PENN SAAST

Introduction to Hardware and Software Algorithms

© Ravi K. Chugh

rchugh@cs.ucsd.edu

July 2007

Contents

Chapter 0

Introduction

Here's an exercise you should try with a few classmates. Each of you write down on a piece of paper a recipe for making a peanut butter and jelly sandwich. After everyone is done, trade recipes and have each person try to make a sandwich by following only the directions that are provided in the recipe. Good luck!

What is an algorithm? You can think of it as a recipe. A recipe is a finite set of instructions that tell the reader step-by-step how to accomplish something. The recipe you just wrote is an algorithm intended for a human to read and follow. When asked to following someone else's recipe step-by-step, word for word, you probably found that there were some omitted details that prevented you from actually being able to make a PB&J sandwich. If you were allowed to interpret the directions and add some of your prior knowledge and experience, you probably would have been able to make the sandwich just fine. When designing an algorithm, or a recipe, for a computer to follow, however, each and every step needs to be explicitly provided. A computer is simply a machine that does what it is told to do in the way of a computer program. So the art/science of computer programming is expressing an algorithm to perform some computation in such a way that a computer can follow your directions, step by step.

As you probably saw, there are usually multiple recipes that achieve the same goal. There are many different ways of making a PB&J sandwich, so what makes them different? What makes one recipe better than another? How quick they take to follow? How readable the instructions are? The minimum amount of mess they create? These are all subjective measures, but given multiple ways of doing something like this, we can reason about which methods are preferable and which are not. For example, if one recipe had you do 10 jumping jacks in between each step, you would think it was a bad recipe. Although it may get the job done, there would be a lot of unnecessary steps along the way. On the other hand, if one algorithm was very concise but had you juggling many items at the same time, you might also think it was a bad recipe because it was hard to follow, even though the resulting sandwich may have been perfectly fine or even better-tasting than normal. These examples highlight two ways that we can subjectively analyze algorithms: the language they use and how long it takes (however "long" is defined) to follow them.

These two metrics are relevant to computer programs, as well. There are many different programming languages, and most algorithms you write can be implemented in many different languages. There are different features in different types of languages, so some algorithms may be easier to express in some rather than others. You may find some languages use more pleasing syntax and are easy to use, while some aren't as friendly to readers.

The other metric is how long a computer program takes to execute. How do we define this precisely? How many instructions the computer must execute? How many seconds it takes to finish? These measurements are not completely objective, however, because different computers have different hardware specs that can skew these measurements, leading to unfair comparisions. Therefore, we use precise mathemetical definitions for the notion of how long an algorithm takes to run that abstract away details like what kind of computer processor it is running on. The fields of algorithms and computational complexity have explored many topics in this area, and a goal of ours will be to learn the basics for how to analyze an algorithm's effectiveness. This skill is useful for computer programmers, who often have to make algorithmic choices when writing programs. These choices can have dramatic effects on the result of an algorithm, so it is important to understand the complexity of the programs, or algorithms, you write.

We won't dive into the formal study of algorithms yet for two reasons: 1) you need to have an understanding of some fundamental programming concepts before it makes sense to talking about analyzing their effectiveness; and 2) talking about writing programs for computers in nonsense without having computers to program. The latter is largely the study of computer architecture and engineering, but it is useful for programmers to have at least a basic understanding of how data is represented by the underlying machine. Some of the details of the hardware have an effect on decisions you make as your program, so now is as a good time as any to learn a little about computer architecture. We will only scratch the surface of topics relating to computer hardware, but it will be enough to give us a better sense of what we are doing as programmers and hopefully excite interest in these fields. Once we have this under our belt, along with the skills to write programs in Java, we will begin learning how to analyze algorithms.

Chapter 1

There are only 10 types of people

The first is those who know binary, and the second is those who don't.

Unlike humans, computers store numbers in binary representation. Actually, we think of computers as understanding binary (do computers really understand anything?), because it is a convenient representation for what is going on with the physics of a computer at a low level.

Our system for representing numbers is called the *decimal*, or base-10, system because we use 10 different symbols: 0 through 9. Every number can be uniquely encoded by a string of these 10 digits. If we have a string of decimal digits $a = a_n ... a_1 a_0$, a represents the number $\sum_{i=0}^{n} a_i * 10^i$. As an example, the string 934 represents $4 * 10^0 + 3 * 10^1 + 9 * 10^2$. We say the rightmost digit of a is the least significant digit and the leftmost is the most significant.

There is nothing special about the symbols 0 through 9 nor the fact that there are 10 of these symbols. We can just as well use strings of only the symbols 0 and 1 to uniquely represent every number. This is the definition of *binary*, or base-2, representation, and the encoding is symmetric to the decimal encoding: a string of binary digits $b = b_n ... b_1 b_0$ represents the number $\sum_{i=0}^{n} b_i * 2^i$. For example, the binary string $1011 = 1 * 2^0 + 1 * 2^1 + 0 * 2^2 + 1 * 2^3$ represents the number 11. Although using base-2 representations of numbers is not appealing for everyday use (the encodings are much longer because there are only 2 symbols), we can use them to perform all arithmetic operations as usual, just as we can with any base representation.

A computer processor is an intricate digital circuit comprising things like wires and millions of transistors. At any given time, each transistor can either block electric current from passing through the wire or let it through. We can think of the transistor as being in state 0 in the former case and state 1 in the latter. These two possible physical states of a transistor lead to the portrayal of computers as using a binary representation of numbers, since all computation at the machine level really is just 1's (passing voltage along wires) and 0's (or not).

In a separate lesson we will see some of the basics of how a computer processor can perform binary arithmetic. For now we will spend some time getting comfortable with binary representation. It will be useful to be able to derive, and eventually recognize, the binary encodings:

0 = 0000	4 = 0100	8 = 1000	12 = 1100
1 = 0001	5 = 0101	9 = 1001	13 = 1101
2 = 0010	6 = 0110	10 = 1010	14 = 1110
3 = 0011	7 = 0111	11 = 1011	15 = 1111

Note that, just as with decimal representations, leading zeroes do not affect the value of the binary representations.

It will also be useful to get familiar with some of the powers of 2:

$2^0 = 1$	$2^4 = 16$	$2^8 = 256$
$2^1 = 2$	$2^5 = 32$	$2^9 = 512$
$2^2 = 4$	$2^6 = 64$	$2^{10} = 1024$
$2^3 = 8$	$2^7 = 128$	

In a binary string, each digit is referred to as a *bit* (short for binary digit). A collection of eight bits is known as a *byte*. An n-bit string can represent (at most) 2^n different numbers, since each of the n bits can take on either of two values. (We'll get to the disclaimer in a bit.)

To add two binary numbers, we work from right to left just as we do with decimal numbers adding each pair of digits and propagating any carry. For example, 0 + 1 sums to 1 with no carry, and 1 + 1 sums to 0 with a carry of 1. Two examples:

$$\begin{array}{cr} & 0101 \\ + & 1001 \\ \hline & 1110 \end{array} \qquad \begin{array}{cr} & 1110 \\ + & 0101 \\ \hline & 10011 \end{array}$$

Notice that with any base representation, summing two numbers can result in a number that requires one more digit than either of the two summands.

Converting from binary to decimal is straightforward, but the other direction takes a little more work. Starting with a decimal number, you repeatedly subtract the largest power of 2 from it without letting it become negative. For each power 2^i, the binary representation has a 1 in the ith place if 2^i was subtracted and 0 if it was not. An example:

$$\begin{array}{cr|l}
 & 155 & \\
- & 128 & (2^7 \text{ yes}) \\ \hline
 & 27 & \\
 & & (2^6 \text{ no}) \\
 & & (2^5 \text{ no}) \\
- & 16 & (2^4 \text{ yes}) \\ \hline
 & 11 & \\
- & 8 & (2^3 \text{ yes}) \\ \hline
 & 3 & \\
 & & (2^2 \text{ no}) \\
- & 2 & (2^1 \text{ yes}) \\ \hline
 & 1 & \\
- & 1 & (2^0 \text{ yes}) \\ \hline
 & 0 &
\end{array}$$

By looking back at which powers of 2 were subtracted, we can read off that the binary representation of 155 is 1001 1011.

The binary encoding scheme above (formalized by the summation notation) does not take into account negative numbers; it starts at 0 and proceeds increasingly forever. This encoding scheme is known as *unsigned magnitude*.

A simple change to this scheme allows for negative numbers: treat the most significant bit (MSB) as the sign bit, and the remaining bits as the usual unsigned magnitude encoding. If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative. This encoding is known as *signed magnitude*. With this representation, adding leading zeroes *does* change the value being represented (if the original value is a negative number).

Now that negative numbers can be represented, we can define subtraction on binary strings just as we do for decimal strings. (Subtraction isn't well-defined in the unsigned magnitude scheme, because if the second operand is larger than the first, there is no way to represent the resulting difference.)

As it turns out, a hardware circuit to perform addition on signed magnitude numbers is a little complicated. Fortunately, there is a clever representation called *2's complement* that allows the hardware circuit to easily compute sums of positive and negative numbers.

In 2's complement, positive numbers are represented as usual. A negative number is represented by starting with the positive representation, bitwise-complementing each bit, and then adding 1. To go from a negative number to its (positive) magnitude, do the same thing: complement each bit and then add 1. For example, to negate 5 (0101), first invert each bit (1010), and then add 1 (1011). We ellide the details for why this works, but try several examples to convince yourself that it does.

A positive unsigned magnitude representation effectively has an infinite number of leading zeroes. Thus, when we have a negative 2's complement number, it is crucial that we treat it as having an infinite number of leading ones. Note that for a 2's complement number, if the MSB is 0, the number is positive; if the MSB is 1, the number is negative.

The beauty of this representation is that adding 2's complement numbers works just like adding unsigned magnitude numbers regardless of the signs of the summands. Two examples:

$$\begin{array}{rll} & 0111 & (7) \\ + & \underline{1011} & (-5) \\ & 0010 & (2) \end{array} \qquad \begin{array}{rll} & 1111 & (-1) \\ + & \underline{1011} & (-5) \\ & 1010 & (-6) \end{array}$$

Note that in both examples, the carryovers did not fit in four bits and were just discarded. Ignoring the carryover is correct with 2's complement because of the implicit infinite zeroes before positive numbers and implicit infinite ones before negative numbers. In the case where more bits are needed to represent a 2's complement number, the appropriate leading zeroes or ones need to fill all the new higher order bits. This process is called *sign extension*. For example, in going from four bits to eight bits, the representation of 5 goes from 0101 to 0000 0101, and the representation of -5 goes from 1011 to 1111 1011.

Now that we've estabished unsigned magnitude, signed magnitude, and 2's complement as three binary encoding schemes, let's examine the range of numbers that can be represented by an n-bit string in each. For an n-bit unsigned magnitude string, we've already seen that 2^n numbers can be represented (0 through $2^n - 1$). The same cannot be said for signed magnitude, because there ends up being two different representations for zero (0000 and 1000), since the sign does not affect the value of zero. Because of this, an n-bit signed magnitude string can represent only $2^n - 1$ different numbers. The first bit is reserved for the sign and the remaining $n-1$ bits are used for magnitude, so the range is $-(2^{n-1}-1)$ to $2^{n-1}-1$.

With 2's complement, zero no longer has two representations (0000 is 0 and 1000 is -8), so an n-bit 2's complement number can represent 2^n different numbers, ranging from -2^{n-1} to $2^{n-1}-1$. The asymmetry of this range leads to one case where the conversion from negative to positive doesn't work per the recipe: for the smallest negative number that can be represented. For example, a 4-bit encoding for -8 (1000) returns back -8 (when looking at only 4 bits). For this reason, the smallest negative number that can be represented by an n-bit 2's complement encoding is sometimes called the *weird number*.

To compare, we'll summarize the meanings for each of the 16 different 4-bit strings with each representation scheme (UM, SM, and TC):

0000 = (0, 0, 0)	1000 = (8, 0, -8)
0001 = (1, 1, 1)	1001 = $(9, -1, -7)$
0010 = (2, 2, 2)	1010 = $(10, -2, -6)$
0011 = (3, 3, 3)	1011 = $(11, -3, -5)$
0100 = (4, 4, 4)	1100 = $(12, -4, -4)$
0101 = (5, 5, 5)	1101 = $(13, -5, -3)$
0110 = (6, 6, 6)	1110 = $(14, -6, -2)$
0111 = (7, 7, 7)	1111 = $(15, -7, -1)$

For 4-bit strings, the ranges for each scheme are: 0 to 15 (UM), -7 to 7 (SM), and -8 to 7 (TC).

Because hardware to do arithmetic on 2's complement numbers is simpler (so also cheaper and faster) than for signed magnitude, 2's complement is used. Now that we understand how 2's complement works, we can analyze the ranges of values different kinds of Java primitive types can hold, since all of Java's integer types use 2's complement representation. You can find an overview of Java's different primitive types at:
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html.

In this section, we will only work integral data types.

type	bits	range
`byte`	8	-128 to 127
`short`	16	-32768 to 32767
`int`	32	approx. -2.1×10^9 to 2.1×10^9
`long`	64	approx. -9.2×10^{18} to 9.2×10^{18}

Although Java does not, some languages, like C++ and C#, also provide unsigned versions of these types. For example, if Java provided it, an `unsigned short` would have the range 0 to 65535 (using unsigned magnitude).

It's important to understand what the ranges of different types of data types are, because exceeding the boundaries can lead to seemingly strange programming bugs. For example, what happens when you add the two bytes 0111 1111 (127) and 0000 1010 (10)? The resulting sum cannot fit in just eight 2's complement bits, so the carryover is lost, and the result is 1000 1001 (-119). This quizzical result, that $127 + 10 = -119$, is computed because the size needed to represent the number 137 *overflowed* the number of bits available for the data type.

To adhere to 2's complement representation and the fact that hardware is built with bits in powers of 2, we make a modification to the title of this section: there are only 0010 types of people.

1.1 Exercises

1. For each of the following numbers, compute the UM, SM, and TC representations with (i) 8 bits, and (ii) the minimum number of bits. Write N/A if the number cannot be represented.
   - 25
   - -43
   - -99
   - -127
   - -128
   - -129
2. For each of the following bit strings, determine what value it encodes in UM, SC, and TC:
   - 0110
   - 1000
   - 1010
   - 0100 1110
   - 1100 1110
   - 1111 1111

Chapter 2

Floating-point numbers

So far we have only considered how integral, or whole, numbers are represented in different encoding schemes. We will now examine how to encode floating-point, or real, numbers in different bases and how they are represented in computers.

In the decimal system, digits to the right of the decimal point are subject to the following encoding scheme: $a_1a_2\ldots a_n = \sum_{i=1}^{n} a_i \frac{1}{10^i}$. Note that digits to the right of the decimal point are indexed from left to right and begin at 0 instead of 1. Similary, in binary notation, bits to the right of the "binary point" follow: $b_1b_2\ldots b_n = \sum_{i=1}^{n} a_i \frac{1}{2^i}$.

To compute the binary representation of a fraction, we can iteratively multiply the equation we are trying to solve to compute one bit at a time. (Recall that in the previous section, we saw how to iteratively *divide* the equation to get the compute the representation of a whole number.) For example:

$$\begin{array}{rcll}
0.375 & = & b_1/2 + b_2/4 + b_3/8 + b_4/16 + \cdots & \text{multiply by 2} \\
0.75 & = & b_1 + b_2/2 + b_3/4 + \cdots & \text{set } b_1 = 0 \\
0.75 & = & b_2/2 + b_3/4 + \cdots & \text{multiply by 2} \\
1.5 & = & b_2 + b_3/2 + \cdots & \text{set } b_2 = 1 \\
0.5 & = & b_3/2 + b_4/4 + \cdots & \text{multiply by 2} \\
1 & = & b_3 + b_4/2 + \cdots & \text{set } b_3 = 1 \\
0 & = & b_4/2 + \cdots & \text{multiply by 2} \\
0 & = & b_4 + \cdots & \text{set } b_4 = 0
\end{array}$$

Thus, we have that the binary representation of 0.375 is 0.011.

Just as with real decimal numbers, patterns of bits to the right of the binary point may repeat forever. For example, consider the binary representation of $\frac{1}{3}$:

$$\begin{array}{rcll}
1/3 & = & b_1/2 + b_2/4 + \cdots & \\
2/3 & = & b_1 + b_2/2 + \cdots & b_1 = 0 \\
2/3 & = & b_2/2 + b_3/4 + \cdots & \\
4/3 & = & b_2 + b_3/2 + \cdots & b_2 = 1 \\
1/3 & = & b_3/2 + b_4/4 + \cdots & \\
2/3 & = & b_3 + b_4/2 + \cdots &
\end{array}$$

Notice that we have encountered this last equation already, so the subsequent steps will produce the same coefficients as before. That is, once we've arrived at the same left-hand side, we can stop because we know what the pattern is. In this case, we would write that the binary representation is $0.\overline{01}$. As it turns out, $\frac{1}{3}$ doesn't have a finite representation in either decimal or binary. It does, however, have a finite representation in base-3, or *ternary*, notation. What would it be?

As an aside, you have probably seen fractional numbers written in ternary notation without knowing it. If you are familiar with baseball, you will recognize that if a pitcher has pitched, say, six full innings and two outs in the seventh, his Innings Pitched will be written 6.2. The fractional portion of this stat is kept in ternary notation, and you've probably been decoding ternary without even knowing it! Note that the integral portion of this statistic is still written in decimal – it would be extremely confusing to report 20.2 as the number of innings pitched!

The issue now is how to encode so-called floating-point (since the binary point may appear anywhere in the string) numbers with computer hardware. Allowing the binary point to appear in arbitrary positions in a collection of bits would require complicated circuits to perform arithmetic. Complicated circuits leads to more expensive and usually less efficient circuits, so floating- point numbers are not encoding so flexibily. Instead, floating-point numbers are always stored in *normalized scientific notation*, meaning there is only one (non-zero) digit to the left of the point. A decimal example: 5022.4 is written as 5.0224×10^3. A binary example: 0.0101001 is written as 1.01001×2^{-2}.

The IEEE 754 standard (designed by the Institute of Electrical and Electronics Engineers) has been adopted by most computer architectures. This standard floating-point data type, called `float` in Java, is 32 bits: 1 for the sign, 8 for the exponent, and 23 for the fraction. Because of the explicit sign bit, the encoding for the exponent is unsigned magnitude. The range of a `float` is huge because of the eight bits reserved for the exponent. A `float` is not as precise as, say, an `int` though because each value in the range cannot be represented (23 bits isn't enough – in fact, no finite number of bits is enough!). For any floating-point representation, there is this tradeoff between *range* and *precision*.

The general encoding for a floating-point value is

$$(-1)^S \times F \times 2^E$$

but there are several low-level details. Because there is only one non-zero digit in binary, the digit to the left of the binary point in normalized scientific notation will always be 1. There is no need to waste a bit on this, so the leading 1 is implicitly assumed; the fraction bits (bits 22 through 0) just contain the values to the right of the binary point. And these values are read from left to right, unlike for whole numbers, just as the indexing works from left to right in the summation notation above.

Because an explicit sign bit is used, the value of the exponent encodes unsigned magnitude. This means that only zero and positive exponents can be represented. Instead, to allow a balanced range of very small (negative) and very large exponents, the value of the actual exponent encoded is 127 less than the unsigned value in bits 23 to 30; the exponent is said to have a *bias* of 127.

Without any other complications, this bias would allow the exponent to take on values between -127 and 128. However, two exponent values – 0 and 255 – are treated differently. With an exponent of 255 and a fraction of 0, a 0 sign bit encodes $+\infty$ and a 1 sign bit encodes $-\infty$. An exponent of 255 and a non-zero fraction encodes `NaN` (not a number) for situations like dividing by zero. If the exponent and the fraction are both 0, the value encoded is 0 (regardless of the sign bit). If the exponent is 0 and the fraction is non-zero, the value represented is *denormalized*, meaning a leading 1 is not implicit. A denormalized value for the `float` data type takes the exponent -126. Although we will not go into the details, this facility for denormalized numbers allows for more precise small numbers at the low end of the `float`'s range.

As we have seen, the corner cases are quite complicated, but most of the possible encodings follow a simple recipe. We also recognize that although floating-point numbers allow fractional numbers to be represented, they also allow integers of huge magnitude to be represented, unlike the integral data data types.We summarize the `float`'s encoding:

S	E	F	meaning
s	$1 \leq e \leq 254$	f	$(-1)^s \times 1.f \times 2^{(e-127)}$
0	0	0	zero
1	0	0	zero
s	0	f `!=`0	$(-1)^s \times 0.f \times 2^{-126}$
0	255	0	$+\infty$
1	255	0	$-\infty$
s	255	f `!=`0	`NaN`

As an example, we'll compute the floating-point representation of -12.75. We need to first compute the binary representation of the magnitude, 12.75, and then rewrite it in normalized scientific notation. The binary representation for the whole part (12) is 1100, and the binary representation for the fractional part (.75) is .11. Thus, 1100.11 in base-2 is 12.75 is base-10. To get this into normalized form, we shift the binary point three places to the left and multiply by 2^3: 1.10011×2^3.

Now that we have the number in normalized scientific notation, we can figure out how the sign, exponent, and fraction bits should be assigned. Since this is a positive number, $s = 0$. We know that the exponent should have a bias of 127, so we want the value that satisfies $e - 127 = 5$. Thus, $e = 132$. The binary representation of 132 is 1000 0010. Finally, since the leading 1 is implicit, the fraction bits should be set with 10011 and 0's for the rest. So the final 32-bit floating-point encoding is:

```
0 10000010 10011000000000000000000
```

2.1 Exercises

1. Another floating-point data type called `double` is even larger. A `double` is 64 bits with 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fraction. The encoding, including the special corner cases, is analogous to that of `float`. Formalize the encoding scheme by deriving a summary table similar to the table given for `float` in this section.

2. For each of the following decimal values, compute their `float` and representations. If you want more practice with binary arithmetic and floating-point encoding, compute their `double` representations as well.

   - 57.8125
   - -2.3

3. What is the largest number that can be represented in a `float` with a value of 0 in the fraction bits? with a `double`? Also approximate these numbers in base-10 normalized scientific notation representation. Note: You can use the approximation that $2^{10} = 1024 \approx 1000 = 10^3$.

4. ★ What is the largest positive integer that can be stored in a `float`? in a `double`? State your answers in terms of powers of 2 and also approximate them in decimal notation.

   Making use of sums of infinite geometric series will be useful in deriving the answers. The formula for an infinite geometric sum is

$$S_\infty = r^0 + r^1 + r^2 + \cdots = \sum_{k=0}^{\infty} r^k = \frac{1}{1-r}$$

   where $0 \leq r < 1$. Notice that if each of the r's has the same coefficient a, the formula becomes:

$$\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}$$

Chapter 3

Bitwise operators

As you will learn in Section 4.1.1 of the Guzdial textbook, computer screens display a large 2-dimensional array of *pixels*, or *picture elements*. Because there are so many pixels, we perceive the result to be a smooth image. Each pixel on the screen is set to a certain value encoded in a scheme known as ARGB, or simply RGB. An enormous spectrum of colors (more than the eye can differentiate) can be created as different combinations of red, green, and blue light. Thus, a color can be specified by certain amounts of each of these three components. The fourth component, the *alpha* component, is used to specify how transparent the pixel's color should appear when composed with colors from, say, other windows or images on the screen.

Modern displays use 32 bits to store the RGB value for each pixel – eight bits for each component – and the value that these bits encode can be thought of as unsigned magnitude values. From our understanding of binary representation, we know that an 8-bit unsigned string can take on the values 0 through 255. For each of the three color components, the higher the value, the higher the intensity of that color. For the alpha component, the higher the value, the lesser the degree of transparency.

When manipulating digital images or the computer screen itself, we often get and set each of a pixel's RGB components separately and perform computation on them. We might want to, for example, create a new `java.awt.Color` object using one of its constructors, the one with parameter list `int r, int g, int b, int a`. Using this particular constructor allows the use of `int`s for each component, even though the 32-bit size of an `int` is much larger than we need to accomodate the range of numbers 0 to 255. We know that a 4-bit binary string can represent up to 256 different values, so if we use `int`s to store the RGB components of a large number of pixels, an unnecessarily large amount of memory (four times the minimum amount) would be used.

One improvement is to use `short`s instead of `int`s to store each component. Even though a `short`'s 16 bits is still more than required to represent 256 numbers, it is half the size used up by an `int`. Although the `Color` class doesn't provide a version of the constructor that explicitly takes `short`s, they can be passed wherever `int`s are expected (because of implicit upcasting).

What about using `byte`s, since their 8-bit size is exactly what's required to represent a component? Although a Java `byte` can represent 256 values, because it is a signed data type, it cannot represent the *correct* 256 values for the task at hand; its range is -128 to 127, as opposed to the 0 to 255 we use for RGB components. What happens if you try to assign a value greater than 127 to a `byte` variable?

One compromise is to instead assign each component a value from -128 to 127. This is not acceptable, however, because it is not intuitive to think about the assigning a pixel a "negative amount of red light" for example. So the next compromise might be to use the numbers -128 to 127 when storing the component values in memory and then offsetting by 128 whenever displaying data to or reading data from users. This solution is also not satisfying, because forgetting to this somewhere could lead to confusing bugs.

What we want is to use a single 32-bit `int` to contain the entire RGB value. Fortunately, most languages, including Java, offer low-level operations to compute directly on the bits, and we can use these to get around the problems we are having balancing data size and data representation (that the `unsigned byte` that we want is not offered by Java). Three topics that we will now explore are bitwise shifting, bit masks, and hexadecimal notation. Armed with these, we will see how to elegantly solve our RGB representation problem.

Java provides three *bit shift* operators: `<<`, `>>`, and `>>>`. `<<` is the *left shift* operator, and it moves all of the digits in the binary encoding `n` positions to the left, where `n` is the right-hand operand provided to the operator. For example, `10 << 1` is 20 since the binary encodings of 10 and 20 are 0000 1010 and 0001 0100. `>>` is the *signed right shift* operator, which moves all bits `n` positions to the right and fills in the leading bits with zeroes or ones depending on the sign of the left-hand operator. In comparision, the *unsigned right shift* operator `>>>` fills in leading zeroes regardless. It is useful to note that left shifting has the effect of multiplying by two `n` times and right shifting has the effect of dividing by two `n` times, up to the limitations of overflow.

Java also provides four *bitwise* operators. The unary operator `~` (NOT) simply flips every bit. The three binary bitwise operators – `&` (AND), `|` (OR), and `^` (XOR) – encode the following *truth tables* for every pair of bits by position:

a	*b*	*a*`&`*b*	*a*`|`*b*	*a*`^`*b*
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

For example, `-1 & 4` is 4 since the (2's complement) binary encodings of -1 and 4 are 1111 and 0100.

When manipulating integer data types in a computer's memory, we often work with groups of 8, 16, 32, and 64 bits at a time, because computer hardware is built that way. Manipulating binary data in these sizes quickly becomes impractical, so the *octal* (base-8) and *hexadecimal* (base-16) representation systems are often used to succinctly describe large numbers of bits. Both of these encoding schemes work symmetrically to decimal and binary, except that with hexadecimal some new symbols need to be used because there are only ten numeric characters. The octal system, which uses the the symbols 0 through 7, is useful for representing groups of three bits. Each of the eight octal symbols represent the following bit patterns:

0 = 000	2 = 010	4 = 100	6 = 110
1 = 001	3 = 011	5 = 101	7 = 111

Because groups of four bits are commonly used, hexadecimal notation is more commonly used; a single hexademical character represents one of 16 values, the number of states a 4-bit string can take. The symbols 0 through 9 represent themselves, 'a' represents 10, 'b' represents 11, and so on through 'f', which represents 15. The sixteen hexadecimal symbols represent the following bit patterns:

0 = 0000	4 = 0100	8 = 1000	c = 1100
1 = 0001	5 = 0101	9 = 1001	d = 1101
2 = 0010	6 = 0110	a = 1010	e = 1110
3 = 0011	7 = 0111	b = 1011	f = 1111

In Java source code, integer constants are interpreted in decimal notation. You can instead tell the compiler to treat integers as octal or hexademical notation. For an octal integer literal, start the number with a `0`. For example, `int i = 014` assigns the value 12 to `i` (since 014 is the octal representation of 12). To enter a hexademical literal, start the number with `0x`. For example, `int i = 0xc` also assigns 12 to `i`.

Using the bitwise AND operator together with a hexademical constant is commonly used to filter away bits that are not currently of interest. For example, suppose we are only interested in the lowest-order eight bits of a 32-bit integer. To filter away the higher-order 24 bits, we can apply the AND operator to the integer and the *bit mask* `0xff`. The result is the value of just the lowest eight bits.

Combining the techniques of bit masking and right bit shifiting, we can now devise a scheme to encode all four components of an ARGB pixel in one 32-bit `int`, let's call it `argb`. Suppose we store the A component in bits 24 to 31, R in 16 to 23, G in 8 to 15, and B in 0 to 7. We can then read the value of each component with the following expressions:

- `argb & 0xff` for the B component,
- `(argb >> 8) & 0xff` for G,
- `(argb >> 16) & 0xff` for R, and
- `(argb >> 24) & 0xff` for A.

Notice that these expressions are shifting the appropriate bits into the eight rightmost ones, and then filtering away everything else using the bit mask. The result of each expression is that component's value in the range 0 to 255. Indeed, the `Color` class offers a constructor that takes a single `int` parameter that is interpreted in this way.

3.1 Exercises

1. Suppose that to extract each of the components out of the 32-bit representation we wanted to use an octal bit mask instead of a hexademical one. What would the four expressions be to extract each of the components?

2. When reading RGB components out of the 32-bit representation, suppose we wanted to apply a hexadecimal bit mask first and then shift second. What would the four expressions be to extract each of the components?

3. In the previous section, we saw a method for converting decimal integers to binary. We now present an alternate way of achieving the same conversion.

   We will work through the same example as before, converting the number 155. What we'd like to compute is the binary string $a_7 \dots a_0$ that satisfies the equation:

$$155 = a_7 2^7 + a_6 2^6 + \cdots + a_1 2^1 + a_0$$

   What we will do is repeatedly divide the entire equation by 2. But to do that, we need to set the stand-alone coefficient to 1 if the number on the left-hand side is odd (and set it to 0 if the number is even). Repeating this process until the left-hand side becomes 0 will give us the correct values for each of the coefficients.

$$\begin{array}{rcll}
155 & = & a_7 2^7 + a_6 2^6 + a_5 2^5 + a_4 2^4 + a_3 2^3 + a_2 2^2 + a_1 2^1 + a_0 & \text{set } a_0 = 1\text{, divide by 2} \\
77 & = & a_7 2^6 + a_6 2^5 + a_5 2^4 + a_4 2^3 + a_3 2^2 + a_2 2^1 + a_1 & \text{set } a_1 = 1\text{, divide by 2} \\
38 & = & a_7 2^5 + a_6 2^4 + a_5 2^3 + a_4 2^2 + a_3 2^1 + a_2 & \text{set } a_2 = 0\text{, divide by 2} \\
19 & = & a_7 2^4 + a_6 2^3 + a_5 2^2 + a_4 2^1 + a_3 & \text{set } a_3 = 1\text{, divide by 2} \\
9 & = & a_7 2^3 + a_6 2^2 + a_5 2^1 + a_4 & \text{set } a_4 = 1\text{, divide by 2} \\
4 & = & a_7 2^2 + a_6 2^1 + a_5 & \text{set } a_5 = 0\text{, divide by 2} \\
2 & = & a_7 2^1 + a_6 & \text{set } a_6 = 0\text{, divide by 2} \\
1 & = & a_7 & \text{set } a_7 = 1\text{, divide by 2}
\end{array}$$

   - Use this technique for computing the binary representations for 43, 84, 111, and 300.

4. Riddle: If only you and dead people can read hexadecimal, how many people can read hexadecimal?

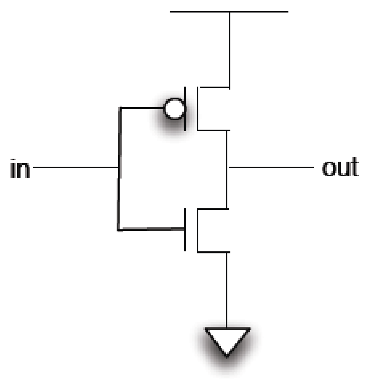
Chapter 4

Gates and circuits

In a previous section we asserted that a computer's state can be thought of as a series of 1's and 0's since it comprises transistors that are either passing or blocking current at any given time. In this section, we take an introductory look at how these two physical states can be used to create circuits that encode arbitrary logical functions, laying the foundation for digital computation.

The MOS transistor, short for *metal-oxide semiconductor*, is the basic block for most computer processors. Two types of MOS transistors, p-mos and n-mos, are circuit elements that encode the states of 1 and 0, of allowing current to pass through a wire or not. An n-mos transistor placed in a circuit between a *source* of electrical current and a *drain* exhibits the following physical properties: if a voltage is applied to the transistor, the circuit is *closed*, so current can pass through the transistor; if no voltage is applied to the transistor, the circuit is *open*, so current cannot pass through. The behavior for a p-mos transistor is the opposite: if a voltage is applied, current cannot pass through; if no voltage is applied, current can pass through. Although we will not explore the physics that explain these electrical properties, we make use of them to build circuits that encode logical functions.

Recall the single-bit NOT function, which simply inverts its input. The following circuit, consisting of one n-mos (on bottom) and one p-mos transistor (on top), encodes the NOT function:

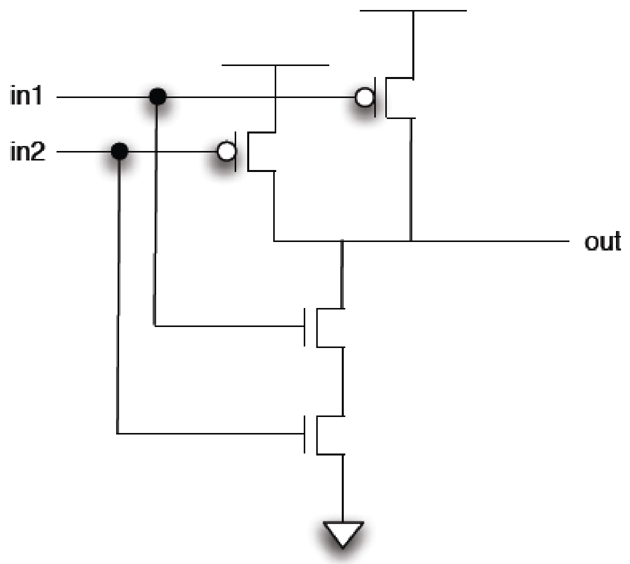


If there is a voltage on the input wire (if the input is 1), the top transistor blocks current and the bottom transistor allows current to pass. Since the bottom transistor is connected to the drain, the output wire has no voltage (the output is 0). Conversely, if there is no voltage on the input wire (if the input is 0), the top transistor allows current to pass while the bottom transistor does not. Since the top transistor is connected to the electrical source, the output wire has voltage (the output is 1). From now on, we will refer to the states of the wires as 1 and 0 instead of voltage or no voltage. Circuits that are made of both n-mos and p-mos transistors are called *CMOS circuits*, short for complementary MOS.

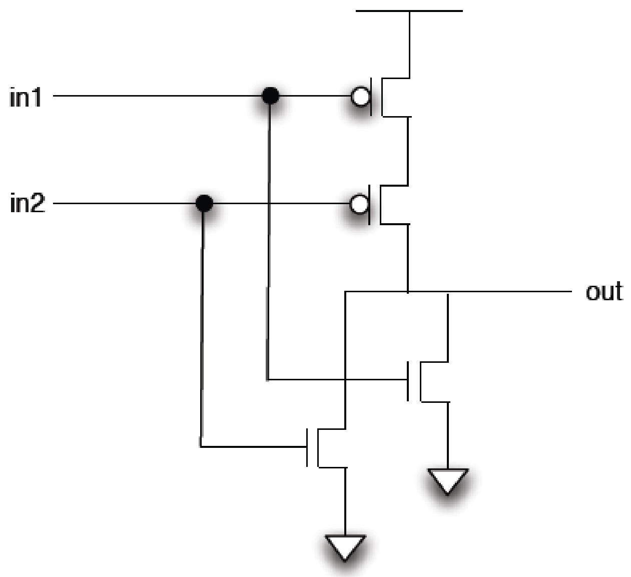
We have seen truth tables for the 2-input functions AND and OR. Two other functions, NAND and NOR, have the following truth tables:

a	b	NAND	NOR
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

The NAND function is simply AND followed by NOT, and NOR is OR followed by NOT. The following CMOS circuit encodes NAND:

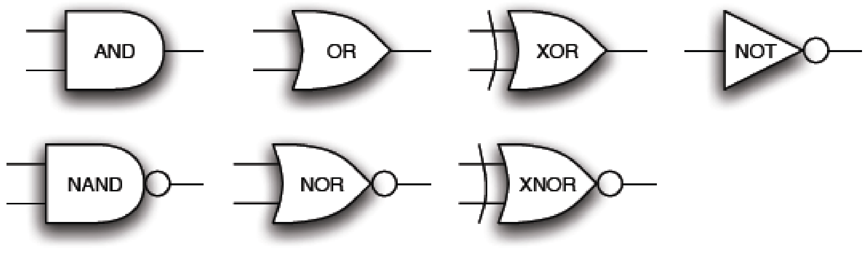


The following CMOS circuit encodes NOR:



The NAND and NOR functions we have seen so far accept two inputs, but their meanings extend easily to any number of inputs. For any number of inputs, NAND and NOR can be implemented by hardware circuitry in the same ways as they are for two inputs. To implement AND and OR with circuits, we simply add the circuitry for NOT after the circuitry NAND and NOR, respectively.

NAND is said to be *logically complete* because any arbitrary logical function can be built out of a circuit containing just NAND gates. This is an extremely powerful property, because it means that hardware to perform any kind of computation can be built out of these very simple *logic gates*. Similarly, NOR is logically complete, as is the set of gates {AND, OR, NOT}. These gates are the building blocks of all digital circuits, and because they will be used so frequently, the following shorthand symbols represent the transistor-level diagrams that encode each function:

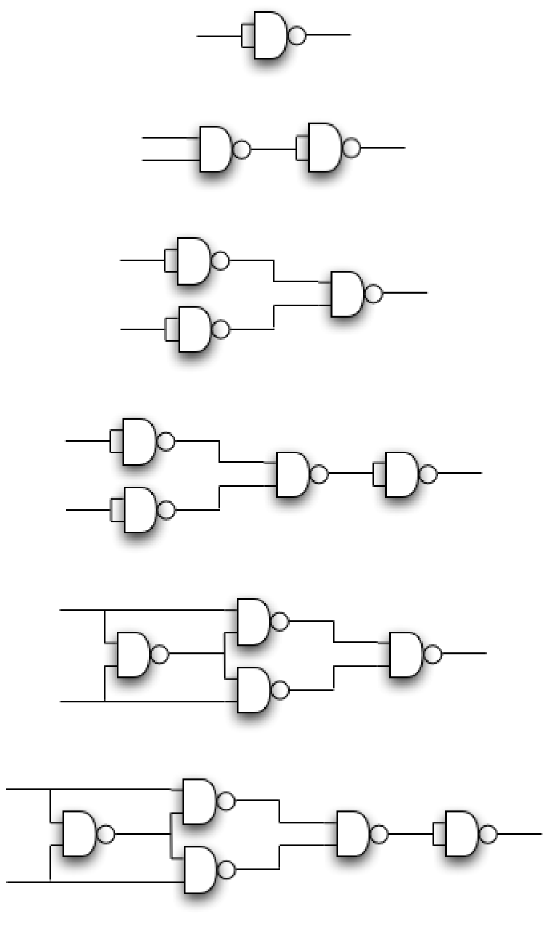


There are two other useful logical functions, XOR and XNOR, in constructing digital circuits. The truth tables for the 2-input versions of these functions are:

a	b	XOR	XNOR
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

Note that XNOR should be thought of as "not-XOR", not "exclusive-NOR".

We've asserted that NAND is logically complete, so we'll take a look at how to implement some other (2-input versions of) logical functions using just (2-input) NAND gates. From top to bottom, we have NOT, AND, OR, NOR, XOR, and XNOR:



We'll take a closer look at each of these circuits:

- NOT. It is easy to check that 0 NAND 0 = 1 and 1 NAND 1 = 0, which is the behavior of NOT.
- AND. Clearly, appending a NOT to NAND encodes AND.
- OR. Checking that this circuit behaves like OR takes a little more work. To check, we work through the truth table that is induced by the circuit. We use the symbols $\overline{a}$ and $\overline{b}$ to mean "NOT a" and "NOT b".

a	b	$\overline{a} = a$ NAND a	$\overline{b} = b$ NAND b	$out = \overline{a}$ NAND $\overline{b}$
0	0	1	1	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	1

We see that the outputs are indeed the correct values for the OR function. In fact, what we have just demonstrated in this truth table is the first of two properties known as *De Morgan's Laws*:

– $\overline{p \text{ AND } q} = \overline{p} \text{ OR } \overline{q}$
– $\overline{p \text{ OR } q} = \overline{p} \text{ AND } \overline{q}$

What the circuit encodes is $\overline{\overline{a} \text{ AND } \overline{b}}$, which equals a OR b by the first De Morgan's Law.

- NOR. Clearly, appending a NOT to OR encodes NOR.
- XOR. Since this circuit is a little complicated, we will work through the truth table it induces:

a	b	$c = a$ NAND b	$d = a$ NAND c	$e = c$ NAND b	$out = d$ NAND e
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	0

We see that the outputs are indeed the correct values for the XOR function.

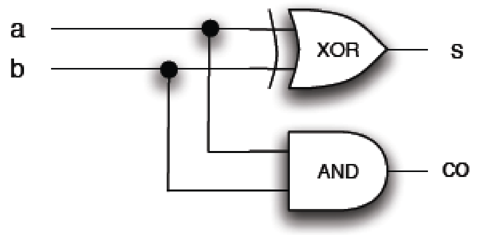
- XNOR. Clearly, appending a NOT to XOR encodes XNOR.

Similarly, all these logical functions can also be built using just NOR gates.

We will now attempt to build a CMOS circuit to add binary numbers. We will first build a circuit that can together two single bits in the hope that we can string several of these together to add two n-bit numbers. Adding two bits yields: sum (s) and carry-out (co). The following truth table summarizes addition on two bits a and b:

a	b	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

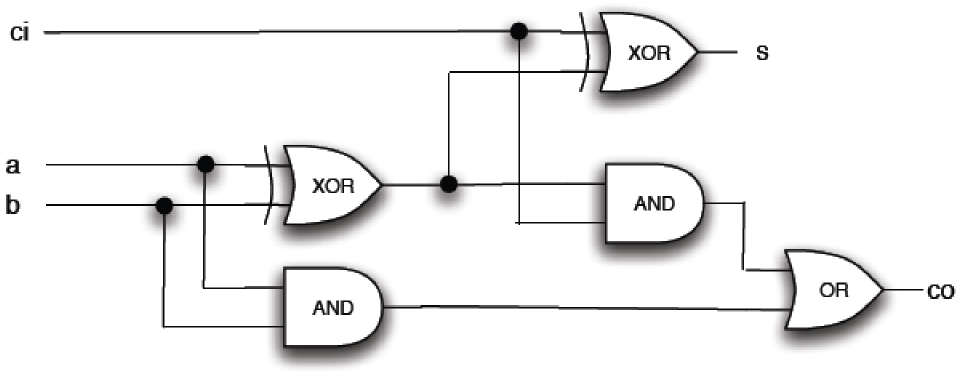
Notice that the column for s has the same values as a XOR b. In other words, s is simply the XOR function. The column for co is the same as a NAND b. After noticing these two identities, the circuit that computes s and co can be constructed as follows:



If we only want to add two bits together, this circuit does just fine. To be able to add multiple-bit numbers together, though, the carry-out from the previous position must be taken into account when summing the next position. The circuit we just constructed, which does not take a carry-in input, is known as a *half adder*. What we want to construct now is a three-input function (the two bits, a and b, and the carry-in, ci) that produces outputs s and co. The truth table for this function is:

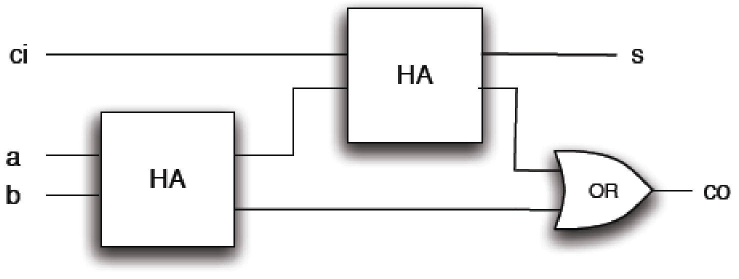
ci	a	b	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The following circuit encodes this function:

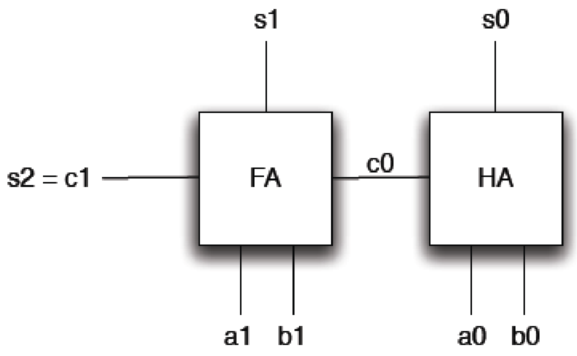


Verify that this circuit does in fact encode the correct truth table.

This is now a complete adder, called a *full adder*, for two bits, and they can be strung together to add multiple-bit numbers. Notice that the full adder is really just two half adders and an OR. We can redraw the full adder treating the half adder circuit as a single unit:



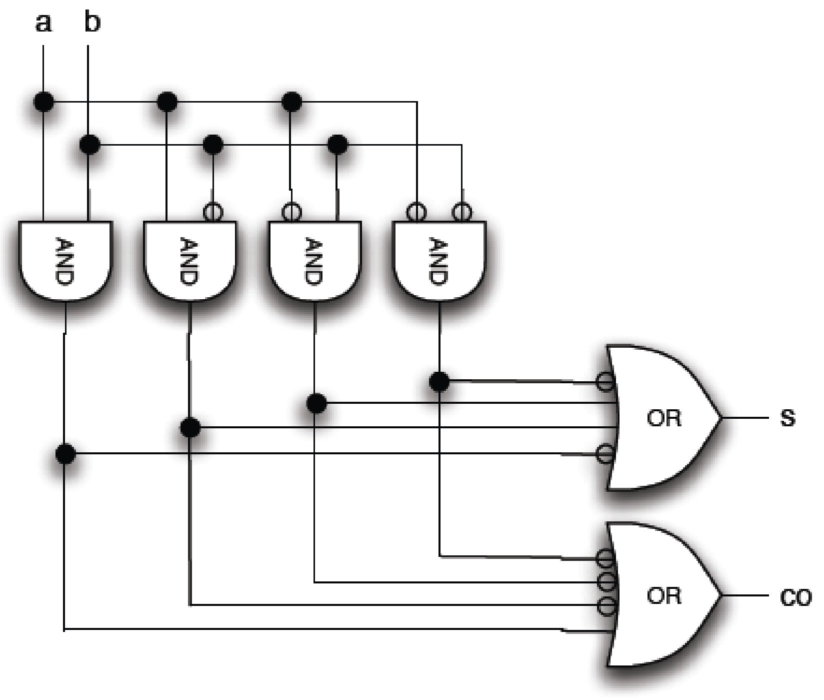
We can now build a 2-bit adder by stringing together a half adder and a full adder:



This technique can be used to build adders for numbers of any size, such as `ints` (32 bits) or `longs` (64 bits). These are called *ripple-carry adders* because the carry from one full adder gets propagated to the next. As more full adders are strung together, however, the time required for the result to be computed, known as the *latency*, increases. In practice, more complicated and very clever circuits have been designed to add multiple-bit numbers more quickly. Although these circuits are more complex, the latency is much less, so they are used in computer processors; you will likely not find ripple-carry adders to add 32-bit numbers in modern processors.

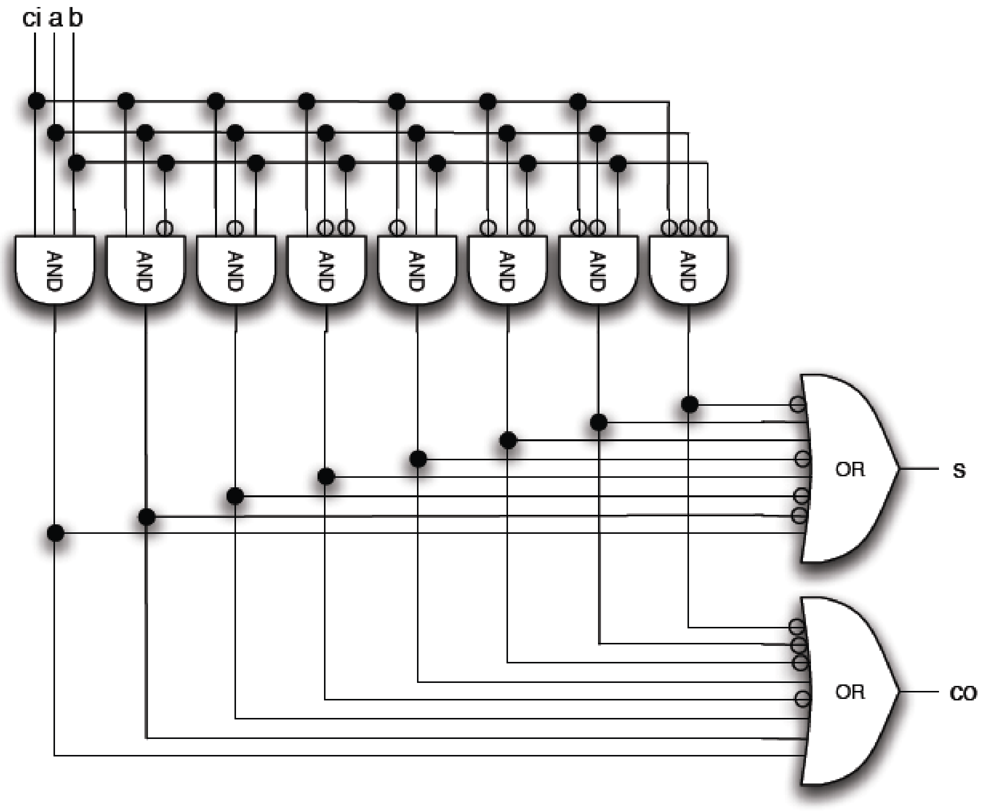
Building the half adder and full adder circuits were easy because their outputs were the same as recognizable functions (or simple combinations of them). Even slightly more complicated functions, though, will require many more gates to encode the desired truth table. Fortunately, we can build a circuit called a *programmable logic array*, or *PLA*, that allows us to encode an arbitrarily complex truth table in an organized way.

To demonstrate, we look at a 2-input 2-output PLA that has the same behavior as the half adder circuit above:



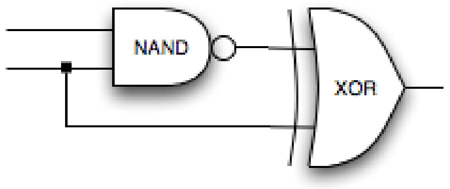
(Note that in our drawings, we use an open circle as shorthand for the NOT, or inverter, gate.) This PLA has 2 inputs, 4 AND gates, and 2 OR gates. In general, for an n-input m-output function, we use a PLA that has 2^n AND gates and m OR gates. To encode any function, the structure of the PLA stays exactly the same except for the connections entering the OR gates. The AND gates are set up to represent every combination of the n inputs. To encode which inputs map to which outputs, we "program" these connecting wires by inserting inverters in places that yield the logical outputs that we want. To program, or configure, the outputs we want, we can place inverters on some wires before they get passed to the OR gates. For example, for the s output for the half adder, we want a 1 for inputs 01 and 10 and a 0 for inputs 00 and 11. Therefore, for the wires going into the s OR gate, we add inverters to the lines that correspond to 00 and 11; leaving the wires for inputs 01 and 10 without inverters allows the value of s to be 1 for these two inputs. For the co output, we want a 1 for 11 and 0 for everything else. Therefore, for the wires going into the co OR gate, we place inverters on the wires corresponding to 00, 01, and 10; this allows just the input 11 to have a co of 1.

Similarly, we can encode the full adder as a 3-input 2-output PLA:

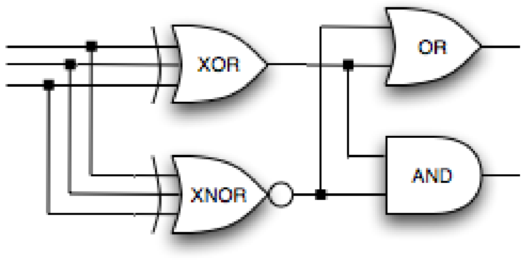


4.1 Exercises

1. Draw transistor-level diagrams for the 3-input NAND and NOR gates.
2. Since NOR is logically complete, all logical functions can be encoded as compositions of NOR gates. Using just NOR gates, draw circuits that implement (2-input versions of) NOT, OR, AND, NAND, XOR, and XNOR.
3. What is the truth table for the following circuit?



4. What is the truth table for the following circuit?



5. Configure a PLA to encode the following truth table:

a	b	c	d	e
0	0	1	0	1
0	1	0	0	1
1	0	0	0	0
1	1	0	1	0

6. Draw a circuit that encodes the 3-input function: NOT (a OR (b AND c))

Chapter 5

Faster addition

The time it takes for a computer processor to perform addition is crucial to its performance. The addition hardware is used in virtually every cycle for tasks like figuring out what the next instruction to execute is and what the addresses of data in memory are. The time required by the addition hardware is a major factor in what a processor's *clock cycle* is. Taking a simplified view, we can think of a computer's clock cycle as how long it takes for a single machine instruction to execute. So having finely-tuned addition circuitry is a high priority.

We have already seen a circuit to add numbers – the *ripple-carry adder* – and we would like to now analyze its performance. The first step towards this goal is deciding on a metric we can use to compare different circuits. The measure we will use is *gate levels*, or *gate delay*. Specifically, we will say that a circuit's gate delay is n if n is the number of gates along the longest path from some input to some output. We make the assumption that all logic gates require the same amount of time to complete. Although there are lower-level physical properties that have real consequences, this simplification will still yield very useful comparisons. Another simplification we make is that a logic gate contributes "one unit of delay" no matter how many inputs it has. In practice, however, a gate that has a large number of inputs does degrade performance.

Let's analyze the gate delay of the circuits we have seen so far:

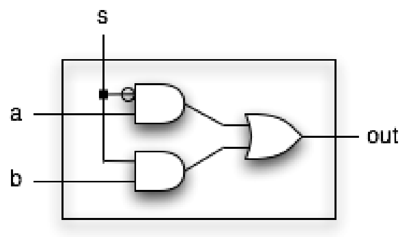
- The longest path that can be taken in the half adder is 1 gate (in fact, every path is 1 gate deep), so the half adder has a gate delay of 1.
- The full adder that we constructed using two half adders has a gate delay of 3; its longest path starts with input wire a, goes through one half adder, goes through the second half adder, and then goes through the OR gate.
- The PLA that encodes the half adder has 2 gate delays, since every path goes through an AND and an OR. Note that we do not count standalone inverters as contributing to gate delay.
- The PLA that encodes the full adder also has 2 gate delays for the same reason.

In fact, since every PLA is structured as a level of ANDs followed by a levl of ORs, every PLA is 2 gate levels. And because we can encode an arbitrary logical function using a PLA, it takes only 2 gate levels to encode any truth table. As alluded to before, this analysis breaks down in practice when there is a large number inputs to the ANDs and ORs of the PLA.

Let's now consider the ripple-carry adder, which adds two n-bit numbers by stringing together n full adders. (In our first 2-bit ripple carry adder, we actually used a half adder, not a full adder, for the first bit. We will later see why we use a full adder even for the first bit. Notice though that this decision will have no effect on the gate delay.) The longest path through this adder will go from an input to the carry-out of the nth full adder, passing through each previous full adder along the way. Since each full adder (encoded in a PLA) has a gate delay of 2, the overall gate delay of an n-bit ripple-carry adder is $2n$.

There are many adder circuits that achieve far better performance than ripple-carry adders. One of the simpler improvements is called the *carry-select adder*. The idea is that instead of having all the full adders just waiting for the carry-in to propagate before beginning their work, in the meantime they can perform the computation for *both* possible values of carry-in. Once the actual value of the carry-in is known, then the corresponding result (that was already calculated) can be selected as the output.

Before we can continue, we need to examine a circuit that can perform this selection operation. A *multiplexor*, or *mux*, is a circuit that does just that: it takes in several inputs and chooses one (or sometimes more) to output. Consider the following circuit:

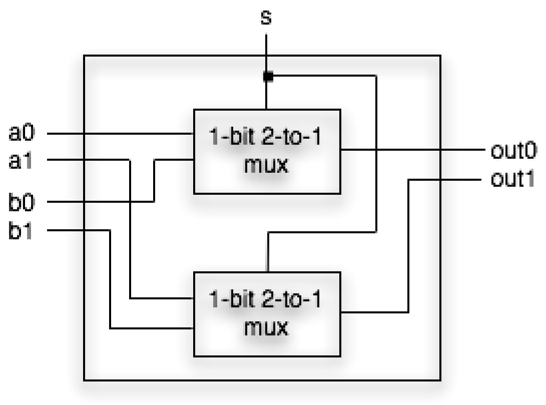


Now consider the truth table it induces:

s	a	b	c	d	out
0	**0**	0	0	0	0
0	**0**	1	0	0	0
0	**1**	0	1	0	1
0	**1**	1	1	0	1
1	0	**0**	0	0	0
1	0	**1**	0	1	1
1	1	**0**	0	0	0
1	1	**1**	0	1	1

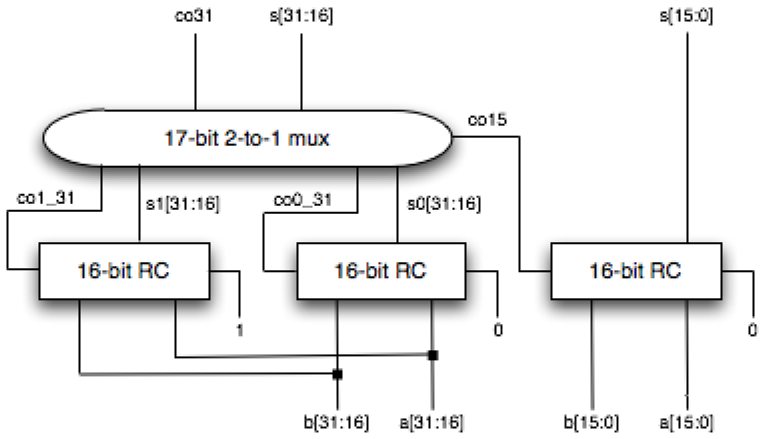
Notice that when $s = 0$, the value of *out* is whatever the value of a is. When $s = 1$, the value of *out* is whatever the value of b is. So given two inputs a and b, one of them is selected to be output depending on the value of s. We call this circuit a 1-bit 2-to-1 mux, since it takes two 1-bit inputs and chooses one of them. The gate delay of this mux is 2.

For the carry-select adder, values for both sum and carry-out will be computed before knowing the value of carry-in. When the time comes to select the results to propagate, there will be more than just one bit of information to select, unlike with the 1-bit mux above. A 2-bit 2-to-1 mux can be built using two 1-bit muxes:



Note that the gate delay of this mux is also 2. In this fashion, we can build a mux to select between any size input. In the case of the 32-bit carry-select adder, we will want to select between 17-bit inputs (16 sum bits and 1 carry-out bit).

We can now make use of 16-bit ripple-carry adders and 17-bit 2-to-1 muxes to build a 32-bit carry-select adder:

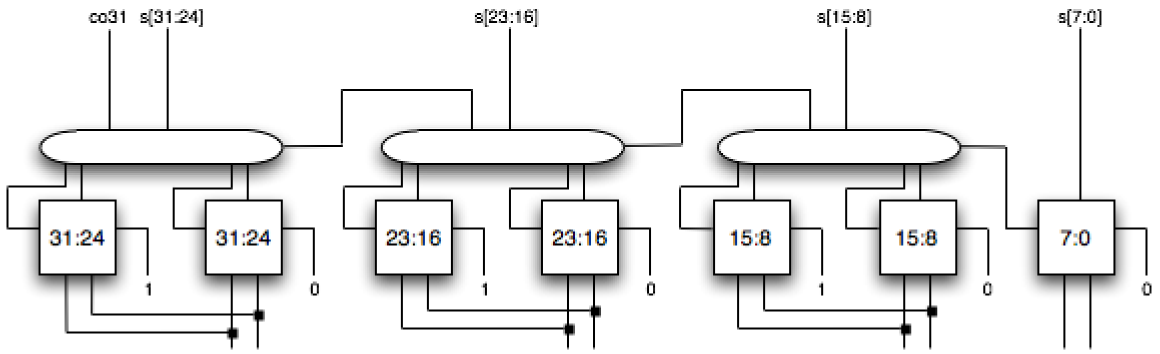


What we have done is separate the addition of the lowest-order eight bits (0 through 15) and the highest-order eight bits (16 through 31). Each part is processed by a separate 16-bit ripple-carry adder as opposed to all 32 bits processed by a 32-bit ripple-carry adder. The addition of bits 0 through the 15 work as normal because the carry-in to the first bit is known to be 0. The result of summing bits 16 through 31 depends on what the carry-out of bit 15 is, however. Instead of waiting the $2 \times 16 = 32$ gate delays for co_{15} to be computed, we instead compute the sum of bits 16 through 31 for both possible values of co_{15}. Therefore, we need two 16-bit ripple-carry adders to process bits 16 through 31 in parallel. Now we have all three ripple-carry adders processing at the same time. After the 32 gate delays it takes for co_{15} to be generated, this value is used as the selecting wire to a mux that chooses which of the two already-computed sum and carry-out pairs to use.

Has this *speculative* approach to summing bits 16 through 31 – speculative because computation is carried out before having all the information it needs to continue – improved the gate delay of addition? To answer this, we need to look at the longest path in this circuit. The longest path goes from an input to the 0 through 15 ripple-carry adder and through the mux. The gate delay of this 32-bit carry-select adder is thus $(2 \times 16) + 2 = 34$, much better than the $2 \times 32 = 64$ of a 32-bit ripple-carry adder.

If we wanted to build any n-bit carry-select adder in this way – breaking up each summand into two equal pieces – the gate delay is that of a ripple-carry adder to process half the bits plus that of the mux. Thus, an n-bit carry-select adder that breaks the inputs into two pieces takes $2(n/2) + 2 = n + 2$ gate delays.

A natural extension is to try and improve performance even further by splitting the inputs into even smaller pieces. The following carry-select adder breaks the summands into four 8-bit pieces (note that we omit most wire labels for brevity):



Since the lowest-order bits (0 through 7) will have a carry-in of 0, the sum for these bits can be computed directly. The sum of bits 8 through 15 depends on $co7$, so the carry-select adder computes sums for both values of $co7$. Similarly, the sum of bits 16 through 23 depends on $co15$, and the sum of bits 24 through 31 depends on $co23$, so these sums are computed for both values of their respective carry-ins. When the carry-ins have been computed, their values are used as the selector wires to the muxes.

To see what performance gains, if any, this circuit achieves, we need to identify the longest path. For this circuit, the longest path goes through the ripple-carry adder for bits 0 through 7 and then through all three muxes. So the gate delay of this 32-bit carry-select adder in four pieces is $2(8) + 2 + 2 + 2 = 22$, which is a significant improvement over the two-piece version.

The performance gain from breaking the adder into more and more pieces diminshes, and even becomes a performance loss, because of the additional muxes that must be traversed. In general, though, for an n-bit carry-select adder in m-pieces, the longest path goes through the first ripple-carry adder (of size n/m) and through all $m - 1$ muxes. Thus, the gate delay is $2(n/m) + 2(m - 1)$.

There are many other highly-optimized adders used in practice, but we will not examine them here. But analyzing just the carry-select adder has served two purposes: 1) to get us thinking about how to analyze algorithms (here the algorithms happened to be implemented in hardware); and 2) to identify a fundamental difference between hardware and software: that computation can be carried out in parallel in hardware whereas it is sequential in software. Although computers can run multiple programs "at one time", this is just an effect generated by the operating system telling the processor to take turns executing each program a little bit at a time. On a single-processor machine, only one program is being executed at a given moment, so software is sequential by nature. In contrast, the carry-select addition algorithm we examined exploits the parallel nature of hardware – that multiple addition circuits can be run at the same time without increasing the gate delay – to achieve a performance gain. Software written for multi-processor computers and multi-computer networks, however, can achieve a level of parallelism in software. We will not explore these avenues any further, but the disciplines of computer architecture, operating systems, and distributed systems await the eager student.

5.1 Exercises

- For $n = 32$ and $n = 64$, compute the optimal value of m for an n-bit carry-select adder. In other words, determine what number of pieces results in a carry-select adder with the smallest gate delay. Assume that m must be a power of two.

Chapter 6

Searching

A very common programming operation is looking for a particular item in a list of items. We will assume that the list of items is already sorted. Because this task if performed so often, we would like to design an algorithm that performs this task quickly. For our purposes now, we will also assume that the list of data is stored in an array. This gives us random access to every element in the array – we can get at every element in the list in a short time (the time it takes to access a variable in memory). We will examine two algorithms that complete this task and begin analyzing how they compare to each other.

The simplest approach is to start at the beginning of the array and walk though all its elements. If we find the element we are looking for, we are done and do not have to continue looking. If we get to the end of the array and still haven't found it, we can declare that the item does not exist in this list. We call this algorithm a *linear search* since we make a straight pass through the list.

This is probably not how you would approach a search through a large list, however. Consider a dictionary, which is a very long list of words in alphabetical order plus some additional data. If you were looking for a particular word, would you start at the beginning and start scanning linearly? Of course not. Instead, you would probably flip to somewhere in the middle of the dictionary and see if you landed on the word you were looking for. If you did, then you would be done. Otherwise, if the search word was lexicographically (standard dictionary sorting order) before the one you found, you would repeat this process on just the pages between the beginning and the current page. If the search word should be after the one you found, you would repeat this search on the pages between the current one nad the last one. You would keep repeating this process, eliminating half the remaining pages in each step, until you either found your word or reached just a single word. In the latter case, you would deem that the word does not exist in the list.

This technique is known as *binary search*, and it is a much quicker way to find an item in a sorted list. Whereas in a linear search only one item is eliminated in each step, about half the remaining items are eliminated in each binary search step. Binary search exploits the fact that the data is already sorted to lead to a quicker algorithm. The question now is how to compare these two algorithms that peform the same task.

Before we even decide on what units we will use to “measure” these algorithms, we can think about different measurements we will make. These algorithms are functions of some input, and we will want to express how long they take to execute in terms of the size of the input. For instance, if we are given a sorted array of n elements, we would like to say something like Algorithm A takes $5n + 30$ steps whereas Algorithm B takes $n^2 - n$ steps. Furthermore, it is useful to consider particular properties of the input, because not all inputs of the same size will require the same time for the algorithm to complete. For example, in our case, consider two arrays of length n and a particular item we are looking for. If the the first array contains this item but the second one does not, the searching algorithms will take different number of steps to complete.

One type of input that we sometimes analyze is the *best case* input for an algorithm. For example, the best case for linear search is that the desired element is first in the list. An input of this kind results in the shortest possible execution of linear search. For binary search, the best case input is an array with the desired item exactly in the middle. Best case analysis is not usually very useful because the best case is often much simpler than other types of inputs. *Worst case* analysis picks an input that requires the algorithm to take the maximum number of steps to finish. For example, if the desired element was not in the list, then both linear and binary search would have to run process the entire array (albeit in different ways) before concluding that the element is not found; this is the worst case for the searching problem. Analyzing an algorithm's worst case provides a lot of information – it provides an upper bound on how long the algorithm may take – and is the most often performed analysis. Another possible analysis considers the *average case.* An average case analysis may more accurately reflect the running time of an algorithm over a large number of runs, especially when some inputs are significantly "worse" than others but do not occur frequently.

We will now give concrete Java implementations of linear and binary search (where the elements in the sorted array are integers), and we will then analyze them:

```
public int linearSearch(int x, int[] data) {
  for (int i=0; i < data.length; i++) {
    if (x == data[i])
      return i;
  }
  return -1;
}
public int binarySearch(int x, int[] data) {
  int bot = 0;
  int top = data.length - 1;
  int mid;
  while (bot <= top) {
    mid = (bot + top) / 2;
    if (x == data[mid]) {
      return mid;
    }
    else if (x < data[mid]) {
      top = mid - 1;
    }
```

```
      else {
        bot = mid + 1;
      }
    }
    return -1;
  }
```

One way to analyze these algorithms is to count the number of items in `data` that each algorithm visits. Although there are other statements getting executed in both algorithms, the number of elements that each compares against may give us a good idea of how much work each is doing. We will also analyze for the worst case, because we want a guarantee on how long the algorithm may take no matter what input is provided. For both algorithms, the worst case input is a `data` array that does not contain the search term `x`.

For this worst case input, `linearSearch` compares `x` with every element in the array. If we call n the length of the array (n=`data.length`), then `linearSearch` compares against at most n elements in the input array. Note that this is an upper bound – no input array can cause more than n elements to be visited – because we performed a worst case analysis.

`binarySearch` is more complicated to analyze because it does not do the same thing in every iteration – there are three different branches inside the `while` loop. The first thing to check is that this algorithm actually does terminate (checking for termination is worthwhile!). The guard of the loop checks that `bot`$\leq$`top`. This condition holds at the first iteration of the loop because of the way these variables are initialized. Inside the loop, `mid` gets a assigned a value somewhere in between `bot` and `top` – ie, `bot`$\leq$`mid`$\leq$`top`. We now need to check all three conditional branches to make sure the guard condition will not remain true forever. In the first conditional branch, the method exits. In the second conditional branch, `top` gets assigned a value strictly smaller than its old one. Similarly, in the third branch, `bot` gets assigned a value strictly larger than its old one. Since in all three branches, the method either returns or decreases the range between `top` and `bot`, this algorithm is guaranteed to terminate.

The question now is to analyze how many steps it takes to terminate (again assuming the input is a worst case input). Our first analysis will be just to count the number of elements visited in the array, just as we did for `linearSearch`. For `binarySearch`, we want to count each iteration of the loop as visiting an item in the array, because if we look at the logic inside the loop, all comparisions are being performed in terms of a single element, `data[mid]`. That is the particular item being visited in one iteration of the loop.

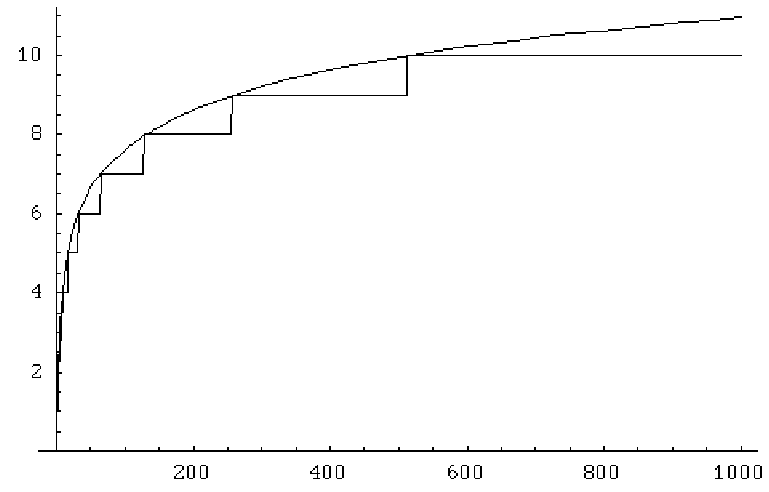
Even this simple analysis is harder than it was for `linearSearch`. This algorithm throws away parts of the array by reassigning `top` or `bot` appropriately. In particular, let m be the number of elements between indices `bot` and `top`, and let `data[mid]` not be equal to `x` (since we are assuming worst case input). So either of the last two branches will be taken. In these branches, there are m elements to start with and then either `bot` or `top` gets reassigned, so there are $(m-1)/2$ elements left on either side of the midpoint. Because m can either be even or odd, this number will not always be the same on both sides of the midpoint. An example:

0	4	6	7	9

Suppose we are looking for the number 2 in this array. Following `binarySearch`, the values in the array that would be visited are 6, 0, and 4 (for `mid`=2, 0, and 1). If we were looking for the number -2 instead, the values in the array that would be visited are 6 and 0. Notice that although in neither case was the element found, they did not compare against the same number of elements. The reason for this is because in the second iteration of the loop, `bot`=0, `top`=1, and `mid`=0. When searching for 2, it would have appear to the right of 0, and there are still elements to the right of 0 that have not yet been searched (the single element 4). When searching for -2, however, it would have to appear to the left 0, and there are no more elements to the left of 0. Because splitting even and odd arrays leads to this asymmetry, the worst case is not just an array that does not contain the search term, it is one that also results in the larger sub-array being searched whenever the two sub-arrays are not of equal length. In the example above, we had an array of two elements in the second iteration. After comparing against one element, the two sub-arrays had 0 and 1 elements, so we choose the sub-array with 1 element.

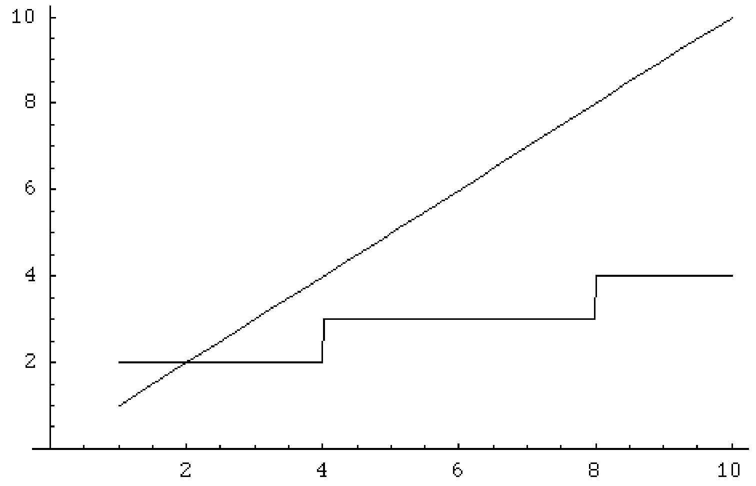
To make the analysis simpler, we will make a couple of assumptions. First we will assume that `data.length` is a power of 2 (ie, $n = 2^k$ for some $k \geq 0$). We will also assume that, when splitting an array of size m around `data[mid]`, both sub-arrays have exactly $m/2$ elements. These assumptions make the worst case analysis much simpler. We start with an array of size n and repeatedly visit one element and split the array in half. This process continues until we reach an array of 1 element; at this point we can deem the element not found. How many times do we split the array in half? Because we assumed $n = 2^k$, there will be $\log_2(n) + 1 = k + 1$ splits to get down to an array of just one element. Since one element is visited before each split, a total of $\log_2(n) + 1$ elements in the input array are visited.

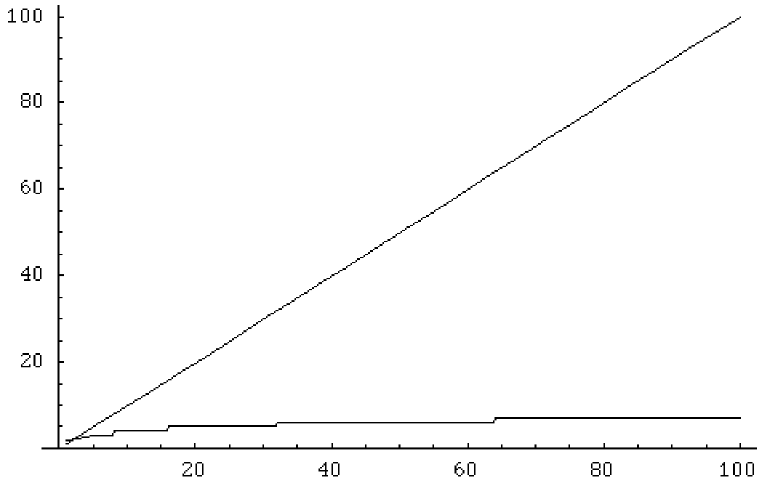
Keep in mind the assumptions we made in achieving this bound. To contrast, the following chart compares the base-2 logarithm function and the actual worst case number of comparisions (a future assignment will show to compute this number) for each input size between 1 and 1000:

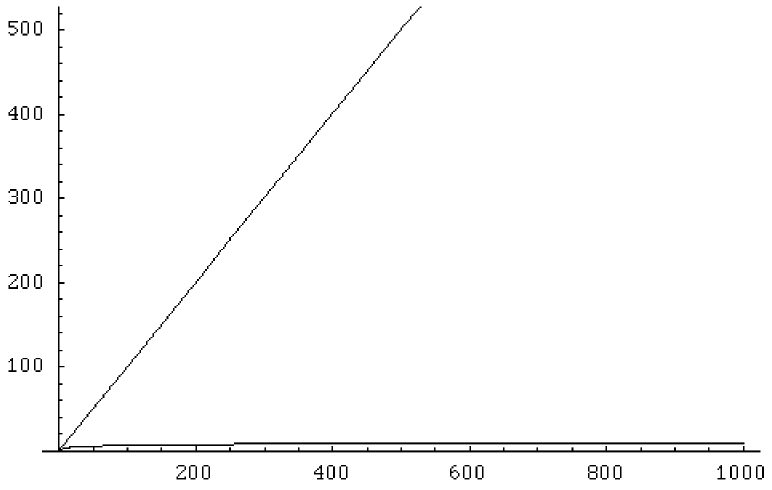


Notice how close our approximation is to the actual worst case.

To get an idea for how much better `binarySearch` does than `linearSearch`, we plot the worst case number of comparisons for each:







So far we have only taken into account the number of elements in `data` that get visited, but both algorithms clearly contain other operations. We will now consider how the other operations – assignments, conditionals, return statements – might affect the performance of the algorithms.

We will adopt a simplified model for how much each statement that gets executed "costs". We will say that every assignment, conditional, and return statement costs one operation regardless of how complicated the expressions within them may be. For example, we will count both the assignments `int a = 2;` and `int b = 2 * (1 - 3) / 5;` as one operation, even though the second will require more instructions for the machine to execute. Without knowing exactly how a particular computer architecture performs low-level tasks like adding two numbers and reading and writing from memory, we do not know exactly how many operations we should charge for each type of Java expression, so we simply assume they are all equal. Even if we did know the details of the computer architecture, we will see that we achieve powerful analysis techniques in spite of these simplifying assumptions.

Let's take another look at `linearSearch`, this time taking into account all of the operations it performs. Again we will use the variable n to denote `data.length`. Let's first analyze the for-loop. The initialization of `i` only occurs once, before the loop body ever executes, so it costs 1 operation. The conditional expression gets tested every time the loop body is entered and one last time when it evaluates to `false`. We are assuming the worst case input, so the loop body will be executed n times. Thus, the conditional expression is evaluated $n+1$ times, costing $n+1$ operations. The increment of the loop variable `i` happens each time the loop body is executed (n times), so the total cost of incrementing is n. Now we consider the rest of the loop body. The equality check is performed once for every iteration of the loop, so the total cost of the equality checks is also n. We assume that the branch never gets taken, so the `return` statement doesn't factor into our analysis. Finally, after the for-loop completes, the `return -1;` statement requires one operation. Thus, we have the total cost is $1 + (n + 1) + n + n + 1 = 3n + 3$.

We can annotate the algorithm with two counters, one for total cost and one for just number of comparisons, to demonstrate our analyses:

```
public int linearSearchCount(int x, int[] data) {
  int operations = 0, comparisons = 0;
  operations++; // initialization of i
  for (int i=0; i < data.length; i++) {
    operations++; // for-loop guard test
    operations++; // equality check
    comparisons++;
    if (x == data[i]) {
      operations++; // return
      System.out.println("ops:\t" + operations);
      System.out.println("comps:\t" + comparisons);
      return i;
    }
    operations++; // increment of i
  }
  operations++; // return
  System.out.println("ops:\t" + operations);
  System.out.println("comps:\t" + comparisons);
```

```
    return -1;
  }
```

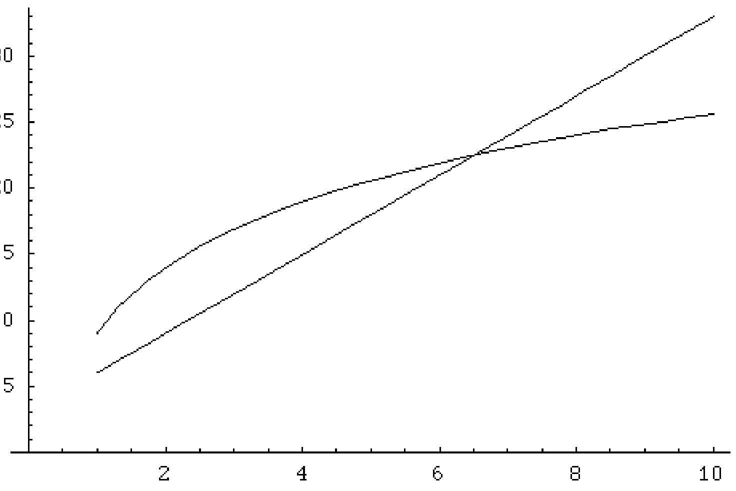
Now we can run `linearSearchCount` on a worst case input to demonstrate our accounting techniques:

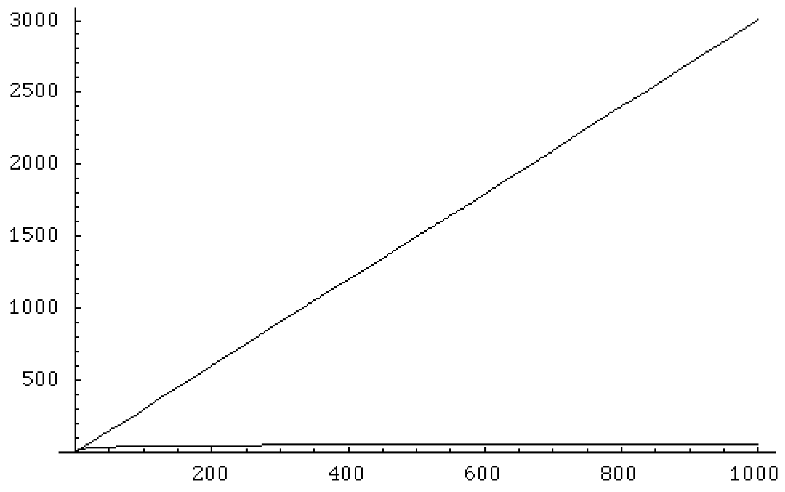
```
> int[] data = {1, 3, 5, 7, 9, 11, 13};
> Searching s = new Searching();
> s.linearSearchCount(0, data)
  ops: 23
  comps: 7
-1
```

We've already talked about the number of comparisons that `binarySearch` makes. Let's now count the total number of steps it takes in the worst case. The assignments of `bot` and `top` take two steps. Let's not count a step for `mid` because no value gets assigned to it; this choice is arbitrary, but once we have made it, we should be consistent. We will use the approximation discussed above that the loop gets executed $\log(n) + 1$ times, so the guard for the while loop gets executed $\log(n) + 2$ times. Inside the loop, we count four steps – the assignment of `mid`, the first conditional with `x`, the second conditional with `x`, and one for the step inside the else. Outside the loop, the return takes one step. Thus, overall we have $2 + \log(n) + 2 + (\log(n) + 1)4 + 1 = 5\log(n) + 9$ steps. Remember that this analysis makes use of the approximation that the loop runs $\log(n) + 1$ times.

Annotating `binarySearch` to compute the number of operations and comparisons on worst case inputs is left as an exercise.

Now that we have functions for the number of steps for both `linearSearch` and `binarySearch`, let's take a look at how their performance compares to each other:





Note that for very small inputs (about $n = 6$ and smaller) `linearSearch` is actually faster than `binarySearch`. But for large inputs, `binarySearch` is dramatically faster.

One interesting thing to note about our implementation of binary search, and most that you will find textbooks or in examples, is that it contains a bug. The offending line is the computation of the midpoint: `int mid = (bot + top) / 2;` The value `(bot + top)` may overflow the range of an `int`, and if it does, the result is a negative number. Dividing this number by two and then trying to index into an array using a negative number will result in an `ArrayIndexOutOfBoundsException`. One way to avoid this possible overflow is: `int mid = bot + ((top - bot)/2);` This goes to show that even the most benign programming tasks – like taking the average of two numbers – have room for bugs! Take a look at the Google Blog post that discusses this surprising bug and the bug report filed with Sun:

```
googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
bugs.sun.com/bugdatabase/view_bug.do?bug_id=5045582
```

6.1 Exercises

1. Design and run experiments to estimate the number of comparisons that binary search takes on worst case inputs. Recall that worst case input for binary search is not quite as simple as just arrays that don't contain the search term, because of the uneven array splitting scenario. But instead of figuring out exactly what the worst case number of comparisons is for a particular n, you will approximate it by running several different test cases where `data.length`$=n$ and where `x` is not contained in `data` (some of these will be worst case inputs and some will be close but not the actual worst case). After running several experiements, compute the averge number of comparisons for these worst-case and "slightly better than worst-case" inputs for sizes $n = 1$ to $n = 100$.

   You will want to define an annotated version of binary search that counts the number elements in the array that are visited. You will also want to, instead of just printing out what the number of comparisons is, modify the return type of binary search to include this information. This will allow you to run a large number of test cases and compute the average across all of them.

Chapter 7

Asymptotic analysis

In the previous section, we devised a simple system for analyzing two searching algorithms. We made the simplifying assumptions that every statement takes the same amount of time to execute, so each statement that gets executed costs one operation. In this section, we will analyze a few more examples in this manner, and then we will introduce some mathematical definitions that allow us to analyze algorithms in a more general way.

Let's take a look at the following method `foo` and count how many steps it takes in the worst case.

```
public void foo(int n, int m) {
  for (int i=0; i < n; i++)
    for (int j=0; j < m; j++)
      System.out.println("hello");
}
```

What is the worst case for this `foo`? For any inputs `n` and `m`, `foo` takes the same number of steps since there are no conditionals and no returns along different paths of the code. So the worst case is simply what gets executed everytime: everything. We'll start with the outer loop. The initialization of `i` gets executed once, the guard check gets executed n+1 times, and the increment of `i` gets executed n times. The loop body runs n times, so whatever the cost of the inner loop is, we will multiply that by n (once for each iteration of the outer loop). For each iteration in the outer loop, we now consider the inner loop. The initialization executes once, the guard check executes $m + 1$ times, and the increment executes m times. The loop body executes m times, and the print statement inside the loop takes one step. Thus, the total number of steps for `foo` is $1+(n+1)+n+n(1+(m+1)+m+m(1)) = 3mn+4n+2$.

What if `foo` was slightly modified so that it only took one parameter, `n`, and used it for the guard of both loops? The total number of steps of this modified version (let's call it `foo2`) would then be $3n^2 + 4n + 2$.

Let's take a look at another example:

```
public void goo() {
  for (int i=0; i < 100; i++)
    for (int j=0; j < 100; j++)
      if (i >= j)
```

```
            System.out.println("hello");
  }
```

There are no parameters to goo, so this method will take the same number of steps every time it is executed and will not be in terms of any variables; it will be some number. For the i loop, initialization takes one step, checking the guard takes 101 steps, and the increment takes 100 steps. The loop body of the outer loop runs 100 times, so we will analyze the loop body and multiple its number of steps by 100. For the j loop, the analysis is the same: 1, 101, and 100. Inside the body of the j loop, the guard for the if always gets checked, requiring one step each time.

Counting how many times the print statement inside the if is a little complicated, though, since it does not always get executed. In particular, it only gets executed if i≥j. Let's take a look at when i is 0 and count how many values of j are less than or equal to 0. There is just one such value, when j is 0. How about when i is 1? There are two such values, when j is 0 and 1. For the last value of i (99), the number of j's less than or equal to i is 100. In general, for any i, there are (i+1) values of j that are less than or equal to i – these values are 0 through i.

We can now put all this together to get the total number of steps:

$$\begin{aligned}
& 1+101+100+100(1+101+100+100(1))+(1+2+3+\cdots+99+100) \\
= \;& 202+100(302)+\textstyle\sum_{i=1}^{100} i \\
= \;& 30402+50(101) \\
= \;& 35452
\end{aligned}$$

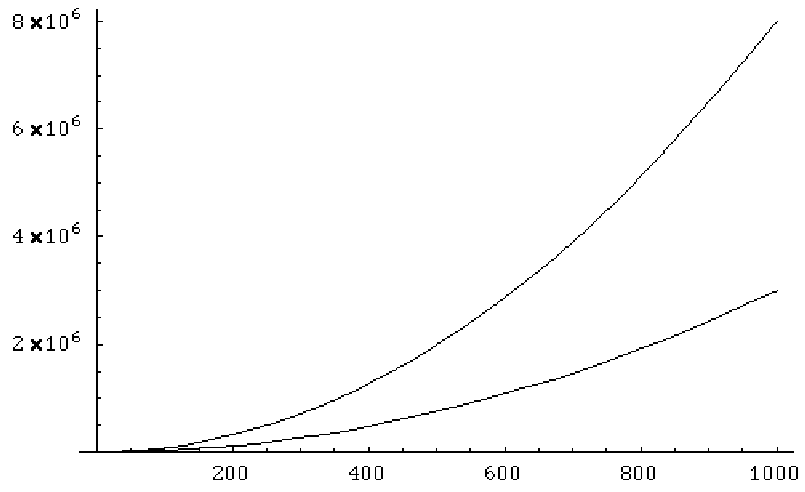
Notice that we counted the number of times the print statement executes outside of the computation for the number of steps of the j loop body. In the way we arranged the sums, the loop body executes 100 times and we multiplied that by 1 for the step need for checking the conditional. Because the number of times the print statement executes is not the same for each j, we cannot simply count in the loop body. So we count all of the print statements separately and add it to the rest of analysis.

The way we computed the sum from 1 to 100 is using a method commonly known as Gauss's trick. We can regroup the summands together in the following way: $(1+100)+(2+99)+(3+98)+\cdots+(49+52)+(50+51)$. Each of the sums in parentheses is 101 and there are 50 such terms. Thus, the sum from 1 to 100 is $50 \times 101 = 5050$. This method extends to the general case of summing the first n integers:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

If we modify goo so that it takes a parameter n and uses it for the bounds of both loops instead of 100 (let's call this version goo2), how many steps would it take? Work through the analysis and verify that it is $7n^2/2 + 9n/2 + 2$.

We have seen two methods, foo2 and goo2, that take more than n^2 steps to run. foo2 takes $3n^2 + 4n + 2$ steps and goo2 takes $3.5n^2 + 4.5n + 2$ steps. Thus, we might argue that foo2 takes much less time than goo2 to run. Let's plot these two running times and see how they compare:



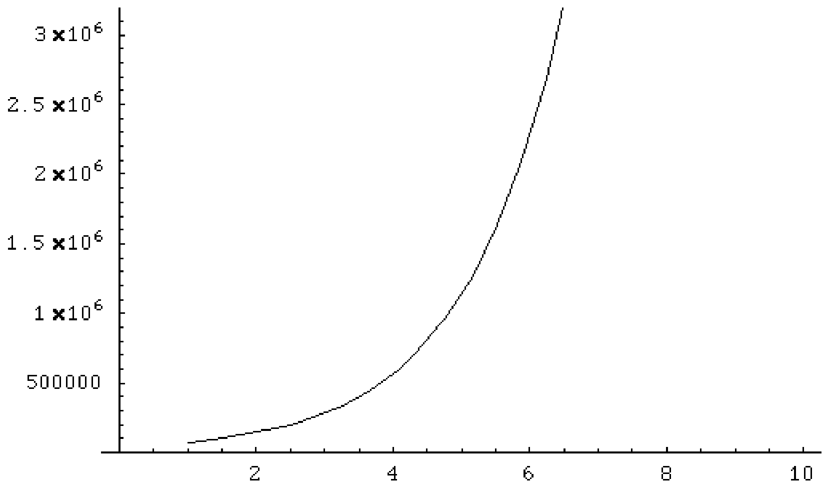
Indeed, it is apparent that `goo2` is slower than `foo2` and that the gap between them increases as the input size gets larger. Let's take a look at another method and re-evaluate our stance on this issue.

To analyze the next method, `hoo`, we introduce one more rule to our simplified approach to counting steps: if a method `g` is called from a method `f`, in the analysis of `f`, we count the number of steps for the method call to be the number of steps that `g` takes to complete in the worst case, not just one like for all other statements. With this in mind, we can analyze `hoo`:

```
public void hoo(int n) {
  for (int i=0; i < (int)Math.pow(2, n); i++) {
    System.out.println("about to call goo");
    goo();
  }
}
```

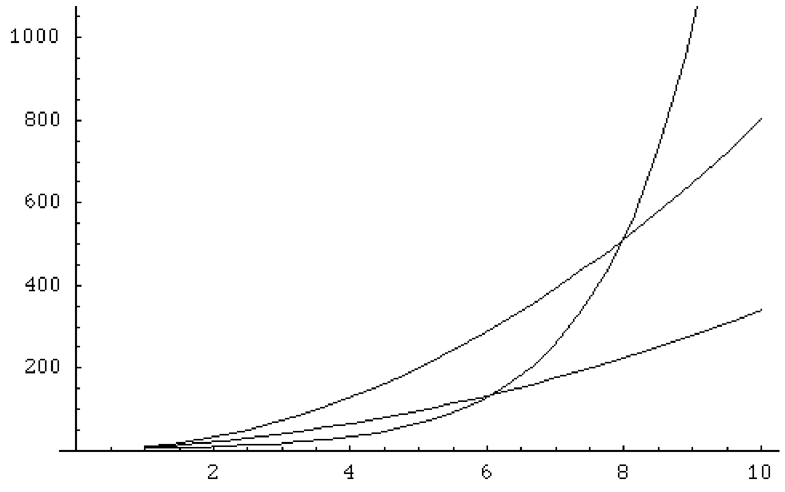
(Note that we are already breaking the rule we just created by counting the call to `Math.pow` as a constant operation.) The initialization of `i` takes one step, the guard is checked $2^n + 1$ times, the increment executes 2^n times, and the loop body runs 2^n times. Each iteration of the loop body takes $1 + 35452 = 35453$ steps. Thus, `hoo` takes $1 + (2^n + 1) + 2^n + 2^n(35453) = 35455(2^n) + 2$ steps.

Let's now plot the running time of `foo2`, `goo2`, and `hoo` on the same graph:



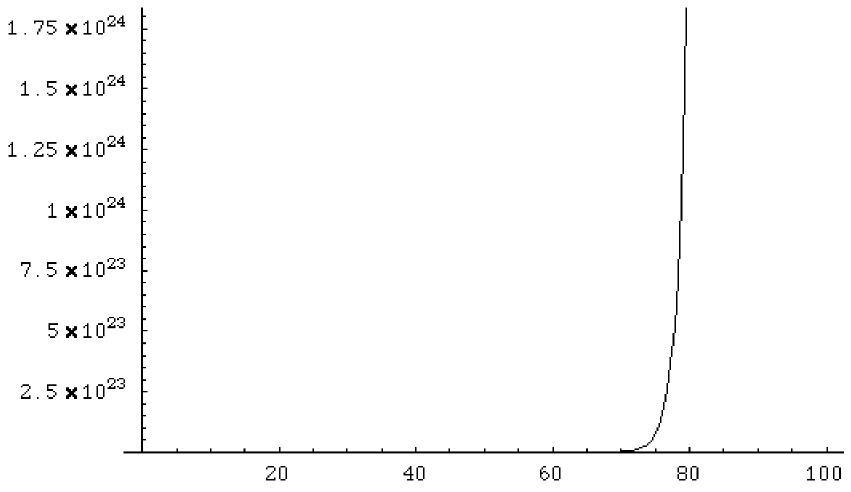
Notice that even for the range $n = 1$ to 10, `hoo`'s running time is huge and that of `foo2` and `goo2` are not even visible because their values are so small! We previously concluded that `goo2` took a lot longer than `foo2`, but compared to `hoo`, they are both trivial – `hoo` completely dwarfs them.

What if we modify `hoo` (let's call it `hoo2`) so that it doesn't call `goo` inside the loop (because that is adding a huge number of operations)? If we get rid of the call to `goo` and keep just the print statement, `hoo2` takes $3(2^n) + 2$ steps to execute. Let's now compare `foo2`, `goo2`, and `hoo2`:



Now we can at least see all three functions on the same plot, although for small values of n. And notice that for very small values of n, `hoo2` is actually faster than the others.

Let's consider a bigger range of n values:



Again, we are at the point when `hoo2` completely dwarfs the running times of the other two around, and $n = 80$ is not even a large input size for practical purposes.

We are now going to rethink the way we've been analyzing running times. The simplified model we have been using counts all operations as requiring the same amount of time even though it is clear that some operations (like multiplying floating point numbers) take much more time than others (like ANDing two bits). Furthermore, we assumed that no matter how big an expression is, it takes the same time to evaluate as any other expression. After making these somewhat absurd simplifications, we then meticulously counted steps for various methods. We then used these results to, for example, say that `goo2` is much slower than `foo2` because the coefficients multiplying the n^2 and n terms were larger (even though we are aware that the actual constants were calculated only our simplifying assumptions).

In addition, the striking difference in running time we observed between functions that take on the order of 2^n steps compared to n^2 steps gives us the feeling that the actual constants in front of these terms don't matter too much. The running times of `foo2` and `goo2` are effectively the same when compared to something that takes orders of magnitude more time, like `hoo2`.

To accomodate these insights, we improve our analysis techniques by making two assumptions:

1. The constants in front of the variables in our running time don't matter. For example, an algorithm that takes $500n$ steps is effectively just as good as one that takes $100n$ steps, or even n steps.

2. What we really care about is how algorithms perform for very large inputs.

Incorporating these two assumptions into our analysis leads us to consider just what happens to running times as the input sizes get large. In other words, what we care about is how these running times grow over time, so we call this approach *asymptotic analysis.*

We now formally define how to asymptotically compare two functions $f(n)$ and $g(n)$. We say $f(n)$ is $O(g(n))$ (read $f(n)$ is Big-O of $g(n)$) if there exists some integer constants $c > 0$ and $N > 0$ such that for all values of $n \geq N$ the following is true:

$$f(n) \leq c \times g(n)$$

If $f(n)$ is $O(g(n))$, the intuition is that f is as least as small as g, for large values of n. Said another way, the growth of f will not become larger than the growth of g.

Let's compute the asymptotic running times for `foo2`, `goo2`, and `hoo2` to see how this works. Let $T_1(n)$, $T_2(n)$, and $T_3(n)$ denote the running times of these functions, respectively. Our first claim is that $T_1(n)$ is $O(n^2)$. To prove this, we need to find values for c and N such that for all $n \geq N$

$$3n^2 + 4n + 2 \leq cn^2$$

There are many values for c and N that would satisfy these constraints; we need to find just one such pair to prove the claim. Let's start by trying $c = 4$. With this value of c, we have:

$$\begin{array}{rcl} 3n^2 + 4n + 2 & \leq & 4n^2 \\ 4n + 2 & \leq & n^2 \end{array}$$

So we now to need to find an N such that this equality is true for all $n \geq N$. Notice that when $n = 5$, $n^2 = 25$ and $4n + 2 = 22$. For all values of n greater than or equal to 5, the n^2 term is bigger than $4n + 2$. So we can pick $N = 5$. This finishes the proof that the running time of `foo2` is $O(n^2)$.

Let's now prove that $T_2(n)$ is also $O(n^2)$. To prove this, we need to find values for c and N such that for all $n \geq N$

$$3.5n^2 + 4.5n + 2 \leq cn^2$$

Let's start by trying $c = 4$ again. The equality then becomes:

$$\begin{array}{rcl} 3.5n^2 + 4.5n + 2 & \leq & 4n^2 \\ 4.5n + 2 & \leq & 0.5n^2 \\ 9n + 4 & \leq & n^2 \end{array}$$

We realize that this last equality holds for all $n \geq 10$, so we pick $N = 10$. Ths finishes the proof that the running time `goo2` is also $O(n^2)$.

Let's now prove that $T_3(n)$ is $O(2^n)$. To prove this, we need to find values for c and N such that for all $n \geq N$

$$3(2^n) + 2 \leq c2^n$$

Let's start by trying $c = 4$ again. The inequality then becomes:

$$\begin{array}{rcl} 3(2^n) + 2 & \leq & 4(2^n) \\ 2 & \leq & 2^n \end{array}$$

This last is equality is true for $n = 1$ and greater, so we pick $N = 1$. This finishes the proof that the `hoo2` runs in $O(2^n)$ time.

What about the running time `goo`, which was a constant function, not one in terms of any variables? It is easy to show that every constant function is $O(1)$. Given an arbitrary constant function $f(n)$ with value k (ie, $f(n) = k$ for all values of n), we need to show that for some c and N, the following is true for all $n \geq N$:

$$k \leq c \times 1$$

We can satisfy these constraints by choosing $c = k$ and $N = 1$. Thus, any constant function is $O(1)$. In particular, the running time of `goo` is $O(1)$.

Because Big-O analysis discards any constants along the way, when counting the number of steps an algorithm takes, we don't have to keep track of the exact number of steps. We can instead use variables to denote constants, because these constants can be thrown away in the asymptotic analysis.

Let's take a second look at `foo2` with this in mind. The outer loop takes $2n + c'$ steps, where c' accounts for constant operations like initialization. The inner loop takes $2n+c''$ steps using similar reasoning. The print statement takes c''' steps, because it is also some constant number of operations. Thus, the total number of steps is $2n + c' + n(2n + c'' + n(c''')) = 2n^2 + 2n + c' + c''n + c'''n^2$. We now want to prove that this function is $O(n^2)$. We need to find values c and N such that for all $n \geq N$, the following is true:

$$2n^2 + 2n + c' + c''n + c'''n \leq cn^2$$

If we set $c = 2 + 2 + c' + c'' + c'''$ we have:

$$\begin{array}{rcl} 2n^2 + 2n + c' + c''n + c'''n & \leq & (2 + 2 + c' + c'' + c''')n^2 \\ & = & 2n^2 + 2n^2 + c'n^2 + c''n^2 + c'''n^2 \end{array}$$

Since each term on the right is greater than or equal to each corresponding term on the left, this inequality is always true, so we pick $N = 1$. So even though we used variables to represent constant numbers in the initial analysis, we still proved the function to be $O(n^2)$ by using those variables in picking values for c and N.

It's important to realize that a Big-O relationship identifies an upper bound on the asymptotic behavior of a function. This upper bound is not guaranteed to be tight, however. For example, we could also easily prove that $T_1(n)$ is $O(n^3)$. But we know this is not the tightest bound for $T_1(n)$, because we have already shown that $T_1(n)$ is $O(n^2)$. Even worse a bound for $T_1(n)$ is $O(2^n)$. This is also easy to show, but this is a terrible bound because we know that the magnitude of its values are actually much smaller than this.

When we establish Big-O bounds, we are usually interested in finding as tight a bound as we can, but there is no guarantee that these bounds actually are. To address this, there are other asymptotic analyses that establish lower bounds for functions. Big-Omega analysis establishes lower bounds in the analagous way that Big-O establishes upper bounds, but we will not pursue these lower bounds here. Instead, we will try to establish the tighester upper bounds we can, but we'll keep in mind that they may not always be the best possible ones.

To conclude this section, let's derive upper bounds on the running times of `linearSearch` and `binarySearch` from the previous section. The number of steps that `linearSearch` takes in the worst case is $c' + 2n + n(c'') = (2 + c'')n + c'$. We show that this is $O(n)$ by picking $c = 2 + c' + c''$ and $N = 1$. The number of steps that `binarySearch` takes is $c' + (\log(n) + c'')(c''')$. Notice that we count the number of iterations of the loop as $\log(n) + c''$. As discussed in the previous section, the exact number is a little difficult to compute when n is not a power of 2, but we can completely avoid this problem by adding the constant c''. No matter how many iterations it actually will take, it will always be a constant number of iterations – something like one more or one less than – away from $\log(n)$. To show that this running time is $O(\log(n))$, we pick $c = c' + c''' + c''c'''$ and $N = 1$.

In this section, we have seen how to quantify the running time of algorithms for large inputs. This approach allows us to compare algorithms without knowing how low-level operations are implemented in the algorithm and in the computer that runs them. Therefore, an $O(n^2)$ algorithm on one computer is an $O(n^2)$ algorithm on another computer, so this analysis allows us to make very strong claims about running times independent of the environment. For most purposes, the asymptotic running time of an algorithm is the primary factor in whether or not the algorithm can be used for practical purposes. On the other hand, low-level details about how individual operations are carried out and what the constants for an algorithm is are important when trying to aggressively optimize an algorithm for a specific task on a specific machine. This is a different problem, though, and not the one that asymptotic analysis is designed to address and solve.

7.1 Exercises

1. Suppose the guard for the if statement in `goo` was `true` instead of `i>=j`. How many steps would `goo` take?
2. Suppose `goo` took two parameters, `n` and `m`, and used `n` for the guard of the outer loop and `m` for the guard of the inner loop. How many steps does this modified version (let's call it `goo3`) take if `n<m`? if `n>m`?
3. Is it correct to say $O(\log n)$ when we really mean log base-2? Show that:
   - $\log_2 n$ is $O(\log_{10} n)$
   - $\log_{10} n$ is $O(\log_2 n)$
4. Order the following functions in order of increasing asymptotic growth rate:
   - 2^n
   - n
   - 4
   - $\sqrt{n}$
   - $n^2 + 3n$
   - $\log n$

- $n \log n$
- $n!$
- $n^5 - n^4$
- 1.5^n

5. Given two algorithms that accomplish the same task, would you ever choose the algorithm with the faster asymptotic growth rate? Why or why not?

Chapter 8

Recurrence relations

So far we have only considered asymptotic growth rates for iterative algorithms. We will now look at how to analyze recursive algorithms. In this section, we will look at recursive implementations of linear and binary search and perform Big-O analyses of them.

Although a programmer would not likely think of linear search as a recursive algorithm, it can easily be programmed as one. Here is a Java implementation of linear search using recursion:

```
public int recLinearSearch(int x, int[] data) {
  return recLinearSearchAux(x, data, 0);
}
public int recLinearSearchAux(int x, int[] data, int i) {
  if (i < 0 || i >= data.length)
    return -1;
  else if (x == data[i])
    return i;
  else
    return recLinearSearchAux(x, data, i+1);
}
```

We would now like to analyze the running time. We first note that `recLinearSearchAux` is all we need to analyze, since `recLinearSearch` simply makes a call to it. Because this method calls itself, we will have to use a slightly modified approach to counting the number of steps. Let $T_1(n)$ refer to the number of steps that `recLinearSearchAux` makes when `x` is not in `data` and n is `data.length - i`. In other words, $T_1(n)$ is the worst case number of steps when there are n elements that still need to be checked.

Because we are only interested in the asymptotic running time, we do not need to be overly meticulous when counting steps. Instead of exact constants, we will use variables to represent some of the operations. Let c be the number of steps it takes to get to the point where the recursive call is made, and let c' be the number of steps when the base case (`i` $=$ 0) is reached.

We now identify the following relationships:

$$\begin{aligned}
T_1(n) &= c + T_1(n-1) \\
T_1(n-1) &= c + T_1(n-2) \\
T_1(n-2) &= c + T_1(n-3) \\
&\vdots \\
T_1(2) &= c + T_1(1) \\
T_1(1) &= c + T_1(0) \\
T_1(0) &= c'
\end{aligned}$$

We now have equations for each $T_1(k)$ in terms of the running time for smaller subproblems. T_1 is called a *recurrence relation* because each value of T_1 is defined in terms of another one. What we would like to compute is a closed-form expression for $T_1(n)$, since that represents the total running time. Notice what happens if we sum all $n+1$ of these equations together. For all $k = 0$ to $n-1$, the term $T_1(k)$ appears exactly once on the left hand side of an equation and exactly once on the right hand side of an equation. Thus, all of these terms cancel out. What we are left with is the equation $T_1(n) = cn + c'$, giving the running time for the entire algorithm. Thus, $T_1(n)$ is $O(n)$. This makes sense intuitively because the algorithm performs the same computation as the iterative version of linear search, using recursive calls instead of a loop.

In contrast to linear search, binary search is naturally thought of as a recursive process, so it is much more likely that a programmer would employ recursion in an implementation of binary search rather than linear search. Here is an implementation:

```
public int recBinarySearch(int x, int[] data) {
  return recBinarySearchAux(x, data, 0, data.length - 1);
}
public int recBinarySearchAux(int x, int[] data, int bot, int top) {
  if (bot > top)
    return -1;
  else {
    int mid = (bot + top) >>> 1;
    if (x == data[mid])
      return mid;
    else if (x < data[mid])
      return recBinarySearchAux(x, data, bot, mid-1);
    else
      return recBinarySearchAux(x, data, mid+1, top);
  }
}
```

Just as with recursive linear search, we only need to analyze the running time of `recBinarySearchAux`. Let $T_2(n)$ be the number of steps that `recBinarySearchAux` takes to run when `top - bot` $= n-1$ (ie, there are n remaining elements to search for `x`). The worst case is when either recursive call is made (the number of steps along both branches that make a recursive call are the same), and we define c to be the number of steps it takes to get to the recursive call. Let c' be the number of steps in the base case (when `bot > top`).

Because each phase eliminates about half the elements in the array, we can say that each recursive call has an input size of half the original. Although the number of remaining elements is not exactly half, we can assume that is because we will only be interested in the asymptotic growth. We identify the following relationships:

$$\begin{aligned}
T_2(n) &= c + T_2(n/2)\\
T_2(n/2) &= c + T_2(n/4)\\
T_2(n/4) &= c + T_2(n/8)\\
&\vdots\\
T_2(2) &= c + T_2(1)\\
T_2(1) &= c + T_2(0)\\
T_2(0) &= c'
\end{aligned}$$

Since dividing n in half $\log n$ times yields 1, when we sum these equations we are left with $T_2(n) = c \log n + c'$. Thus, recursive binary search is $O(\log n)$.

So far when analyzing algorithms, we have only been concerned with the running time, or the number of steps required to execute. Analyzing the memory usage of algorithms is also important, especially when dealing with large data sets. Memory usage becomes an even more important factor with recursive algorithms because each method call requires memory to keep tracking of parameters and local variables defined within each method. We will not, however, pursue this kind of analysis in this section.

8.1 Exercises

1. In a previous section, we ran an experiment to calculate the largest number of comparisons that could be made by a binary search for various sizes of input arrays. Now write a recursive method to calculate this worst case number of comparisons for binary search. Analyze the asymptotic growth rate of your algorithm.

2. *Fibonacci numbers.* This famous sequence of numbers is defined recursively, or inductively, as follows:

$$F_n = \begin{cases} 1 & \text{if } n \in \{1,2\} \\ F_{n-1} + F_{n-2} & \text{if } n \geq 3 \end{cases}$$

   (a) Write an annotated version of the recursive algorithm to compute the nth Fibonacci number. This modified version should count the number of times that the method is called. Plot these counts as a function of n.

   (b) A rigorous analysis shows that the recursive version of Fibonacci has an asymptotic growth rate, making it unusable for practical purposes.

   Consider the following non-recursive implementation of Fibonacci. What is its running time?

```
public int fib2(int n) {
  if (n <= 2)
    return 1;
  int[] cache = new int[n+1];
  // cache[0] unused
  cache[1] = 1;
  cache[2] = 1;
  for (int i=3; i <= n; i++)
    cache[i] = cache[i-1] + cache[i-2];
  return cache[n];
}
```

(c) Can you write an even better implementation than `fib2`, one that does not require an array?

3. (a) What is the running time of `mystery`?

```
public void mystery(int n) {
  if (n == 0)
    return;
  mystery(n-1);
  for (int i=0; i < n; i++)
    System.out.println("hello");
}
```

(b) What is the running time if the recursive call is changed to `mystery(n-2)`?

Chapter 9

Sorting

When we examined searching algorithms in a previous section, we made the assumption that the data array was already in sorted order (in particular, increasing sorted order). The binary search algorithm exploited this property to achieve a very fast running time. When searching for an item in an unsorted list, however, linear search is the best we can do. With large lists, linear search becomes a slow operation, so we would like to design algorithms that sort data, and sort them quickly, so that we can exploit the nice properties of sorted data in other computational taks, like searching.

The first algorithm we will look at is a very natural one. In one pass through the data, we find the smallest element and put it into the first position of the array. In the second pass, we find the second smallest element and put it into the second position of the array. We repeat this process until we have all elements in the correct sorted order. Because in each pass we select the minimum remaining element, this algorithm is called *selection sort*. Here is a concrete Java implementation:

```
public static void selectionSort(int[] data) {
  int
    n = data.length,
    minIndex,
    minValue,
    temp;

  for (int i=0; i < n-1; i++) {
    minValue = data[i];
    minIndex = i;
    for (int j = i+1; j < n; j++) {
      if (data[j] < minValue) {
        minValue = data[j];
        minIndex = j;
      }
    }
    temp = data[i];
    data[i] = data[minIndex];
```

```
      data[minIndex] = temp;
    }
  }
```

We will now analyze the running time of `selectionSort`. The initialization steps before the loop take some constant number of operations, as do the steps within in the inner-loop. So the asymptotic running time depends only the number of times the loops execute. Before we analyze the loops of `selectionSort`, let's consider a simpler nested loop case:

```
  for (int i=0; i < n; i++)
    for (int j=0; j < n; j++)
      // DO SOMETHING
```

The number of times that the body of the inner-loop gets executed is n^2, so this clearly runs in $O(n^2)$ time.

The loop bounds of `selectionSort` are slightly, different, however. So the question is how do the modified loop bounds effect the running time? We have already established that we just need to count the number of times each loop runs to determine the overall running time, so we'll do just that – for each each value of i, we will enumerate all the values of j:

$$\begin{array}{r|cccccc}
i=0 & 1 & 2 & 3 & \cdots & n-2 & n-1 \\
i=1 & & 2 & 3 & \cdots & n-2 & n-1 \\
i=2 & & & 3 & \cdots & n-2 & n-1 \\
\vdots & & & & \ddots & \vdots & \vdots \\
i=n-3 & & & & & n-2 & n-1 \\
i=n-2 & & & & & & n-1
\end{array}$$

Summing up the number of j indices for each i, we have that the inner-loop body executes $(n-1)+(n-2)+\cdots+2+1=\sum_{i=1}^{n-1} i = n(n-1)/2$ times. So even though each loop's bounds do not range over all n possible values, the total number of iterations is still $O(n^2)$. Thus, we have that `selectionSort` runs in $O(n^2)$ time.

Another simple sorting scheme called *bubble sort* also moves one element to its correct position at the end of each pass. Instead of finding the minimum (or maximum) element before any swapping, bubble sort performs swaps between each pair of indices, "bubbling" the appropriate element to the end of the array. The algorithm we present here pushes larger element from the front to the back of the array. Sometimes this approach is called *brick sort*, while bubble sort refers to pushing smaller elements from the back of the array to the front. We will not observe this naming distinction here.

Here's a concrete implementation of bubble sort:

```
  public static void bubbleSort(int[] data) {
    int temp;
    int n = data.length;
    for (int i=n-1; i >= 0; i--) {
      for (int j=0; j < i; j++) {
        if (data[j] > data[j+1]) {
          temp = data[j];
          data[j] = data[j+1];
          data[j+1] = temp;
        }
      }
    }
  }
```

Note that unlike with selection sort's one swap per outer-loop iteration, many swaps can (and often are) performed in an iteration of the `i`-loop. This process incrementally pushes larger elements to the end of the array, and gets one element in the absolutely correct spot in each pass. The loop bounds for bubble sort are again $O(n)$ for each, so bubble sort runs in $O(n^2)$ time.

Bubble sort unnecessarily continues processing even when if the array becomes sorted without requiring all n iterations of the outer loop. A simple optimization cures this:

```
  public static void bubbleSort2(int[] data) {
    int temp;
    int n = data.length;
    boolean swapped = true;
    for (int i=n-1; i >= 0 && swapped; i--) {
      swapped = false;
      for (int j=0; j < i; j++) {
        if (data[j] > data[j+1]) {
          temp = data[j];
          data[j] = data[j+1];
          data[j+1] = temp;
          swapped = true;
        }
      }
    }
  }
```

The `swapped` flag records whether or not there was at least one swap in the previous iteration of the outer-loop. If not, the array is already in sorted order and the algorithm and can forgo the remaining iterations. As a result, the best-case input (when `data` is already sorted) now runs in $O(n)$ time, although the worst-case still takes $O(n^2)$. Note that this speedup for mostly-sorted data comes at the price of the overhead for maintaining and checked the `swapped` flag.

Although the $O(n^2)$ running time for these two sorting algorithms does not seem excessively large (especially after having previously considered some very slow exponential functions), this quadratic bound becomes a severe limitation when input sizes become large.

Fortunately, there are several sorting algorithms that achieve a much better bound. One surprisingly easy but fast algorithm is called *merge sort*. This algorithm exploits the fact that combining two already-sorted arrays into one combined sorted array is simple. Consider the following `merge` operation to perform this task:

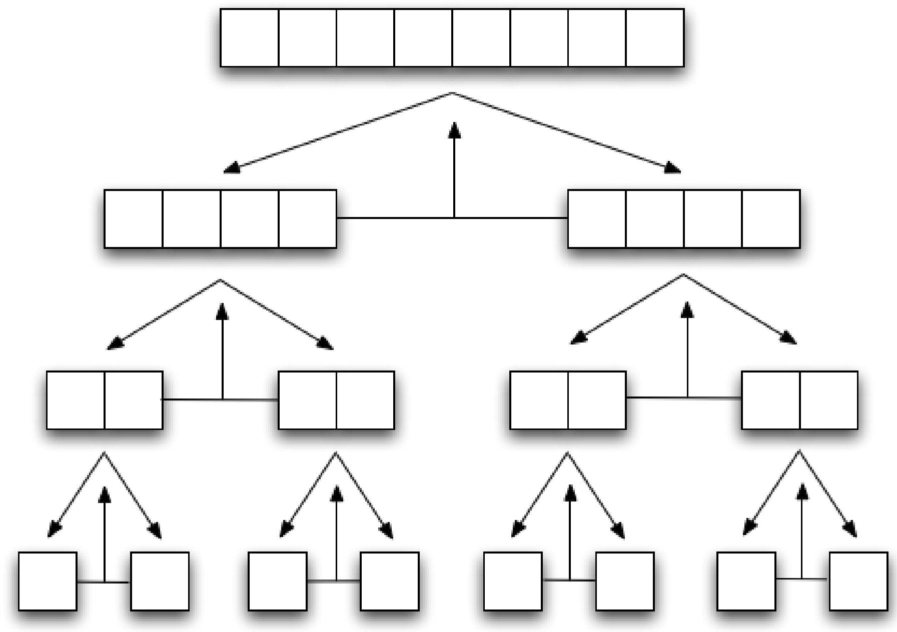
```
public static int[] merge(int[] a, int[] b) {
  int[] merged = new int[a.length + b.length];
  int i = 0, j = 0, k = 0;

  while (j < a.length || k < b.length) {
    if (j >= a.length)
      merged[i++] = b[k++];
    else if (k >= b.length)
      merged[i++] = a[j++];
    else if (a[j] < b[k])
      merged[i++] = a[j++];
    else
      merged[i++] = b[k++];
  }
  return merged;
}
```

Because each input array is already sorted, the smallest overall element is either the smallest in `a` or the smallest in `b`. After selecting this element, the next smallest is either the smallest remaining in `a` or the smallest remaining in `b`. This process continues until either of the inputs is completely consumed, in which case the remaining elements from the other input consists the remaining elements for the output array.

If we call n the number of elements in both `a` and `b`, we can see that `merge` runs in $O(n)$ time, since each element in `a` and `b` is visited exactly once and a constant number of operations are needed at each step. The linear running time of `merge` is the key component for merge sort.

Merge sort breaks up the input array into two halves, recursively sorts them, and then uses the `merge` operation to piece them together. The base case for the recursion is an array of length one, since a single element is by itself sorted. After a `merge` call is used to combine two one-element arrays, the result is a sorted two-element array that is used by its caller. This process continues up the recursion tree until the original call receives its two sorted halves, which it combines using `merge`.

In the following diagram, it is useful to think of merge sort splitting its input array on the way down the tree, and `merge` combining sorted arrays along the way up the recursion tree.



Here is a concrete implementation of merge sort that makes use of the `merge` method above:

```
public static int[] mergeSort(int[] data) {
  if (data.length < 2)
    return data;

  int n = data.length;
  int half = n / 2;
  int[] left = new int[half];
  int[] right = new int[n - half];

  for (int i=0; i < n; i++) {
    if (i < half)
      left[i] = data[i];
    else
      right[i - half] = data[i];
  }
  return merge(mergeSort(left), mergeSort(right));
}
```

We would now like to obtain a bound for `mergeSort`. The amount of work before the loop can be bounded by some constant c_1. Let the number of steps for the loop to run be c_2n. The return statement is the most interesting part, because it contains two recursive calls and a call to `merge`. Thus, there are three things to count here: the two recursive calls each with a size of about half the original input ($T(n/2)$), and linear time `merge` operation on the entire input size ($c_3 n$). Thus, the recurrence is $T(n) = c_1 + c_2 n + 2T(n/2) + c_3 n$. Notice that since the number of recursive calls is more than one, the intermediate $T(k)$ terms will not simply cancel if we add up every equation as we have done before. Instead, we will have to scale each equation to cancel out the additional terms. In particular, each successive equation will have to be scaled by an additional power of 2. This results in the following system:

$$\begin{array}{rcccccccc}
T(n) & = & c_1 & + & c_2 n & + & 2T(n/2) & + & c_3 n \\
2T(n/2) & = & 2c_1 & + & 2c_2(n/2) & + & 4T(n/4) & + & 2c_3(n/2) \\
4T(n/4) & = & 4c_1 & + & 4c_2(n/4) & + & 8T(n/8) & + & 4c_3(n/4) \\
 & \vdots & & & & & & & \\
(n/2)T(2) & = & (n/2)c_1 & + & (n/2)c_2(2) & + & nT(1) & + & (n/2)c_3(2) \\
nT(1) & = & 1 & & & & & &
\end{array}$$

These equations simplify to:

$$\begin{array}{rcccccccc}
T(n) & = & c_1 & + & c_2 n & + & c_3 n & + & 2T(n/2) \\
2T(n/2) & = & 2c_1 & + & c_2 n & + & c_3 n & + & 4T(n/4) \\
4T(n/4) & = & 4c_1 & + & c_2 n & + & c_3 n & + & 8T(n/8) \\
 & \vdots & & & & & & & \\
(n/2)T(2) & = & (n/2)c_1 & + & c_2 n & + & c_3 n & + & nT(1) \\
nT(1) & = & 1 & & & & & & \\
\hline
T(n) & = & c_1(1+2+\cdots+n/2) & + & (\log n - 1)c_2 n & + & (\log n - 1)c_3 n & + & 1
\end{array}$$

The first term in the sum has less than $\log n$ elements, and each is less than n. Thus, the term can be bounded by $n \log n$. We now have

$$\begin{aligned}
T(n) &\leq n\log n + (\log n - 1)c_2 n + (\log n - 1)c_3 n + 1 \\
&= n\log n + c_2 n\log n + c_3 n \log n - c_2 n - c_3 n + 1
\end{aligned}$$

Thus, we have that `mergeSort` runs in $O(n \log n)$ time. So merge sort is indeed much faster than the $O(n^2)$ sorting algorithms we have seen.

In fact, it can be shown that the lower bound on *any* comparison sort – a sorting algorithm that does not know anything about the range or values of the data, so it must use comparison as the only means to sort – is $O(n \log n)$. Thus, merge sort is one of several asymptotically optimal sorting algorithms.

9.1 Exercises

1. Which algorithm performs more swaps in general, `selectionSort` or `bubbleSort`?
2. The implementations of `selectionSort`, `bubbleSort`, and `bubbleSort2` all modify the input array directly, or *in place*, whereas `mergeSort` returns the sorted array as a return

value. Modify the first three sorting algorithms so they return the sorted array without modifying the input array.

3. Another type of sorting algorithm is called insertion sort. With this algorithm, at the end of each phase i, the elements in the first $i + 1$ positions are in sorted order. This is trivially true when $i = 0$. In the $i = 1$ phase, the element in position 1 is inserted either to the left or right of the element in position 0. In the $i = 2$ phase, the element in position 2 is inserted either to the left of the element in position 0, to the left of the element in position 1, or to the right of position 1. So in each phase i, the element at index i is inserted into the correct place within the first i elements. When inserting this element in the correct position, several elements in the array have to be shifted over to the right.

   Implement insertion sort and analyze its running time.

4. Create annotated versions of each sorting algorithm in this section. Design and run experiments to compare how they perform in practice.

Chapter 10

Quizzes

10.1 Quiz 1 – July 13, 2007

1. For each 8-bit string, compute its decimal value in UM, SM, and TC:
   - `0110 0001`
   - `1001 1000`
   - `1000 0000`
2. For each decimal number, compute its 4-bit binary representation in SM and TC:
   - `-1`
   - `-5`
3. - What do the following 32 bits represent as a `float`?

     `1 11111111 00000000000000000000000`
   - What do the following 32 bits represent as a `float`?

     `1 10000000 10100000000000000000000`
   - How is the decimal number `6.50` encoded as a `float`?
4. Consider a 32-bit `int` variable `argb` that stores the ARGB components of a pixel as we saw in class: the A component in bits 24 to 31, R in bits 16 to 23, G in bits 8 to 15, and B in bits 0 to 7. Give expressions to extract each component using bit shift operators and a decimal integer bit mask, *not* a hexadecimal one.
5. Recall that in Java decimal literals are prefixed by nothing, octal literals are prefixed by `0`, and hexadecimal literals are prefaced by `0x`.
   (a) Do `5`, `05`, and `0x5` represent the same number? If not, how do you represent 5 in octal and hex?
   (b) Do `21`, `021`, and `0x21` represent the same number? If not, how do you represent 21 in octal and hex?
6. A collection of four bits is sometimes called a *nibble* because it is half the size of a byte. Suppose Java offered a 4-bit integer primitive type called `nibble` and that its encoding scheme was 2's complement. Also suppose Java offered a 4-bit integer primitive type called `unsigned nibble` and that its encoding scheme was unsigned magnitude.
   (a) What is the range of `unsigned nibble`?
   (b) Next to each of the following statements, write what the value of `n` is after each statement gets executed. Assume that these statements are executed in DrJava in succession.

```
> unsigned nibble n = 0

> n += 3
> n += 3

> n += 3

> n += 3

> n += 3

> n += 3
```

(c) What is the range of `nibble`?

(d) Next to each of the following statements, write what the value of `n` is after each statement gets executed. Assume that these statements are executed in DrJava in succession.

```
> nibble n = 0

> n += 3

> n += 3

> n += 3

> n += 3

> n += 3

> n += 3
```

7. In addition to primitive types for single-precision floating-point (`float`) and double-precision floating-point (`double`), suppose that Java offered a "half-precision" floating-point type called `half`. Suppose a `half` is 16 bits – 1 for sign, 6 for exponent, and 9 for fraction – and its encoding scheme is analagous to `float` and `double`.

   (a) For what range of exponents e is the value interpreted as a normalized number? Recall that normalized here means there is exactly one 1 to the left of the binary point.

   (b) What is the bias of the exponent? In other words, what value is subtracted from the actual value in the exponent bits?

   (c) What do the following 16 bits represent as a `half`?

   `1 100011 000100000`

8. Consider the following two encoding schemes. For each, can zero and all positive numbers be represented? If so, can they be represented uniquely? If not, explain why.

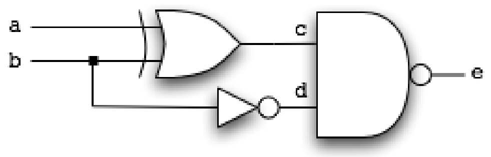
   (a) Strings of 0's, 1's, 2's, and 3's where $a_{n-1}...a_1a_0 = \sum_{i=0}^{n-1} a_i * 3^i$

(b) Strings of 0's, 1's, 2's, and 3's where $a_{n-1}...a_1a_0 = \sum_{i=0}^{n-1} a_i * 5^i$

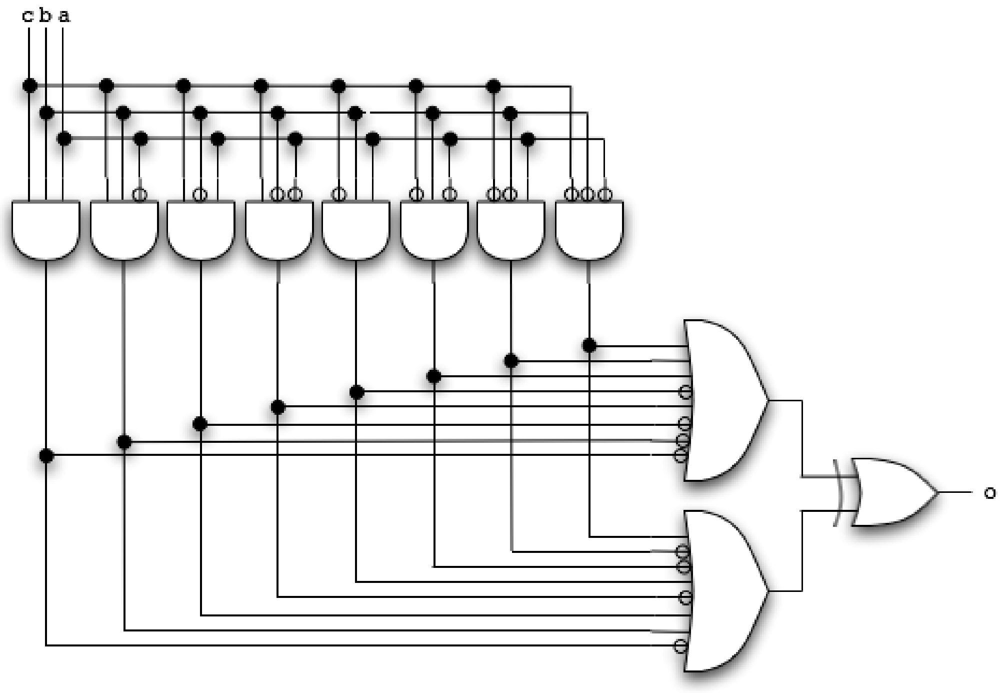
9. The system of using tally marks to count is a form of *unary notation.* Can fractions be encoded in a unary system? If so, what might the encoding scheme be? If not, why not?

10.2 Quiz 2 – July 20, 2007

1. Write out the truth table that this circuit encodes:



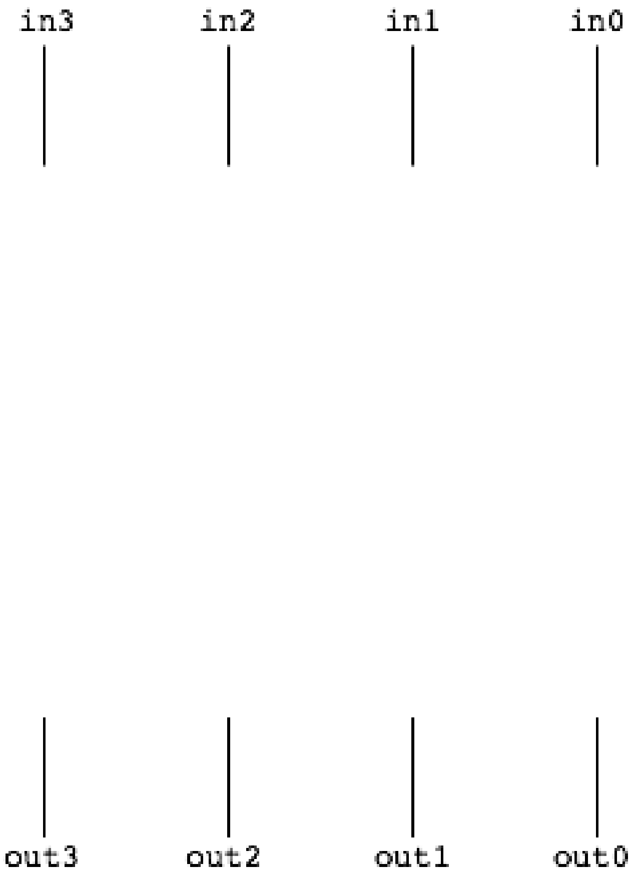
2. Write out the truth table that this circuit encodes:



3. (a) Draw a circuit that takes a 4-bit number, performs the following two operations on them, and outputs the resulting 4-bit number:

   - invert all the bits
   - add 1 to the number

   You may use boxes labeled `FA` to denote full adders and boxes labeled `HA` to denote half adders. Clearly label all wires. The input and output wires are already drawn for you.



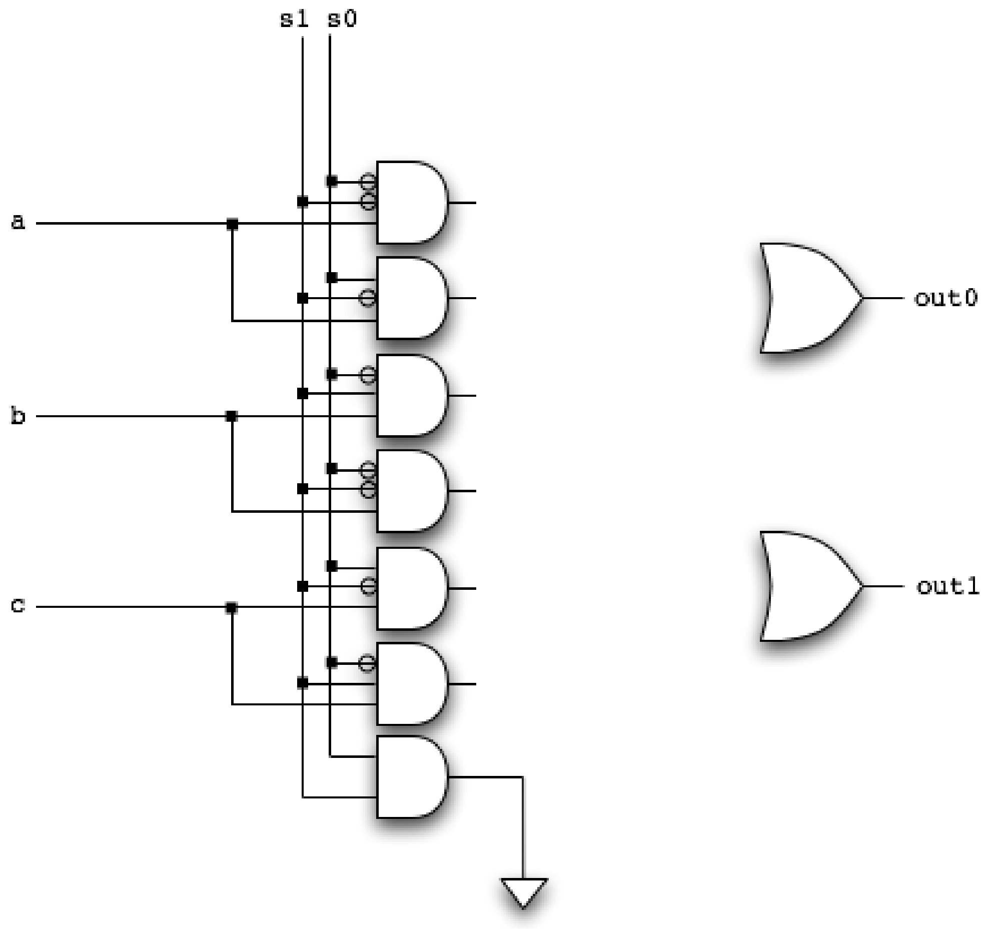
   (b) Explain what this circuit does.

4. We have seen a couple of 2-to-1 muxes, circuits that take two inputs and select one of them to output. See the Reference section for the 1-bit 2-to-1 mux.

   We would now like to build a mux that takes 3 inputs and chooses 2 of them not just one. Let's call the inputs a, b, and c and the two outputs $o1$ and $o0$. Let the following truth table summarize the behavior we would like this mux to have. Notice that there are two selector wires, $s1$ and $s0$.

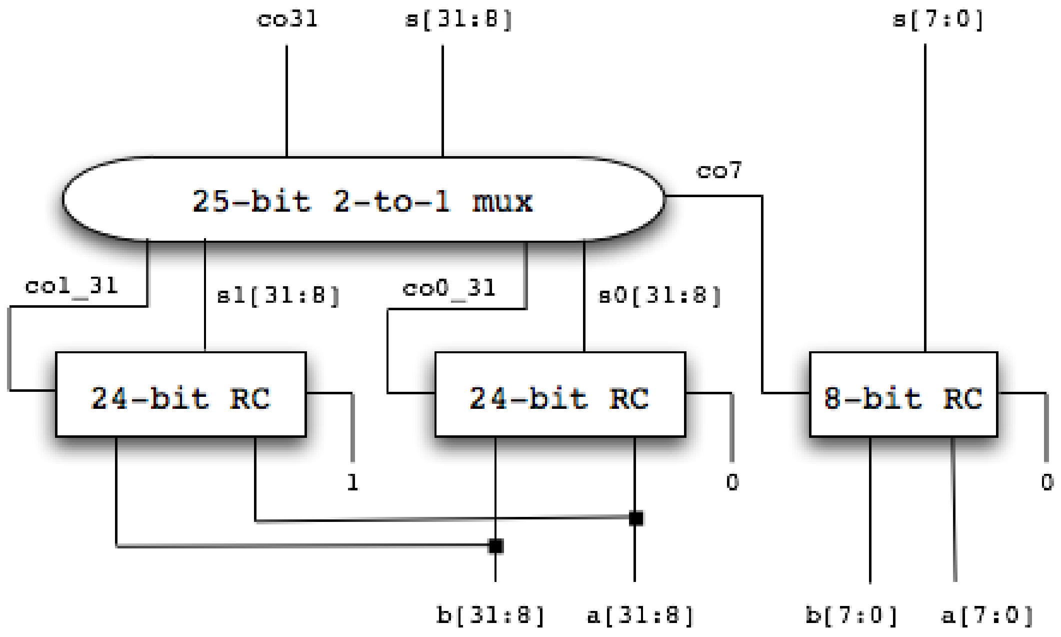
$s1$	$s0$	$o1$	$o1$
0	0	b	a
0	1	a	c
1	0	c	b

   When both selector wires are 1, the voltage is connected to ground. You do not need to do anything more for this case. In the following diagram, fill in the appropriate connecting wires to achieve the desired behavior for this 1-bit 3-to-2 mux.



5. (a) Why can we encode any boolean function using a circuit with a gate delay of 2?

   (b) Consider the following 32-bit carry-select adder in two uneven pieces. The lowest-order 8 bits are computed by an 8-bit ripple-carry adder, and the higher-order 24 bits are computed in parallel by two 24-bit ripple-carry adders. The carry-out from bit 7 is used to select the appropriate sum of the higher-order 24 bits.



   What is the gate delay of this adder?

6. Consider the following alternative implementation of linear search on a sorted array of integers with no duplicates:

```
public int backwardsLinearSearch(int x, int[] data) {
  for (int i = data.length - 1; i >= 0; i--) {
    if (x == data[i])
      return i;
  }
  return -1;
}
```

   (a) Is the best case input for `backwardsLinearSearch` the same as for `linearSearch`? If not, what is it?

   (b) Is the worst case input for `backwardsLinearSearch` the same as for `linearSearch`? If not, what is it?

   (c) Let the integer `x` be contained in the integer array `data`. When do `linearSearch` and `backwardsLinearSearch` take the exact same number of steps when `x` and `data` are passed as arguments?

7. For each of the following, write "T" if the statement is true or "F"" if the statement is false. Make sure your answers are legible.

   1. ______ x is $O(x)$
   2. ______ x is $O(\sqrt{x})$
   3. ______ 10 is $O(9)$
   4. ______ $13x^2 + 3$ is $O(0.00001x^2)$
   5. ______ $x \log x$ is $O(\log x)$

   The following statements pertain to all functions $f(x)$ and $g(x)$:

   6. ______ If $f(x)$ is $O(g(x))$, then $g(x)$ is the tightest possible upper bound of $f(x)$.
   7. ______ If $f(x)$ is $O(g(x))$, then $f(x) \leq g(x)$ for all x.
   8. ______ If $f(x)$ is $O(g(x))$, then $f(x)$ is $O(f(x) + g(x))$.
   9. ______ If $f(x)$ is $O(g(x))$, then $f(x) = g(x)$ for some x.
   10. ______ If $f(x)$ is $O(g(x))$, then $g(x)$ is not $O(f(x))$.

8. Show that $5n^2 + 2n + 70$ is $O(n^2)$. That is, find integer constants $c > 0$ and $N > 0$ such that for all $n \geq N$, the following inequality is true:

$$5n^2 + 2n + 70 \leq cn^2$$

9. Consider the following methods `f` and `g`:

```
public void f(int n) {
  for (int i = 0; i < n; i++)
    if (i % 2 == 0)
      System.out.println("hello");
}
public void g(int n) {
  for (int i = 0; i < n; i++)
    f(i);
}
```

   For the next two parts, use the model for counting steps that we have been using in lecture.

   (a) Count the number of steps that `f` takes when `n` is even? when `n` is odd? These should be functions of n.

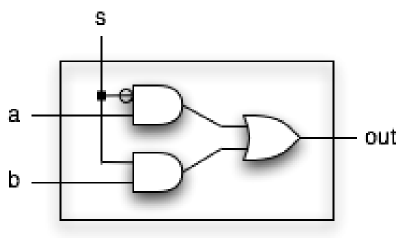
(b) Count the number of steps that `g` takes when `n` is even? when `t` is odd? These should be functions of n.

(c) Give a tight asymptotic upper bound for the running time of `f`. You do not need to prove that it is correct.

(d) Give a tight asymptotic upper bound for the running time of `g`. You do not need to prove that it is correct.

Reference

Here is a 1-bit 2-to-1 mux:



And its behavior:

s	out
0	a
1	b

10.3 Quiz 3 – July 26, 2007

1. Consider the following class definition.

```
public class RecurrenceRelations {
  private boolean callTwice = true;

  public void f(int n) {
    if (n > 1) {
      System.out.println("before recursive call");
      f(n/2);
      System.out.println("after recursive call");
    }
  }
  public void g(int n) {
    f(2*n);
  }
  public void h(int n) {
    if (n > 1) {
      if (callTwice) {
        h(n/2);
        h(n/2);
        callTwice = false;
      }
      else {
        h(n/2);
      }
    }
  }
}
```

   Assume that this is the entire class definition (so `callTwice` cannot be modified by any other code).

   For each method – `f`, `g`, and `h` – determine its worst-case asymptotic running time in terms of n and explain why.

   Note: assume that n is a power of 2 (ie, $n = 2^k$ for some k).

2. We would like to search for an integer `x` in an **unsorted** array of integers called `data`. For each of the three following algorithms, give the asymptotic running time. Then decide which of the three is the best for this task.

   (a) Linear search through `data` to look for `x`.

   (b) First sort the data using `selectionSort` and use binary search to look for `x` in this sorted array.

(c) First sort the data using `mergeSort` and use binary search to look for `x` in this sorted array.

Chapter 11

Solutions

11.1 Chapter 1 Solutions

		8 bits	min bits
25	UM	0001 1001	1 1001
25	SM	0001 1001	01 1001
25	TC	0001 1001	01 1001
43	UM	0010 1011	10 1011
-43	UM	N/A	N/A
-43	SM	1010 1011	110 1011
-43	TC	1101 0101	101 0101
99	UM	0110 0011	110 0011
-99	UM	N/A	N/A
-99	SM	1110 0011	1110 0011
-99	TC	1001 1101	1001 1101
127	UM	0111 1111	111 1111
-127	UM	N/A	N/A
-127	SM	1111 1111	1111 1111
-127	TC	1000 0001	1000 0001
128	UM	1000 0000	1000 0000
-128	UM	N/A	N/A
-128	SM	N/A	1 1000 0000
-128	TC	1000 0000	1000 0000
129	UM	1000 0001	1000 0001
-129	UM	N/A	N/A
-129	SM	N/A	1 1000 0001
-129	TC	N/A	1 0111 1111

	UM	SM	TC
0110	6	6	6
1010	10	-2	-6
1000	8	0	-8
0100 1110	78	78	78
1100 1110	206	-78	-50
1111 1111	255	-127	-1

11.2 Chapter 3 Solutions

1.

S	E	F	meaning
s	$1 \leq e \leq 2046$	f	$(-1)^s \times 1.f \times 2^{(e-1023)}$
0	0	0	zero
1	0	0	zero
s	0	f`!=`0	$(-1)^s \times 0.f \times 2^{-1022}$
0	2047	0	$+\infty$
1	2047	0	$-\infty$
s	2047	f`!=`0	`NaN`

2.

The binary representation of 57 is 11 1001 and of .8125 is 0.1101. To get 111001.1101 in normalized scientific notation, we move the binary point five places to the left and multiple by 2^5: 1.110011101×2^5. Since the bias for the exponent is 127 for a `float`, we set $e = 132$, which is 1000 0100 in binary. Since the number is positive, $s = 0$. Omitting the leading 1, the leftmost fraction bits are set to 110011101; the remaining 14 bits are 0. Thus, the 32-bit `float` representation is:

```
0 10000100 11001110100000000000000
```

To encode the same number in a `double`, we only have to re-compute the value for the exponent, since `double`s have a bias of 1023. Thus, $e = 1028$, which is 100 0000 0100. The sign and fraction bits are the same, except that we now have 43 remaining fraction bits to fill with 0's.

Calculating the binary representation of -2.3 requires a bit work, since 0.3 is not easily represented in binary. We note that the whole part (2) is represented by 10. We now calculate the representation of 0.3, without showing all the intermediate steps of multiplying equations by 2:

$$\begin{array}{rcll}
0.3 & = & b_1/2 + b_2/4 + \cdots & \\
0.6 & = & b_1 + b_2/2 + \cdots & b_1 = 0 \\
1.2 & = & b_2 + b_3/2 + \cdots & b_2 = 1 \\
0.4 & = & b_3 + b_4/2 + \cdots & b_3 = 0 \\
0.8 & = & b_4 + b_5/2 + \cdots & b_4 = 0 \\
1.6 & = & b_5 + b_6/2 + \cdots & b_5 = 1 \\
1.2 & = & b_6 + b_7/2 + \cdots & b_6 = b_2
\end{array}$$

When computing b_6, we have arrived at a left-hand value that we have seen before, 1.2. Now we know that all of the $b_i's$ that we've computed since the first time we saw 1.2 (1001) will repeat forever. So the binary representation of 0.3 $(0.0\overline{1001})$.

Now we have that 2.3 is represented as $10.0\overline{1001}$, or, in normalized scientific notation, $1.00\overline{1001} \times 2^1$. To offset the bias, we set $e = 128$. The sign bit is 1, since -2.3 is negative. The fraction bits are set to 00 and then the repeating 1001 until all 23 bits are used up. Note that because the pattern repeats forever, an infinite number of bits would be required to specify -2.3 exactly. The final `float` representation is:

```
1 1000 0000 00100110011001100110010
```

To encode the same number in a `double`, we need to re-compute the exponent. To offset the bias of 1023, we set $e = 1024$, which is 100 0000 0000. The fraction bits are again set to 00 and then 1001 until the 52 bits run out. Note that the `double`'s representation is more precise than the `float`'s, since it has more significant digits for the representation. But, of course, it still isn't exact.

3.

For a `float`, the largest positive finite number with $f = 0$ is achieved by setting $s = 0$ and $e = 254$.

$$\begin{array}{rcl}
(-1)^0 \times 1.0 \times 2^{254-127} & = & 2^{127} \\
& = & 2^7(2^{10})^{12} \\
& \approx & 2^7(10^3)^{12} \\
& = & 2^7(10^{36}) \\
& = & 128 \times 10^{36} \\
& = & 1.28 \times 10^{38}
\end{array}$$

For a `double`, the largest positive finite number with $f = 0$ is achieved by setting $s = 0$ and $e = 2046$.

$$\begin{array}{rcl}
(-1)^0 \times 1.0 \times 2^{2046-1023} & = & 2^{1023} \\
& = & 2^3(2^{10})^{102} \\
& \approx & 2^3(10^3)^{102} \\
& = & 2^3(10^{306}) \\
& = & 8 \times 10^{306}
\end{array}$$

4.

For a `float`, the largest positive finite number is achieved by setting $s = 0$, $e = 254$, and all 23 bits of f to 1. These values encode:

$$max_{\texttt{float}} = (-1)^0 \times (1 + \frac{1}{2^1} + \frac{1}{2^2} + \cdots + \frac{1}{2^{23}}) \times 2^{(254-127)}$$

We define the following constants:

$$\begin{array}{rcl} A & = & 1 + \frac{1}{2^1} + \frac{1}{2^2} + \cdots + \frac{1}{2^{23}} \\ B & = & 1 + \frac{1}{2^1} + \frac{1}{2^2} + \cdots \\ C & = & \frac{1}{2^{24}} + \frac{1}{2^{25}} + \frac{1}{2^{26}} + \cdots \end{array}$$

We have $A = B - C$, and we can compute B and C since they are infinite geometric sums. Using the formula for the infinite geometric sum B, where $a = 1$ and $r = \frac{1}{2}$, we have $B = 2$. Using the formula for C, where $a = \frac{1}{2^{24}}$ and $r = \frac{1}{2}$, we have $C = \frac{1}{2^{23}}$. Thus, $A = B - C = 2 - 2^{-23}$. Plugging in A into the equation above, we get $max_{\texttt{float}} = (2 - 2^{-23})2^{127} = 2^{128} - 2^{104}$.

To approximate this value in decimal, be begin by distributing A in the equation for $max_{\texttt{float}}$, yielding:

$$max_{\texttt{float}} = 2^{127} + 2^{126} + \cdots + 2^{104}$$

We will approximate the summands in decimal.

$$\begin{array}{rcl} 2^{127} & = & 2^7(2^{10})^{12} \\ & \approx & 2^7(10^3)^{12} \\ & = & 2^7(10^{36}) \\ & = & 128 \times 10^{36} \\ & = & 1.28 \times 10^{38} \end{array}$$

Thus, $2^{126} \approx 0.64 \times 10^{38}$, $2^{125} \approx 0.32 \times 10^{38}$, and so on. For the last summand, $2^{104} \approx (1.28/2^{22}) \times 10^{38}$. These terms have become so small that their impact on the sum is neglible, so we can go one step further and assume that this sequence continues infinitely (seen each additional term's contribution will be neglibile). With this approximation we have:

$$max_{\texttt{float}} \approx (1.28 + 0.64 + 0.32 + \cdots) \times 10^{38}$$

This infinite sum in this equation is 2.56, so we have $max_{\texttt{float}} \approx 2.56 \times 10^{38}$. Recall that we assumed $2^{1024} \approx 10^3$ to derive this result; the actual maximum value (computed using a high-precision calculator) is about 3.4×10^{38}.

To compute the largest value represented by a `double` we peform the same computations, so we will summarize the intermediate results. The largest value is realized by setting $s = 0$, $e = 2046$, and all 52 bits of f to 1. The new coefficient C has value 2^{-52} while B is still 2. Thus, $A = 2 - 2^{-52}$. This leads to $max_{\texttt{double}} = (2 - 2^{-52}) \times 2^{1023} = 2^{1024} - 2^{971}$.

We will approximate the decimal value of $max_{\texttt{double}}$ in the same fashion as before, approximating each of the summands in:

$$max_{\texttt{double}} = 2^{1024} + 2^{1023} + \cdots + 2^{972}$$

The decimal approximations for the summands are $2^{1024} \approx 8 \times 10^{306}$, $2^{1023} \approx 4 \times 10^{306}$, and so on. Again we assume that there are an infinite number of these summands because their values become neglibility small. We are left with $max_{\texttt{double}} \approx 16 \times 10^{306} = 1.6 \times 10^{307}$. The actual value is about 1.8×10^{308}. Note that our approximation error is more significant than it was for `float`.

11.3 Chapter 2 Solutions

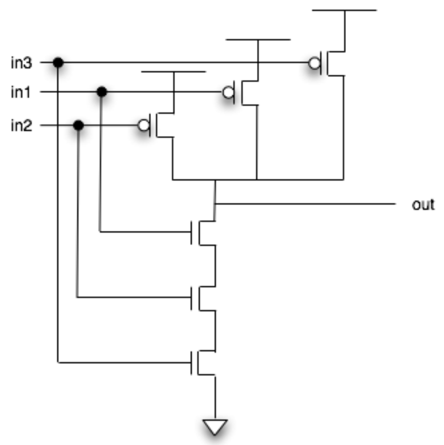
1. The value for the bit mask is still 255, since we will be interested in the lowest-order eight bits at a time. Since the octal representation of 255 is 377, the expressions we want are:
    - B is `argb & 03777`
    - G is `(argb >> 8) & 03777`
    - R is `(argb >> 16) & 03777`
    - A is `(argb >> 24) & 03777`
2. - B is `(argb & 0xff) >> 0`
    - G is `(argb & 0xff00) >> 8`
    - R is `(argb & 0xff0000) >> 16`
    - A is `(argb & 0xff000000) >>> 24`

    Note that we have to use unsigned right shift for the A component even though signed right shift works with the others. Why?
3. - 43: 10 1011
    - 84: 101 0100
    - 111: 110 1111
    - 300: 1 0010 1100
4. Hint: `0xdead`

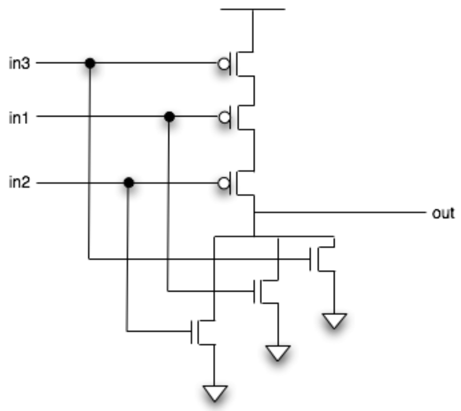
11.4 Chapter 4 Solutions

1.

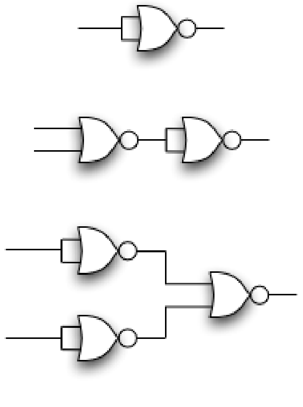
NAND:

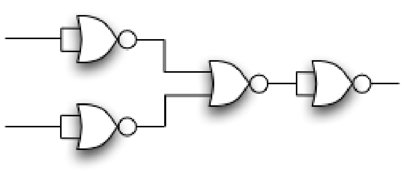


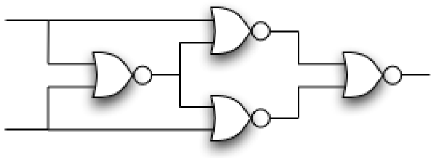
NOR:

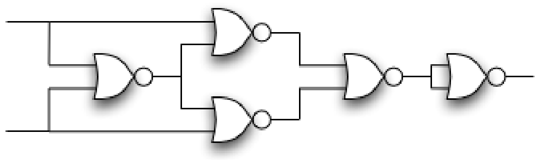


2. From top to bottom, NOT, OR, AND, NAND, XNOR, XOR:









3.

a	b	c	d
0	0	1	1
0	1	1	0
1	0	1	1
1	1	0	1

4. We can reason about the logical meaning of this circuit without examining the truth table. The first output is logically:

$$\begin{aligned} \text{XOR OR XNOR} &= \text{XOR OR (NOT XOR)} \\ &= 1 \end{aligned}$$

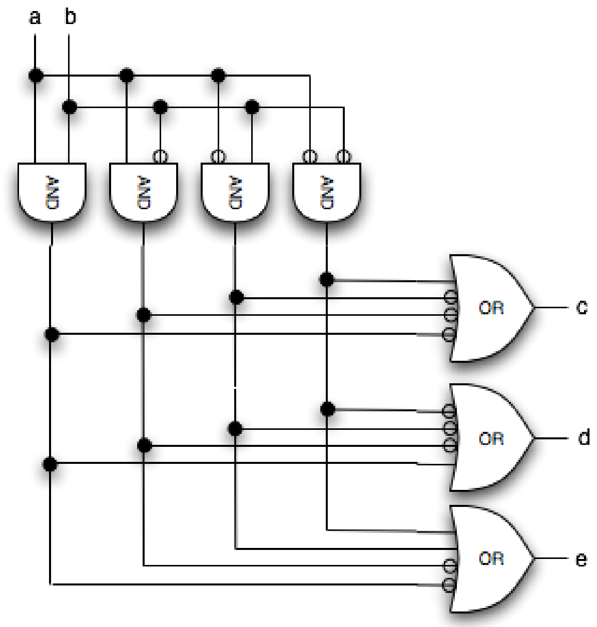
This last equality holds since for any truth statement a, either a is true or NOT a is true.

The second output is logically:

$$\begin{aligned} \text{XOR AND XNOR} &= \text{XOR AND (NOT XOR)} \\ &= 0 \end{aligned}$$

This last equality holds since for any truth statement a, exactly one of a and NOT a is true.

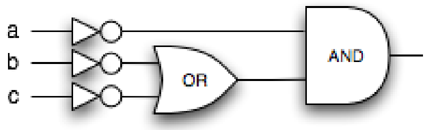
5.



6. We can use De Morgan's Laws to rewrite this logical statement:

$$\begin{aligned} \overline{a \text{ OR } (b \text{ AND } c)} &= \overline{a} \text{ AND } \overline{(b \text{ AND } c)} \\ &= \overline{a} \text{ AND } (\overline{b} \text{ OR } \overline{c}) \end{aligned}$$

This logical expression is encoded by the circuit:



11.5 Chapter 5 Solutions

1.

$n = 32$:

$$\begin{aligned} m = 1: \quad 2(32) + 2(0) &= 64 \\ m = 2: \quad 2(16) + 2(1) &= 34 \\ m = 4: \quad 2(8) + 2(3) &= 22 \\ m = 8: \quad 2(4) + 2(7) &= 22 \\ m = 16: \quad 2(2) + 2(15) &= 34 \\ m = 32: \quad 2(1) + 2(31) &= 64 \end{aligned}$$

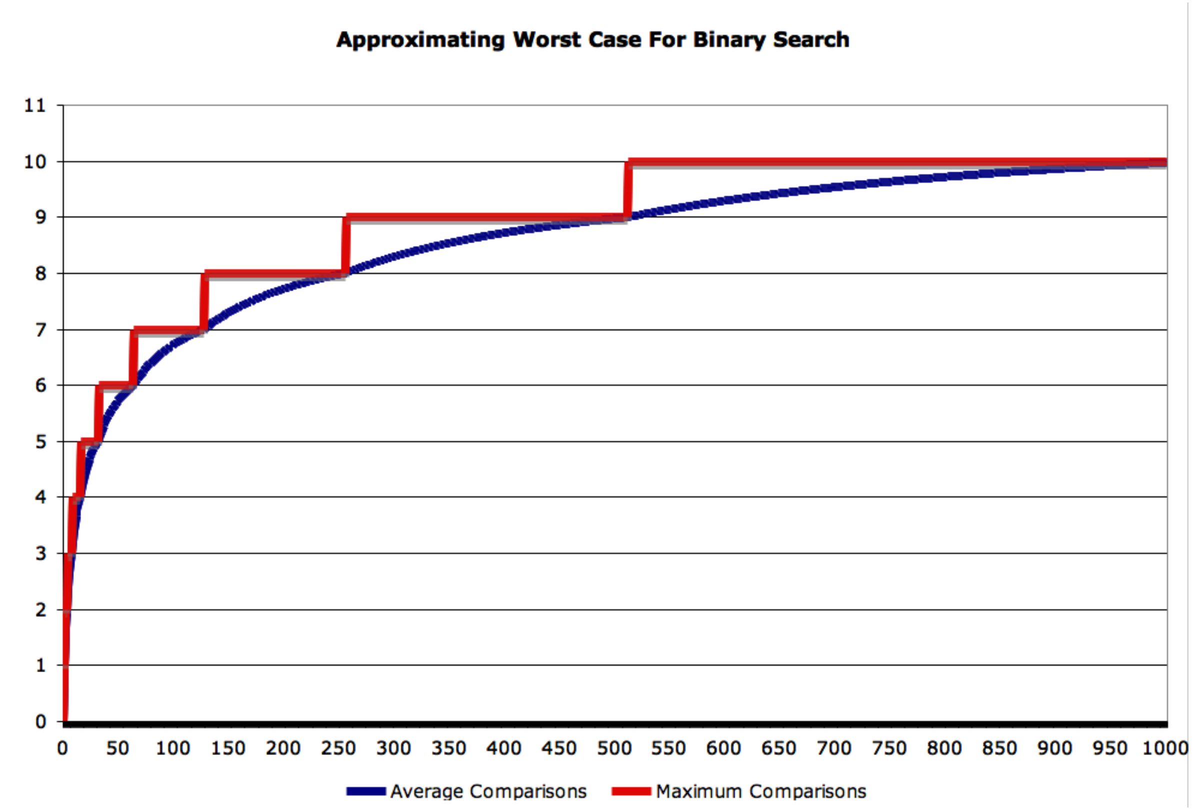
The optimal value is either 4 or 8.

$n = 64:$

$$\begin{array}{rrcl}
m = 1: & 2(64) + 2(0) & = & 128 \\
m = 2: & 2(32) + 2(1) & = & 66 \\
m = 4: & 2(16) + 2(3) & = & 38 \\
m = 8: & 2(8) + 2(7) & = & 30 \\
m = 16: & 2(4) + 2(15) & = & 38 \\
m = 32: & 2(2) + 2(31) & = & 66 \\
m = 64: & 2(1) + 2(63) & = & 128
\end{array}$$

The optimal value is 8.

11.6 Chapter 6 Solutions



11.7 Chapter 7 Solutions

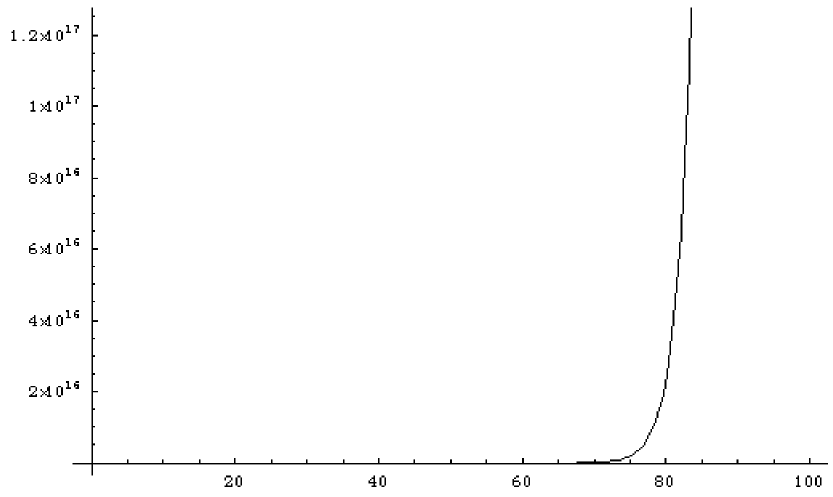
11.8 Chapter 8 Solutions

1.
```
public static int worstCase(int size) {
    if (size == 1) {
      return 1;
    }
    else if (size == 2) {
      return 2;
    }
    else {
      int half = (size-1)/2;
      if (size % 2 == 1) {
        return 1 + worstCase(half);
      }
      else {
        return 1 + worstCase(half + 1);
      }
    }
  }
```

Let c be the constant number of steps that it takes in the worst case. Thus, we can use c no matter which branch of execution is taken. When `worstCase` makes a recursive call, the running time is $T(n) = c + T(n/2)$. This recurrence relation behaves like the one we saw for `binarySearch`, so if we write out all the equations and sum them up, we are left with $T(n)$ is $O(\log n)$.

Although this is a good running time, when calling `worstCase` on a lot of inputs like all numbers from 0 to 1000 – like we would if we were trying to chart the actual number of worst-case comparisons for each input size – there are a lot of repeated calls to `worstCase` with the same input. For example, `worstCase(4)` is computed for `worstCase(9)`, `worstCase(18)`, `worstCase(37)`, and so on. Repeatedly computing these values is unnecessary; instead we can store each result in an array the first time it is computed, and subsequent requests for can simply look up the value in the array.

2. (a) There is really no reason to even annotate `fib`, since it does little more than make recursive calls. The difference in saying it takes, for example, 2 steps rather than 1 step before each call is meaningless, so we can simply look at the values of the Fibonacci numbers themselves to get an idea of the running time of the recursive method.



We can see from this plot that the Fibonacci numbers seem to have exponential growth. This can be proven using mathematical induction.

(b) There are no recursive calls and the loop makes $O(n)$ iterations with a constant number of operations in each iteration. Thus, `fib2` runs in $O(n)$ time.

(c)
```
public static int fib3(int n) {
  if (n <= 2)
    return 1;
  int a = 1, b = 1, c;
  for (int i=2; i < n; i++) {
    c = a + b;
    a = b;
    b = c;
  }
  return b;
}
```

3. (a)

$$\begin{array}{rcl} T(n) & = & T(n-1)+cn \\ T(n-1) & = & T(n-2)+c(n-1) \\ T(n-2) & = & T(n-3)+c(n-2) \\ & \vdots & \\ T(1) & = & T(0)+c(1) \\ T(0) & = & 1 \\ \hline T(n) & = & c(1+2+\cdots+(n-1)+n)+1 \\ & = & c(\sum_{i=1}^{n} i)+1 \\ & = & c(n(n+1)/2)+1 \end{array}$$

Thus, the running time is $O(n^2)$.

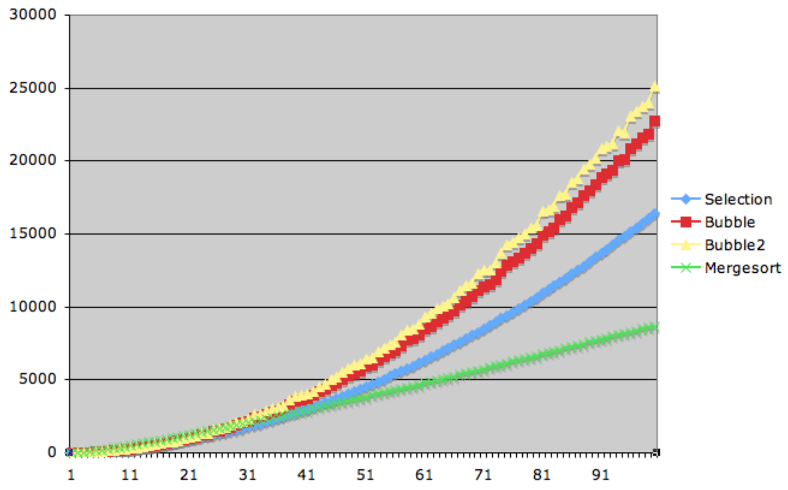
(b) We first note that if n is odd, the method would not terminate! If n is even, we identify the following recursive identities:

$$\begin{array}{rcl}
T(n) & = & T(n-2) + cn \\
T(n-2) & = & T(n-4) + c(n-2) \\
 & \vdots & \\
T(2) & = & T(0) + c(2) \\
T(0) & = & 1 \\
\hline
T(n) & = & c(2 + 4 + 6 + \cdots + (n-2) + n) + 1 \\
 & = & c \times 2(1 + 2 + 3 + \cdots + (n/2)) + 1 \\
 & = & 2c(\sum_{i=1}^{n/2} i) + 1 \\
 & = & 2c((n/2)((n/2) + 1)/2) + 1
\end{array}$$

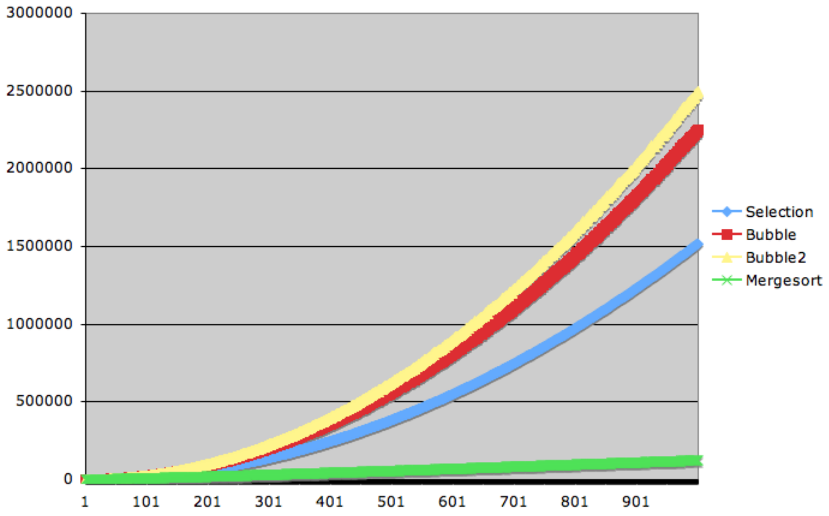
Thus, the running time is still $O(n^2)$.

11.9 Chapter 9 Solutions

1. `selectionSort` performs exactly one swap in each iteration of i, wherease `bubbleSort` can perform up to as many as i. Thus, `bubbleSort` performs more swaps than `selectionSort`.
2. For each, the only change that needs to be made is to first copy the contents of `data` into a new array and use this new array to perform all of the sorting steps. Returning this new array, and not modifying `data` at all, leaves the input array in the same state as it is when it is passed to the method.
3.
```
public static void insertionSort(int[] data) {
  int temp;
  for (int i = 1; i < data.length; i++) {
    temp = data[i];
    for (int j = i; j >= 0; j--) {
      if (j > 0 && temp <= data[j-1]) {
        data[j] = data[j-1];
      }
      else {
        data[j] = temp;
        break;
      }
    }
  }
}
```
   By looking at the bounds of each loop, we notice that, just as for `bubbleSort` and `insertionSort`, each loop takes $O(n)$ iterations. Thus, the running time of `insertionSort` is $O(n^2)$.


30000
25000
20000
15000
10000
5000
0
1
11
21
31
41
51
61
71
81
91
Selection
Bubble
Bubble2
Mergesort

4.


3000000
2500000
2000000
1500000
1000000
500000
0
1
101
201
301
401
501
601
701
801
901
Selection
Bubble
Bubble2
Mergesort

11.10 Quiz 1 Solutions

1. For each 8-bit string, compute its decimal value in UM, SM, and TC:

   (a) `0110 0001`

   Answer: **UM: 97; SM: 97; TC: 97**

   (b) `1001 1000`

   Answer: **UM: 152; SM: -24; TC: -104**

   (c) `1000 0000`

   Answer: **UM: 128; SM: 0; TC: -128**

2. For each decimal number, compute its 4-bit binary representation in SM and TC:

   (a) `-1`

   Answer: **SM: 1001; TC: 1111**

   (b) `-5`

   Answer: **SM: 1101; TC: 1011**

3. (a) What do the following 32 bits represent as a `float`?

   ```
   1 11111111 00000000000000000000000
   ```

   Answer: $-\infty$

(b) What do the following 32 bits represent as a `float`?

`1 10000000 10100000000000000000000`

Answer: $s = 1, e = 128, f = \frac{1}{2} + \frac{1}{8}$

$$\begin{aligned}(-1)^1 \times (1.101) \times 2^1 &= (-1)(1 + \tfrac{1}{2} + \tfrac{1}{8})(2) \\ &= (-1)(\tfrac{13}{8})(2) \\ &= -\tfrac{26}{8} \\ &= -3.25\end{aligned}$$

(c) How is the decimal number `6.50` encoded as a `float`?

Answer: $6.50 = 110.1 = 1.101 \times 2^2$

$s = 0, e = 2 + 127 = 129_{ten} = 10000001, f = 10100...$

Alternatively, if we notice that $6.50 = 3.25 \times 2^1$, we know that in scientifc notation, the only difference between 3.25 and 6.50 is the exponent. So we know f will stay the same, and we set e to be the exponent for 3.25 plus the extra one (128+1=129).

4. Consider a 32-bit `int` variable `argb` that stores the ARGB components of a pixel as we saw in class: the A component in bits 24 to 31, R in bits 16 to 23, G in bits 8 to 15, and B in bits 0 to 7. Give expressions to extract each component using bit shift operators and a decimal integer bit mask, *not* a hexadecimal one. **10** *points*

Answer:

- **A:** `(argb >> 24) & 255`
- **R:** `(argb >> 16) & 255`
- **G:** `(argb >> 8) & 255`
- **B:** `argb & 255`

Some people tried to mask first and then shift. This is very difficult (without a machine) because the decimal bit masks become very large and messy to compute. For each component, the following are the bitmasks (in hex and decimal) that must be used if masking before shifting:

- **A:** `-16777216 = 0xff00000`
- **R:** `16711680 = 0xff0000`
- **G:** `65280 = 0xff00`
- **B:** `255 = 0xff`

5. Recall that in Java decimal literals are prefixed by nothing, octal literals are prefixed by `0`, and hexadecimal literals are prefaced by `0x`. **10** *points*

(a) Do `5`, `05`, and `0x5` represent the same number? If not, how do you represent 5 in octal and hex?

Answer: **All the same value.**

(b) Do `21`, `021`, and `0x21` represent the same number? If not, how do you represent 21 in octal and hex?

Answer: `21=025=0x15`

6. A collection of four bits is sometimes called a *nibble* because it is half the size of a byte. Suppose Java offered a 4-bit integer primitive type called `nibble` and that its encoding scheme was 2's complement. Also suppose Java offered a 4-bit integer primitive type called `unsigned nibble` and that its encoding scheme was unsigned magnitude. **15** *points*

   (a) What is the range of `unsigned nibble`?

   Answer: 0 **to** 15

   (b) Next to each of the following statements, write what the value of `n` is after the statement gets executed. Assume that these statements are executed in DrJava in succession.

```
> unsigned nibble n = 0

> n += 3  // 3

> n += 3  // 6

> n += 3  // 9

> n += 3  // 12

> n += 3  // 15

> n += 3  // 2
```

   (c) What is the range of `nibble`?

   Answer: −8 **to** 7

   (d) Next to each of the following statements, write what the value of `n` is after the statement gets executed. Assume that these statements are executed in DrJava in succession.

```
> nibble n = 0

> n += 3  // 3

> n += 3  // 6

> n += 3  // -7

> n += 3  // -4

> n += 3  // -1

> n += 3  // 2
```

7. In addition to primitive types for single-precision floating-point (`float`) and double-precision floating-point (`double`), suppose that Java offered a "half-precision" floating-point type called `half`. Suppose a `half` is 16 bits – 1 for sign, 6 for exponent, and 9 for fraction – and its encoding scheme is analagous to `float` and `double`.

   (a) For what range of exponents e is the value interpreted as a normalized number? Recall that normalized here means there is exactly one 1 to the left of the binary point.

   Answer: $1 \leq e \leq 62$

   (b) What is the bias of the exponent? In other words, what value is subtracted from the actual value in the exponent bits?

   Answer: **31**

   (c) What do the following 16 bits represent as a `half`?

   `1 100011 000100000`

   Answer: $s = 1, e = 35, f = \frac{1}{2^4}$

$$\begin{aligned}(-1)^1 \times (1.0001) \times 2^{35-31} &= (-1)(1+\tfrac{1}{2^4})(2^4) \\ &= (-1)(2^4+1) \\ &= -2^4 - 1 \\ &= -17\end{aligned}$$

8. Consider the following two encoding schemes. For each, can zero and all positive numbers be represented? If not, give an example of a number that cannot be represented. If so, can they be represented uniquely? If not, explain why or give an example to demonstrate. **5** *points*

   (a) Strings of 0's, 1's, 2's, and 3's where $a_{n-1}...a_1a_0 = \sum_{i=0}^{n-1} a_i * 3^i$

   Answer: **Attempting to encode base-3 with 4 symbols allows all non-negative numbers to represented but not uniquely. Let's start counting (the decimal representation is in parens): 0 (0), 1 (1), 2 (2), 3 (3), 10 (3), 11 (4), 12 (5), 13 (6), 20 (6), ... Notice there are two ways to encode each multiple of 3.**

   (b) Strings of 0's, 1's, 2's, and 3's where $a_{n-1}...a_1a_0 = \sum_{i=0}^{n-1} a_i * 5^i$

   Answer: **Attempting to encode base-5 with 4 symbols does not work. Let's start counting: 0 (0), 1 (1), 2 (2), 3 (3), 10 (5), 11 (6), 12 (7), 13 (8), 20 (10), 21 (11), ..., 33 (18), 100 (25), ... Notice that there are values that get skipped so they don't have any representation.**

9. The system of using tally marks to count is a form of *unary notation.* Can fractions be encoded in a unary system? If so, what might the encoding scheme be? If not, why not?

   Answer: **If we apply the typical encoding for fractions, we would have $a_1a_2...a_n = \sum_{i=1}^{n} a_i\frac{1}{1^i}$. $\frac{1}{1^i}$ is always 1, however, so this sum doesn't actually represent fractional numbers. With only 1 symbol, there is no fraction that we can repeatedly split (like $\frac{1}{10}$ in base-10 and $\frac{1}{2}$ in base-2).**

Reference – You may tear this sheet off.

A `float` is 32 bits – 1 for sign, 8 for exponent, and 23 for fraction – and has the following encoding scheme:

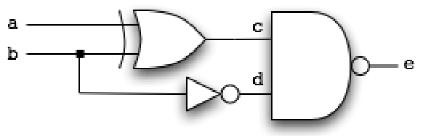
S	E	F	meaning
s	$1 \leq e \leq 254$	f	$(-1)^s \times 1.f \times 2^{(e-127)}$
0	0	0	zero
1	0	0	zero
s	0	f `!=`0	$(-1)^s \times 0.f \times 2^{-126}$
0	255	0	$+\infty$
1	255	0	$-\infty$
s	255	f `!=`0	`NaN`

A `double` is 64 bits – 1 for sign, 11 for exponent, and 52 for fraction – and has the following encoding scheme:

S	E	F	meaning
s	$1 \leq e \leq 2046$	f	$(-1)^s \times 1.f \times 2^{(e-1023)}$
0	0	0	zero
1	0	0	zero
s	0	f `!=`0	$(-1)^s \times 0.f \times 2^{-1022}$
0	2047	0	$+\infty$
1	2047	0	$-\infty$
s	2047	f `!=`0	`NaN`

11.11 Quiz 2 Solutions

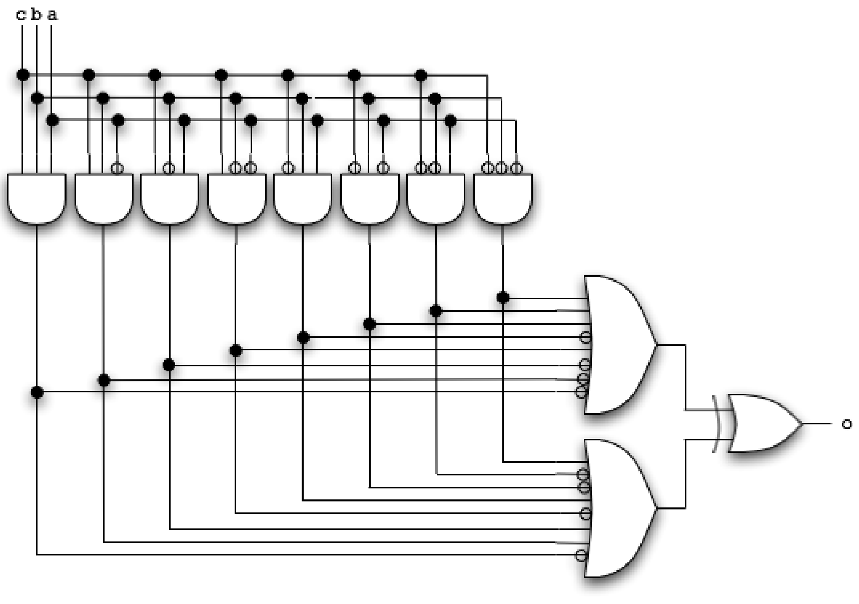
1. Write out the truth table that this circuit encodes:



Answer:

a	b	c	d	e
0	**0**	**0**	**1**	**1**
0	**1**	**1**	**0**	**1**
1	**0**	**1**	**1**	**0**
1	**1**	**0**	**0**	**1**

2. Write out the truth table that this circuit encodes:



Answer:

c	b	a			o
0	**0**	**0**	**1**	**1**	**0**
0	**0**	**1**	**1**	**0**	**1**
0	**1**	**0**	**1**	**0**	**1**
0	**1**	**1**	**0**	**1**	**1**
1	**0**	**0**	**1**	**0**	**1**
1	**0**	**1**	**0**	**1**	**1**
1	**1**	**0**	**0**	**1**	**1**
1	**1**	**1**	**0**	**0**	**0**

3. Consider the following alternative implementation of linear search on a sorted array of integers with no duplicates:

```
public int backwardsLinearSearch(int x, int[] data) {
  for (int i = data.length - 1; i >= 0; i--) {
    if (x == data[i])
      return i;
  }
  return -1;
}
```

(a) Is the best case input for `backwardsLinearSearch` the same as for `linearSearch`? If not, what is it? **3** *points*

Answer: **No. For `backwardsLinearSearch`, the best case input is when `x` is the last element in `data` (at index `data.length - 1`.**

(b) Is the worst case input for `backwardsLinearSearch` the same as for `linearSearch`? If not, what is it? **3** *points*

Answer: **Yes, the worst case for both is if `x` is not in `data`. Some people said that the worst case for `backwardsLinearSearch` is when `x` is the first element in `data`. This is not as bad as not finding it at all, because finding the element results in the last loop increment and conditional check not getting executed.**

(c) Let the integer `x` be contained in the integer array `data`. When do `linearSearch` and `backwardsLinearSearch` take the exact same number of steps when `x` and `data` are passed as arguments? **4** *points*

Answer: **If `data` has an odd number of elements and `x` is in the middle position (index `data.length/2`).**

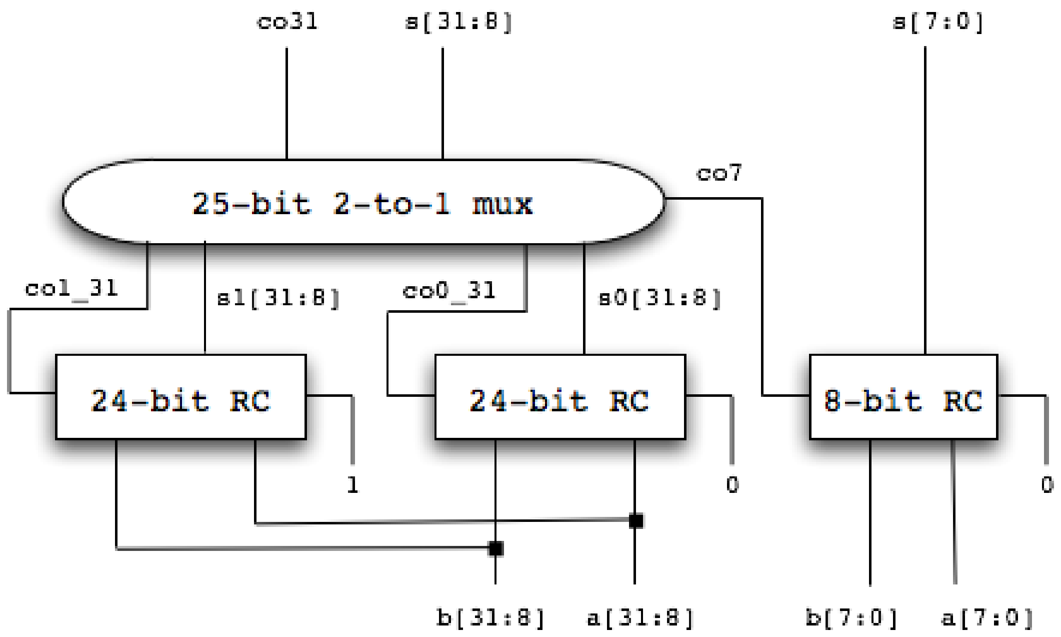
4. For each of the following, write "T" if the statement is true or "F" if the statement is false. Make sure your answers are legible. **10** *points*

1. <u>T</u> x is $O(x)$
2. <u>F</u> x is $O(\sqrt{x})$
3. <u>T</u> 10 is $O(9)$
4. <u>T</u> $13x^2 + 3$ is $O(0.00001x^2)$
5. <u>F</u> $x \log x$ is $O(\log x)$

5. (a) Why can any boolean function be encoded using a circuit with a gate delay of 2?

Answer: **Because any truth table can be encoded in a PLA, and a PLA consists of one level of ANDs and one level of ORs. Every path goes through exactly one AND and one OR, so the longest path has gate delay 2.**

(b) Consider the following 32-bit carry-select adder in two uneven pieces. The lowest-order 8 bits are computed by an 8-bit ripple-carry adder, and the higher-order 24 bits are computed in parallel by two 24-bit ripple-carry adders. The carry-out from bit 7 is used to select the appropriate sum of the higher-order 24 bits.



What is the gate delay of this adder?

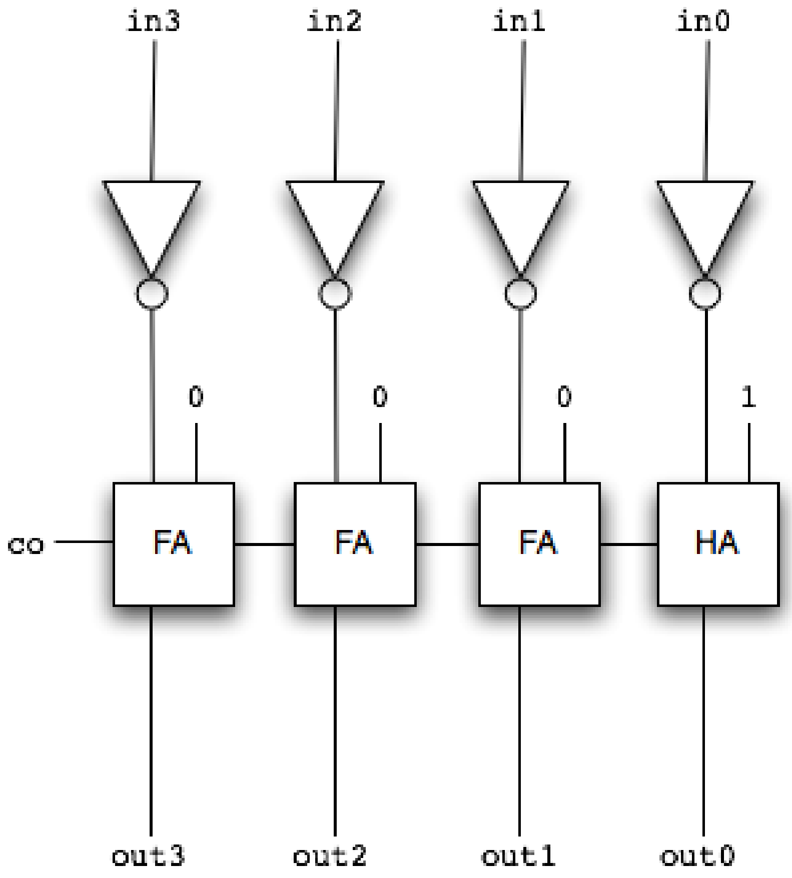
Answer: **The longest path is through one of the 24-bit ripple-carry adders and through the mux. The 24-bit ripple-carry adder takes** $2(24) = 48$ **gate delays. The mux takes 2 gate delays, since a 25-bit mux is built from 25 1-bit muxes in parallel. Thus, the total gate delay is 50.**

6. (a) Draw a circuit that takes a 4-bit number, performs the following two operations on them, and outputs the resulting 4-bit number: **5** *points*

- invert all the bits
- add 1 to the number

You may use boxes labeled `FA` to denote full adders and boxes labeled `HA` to denote half adders. Clearly label all wires. The input and output wires are already drawn for you.

Answer:



(b) Explain what this circuit does. **5** *points*

Answer: **This circuit is useful for converting a positive number b in TC to $-b$ in TC and for converting a negative number $-b$ in TC to b in TC. This type of behavior is useful for building a single circuit that performs addition and subtraction.**

7. Show that $5n^2 + 2n + 70$ is $O(n^2)$. That is, find integer constants $c > 0$ and $N > 0$ such that for all $n \geq N$, the following inequality is true:

$$5n^2 + 2n + 70 \leq cn^2$$

Answer: **There are an infinite number of constants c and N that prove this claim. If we pick $c = 77$, then the inequality becomes:**

$$2n + 70 \leq 72n^2$$

This is true for all $n \geq 1$, since $2(1) + 70 = 72(1^2)$, and both sides of the inequality are monotone increasing functions. Thus, we can pick $N = 1$.

8. Consider the following methods `f` and `g`:

```
public void f(int n) {
  for (int i = 0; i < n; i++)
    if (i % 2 == 0)
      System.out.println("hello");
}
public void g(int n) {
  for (int i = 0; i < n; i++)
    f(i);
}
```

For the next two parts, use the model for counting steps that we have been using in lecture.

(a) What is the number of steps that `f` takes when `n` is even? when `n` is odd? These should be functions of n.

Answer:
1 **for initialization,** $n + 1$ **guard checks, and** n **increments. These** $2n + 2$ **steps are run for both even and odd** n**s. Now we count the number of print functions by case.**
For n **is even, there are** $n/2$ **numbers that are even, so there are** $0.5n$ **prints. Thus, for** n **is even, the total number of steps is** $2n + 2 + 0.5n = 3.5n + 2$.
For n **is odd, there are** $(n+1)/2$ **even numbers, so there are** $0.5n+0.5$ **prints. Thus, for** n **is odd, the total number of steps is** $2n + 2 + 0.5n + 0.5 = 3.5n + 2.5$.

(b) What is the number of steps that `g` takes when `n` is even? when `n` is odd? These should be functions of n. **5** *points*

Answer:

$2n+2$ is the number of steps not including the time for all the calls to `f`.

Counting the total cost of calls to `f` is tricky, because each time it is called with a different input – whatever the value of `i` is. We use $f(i)$ to denote the number of steps `f` takes when given input `i`.

When n is even, $n/2$ values of `i` are even and $n/2$ are odd. Thus, the time for all `f` calls is

$$\begin{aligned}
& f(0)+f(1)+\cdots+f(n-2)+f(n-1) \\
= \; & 3.5(0)+2+3.5(1)+2.5+\cdots+3.5(n-2)+2+3.5(n-1)+2.5 \\
= \; & (n/2)(2)+(n/2)(2.5)+3.5\sum_{i=0}^{n-1} i \\
= \; & n+1.25n+3.5(n^2/2-n/2) \\
= \; & 1.75n^2+0.5n
\end{aligned}$$

This plus $2n+2$ gives $1.75n^2+2.5n+2$ total steps.

When n is odd, $(n+1)/2$ values `i` are even and $(n-1)/2$ are odd. Thus, time for all `f` calls is

$$\begin{aligned}
& f(0)+f(1)+\cdots+f(n-2)+f(n-1) \\
= \; & 3.5(0)+2+3.5(1)+2.5+\cdots+3.5(n-2)+2.5+3.5(n-1)+2 \\
= \; & ((n+1)/2)(2)+((n-1)/2)(2.5)+3.5\sum_{i=0}^{n-1} i \\
= \; & n+1+1.25n-1.25+(n^2/2-n/2) \\
= \; & 1.75n^2+0.5n-0.25
\end{aligned}$$

This plus $2n+2$ gives $1.75n^2+2.5n+1.75$ steps.

(c) Give a tight asymptotic upper bound for the running time of `f`. You do not need to prove that it is correct. **2.5** *points*

Answer:

n is a tight upper bound, since the running time of `f` is $O(n)$.

(d) Give a tight asymptotic upper bound for the running time of `g`. You do not need to prove that it is correct. **2.5** *points*

Answer:

n^2 is a tight upper bound, since the running time of `g` is $O(n^2)$.

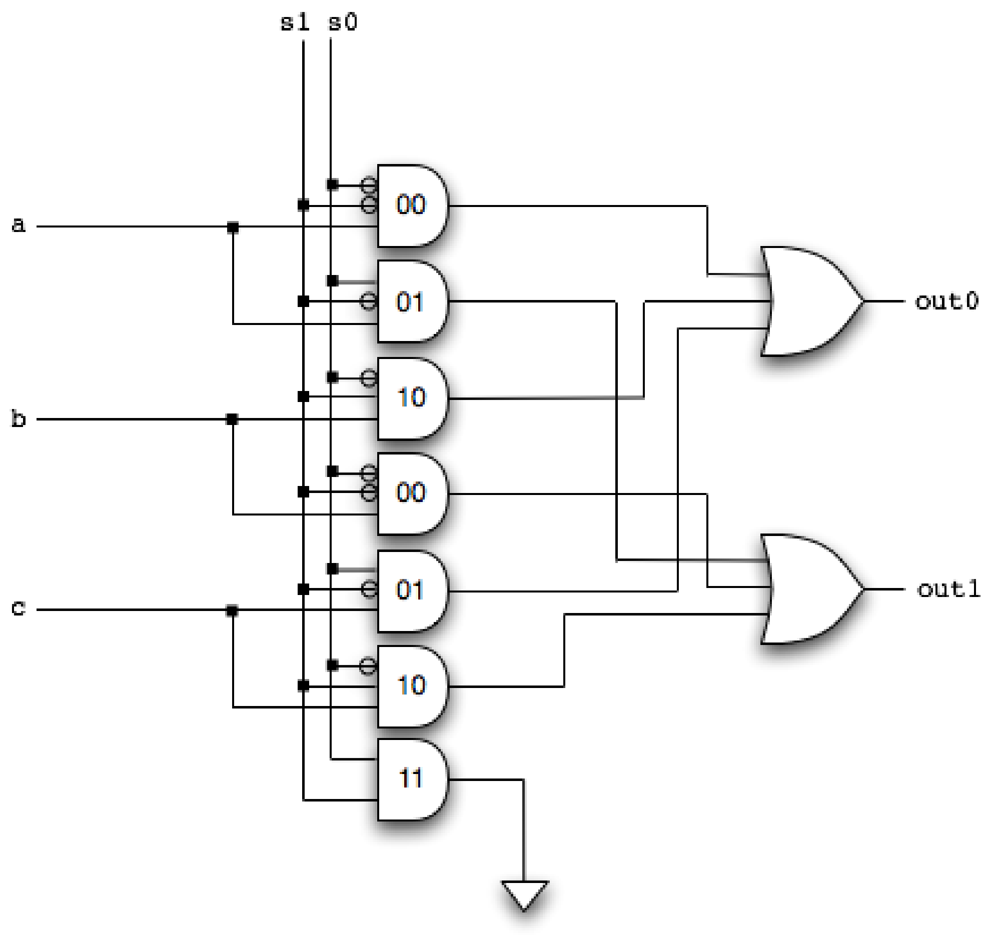
9. We have seen a couple of 2-to-1 muxes, circuits that take two inputs and select one of them to output. See the Reference section for the 1-bit 2-to-1 mux.

   We would now like to build a mux that takes 3 inputs and chooses 2 of them not just one. Let's call the inputs a, b, and c and the two outputs $o1$ and $o0$. Let the following truth table summarize the behavior we would like this mux to have. Notice that there are two selector wires, $s1$ and $s0$.

$s1$	$s0$	$o1$	$o0$
0	0	b	a
0	1	a	c
1	0	c	b

   When both selector wires are 1, the voltage is connected to ground. You do not need to do anything more for this case. In the following diagram, fill in the appropriate connecting wires to achieve the desired behavior for this 1-bit 3-to-2 mux.

   Answer:



11.12 Quiz 3 Solutions

1. The recurrence relation for `f` is $T(n) = c + T(n/2)$. We have seen this before (ie, recursive binary search), so we know that $T(n)$ is $O(\log n)$ time.

   The time it takes for `g` to execute is the time it takes `f` to execute with input `2n`. Since the input $2n$ is is asymptotically the same size as n, the call to `f` takes $O(\log n)$ time.

   The worst-case execution of `h` includes the `if` branch, where two recursive calls are made. Once this branch is taken for the first time, however, it will never be taken again (since `callTwice` is set to `false` and can never be changed back to `true`). Thus, the worst-case is when two recursive calls are made the first time and then one recursve call thereafter. This gives the following relationships:

$$\begin{array}{rcl} T(n) & = & c + 2T(n/2) \\ 2T(n/2) & = & 2c + 2T(n/4) \\ 2T(n/4) & = & 2c + 2T(n/8) \\ & \vdots & \\ 2T(4) & = & 2c + 2T(2) \\ 2T(2) & = & 2c + 2T(1) \\ 2T(1) & = & 0 \\ \hline T(n) & = & c + 2c(\log n - 2) \\ & = & 2c \log n - 3c \end{array}$$

   Thus, `h` runs in $O(\log n)$ time.

2. (a) Linear search takes $O(n)$ time.

   (b) `selectionSort` takes $O(n^2)$ time and binary search takes $O(\log n)$ time. So overall this takes $O(n^2)$ time.

   (c) `mergeSort` takes $O(n \log n)$ time and binary search takes $O(\log n)$ time. So overall this takes $O(n \log n)$ time.

   Thus, the first algorithm is the best for this particular task.