We again apply the absorption law, noting that: $\neg p \vee (q \wedge \neg p) \equiv \neg p$. Thus:

$$\begin{aligned} \neg p \vee (q \wedge \neg p) \vee (q \wedge r) &\equiv \neg p \vee (q \wedge r) \\ &\equiv p \implies (q \wedge r). \end{aligned}$$

We obtain the last equivalence from the fact that $a \implies b \equiv \neg a \vee b$. The result follows. □

We next discuss the inverse, converse, and contrapositive of an implication.

Definition 7 (Inverse)**.** Let p, q be propositions. The *inverse* of the implication $p \implies q$ is the implication $\neg p \implies \neg q$. We note that the inverse, $\neg p \implies q$ may be written as $\neg(\neg p) \vee \neg q \equiv p \vee \neg q$. So the original implication $p \implies q$ is **not** equivalent to its inverse $\neg p \implies \neg q$.

Definition 8 (Converse)**.** Let p, q be propositions. The *converse* of the implication $p \implies q$ is the implication $q \implies p$. We note that the converse, $q \implies p$, may be written as $\neg q \vee p$. So the original implication $p \implies q$ is **not** equivalent to its converse $q \implies p$.

Definition 9 (Contrapositive)**.** Let p, q be propositions. The *inverse* of the implication $p \implies q$ is the implication $\neg q \implies \neg p$. We note that the contrapositive, $\neg q \implies \neg p$, may be written as $\neg(\neg q) \vee \neg p \equiv q \vee \neg p$. So the original implication $p \implies q$ **is** in fact equivalent to its contrapositive $\neg q \implies \neg p$.

Example 20. Again let p be the proposition that the time is 10:30 AM, and let q be the proposition that we take our break. Consider the implication $p \implies q$.

- **Inverse:** The inverse of this implication, $\neg p \implies \neg q$, reads as: *If the time is not 10:30 AM, then we do not take our break.*
- **Converse:** The converse of this implication, $q \implies p$, reads as: *If we take our break, then the time is 10:30.*
- **Contrapositive:** The contrapositive of this implication, $\neg q \implies \neg p$, reads as: *If the time is not 10:30 AM, then we do not take our break.*

4.3.1 Exercises

(Required) Problem 19. Let p, q, r be propositions. Prove that:

$$[(p \implies r) \wedge (q \implies r)] \equiv [(p \vee q) \implies r].$$

(Required) Problem 20. Let p, q, r be propositions. Prove that the following statement is always true:

$$[(p \vee q) \wedge (\neg p \vee r)] \implies (q \vee r).$$

(Required) Problem 21. Let p be the proposition that *I bought a lottery ticket*, and let q be the proposition that *I won the million dollar jackpot.* Express each of the following propositions as English sentences.

(a) $\neg p$

(b) $p \vee q$

(c) $p \wedge q$

(d) $p \implies q$

(e) $(p \implies q) \wedge (q \implies p)$

(f) $\neg p \implies \neg q$

(g) $q \implies p$

(h) $\neg q \implies \neg p$.

(i) $\neg p \wedge \neg q$.

(j) $\neg p \vee (p \wedge q)$.

4.4 Rules of Inference

In this section, we introduce four key rules of inference, as well as their applications in deductive reasoning. We begin by stating these rules and illustrating some applications.

The first rule we will discuss is Modus Ponens. Informally, Modus Ponens states that if $p \implies q$ and we know that p holds, then q holds as well.

Theorem 4.5 (Modus Ponens)**.** *Let p, q be propositions. The following statement is a tautology:*

$$\Big[(p \wedge (p \implies q)) \implies q\Big].$$

Proof. We have that:

$$\begin{aligned}
\Big[(p \wedge (p \implies q)) \implies q\Big] &\equiv \neg(p \wedge (\neg p \vee q)) \vee q && (1)\\
&\equiv (\neg p \vee \neg(\neg p \vee q)) \vee q && (2)\\
&\equiv (\neg p \vee (p \wedge \neg q)) \vee q && (3)\\
&\equiv ((\neg p \vee p) \wedge (\neg p \vee \neg q)) \vee q && (4)\\
&\equiv (T \wedge (\neg p \vee \neg q)) \vee q && (5)\\
&\equiv (\neg p \vee \neg q) \vee q && (6)\\
&\equiv \neg p \vee (\neg q \vee q) && (7)\\
&\equiv \neg p \vee T && (8)\\
&\equiv T. && (9)
\end{aligned}$$

Here, line (1) follows from the fact that $a \implies b \equiv (\neg a \vee b)$. Lines (2)-(3) follow from DeMorgan's Law. Line (4) follows from the Distributive Law. Lines (5) and (7) follow from the Complement Law. Line (6) follows from the fact that T is the identity for $\wedge$. Line (7) follows from the Associative Law, and line (9) follows from the Annihilator Law. □

Example 21. We illustrate the use of Modus Ponens with the following example. Suppose we know that everyone who is sane can do logic. Next, I tell you that I am sane. Modus Ponens provides that I can do logic. We illustrate this symbolically, so that the application of Modus Ponens is clear.

- Let S be the statement: *it is sane.*
- Let L be the statement: *it can do logic.*
- The implication *everyone who is sane can do logic*, can be written as $S \implies L$.
- Next, we are told that *I am sane.* So we have $S \wedge (S \implies L)$. Modus Ponens thus tells us that L holds. In other words, I can do logic.

We next discuss Modus Tollens, which is the contrapositive form of Modus Ponens. Intuitively, Modus Tollens states that if the conclusion is false, then the hypothesis must also be false. This is formalized as follows.

Theorem 4.6 (Modus Tollens)**.** *Let p, q be proposition, and suppose that $p \implies q$. The following statement holds:*

$$[\neg q \wedge (p \implies q)] \implies \neg p.$$

Example 22. We illustrate the use of Modus Tollens with the following example. Again, suppose we know that everyone who is sane can do logic. Suppose I tell you that I cannot do logic. As a result, Modus Tollens provides that I am not sane. We illustrate this symbolically, so that the application of Modus Tollens is clear.

- Let S be the statement: *it is sane.*
- Let L be the statement: *it can do logic.*
- The implication *everyone who is sane can do logic*, can be written as $S \implies L$.

- Next, we are told that *I cannot do logic.* So we have that $\neg L \wedge (S \implies L)$. Modus Tollens thus tells us that $\neg S$ holds. In other words, I am not sane.

The next rule we discuss is Hypothetical Syllogism, which establishes that implication is transitive. That is, if $p \implies q$ and $q \implies r$, then $p \implies r$. Formally, we have the following.

Theorem 4.7 (Hypothetical Syllogism)**.** *Let p, q, r be propositions that $p \implies q$ and $q \implies r$. Then $p \implies r$. Symbolically, we have the following tautology:*

$$\Big[(p \implies q) \wedge (q \implies r)\Big] \implies (p \implies r).$$

Example 23. Suppose we are told the following.

- I find all of your presents useful.
- None of my saucepans are of the slightest use.

We convert these statements to symbolic implications.

- Let U be the statement: *it is useful.*
- Let P be the statement: *it is one of your presents.*
- Let S be the statement: *it is one of my saucepans.*
- The implication *I find all of your presents useful* can be written as $P \implies U$.
- The implication *None of my saucepans are of the slightest use* can be written as $U \implies \neg S$.
- As $P \implies U$ and $U \implies \neg S$, Hypothetical Syllogism provides that $P \implies \neg S$. In English, this reads that: none of your presents were saucepans.

The last rule of inference we will introduce is Disjunctive Syllogism. Informally, Disjunctive Syllogism states that if p or q must be true and we know that p is false, then q must be true. Formally, we have the following.

Theorem 4.8 (Disjunctive Syllogism)**.** *Let p, q be propositions. We have that:*

$$((p \vee q) \wedge \neg p) \implies q.$$

Example 24. Suppose that someone studies either medicine or law. If we know this person does not study medicine; then by Disjunctive Syllogism, we know they study law.

Our rules of inference allow us to use draw conclusions from sequences of logical statements. Lewis Carroll problems provide worthwhile examples for practicing with these rules of inference. We illustrate with an example.

Example 25. A Lewis Carroll problem consists of a number of logical statements. The goal is to organize the implications in such a way as to draw a culminating conclusion. Consider the following Lewis Carroll problem.

- No ducks waltz.
- No officers ever decline to waltz.
- All my poultry are ducks.

Our first step is to re-write these implications symbolically. To do this, we begin by using symbols to represent the propositions.

- Let D represent the proposition: *it is a duck.*
- Let W represent the proposition: *it waltzes.*
- Let O represent the proposition: *it is an officer.*
- Let P represent the proposition: *it is my poultry.*

- We may write the statement *No ducks waltz* as $W \implies \neg D$.
- We may write the statement *No officers ever decline to waltz* as $\neg W \implies \neg O$.
- We may write the statement as *All my poultry are ducks* as $P \implies D$.

Our **goal** is to use Hypothetical Syllogism to chain the implications together, in order to draw a final conclusion. However, not all the implications are in the appropriate form to use Hypothetical Syllogism. For example, we have $W \implies \neg D$ and $\neg W \implies \neg O$. In both of these implications, there is a W in the hypothesis, whereas we want a W in the hyptohesis of one implication and a W in the conclusion of the other. In particular, the hypothesis and conclusion have to both contain W or both contain $\neg W$; we **cannot** mix and match.

Note however that an implication and its contrapositive are equivalent. Given that $W \implies \neg D$, we equivocally have that $D \implies \neg W$. We now apply Hypothetical Syllogism to the implications $D \implies \neg W$ and $\neg W \implies \neg O$ to obtain that $D \implies \neg O$.

We apply the Hypothetical Syllogism again, this time to the implications $P \implies D$ and $D \implies \neg O$ to conclude that $P \implies \neg O$.

So we have the chain of implications:

$$P \implies D \implies \neg W \implies \neg O.$$

In English, none of my poultry are officers.

4.4.1 Exercises

(Required) Problem 22. Prove Theorem 4.6 (Modus Tollens): suppose p, q are propositions such that $p \implies q$. The following is a tautology:

$$[\neg q \wedge (p \implies q)] \implies \neg p.$$

(Required) Problem 23. Solve the following Lewis Carroll problem.

- All the old articles in this cupboard are cracked.
- No jug in this cupboard is new.
- Nothing in this cupboard, that is cracked, will hold water.

(Required) Problem 24. Solve the following Lewis Carroll problem.

- It is not sunny this afternoon and it is colder than yesterday.
- We will go swimming only if it is sunny.
- If we do not go swimming, then we will take a canoe trip.
- If we take a canoe trip, then we will be home by sunset.

(Required) Problem 25. Solve the following Lewis Carroll problem.

- Things sold in the street are of no great value.
- Nothing but rubbish can be had for a song.
- Eggs of the Great Auk are very valuable.
- It is only what is sold in the street that is really rubbish.

(Required) Problem 26. Solve the following Lewis Carroll problem.

- The only animals in this house are cats.
- Every animal is suitable for a pet, that loves to gaze at the moon.
- When I detest an animal, I avoid it.
- No animals are carnivorous, unless they prowl at night.
- No cat fails to kill mice.
- No animals ever take to me, except what are in this house.
- Kangaroos are not suitable for pets.
- None but carnivora kill mice.
- I detest animals that do not take to me.
- Animals, that prowl at night, always love to gaze at the moon.

4.5 Logical Operators in Java

In this section, we discuss logical operators and constructing `boolean` conditions in Java involving multiple simpler conditions. We begin with Java's equality and relational operators, for constructing these simpler conditions.

- Equality: `==` We note that the `==` operator works well for comparing primitives, though not as well for comparing Objects like Strings. We will expand on this later in the course during our discussion of Objects.
- Not Equal To: `!=`
- Greater Than: `>`
- Greater Than or Equal To: `>=`
- Less Than: `<`
- Less Than Or Equal To: `<=`

Example 26. Suppose we have the variable `x` of type `int`. To test if `x` stores the value 5, we may use the following:

```
if(x == 5){
   ...
}
```

The other relational operators work similarly as the `==` operator.

In order to construct more complicated `boolean` conditions, Java provides the following operators.

- Conditional And: `&&`
- Conditional Or: `||`
- Exclusive Or Operator: $\wedge$
- Not Operator: `!`

The `&&`, `||`, and $\wedge$ operators are all binary operators: they take two `boolean` values and return a single `boolean` value.

Example 27. Again, suppose we have a `int` variable `x`. The following if statement tests whether `x < 0` or `x == 5`.

```
if(x < 5 || x == 5){
   ...
}
```

The `&&` and $\wedge$ operators work similarly.

Example 28. The `!` is a unary operator, which takes as input a single `boolean` value and inverts it. Again, suppose we have a `int` variable `x`. The following if statement tests whether `x` $\geq$ `0`, using the `!` operator.

```
if( !(x < 0) ){
   ...
}
```

5 Loops

Frequently, one will want to execute the same commands repeatedly, terminating only when a given condition is satisfied. Loop constructs allow for implementing such iterative logic.

5.1 While Loops

The `while` loop is the appropriate looping construct to use when we wish to iterate until a given condition is satisfied. The syntax for the `while` loop is as follows.

```
while(boolean-condition){
   //code
}
```

Java evalutes the loop as follows. First, Java checks to see if the `boolean`-condition evaluates to `true`. If this is the case, the code inside the loop (between the curly braces) is executed. Afterwards, Java then checks the `boolean`-condition again to see if it evaluates to `true`, in which case the code inside the loop is executed a second time. We proceed in a similar manner until the `boolean`-condition evaluates to `false`. Note that if the `boolean`-condition never evaluates to `true`, then the code inside the loop is never executed and Java proceeds to execute the next line immediately following the loop.

Example 29. Consider the following example. Here, we begin by assuming that the user wishes to play again, which is indicated by initializing the variable `playAgain` to `true`. As `playAgain == true`, the code inside the `while` loop is executed. Inside the loop, the user is prompted to enter `0` if they want to play again and `1` otherwise. If the user enters `0`, then `response == 0` evaluates to `true`. So `playAgain` is assigned the value `true`. If the user enters an `int` other than `0`, `playAgain` is assigned the value of `false`.

```
import javax.swing.JOptionPane;

public class WhileDemo1{

    public static void main(String[] args){
        boolean playAgain = true;
        while(playAgain){
                String input = JOptionPane.showInputDialog(null,
                        "Enter 0 to Play Again or 1 to Quit.");

                int response = Integer.parseInt(input);
                playAgain = (response == 0);
        }
    }
}
```

5.2 Do-While Loops

A `do-while` loop works similarly to a `while` loop, except that a `do-while` loop is guaranteed to execute at least once. In contrast, a `while` loop may never execute if the `boolean`-condition is never `true`. The syntax for a `do-while` loop is as follows.

```
do{
   //code
}while(boolean-condition);
```

Example 30. Here, we extend Example 27. Notice the code in the `while` and the `do-while` loop is the same. However, as `playAgain` is initialized to `false`, the `while` loop is never executed. However, the `do-while` is executed. As in Example 27, if the user enters 0, `playAgain`

```
import javax.swing.JOptionPane;

public class WhileDemo2{

    public static void main(String[] args){
        boolean playAgain = false;
        while(playAgain){
                String input = JOptionPane.showInputDialog(null,
                        "Enter 0 to Play Again or 1 to Quit.");

                int response = Integer.parseInt(input);
                playAgain = (response == 0);
        }

        do{
                String input = JOptionPane.showInputDialog(null,
                        "Enter 0 to Play Again or 1 to Quit.");

                int response = Integer.parseInt(input);
                playAgain = (response == 0);
        }while(playAgain);
    }
}
```

5.2.1 Exercises

(Required) Problem 27. Modify the program in Example so that if the user enters an integer other than 0 or 1, the program will prompt the user to enter 0 or 1 until they do so. This is called *validating user input*.[6]

(Required) Problem 28. Write a program to take as input a non-negative integer n as input and outputs the integers $0, 1, \ldots, n$. Your program should validate that the integer entered was non-negative. Additionally, your program should ask the user if they wish to play again.

(Advanced) Problem 1. Write a guessing game program, which generates a random integer between 0 and 100. The random number is not revealed to the user; their job is to guess it. Your program should tell the user if their guess is too high, too low, or accurate. After the round ends, your program should offer the option to play again.

Here are some ideas for additional features.

- Allow the user to customize the range for the random number.

[6] It is often the case that the user will not enter valid input, either by mistake or intentionally. Failing to handle user input appropriately can create major issues, such as with Little Bobby Tables.

- Track the number of guesses the user makes on a given round.
- Track the number of guesses in the user's best round. [Think about how to define the *best round*. Guessing the correct number in two tries if there are only five options is not as good, as if there were 1000 options.]
- Update the range on the display based on the user guesses. As an example, suppose the random number lies in the range $0 - 100$. If the user guesses 10 and the random number is higher than 10, update the display to indicate that the random number is in the range $10 - 100$ rather than $0 - 100$.
- How should the computer optimally guess this number? Design a computer to guess the number in the background. Your program should compare the number of guesses the computer and player took. Did the player beat the computer on a given round?

(Advanced) Problem 2. In the Game of Nim, there are n stones and two players, which take turns removing stones. A player can remove either one, two, or three stones during their turn. A player loses if there are no stones to remove on their turn.

(a) Your job is to implement a program that prompts the users for the number of stones, then allows two players to play the Game of Nim. After the game ends, the program should allow the users the option to play again.

(b) Play the Game of Nim with a friend for a few rounds. When does Player 1 have a winning strategy? What about Player 2? Explain your reasoning.

(c) **(Challenge)** Design an artificial intelligence player to compete against the user in the Game of Nim.

5.3 For Loops

A `for` loop is used to execute a given number of iterations, such as counting from 0 to $n - 1$ or traversing arrays[7]. The syntax of a for loop is as follows.

```
for(var-declaration; boolean-condition; var-modification){
  //code
}
```

The loop is executed as follows.

(a) The `var-declaration` commands are executed.

(b) Next, the `boolean-condition` is evaluated. If the condition evaluates to `true`, the loop is executed. Otherwise, the loop is skipped.

(c) If the condition evaluates to `true`, then the code inside the loop (located between the curly braces) is executed.

(d) After the code inside the loop is executed, the `var-modification` commands are executed. Then, the `boolean` condition is checked again, so we start back at (b).

Example 31. Consider the following example. Here, we begin by initializing `i = 0`. As $0 < 3$, the loop executes, so `i` is printed out to the console. Afterwards, the value in the variable `i` is incremented by 1. In the context of this loop, `i++` is equivalent to `i += 1`.

```
public class ForDemo1{

    public static void main(String[] args){
        for(int i = 0; i < 3; i++){
                System.out.println(i);
        }
    }
}
```

[7] We will discuss arrays in the next section.

The output of this program is as follows:

```
0
1
2
```

We note that after 2 is printed, `i++` updates the value in `i` so that `i == 3`. As `3 < 3` evaluates to `false`, the `for` loop terminates.

5.3.1 Exercises

(Required) Problem 29. Write a program to accept user input n, a non-negative integer, and print the first n even integers.

(Required) Problem 30. Write a program to accept user input n, a non-negative integer, and count the number of non-negative integers that are divisible by 3 or 5 (or both). To test if a positive integer k is divisible by 3, we check that `k%3 == 0`. Here, the `%` is the *modulus* operator, which returns the remainder after long division, when dividing `k` by 3.

(Advanced) Problem 3. Can you provide a formula for the number of non-negative integers that are divisible by both 3 and 5? Using this formula, you could implement a solution to Problem 30 using a constant number of steps, regardless of how large n gets. A solution to Problem 30 using a loop requires a linear number of steps, which grows with n.

(Required) Problem 31. Write a program that accept an integer n as user input, then prints a right triangle and its reflection. For example, if the user $n = 3$, the following shape is printed.

```
*
**
***
**
*
```

(Advanced) Problem 4. Modify your program in Problem 31 to center the shape. That is, if the user enters $n = 3$, the following shape is printed:

```
  *
 ***
*****
 ***
  *
```

Note: Unlike Problem 31, there are an odd number of stars on each row.

6 Number Theory

Number Theory is one of the oldest branches of mathematics, with results dating back thousands of years. An incredibly pure and beautiful subject, Number Theory was considered useless until the 20th century with the development of public-key cryptography. In the most general sense, Number Theory studies various sets of numbers. The positive integers, particularly the prime numbers, are of particular importance. The prime numbers are the building blocks of the positive integers, which is the *Fundamental Theorem of Arithmetic*. Determining the prime factorization of a positive integer is a computationally difficult problem, though not known to be NP-Complete. Cryptography and computational number theory are active areas of research in complexity theory and quantum computing, so they are worth mentioning here. It is quite possible to make significant headway in Number Theory with three tools: the Euclidean algorithm, the Pigeonhole Principle, and the Chinese Remainder Theorem.

We begin with number base conversion before discussing divisibility.

The exposition in this section has benefitted from [6, 10].

6.1 Number Base Conversion

In this section, we discuss how to convert between binary and decimal representations of non-negative integers. Consider the number 123. Here, the 3 belongs to the 1's place, the 2 belongs to the 10's place, and the 1 belongs to the hundred's place. So we may write:

$$\begin{aligned} 123 &= 1 * 100 + 2 * 10 + 3 * 1 \\ &= 1 * 10^2 + 2 * 10^1 + 3 * 10^0. \end{aligned}$$

Here, 10 is the base, so we say that 123 is written in base-10 or decimal. This is denoted 123_{10}. As humans, we are used to dealing with numbers in base-10. However, computers represent numbers in binary. Therefore, it is important to understand how to convert between binary and decimal representations of a number.

Now in order to represent a number in binary, the goal is to write the number as powers of 2. Consider the binary number 101_2. Here, 1 belongs to the 1's place, 0 belongs to the 2's place, and 1 belongs to the 4's place. Converting 101_2 to its base-10 representation simply amounts to adding up the non-zero powers of 2. So:

$$\begin{aligned} 101_2 &= (1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10} \\ &= (4 + 1)_{10} \\ &= 5_{10}. \end{aligned}$$

Note that when we represent a number in base-10, we may draw from the digits $0, 1, \ldots, 9$. There is no single digit to represent, for instance, $11, 12$, or 254. Similarly, only 0 and 1 are the only digits permitted in the binary representations of a number.

We now discuss converting the base-10 representation of a number to binary. Recall that the long division procedure provides both a quotient and a remainder. We iteratively perform long division by 2 on the quotients, and then string together these remainders. This sequence of remainders constitutes the binary representation of the number.

Example 32. We convert 11_{10} to binary.

- We first note that $11 = 2(5) + 1$. That is, $11/2 = 5\,R1$. The remainder belongs to the 2^0's place.
- Now $5 = 2(2) + 1$. That is, $5/2 = 2\,R1$. The remainder belongs to the 2^1's place.
- Now $2 = 2(1) + 0$. That is, $2/2 = 1\,R0$. The remainder belongs to the 2^2's place.
- Finally, $1 = 2(0) + 1$. That is, $2/1 = 0\,R1$. The remainder belongs to the 2^3's place.

So $11_{10} = 1011_2$. It is easy to check that our conversion is correct. Observe that:

$$\begin{aligned} 1011_2 &= (1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} \\ &= (8 + 0 + 2 + 1)_{10} \\ &= 11_{10}. \end{aligned}$$

6.1.1 Two's Complement

In this section, we discuss how to represent negative numbers in binary using Two's Complement. We proceed first by representing the positive number in binary. Next, we invert the bits and then add 1. Note that as we invert the bits, it is necessary to specify the number of bits used to represent the number. Consider the following example.

Example 33. Suppose we want an 8-bit representation of -30_{10}. We start by representing 30_{10} in binary:

$$30_{10} = 0001\,1110_2.$$

Next we invert the bits, to obtain: $1110\,0001_2$. Adding 1 to the result, we obtain: $1110\,0010_2$.

6.1.2 Exercises

(Required) Problem 32. Convert 23_{10} to binary.

(Required) Problem 33. Convert -45 to binary using an 8-bit representation.

(Required) Problem 34. Convert 110110_2 to decimal.

(Required) Problem 35. Write a program that takes as input an integer n, which is a decimal integer. Your program then prints off each digit, one at a time. You may only use the division `/` and modulus `%` operators.

(Required) Problem 36. Write a program that converts a binary number to decimal.

(Required) Problem 37. Write a program that converts a decimal number to binary.

6.2 Divisibility

Intuitively, we say that a divides b if b is a multiple of a. This is formalized as follows.

Definition 10 (Divisibility)**.** Let $a, b \in \mathbb{Z}$. We say that a divides b, denoted $a|b$ if there exists a $q \in \mathbb{Z}$ such that $aq = b$. If a does not divide b, we denote this $a \nmid b$.

Example 34. Suppose $a = 2$ and $b = 4$. We observe that $2|4$, selecting $q = 2$. However, $3 \nmid 5$.

Lemma 6.1. *Let a, b, c be integers such that $a|b$ and $b|c$. Then $a|c$.*

Proof. As $a|b$ and $b|c$, there exist integers h, k such that $ah = b$ and $bk = c$. It follows that $(ah)k = a(hk) = c$. So $a|c$. □

The division algorithm is the next big result. Recall the long division procedure from grade school. This procedure gives us the integers modulo n. It also implies the Euclidean algorithm, as well as the correctness of our various number bases such as the binary number system and our standard decimal number system.

Theorem 6.1 (Division Algorithm)**.** *Let $a, b \in \mathbb{Z}$ such that $b > 0$. Then there are unique integers q and r such that $a = bq + r$ with $0 \leq r < b$. We refer to q as the quotient and r as the remainder.*

Example 35. We have $2|4$; which, by the division algorithm, we can write as $4 = 2(2) + 0$ setting $q = 2, r = 0$. For the example of $3 \nmid 5$, we write $5 = 3(1) + 2$, setting $q = 1, r = 2$.

The next result of interest concerns divisors. A natural question is to find common divisors between two integers. Of particular interest is the greatest common divisor and how to efficiently compute it. As it turns out, this is a computationally easy problem, taking $\mathcal{O}(\log(n))$ time to solve.

Definition 11 (Euclidean Algorithm)**.** The Euclidean Algorithm takes as input two positive integers a and b and returns their greatest common divisor. The Euclidean Algorithm works by repeatedly applying the Division Algorithm until the remainder is 0.

```
function gcd(integer a, integer b):
    if a < b:
        return gcd(b, a)
    while b ≠ 0:
        t := b
        b := a mod b
        a := t
    return a
```

Theorem 6.2. *The Euclidean Algorithm terminates and returns the greatest common divisor of the two inputs.*

Proof. Without loss of generality, suppose $a \geq b$. Otherwise, the first iteration swaps a and b. The procedure applies the division algorithm repeatedly, resulting in the following sequence of equations:

$$\begin{aligned}
a &= b \cdot q_0 + r_0 \\
b &= r_0 \cdot q_1 + r_1 \\
r_0 &= r_1 \cdot q_2 + r_2 \\
r_1 &= r_2 \cdot q_3 + r_3 \\
&\vdots \\
r_{n-2} &= r_{n-1} \cdot q_n + r_n \\
r_{n-1} &= r_n \cdot q_{n+1}.
\end{aligned}$$

By the division algorithm, we note that $0 \leq r_{n-1} < r_{n-2} < \ldots < r_1 < r_0 < b$. So the algorithm terminates.

Now let $a, b \in \mathbb{Z}^+$ satisfying $a \geq b$, and let r satisfy the Division Algorithm: $a = bq + r$. It suffices to show that $\gcd(a, b) = \gcd(b, r)$, as the Euclidean algorithm computes $\gcd(r_i, r_{i+1})$ for all $i \in [n-1]$. Let d be a common divisor of a and b. Then $d \big| (a - bq)$, which implies $d \big| r$. So d is a common divisor of b and r. Now let k be a common divisor of b and r. Then we have $k \big| (bq + r)$, which implies k is a common divisor of a and b. Thus, $\gcd(a, b) = \gcd(b, r)$. Now as $\gcd(r_n, 0) = r_n$, it follows that r_n is the greatest common divisor a and b. □

Example 36. We use the Euclidean Algorithm to compute $\gcd(16, 6)$.

- $16 = 6(2) + 4$ $(a_0 = 16, b_0 = 6, r_0 = 4)$
- $6 = 4(1) + 2$ $(a_1 = 6, b_1 = 4, r_1 = 2)$
- $4 = 2(2) + 0$ $(a_2 = 4, b_2 = 2, r_2 = 0)$

At iteration 3, we would have $b_3 = r_2 = 0$, so the algorithm terminates. We observe $\gcd(16, 6) = 2$.

6.2.1 Exercises

(Required) Problem 38. Using the Euclidean Algorithm, compute **by hand** $\gcd(16, 27)$.

(Required) Problem 39. Implement the Euclidean Algorithm.

(Required) Problem 40. Suppose that $a, b \geq 0$ are integers such that $a \big| b$ and $b \big| a$. Prove that $a = b$.

(Required) Problem 41. Suppose we remove the assumption from Problem 40 that $a, b \geq 0$. Is it still true that if $a \big| b$ and $b \big| a$, that $a = b$? If so, prove it. If not, provide a counterexample.

(Required) Problem 42. Explain why $2 \big| n(n+1)$, for any integer n.

(Required) Problem 43. Explain why $3 \big| n(n+1)(n+2)$, for any integer n.

(Required) Problem 44. Suppose a, b, c, d are integers, with $a \neq 0$ and $c \neq 0$. Suppose that $a \big| b$ and $c \big| d$. Prove that $ac \big| bd$.

6.3 Modular Arithmetic

The language of congruence and modular arithmetic was introduced by the German mathematician, Karl Gauss. Modular arithmetic enables us to work with the divisibility relation in the same manner as equations. Intuitively, modular arithmetic provides an algebraic structure arising from the remainders upon doing long division. This is formalized as follows.

Definition 12 (Congruence Relation)**.** Let $n > 0$ be an integer, and let a, b be integers. We say that a is *congruent* to b *modulo* n, denoted $a \equiv b \pmod{n}$, precisely if $n \big| (a - b)$.

Example 37. Take $n = 5$. Observe that 9 has a remainder of 4 after long division by 5, so $9 \equiv 4 \pmod{5}$. The congruence relation is also well-defined for negative numbers; we simply "go backwards". Observe that $-2 \equiv 3 \pmod{5}$.

Definition 13. Let $n > 0$ be an integer. The *integers modulo* n, denoted $\mathbb{Z}_n$ or $\mathbb{Z}/n\mathbb{Z}$, consists of the set $\{0, 1, \ldots, n-1\}$, along with the inherent operations of addition and multiplication. Addition works by adding two remainders, and then taking the remainder after long division by n. Multiplication works by multiplying two remainders, and then taking the remainder after long division by n.

Example 38. Let $n = 5$. The set $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$. We perform some basic arithmetic operations.

- Consider $3 + 1 \equiv 4 \pmod{5}$. Similarly, $3 - 1 \equiv 2 \pmod{5}$.
- Consider $3, 4 \in \mathbb{Z}_5$. In the integers, $3 + 4 = 7$. Now 7 has a remainder of 2, when dividing by 5. So $7 \equiv 2 \pmod{5}$.
- Consider $2, 3 \in \mathbb{Z}_5$. In the integers, $2 \cdot 3 = 6$. Now 6 has a remainder of 1, when dividing by 5. So $6 \equiv 1 \pmod{5}$.

So we can make sense of addition, subtraction, and multiplication in $\mathbb{Z}_n$. We next discuss division. In the real numbers, division is the same as multiplying by the reciprocal of a number. For example, we have that:

$$3/2 = 3 \cdot \frac{1}{2}.$$

Here, $\frac{1}{2}$ is the multiplicative inverse of 2, which satisfies $2 \cdot \frac{1}{2} = 1$. The notion of a multiplicative inverse is formalized as follows.

Definition 14. Let x be a number. We say that y is the *multiplicative inverse* of x if $xy = yx = 1$. We denote that y is a multiplicative inverse of x using the notation $y = x^{-1}$.

We note that 0 has no multiplicative inverse.

Example 39. Take $n = 8$.

- Observe that $3 \in \mathbb{Z}_8$ has a multiplicative inverse. In particular, 3 is its own multiplicative inverse. That is, $3 \cdot 3 \equiv 1 \pmod{8}$.
- Observe that $4 \in \mathbb{Z}_8$ has no multiplicative inverse. In particular, if $x \in \mathbb{Z}_8$ shares a common factor with 8, then x is not invertible modulo 8. So $0, 2, 4, 6$ do not have inverses modulo 8.

The Euclidean Algorithm can be used to compute modular inverses, namely by working backwards after determining the greatest common divisor. We refer to this as the Extended Euclidean Algorithm. Formally, suppose we wish to compute $a^{-1} \pmod{b}$. The Extended Euclidean Algorithm returns a linear equation of the form:

$$ax + by = \gcd(a, b).$$

Here, $a^{-1} = x$. Now in order for a to be invertible $\pmod{b}$, $\gcd(a, b) = 1$.

Example 40. Suppose we wish to determine the multiplicative inverse of 24 (mod 13). We first use the Euclidean algorithm to determine $\gcd(24, 13)$:

$$24 = 13 + 11$$
$$13 = 11 + 2$$
$$11 = 2(5) + 1$$
$$2 = 1(2)$$

So $\gcd(24, 13) = 1$. Now working backwards, we have:

$$1 = 11 - 2(5)$$
$$1 = 11 - (13 - 11)(5) = 6(11) - 13(5)$$
$$1 = 6(24 - 13) - 13(5) = 6(24) - 11(13)$$

So $24(6) + 13(-11) = 1$. We conclude that the multiplicative inverse of 24, modulo 13, is 6. That is:

$$24(6) \equiv 1 \pmod{13}.$$

6.3.1 Exercises

(Required) Problem 45. Find the positive integers m such that each of the congruences is true.

(a) $27 \equiv 5 \pmod{m}$.

(b) $1000 \equiv 1 \pmod{m}$.

(c) $1331 \equiv 0 \pmod{m}$.

(Required) Problem 46. Evaluate $2^{644} \pmod{645}$.

(Required) Problem 47. Show that if a is an odd integer, then $a^2 \equiv 1 \pmod{8}$.

(Required) Problem 48. Using the Extended Euclidean Algorithm, determine the multiplicative inverse of 421 modulo 111.

6.4 Primality

Definition 15. Let $n > 1$ be an integer. We say that n is *prime* if its only positive divisors are itself and 1. We say that n is composite otherwise.

Example 41. The integers $2, 3, 5, 7, 13, 101$, and 163 are prime. However, $4 = 2 \cdot 2, 33 = 3 \cdot 11$, and $111 = 3 \cdot 37$ are all composite.

Lemma 6.2. *Every integer greater than* 1 *has a prime divisor.*

Proof. The proof is by contradiction. Suppose there exists an integer $n > 1$ with no prime divisor. Without loss of generality, suppose n is the smallest such integer. As n has no prime divisor, n is not prime. So n is composite. Therefore, we may write $n = ab$, where $1 < a \leq b < n$. As n is the least such composite number with no prime divisor, it follows that a and b both have prime divisors. Let p be a prime divisor of a. So p divides n, a contradiction. □

Using Lemma 6.2, we prove that there are infinitely many primes.

Theorem 6.3. *There are infinitely many primes.*

Proof. Suppose for a contradiction that there are only finitely many primes $p_1, \ldots, p_n$. Let $Q = p_1 \cdot p_2 \cdots p_n + 1$. Observe that for each $i \in \{1, \ldots, n\}$, p_i does not divide Q. By Lemma 6.2, there exists a prime p that divides Q. So p is not one of $p_1, \ldots, p_n$, a contradiction. □

Theorem 6.4. *If n is a composite integer, then n has a prime factor not exceeding $\sqrt{n}$.*

Proof. As n is composite, we may write $n = ab$, where $1 < a \leq b < n$. It must be the case that $a \leq \sqrt{n}$. Otherwise, $b \geq a > \sqrt{n}$, in which case $ab > \sqrt{n} \cdot \sqrt{n} = n$. Now by Lemma 6.2, a has a prime divisor p, which in turn divides n. □

6.4.1 Exercises

(Required) Problem 49. Using Theorem 6.4, implement a Java program to test if a number is prime.

(Required) Problem 50. A *Fermat number* is of the form $F_n = 2^{2^n} + 1$. You may assume that for any distinct i, j, $\gcd(F_i, F_j) = 1$. Using this fact, provide a proof that there exist infinitely many primes.

(Required) Problem 51. Show that any integer of the form $n^3 + 1$ is composite, except for 2.

7 Arrays and Strings

In this section, we discuss arrays and Strings. We begin with motivation. Suppose we have a gradebook application with 100 quiz grades. It is quite tedious to declare, store, and write code to manipulate 100 variables named `quiz1, quiz2, ..., quiz100`. An array is a construct that provides a means to effectively manage such variables, using an index for each block. More precisely, an array is a container Object, which contains a fixed number of values of a given type. Note that once the array is created, its length is fixed and cannot be changed. The real power of arrays is that manipulating the values at arbitrary indices in the array. That is, loops are effective tools for succinctly manipulating arrays.

After the discussion of arrays, Strings are next introduced. We have already worked some with Strings, which provide a means for storing text rather than numbers or single characters. Internally, the String class manages an array storing `char` values. The String class then provides methods to perform desired manipulations. Thus, while the array concepts transfer almost entirely when working with Strings, there are important syntactical differences when expressing the logic in Java.

7.1 Arrays

Recall that an array is a container which stores a fixed number of values, all of the same given type. An array can be declared in one of two ways: (i) by specifying the length upon construction, or (ii) by populating the array with values upon declaration.

In order to declare an array by length, we need the type and the length. The syntax is as follows.

```
type[] name = new type[length];
```

Note that the square brackets can be attached to the `type` or the `name`.

```
type name[] = new type[length];
```

We note that arrays are indexed from $0, 1, 2, \ldots,$ `length` $- 1$. So if an array has 5 elements, the indices are $0, 1, 2, 3,$ and 4.

Example 42. The following example creates a `int` array and populates it with the first 5 odd numbers.

```
int[] arr = new int[5];
arr[0] = 1;
arr[1] = 3;
arr[2] = 5;
arr[3] = 7;
arr[4] = 9;
```

Alternatively, we may populate the array with values as follows:

```
int[] arr = {1, 3, 5, 7, 9};
```

When there are only five elements, it is quite feasible to manually populate the array. If the array instead had 1000 elements for instance, populating the array manually is tedious. Rather, loops can be used to automate such a task.

Example 43. The following example of code populates a `int` with the first thousand odd numbers. Note that the length is an attribute of the array, which we can access using the syntax `arr.length`, where `arr` is the name of our array. Using `arr.length` over the magic constant 1000 better allows us to understand the logic of the loop. Writing code where the logic purpose and logic are evident is of key importance when debugging one's program, as well as modifying code at a later point.

```
int[] arr = new int[1000];
for(int i = 0; i < arr.length; i++){
   arr[i] = 2 * i + 1;
}
```

We note that when declaring an array based on the length, the elements in the array are initialized to their default values. For numerical primitives (`byte`, `short`, `int`, `long`, `float`, and `double`), the default value is 0. For `char`, the default value is an empty character. The default value of `boolean` is `false`. Objects, such as Strings, have a default value of `null`. So for instance, the following line of code will declare a String array of 10 elements, where each element is initialized to `null`.

```
String[] arr = new String[10];
```

It is often desirable to output the contents of an array. Consider the following code segment:

```
int[] arr = new int[1000];
for(int i = 0; i < arr.length; i++){
   arr[i] = 2 * i + 1;
}
System.out.println(arr);
```

The output of this program resembles the following: `[I@1540e19d`. This is the array's hashcode, which represents its location in memory. In order to print out the elements of an array, we can print the contents by examining each element.

```
for(int i = 0; i < arr.length; i++){
   System.out.print(arr[i] + " ");
}
```

7.1.1 Exercises

(Required) Problem 52.

(a) Write a program that populates a `int[]` with random elements. Your program should then prompt the user for a value, with the goal of checking if said value is in the array. The program should then inform the user as to whether the value they provided belongs to the random array.

(b) In the worst case, how many comparisons (in terms of n, the length of the array) are needed to determine if the value the user entered belongs to the array?

(Advanced) Problem 5. Suppose the array from Problem 52 is sorted prior to prompting the user for input.

(a) Provide an algorithm to search the array so that in the worst case, fewer comparisons are required than in part (b) of Problem 52. Outline your algorithm in sufficient detail so that one of your classmates could easily understand and implement your solution.

(b) In the worst case, how many comparisons (in terms of n, the length of the array) are needed to determine if the value the user entered belongs to the array?

(Required) Problem 53. Write a program that prompts the user for the number of elements in the `int` array, and then prompts the user for the array elements. Your program then provides the user with the following options:

- Find the largest element in the array.
- Find the smallest element in the array.
- Reverse the contents of the array.

(Required) Problem 54. Write a program that simulates rolling a 6-sided die 1000 times. Using an array, count the number of times each value on the die appears. Output this result to the user at the end.

(Advanced) Problem 6. Write a program that accepts as user input an integer $n \geq 0$. Your program should print every prime number that is at most n. For this problem, please use a `boolean` array of length $n + 1$, where the element at index i is `true` precisely if i is prime. Note that 0 and 1 are **not** considered prime.

7.2 Strings

A String is an object which stores text. In Java, Strings are implemented via the String class, which manages a `char[]` internally. As a result, the same logic used to work with arrays applies directly to manipulating Strings. As the String class prohibits direct access to the underlying `char[]`, there are differences in syntax when working with Strings as opposed to arrays. In this section, we introduce this syntax as well as highlight the conceptual similarities between Strings and arrays.

- **String charAt() Method.** As with arrays, it is useful to access individual characters in a String. This is accomplished using the String `charAt(int index)` method. For instance, if we wanted to access the first character in the String `str` declared above, we access `str.charAt(0)`.

  Example 44. Consider the following example.

  ```
  String str = "This is some text.";
  char firstChar = str.charAt(0);
  System.out.println(firstChar);
  ```

  Note that the output of this code sample is `T`.

  Now suppose we wish to print out every character in a given String. To do this, we need access to the String's length, whichis obtained using the String `length()` method.

  Example 45. Consider the following example.

  ```
  String str = "This is some text.";
  for(int i = 0; i < str.length(); i++){
     System.out.println(str.charAt(i));
  }
  ```

- **Concatenation.** Next, we discuss concatenation, in which we take two Strings and glue them together.

  Example 46. Consider the following example.

  ```
  String hello = "Hello";
  String world = "World";
  String helloWorldNoSpace = hello + world;
  String helloWorldSpace = hello + " " + space;
  ```

  Note that `helloWorldNoSpace` holds the String `"HelloWorld"`, while `helloWorldSpace` holds the text `"Hello World"`.

- **Comparing Strings.** Recall that when comparing primitives, the `==` operator is used. For Strings, the `==` operator is not an effective tool. The `==` operator compares the values of the two variables. For primitives, the literals are the actual values. However, for Objects such as Strings, the literal values are the locations in memory. In order to compare Strings based on the text rather than memory locations, we may use the String `equals()` and `equalsIgnoreCase()` methods.

  Example 47. Consider the following example.

```
String x = "Hello";
String y = "hEllO";
if(x.equals("Hello") ){
   System.out.println("x is equal to Hello.");
}
System.out.println(x + " and " + y + " are equal:  " + x.equals(y));
System.out.println(x + " and " + y + " are equalIgnoreCase:  " + x.equalsIgnoreCase(y));
```

  The output of this code is:

```
x is equal to Hello.
Hello and hEllO are equal:  false
Hello and hEllO are equalIgnoreCase:  true
```

- **Searching for Substrings.** The String class provides a number of methods for searching for substrings within a given String. We introduce the `indexOf(String str)` and `indexOf(String str, int offset)`. The `indexOf(String str)` method returns the index where the first occurrence of `str`. If `str` is not contained in the given String, the `indexOf(String str)` method returns -1. The `indexOf(String str, int index)` method operates similarly, with the exception that it searches for `str` after the supplied offset.

  Example 48. Consider the following example.

```
String str = "abcabc";
System.out.println("First index of ab:  " + str.indexOf("ab"));
System.out.println("Second index of ab:  " + str.indexOf("ab", 1));
```

  The output of this code sample is:

```
First index of ab:  0
Second index of ab:  3
```

- **Modifying Strings.** We next discuss how to modify the contents of a String. We first note that the String class is `immutable`. This means that any changes we make don't affect the given String; rather, a new String is returned with the updated changes. Consider the `replace(CharSequence substr, CharSequence replacement)` method, which returns a new String where each instance of `substr` is replaced with `replacement`. Note that a String is a CharSequence, so the `replace()` method accepts String parameters.

  Example 49. Consider the following example.

```
String str = "abcabc";
System.out.println(str.replace("a", "e"));
System.out.println(str);
```

Note that the ouput of this code segment is:

```
ebcebc
abcabc
```

We conclude with the `trim()` method, which removes all leading and trailing whitespace, though not any of the interior whitespace.

Example 50. Consider the following example.

```
String str = " ef gh ";
System.out.println(str.trim());
System.out.println(str);
```

The output of this code segment is:

```
ef gh
   ef gh
```

7.2.1 Exercises

(Required) Problem 55. A *palindrome* is a phrase that is spelled the same forwards and backwards, ignoring punctuation and spacing. For example, `racecar` and `A man, a plan, a canal, Panama` are both palindromes.

(a) Write a program that takes as input a String and then informs the user as to whether their String is a palindrome. Do so without using built-in Java functionality that trivializes the problems. Arrays, String class functionality, and loops are fine to use for this part.

(b) Recall that the String class is immutable. That is, any changes we wish to make result in a new String object being returned. The StringBuilder class is like the String class, except StringBuilder objects are mutable. Take some time to learn about the StringBuilder class (see the Tutoracles and documentation). Does StringBuilder have functionality that would be helpful in part (a)? If so, repeat part (a) using such functionality.

(Required) Problem 56. The Caesar cipher is a cryptosystem that takes as input the plaintext and produces as output the ciphertext by shifting each letter of the plaintext 3 characters over in the alphabet. So the plaintext `ABC` is encrypted to `DEF`. Note that the shift is cyclic; so for example, `Z` is encrypted to `C`. Your job is to implement the Caesar cipher.

You may find it helpful to treat `char` values as numbers. The correlation between `char` values and `int` values is given by the ASCII table.

(Required) Problem 57. Suppose you are given three strings: `x`, `y`, and `z`. The String `z` is said to be a shuffle of `x` and `y` if it can be formed by interleaving the characters of `x` and `y` in a way that maintains the left to right ordering of the characters from each string. For example, given:

```
String x = "abc";
String y = "def";
```

`String third = "dabecf";` is a valid shuffle since it preserves the character ordering of the two strings. Write a program that takes as input three Strings and determines whether the third String is a valid shuffle of the first two Strings.

8 Combinatorics

Problems from Combinatorics and Graph Theory arise frequently in computer science, especially the more theoretical areas. Combinatorics studies finite and countable discrete structures. We will focus on techniques for counting structures satisfying a given property, which is the goal of enumerative combinatorics. Computer scientists are often interested in finding optimal structures (combinatorial optimization). Many of these problems are computationally difficult. Addtionally, many enumerative problems also suffer from issues of complexity. The complexity class `#P` deals with the complexity of enumeration.

The exposition in this section has benefitted from several sources, including [6, 7].

8.1 Set Theory

Definition 16 (Set)**.** A *set* is collection of distinct elements, where the order in which the elements are listed does not matter. The size of a set S, denoted $|S|$, is known as its *cardinality* or *order*. The members of a set are referred to as its elements. We denote membership of x in S as $x \in S$. Similarly, if x is not in S, we denote $x \notin S$.

Example 51. Common examples of sets include $\mathbb{R}, \mathbb{Q}$ and $\mathbb{Z}$. The sets $\mathbb{R}^+, \mathbb{Q}^+$ and $\mathbb{Z}^+$ denote the strictly positive elements of the reals, rationals, and integers respectively. We denote $\mathbb{N} = \{0, 1, \ldots\}$. Let $n \in \mathbb{Z}^+$ and denote $[n] = \{1, \ldots, n\}$.

We now review several basic set operations, as well as the power set.

Definition 17. Set Union Let A, B be sets. Then the *union* of A and B, denoted $A \cup B$ is the set:

$$A \cup B := \{x : x \in A \text{ or } x \in B\}$$

Example 52. Let $A = \{1, 2, 3\}$ and $B = \{4, 5, 6\}$. Then $A \cup B = \{1, 2, 3, 4, 5, 6\}$. Now let $C = \{3, 4\}$. So $A \cup C = \{1, 2, 3, 4\}$. Notice that even though $3 \in A$ and $3 \in C$, we only include 3 once in $A \cup C$. Remember that a set contains *distinct* elements.

Definition 18. Set Intersection Let A, B be sets. Then the *intersection* of A and B, denoted $A \cap B$ is the set:

$$A \cap B := \{x : x \in A \text{ and } x \in B\}$$

Example 53. Let $A = \{1, 2, 3\}$ and $B = \{1, 3, 5\}$. Then $A \cap B = \{1, 3\}$. Now let $C = \{4\}$. So $A \cap C = \emptyset$.

Definition 19 (Symmetric Difference)**.** Let A, B be sets. Then the *symmetric difference* of A and B, denoted $A \triangle B$ is the set:

$$A \triangle B := \{x : x \in A \text{ or } x \in B, \text{ but } x \notin A \cap B\}$$

Example 54. Let $A = \{1, 2, 3\}$ and $B = \{1, 3, 5\}$. Then $A \triangle B = \{2, 5\}$.

For our next two definitions, we let U be our *universe*. That is, let U be a set. Any sets we consider are subsets of U.

Definition 20 (Set Complementation)**.** Let A be a set contained in our universe U. The *complement* of A, denoted A^C, A', or $\overline{A}$, is the set:

$$\overline{A} := \{x \in U : x \notin A\}$$

Example 55. Let $U = [5]$, and let $A = \{1, 2, 4\}$. Then $\overline{A} = \{3, 5\}$.

Definition 21 (Set Difference)**.** Let A, B be sets contained in our universe U. The *difference* of A and B, denoted $A \setminus B$ or $A - B$, is the set:

$$A \setminus B = \{x : x \in A \text{ and } x \notin B\}$$

Example 56. Let $U = [5]$, $A = \{1, 2, 3\}$ and $B = \{1, 2\}$. Then $A \setminus B = \{3\}$.

Remark: The Set Difference operation is frequently known as the *relative complement*, as we are taking the complement of B relative to A rather than with respect to the universe U.

Definition 22 (Cartesian Product)**.** Let A, B be sets. The *Cartesian product* of A and B, denoted $A \times B$, is the set:

$$A \times B := \{(a, b) : a \in A, b \in B\}$$

Example 57. Let $A = \{1, 2, 3\}$ and $B = \{a, b\}$. Then $A \times B = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$.

Definition 23 (Power Set)**.** Let S be a set. The *power set* of S, denoted 2^S or $\mathcal{P}(S)$, is the set of all subsets of S. Formally:

$$2^S := \{A : A \subset S\}$$

Example 58. Let $S = \{1, 2, 3\}$. So $2^S = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Remark: For finite sets S, $|2^S| = 2^{|S|}$; hence, the choice of notation.

Definition 24 (Subset)**.** Let A, B be sets. A is said to be a *subset* of B if for every $x \in A$, we have $x \in B$ as well. This is denoted $A \subset B$ (equivocally, $A \subseteq B$). Note that B is a *superset* of A.

Example 59. Let $A = [3], B = [6], C = \{2, 3, 5\}$. So we have $A \subset B$ and $C \subset B$. However, $A \not\subset C$ as $1 \notin C$; and $C \not\subset A$, as $5 \notin A$.

8.1.1 Exercises

(Required) Problem 58. Let $A = \{0, 2, 4, 6, 8\}, B = \{1, 3, 5, 7\}$ and $C = \{2, 8, 4\}$. Find:

(a) $A \cup B$

(b) $A \setminus B$

(c) $A \setminus C$

(d) $A \cap C$

(e) $A \setminus (B \cup C)$

(Required) Problem 59. Let $A = \{1, 2, 3\}$ and $B = \{2, 4\}$. Find:

(a) $(A \times B) \cap (B \times B)$

(b) $(A \times B) \setminus (B \times B)$

(c) $(A \cap B) \times A$

(d) $2^A \cap 2^B$

(e) $2^{A \cap B}$

(f) $2^A \setminus 2^B$

(Required) Problem 60. Let A and B be sets. Determine which of the following expressions evaluates to $\emptyset$. Justify your answer. If an expression does not evaluate to $\emptyset$, provide a counterexample.

(a) $A \cap \emptyset$

(b) $(A \setminus B) \cap (B \setminus A)$

(c) $(A \setminus B) \setminus (B \setminus A)$

8.2 Rules of Sum and Product

We begin with the Rule of Product and Rule of Sum. These two rules provide us the means to derive the basic combinatorial formulas, as well as enumerate more complicated sets. Let's discuss the intuition. The Rule of Sum allows us to count the number of elements in disjoint sets. Suppose you go to the store and have enough money for a single candy bar. If there are only five candy bars on the shelf, then you have five choices. Your choices are all disjoint- you can choose a Snickers or a Twix, but not both. Now suppose you and a friend go to the store and each have enough to purchase a single candy bar. Each of you only has five selections, and your selection of a candy bar does not affect your friend's ability to select a candy bar (and vice versa). This is the idea behind independence. The Rule of Product tells us that when two events are independent, that we multiply the number of outcomes for each to determine the number of total outcomes. In other words, there are 25 ways for you and your friend to each purchase one candy bar. We now formalize these notions.

Theorem 8.1 (Rule of Sum)**.** *Let X and Y be disjoint sets. Then $|X \cup Y| = |X| + |Y|$.*

Example 60. Suppose we can go to either McDonald's or KFC for dinner. McDonald's has 12 combo meals, while KFC has 10 combo meals. The Rule of Sum provides that we have $12+10 = 22$ combo meals from which to choose.

Frequently, two sets will not be disjoint. In such cases, we use the *Principle of Inclusion-Exclusion*. When we have two sets, we modify the formula in the Rule of Sum simply subtract out the number of elements the two sets share, so that these shared elements are not double counted. Generalizing the Rule of Sum, we have:

$$|X \cup Y| = |X| + |Y| - |X \cap Y|.$$

Consider the following example.

Example 61. Amazon has 132,000 cookbooks. Of these, 20,000 were on regional cooking, 5,000 were on vegetarian cooking, and 24,000 were on either regional or vegetarian cooking (or both). How many of these books were on neither regional nor vegetarian cooking?

Let A be the set of books on regional cooking, and let B be the set of vegetarian cookbooks. We want to determine $|(A \cap B)'|$. First, we determine $|A \cap B|$. We have that $|A| = 20000, |B| = 5000$, and $|A \cup B| = 24000$. Now:

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

So: $|A \cap B| = 1,000$. Thus, $|(A \cap B)'| = 132,000 - 1,000 = 131,000$.

Theorem 8.2 (Rule of Product)**.** *Let X and Y be sets. Then $|X \times Y| = |X| \cdot |Y|$.*

We now consider the word problem. We fix an alphabet, which is a finite set of characters denoted Σ. Some examples are $\Sigma = \{0, 1\}, \Sigma = \{a, b, c\}$, and the English alphabet. A word is a sequence (order matters) of characters formed from elements in Σ. Formally, a word of length n is an element $\omega \in \Sigma^n$. The Rule of Product immediately implies that there exist $|\Sigma|^n$ such words of length n. Some additional details will be provided in proof of the next proposition.

Proposition 8.1. *Let Σ be an alphabet. There are $|\Sigma|^n$ words of length n over Σ.*

Proof. We consider the n positions of $\omega \in \Sigma^n$ given by $\omega_1\omega_2 \ldots \omega_n$. Each $\omega_i \in \Sigma$. The selection of each ω_i is independent of the remaining characters in ω. So by rule of product, we multiply $\prod_{i=1}^{n} |\Sigma| = |\Sigma|^n$ such words. □

Example 62. If we consider $\Sigma = \{0, 1\}$, then Proposition 8.1 implies that there exist 2^n binary strings of length n.

Example 63. A three-course meal at a restaurant consists of an appetizer, main course, and dessert. There are 5 appetizers, 34 entrees, and 10 desserts. We may view each three-course meal as an ordered triple of the form:

(Appetizer, Main Course, Dessert).

There are 5 choices for an appetizer, 34 choices for a main course, and 10 choices for a dessert. By the Rule of Product, we have that there are: $5 \times 34 \times 10$ possible three-course meals.

8.2.1 Exercises

(Required) Problem 61. The dining hall offers a total of 14 desserts, of which 8 have ice cream as a main ingredient and 9 have fruit as a main ingredient. Assuming that all of them have either ice cream, fruit, or both a a main ingredient, how many have both?

(Required) Problem 62. In a study of Tibetan children, a total of 1556 children were examined. Of these, 615 had cavities. Of the 1313 children living in non-urban areas, 504 had cavities.

(a) How many children living in urban areas had cavities?

(b) How many children living in urban areas did not have cavities?

(Required) Problem 63. When Baskin-Robbins was founded in 1945, it made 31 different flavors of ice cream. If you had a choice of having your ice cream in a cone, a cup, or a sundae, how many different desserts could you have?

(Required) Problem 64. Of the 4700 students at Medium Suburban College (MSC), 50 play soccer, 60 play lacrosse, and 96 play football. Only 4 students play both soccer and locrosse, 6 play soccer and football, and 16 play both lacrosse and football. No students play all three sports.

(a) How many students play no sports?

(b) How many students play only soccer?

(Required) Problem 65. Determine the number of binary strings of length 8. Do the same for ternary strings (strings over the alphabet $\Sigma = \{0, 1, 2\}$).

(Required) Problem 66. While selecting candy for students in his class, Professor Murphy must choose between gummy candy and licorice nibs. Gummy candy comes in three sizes, while packets of licorice nibs come in two sizes. If he chooses gummy candy, he must select gummy bears, gummy worms, or gummy dinos. If he chooses licorice nibs, he must choose between black and red. How many choices does he have?

(Required) Problem 67. How many seven-digit phone numbers do not begin with one of the prefixes: $1, 911, 411$, or 555?

(Required) Problem 68. A Social-Security Number (SSN) is a sequence of 9 digits.

(a) How many SSNs are possible?

(b) How many SSNs begin with either 023 or 003?

(c) How many SSNs are possible if no two adjacent digits are permitted? [**Note:** So $235 - 93 - 2345$ is permissible, but $126 - 67 - 8189$ is not permissible as there are two consecutive 6's.]

8.3 Permutations and Combinations

With the Rule of Product and Rule of Sum in mind, we are ready to begin with elementary counting techniques. The first combinatorial object of interest is the permutation. Intuitively, a permutation is a reordering of a sequence. Formally, we define it as follows.

Definition 25 (Permutation)**.** Let X be a *finite* set. A permutation is a one-to-one function from X to itself, $\pi : X \to X$.

Example 64. Let $X = \{1, 2, 3\}$. The following are permutations of X: 123, 321, and 213. However, 331 is not a permutation of X since 3 is repeated twice. For the permutation 213, the function $\pi : X \to X$ maps $\pi(1) = 2, \pi(2) = 1, \pi(3) = 3$.

The natural question to ask is how many permutations exist on n elements (where $n \in \mathbb{N}$). The answer is $n!$ permutations, where $n!$ is the product $n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$. Note that $0! = 1$. We prove this formally using the Rule of Product.

Proposition 8.2. *Let X be an n-element set. There are $n!$ permutations of X.*

Proof. Without loss of generality $X = [n]$. We define a permutation $\pi : X \to X$. Observe that $\pi(1)$ can map to any of the n elements in X. As π is a bijection, only 1 can map to $\pi(1)$. This leaves $n-1$ elements, to which $\pi(2)$ can map. The selection of $\pi(2)$ is independent of $\pi(1)$; so by Rule of Product, we multiply to obtain $n(n-1)$ ways of selecting $\pi(1)$ and $\pi(2)$. Proceeding in this manner, there are $\prod_{i=1}^{n} i = n!$ possible permutations. □

Proposition 8.3. $0! = 1$

Proof. There exists exactly one function $f : \emptyset \to \emptyset$. □

Remark: When we have an n element set and want to permute $r \leq n$ elements, there are $P(n,r) = \frac{n!}{(n-r)!}$ such restricted permutations.

Example 65. Suppose we have four songs we wish to play in sequence. There are 4! ways of constructing a playlist.

Example 66. Suppose we want to construct a playlist of 10 movies, chosen from 90 movies available on Netflix. There are:

$$P(90,10) = \frac{90!}{(90-10)!} = \frac{90!}{80!}$$

such ways of constructing a playlist.

On the surface, the permutation and word problems may be hard to distinguish. The key difference between the two is the notion of replacement. In the permutation problem, we are given a fixed set of distinct elements. Each element is to be used precisely once in the sequence. In contrast, the word problem provides a fixed set of letters, each of which can be used for none, any, or all of the word's characters. That is, each time a letter is chosen from the alphabet, it is replaced by another instance of itsef to be used later. So 000 is a word but not a permutation because 0 is repeated.

We now discuss combination problems, which describe discrete structures where order does not matter. The simplest of these problems is the subset problem. Much like the permutation problem, the subset problem considers a set X with $|X| = n$. We seek to determine how many subsets of order k from X exist. For example, if $X = \{1,2,3\}$, there exist three subsets of X with two elements. These subsets are $\{1,2\}, \{1,3\}$, and $\{2,3\}$.

Definition 26 (Binomial Coefficient)**.** Let X be a set with n elements. Let $k \in \mathbb{N}$. We denote the *binomial coefficient* $\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$, which is read *$n$ choose k*. We denote $\binom{X}{k}$ as the set of all k-element subsets of X. That is, $\binom{X}{k} = \{S \subset X : |S| = k\}$.

Proposition 8.4. *The binomial coefficient $\binom{n}{k}$ counts the number of k-element subsets of an n-element set.*

Example 67. Suppose we have 10 people from which to choose a 3-person committee. There are $\binom{10}{3}$ ways of selecting such a committee.

8.3.1 Exercises

(Required) Problem 69. How many five-letter sequences are possible using b, o, g, e, y once each?

(Required) Problem 70. How many unordered sets are there that contain 3 objects from the set $\{1, 2, \ldots, 7\}$?

(Required) Problem 71. How many ordered sequences are there that contain 3 objects from the set $\{1, 2, \ldots, 7\}$, assuming that repeated elements are not allowed?

(Required) Problem 72. How many ways are there to pick a 7-person basketball team from 20 possible players? How many teams if the weakest player and strongest player must be on the team?

(Required) Problem 73. How many ways are there to distribute six different books among 12 children if no child gets more than one book?

(Advanced) Problem 7. For each of the following, determine the number of such poker hands.

(a) A *Royal Flush* consists of ranks 10, J, Q, K, A, all of the same suit.

(b) A *Straight Flush* consists of five cards with values in a row, all of the same suit. An Ace may be considered as the highest rank (so Ace immediately follows king) or lowest rank (so Ace immediately precedes 2), but not both. So for example, A, 2, 3, 4, 5 (with all cards of the same suit) is a straight flush, but Q, K, A, 2, 3 is **not** a straight flush.

(c) A *Straight* consists of five cards with consecutive ranks, not all of the same suit. Just as in a straight flush, the Ace may be considered as the highest rank or lowest rank, but not both.

(d) A *Flush* consists of five cards, all of the same suit.

(e) A *Four of a Kind* consists of four cards of one rank, and a fifth card of a different rank.

(f) A *Full House* consists of three cards of one value, and two cards whose values are the same as each other but different than the first three cards. (E.g., three Queens and two 10's).

(g) A *Two Pair* consists of two pairs, one pair of the same value and a second pair of a different value. The fifth card has yet a different value. (E.g., two 10's, two Queen's, and a King).

(h) A *One Pair* consists of a pair of one value, and then three additional cards each of a different value. (E.g., 10, 10, Ace, Jack, Queen).

(Challenge) Problem 1. A *derangement* is a permutation π such that for all x, $\pi(x) \neq x$. That is, π has no fixed points. Fix $n \in \mathbb{N}$. Determine the number of derangements on $[n]$. We denote this count as D_n.

9 Graph Theory

To quote Bud Brown, "Graph theory is a subject whose deceptive simplicity masks its vast applicability." Graph theory provides simple mathematical structures known as graphs to model the relations of various objects. The applications are numerous, including efficient storage of chemicals (graph coloring), optimal assignments (matchings), distribution networks (flows), efficient storage of data (tree-based data structures), and machine learning. In automata theory, we use directed graphs to provide a visual representation of our machines. Many elementary notions from graph theory, such as path-finding and walks, come up as a result. In complexity theory, many combinatorial optimization problems of interest are graph theoretic in nature. Therefore, it is important to discuss basic notions from graph theory. We begin with the basic definition of a graph.

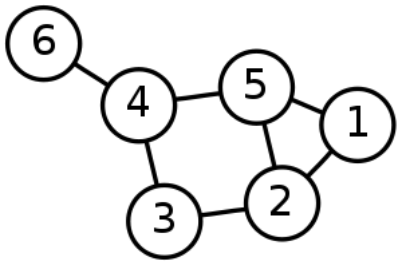
The exposition in this section has benefitted from [3, 6, 14].

9.1 Introduction to Graphs

Definition 27 (Simple Graph)**.** A simple graph is a two-tuple $G(V, E)$ where V is a set of vertices and $E \subset \binom{V}{2}$.

By convention, a simple graph is referred to as a *graph*, and an edge $\{i, j\}$ is written as ij. In simple graphs, $ij = ji$. Two vertices i, j are said to be *adjacent* if $ij \in E(G)$. Now let's consider an example of a graph.

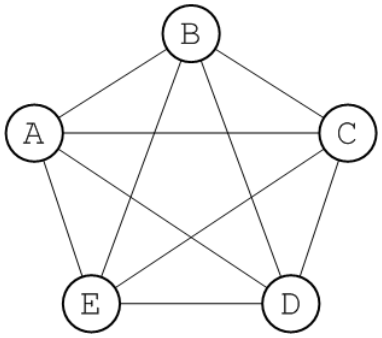
Example 68. Let $G(V, E)$ be the graph where $V = [6]$ and $E = \{12, 15, 23, 25, 34, 45, 46\}$. This graph is pictured below.



We now introduce several common classes of graphs.

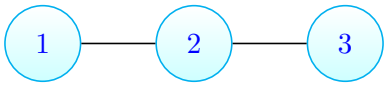
Definition 28 (Complete Graph)**.** The complete graph, denoted K_n, has the vertex set $V = [n]$ and edge set E which consists of **all** two-element subsets of V. That is, K_n has all possible edges between vertices.

Example 69. The complete graph on five vertices K_5 is pictured below.



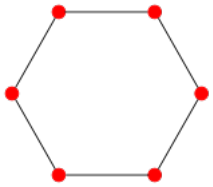
Definition 29 (Path Graph)**.** The path graph, dentoed P_n, has vertex set $V = [n]$ and the edge set $E = \{\{i, i+1\} : i \in [n-1]\}$.

Example 70. The path on three vertices P_3 is shown below.



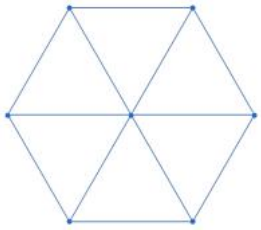
Definition 30 (Cycle Graph)**.** Let $n \geq 3$. The cycle graph, denoted C_n, has the vertex set $V = [n]$ and the edge set $E = \{\{i, i+1\} : i \in [n-1]\} \cup \{\{1, n\}\}$.

Example 71. Intuitively, C_n can be thought of as the regular n-gon. So C_3 is a triangle, C_4 is a quadrilateral, and C_5 is a pentagon. The graph C_6 is pictured below.



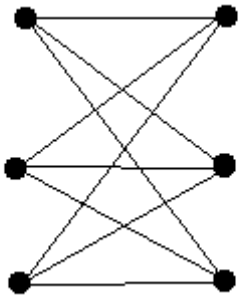
Definition 31 (Wheel Graph)**.** Let $n \geq 4$. The wheel graph, denoted W_n, is constructed by joining a vertex n to each vertex of C_{n-1}. So we take $C_{n-1} \dot{\cup} n$ and add the edges vn for each $v \in [n-1]$.

Example 72. The wheel graph on seven vertices W_7 is pictured below.



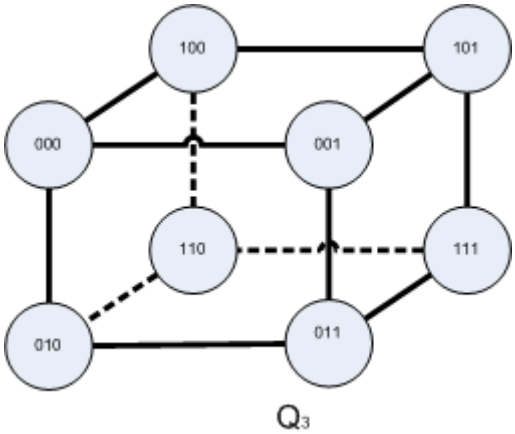
Definition 32 (Bipartite Graph)**.** A bipartite graph $G(V, E)$ has a vertex set $V = X \dot{\cup} Y$, with edge set $E \subset \{xy : x \in X, y \in Y\}$. That is, no two vertices in the same part of V are adjacent. So no two vertices in X are adjacent, and no two vertices in Y are adjacent.

Example 73. A common class of bipartite graphs include even-cycles C_{2n}. The complete bipartite graph is another common example. We denote the complete bipartite graph as $K_{m,n}$ which has vertex partitions $X \dot{\cup} Y$ where $|X| = m$ and $|Y| = n$. The edge set $E(K_{m,n}) = \{xy : x \in X, y \in Y\}$. The graph $K_{3,3}$ is pictured below.



Definition 33 (Hypercube)**.** The hypercube, denoted Q_n, has vertex set $V = \{0, 1\}^n$. Two vertices are adjacent if the binary strings differ in precisely one component.

Example 74. The hypercube Q_2 is isomorphic to C_4 (isomorphism roughly means that two graphs are the same, which we will formally define later). The hypercube Q_3 is pictured below.



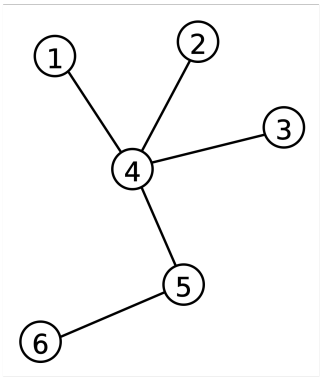
Definition 34 (Connected Graph)**.** A graph $G(V, E)$ is said to be connected if for every $u, v \in V(G)$, there exists a $u-v$ path in G. A graph is said to be *disconnected* if it is not connected; and each connected subgraph is known as a *component*.

Example 75. So far, every graph presented has been connected. If we take two disjoint copies of any of the above graphs, their union forms a disconnected graph.

Definition 35 (Tree)**.** A Tree is a connected, acyclic graph.

Example 76. A path is an example of a tree. Additional examples include the binary search tree, the binary heap, and spanning trees of graphs.

Example 77. The following is an example of a tree.



Definition 36 (Degree)**.** Let $G(V, E)$ be a graph and let $v \in V(G)$. The degree of v, dentoed $\deg(v)$ is the number of edges containing v. That is, $\deg(v) = |\{vx : vx \in E(G)\}|$.

Example 78. Each vertex in the Cycle graph C_n has degree 2. In Example 17, $\deg(6) = 1$ and $\deg(5) = 3$.

Theorem 9.1 (Handshake Lemma)**.** *Let $G(V, E)$ be a graph. We have:*

$$\sum_{v \in V(G)} deg(v) = 2|E(G)|.$$

Proof. The proof is by double counting. The term $\deg(v)$ counts the number of edges incident to v. Each edge has two endpoints v and x, for some other $x \in V(G)$. So the edge vx is double counted in both $\deg(v)$ and $\deg(x)$. Thus,

$$\sum_{v \in V(G)} \deg(v) = 2|E(G)|.$$

□

Remark: The Handshake Lemma is a *necessary condition* for a graph to exist. That is, all graphs satisfy the Handshake Lemma. Consider the following: does there exist a graph on 11 vertices each having degree 5? By the Handshake Lemma, $11 \cdot 5 = 2|E(G)|$. However, 55 is not even, so no such graph exists. Note that the Handshake Lemma is not a *sufficient condition*. That is, there exist degree sequences such as $(3, 3, 1, 1)$ satisfying the Handshake Lemma which are not realizable by any graph. Theorems such as Havel-Hakimi and Erdós-Gallai provide conditions that are both sufficient and necessary for a degree sequence to be realizable by some graph.

Next, the notion of a walk will be introduced. Walks on graphs come up frequently in automata theory. Intuitively, the sequence of transitions in an automaton is analogous to a walk on a graph. Additionally, algorithms like the State Reduction procedure and Brzozowski Algebraic Method that convert finite state automata to regular expressions are based on the idea of a walk on a graph.

Definition 37 (Walk)**.** Let $G(V, E)$ be a graph. A walk of length n is a sequence $(v_i)_{i=0}^{n}$ such that $v_i v_{i+1} \in E(G)$ for all $i \in \{0, \ldots, n-1\}$. If $v_0 = v_n$, the walk is said to be *closed*.

Let us develop some intuition for a walk. We start a given vertex v_0. Then we visit one of v_0's neighbors, which we call v_1. Next, we visit one of v_1's neighbors, which we call v_2. We continue this construction for the desired length of the walk. The key difference between a walk and a path is that a walk can repeat vertices, while all vertices in a path are distinct.

Example 79. Consider a walk on the hypercube Q_3. The sequence of vertices $(000, 100, 110, 111, 101)$ forms a walk, while $(000, 100, 110, 111, 101, 001, 000)$ is a closed walk. The sequence $(000, 111)$ is not a walk because 000 and 111 are not adjacent in Q_3.

We now define the adjacency matrix, which is useful for enumerating walks of a given length.

Definition 38 (Adjacency Matrix)**.** Let $G(V, E)$ be a graph. The adjacency matrix A is an $n \times n$ matrix where:

$$A_{ij} = \begin{cases} 1 & : ij \in E(G) \\ 0 & : ij \notin E(G) \end{cases} \tag{10}$$

Example 80. Consider the adjacency matrix for the graph K_5:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{11}$$

Theorem 9.2. *Let $G(V, E)$ be a graph, and let A be its adjacency matrix. For each $n \in \mathbb{Z}^+$, A^n_{ij} counts the number of walks of length n starting at vertex i and ending at vertex j.*

The proof of Theorem 9.2 uses the Proof by Induction technique. We will formally prove this theorem after we introduce Proof by Induction, later in the course.

For now, we will prove one more theorem before concluding with the graph theory section. In order to prove this theorem, the following lemma (or helper theorem) is needed. As the proof of this lemma is by induction, we omit the proof.

Lemma 9.1. *Let $G(V, E)$ be a graph. Every closed walk of odd length at least 3 in G contains an odd-cycle.*

We now characterize bipartite graphs.

Theorem 9.3. *A graph $G(V, E)$ is bipartite if and only if it contains no cycles of odd length.*

Proof. Suppose first that G is bipartite with parts X and Y. Now consider a walk of length n. As no vertices in a fixed part are adjacent, only walks of even lengths can end back in the same part as the staring vertex. A cycle is a walk where all vertices are distinct, save for v_0 and v_n which are the same. Therefore, no cycle of odd length exists in G.

Conversely, suppose G has no cycles of odd length. We construct a bipartition of $V(G)$. Without loss of generality, suppose G is connected. For if G is not connected, we apply the same construction to each connected component. Fix the vertex v. Let $X = \{u \in V(G) : d(u, v) \text{ is even }\}$, where $d(u, v)$ denotes the distance or length of the shortest uv path. Let $Y = \{u \in V(G) : d(u, v) \text{ is odd }\}$. Clearly, $X \cap Y = \emptyset$. So it suffices to show no vertices within X are adjacent, and no vertices within Y are adjacent. Fix $v \in X$ and suppose to the contrary that two vertices in $y_1, y_2 \in Y$ are adjacent. Then there exists a closed walk of odd length $(v, \ldots, y_1, y_2, \ldots v)$. By Lemma 9.1, G must contain an odd-cycle, a contradiction. By similar argument, no vertices in X can be adjacent. So G is bipartite with bipartition $X \dot\cup Y$. □

9.1.1 Exercises

(Required) Problem 74. There used to be 26 teams in the NFL, with 13 teams in each of two conferences. The NFL had a rule that said that each team's 14 game schedule needed to include 11 games against teams in its own conference and 3 games against teams in the other conference. By modeling this problem as a graph, determine that this rule could not be satisfied.

(Required) Problem 75. Explain by $1 + 2 + \ldots + n = \binom{n+1}{2}$ by counting handshakes in two ways.

(Required) Problem 76. Suppose we have n couples at a party. Each person shakes hands with everyone else, with the exception of their partner. Note that a person does not shake hands with themselves. How many handshakes occurred at this party?

(Required) Problem 77. Recall that the graph Q_n is the hypercube, whose vertex set is the set of strings $\{0,1\}^n$ and two vertices v_1, v_2 are adjacent if and only if they differ in precisely one position. Answer the following:

(a) How many vertices belong to Q_n?

(b) How many edges belong to Q_n?

(c) Suppose that a string $\omega \in \{0,1\}^n$ has exactly i digits that are 1's. Explain why ω's neighbors in Q_n only have either exactly $(i+1)$ or exactly $(i-1)$ digits that are 1's.

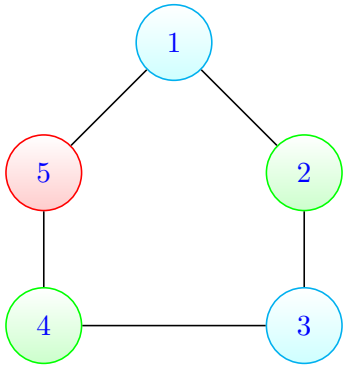
(d) Prove that the graph Q_n is bipartite by giving an explicit bipartition of $\{0,1\}^n$. [**Hint:** Use part (c) to construct your bipartition.]

9.2 Coloring

In this section, we introduce the notion of a graph vertex coloring. Informally, a proper vertex coloring of a graph $G(V,E)$ is an assignment of colors to V such that if two vertices $u, v \in V(G)$ are adjacent, then u and v receive different colors. This is formalized as follows.

Definition 39. A *vertex coloring* of a graph $G(V,E)$ is a function $\phi : V(G) \to [n]$ such that whenever $uv \in E(G)$, $\phi(u) \neq \phi(v)$. The *chromatic number* of G, denoted $\chi(G)$, is the smallest $n \in \mathbb{N}$ such that there exists a coloring $\phi : V(G) \to [n]$.

Example 81. Consider the graph C_5, shown below. We show that $\chi(C_5) = 3$. First, we provide a coloring of C_5 to show that $\chi(C_5) \leq 3$. Consider the following.



Now observe that vertex 5 **cannot** be colored using either green or blue. In particular, regardless of how we color the vertices of C_5 using green and blue, there will always be a vertex v adjacent to both a green and blue vertex. For this reason, a third color for v. So $\chi(C_5) \geq 3$, and we may conclude that $\chi(C_5) = 3$. More generally, if n is odd, $\chi(C_n) = 3$, by the same argument as for C_5.

Example 82. Suppose n is even, and consider the cycle graph C_n. As C_n has an edge, $\chi(C_n) \geq 2$. To show that $\chi(C_n) \leq 2$, we exhibit a two-coloring of C_n. Here, we assign the even indexed vertices of C_n the color `green`, and we assign the odd indexed vertices of C_n `blue`. As the number of vertices in C_n is even, we have that odd-indexed vertices will only be adjacent to even-indexed vertices, and vice-versa. In particular, vertex n is even, so we will not have a case such as in C_5, where two odd vertices are adjacent. For this reason, the two-coloring we provided is valid. Thus, $\chi(C_n) \leq 2$, and we conclude that $\chi(C_n) = 2$.

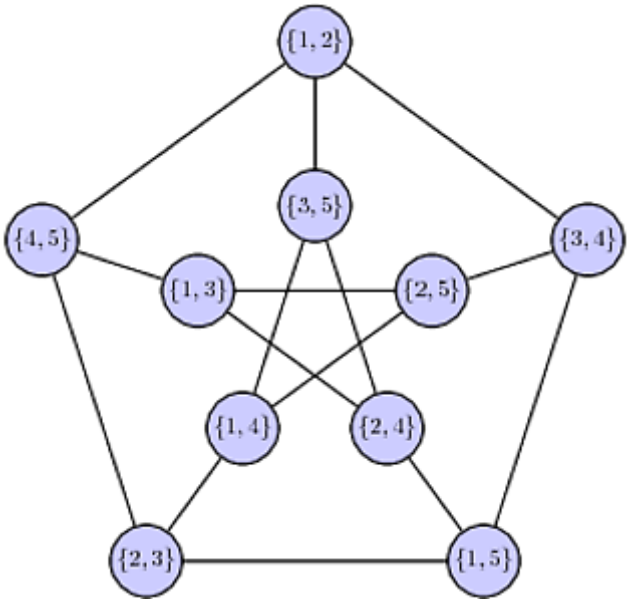
Example 83. Consider the complete graph on n vertices, K_n. We claim that $\chi(K_n) = n$.

- First, we establish the upper bound that $\chi(K_n) \leq n$. Consider the coloring φ, which assigns vertex i the color i. As each vertex receives a distinct color, no two adjacent vertices receive the same color. So φ is a valid coloring. Thus, $\chi(K_n) \leq n$.
- We next establish that $\chi(K_n) > n-1$. To see this, observe that for distinct any $i, j \in [n]$, vertices i and j are adjacent in K_n. So we cannot assign i and j the same color. It follows that we must use at least n colors to properly color K_n. So $\chi(K_n) \geq n$.

We conclude that $\chi(K_n) = n$, as desired.

9.2.1 Exercises

(Required) Problem 78. Let $\binom{[5]}{2}$ denote the set of 2-element subsets of [5]. The *Petersen graph* is a simple graph $\mathcal{P}$ with vertex set $\binom{[5]}{2}$. Two vertices $\{a, b\}$ and $\{c, d\}$ are adjacent in the Petersen graph if and only if $\{a, b\} \cap \{c, d\} = \emptyset$. Determine $\chi(\mathcal{P})$.



(Required) Problem 79. Let $G(V, E)$ be a graph. The *clique number* of G, dentoed $\omega(G)$, is the order of the largest complete subgraph of G. That is, if $\ell = \omega(G)$, then K_ℓ is a subgraph of G; and for any $h > \ell$, K_h is **not** a subgraph of G. Explain why $\omega(G) \leq \chi(G)$.

(Required) Problem 80. Let $G(V, E)$ be a graph.

(a) Prove that if G is bipartite, then G can be colored using at most 2 colors.

(b) Conversely, suppose that G can be colored using at most 2 colors. Prove that G is bipartite.

(Required) Problem 81. Let T be a tree with at least 2 vertices. Determine $\chi(T)$.

(Required) Problem 82. Let $d > 0$. Determine $\chi(Q_d)$.

(Required) Problem 83. For this problem, we consider the Wheel graph W_n, for $n \geq 4$.

(a) Determine $\chi(W_n)$.

(b) Determine all values of n such that W_n is bipartite. Clearly justify your answer.

10 Methods and Sorting

Frequently when programming, we find ourselves using certain segments of logic repeatedly. A method provides of a means of encapsulating this logic. So rather than rewriting and tweaking this logic each time it is used, a single line can instead be used to invoke the method. We have already invoked methods, such as `JOptionPane.showInputDialog()` or the `Random` class' `nextInt()` method. In this section, we introduce the syntax for writing and invoking methods, as well as discussion on designing methods to be modular and reusable.

10.1 Methods

In order to define a method, the following syntax is used.

```
access-modifier < static > return-type name(type1 param1, type2 param2, ..., typeN paramN){
   // code to be executed
}
```

We note the following.

- In the `access-modifier` slot, we may use `public` , `private` , `protected` , or omit an access modifier altogether. So far, we have only worked with `public` methods like the `main()` method. The `public` , `private` , `protected` , and no access modifiers all impact restrictions on the method's visibility to other classes. We defer discussions of access modifiers until object-oriented programming, where there will be sufficient context to discuss them. In this section, we will continue to use the `public` modifier.

- The `static` is optional. If the `static` keyword is present, then that method is associated with the class. For example, the `JOptionPane.showInputDialog()` method is a `static` method. In contrast, the `Random` class' `nextInt()` method is non-static; we invoke it from a variable of type `Random`, rather than through the `Random` class.

  Important: In a given class, `static` methods **cannot** invoke non-`static` methods. For this reason, we will only be using `static` methods in this section.

- Just like mathematical function, Java methods can return values. In this case, `return-type` can be a primitive datatype like `int`, or an Object like an array or String. Unlike mathematical functions, however, Java methods can alternatively **not** return values. In this case, the `return-type` is `void` .

- The `name` is just the name of the method. Note that method names adhere to the same conventions as variable names.

- Just like mathematical functions, Java methods can accept parameters as input. In which case, we specify these parameters by declaring variables. Note that we do **not** assign values to these variables in the method declaration; rather, values are passed when we invoke said method.

Example 84. Consider the following example, which determines whether a given element is in the array. If the element is in the array, the method returns the index. Otherwise, the method returns -1.

```
public static int findElem(int[] arr, int key){
        for(int i = 0; i < arr.length; i++){
                if(arr[i] == key){
                        return i;
                }
        }

        return -1;
}
```

We illustrate how to use the `findElem()` method with the following code sample.

```
public class FindElemDemo{

        public static void main(String[] args){

                int[] arr = { 1, 3, 5, 7, 9, 11 };

                System.out.println("11 is located at index: " + findElem(arr, 11));
                System.out.println("2 is not in the array: " + findElem(arr, 2));

        }

        public static int findElem(int[] arr, int key){
                for(int i = 0; i < arr.length; i++){
                        if(arr[i] == key){
                                return i;
                        }
                }

                return -1;
        }
}
```

The output of this program is as follows.

```
11 is located at index:  5
2 is not in the array:  -1
```

Example 85. We next illustrate the power of a `void` method. Consider the following example which doubles every element in the array.

```
public static void doubleElems(int[] arr){

        for(int i = 0; i < arr.length; i++){
                arr[i] *= 2;
        }
}
```

We illustrate usage of this method with the following code sample.

```
public class DoubleElemsDemo{

        public static void main(String[] args){

                int[] arr = { 1, 3, 5, 7, 9, 11 };

                System.out.print("Original array: ");
                for(int i = 0; i < arr.length; i++){
                        System.out.print(arr[i] + " ");
                }

                doubleElems(arr);
                System.out.print("\nModified array: ");
                for(int i = 0; i < arr.length; i++){
                        System.out.print(arr[i] + " ");
                }

        }

        public static void doubleElems(int[] arr){

                for(int i = 0; i < arr.length; i++){
                        arr[i] *= 2;
                }
        }
}
```

The output of this program is as follows.

```
Original array:  1 3 5 7 9 11
Modified array:  2 6 10 14 18 22
```

We conclude this section with some discussion about designing methods. It is generally advisable to design methods that can be used *beyond* the scope of your current project/program. For this reason, it is advisable to avoid collecting user input in a method. For instance, in the `findElem()` method in Example 84, we accept the array and key as parameters rather than prompting for user input to populate the array or determine the key for which to search. In this manner, the `findElem()` method is reusable. In general, it is good practice to separate the logic of the program from the user interface.

10.1.1 Exercises

(Required) Problem 84. Write a method `pow()` which accepts parameters of type `int` for the `base` and `exponent`. The `pow` method then returns $\texttt{base}^{\texttt{exponent}}$.

(Required) Problem 85. Write a method `findMax()` which accepts a `int[]` parameter and returns the index of the largest element.

(Required) Problem 86. Write a method `swap()` which accepts a `int[]` parameter, as well as two parameters representing indices in the array. The `swap()` method then swaps the elements in the array at those indices. So for example,

$$\texttt{swap}([1,3,5,7],0,3)$$

updates the array, so that it stores in order: `[7, 3, 5, 1]`.

(Required) Problem 87. The *Fibonacci sequence* is defined as follows: $F_0 = 0, F_1 = 1$; and for $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$. Write a method that accepts a a `int` parameter n and returns the nth Fibonacci number.

(Required) Problem 88. The *span* of an element is the number of elements between (and including) the left-most and right-most of said element. A single value has a span of 1. Write a method that takes as parameter a `int[]` and returns the largest span in the array.

We have the following samples.

```
maxSpan([1, 2, 1, 1, 3]) → 4
maxSpan([1, 4, 2, 1, 4, 1, 4]) → 6
maxSpan([1, 4, 2, 1, 4, 4, 4]) → 6
```

10.2 Selection Sort

The selection sort algorithm takes as input an array and sorts it, as follows. On the kth iteration, the algorithm find the kth largest element and swaps it with the kth element from the end of the array.

```
function selectionSort(integer[] arr):
    for k = 0 to arr.len-1:
        kMaxIndex := findMax(arr, 0, arr.len-1-k)
        swap(arr, kMaxIndex, arr.len-1-k)
    end for
```

Note that `findMax(arr, 0, arr.len-1-k)` finds the largest element in the array within the indices $0, 1, \ldots,$ `arr.len-1-k`. The `swap` method swaps the elements in `arr` at positions `kMaxIndex` and `arr.len-1-k`.

Example 86. We use selection sort to sort the following array: `[64, 25, 12, 22, 11]`.

- We first find the largest element and swap it with the last element. Observe that 64 is the largest element, so we swap it with 11. The array now stores, in order: `[11, 25, 12, 22, 64]`.
- We now find the second largest element and swap it with the second to last element. Observe that 25 is the second largest element, so we swap it with 22. The array now stores, in order: `[11, 22, 12, 25, 64]`.
- We now find the third largest element and swap it with the third to last element. Observe that 22 is the third largest element, so we swap it with 12. The array is now stored: `[11, 12, 22, 25, 64]`.

10.2.1 Exercises

(Required) Problem 89. Implement the selection sort algorithm as a method. Use separate methods to swap elements and find the kth largest element in the array. You should do Exercises 85 and 86 first.

10.3 Bubblesort

The Bubblesort is a rather approach for sorting an array. It is often introduced as an academic exercise, with the goal of highlighting that there are better approaches. Intuitively, the Bubblesort has a main loop, which begins by assuming the array is sorted. Inside the main loop, the algorithm iterates through the array, comparing consecutive elements. If a pair of consecutive elements are out of order, the algorithm swaps them and updates its `boolean` flag to indicate that the array was not sorted at the start of the outer loop iteration.

The Bubblesort is formalized with the following pseudo-code.

```
function bubbleSort(integer[] arr):
    swapped := false
    do{
       swapped := false
       for k = 0 to arr.len-2:
         if arr[k] > arr[k+1]:
             swap(arr, k, k+1)
             swapped := true
         end if
       end for
    }while(swapped)
```

Example 87. We use Bubblesort to sort the following array: `[64, 25, 12, 22, 11]`.

- **Iteration 1 of Outer Loop**:
  - We observe that $64 > 25$. So we swap 64 and 25. The array is now: `[25, 64, 12, 22, 11]`.
  - We observe that $64 > 12$. So we swap 64 and 12. The array is now: `[25, 12, 64, 22, 11]`.
  - We observe that $64 > 22$. So we swap 64 and 22. The array is now: `[25, 12, 22, 64, 11]`.
  - We observe that $64 > 11$. So we swap 64 and 11. The array is now: `[25, 12, 22, 11, 64]`.
- **Iteration 2 of Outer Loop**:
  - We observe that $25 > 12$. So we swap 25 and 12. The array is now: `[12, 25, 22, 11, 64]`.
  - We observe that $25 > 22$. So we swap 25 and 22. The array is now: `[12, 22, 25, 11, 64]`.
  - We observe that $25 > 11$. So we swap 25 and 11. The array is now: `[12, 22, 11, 25, 64]`.
  - We observe that $25 < 64$, so no swap is made. The array remains: `[12, 22, 11, 25, 64]`.
- **Iteration 3 of Outer Loop:**
  - We observe that $12 < 22$, so no swap is made. The array remains: `[12, 22, 11, 25, 64]`.
  - We now observe that $22 > 11$. So we swap 22 and 11. The array is now: `[12, 11, 22, 25, 64]`.
  - As $22 < 25$ and $25 < 64$, no additional swaps occur during this iteration of the outer loop.
- **Iteration 4 of Outer Loop:**
  - We observe that $12 > 11$. So we swap 12 and 11. The array is now: `[11, 12, 22, 25, 64]`. The array is now sorted. The inner loop finishes checking the array, and then the algorithm terminates.

10.3.1 Exercises

(Required) Problem 90. Design a Java method to implement Bubblesort.

10.4 Insertion Sort

The Insertion Sort algorithm is rather intuitive, more so than Selection Sort. Intuitively, the Insertion Sort examines the sub-array from indicies $0, \ldots, i$ at iteration i. The algorithm then removes the ith element and compares it to each of elements from indices $i-1, i-2, \ldots, 0$ to determine the precise position to insert the ith element. Once we determine where to insert the ith element, the algorithm shifts the remaining elements down to make room. This is formalized with the following pseudo-code.

Here, the `while` determines where to insert the element at the end of the sub-array being considered. Additionally, the `while` loop handles shifting the elements down one at a time. The `for` loop manages an index i, which tells us which sub-array to consider.

```
function insertionSort(integer[] arr):
    if arr.length < 2:
       return
    endif

    for i = 1 to arr.length - 1:
       temp := arr[i]
       j := i-1

      while j >= 0 AND arr[j] > temp
            arr[j+1] := arr[j]
            j := j-1
      end while

      arr[j+1] := temp
    end for
```

Example 88. We use Insertion Sort to sort the following array: `[64, 25, 12, 22, 11]`.

- **Iteration 1 of Outer Loop:** We are considering the subarray `[64, 25]`. We begin by storing 25 in `temp`. As $25 < 64$, we shift 64 down one slot and insert 25 into position 0. So the array is now `[25, 64, 12, 22, 11]`.
- **Iteration 2 of Outer Loop:** We are considering the subarray `[25, 64, 12]`. We begin by storing 12 in `temp`. As $12 < 64$, we shift 64 down one element. Now as $12 < 25$, we shift 25 down one element and insert 12 into position 0. So the array is now: `[12, 25, 64, 22, 11]`.
- **Iteration 3 of Outer Loop:** We are considering the subarray `[12, 25, 64, 22]`. We begin by storing 22 in `temp`. As $22 < 64$, we shift 64 down one element. Now as $22 < 25$, we shift 25 down one element. As $22 > 12$, we leave 12 where it is. So we insert 22 at position 1. So the array is now `[12, 22, 25, 64]`.
- **Iteration 4 of Outer Loop:** We are considering the entire array: `[12, 22, 25, 64, 11]`. We begin by storing 11 in temp. As $11 < 64$, we shift 64 down one position. Now as $11 < 25$, we shift 25 down by one position. Similarly, we shift 22 and 12 each down by one position. So we insert 11 at position 0. So the sorted array is: `[11, 12, 22, 25, 64]`.

10.4.1 Exercises

(Required) Problem 91. Design a Java method to implement Insertion Sort.

11 Asymptotics

In computer science, efficient solutions to problems are highly valued. Algorithms that use little in the way of resources, such as time or space, are highly desirable. In order to compare the efficiency of an algorithm, a measure of its complexity is needed. This motivates the study of Big-O notation, as well as the field of Computational Complexity Theory. Intuitively, the Big-O notation formalizes the notion of asymptotic upper bound, or upper bound except for a finite number of initial values. In this section, we introduce the Big-O notation as well as key preliminaries from high school algebra.

11.1 Algebra Preliminaries

11.1.1 Exponential Functions

Definition 40. Let $b > 0$, with $b \neq 1$. An *exponential function* is of the form $f(x) = b^x$.

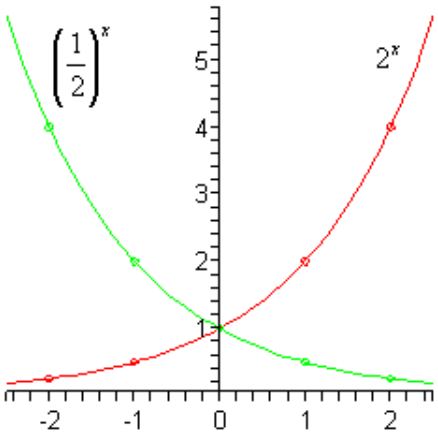
Example 89. Common examples of exponential functions include $2^x, 3^x, e^x$, and $\left(\frac{1}{2}\right)^x$.

Remark: Recall that e is *Euler's constant*, which is approximately 2.718. Formally, e satisfies the following:

$$\left(1+\frac{1}{n}\right)^n \to e \text{ as } n \to \infty.$$

In other words, as n gets really large, $\left(1+\frac{1}{n}\right)^n$ gets really close to e.

We note that if $b \in (0,1)$, then b^x is a decreasing function and $b^x \to 0$ as $x \to \infty$. If instead $b > 1$, then b^x is an increasing function, and so $b^x \to \infty$ as $x \to \infty$. We provide the following graph for intuition.



In particular, observe that $b^0 = 1$. Furthermore, an exponential function **never** evaluates to 0; that is, for every input value x, $b^x \neq 0$.

We next introduce the rules of exponents, which are useful in manipulating exponential expressions. They are as follows:

- $b^x b^y = b^{x+y}$.
- $\frac{b^x}{b^y} = b^{x-y}$.
- $(b^x)^y = b^{xy}$.

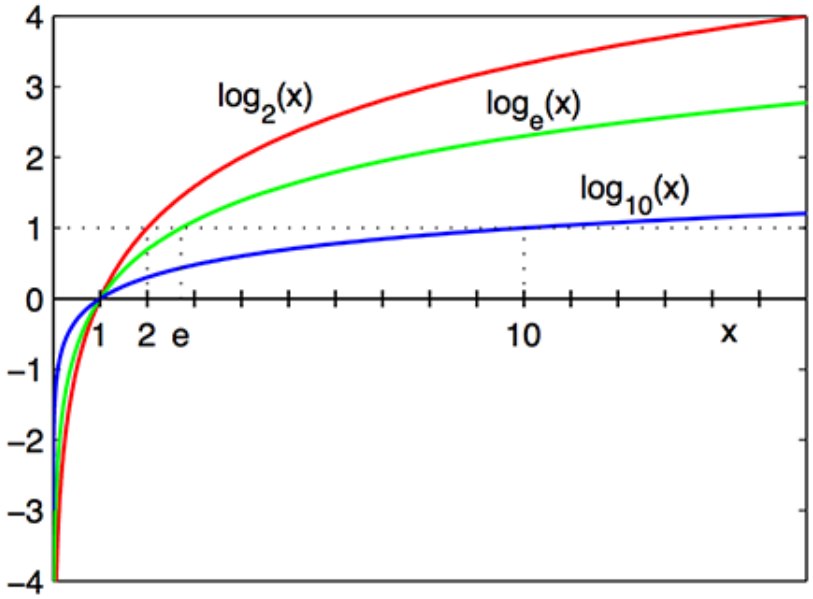
Example 90. We have that $2^3 \cdot 2^4 = 2^7$. Similarly, $\frac{2^3}{2^4} = 2^{-1}$. Note that $2^{-1} = \frac{1}{2}$. Now $(2^3)^4 = 2^{12}$.

11.1.2 Logarithms

Logarithms are inverse functions of exponential functions. That is, for $b > 0$, $\log_b(x)$ is the function that satisfies the following: $\log_b(b^x) = x$ and $b^{\log_b(x)} = x$. Note that a logarithm is keyed to the base. So $\log_2(x)$ and $\log_3(x)$ are different functions, just like 2^x and 3^x are different functions.

Remark: We note that $\ln(x) = \log_e(x)$, which is referred to as the *natural logarithm*. It is also worth noting that $\log(x)$, without the base explicitly specified, refers to the different logarithms depending on the discipline. In computer science, $\log(x)$ refers to $\log_2(x)$. In pure math, $\log(x)$ refers to $\ln(x)$. High school math classes adopt the convention that $\log(x)$ refers to $\log_{10}(x)$. When in doubt, **ask** about the convention. **In this class, we adopt the convention that:** $\log(x) = \log_2(x)$.

We next include a graph of some logarithms for perspective.



We note that for any $b > 0$, $\log_b(1) = 0$. Furthermore, $\log_b(x)$ never touches the y-axis. Logarithms are also increasing functions.

We next introduce the rules of logarithms, including the following.

- $\log_b(xy) = \log_b(x) + \log_b(y)$.
- $\log_b\left(\dfrac{x}{y}\right) = \log_b(x) - \log_b(y)$.
- $\log_b(x^n) = n\log_b(x)$.

Example 91. We decompose $\ln\left(x\sqrt{y^2+z^2}\right)$ as much as possible using the rules of logarithms. Observe that:

$$\begin{aligned}\ln\left(x\sqrt{y^2+z^2}\right) &= \ln(x) + \ln(\sqrt{y^2+z^2}) \\ &= \ln(x) + \frac{1}{2}\ln(y^2+z^2).\end{aligned}$$

Example 92. We write $\dfrac{1}{3}\ln(a) - \ln(b)$ as a single logarithm. Observe that:

$$\begin{aligned}\frac{1}{3}\ln(a) - \ln(b) &= \ln(a^{1/3}) - \ln(b) \\ &= \ln\left(\frac{a^{1/3}}{b}\right).\end{aligned}$$

11.1.3 Exercises

(Required) Problem 92. Write the following expression as a power of 3: $(3^3 \cdot 9^5)^{-2}$.

(Required) Problem 93. Write the following as a power of 5: $(5^2)^4 \cdot 25^2 \cdot (125)^{-5}$.

(Required) Problem 94. Decompose $\ln(3x^4y^{-7})$ as much as possible using the rules of logarithms.

(Required) Problem 95. Decompose $\ln\left(\dfrac{x-4}{y^2\sqrt[5]{z}}\right)$ as much as possible using the rules of logarithms.

(Required) Problem 96. Using the rules of logarithms, write the following expression as a single logarithm.

$$3\ln(t+5) - 4\ln(t) - 2\ln(s-1).$$

11.2 Big-O

Big-O notation formalizes the notion of an asymptotic upper bound. Informally, we say that a function $f(n) \in \mathcal{O}(g(n))$ (pronounced $f(n)$ is Big-O $g(n)$) if $g(n)$ grows *at least as fast* as $f(n)$. That is, Big-O formalizes the notion of a weak asymptotic upper bound. This is formalized as follows.

Definition 41. Suppose $f(x), g(x)$ are real-valued functions. We say that $f(x) \in \mathcal{O}(g(x))$ if there exist constants $C, k \in \mathbb{Z}^+$ such that $|f(x)| \leq C \cdot |g(x)|$ for all $x \geq k$.

We provide some examples.

Example 93. Take $f(x) = x$ and $g(x) = x\log(x)$. We take $C = 1$ and $k = 2$, as in the definition of Big-O. Observe that at $x = 2$, we have:

$$f(2) = 2$$
$$g(2) = 2\log(2) = 2.$$

Now as $\log(x)$ is an increasing function, it follows that $|f(x)| \leq |g(x)|$ for all $x \geq 2$. So $f(x) \in \mathcal{O}(g(x))$.

Example 94. Take $f(x) = x\log(x)$ and $g(x) = x^2$. We take $C = k = 1$ as witnesses that $f(x) \in \mathcal{O}(g(x))$. Showing this more rigorously requires the use of Calculus techniques.

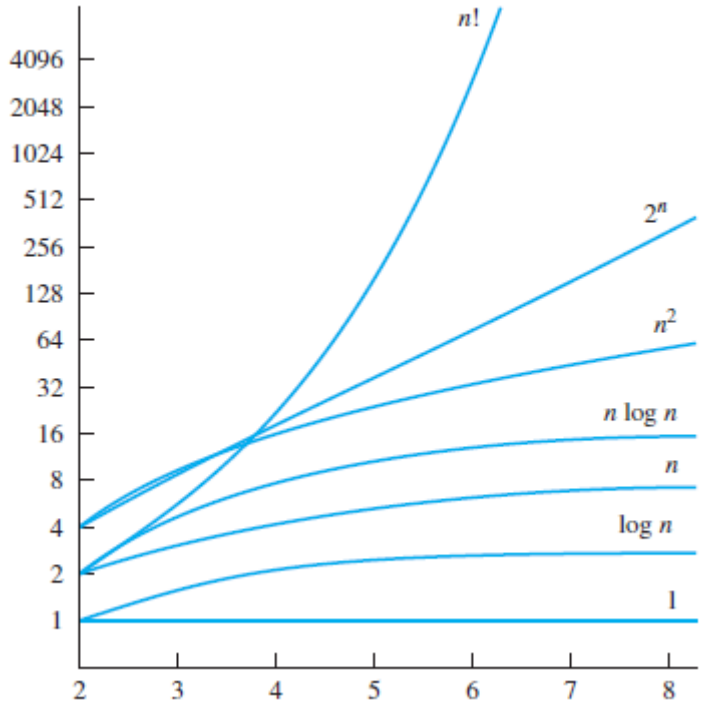
Example 95. If $n > m$, then $x^m \in \mathcal{O}(x^n)$. We illustrate this with x^2 and x^3. Taking $C = k = 1$, we have that $x^2 \in \mathcal{O}(x^3)$.

We also note that $x^3 \notin \mathcal{O}(x^2)$. We argue this by contradiction. Suppose to the contrary that $x^3 \in \mathcal{O}(x^2)$. So there exist constants $C, k \in \mathbb{Z}^+$ such that for all $x \geq k$, we have:

$$x^3 \leq Cx^2.$$

So $x \leq C$ for all $x \geq k$. Taking $x > \max\{C, k\}$, we achieve a contradiction. Thus, we conclude that $x^3 \notin \mathcal{O}(x^2)$.

There are several key functions, which appear frequently in algorithm analysis, including: $\log(x)$, $\sqrt[n]{x}, x^m, b^x$ (where $b > 1$), $n!$, and n^n. We have listed the functions in increasing order according to the Big-O order. So for instance, $\log(x) \in \mathcal{O}(\sqrt[n]{(x)})$, $\sqrt[n]{x} \in \mathcal{O}(x^m)$, and $n! \in \mathcal{O}(n^n)$. The following graph provides some visual intuition as to this ordering.



Using the definition of Big-O, we establish some useful results.

Proposition 11.1. *Suppose $f(x)$ is a polynomial of degree n; that is, $f(x) = a_0x^n + a_1x^{n-1} + \ldots + a_n$. Then $f(x) \in \mathcal{O}(x^n)$.*

Proof. We note that if $x > 1$, that:

$$\begin{aligned} |f(x)| &= |a_0x^n + a_1x^{n-1} + \ldots + a_n| \\ &\leq |a_0|x^n + |a_1|x^{n-1} + \ldots + |a_n| \\ &\leq |a_0|x^n + |a_1|x^n + \ldots + |a_n|x^n \\ &= x^n(|a_0| + |a_1| + \ldots + |a_n|). \end{aligned}$$

The second line follows from the triangle inequality. So we take $k = 2$ and $C = 1$. Thus, $f(x) \in \mathcal{O}(x^n)$. □

Proposition 11.2. *Let $f(n) = 1 + 2 + 3 + \ldots + n$. Then $f(n) \in \mathcal{O}(n^2)$.*

Proof. We take $C = k = 1$. So for $n \geq k$, we have that:

$$\begin{aligned} |f(n)| &= 1 + 2 + \ldots + n \\ &\leq n + n + \ldots + n \\ &\leq n(n) \\ &= n^2. \end{aligned}$$

So $f(n) \in \mathcal{O}(n^2)$. □

11.2.1 Exercises

(Required) Problem 97. Show that $\log(x^2 + 1) \in \mathcal{O}(\log(x))$.

(Required) Problem 98. Which of the following functions are $\mathcal{O}(x^2)$? Justify your answer.

(a) $17x + 11$

(b) $x^2 + 1000$

(c) 2^x

(Required) Problem 99. For each of the following functions $f(x)$, find the least integer n such that $f(x) \in \mathcal{O}(x^n)$.

(a) $f(x) = 2x^3 + x^2 \log(x)$

(b) $f(x) = 3x^3 + (\log(x))^4$

(c) $f(x) = 2x^3 + x^3 \log(x)$

(Required) Problem 100. Show that $\log(n!) \in \mathcal{O}(n \log(n))$.

(Required) Problem 101. Suppose that $f_1(x), f_2(x) \in \mathcal{O}(g(x))$. Show that $(f_1 + f_2)(x) \in \mathcal{O}(g(x))$.

(Required) Problem 102. Suppose that $f(x) \in \mathcal{O}(g(x))$ and $g(x) \in \mathcal{O}(h(x))$. Show that $f(x) \in \mathcal{O}(h(x))$.

12 ArrayLists

Recall that an array is a construct that provides a means to effectively manage such variables, using an index for each block. One limitation of an array is that its size is fixed once it is initialized. It is **not** possible to resize an array at a later point. The ArrayList class serves as a resizable variant to the array. In this section, we introduce the ArrayList class and illustrate its usage.

Recall that when creating an array, we must specify the `type` and the `name`:

```
type name = new type[length];
```

When constructing an ArrayList, we similarly specify the type as follows:

```
ArrayList<Type> list = new ArrayList<>();
```

So for example, to construct an `ArrayList` storing Strings, we use the following syntax.

```
ArrayList<String> list = new ArrayList<>();
```

Note that while arrays can directly store primitives, ArrayLists can **only store Objects**. Java provides a workaround with the primitive wrapper classes Byte, Short, Integer, Long, Float, Double, Boolean, and Character. That is, if we wish to store an `ArrayList` containing `int` values, we use an `ArrayList<Integer>`. Values of type `int` can be directly added to the `ArrayList`.

We next discuss key ArrayList methods to add, update, remove, and search for elements.

- **Adding Elements:** There are two `add()` methods in the `ArrayList<Type>` class.
  - The `add(Type elem)` method adds the desired element to the end of the list.
  - In contrast, the `add(int index, Type elem)` method adds the element at the specified index. If necessary, inserting `elem` shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

  Example 96. We illustrate the `add()` methods with the following code sample.

```
import java.util.ArrayList;

public class AddDemo{

        public static void main(String[] args){
                ArrayList<Integer> list = new ArrayList<Integer>();

                for(int i = 0; i < 5; i++){
                        list.add(i);
                        System.out.println("List: " + list);
                }

                list.add(3, 13);
                System.out.println("List: " + list);

        }
}
```

Note that once we add 13 at index 3, the elements 3 and 4 are each shifted down by one. The output of the following program is as follows.

```
List:  [0]
List:  [0, 1]
List:  [0, 1, 2]
List:  [0, 1, 2, 3]
List:  [0, 1, 2, 3, 4]
List:  [0, 1, 2, 13, 3, 4]
```

- **Replacing Elements:** If we want to replace at a given element, we use the `set(int index, Type elem)` method.

Example 97. We illustrate the `set()` method as follows.

```
import java.util.ArrayList;

public class SetDemo{

        public static void main(String[] args){
                ArrayList<Integer> list = new ArrayList<Integer>();

                for(int i = 0; i < 5; i++){
                        list.add(i);
                        System.out.println("List: " + list);
                }

                list.set(3, 13);
                System.out.println("List: " + list);

        }
}
```

The output of this program is as follows. Here, the `set()` method replaces the 3 with 13 rather than shifting 3 and 4 down.

```
List:  [0]
List:  [0, 1]
List:  [0, 1, 2]
List:  [0, 1, 2, 3]
List:  [0, 1, 2, 3, 4]
List:  [0, 1, 2, 13, 4]
```

- **Removing Elements:** We can remove an element by specifying the index or the element.
  - The `remove(int index)` method not only removes the element at the specified index, but returns it as well.
  - The `remove(Type elem)` method removes the first occurrence of `elem` in the ArrayList, if `elem` is present. If `elem` is removed, the `remove()` method returns `true`. Otherwise, the `remove()` method returns `false`.

Example 98. We illustrate the `remove()` methods with the following code sample.

```
import java.util.ArrayList;

public class RemoveDemo{

        public static void main(String[] args){
                ArrayList<String> list = new ArrayList<>();
                list.add("CTY");
                list.add("FCPS");
                list.add("Java");

                System.out.println("List: " + list);
                System.out.println("Removing Elem 0: " + list.remove(0));
                System.out.println("List: " + list);

                System.out.println("Remove C++: " + list.remove("C++"));
                System.out.println("List: " + list);

        }
}
```

The output of this code sample is the following.

```
List:  [CTY, FCPS, Java]
Removing Elem 0:  CTY
List:  [FCPS, Java]
Remove C++:  false
List:  [FCPS, Java]
```

- **Accessing Elements and Size:** We use the `get(int index)` to access the element at the specified index. Like arrays, the `ArrayList` class indexes its elemens $0, 1, \ldots,$ length $- 1$. The `size()` method is used to determine the number of elements in the array.

Example 99. We illustrate the `get()` and `size()` methods with the following code sample.

```
import java.util.ArrayList;

public class GetDemo{

        public static void main(String[] args){
                ArrayList<Integer> list = new ArrayList<Integer>();

                for(int i = 0; i < 5; i++){
                        list.add(2*i);
                        System.out.println("List: " + list);
                }

                System.out.println("Elem at Index 4: " + list.get(4));
                System.out.println("List Size: " + list.size());

        }
}
```

The output of this code sample is the following.

```
List:  [0]
List:  [0, 2]
List:  [0, 2, 4]
List:  [0, 2, 4, 6]
List:  [0, 2, 4, 6, 8]
Elem at Index 4:  8
List Size:  5
```

12.1 Exercises

(Required) Problem 103. Generate 100 distinct random numbers. That is, your program should ensure that all the random numbers stored are unique.

(Required) Problem 104. Write a method named `backwardChars` that takes as input a String and returns an `ArrayList<Character>` with the characters of the String in reverse order. The method header is as follows:

```
public static ArrayList<Character> backwardChars(String str)
```

(Required) Problem 105. Return a version of the given array where each zero value in the array is replaced by the largest odd value to the right of the zero in the array. If there is no odd value to the right of the zero, leave the zero as a zero. The method header is as follows:

```
public static ArrayList<Integer> zeroMax(ArrayList<Integer> nums)
```

`zeroMax({0, 5, 0, 3})` $\rightarrow \{5, 5, 3, 3\}$
`zeroMax({0, 4, 0, 3})` $\rightarrow \{3, 4, 3, 3\}$
`zeroMax({0, 1, 0})` $\rightarrow \{1, 1, 0\}$

(Required) Problem 106. Write a method to implement the binary search algorithm, taking an `ArrayList` as an input. If the element is found, the method should return the index of the element. Otherwise, the method returns -1.

```
public static int binSearch(ArrayList<Integer> arr, int key)
```

13 Object-Oriented Programming

Object-Oriented Programming is a paradigm designed to model real-world objects. Informally, an Object is a noun, which has state and behavior. The state of an Object is described using attributes, which we refer to as instance variables. Object behavior is described via methods. That is, an Object only does something when a method is invoked.

There are a number of benefits to Object-Oriented Programming, including modularity, reusability, and encapsulation. Modularity refers to the fact that a Objects are individual *modules*, which can be designed (via a `class`) and maintained independently of other aspects of the program. As a result, Object code can easily be reused by other developers. Similarly, we can easily reuse code that others have written (such as the `java.util.Random` or `javax.swing.JOptionPane` classes). In particular, it is not necessary to be aware of the code underneath the hood of an Object in order to use it. That is, we do not have to be aware of how a car works in order to drive. This is referred to as *encapsulation*.

This section has benefitted from [8].

13.1 Class Design

In order to design an Object, a class is used. An Object is an instance of the class. Intuitively, we may think of a class as a recipe and an Object as the food made from said recipe. Alternatively, a class is analogous to a blueprint, and an Object is like the house that was constructed from the blueprint.

After determining the attributes and behavior of an Object, designing a class has three key components: specifying the attributes, defining the constructors, and defining the methods.

We work through an example to design a video game player.

- **Attributes:** There are a number of attributes for a player, including the name, level, health, and experience. We represent these attributes as *instance variables*, which are non-static variables that we declare in the class that are not in any method. As the variables are not static, they are associated with the Object and not the class. For instance, each Player has their own name.

  We next note that all variables are declared as `private` . This means they are not visible outside of the Player class. Using the `private` keyword in this manner goes towards encapsulation and information hiding. Other components do not need to know the inner-workings of a given class. Such information is only shared on a need-to-know basis, using methods to control access. In many cases, the instance variables play a key role in the Object's funcitonality. If they are modified inappropriately,

```
public class Player{

    private String name;
    private int level;
    private int health;
    private int experience;
}
```

- **Constructors:** Constructors are methods that Java uses to create the Object. Within a constructor, we initialize the attributes of the Object. Like methods, constructors can be defined to accept parameters. We may use such parameters to initialize the attributes of the class.

  Consider the following code sample. The first `Player()` constructor is called a *no-args constructor*, since it does not accept any parameters. A no-args constructor initializes the Player object's attributes to default values.

  The second constructor accepts parameters for `name`, `health`, `level`, and `experience`. It then assigns the parameter values to the instance variables. Consider the following line.

```
this.name = name;
```

  Note that `this.name` refers to the instance variable `private String name`, while `name` to the right of the assignment operator refers to the parameter `String name` defined in the constructor here:

```
public Player(String name, int health, int level, int experience)
```

```
public class Player{

        private String name;
        private int health;
        private int level;
        private int experience;

        public Player(){
                this.name = "Mace Windu";
                this.health = 0;
                this.level = 0;
                this.experience = 0;
        }

        public Player(String name, int health, int level, int experience){
                this.name = name;
                this.health = health;
                this.level = level;
                this.experience = experience;
        }
}
```

- **Methods:** We finally define the behaviors or methods for the Player class. The getter methods below are accessor methods; they return the current value in the corresponding instance variable. We also have a mutator or setter method, namely the `setName()` method, which updates the value in the `this.name` instance variable based on the provided parameter.

  We next discuss the `decreaseHealth()` method, which validates that the damage received is positive before decreasing the player's health. Note that as `health` is `private` , this validation about damage is enforced.

  Similarly, the `addExperience()` method validates that `exp` is positive before adding it to `this.experience`. Additionally, the `addExperience()` method levels up the Player every thousand experience points.

```
public class Player{

        private String name;
        private int health;
        private int level;
        private int experience;

        public Player(){
                this.name = "Mace Windu";
                this.health = 0;
                this.level = 0;
                this.experience = 0;
        }

        public Player(String name, int health, int level, int experience){
                this.name = name;
                this.health = health;
                this.level = level;
                this.experience = experience;
        }

        public String getName(){
```

```
                return name;
        }

        public void setName(String name){
                this.name = name;
        }

        public int getHealth(){
                return health;
        }

        public void decreaseHealth(int damage){
                if(damage <= 0){
                        return;
                }

                this.health -= damage;
        }


        public int getLevel(){
                return level;
        }

        public int getExperience(){
                return experience;
        }

        public void addExperience(int exp){
                if(exp <= 0){
                        return;
                }

                experience += exp;

                if(experience % 1000 == 0){
                        level++;
                }
        }
}
```

We illustrate how to use the Player class with the following code sample.

```
public class PlayerDemo{

        public static void main(String[] args){
                Player player = new Player("Yoda", 100, 1, 0);

                for(int i = 0; i < 5; i++){
                        player.addExperience(200);
                        System.out.println("Player Level: " + player.getLevel());
                }

                player.decreaseHealth(30);
                System.out.println("Player Health: " + player.getHealth());
        }
}
```

The output of this program is as follows.

```
Player Level:  1
Player Level:  1
Player Level:  1
Player Level:  1
Player Level:  2
```

13.2 Exercises

(Required) Problem 107. Design a Rectangle class, which provides getter and setter methods for the length and width. Additionally, the Rectangle class should compute the area and perimeter.

(Required) Problem 108. Design a Product class, which has attributes name, price, and quantity. Then design an inventory management program, where the user can add an Item, modify an existing Item, display the inventory, or exit.

14 Automata Theory

Theoretical computer science is divided into three key areas: automata theory, computability theory, and complexity theory. The goal is to ascertain the power and limits of computation. In order to study these aspects, it is necessary to define precisely what constitutes a model of computation as well as what constitutes a computational problem. This is the purpose of automata theory. The computational models are automata, while the computational problems are formulated as formal languages. A common theme in theoretical computer science is the relation between computational models and the problems they solve. The Church-Turing thesis conjectures that no model of computation that is physically realizable is more powerful than the Turing Machine. In other words, the Church-Turing thesis conjectures that any problem that can be solved via computational means, can be solved by a Turing Machine. To this day, the Church-Turing thesis remains an open conjecture. For this reason, the notion of an algorithm is equated with a Turing Machine. In this section, the simplest class of automaton will be introduced- the finite state automaton, as well as the interplay with regular languages which are the computational problems finite state automata solve.

14.1 Regular Languages

In order to talk about regular languages, it is necessary to formally define a language.

Definition 42 (Alphabet)**.** An alphabet Σ is a finite set of symbols.

Example 100. Common alphabets include the binary alphabet $\{0, 1\}$, the English alphabet $\{A, B, \ldots, Z, a, b, \ldots, z\}$, and a standard deck of playing cards.

Definition 43 (Kleene Closure)**.** Let Σ be an alphabet. The Kleene closure of Σ, denoted Σ^*, is the set of all finite strings whose characters all belong to Σ. Formally, $\Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n$. The set $\Sigma^0 = \{\epsilon\}$, where ϵ is the empty string.

Definition 44 (Language)**.** Let Σ be an alphabet. A language $L \subset \Sigma^*$.

We now delve into regular languages, starting with a definition. This definition for regular languages is rather difficult to work with and offers little intuition or insights into computation. Kleene's Theorem (which will be discussed later) provides an alternative definition for regular languages which is much more intuitive and useful for studying computation. However, the definition of a regular language provides some nice syntax for regular expressions, which are useful in pattern matching.

Definition 45 (Regular Language)**.** Let Σ be an alphabet. The following are precisely the regular languages over Σ:

- The empty language $\emptyset$ is regular.
- For each $a \in \Sigma$, $\{a\}$ is regular.
- Let L_1, L_2 be regular languages over Σ. Then $L_1 \cup L_2, L_1 \cdot L_2$, and L_1^* are all regular.

Remark: The operation $\cdot$ is string concatenation. Formally, $L_1 \cdot L_2 = \{xy : x \in L_1, y \in L_2\}$.

A regular expression is a concise algebraic description of a corresponding regular language. The algebraic formulation also provides a powerful set of tools which will be leveraged throughout the course to prove languages are regular, derive properties of regular languages, and show certain collections of regular languages are decidable. The syntax for regular expressions will now be introduced.

Definition 46 (Regular Expression)**.** Let Σ be an alphabet. A regular expression is defined as follows:

- $\emptyset$ is a regular expression, and $L(\emptyset) = \emptyset$.
- ϵ is a regular expression, with $L(\epsilon) = \{\epsilon\}$.
- For each $a \in \Sigma$, $L(a) = \{a\}$.
- Let R_1, R_2 be regular expressions. Then:
  - $R_1 + R_2$ is a regular expression, with $L(R_1 + R_2) = L(R_1) \cup L(R_2)$.
  - $R_1 R_2$ is a regular expression, with $L(R_1 R_2) = L(R_1) \cdot L(R_2)$.
  - R_1^* is a regular expression, with $L(R_1^*) = (L(R_1))^*$.

Like the definition of regular languages, the definition of regular expressions is bulky and difficult to use. We provide a couple examples of regular expressions to develop some intuition.

Example 101. Let L_1 be the set of strings over $\Sigma = \{0, 1\}$ beginning with 01. We construct the regular expression $01\Sigma^* = 01(0 + 1)^*$.

Example 102. Let L_2 be the set of strings over $\Sigma = \{0, 1\}$ beginning with 0 and alternating between 0 and 1. We have two cases: a string ends with 0 or it ends with 1. Suppose the string ends with 0. Then we have the regular expression $0(10)^*$. If the string ends with 1, we have the regular expression $0(10)^*1$. These two cases are disjoint, so we add them: $0(10)^* + 0(10)^*1$.

Remark: Observe in Example 102 that we are applying the Rule of Sum. Rather than counting desired objects, we are listing them explicitly. Regular Expressions behave quite similarly to the ring of integers, with several important differences, which we will discuss shortly.

14.1.1 Exercises

(Required) Problem 109. Let L_1 be the language over $\Sigma = \{a, b\}$ where the number of a's is divisible by 3.

(Required) Problem 110. Let L_2 be the language over $\Sigma = \{0, 1\}$ where each string in L_2 contains both 00 and 11 as substrings.

(Required) Problem 111. Provide a regular expression and a finite state machine for the following languages:

(a) Let L_1 be the language over $\{0, 1\}$ containing strings with both 01 and 10 as substrings.

(b) Let L_2 be the language over $\{0, 1\}$ in which every 0 is either immediately preceeded or followed by 1.

(c) Let L_3 be the set of strings over $\{0, 1\}$ which do not begin with 000.

(d) Let L_4 be the set of strings over $\{0, 1\}$, where each string is the binary representation of a natural number divisible by 8.

14.2 Finite State Automata

The finite state automaton (or FSM) is the first model of computation we shall examine. We then introduce the notion of language acceptance, culminating with Kleene's Theorem which relates regular languages to finite state automata.

We begin with the definition of a deterministic finite state automaton. There are also non-deterministic finite state automata, both with and without ϵ transitions. These two models will be introduced later. They are also equivalent to the standard deterministic finite state automaton.

Definition 47 (Finite State Automaton (Deterministic))**.** A *Deterministic Finite State Automaton* or *DFA* is a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the finite set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to Q$ is the state transition function, q_0 is the initial state, and $F \subset Q$ is the set of accepting states.

A We now select a string $\omega \in \Sigma^*$ as the input string for the FSM. From the initial state, we transfer to another state in Q based on the first character in ω. The second character in ω is examined and another state transition is executed based on this second character and the current state. We repeat this for each character in the string. The state transitions are dictated by the state transition function δ associated with the machine. A string ω is said to be accepted by the finite state automaton if, when started on $q_0 with \omega$ as the input, the finite state automaton terminates on a state in F. The language of a finite state automaton M is defined as follows:

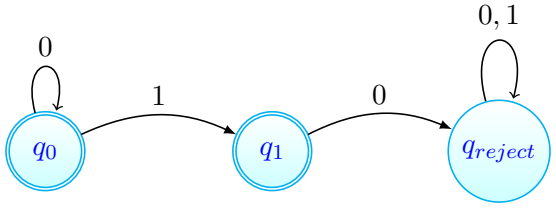
Definition 48 (Language of FSM)**.** Let M be a FSM. The language of M is the set of strings it accepts.

Let us consider an example of a DFA to develop some intuition.

Example 103. Let $\Sigma = \{0, 1\}$, and let $Q = \{q_0, q_1, q_{reject}\}$. We define $\delta(q_0, 0) = q_0, \delta(q_0, 1) = q_1, \delta(q_1, 1) = 1$, and $\delta(q_1, 0) = q_{reject}$. Finally, we have $\delta(q_{reject}, 0) = \delta(q_{reject}, 1) = q_{reject}$. We have the accepting set of states $F = \{q_0, q_1\}$. Observe that this finite state automata accepts the language 0^*1^*. We start at state q_0. For each 0 read in, we simply stay at state q_0. Then when we finish reading in 0's, we transition to q_1 if any 1's follow the sequence of 0's. At q_1, we simply eat away at the 1's. If a 0 is read after any 1's have been recognized, then we transition to a *trap* state or a *reject* state.

We represent FSMs pictorally using labeled directed graphs $G(V, E, L)$ where each vertex of V corresponds to the set of states Q. There is a directed edge $(q_i, q_j) \in E(G)$ if and only if there exists a transition $\delta(q_i, a) = q_j$ for some $a \in \Sigma \cup \{\epsilon\}$. The label function $L : E(G) \to 2^{\Sigma \cup \{\epsilon\}}$ maps $(q_i, q_j) \mapsto \{a \in \Sigma \cup \{\epsilon\} : \delta(q_i, a) = q_j\}$.

The FSM diagram for Example 39 is shown below:



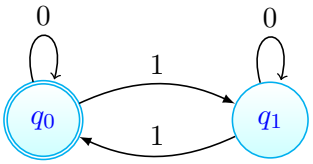
Recall from the introduction that we are moving towards the notion of an algorithm. This is actually a good starting place. Observe that a finite state automaton has no memory beyond the current state. It also has no capabilities to write to memory. Conditional statements and loops can all be reduced to state transitions, so this is a good place to start.

Consider the following algorithm to recognize binary strings with an even number of bits.

```
function evenParity(string ω):
    parity := 0
    for i := 0 to len(ω):
        parity := (party + ω_i) (mod 2)
    return parity == 0
```

So this algorithm accepts a binary string as input and examines each character. If it is a 1, then parity moves from $0 \to 1$ if it is 0, or from $1 \to 0$ if its current value is 1. So if there are an even number of 1's in ω, then parity will be 0. Otherwise, parity will be 1.

The following diagram models the algorithm as a finite state automaton. Here, we have $Q = \{q_0, q_1\}$ as our set of states with q_0 as the initial state. Observe in the algorithm above that parity only changes value when a 1 is processed. This is expressed in the finite state automata below, with the directed edges indicating that $\delta(q_i, \epsilon) = \delta(q_i, 0) = q_i, \delta(q_0, 1) = q_1$, and $\delta(q_1, 1) = q_0$. A string is accepted if and only if it has parity $= 0$, so $F = \{q_0\}$.



From this finite state automaton and algorithm above, it is relatively easy to guess that the corresponding regular expression is $(0^*10^*1)^*$. Consider 0^*10^*1. Recall that 0^* can have zero or more 0 characters. As we are starting on q_0 and $\delta(q_0, 0) = q_0, 0^*$ will leave the finite state automaton on state q_0. So then the 1 transitions the finite state automaton to state q_1. By similar analysis, the second 0^* term keeps the finite state automaton at state q_1, with the second 1 term sending the finite state automaton back to state q_0. The Kleene closure of 0^*10^*1 captures all such strings that will cause the finite state automaton to halt at the accepting state q_0.

In this case, the method of judicious guessing worked nicely. For more complicated finite state automata, there are algorithms to produce the corresponding regular expressions. This is a standard topic in a Theory of Computation course.

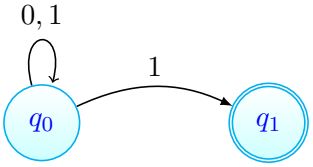
We briefly introduce NFAs and ϵ-NFAs prior to discussing Kleene's Theorem.

Definition 49 (Non-Deterministic Finite State Automata)**.** A *Non-Deterministic Finite State Automaton* or *NFA* is a five-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, q_0 is the initial state, and $F \subset Q$ is the set of accept states.

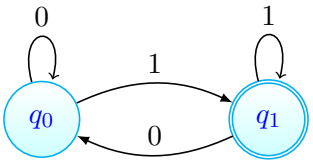
Remark: An ϵ-NFA is an NFA where the transition function is instead defined as $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is the transition function. An NFA is said to accept a string ω if there exists a sequence of transitions terminating in accepting state. There may be multiple accepting sequences of transitions, as well as non-accepting transitions for NFAs. Observe as well that the other difference between the non-deterministic and deterministic finite state automata is that the non-deterministic variant's transition function returns a subset of Q, while the deterministic variant's transition function returns a single state. So we trivially have that a DFA is an NFA (ignoring the non-determinism). Similarly, an NFA is also an ϵ-NFA.

It is often easier to design efficient NFAs than DFAs. Consider an example below.

Example 104. Let L be the language given by $(0+1)^*1$. An NFA is given below. Observe that we only care about the last character being a 1. As $\delta(q_0, 1) = \{q_0, q_1\}$, the FSM is non-deterministic.



An equivalent DFA requires more thought in the design. We change state immediately upon reading a 1, then additional effort is required to ensure 1 is the last character of any valid string. At q_1, we transition to q_0 upon reading a 0, as that does not guarantee 1 is the last character of the string. If at q_1, we remain there upon reading in additional 1's.



We conclude with the statement of Kleene's Theorem, which provides the equivalence between regular languages and finite state machines.

Theorem 14.1 (Kleene)**.** *A language L is regular if and only if it is accepted by some DFA.*

14.2.1 Exercises

(Required) Problem 112. Let L_1 be the language over $\Sigma = \{a, b\}$ where the number of a's is divisible by 3. Provide a FSM for L_1.

(Required) Problem 113. Let L_2 be the language over $\Sigma = \{0, 1\}$ where each string in L_2 contains both the substrings 00 and 11.

(Required) Problem 114. Let $\Sigma = \{0, \ldots, 9\}$. Let L_1 be the language (over Σ) of base-10 natural numbers which are even. Let L_2 be the language (over Σ) of base-10 natural numbers which are multiple of 3. [Note that ϵ is not contained in either L_1, L_2]. Assume the digits are read from most-significant to least-significant.

(a) Construct a DFA for L_1.

(b) Construct a DFA for L_2. [**Hint:** Let $n \in \mathbb{N}$. Then n is a multiple of 3 if and only if the sum of n's digits is a multiple of 3.]

(c) Construct a Finite State Machine to accept $L_1 \cdot L_2 = \{xy : x \in L_1, y \in L_2\}$.

(d) **(Challenge)** Construct a DFA to accept $L_1 \cap L_2$. [**Hint:** Think about how to run two DFAs in parallel.]

(Challenge) Problem 2. Let $n \in \mathbb{N}$. Prove that n is a multiple of 3 if and only if the sum of n's digits is a multiple of 3.

(Required) Problem 115. Construct an Finite State Machine to accept $(a + bc)(a + b)^*$.

14.3 Context-Free Grammars

In this section, we will introduce the notion of grammars to generate languages. Grammars provide a recursive set of rules used to generate strings. The recursive structure allows for effective parsing mechanisms. Grammars are particularly useful in programming and markup language design.

Recall that the goal of automata theory is to formalize the notions of an algorithm and computation machines. This is perhaps the most intuitive way to introduce context-free languages. Context-Free Languages are those languages accepted by a machine called a pushdown automaton. Conceptually, a pushdown automaton starts with a finite state automaton then adds a stack. So now the computation machine has memory aside from the current state and character. Observe as well that all regular languages are context free. This is easy to see, as a pushdown automaton can accept a regular language simply by ignoring the stack.

Some common examples of context-free languages are $\{0^n 1^n : n \geq 0\}$ and the language of balanced parentheses. Examples of strings with balanced parentheses include (()) and ()(), while (() is unbalanced. These languages

will be analyzed in greater detail later.

Formally, a context-free language is exactly the set of strings generated by a context-free grammar. The term *context-free grammar* is often times abused to denote the language itself. The context-free grammar will be formally introduced and examined. The pushdown automaton will not be covered.

Context-Free Grammar: A Context-Free Grammar is a four-tuple (N, T, P, S) where:

- N is the set of non-terminal symbols. Each non-terminal symbol represents a set of strings- exactly those strings which can be reached by it. Note that non-terminals may reach other non-terminals.
- T is the set of terminal symbols, which is equivocally the alphabet for the language.
- P is the set of productions or rules. Each production represents the recursive definition of the language. A production consists of a non-terminal symbol as the head, followed by the production symbol $\rightarrow$. The string $\omega \in (N+T)^*$ on the right-hand side of the production symbol is known as the body.
- S is the start symbol. The context-free grammar is generated starting at S and following the productions until only terminals remain.

Example 105. Consider the example above with $L = \{0^n 1^n : n \geq 0\}$. Let's construct a context-free grammar to generate the language L. Let $G = (N, T, P, S)$ be the grammar. The terminal characters are clearly $T = \{0, 1\}$. As grammars define languages recursively, the goal is to build L from the ground up. So what are the base cases? They are $\epsilon, 01$. Now using these building blocks, how is 0011 constructed? The only answer is to stick a 01 in the middle of another 01, giving 0(01)1. More generally, $0^n 1^n$ is constructed by nesting n 01 terms. So the grammar can be constructed with the single non-terminal symbol S and the production rules:

$$S \rightarrow \epsilon$$
$$S \rightarrow 01$$
$$S \rightarrow 0S1$$

More succinctly, we write:

$$S \rightarrow 0S1|01|\epsilon$$

Example 106. Now consider the language of balanced parentheses. We seek to build a grammar $G = (N, T, P, S)$ to generate this language. The terminal symbols are clearly $T = \{(,)\}$. Just like in the last example, it is important to start from the bottom up. So what are the building blocks for this language? They are $\epsilon, ()$. Now there are two cases to consider. The first is similar to the example for $L = \{0^n 1^n : n \geq 0\}$, where parentheses can be nested. The other case is when a pair of balanced parentheses are right next to each other: ()(). A single non-terminal is required, so $N = \{S\}$, and the production rules simply deal with the cases mentioned above:

$$S \rightarrow \epsilon$$
$$S \rightarrow (S)$$
$$S \rightarrow SS$$

Definition 50 (Yields Relation)**.** Let $G(N, T, P, S)$ be a grammar, and let $\alpha, \beta \in (N+T)^*$. We say that the string α *yields* β, denoted $\alpha \implies {}^*\beta$, if it is possible to obtain β starting by using the productions in P finitely many times. A *derivation* of β (from α) is the sequence of productions used to produce β from α. A *leftmost* (resp. *rightmost*) derivation is where at each stage, we replace the leftmost (resp. rightmost) non-terminal.

Example 107. Consider again the grammar:

$$S \rightarrow AB|\epsilon$$
$$A \rightarrow aAb|aAbb|\epsilon$$
$$B \rightarrow bB|\epsilon$$

A leftmost derivation of the string ab is given by:

$$S \implies AB \implies aAbB \implies abB \implies ab$$

Similarly, a rightmost derivation of the string ab is given by:

$$S \implies AB \implies A \implies aAb \implies ab$$

Definition 51 (Language of Grammar)**.** Let $G(N, T, P, S)$ be a grammar. The *language* of G, denoted:

$$L(G) = \{w \in T^* : S \implies {}^*w\},$$

is simply the set of words formed from the terminal symbols, that can be formed by starting at S and using finitely many derivations from the grammar rules.

Example 108. Consider the following grammar.

$$S \to aSc|B$$
$$B \to bBc|\epsilon$$

In order to determine the language of the grammar, we start by determining the language of each non-terminal.

- As B has no dependencies other than on itself, we start by determining the language of B, denoted $L(B)$. We note that $B \to bBc$ matches one b with one c. Furthermore, all of the b's come before all of the c's. Now the base case occurs with the rule $B \to \epsilon$. So $L(B) = \{b^n c^n | n \in \mathbb{N}\}$.
- Next, we determine the language of S. As S is the start symbol, $L(S)$ is in fact the language generated by the grammar. By similar logic as the previous case, $S \to aSc$ matches one a with one c, where the a terms all come before the c terms. Now the base case $S \to B$ indicates that the grammar selects a string from $L(B)$. So:
$$L(S) = \{a^m \omega c^m | m \in \mathbb{N}, \omega \in L(B)\}.$$
As $\omega \in L(B)$, ω is of the form $b^n c^n$. Thus, we have that the language of the grammar is:
$$\begin{aligned} L &= \{a^m b^n c^n c^m | m, n \in \mathbb{N}\} \\ &= \{a^m b^n c^{n+m} | m, n \in \mathbb{N}\}. \end{aligned}$$

Example 109. Ask students for the language of the following grammar. (**Answer:** $L = \{a^i b^j : i \leq 2j\}$):

$$S \to AB|\epsilon$$
$$A \to aAb|aAbb|\epsilon$$
$$B \to bB|\epsilon$$

In order to determine the language of this grammar, we determine $L(A)$ and $L(B)$ first. Now the language of the grammar is $L(A) \cdot L(B) \cup \{\epsilon\}$. Here, $L(A) \cdot L(B)$ is the concatenation of $L(A)$ and $L(B)$.

- We first determine $L(B)$. We note that B generates a finite sequence of b characters, with the base case $B \to \epsilon$. So $L(B) = \{b^n | n \in \mathbb{N}\} = b^*$.
- We next determine $L(A)$. We note that A matches an a with either b or bb. The base case is $A \to \epsilon$. So $L(A) = \{a^i b^j | i, j \in \mathbb{N}, j \leq 2i\}$.
Now $L(S) = L(A) \cdot L(B)$, where $\epsilon \in L(A)$ and $\epsilon \in L(B)$. So the language of the grammar is as follows:
$$\begin{aligned} L &= \{a^i b^j b^k | i, j, k \in \mathbb{N}, j \leq 2i\} \\ &= \{a^i b^{j+k} | i, j, k \in \mathbb{N}, j \leq 2i\}. \end{aligned}$$

14.3.1 Exercises

(Required) Problem 116. Construct a grammar over $\{a, b, c\}$ whose language is $\{a^m b^{2n} c^m : m, n > 0\}$.

(Required) Problem 117. Construct a grammar over $\{a, b, c\}$ whose language is $\{a^n b^m c^{2n+m} : m, n > 0\}$.

(Required) Problem 118. Determine the language of the following grammar.

$$S \to AC$$
$$A \to aAc|B$$
$$B \to bBc|\epsilon$$
$$C \to cC|\epsilon$$

(Required) Problem 119. Determine the language of the following grammar.

$$S \to aSaa|B$$
$$B \to bbBdd|C$$
$$C \to bd$$

15 Proof by Induction

Many theorems of interest ask us to prove a proposition holds for all natural numbers. Verifying such statements for all natural numbers is challenging, due to the fact that our domain is not just large but infinite. Informally, proof by induction allows us to verify a small subset of base cases. Together, these base cases imply the subsequent cases. Thus, the desired theorem is proven as a result.

Intuitively, we view the statements as a sequence of dominos. Proving the necessary base cases knocks (i.e., proves true) the subsequent dominos (statements). It is inescapable that all the statements are knocked down; thus, the theorem is proven true.

The most basic form of induction is the Principle of Weak Induction.

Definition 52 (Principle of Weak Induction)**.** Let $P(n)$ be a proposition regarding an integer n, and let $k \in \mathbb{Z}$ be fixed. If:

(a) $P(k)$ holds; and

(b) For every $m \geq k$, $P(m)$ implies $P(m + 1)$

Then for every $n \geq k$, $P(n)$ holds.

The definition of the Principle of Weak Induction in fact provides a format for structuring proofs.

- First, we verify a single base case: for some $k \in \mathbb{N}$, the statement $P(k)$ holds.
- Second, we assume that $P(m)$ holds for some integer $m \geq k$. This step is known as the **inductive hypothesis**, and it is indispensible for a proof by induction. We must and do use the fact that $P(m)$ is true when proving that $P(m + 1)$ holds.
- In our final step, we show that for an arbitrary $m \geq k$ that $P(m)$ implies $P(m + 1)$. This is known as the **inductive step**.

We illustrate this proof technique with the following example.

Proposition 15.1. *Fix $n \in \mathbb{N}$. We have:* $\displaystyle\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Proof. We prove this theorem by induction on $n \in \mathbb{N}$. Our first step is to verify the base case: $n = 0$. In this case, we have $\sum_{i=0}^{n} i = 0$. Note as well that $\frac{0 \cdot 1}{2} = 0$. Thus, the proposition holds when $n = 0$.

Now for our inductive hypothesis: suppose the proposition holds true for some $k \geq 0$. This is usually, though not always, quite easy to check.

Inductive Step: We prove true for the $k + 1$ case. Consider:

$$\sum_{i=0}^{k+1} i = k + 1 + \sum_{i=0}^{k} i$$

By the inductive hypothesis, $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$. Now we have:

$$\begin{aligned} & k + 1 + \sum_{i=0}^{k} i \\ &= k + 1 + \frac{k(k+1)}{2} \\ &= \frac{2(k+1) + k(k+1)}{2} \\ &= \frac{k^2 + 3k + 2}{2} \\ &= \frac{(k+1)(k+2)}{2} \end{aligned}$$

As desired. So by the Principle of Weak Induction, the result follows. □

Notice that the inductive hypothesis was imperative in the inductive step. Once we used that $\sum_{i=0}^{k} i = \frac{k(k+1)}{2}$, it was a matter of algebraic manipulation to obtain that $\sum_{i=0}^{k+1} i = \frac{(k+1)(k+2)}{2}$. As we have verified a base case when $n = 0$ and proven that $S(k)$ implies $S(k + 1)$ for an arbitrary $k \geq 0$, the Principle of Weak Induction affords us that the proposition is true.

We now examine a second example applying the Principle of Weak Induction.

Proposition 15.2. *For each $n \in \mathbb{N}$ and each $x > -1$, $(1 + x)^n \geq 1 + nx$.*

Proof. The proof is by induction on n. Consider the base case of $n = 0$. So we have $(1 + x)^n = 1 \geq 1 + 0x = 1$. So the proposition holds at $n = 0$.

Now for our inductive hypothesis: suppose that $(1 + x)^k \geq 1 + kx$ for some $k \geq 0$.

Inductive Step: We have that $(1 + x)^{k+1} = (1 + x)^k(1 + x)$. By the inductive hypothesis, $(1 + x)^k \geq (1 + kx)$. So:

$$(1 + x)^k(1 + x) \geq (1 + kx)(1 + x) = 1 + (k + 1)x + kx^2 \geq 1 + (k + 1)x$$

Where the last inequality follows from the fact that kx^2 is non-negative; so removing it from the right hand side will not increase that side. So by the Principle of Weak Induction, the result follows. □

We next introduce the Principle of Strong Induction. Intuitively, strong induction is useful in proving theorems of the form "for all n, $P(n)$" where $P(k)$ alone does not neatly lend itself to forcing $P(k+1)$ to be true. Instead, it may be easier to leverage some subset of $\{P(0), \ldots, S(k)\}$ to force $P(k + 1)$ to be true. Strong induction allows us to use any or all of $P(0), \ldots, P(k)$ to prove that $P(k + 1)$ is true. The Principle of Strong Induction is formalized as follows.

Definition 53 (Principle of Strong Induction)**.** Let $P(n)$ be a proposition regarding an integer n, and let $k \in \mathbb{Z}$ be fixed. If:

- $P(k)$ is true; and
- For every $m \geq k$, $[P(k) \wedge P(k+1) \wedge \ldots \wedge P(m)]$ implies $P(m+1)$

Then for every $n \geq k$, the statement $P(n)$ is true.

Just as with the Principle of Weak Induction, the Principle of Strong Induction provides a format for structuring proofs.

- First, we verify for all base cases k, the statement $P(k)$ holds. This ensures that subsequent cases which rely on these early base cases are sound. For example, strong inductive proofs regarding graphs or recurrence relations may in fact have several base cases, which are used in constructing subsequent cases.
- Second, we assume that for some integer $m \geq k$, $P(k), \ldots, S(m)$ **all** hold. Notice our inductive hypothesis using strong induction assumes that **each** of the previous cases are true, while the inductive hypothesis when using weak induction only assumes $P(m)$ to be true. Strong induction assumes the extra cases because we end up using them.
- In our final step (the inductive step), we show that for an arbitrary $m \geq k$ that $P(k) \wedge P(k+1) \wedge \ldots \wedge P(m)$ implies

Remark: The Principle of Weak Induction and the Principle of Strong Induction are equally powerful. That is, any proof using strong induction may be converted to a proof using weak induction, and vice versa. In practice, it may be easier to use strong induction, while weak induction may be clunky to use.

We illustrate how to apply the Principle of Strong Induction with a couple examples.

Proposition 15.3. *Let $f_0 = 0, f_1 = 1$; and for each natural number $n \geq 2$, let $f_n = f_{n-1} + f_{n-2}$. We have $f_n \leq 2^n$ for all $n \in \mathbb{N}$.*

Proof. The proof is by strong induction on $n \in \mathbb{N}$. Observe that $f_0 = 0 \leq 2^0 = 1$. Similarly, $f_1 = 1 \leq 2^1 = 2$. So our base cases of $n = 0, 1$ hold. Now fix $k \geq 1$; and suppose that for all $n \in \{0, \ldots, k\}$, $f_n \leq 2^n$. We now prove that $f_{k+1} \leq 2^{k+1}$. By definition of our sequence, $f_{k+1} = f_k + f_{k-1}$. We now apply the inductive hypothesis to f_k and f_{k+1}. Thus:

$$\begin{aligned} f_{k+1} &\leq 2^k + 2^{k-1} \\ &= 2^k \left(1 + \frac{1}{2}\right) \\ &\leq 2^k \cdot 2 = 2^{k+1} \end{aligned}$$

As desired. So by the Principle of Strong Induction, the result follows. □

Proposition 15.4. *Every positive integer can be written as the product of a power of 2 and an odd integer.*

Proof. The proof is by strong induction on $n \in \mathbb{Z}^+$. We have our base case $n = 1$. So $n = 1 \cdot 2^0$. Thus, the proposition holds for $n = 1$. Now fix $k \geq 1$, and suppose the proposition holds true for all $n \in [k]$. We prove true for the $k + 1$ case. We have two cases:

- Case 1: Suppose $k + 1$ is odd. Then $k + 1 = (k + 1) \cdot 2^0$, and we are done.
- Case 2: Suppose instead $k + 1$ is even. Then $k + 1 = 2h$ for some $h \in \mathbb{Z}^+$. By the inductive hypothesis, $h = m2^j$ for some odd integer m and $j \in \mathbb{N}$. Thus, $k + 1 = m2^{j+1}$, and we are done.

So by the Principle of Strong Induction, the result follows. □

15.1 Exercises

(Required) Problem 120. Prove by induction that $3^n < n!$ for all $n > 6$. [Note- This is one way of showing that $3^n \in \mathcal{O}(n!)$].

(Required) Problem 121. Prove by induction that:

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

(Required) Problem 122. Prove by induction that for every integer $n \geq 8$, there exist constants $a, b \in \mathbb{Z}$ such that:

$$n = 3a + 5b.$$

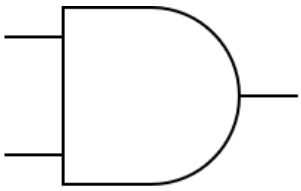
(Advanced) Problem 8. The *Fibonacci sequence* is given by $F_0 = 0, F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. Prove by induction that $\gcd(F_{n+1}, F_n) = 1$ for all $n \in \mathbb{N}$. [**Hint:** Apply the Euclidean algorithm- the variant **without** the modulus operator.]

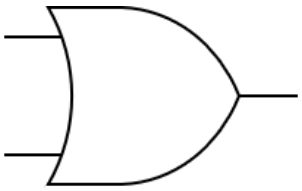
16 Combinatorial Circuits

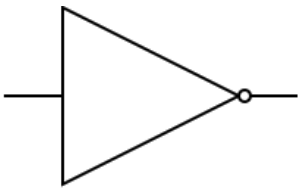
The motivation for circuits is apparent, simply by taking apart one's computer. At the hardware level, much of a computer's functionality is governed by circuits. More abstractly, circuits readily appear in theoretical computer science, such as in computational complexity [13]. Furthermore, circuits are the standard model of quantum computation [4]. In computer science and electrical engineering, there are two classes of circuits: *combinatorial circuits* and *sequential circuits*. The combinatorial circuit model studies circuits whose outputs only depend on the inputs and not the state of the circuit. That is, combinatorial circuits do not have memory [11, 13]. In contrast, sequential circuits may take into account the current state of the circuit in its calculations. In this section, attention is restricted to combinatorial circuits.

16.1 Logic Gates

In this circuit model of computation, we have nodes that are either inputs or logic gates. For instance, the `AND`, `OR`, and `NOT` gates are pictured below. Notice the `AND` and `OR` gates each have two inputs and return one output. Similarly, the `NOT` gate has a single input and returns a single output.

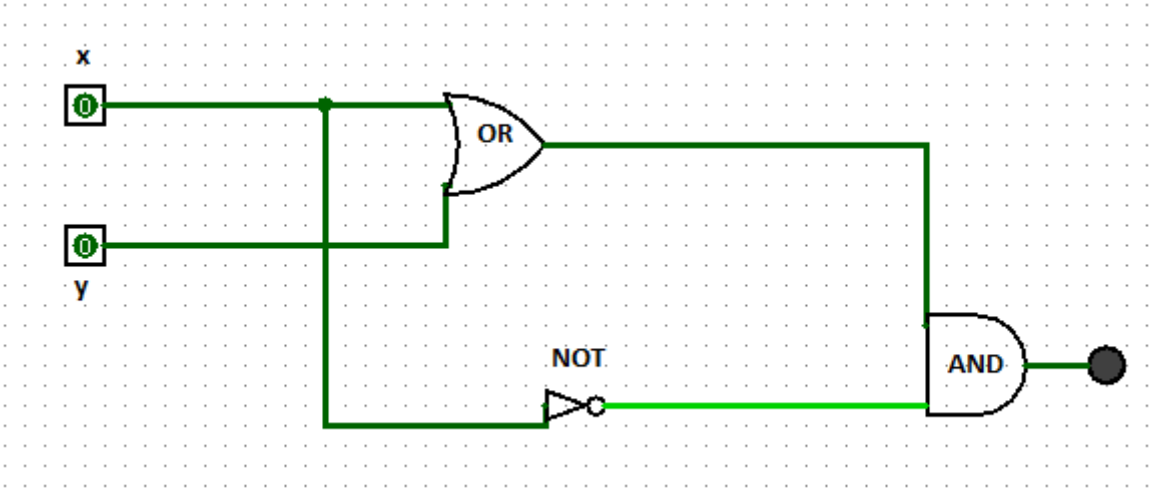

AND


OR


NOT

The goal of designing combinatorial circuits is to utilize the logic gates to compute Boolean functions. We illustrate this with an example.

Example 110. The following circuit computes $(x \vee y) \wedge \neg x$. Note that when both $x = 0$ and $y = 0$, the circuit outputs 0.



16.1.1 Exercises

(Required) Problem 123. Using only the `AND`, `OR`, and `NOT` gates, implement a circuit to compute $x \oplus y$.

(Required) Problem 124. Suppose we have a committee of three people voting on a motion, which passes by majority vote of two or more `YES` votes. Design a circuit to determine if the motion passes.

(Required) Problem 125. A *Half-Adder* is a circuit that adds to single-digit binary numbers x and y. The Half-Adder outputs the sum $x + y \pmod 2$, as well as the carry bit. Note that the carry bit is a 1 precisely if a carry is generated by the addition .

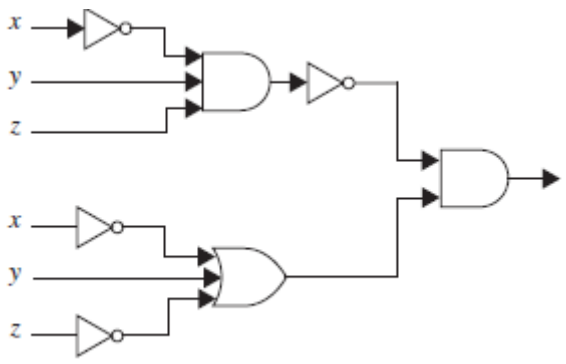
1. Implement the Half-Adder using only the `AND`, `OR`, and `NOT` gates. Your circuit in Problem 123 may be helpful.
2. Implement the Half-Adder, this time using the `XOR` gate. You may still use the `AND`, `OR`, and `NOT` gates. Comment on the difference in the number of gates required between your implementations in part (a) vs. part (b).

(Advanced) Problem 9. A *Full-Adder* generalizes the Half-Adder, in that it takes two single-digit bits as input and outputs their sum as a two-digit binary value. Additionally, the Full-Adder takes into account a carry bit from a previous adder circuit. Formally, the Full-Adder takes as input three Boolean values x, y, and c_i and outputs a two-digit binary string $(c_i + c, x + y + c_i)$, where c is the carry bit generated from adding $x + y + c_i$. Note that $x + y$ and $c_i + c$ are taken modulo 2.

Implement a Full-Adder circuit. You may use your Half-Adder circuit, as well as the `AND`, `OR`, `NOT`, and `XOR` gates.

(Challenge) Problem 3. Suppose $x = (x_2, x_1, x_0)$ and $y = (y_2, y_1, y_0)$ are three-digit binary numbers. Using your Full-Adder and Half-Adder circuits, design a circuit to add $x + y$. Your circuit should output each digit of $x + y$, as well as a carry bit.

(Required) Problem 126. Determine the Boolean expression that the following circuit computes.



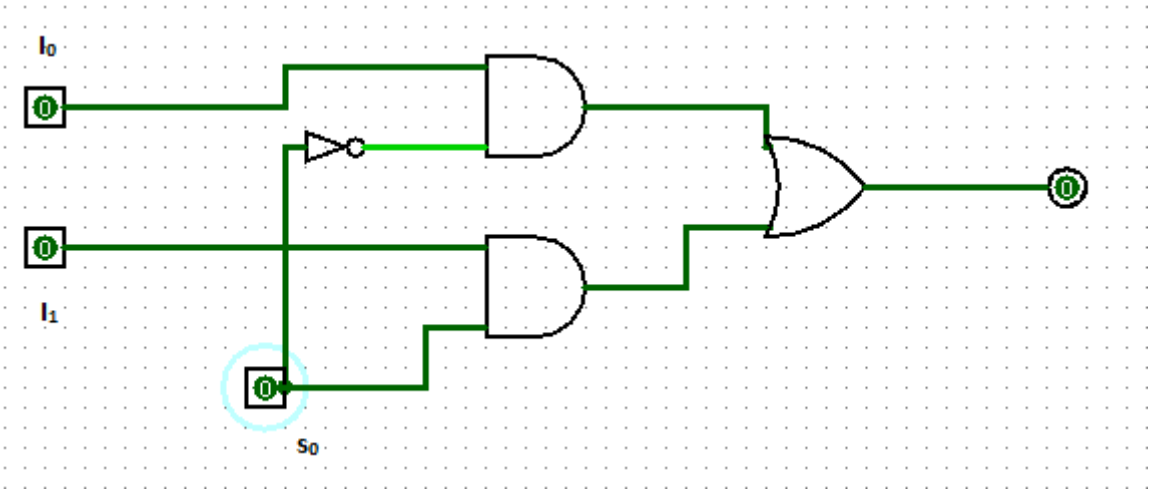
16.2 Multiplexers

In this section, we discuss multiplexers. Intuitively, a multiplexer accepts inputs and provides a means to specify which of these inputs are passed along to a single output. This is formalized as follows.

Definition 54 (Multiplexer)**.** Fix $n \in \mathbb{N}$. A $2^n - n$ multiplexer (mux) accepts two types of inputs, with a single output. The two types of inputs are as follows:

- 2^n binary inputs $I_{2^{n-1}}, I_{2^{n-2}}, \ldots, I_1, I_0$, and
- n binary inputs that serve as selection bits $s = s_{n-1}s_{n-2}\ldots s_1 s_0$. In particular, we view s as the binary representation of a base-10 number, which specifies which of the 2^n inputs $I_{2^{n-1}}, \ldots, I_0$ are selected for the output.

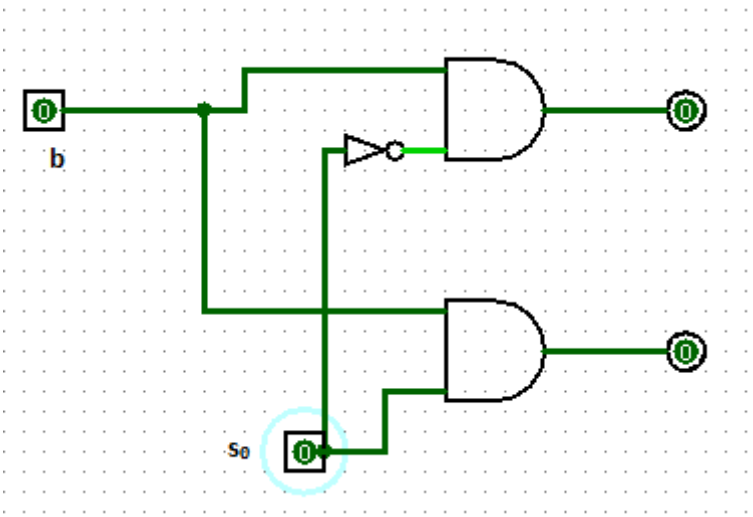
Example 111. Consider the following $2-1$ multiplexier. Note that when $s_0 = 0$, then this circuit returns the value of I_0. If instead $s_0 = 1$, then this circuit returns the value of I_1.



Now suppose that I_k is selected as the output value, there are $2^{2^{n-1}}$ possible configurations of the remaining 2^{n-1} binary inputs $I_{2^{n-1}}, \ldots, I_{k+1}, I_{k-1}, \ldots, I_0$. So while it is not possible to recover each of the 2^n binary inputs $I_{2^{n-1}}, I_{2^{n-2}}, \ldots, I_1, I_0$ from the output of the multiplexer and configuration of the selection bits, it is possible to recover the state of I_k. This motivate the demultiplexer.

Definition 55 (Demultiplexer)**.** Let $n \in \mathbb{Z}^+$. An n-2^n demultiplexer (demux) takes as input a single bit b, as well as n selection bits $s = s_{n-1}s_{n-2}\ldots s_1 s_0$, and outputs 2^n bits. The base-10 representation of s specifies the output bit which matches the input bit b. All other output bits receive the value 0.

Example 112. Consider the following $1-2$ demultiplexer. Note that when $s_0 = 0$, the top output bit corresponds to b. When $s_0 = 1$, the bottom output bit corresponds to b.



16.2.1 Exercises

(Required) Problem 127. Construct a $4-2$ multiplexer.

(Required) Problem 128. Construct a $2-4$ demultiplexer.

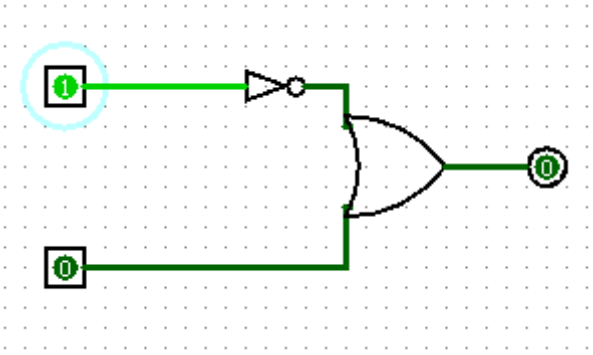
16.3 Encoders

An encoder is a circuit, which is used to specify the high-order bit from 2^n inputs. This is formalized as follows.

Definition 56 (Encoder)**.** Let $n \in \mathbb{N}$. A $2^n - n$ encoder takes 2^n outputs. Exactly one of the 2^n inputs is set to 1; the other $2^n - 1$ inputs are set to 0. The n outputs are viewed as the binary representation of the base-10 number indicating the high-order bit.

Remark: Note that the definition of an encoder assumes at exactly one of the inputs is set to 1. Otherwise, the output is invalid.

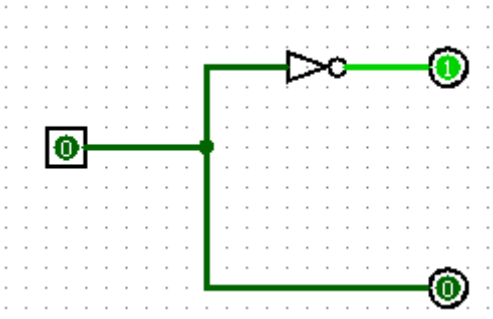
Example 113. Consider the example of a $2-1$ encoder. Observe that if only the top bit is set to 1 (that is, the input is 01), then the output of this circuit is 0. If instead the bottom bit is set to 1 and the top bit is set to 0, then the output of this circuit is 1.



We next discuss the decoder circuit, which allows us to recover the high bit for 2^n outputs given n inputs specifying the position. This is formalized as follows.

Definition 57 (Decoder)**.** Let $n \in \mathbb{Z}^+$. An $n - 2^n$ decoder accepts as input n bits, which constitute the binary representation of a base-10 number. In turn, the base-10 number specifies the single output bit that is set to 1. The other 2^{n-1} output bits are set to 0.

Example 114. We consider a $1-2$ decoder circuit. If the input bit is 0, then the top output bit is 1. Otherwise, only the bottom output bit is 1.



16.3.1 Exercises

(Required) Problem 129. The $2-1$ encoder is the only encoder circuit requiring use of the `NOT` gate. For all other values of n, the $2^n - n$ encoder circuit can be implemented using only `OR` gates. Implement the $4-2$ encoder. [**Hint:** Start by examining the binary strings of length 2. When is the second bit 1? When is the first bit 1? Clearly articulate this logic in plain English before writing the circuit.]

(Required) Problem 130. Implement the $8-3$ encoder.

(Required) Problem 131. Implement a $2-4$ decoder.

16.4 Boolean Normal Forms

It is often desirable to represent a Boolean function using a *normal form*, which satisfies certain structural properties. In this section, we introduce the *Conjunctive Normal Form* (CNF) and *Disjunctive Normal Form* (DNF), which are of key importance in digital logic and computational complexity. We motivate this as follows. Suppose we are given a Boolean function $\varphi : \{0,1\}^n \to \{0,1\}$, and we wish to determine if there is a sequence of inputs $(x_1, \ldots, x_n)$ such that $\varphi(x_1, \ldots, x_n) = 1$. In general, this is a computationally difficult problem[8]. However, if a Boolean function can be represented efficiently, using only polynomially many terms, it may be possible to efficiently check if it is satisfiable. It turns out that checking if a Boolean function in CNF is satisfiable, remains NP-Complete. However, the satisfiability problem is efficiently solvable for Boolean functions in DNF.

We begin with the definitions of CNF.

Definition 58 (Clause)**.** A *clause* is a Boolean function $\varphi : \{0,1\}^n \to \{0,1\}$ where φ consists of input variables or their negations, all added together (where addition is the OR operation).

Definition 59 (Conjunctive Normal Form)**.** Let $C_1, \ldots, C_k$ be clauses. We say that $\psi : \{0,1\}^n \to \{0,1\}$ is in *Conjunctive Normal Form* if:

$$\psi = C_1 \wedge C_2 \wedge \ldots \wedge C_k.$$

It is helpful to think of CNF as an AND or ORs.

We now consider some examples of CNF.

Example 115. The following functions are all in CNF.

- $(A \vee \neg B \vee C) \wedge (\neg D \vee E \vee F)$
- x
- $x \vee y$
- $(x \vee y) \wedge z$

In contrast, the following functions are **not** in CNF.

- $\neg(x \vee y)$ is not in CNF, as the OR is nested within the NOT. Note that $\neg(x \vee y)$ could be writtin in CNF, as follows: $\neg x \vee \neg y$.
- $(x \wedge y) \vee z$

We next introduce DNF.

Definition 60 (Disjunctive Normal Form)**.** A Boolean function $\varphi : \{0,1\}^n \to \{0,1\}$ is in *Disjunctive Normal Form* (DNF) if:

$$\varphi = D_1 \vee D_2 \ldots \vee D_k,$$

where each D_i consists of variables or their negations all multiplied together (where multiplication is the AND operation).

It is helpful to think of DNF as an OR or ANDs.

We now consider some examples of DNF.

Example 116. The following formulas are in DNF.

- $(x \wedge y \wedge \neg z) \vee (\neg a \wedge b \wedge c)$
- $(x \wedge y) \vee z$

[8]This is referred to as the Boolean Satisfiability Problem or SAT, which is known to be NP-Complete.

- $x \wedge y$
- x

The following formulas are not in DNF.

- $\neg(x \vee y)$ is not in DNF, as the `OR` is nested inside the `NOT`. Note that $\neg x \wedge \neg y$ is also in DNF.
- $x \vee (y \wedge (c \vee d))$ is not in DNF, as the `OR` is nested inside the `AND`.

We next discuss how to construct the CNF and DNF representations of a Boolean function. We begin by examining the truth table of the Boolean function. The rows that evaluate to 1 provide the basis for the DNF, keeping the values for each variable in their respective columns. In order to compute the CNF, we examine the rows that evaluate to 0 and take the negations of each value.

Example 117. Consider the function $\varphi : \{0,1\}^3 \to \{0,1\}$, given by the following truth table.

x	y	z	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

We now compute the CNF and DNF.

- **CNF:** To compute the CNF, we examine the rows that evaluate to 0. We include rows $1, 4, 6$, and 7. For each row, we invert the literals. So if $x = 0$ in the given row, we record x. If instead $x = 1$, we record $\neg x$. These rows contribute the following clauses.
  - **Row 1:** $(x \vee y \vee z)$
  - **Row 4:** $(x \vee \neg y \vee \neg z)$
  - **Row 6:** $(\neg x \vee y \vee \neg z)$
  - **Row 7:** $(\neg x \vee \neg y \vee z)$

  So the CNF formulation of φ is:

$$\varphi = (x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z).$$

- **DNF:** To compute the DNF, we examine the rows that evaluate to 1. We include rows $2, 3, 5$, and 8. These rows contribute the following terms.
  - **Row 2:** $(\neg x \wedge \neg y \wedge z)$
  - **Row 3:** $(\neg x \wedge y \wedge \neg z)$
  - **Row 5:** $(x \wedge \neg y \wedge \neg z)$
  - **Row 7:** $(x \wedge y \wedge z)$

  So the DNF formulation of φ is:

$$\varphi = (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge \neg z) \vee (x \wedge y \wedge z).$$

16.4.1 Exercises

(Required) Problem 132. Suppose φ is given by the following truth table.

x	y	z	
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

(a) Write φ in CNF.

(b) Write φ in DNF.

(Required) Problem 133. Consider the following Boolean functions in DNF. For each function ϕ, determine if there is a sequence of inputs such that ϕ evaluates to 1 on those inputs.

(a) $x \wedge \neg x$

(b) $(x_1 \wedge \neg x_2 \wedge x_3 \wedge x_4) \vee (\neg x_1 \wedge \neg x_2 \wedge x_1 \wedge x_3)$

(c) $(x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3)$

(Advanced) Problem 10. Suppose $\varphi : \{0,1\}^n \to \{0,1\}$ is a Boolean function in DNF. Describe an efficient algorithm to check whether φ has a satisfying instance.

(Required) Problem 134. The `NAND` operation, $\texttt{NAND}(x, y)$ computes $\neg(x \wedge y)$. Show how to use the `NAND` operation can be used to realize the `AND`, `OR`, and `NOT` operations.

17 Recursion

Recursion is a problem solving technique that examines smaller instances of the same problem. After using the information obtained by solving the smaller instances, the recursive approach then constructs a solution to the original instance. We illustrate recursion with the following meme.

In this chapter, we introduce recursion as an algorithmic technique, as well as discuss how the Java call stack handles recursive method calls.

17.1 Call Stack

TODO

17.2 Applying Recursion

When designing a recursive approach, a bottom-up approach is employed. That is, we first consider the minimal cases, which we refer to as the base cases. The recursive calls repeatedly construct smaller instances of the same problem, until a base case is reached. So in any recursive solution, there are two components: (i) the base case and (ii) the recursive call(s). We illustrate this with some examples.

Example 118. We first discuss computing the factorial recursively. Recall that $n! = n \cdot (n-1) \cdots 2 \cdot 1$. In particular, we observe that: $n! = n \cdot (n-1)!$, which suggests the recrusive call: we invoke the factorial method passing $n-1$ as the input and then multiply the result by n.

We now examine the base cases. Recall that $0! = 1! = 1$, which we set as our base cases. Consider the following method. The base case is first considered by the if statement. This ensures that we avoid recursively invoking the `factorial()` method non-stop. For the sake of handling negative numbers, we adopt the convention that if $n < 0$, `factorial(n)` returns 1.

```
public int factorial(int n){
        if(n <= 1){
                return 1;
        }

        return n * factorial(n-1);
}
```

We now walk through how `factorial(4).` is computed.

- Initially, `factorial(4)` is invoked. The `factorial(4)` call invokes `factorial(3)`.
- Next, `factorial(3)` invokes `factorial(2)`.
- Finally, `factorial(2)` invokes `factorial(1)`.
- Now `factorial(1)` returns 1 and returns control to `factorial(2)`.
- Next, `factorial(2)` uses the value of 1 returned from `factorial(1)`, in order to return $2 \cdot 1$. Control is now returned to `factorial(3)`.
- Next, `factorial(3)` uses the value of 2 returned from `factorial(2)`, in order to return $3 \cdot 2$. Control is now returned to `factorial(4)`.
- Finally, `factorial(4)` uses the value of 6 returned from `factorial(3)`, in order to return $4 \cdot 6$. Control is now returned to the method that invoked `factorial(4)`.

Recursion need not be linear or sequential, as in the case of the factorial. It is often the case that a recursive method makes multiple recursive calls. We illustrate this with a recursive method to compute the nth Fibonacci number.

Example 119. Recall that the Fibonacci numbers are defined as follows: $F_0 = 0$, $F_1 = 1$; and if $n \geq 2$, then $F_n = F_{n-1} + F_{n-2}$. So if we wish to compute F_n for $n \geq 2$, it suffices to compute F_{n-1} and F_{n-2}. Of course, if either of $n-1 \geq 2$ or $n-2 \geq 2$, then we need additional recursive calls. Note that `fib(n-1)` must return control before `fib(n-2)` is invoked.

```
public int fib(int n){
        System.out.println("Fib: " + n);
        if(n <= 1){
                return n;
        }
```

```
        return fib(n-1) + fib(n-2);
}
```

For the sake of illustrating the recursive calls, a print statement has been included at the top of the method. The output of invoking `fib(4)` is as follows.

```
Fib: 4
Fib: 3
Fib: 2
Fib: 1
Fib: 0
Fib: 1
Fib: 2
Fib: 1
Fib: 0
```

17.2.1 Exercises

Note that the goal of the following exercises is to force you to practice with recursion. While there are often iterative solutions, applying them here would defeat the purpose of learning recursion.

(Required) Problem 135. Write a **recursive** method named `allStar()` that takes as input a String `str` and returns a new String, where all the characters in `str` are separated by `*`. So for example, `allStar(abcd)` returns `a*b*c*d`.

(Required) Problem 136. Write a **recursive** method `count2()` which takes as input an `int n` and returns the number of 2's that occur in `n`. So for example, `count2(12) = 1`, while `count2(666) = 0`. You may only use the division and modulus operators, and not any String class functionality.

(Advanced) Problem 11. Write a **recursive** method `makeChange()` which takes as input an `int n`, the monetary amount in pennies for which to make change. The method returns the minimum number of coins needed to make change for n using quarters, dimes, nickels, and pennies.

(Required) Problem 137. Write a **recursive** method `nestParen()` which takes as input a String `str`. The method returns `true` if it is nesting zero or more pairs of parentheses. The method returns false otherwise. For example:

`nestParen("(())")` → `true`
`nestParen("((()))")` → `true`
`nestParen("(((x))")` → `false`

17.3 Mergesort

Mergesort is a recursive algorithm, which partitions the array in half. The algorithm then sorts each half, after which it merges the sorted arrays together. The mergesort algorithm is significantly more efficient than insertion sort, selection sort, and bubblesort, having time complexity $\mathcal{O}(n \log(n))$, while the other three algorithms all have time complexity $\mathcal{O}(n^2)$.

We decompose the algorithm to identify the base cases and the recursive calls.

- **Base Cases:** There are two base cases.
  - **Case 1:** Suppose that the input array has at most 1 element. In this case, the array is already sorted. So there is nothing more to be done, and the algorithm returns.

- **Case 2:** Suppose the input array has at 2 elements. If the array is sorted, then there is nothing more to be done. So the algorithm returns. If instead the elements are out of order, then the algorithm swaps them before returning.

- **Recursive Case:** The algorithm first creates two new arrays, named `left` and `right`. Denote n as the length of the array. The `left` array stores the first $n/2$ elements, while the `right` array stores the last $n/2$ elements. The algorithm recursively sorts `left` first. Afterwards, the algorithm then recursively sorts `right`. After `left` and `right` are sorted, then the algorithm merges them back into the original array. In the merge step, care is taken to ensure the elements are inserted in order, so that the array is sorted when the method terminates.

We provide the following pseudocode.

```
mergesort(int[] arr){

  if arr.length <= 1
      return

   else if arr.length == 2
      if arr[0] > arr[1]
          swap(arr, 0, 1)
       return

  int[] left <-- elements in arr from 0 through arr.length/2 - 1
  int[] right <-- elements in arr from arr.length/2 onwards

  mergesort(left)
  mergesort(right)

  merge(arr, left,right)
```

Example 120. We work through an example illustrating mergesort. Suppose we wish to sort `[2,3,1,5,7,6,4]` using mergesort.

- The algorithm begins by partitioning the array into two arrays, `left = [2, 3, 1, 5]` and `right = [7, 6, 4]`. Next, the algorithm recursively sorts the `left` array.

  - **Recursive Call:** `mergesort(left)`. The algorithm partitions `[2, 3, 1, 5]` into two subarrays: `[2, 3]` and `[1, 5]`. The algorithm then recursively sorts `[2, 3]` and `[1, 5]`.
    - **Recursive Call:** `mergesort([2, 3])`. As `[2, 3]` has length 2 and is sorted, the algorithm returns.
    - **Recursive Call:** `mergesort([1, 5])`. As `[1, 5]` has length 2 and is sorted, the algorithm returns.

    Next, the algorithm merges `[2, 3]` and `[1, 5]`, leaving the array `[1, 2, 3, 5]`.
  - **Recursive Call:** `mergesort(right)`. The algorithm partitions `[7, 6, 4]` into two subarrays: `[7, 6]` and `[4]`.
    - **Recursive Call:** `mergesort([7, 6])`. As `[7, 6]` has length 2 and is not sorted, the algorithm swaps the elements, leaving: `[6, 7]`.
    - **Recursive Call:** `mergesort([4])`. As `[4]` has length 1, the algorithm returns.

    The algorithm merges `[6, 7]` and `[4]`, leaving the array `[4, 6, 7]`.

  Finally, the algorithm merges `[1, 2, 3, 5]` and `[4, 6, 7]`, leaving `[1, 2 , 3, 4, 5, 6, 7]` as the final sorted array. The algorithm now returns.

17.3.1 Exercises

(Required) Problem 138. Implement Mergesort in Java. Note that you will have to implement the `merge()` method.

References

[1] Java language specification: Conversions and contexts. URL https://docs.oracle.com/javase/specs/jls/se10/html/jls-5.html.

[2] David Aspnes. Notes on discrete mathematics. 9 2018. URL http://www.cs.yale.edu/homes/aspnes/classes/202/notes.pdf.

[3] Reinhard Diestel. *Graph Theory*. 2010.

[4] Stephen Fenner. *Course Notes for CSCE 790-002 Quantum Computing and Information Fall 2017*. 2017. URL https://cse.sc.edu/~fenner/csce790/notes/course-notes.pdf.

[5] G.N. Hile. 3e lewis carroll puzzles. URL http://www.math.hawaii.edu/~hile/math100/logice.htm.

[6] Michael Levet. Theory of computation- lecture notes. 2018. URL http://people.math.sc.edu/mlevet/Lecture_Notes.pdf.

[7] Nicholas A. Loehr. *Combinatorics*. CRC Press Taylor & Francis Group, 2018.

[8] Oracle. What is an object? URL https://docs.oracle.com/javase/tutorial/java/concepts/object.html.

[9] David Pearson. A polynomial-time algorithm for the change making problem. 6 1994. URL https://graal.ens-lyon.fr/~abenoit/algo09/coins2.pdf.

[10] Kenneth Rosen. *Elementary Number Theory*. 6 edition, 2011.

[11] Kenneth Rosen. *Discrete Mathematics and Its Applications*. 7 edition, 2012.

[12] G S. Lueker. Two np-complete problems in non negative integer programming. *Computer Science Laboratory*, 178, 01 1975.

[13] John Savage. *Models of Computation*. 2008. URL http://cs.brown.edu/people/jsavage/book/.

[14] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2 edition, September 2000. ISBN 0130144002.