Essentials of Computer Science Principles with JavaScript

Kenneth Carter Dodd

2015

Contents

Chapter 1

Information

We use the word information frequently. Think of an example that you think contains information. Perhaps you thought of a book or a website. Can you identify what exactly about these things contain information, or how you might quantify an amount of information it contains?

1.1 20 Questions

If you are not familiar with it, 20 Questions is a game where one person thinks of an object which other people must guess. The person who knows the object may be asked any yes or no question until 20 such questions have been answered. The premise of the game is that if the questions are clever enough, only 20 questions are needed to guess what it is. But, if the questions are not good then it may not be enough to guess the object. But think of how many objects actually exist; could 20 questions really be enough to narrow it down?

Lets start with a simple example and limit the possibilities to the letters of the English alphabet. There are 26 letters. How many yes or no questions are required to guess one of the letters? One could adopt the strategy of asking questions like "is it the letter A?". This is a gamble. On the one hand, the letter could be guessed on the first try if the answer is 'yes'. But this type of question only eliminates one possibility at a time if the answer is 'no', and so it could take up to 25 questions to eliminate all but one of the letters.

```
         ABCDEFGHIJKLMNOPQRSTUVWXYZ
Guess A:  BCDEFGHIJKLMNOPQRSTUVWXYZ
Guess I:  BCDEFGH JKLMNOPQRSTUVWXYZ
```

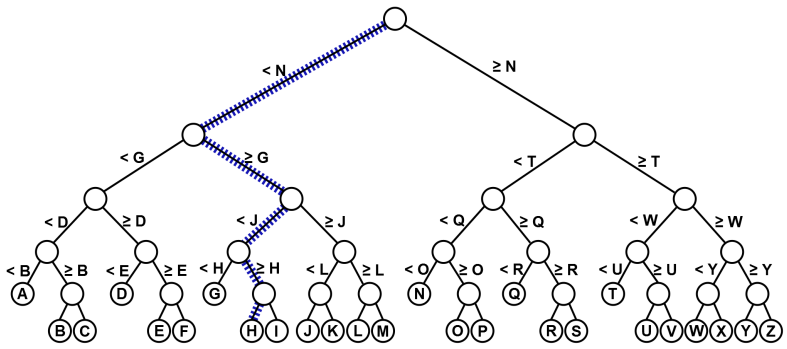
```
Guess E:  BCD FGH JKLMNOPQRSTUVWXYZ
Guess Y:  BCD FGH JKLMNOPQRSTUVWX Z
Guess X:  BCD FGH JKLMNOPQRSTUVW  Z
Guess G:  BCD F H JKLMNOPQRSTUVW  Z
Guess Z:  BCD F H JKLMNOPQRSTUVW
Guess Q:  BCD F H JKLMNOP RSTUVW
Guess P:  BCD F H JKLMNO  RSTUVW
Guess V:  BCD F H JKLMNO  RSTU W
Guess U:  BCD F H JKLMNO  RST  W
Guess W:  BCD F H JKLMNO  RST
Guess O:  BCD F H JKLMN   RST
Guess M:  BCD F H JKL N   RST
Guess S:  BCD F H JKL N   R T
Guess T:  BCD F H JKL N   R
Guess R:  BCD F H JKL N
Guess B:   CD F H JKL N
Guess D:   C  F H JKL N
Guess K:   C  F H J L N
Guess C:      F H J L N
Guess J:      F H   L N
Guess L:      F H     N
Guess N:      F H
Guess F:        H
```

We could ask questions like "does the letter come before the letter N?". If the answer is 'yes', then half of the letters are eliminated and half remain as possibilities. If the answer is 'no', then the other half are eliminated and half remain. The 13 remaining letters can be divided again, and again eliminating half of the letters with each 'yes' or 'no' answer until only one letter remains. So, at most 5 answers are needed to guess the letter.

```
           ABCDEFGHIJKLMNOPQRSTUVWXYZ
Guess < N: ABCDEFGHIJKLM
Guess < G:       GHIJKLM
Guess < J:       GHI
Guess < H:        HI
Guess < I:        H
```

These relationships can be represented by what is called a binary tree. The 'root' of the tree is at the top. From each node in the tree there are

two possible ways to go further down the tree. Each time the tree splits this represents a yes/no or left/right, or whatever the *binary* difference may be which is used to exclude the other branches of the tree. Looking at this figure it can be seen that any letter can be guessed in either 4 or 5 guesses.



This kind of strategy seems to rely on a way to compare the letters to see if they come 'before' or 'after' another letter in order to properly answer the question "does the letter come before the letter N?". But we could have instead asked "is the letter among one of the letters A, B, C, D, E, F, G, H, I, J, K, L, or M?" This may seem like cheating, but this still qualifies as a single yes or no question with a single response.

What we are doing is eliminating groups of letters at a time, and we are free to eliminate any number at a time. That is why this game works for a much broader set of objects than just those that can be ordered like letters or numbers. In the game, questions can be things like "is it a person?" or "is it made of metal?".

The real reason these kind of questions work is that every object has some defining properties that can be used to differentiate it from all other objects, and the premise of the game is that any property can be reduced to a 'yes' or 'no' question. We can think of each object as either having a property, or not having it, which gives the answer to any possible question. How we differentiate things from each other is the essence of information. If we can't tell any difference, then there is no information.

1.2 Communication

Information is defined by being able to rule out alternatives. A statement contains no information in itself unless we also know how the statement could

have been different. Saying that it is raining outside only contains information because we have alternatives to it raining, such as a clear sky or snowing. The alternatives don't have to necessarily be logical or possible in reality.

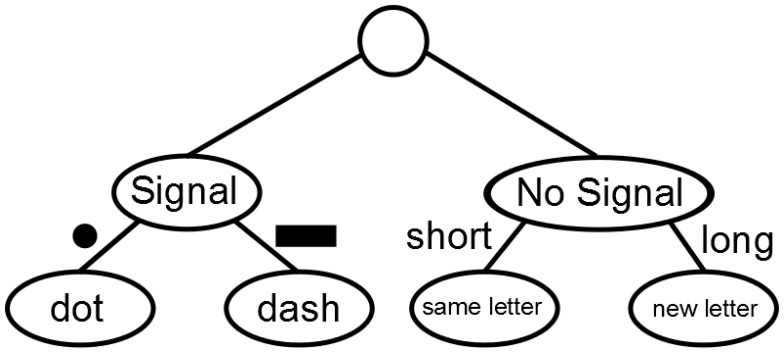
For instance, we can say that the formula of water is H_2O. This contains information even though we do not think that water actually could have another formula. This is to differentiate the formula of water from some other substance, for example ethanol which is C_2H_6O. It also does not rule out some other substance having the same formula. For instance diethyl ether is also C_2H_6O, but it has a different structure which gives it different chemical properties. More information is needed to differentiate chemicals with the same formula.

To explore the idea of information we will look at forms of communication: how to send information from one place to another. There are several forms of communication based only on the ability to send what is called a binary signal. The word binary basically means a set of two, or a pair, like in the word bicycle for two wheels. Here a binary signal means a signal that only has two possible states. When we receive one state, the only information we have gained is that we didn't get the other state.

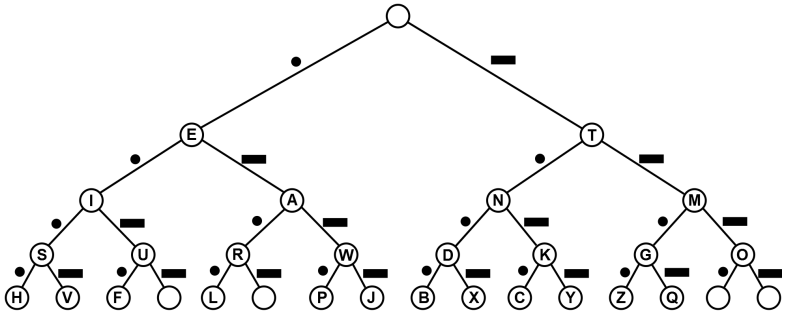
Examples of this might be a smoke signal, or a light, or an electrical signal that either on or off. All we can tell is if we can see the signal, or we can't. This is exactly like a 'yes' or 'no'. So, we know we could receive answers to play the 20 questions game by this method, but how could we ask the questions if all we can do is say 'yes' or 'no'?

In this case, we seem to need so make some per-arrangements. Suppose ahead of time we worked out a question that had 'yes' or 'no' answer, so that when we are sending the signal they know what question we're answering. This doesn't seem very useful because we would need a way to change which question we're answering for each signal.

Since we have the ability to turn the signal on and off over time, we could make longer or shorter signals to mean different things. A short signal could be a 'yes' to one question, and a long signal could be a 'yes' to a different question. Similarly we could think of short or long pauses without a signal as being a 'no' to different questions.



In Morse code, a sequence of short and long signals are used to specify a letter of the alphabet. Short signals (dots) and long signals (dashes) tells us which way to go in the tree of possible letters, and rules out the alternatives, with short pauses in between each signal. A long pause is used to indicate that a new letter is being started.

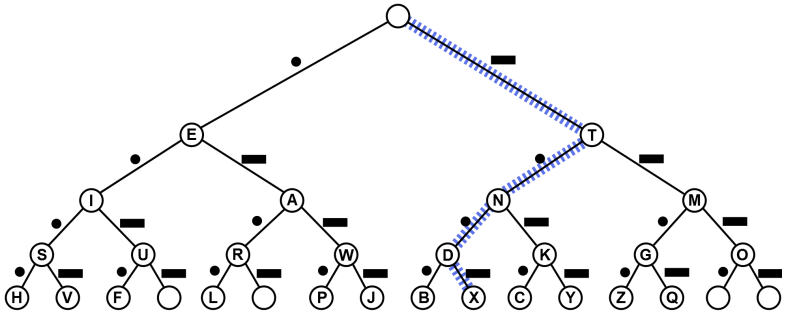


For example, suppose the first thing we receive is a short signal. In the above tree, you would start at the very top circle, called a node of the tree. If the first signal is a dot (short), then the letter is somewhere in the left side of the tree (E, I, A, S, U, R, W, H, V, F, L, P, or J). If the first signal is a dash, then it must be in the right side of the tree.

The next thing that will happen is to have either a short pause or a long pause. If it is a short pause, then we know we need more signals to know what letter it is. If it is a long pause then we have all the signals and we know which letter it is. So, if we receive a short signal and then a long pause, then that means the first letter is an 'E': •.

The letter 'X' is in the right hand side of the tree, so the first signal will be a dash. From there it is two dots and then another dash to get to the 'X' branch of the tree: ▬ • • ▬. Notice that any letter can be specified by as little as 1 signal, and no more than 4 signals. It seems like this requires

fewer guesses than in the 20 questions game, but we are getting additional information from the pauses between the signals. We can use the pauses to have different number of signals mean different letters, and so we don't need as many as if we used a fixed number of signals for every letter.



A word is formed by ending a letter with an even longer pause, and then continuing with the next letter. To form the word cat, we first need a 'c' (▬•▬•) then an 'a' (•▬) then a 't' (▬). Putting the letters together with pauses looks like.

▬•▬• •▬ ▬

Going to the next word is a longer pause. Try decoding the following message: •• ▬• ▬ • •▬ ▬•▬• •••• •• ▬• ▬▬• ▬▬▬ ▬ •••• • •▬• ••• •▬▬ • ▬ • •▬ ▬•▬• •••• ▬▬▬ ••▬ •▬• ••• • •▬•• •••▬ • •••

Feel free to come up with your own messages and try sending it to someone over a distance using a two-state signal such as a flashlight.

1.3 Digital Signals

Morse code is an example of a digital signal, but the idea can be generalized to send much more information than letters of the alphabet. Morse code was created at a time when the encoding, transmission, and decoding had to be done manually, and was limited by human speed. Modern technology is able to do these functions much more rapidly, and uses digital signals to send text, audio, video, and other forms of information for cellphones, television, and the internet. So, how does all that information get broken down, if not with dots and dashes?

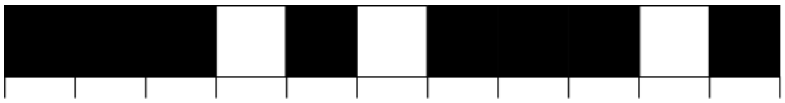
The way this is done is very similar to how sheet music is written. In music, individual notes are supposed to played for a specific amount of time, which is regulated by a beat frequency. Music can be played faster or slower by making the beats shorter or longer, but the relation between how long each note lasts is fixed by the sheet music.

In a digital signal, if we know the beat frequency all we need to do is look for *changes* in the signal. Even if the signal does not change we can gain information because it *could* have changed in the same way that every beat *could* be a different note.

We already used this kind of scheme with the Morse code by having long versus short signals, and we can be specific about how many beats each element lasts:

Morse code	**# of beats**
dot	1
dash	3
pause between elements	1
pause between letters	3
pause between words	7

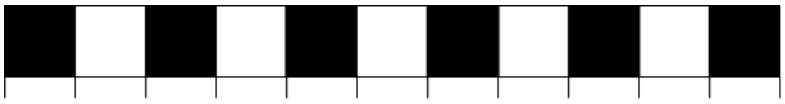
Using these rules we could break up the letter 'C' into 11 individual beats from the code ▬ ● ▬ ●. In the following image each box represents one beat. If the box is filled in that represents a signal, and if it is empty then there is no signal.



If we loosen the restrictions on how long each signal or pause has to last, allowing them to last any number of beats, then the total number of different possible sequences is much larger for a given number of beats. If any beat can have either value, then it is called a bit. Bit is short of binary digit. Since this communication is broken down into individual binary digits we call it digital.

Each bit has two possible values, and so doubles the number of total possibilities. Two bits have four possible ways they could happen. Three bits have 8 ways. Eleven bits has $2^{11} = 2048$ different ways, and so on.

I could set the signals with a regular pattern of bits, perhaps to tell the receiver what the beat frequency is.



Or I could set the bits randomly, such as with the flips of a coin.

A digital signal is an abstract idea, and so can be implemented by different hardware using different mediums, and converted from one to another. When you load a web page, that stream of bits originated as a file on a hard-drive which was probably stored by magnetic fields on a metal disk. Then converted to electrical pulses when loaded into the servers memory where it was stored by transistors and capacitors.

It's then converted again and sent into the network electronically over wires to a service provider. It may then be converted into pulses of light which travel through optical fibers until it reaches another large service provider. Or it is converted to radio waves and transmitted to a satellite which does additional conversion from electrical to radio to send it to another part of the planet. Finally the process is complete bringing it into your computers memory where it is once again stored by little capacitors. Your computer then converts the bits into images and sounds through your browser so that you can experience it on your monitor or speakers.

However, we don't have to worry about *how* the information is physically being stored or transmitted. The low level hardware is abstracted away so that we only have to deal with the idea of bits.

1.4 Binary Numbers

The binary number system can be analyzed in several different ways. In one view, the concept of a particular number exists apart from the way we write it, and decimal versus binary are just two ways of doing it. The number 5 is a concept apart from the symbol '5'. So is the number '6'. But just like with letters, we need a way to tell which number we're talking about. One way to do it is to have a separate symbol for every number, but that is not practical.

There is no single way to represent a number, and in fact Morse code has its own way of representing numbers. But we will use a standard representation that closely resembles the decimal system that is useful for doing arithmetic. Instead of basing a digit on a power of ten ($1 = 10^0, 10 = 10^0, 100 = 10^2, 1,000 = 10^3$...), we base it on a power of 2 ($1 = 2^0, 2 = 2^1, 4 = 2^2, 8 = 2^3, 16 = 2^4$...). In binary there are only two symbols: zero (0) and one (1), and represent the binary digits (bits). This is also called base-2, as opposed to base-10, since the 'base' of the exponent used is 2.

power	decimal	binary
2^0	1	1
2^1	2	10
2^2	4	100
2^3	8	1000
2^4	16	10000
2^5	32	100000
2^6	64	1000000

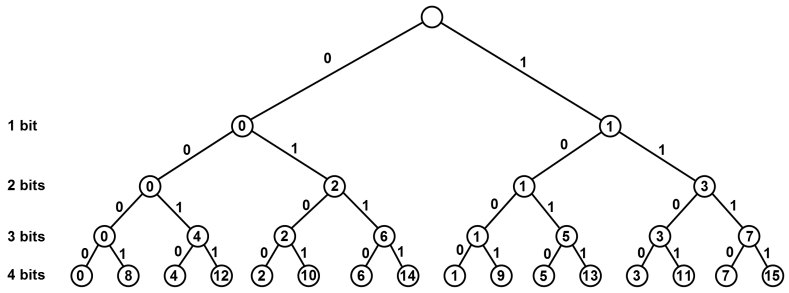
Like decimal numbers, the least significant digit (the 1's place) starts on the right, and more significant digits go toward the left. To write the number 2, a 1 is put in the 2's place, and a 0 in the 1's place: $2_{(10)} = 10_{(2)}$. I am also using the subscript (10) and (2) to separate base 10 from base 2 numbers. The number 2 looks like 10 in binary because the 1 is in the 2's place, and every zero to the right represents a power of 2, like the zeros represent powers of 10 in the decimal system.

Converting a number from binary is fairly straight forward as long as you know the numeric value of each place. If there is a 1 in that place then you add it to the total, and if there is a zero then you don't. For example, the number 1011 has a 1 in the 1s, 2s, and 8s place so we add them together to get 1+2+8 = 11. The number 1111 has all four which gives 1+2+4+8 = 15. And notice that adding them all together is 1 less than the value of the next bit, which is 16.

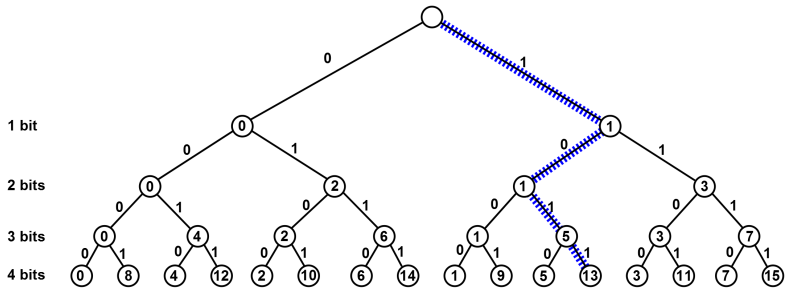
Converting a number to binary starts with the most significant bit that has a value less than the number you want to write in binary. To write the number 37, we start by finding the largest power of two that is less than 37. 64 is too big, so we start with 32. That means the 32s bit is set to one. The remainder is 5 ($37 - 32$), and so we have to set the next biggest bit that is smaller than 5. 16 and 8 are too big so the next bit is in the 4s place, which

leaves a remainder of 1 $(5 - 4)$. 2 is too big so it has to go in the 1s place (which makes sense). Writing the bits out with zeros for the bits that didn't get used gives: 100101. Try converting it back to decimal to verify it gives the correct number.

Like with Morse code letters, we can construct each number based on a yes/no binary method. If we start with the least significant bit, it has two possible values: 0 or 1. Once that bit is determined by moving down one branch of the tree, the second bit has two possible values. Every additional bit has two possible values, doubling the possible number of numbers.



You may notice that there are not as many numbers in this tree as there are letters in the Morse code tree. That is because binary numbers are not designed to take advantage of a variable length number of bits. Adding another zero most significant bit does not change the number like adding additional dots did for letters. 0100 is the same number as 100. Converting to/from binary can be thought of as traversing the tree to the number in question. Remember that the top of the tree is the *least* significant bit. So, 1101 leads to the number 13.



While individual bits carry information, one can see that quite a few bits are required to represent anything useful for us humans. A byte is a short sequence of bits, and is the smallest 'chunk' of information a computer tends deal with at one time. It is currently widely accepted to mean exactly 8 bits,

although historically the size of a byte has varied.

8 bits have a total number of $2^8 = 256$ possible values. We can take a continuous stream of bits and group every multiple of 8 together into a continuous stream of bytes. This makes using the stream of information a little easier since each 'chunk' can differentiate between 256 possibilities, instead of just 2.

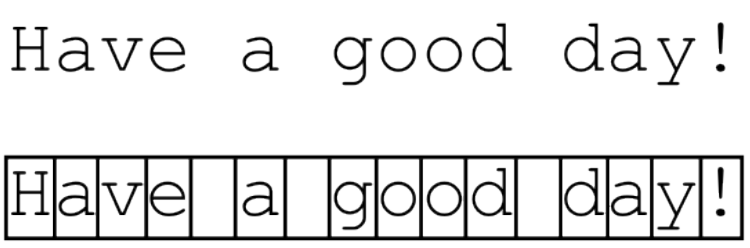
If we are limited to 1 byte, then a binary number has to have exactly 8 digits. The number 0 would be 00000000, the number 1 would be 00000001, and so on. The most significant bit has a value of $2^7 = 128$, and so the byte 10000000 in binary is equivalent to the number 128. If all the bits are set, 11111111 has a value of $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 255$. So a byte can have integer values from 0 up to 255.

Bigger numbers of course require more bytes to be represented. We will deal with how to represent negative numbers and fractions in a later chapter. However, the basic idea is that we are limited by how many possibilities can be represented.

1.5 Digital Text

The basic idea of storing or sending digital text comes straight from how type-writers functioned. A piece of text is broken up in to a sequence of individual blocks. A page is broken into a sequence of lines. And each line is broken in to a sequence of characters.

The available characters must be pre-determined before the text can be compiled. For a type-writer, this meant creating a block for each character. This was efficient because the same characters could be reused to form different words and pages.



For a digital text, the available characters are determined first. Basically, a number, or code, is assigned for each separate character we wish to be able to print. Capital letters are separate characters than lower case. A space is a character. More characters are needed for punctuation and other symbols like $, &, and any other symbol we wish to be able to represent.

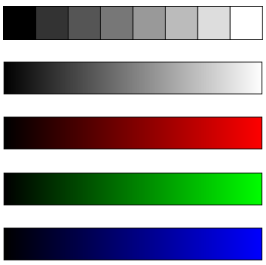
We don't want all the text to be on the same line, and so there needs to be a way to tell where the text appears in the vertical direction. Just as there is a character to represent a space, which basically means moving over a certain amount without printing anything, there is a character to represent the action of going to a new line of text without printing anything. This is called the new-line character. Even though we can't 'see' the newline character on a page of text, just like the space character, we can its result. A new-line is usually what happens when you hit 'enter' on your keyboard when typing.

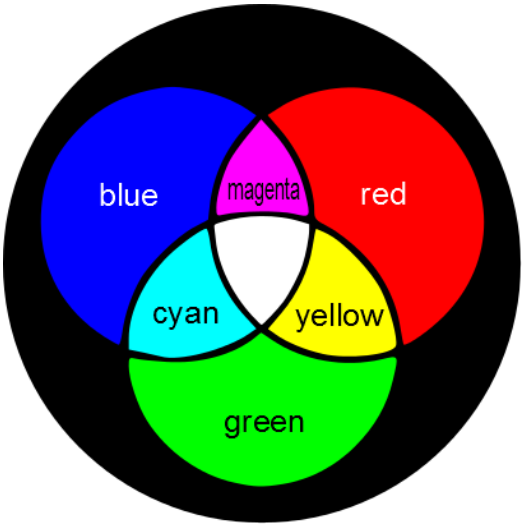
Now what numbers, specifically, represent each character? For example, what might be the number for the letter 'A'? Well, it really depends. In some sense the numbers chosen are arbitrary. However, there are certain standards that have been established.

If you wish to represent a single character by 8 bits (an 8 bit number), then you would probably use the ASCII standard. In ASCII, the letter 'A' is number 65 (in decimal). The letter 'a' is 97. A space is 32, and a new line is 10. Since there are 8 bits to work with, there are 256 possible characters, although not all the numbers have a particular meaning specified.

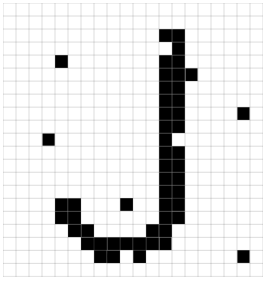
The ASCII standard is somewhat English centric. Many languages require additional characters. The Unicode standards are a way to add all the possible characters needed for any language to be represented, and itself varies in how many bits are required to represent individual characters. There is an 8-bit version which is slowly replacing ASCII.

1.6 Digital Audio and Visual





1.7 Error Correction



1.8 Encryption

1.9 Exercises

1. A single flashlight has two possible states that can be perceived by a person at a distance (on and off). How many different states could be perceived using two flashlights? Assume the flashlights are identical. If they are held side-by-side, could someone tell which one was on, when only one of them is on? If someone is far enough away, will the two flashlights appear as a single flashlight when they are both on?

2. Suppose a red filter is added to one of the flashlights, and a blue filter is added to the other one. How will having two different colors change how many possible states can be perceived by a person at a distance away? If the two lights look like a single light from far away, what color will they appear when both flashlights are on?

3. Suppose we have a 4 state transmission system labeled by the following digits: 0 (off), 1 (on), 2 (on), 3 (on). Come up with a code that uses the off state to separate signals, the letters of the alphabet are specified by a sequence of digits, and a space is also a sequence of digits. What is the minimum sequence of digits needed to send any of the 26 letters or a space?

Chapter 2

Control Structures

To make our code capable representing any computation, certain control structures are required. We need a way of running a particular command conditionally on some other information. And we need a way of executing the same code more than once.

2.1 if-else

Conditional execution in JavaScript is implemented using the if-else structure, or just called an if-statement. To use an if-statement I first need a quantity that reduces to a boolean value: either true or false. This value becomes the condition under which a particular piece of code is executed if it is true. The condition is placed between the set of open and closed parenthesis. Immediately following the parenthesis is the piece of code that will be executed if the condition is true. The code is grouped together using an open-closed set of curly brackets.

```
1 var a = true;
2 var b = false;
3
4 if (a) {
5     console.log("This will be printed.");
6 }
7
8
9 if (b) {
10     console.log("This will not be printed.");
11 }
```

I can also specify that something is done if the value is *not* true, which will fall under the `else` part. Not true is the same as saying false.

```
if (a) {
    // nothing done here
}else{
    console.log("This will not be printed.");
}


if (b) {
    // nothing done here
}else{
    console.log("This will be printed.");
}
```

Even though the above example is completely valid, it can be simplified since there are no operations done in the top part of the if-else. I can use the logical 'not', `!`, to negate the boolean values. True becomes false, and false becomes true. This flips the roles of the if-else blocks, and so I can get rid of the 'else' part, while keeping the same logic. That is, this version of the control structure is equivalent to the previous one.

```
if (!a) {
    console.log("This will not be printed.");
}


if (!b) {
    console.log("This will be printed.");
}
```

In the following example, I am testing to see if a particular number is even or odd, and I want it to print something different in each case. To see if a number is even, I used the modulus operator. If the number divided by 2 has a zero remainder, then the modulus will return zero, and I know that 2 is a factor of the number. The `===` operator returns true if `myNumber % 2` is zero, and false if it is not zero. You should always give an if-statement a boolean value, and so comparison operators can be used to convert numbers to booleans. The comparison has the lowest order of operations, and so it will be done last.

```
var myNumber = 321;

if (myNumber % 2 === 0)
    console.log(myNumber + " is even.");
else
    console.log(myNumber + " is odd.");
```

All of the operations together, `myNumber % 2 === 0`, becomes the condition of the if-statement. Only the first thing following the condition is associated with the 'true' case. The piece of code immediately following the else is not executed if the condition is true, but it is if the condition is false. If you wish to have multiple commands associated with the condition, curly braces are used to group the code together.

```
1 var myNumber = 321;
2 var halfInteger;
3
4 if (myNumber % 2 === 0) {
5     console.log(myNumber + " is even.");
6     halfInteger = myNumber/2;
7 } else {
8     console.log(myNumber + " is odd.");
9     halfInteger = (myNumber - 1)/2;
10 }
11
12 console.log(halfInteger + " is half integer.");
```

Here I not only print out whether the number is even, but divide it by two if it is even. If it is odd, I subtract 1 from the number before dividing so that the result is still an integer. If I had not put curly braces around each set of code, then the interpreter would not have understood that I wanted the second statement in each block to only be executed for that condition.

2.2 nested if-else

Any code can be placed in an if-statement, including more if-statements. A common problem is that a variable could have several possible values, and something different should happen for each value.

```
1 var first = "hello";
2 var second;
3
4 if (first === "foo") {
5
6     second = "bar";
7
8 }else if (first === "hello") {
9
10     second = "world";
11
```

```
}else{
    second = "idk";
}

console.log(first + " " + second);
```

The first if-statement checks for one value. If it is that value it does the thing it should for it. If it's not that value, it goes to the `else` part. The only thing in the else of the first if-statement is another if-statement that checks for another value. If it's the second value it does that thing, but if it's not it goes to the else of the second if-statement.

However, any structure of nested if-else statements is valid.

```
var clouds = true; // is it cloudy outside
var rain = false; // is in raining outside
var daylight = true; // is it during the daytime
var lights = false; // are the lights on in the house

var activity;

if (daylight) {

    if (clouds) {

        if (rain) {
            activity = "play video games.";
        }else{
            activity = "play basketball.";
        }

    }else{
        activity = "go swimming.";
    }
}else{

    if (lights) {
        activity = "work on homework.";
    }else{
        activity = "go to sleep.";
    }
}

console.log("I think I'm going to " + activity);
```

What will be printed above given the values of `daylight`, `clouds`, and `rain`? Does the value of `lights` affect what's printed, or when would it?

What would the values have to be in order for them to decide to "go swimming"?

Certain logic can sometimes be expressed in more that one way. Boolean algebra can be used to rewrite if-else structures into a series of equivalent if-statements using the logical operators.

```
var clouds = true; // is it cloudy outside
var rain = false; // is in raining outside
var daylight = true; // is it during the daytime
var lights = false; // are the lights on in the house

var activity;

if (daylight && clouds && rain) {

    activity = "play video games.";

}else if (daylight && clouds && !rain) {

    activity = "play basketball.";

}else if (daylight && !clouds) {

    activity = "go swimming.";

}else if (!daylight && lights) {

    activity = "work on homework.";

}else if (!daylight && !lights) {

    activity = "go to sleep.";
}

console.log("I think I'm going to " + activity);
```

While this is logically equivalent to the first structure, it checks and rechecks the same values at each if-statement. It is also harder to see which scenarios, or cases, have a prescribed activity. And what if a `!` was accidentally omitted? While the choice of which types of structures to use is up to you, your choice should be guided by both the purpose of the structure, and how easy it is to understand and debug if it doesn't work as expected.

Let's look at exactly how an if-else can be re-written with logical operators, or vice-versa. A single 'and' operation can be thought of as a nested

if-else statement where both conditions have to be true for the result to be true. If either condition is false (or both), the result is false. The two if-else structures in the following example are equivalent.

```
var a = true;
var b = false;
var result;


if (a) {

    if (b) {
        result = true;
    }else{
        result = false;
    }
}else{
    result = false;
}

if (a && b) {
    result = true;
}else{
    result = false;
}
```

For an 'or' operation, if at least one of the conditions is true (or both), then the result is true. This can be represented by a series of if-else statements. The only time the result is false is if both conditions are false.

```
var a = true;
var b = false;
var result;

if (a) {
    result = true;
}else if (b) {
    result = true;
}else {
    result = false;
}

if (a || b) {
    result = true;
}else{
    result = false;
}
```

These kind of examples also help to explain what is called short-circuit behavior in boolean operators. Take a look back at the 'and' example. If the value of `a` was false, then the first if-else structure would never even check the value `b` because it would have immediately gone to the else part. The same is true with the `&&` operator. If `a` is false, then the value of `b` doesn't matter and so it doesn't check it. For the `||` operator, if the first value is true, then the value of the second doesn't matter since only one of them has to be true for the result to be true.

It may seem like we don't need to worry about short circuit behavior as programmers, since why should we care if it checks a value or not. For the most part it doesn't matter for writing programs. However, if you make a function call in the condition of an if-statement, short-circuit behavior could matter if that function call does something in addition to being a condition.

2.3 while

The iteration control structure has to be tied to a conditional structure, so that it knows when to stop. A `while` loop initially behaves similarly to an if-statement. The condition is placed in the parenthesis immediately following the `while` keyword. If the condition is false, then execution skips the code in the braces. If the condition is true, it executes the code in the braces, but after it's done it re-evaluates the condition. It continues to execute the code in the braces, and evaluates the condition, until the condition is false.

```
1 var x = 0;
2
3 while(x < 5) {
4     x++;
5 }
```

In the above example, the variable `x` is called the loop control variable. Its initial value of `0` causes the condition `x < 5` to be true, and so it then executes the code in the braces. The only thing it does is to add 1 to the variable. It then checks the condition again, which is still true. It keeps repeating this process adding 1 to `x` each time, until it gets to 5. When `x` is 5, the condition becomes false since 5 is not less than 5. The loop executed the code a total of 5 times before it stopped.

A loop will continue to repeat itself for as long as the condition is true. If nothing occurs to cause the condition to be false, the loop will run for as

long as the program is capable: this is called an infinite loop.

```
var x = 0;
var y = 0;

while(x < 5) {
    y++;
}
```

The above example will run forever, even though it might look very similar to the first example. The issue is that the variable being changed inside the loop is different than the variable being used in the condition. `x < 5` will always be true, and thus it will loop forever.

A while loop does not necessarily have to proceed in a linear fashion. How many times does the following loop repeat, and what is the final value of `x`?

```
var x = 1;

while(x < 32) {
    x *= 2;
}
```

Consider the following loop as a brain teaser.

```
var x = 1;
var divisor = 1;

while(x < 5) {

    if (x % divisor === 0){
        x /= divisor;
        divisor++;
    }

    x++;
}
```

Does this loop ever stop? If so, how many times does it loop? Just by looking at the terminating condition, and assuming it does stop eventually, what must be the final value of `x`?

2.4 for

The 'for' loop is a shorter way of writing a linear while loop. If I know ahead of time exactly how many times I want the loop to repeat, there is a regular pattern I can follow. First, create a variable that starts at zero. The loop condition is as long as that variable is less than the number of times I want it to loop. And after each loop I add 1 to the variable.

```
1 for(var x = 0; x < 5; x++) {
2     console.log(x);
3 }
```

When you write a for loop, the stuff inside the parenthesis is not just the loop condition. It also contains the initial value of the loop control variable, and the increment operation done after each time through the loop. In every other way this for loop works exactly the same as the while loop. It loops exactly 5 times.

Now, what numbers get printed to the console? The first time through the loop the value of `x` is 0, so 0 gets printed. The 1, 2, 3, and 4. After it prints the 4 the increment `x++` adds 1 to x, making it 5. But `5 < 5` is false, and so the loop doesn't repeat. Even though the final value is 5, it never gets printed.

Index