1

Arithmetic Reconsidered

The only prerequisite for understanding this book is experience with grade school arithmetic.[1] The most difficult aspect of calculus is handling the necessary arithmetic and algebra, and we will be doing plenty of those! In order to make things go more smoothly, we will spend Chapter 1 throughly reviewing arithmetic in a variety of settings. The most useful application of arithmetic, besides calculus, is programming, so we will do some programming.[2] I have provided many opportunities for us to practice. Time spent wrestling with them will turn our efforts, trying to understand the difficult and abstract concepts that follow, from almost impossible to merely very difficult.

However, these concepts are only very difficult because they are worth knowing about. Take the Schrödinger equation for example. No one expects $i\hbar\frac{\mathrm{d}}{\mathrm{d}t}|\Psi\rangle = H|\Psi\rangle$ to be easy on anyone's first exposure. Nevertheless, this marvelous equation makes most modern technologies possible, and is worth understanding. However, we will have to overcome many challenging ideas in order to get there. Ideas, that will take lots of time and energy to digest. You will be frustrated. Let me offer you a tip, a "cheat code" if you will. Most of the difficulties can be overcome once you remember: *almost everything* we do here is just arithmetic. Some hairier than others, but still arithmetic.

You have spent many years doing arithmetic, just take things slowly. You got this. Let us begin.

1.1 Units

Let's start at the very beginning~ ♪

All physical theories must have something to say quantitatively about the world around us. In order to communicate coherently about real world objects, we must agree on a set of units. For example, the distance from one café to another might be 50 meters. Or is it 164 feet?

This is the one case where trying to please everyone turns out to be helpful. In order to make everyone happy for now, let us agree to refer to all sorts of distance measurements as a **L**ength. Thus the height of a building and the distance from the earth to the sun are both instances of **L**engths.

[1] In particular, I will assume that you are familiar with the symbols $+$, $-$, $\times$, $/$, $=$, $\leq$, and $\geq$.

[2] We will start with the basics, so no previous experience with programming is required.

Now, in order to indicate speed, we usually divide something by time. For example, 6 slices of pizzas per hour might mean the speed at which pizza slices were consumed. Similarly, the distance from the earth to the sun, divided by the time it takes for light to hit the earth from the sun indicates the speed of light. Thus dividing a length by time gives us speed:

$$\frac{\textbf{L}\text{ength}}{\textbf{T}\text{ime}} = \text{Speed}.$$

Some like to use seconds to measure time, others like to use hours; we will just call all time measurements **T**ime. Suppose I ate 6 slices of pizza per hour for 2 hours. Then, I ate a total of: 6 slices/hour $\times$ 2 hours = 12 slices. If we multiplied the speed of light by 1 year, which is a **T**ime,

$$\underbrace{\frac{\textbf{L}\text{ength}}{\cancel{\textbf{T}\text{ime}}}}_{\text{speed of light}} \times \underbrace{\cancel{\textbf{T}\text{ime}}}_{\text{1 year}} = \underbrace{\textbf{L}\text{ength.}}_{\text{one lightyear}}$$

Another fundamental type of measurement is mass, which for now we will use interchangeably with weight. Some folks use kilograms, others use pounds. We will just call these **M**ass.

Einstein told us that Energy is mass times the speed of light squared. In symbols, this is $E = mc^2$, where E is energy, m is mass, and c is the speed of light. Notice that when using symbols, we omit the $\times$ symbol. How are scientists not confused by this omission? By agreeing to only use single letters. This means it is necessary to not only use letters from the Latin alphabet, but also Greek, Hebrew, fonts like $\mathfrak{Fraktur}$, and symbols like $\flat$ and $\sharp$. Thus $E = mc^2$ means $E = m \times c^2$, which in turn means $E = m \times c \times c$. Since m is a **M**ass and c is **L**ength divided by **T**ime (speed),

$$\begin{aligned}\text{Energy} &= \textbf{M}\text{ass} \times \left(\frac{\textbf{L}\text{ength}}{\textbf{T}\text{ime}}\right)^2 = \textbf{M}\text{ass} \times \frac{\textbf{L}\text{ength}}{\textbf{T}\text{ime}} \times \frac{\textbf{L}\text{ength}}{\textbf{T}\text{ime}} \\ &= \textbf{M}\text{ass} \times \frac{\textbf{L}\text{ength}^2}{\textbf{T}\text{ime}^2}.\end{aligned}$$

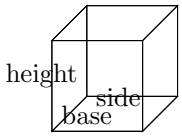


Figure 1.1: A cube with a base, side, and height.

Here is my first challenge for you. The main challenge with this one is getting your pencil and paper out. Later ones will not be this easy! Solutions are at https://dayeolchoi.com/calculus.

Challenge 1

(a) We can simplify arithmetic expressions using cancellation. For example,

$$\frac{5 \times 10 \times 3}{3 \times 2} \times 2 = \frac{5 \times 10 \times \cancel{3}}{\cancel{3} \times \cancel{2}} \times \cancel{2} = 5 \times 10.$$

Try simplifying the following expressions. By convention, $2^2 = 2 \times 2$ and $5^3 = 5 \times 5 \times 5$.

$$5^5 \times \frac{3}{5^3} \qquad\qquad \frac{5^5 \times \frac{3}{5^3}}{2^2}.$$

(b) The volume of say, a cube, is the base of the cube multiplied with the side of the cube, and the hight of the cube (Figure 1.1). A density of a substance is its **M**ass divided by volume. Use cancellation to simplify the following expression as much as possible. Do you recognize it?

$$\frac{\textbf{L}\text{ength}^5 \times \text{density}}{\textbf{T}\text{ime}^2}.$$

(c) Remember that we cannot add apples to oranges. Thus the mathematical expression

$$2 \text{ oranges} + 3 \text{ apples}$$

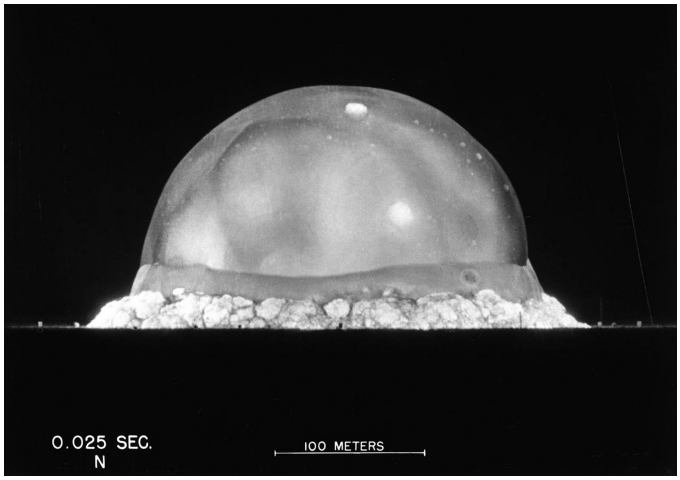
makes no sense. Similarly, we cannot add a **L**ength to **T**ime. Identify which of the following arithmetic expressions are valid:

$$\frac{\textbf{L}\text{ength}^5 \times \text{density}}{\textbf{T}\text{ime}^2} + \frac{\textbf{L}\text{ength}}{\textbf{T}\text{ime}}, \quad \frac{\textbf{L}\text{ength}^5 \times \text{density}}{\textbf{T}\text{ime}^2} + \text{Energy}, \quad \frac{\textbf{L}\text{ength}^5 \times \text{density}}{\textbf{T}\text{ime}^2} + \frac{\text{Energy}^2}{\textbf{T}\text{ime}}.$$

What we have found in Challenge 1 is that the laws of nature are constrained rather strongly. For example, if Einstein told us $E = mc + m/c$ or $E = m + c^2$, there's no way either could be true because the units cannot match. We will now apply this idea to a more serious setting.

Hot and cold

Below is an image of the Trinity nuclear test, the first detonation of a nuclear weapon in history. The campaigns at Iwo Jima, Okinawa, and many others, were proving to be far too brutal and deadly to its participants (including civillians) during World War II. There was a real need for a weapon so dangerous that the other side would have no choice but to surrender, thus ending the war. The atomic bombing of Hiroshima and Nagasaki killed over a hundred thousand people. The bombing also immediately led to the end of the Second World War; the surrender of Japan meant a full scale invasion of the island (Operation Downfall) was cancelled. The counterpart to Operation Downfall was Japan's Operation Ketsugō, which called for "the glorious death of one hundred million." Whether this was realistic or not, an invasion of the island to end the war would have killed several millions of people from both sides.



Trinity ushered in the atomic age and ended the Second World War. However, another one was to follow immediately: the First Cold War. A fear of an imminent end of humanity always loomed in the air, nuclear warfare could happen at any moment. However, many of us were born after the dissolution of the Soviet Union, and we have little idea how bad things were back then.

Nevertheless, you and I reap the benefits of those times. I am writing these words on a computer, first developed during World War II to facilitate calculations for ballistics and explosives. I use a program to compile the words I write into an image file that you can read; the first reprogrammable computers were developed to do numerical calculations for the hydrogen bomb. The internet originates from ARPANET, a project designed for sustained communications during a nuclear war.[3] Our phones (a miniaturized computer) have a long battery life because it uses lithium ion batteries, developed by the CIA during the heat of the Cold War for use in their spy gadgets. The microwave oven I use to reheat leftover pizza is a derivative of radar technology designed to identify enemy planes for a takedown. If I'm feeling hungry were to order food online at home, I know my order will arrive correctly because of GPS technology, developed to pinpoint ballistic missile submarines and mobile launch platforms. A GPS user finds their location by receiving geolocation information from satellites. Satellites were developed by the Soviet Union because they lacked airborne bombing capabilities and needed to develop intercontinental ballistic missiles (ICBM) that could reach continental US.[4]

We return to Trinity, the test that started the atomic age. As we can see from the test image, the energy from the bomb is released in what appears to be a spherical blast. The **radius** of a sphere is the distance from the center of the sphere to its boundary. Thus radius is a **L**ength. The *radius* of the blast (let's label this with the letter R) will be proportional to the *energy* of the bomb (we'll refer to this with the letter E), and the **T**ime since blast, t.[5] If the bomb was surrounded by dense material, such as concrete and steel, we'd imagine the blast radius will be small. On the other hand, if the bomb was surrounded by less dense material like air, the blast radius will be larger. We will refer to the *density* of the surrounding material with the Greek letter ρ (rho). Below is a table summarizing what we have. Later on, we will use the shorthand appearing in the fourth column to save on space. As is customary when using symbols, $\mathrm{ML}^2/\mathrm{T}^2$ omits the '$\times$' symbol.

variable	meaning	unit type	unit shorthand
R	Radius (of a Blast)	**L**ength	L
E	Energy (of a Bomb)	**M**ass $\times \frac{\textbf{L}\mathrm{ength}^2}{\textbf{T}\mathrm{ime}^2}$	$\mathrm{ML}^2/\mathrm{T}^2$
t	Time passed	**T**ime	T
ρ	Density (of the surrounding)	**M**ass/**L**ength^3	$\mathrm{M/L}^3$

What we would like to know is the simplest formula for the energy of a bomb, E, given that we know its blast radius R at time t with surrounding material density ρ (there will be other contributing factors, but the ones we have look to be the most important). From Challenge 1, we know that there is a simple formula to express such cases. Since

$$\text{Energy} = \frac{\textbf{L}\mathrm{ength}^5 \times \text{density}}{\textbf{T}\mathrm{ime}^2}$$

[3] The first email was sent over the ARPANET, for example.
[4] A satellite with a nuclear weapon payload and reduced angle of launch is an ICBM.
[5] Within seconds of the blast, a larger value of t will lead to a larger blast radius R.

we see that energy E must be proportional to $\frac{R^5\rho}{t^2}$, by considering the units involved. I say proportional to, because the simplest formula for energy E could be

$$E = 3\frac{R^5\rho}{t^2}, \text{ or } E = 3.14\frac{R^5\rho}{t^2}, \text{ or } E = 5\frac{R^5\rho}{t^2} \dots.$$

We cannot rule any such possibilities because a number itself has no units. Indeed, the numbers 3, 3.14, and 5 are all just abstract objects that live in our minds; the number 3 does not exist in the real world. In particular, the numbers 0, 1, 2, 3, 4, ... are abstract objects that we use to count, and we call such numbers **counting numbers** or **natural numbers**.

In the spirit of simplicity, I will assert for no good reason that

$$E = \frac{R^5\rho}{t^2}. \tag{1.1}$$

This jump in logic is not much worse than excluding, for example, the formula $E = \frac{R^5\rho}{t^2} + m\frac{R^2}{t^2}$ (m is the **Mass** of the bomb) which also works in terms of units, but is much less simple and seems much less natural, because it introduces additional complexity via a new factor '$m\frac{R^2}{t^2}$' without any good reason.

Challenge 2

(a) Does equation 1.1 make sense? Is an increase in blast radius associated with more energy? If R and t were the same, but we had a very dense material (thus a high density ρ), how does that affect the energy? What if the time to reach a specific blast size was smaller, what would that tell us about energy E?

(b) Using a calculator, the nuclear test image, and equation 1.1, estimate the energy released by the trinity experiment. We'll use meters (m) for radius R, kg/m^3 for density ρ, and seconds (s) for time t. Thus E has the unit $\text{kg}\cdot\text{m}^2/\text{s}^2$.[6] Just eyeball the value for radius R, and use the fact that air density ρ is about 1.2kg/m^3.

(c) The unit of energy $\text{kg}\cdot\text{m}^2/\text{s}^2$ is called a **joule**. However, thousands of tons of TNT, called **kilotons**, is the standard convention for expressing the explosive energy released by a fission weapon like Trinity.[7] If $4.2 \cdot 10^9$ joules is about 1 ton of TNT, how many kilotons do you estimate was released in the Trinity test? This is just an estimate, feel free to round to the nearest kiloton.

(d) Look up the yield of the Trinity nuclear test online and compare with your result from (c).[8]

Hopefully, this Challenge gives a first indication that arithmetic is more than what we punch into calculators. The natural next step is to figure out how to obtain the formula

$$E = \frac{R^5\rho}{t^2}.$$

To do this, we'll need to review a bit of multiplication.

[6] The $\cdot$ is a shortened form of $\times$. We need a multiplication symbol because units aren't always single letters.

[7] The term kiloton is apparently due to John von Neumann, from his time at the Manhattan project as a consultant to the United States Government.

[8] G.I. Taylor was one of the first outside the Manhattan Project's core group to estimate the yield of Trinity based on blast photos. This was in 1950 when not only was Trinity's yield a *Top Secret*, but only one country in the world had any nuclear arsenal. G.I. Taylor did not use this method; rather he used his physics expertise to obtain surprisingly accurate results.

1.2 Exponentiation

The multiplication $13 \cdot 9$ can be done relatively easily in our heads if we remember that[9]

$$13 \cdot 9 = (10 + 3) \cdot 9 = 90 + 27.$$

Since $13 \cdot 9 = 9 \cdot 13$, an entirely equivalent calculation is

$$13 \cdot 9 = 9 \cdot 13 = 9 \cdot (10 + 3) = 90 + 27.$$

In order to make general mathematical statements, we will almost always use symbols in place of numbers, just like we used R to refer to radius (of a blast). Suppose we denote any three numbers with the letters a, b, and c. Then the above calculations may be expressed as

$$(a + b) \cdot c = a \cdot c + b \cdot c \quad \text{and} \quad a \cdot (b + c) = a \cdot b + a \cdot c.$$

Once again, we usually skip the '$\cdot$' when using symbols and write $(a+b)c = ac+bc$ and $a(b+c) = ab+ac$, respectively. Thus $a(b+c+d) = ab+ac+ad$ and $(i+j+k+l)z = iz+jz+kz+lz$.

Challenge 3

(a) Using the fact that $(a+b)(c+d) = ac+ad+bc+bd$, where a, b, c, d are numbers, prove that $(10+x)(10+y) = 10 \cdot (10+x+y) + xy$. Part (b) is an application of this fact.

(b) Do the multiplication $16 \cdot 14$ in your head. Next, do the multiplication $116 \cdot 114$ in your head.

Now that we have reviewed the multiplication of two numbers, let us review the multiplication of a finite collection of numbers. We know that $1000 = 10 \cdot 10 \cdot 10$ and $10000 = 10 \cdot 10 \cdot 10 \cdot 10$. As a convenient notation, let us agree to write $1000 = 10^3$ and $10000 = 10^4$ instead. Similarly, $0.1 = 1/10$ is written as 10^{-1}, which means that $0.0001 = 0.1 \cdot 0.1 \cdot 0.1 \cdot 0.1 = 10^{-4}$. This bookkeeping convention is called **exponentiation** and we typically indicate this using the word **power**. For example, 10^4 is 10 to the power of 4 and 10^{-4} is 10 to the power of -4. The number we are exponentiating is called the **base**. Thus 10 is the base of both 10^4 and 10^{-4}.

Below are the rules for exponentiation. I have denoted the base with letters a and d. The powers are denoted with the letters b and c. We agree not to take 0 to be a base in any of the following rules, because *0 multiplied by anything is 0 and one cannot divide by 0*.

- $\boxed{a^0 = 1}$ as in $3^0 = 1$ and $\mathbf{L}\text{ength}^0 = 0$,
- $\boxed{a^{-b} = \dfrac{1}{a^b}}$ as in $8^{-1} = \frac{1}{8}$ and $\mathbf{T}\text{ime}^{-2} = \frac{1}{\mathbf{T}\text{ime}^2}$.[10]
- $\boxed{a^b \cdot a^c = a^{b+c}}$ as in $6^2 \cdot 6^5 = 6^7$ and $\mathbf{M}\text{ass}^3 \cdot \mathbf{M}\text{ass}^{-4} = \mathbf{M}\text{ass}^{-1}$,
- $\boxed{(a \cdot d)^b = a^b \cdot d^b}$ as in $(2 \cdot 5)^4 = 2^4 \cdot 5^4$ and $(\mathbf{M}\text{ass} \cdot \mathbf{T}\text{ime})^2 = \mathbf{M}\text{ass}^2 \cdot \mathbf{T}\text{ime}^2$,
- $\boxed{(a^b)^c = a^{bc}}$ as in $(2^{-3})^4 = 2^{(-3)\cdot 4} = 2^{-12}$ and $(\mathbf{L}\text{ength}^2)^4 = \mathbf{L}\text{ength}^8$.

[9] From now on, we will use the symbol $\cdot$ instead of $\times$.

[10] From this rule, it follows that $\frac{b}{1/a} = ab$. Indeed, if a pizza has 8 slices, then if we divide three pizza into equal slices, are there will be 24 slices: $\frac{3}{1/8} = 3 \cdot 8$.

Fractions are allowed as powers; the number $a^{1/2}$ is usually written $\sqrt{a}$. Thus, one may write

$$5^{3/2} = (5^3)^{1/2} = \sqrt{5^3}.$$

More generally, for some nonzero counting number n, one sees $a^{1/n}$ written as $\sqrt[n]{a}$, called the n**th root**. For example, $3^{1/5} = \sqrt[5]{3}$. The 2nd root is usually called the **square root**, thus $\sqrt{5^3}$ is the square root of 5^3. You may be wondering what happens if we take fractional powers of negative numbers, like $\sqrt{-1}$, equivalent to $(-1)^{1/2}$. Let us postpone that discussion for now. The key take away from fractional powers is that using the fifth exponentiation rule:

$$\left(a^{p/q}\right)^q = a^{(p/q)\cdot q} = a^p$$

if a is a nonnegative number, and p, q are nonzero counting numbers. For example, $\left(54^{5/3}\right)^3 = 54^5$.

Simultaneous equations

We are now ready to obtain a formula for the nuclear blast yield. For reasons that will be clear later, we will first calculate a formula for the radius of a nuclear blast. Hence, we will first find how the radius of a nuclear blast R is related to the energy of a bomb E, time since blast t, and surrounding material density ρ.

As we have seen before, equations such as

$$R = E + t + \rho \quad \text{or} \quad R = E + t \cdot \rho$$

are not possible because the units don't match. For example, in the former case, we know it makes no sense to add energy to time and density. On the other hand, the simplest formula that could work is

$$R = d \cdot E^a \cdot t^b \cdot \rho^c, \tag{1.2}$$

where a, b, c, and d are unknown numbers. The first three are those that we need to find in order to make the units match. The number d on the other hand has no units; such numbers are called **dimensionless constants**.

For instance, notice that R is a length, so it has no units of time T.[11] However, on the right hand side of equation 1.2, the variables E and t have unit T. This means we somehow need to set the numbers a and b such that the unit T cancels out. Similarly, the radius R is independent of mass M. But the variables E and ρ have unit M. So we will have to set a and c in order to cancel out the unit M.

To proceed, let us convert our equation 1.2 into one consisting purely of units. Since the number d has no units, we'll put it aside for now. Using our handy table of units from earlier, we can write

$$\mathrm{L} = \left(\frac{\mathrm{ML}^2}{\mathrm{T}^2}\right)^a \cdot \mathrm{T}^b \cdot \left(\frac{\mathrm{M}}{\mathrm{L}^3}\right)^c.$$

We can use the exponentiation rules from before to simplify the right hand side of the equation

$$\left(\frac{\mathrm{ML}^2}{\mathrm{T}^2}\right)^a \cdot \mathrm{T}^b \cdot \left(\frac{\mathrm{M}}{\mathrm{L}^3}\right)^c = \frac{\mathrm{M}^a\mathrm{L}^{2a}}{\mathrm{T}^{2a}} \cdot \mathrm{T}^b \cdot \frac{\mathrm{M}^c}{\mathrm{L}^{3c}} = \mathrm{M}^a \cdot \mathrm{M}^c \cdot \frac{\mathrm{T}^b}{\mathrm{T}^{2a}} \cdot \frac{\mathrm{L}^{2a}}{\mathrm{L}^{3c}} = \mathrm{M}^{a+c} \cdot \mathrm{T}^{b-2a} \cdot \mathrm{L}^{2a-3c}.$$

[11] We are using the unit shorthand: **M**ass is M, **L**ength is L, and **T**ime is T.

Thus

$$\mathrm{L} = \mathrm{M}^{a+c} \cdot \mathrm{T}^{b-2a} \cdot \mathrm{L}^{2a-3c} \text{ or equivalently, } \mathrm{M}^0 \cdot \mathrm{T}^0 \cdot \mathrm{L}^1 = \mathrm{M}^{a+c} \cdot \mathrm{T}^{b-2a} \cdot \mathrm{L}^{2a-3c}.$$

It is now clear what we need to do to make everything work. We need to set the power of M at $a + c = 0$, the power of T at $b - 2a = 0$, and the power of L at $2a - 3c = 1$.

The first requirement is easy, let's set $a = -c$. That means that $2a = -2c$, and plugging this into the third requirement, we have $1 = 2a - 3c = -5c$. Thus $c = -1/5$ and $a = 1/5$. The only thing left is finding b, so let us look at the second requirement: $b - 2a = 0$, which is equivalent to $b = 2a$ (by adding $2a$ to both sides). We already know what a is, so $b = 2a = 2/5$. Therefore,

$$L = \left(\frac{\mathrm{ML}^2}{\mathrm{T}^2}\right)^{1/5} \cdot \mathrm{T}^{2/5} \cdot \left(\frac{\mathrm{M}}{\mathrm{L}^3}\right)^{-1/5},$$

or in our original equation form

$$R = d \cdot E^{1/5} \cdot t^{2/5} \cdot \rho^{-1/5}.$$

We now know the relationship between, say, the energy contained in a nuclear bomb and its blast radius. Let us invert the relationship so that we have energy E expressed as a combination of R, t and ρ. Taking the power of 5 to both sides, we get

$$R^5 = d^5 \cdot \frac{E \cdot t^2}{\rho}.$$

Now, multiply each side by $\frac{\rho}{d^5 \cdot t^2}$ and set $\alpha = 1/d^5$ to get

$$E = \alpha \frac{R^5 \cdot \rho}{t^2}, \text{ where } \alpha \text{ is a dimensionless constant.}$$

Setting $\alpha = 1$ gives us the formula we were looking for. Once again, we are doing arithmetic with units, so unit-less numbers (dimensionless constants) cannot be determined by this procedure.

Although our process for finding the formula was fairly long, the main problem was that of finding three unknown numbers a, b, and c such that the equations

$$a + c = 0, \; b - 2a = 0, \text{ and } 2a - 3c = 1$$

are all satisfied simultaneously.[12]

The act of taking a problem, determining the relevant factors and their corresponding units, and using these to investigate relationships between the factors is called **dimensional analysis**[13]. This is a useful skill, and I will be counting on you to do your own dimensional analysis later on. Rest assured, however, all dimensional analysis we will encounter in this book are much simpler than the Trinity problem.

[12] Here, there are three equations that must be satisfied, because we need to make sure that the units of mass, length, and time each match up. Furthermore, there are three unknown numbers (called a, b, c here) because there are three variables (energy E, time t, and density ρ) that form what we want (radius R).

[13] Why not call it unit analysis? Because unlike meters, kilograms, and seconds, **L**ength, **M**ass, and **T**ime are not strictly speaking, units. These are called **dimensions**, hence the name: *dimensional* analysis.

A search problem

If there are 5 pigeons in pigeonholes, but only 4 pigeonholes, then one of the pigeonholes must have at least 2 pigeons. More generally, if there are m pigeons, and n pigeonholes, with $m > n$, then at least one pigeonhole has 2 or more pigeons. This is called the **pigeonhole principle**.

The following challenge is a an application of exponentiation and the pigeonhole principle.

Challenge 4 Here is a very simple algorithm a device could use for finding music titles after hearing the first few segments of some music. A music begins with a starting pitch; check if the next pitch is the same, higher, or lower. If lower, register the number 0 in a box; if the same, register the number 1; if higher, register the number 2. Check the next pitch, and place the evaluation of the pitch difference in a box to the right. Repeat for each subsequent pitch. Once we are done, we might be able to obtain a long sequence of boxes with numbers in them called a **signature** of a song. An example is:

1	0	2	1	2	2	2	0	$\cdots$.

Whenever someone needs to search for a song, all we need to do is to compare the signature of the song to an existing database of signatures. On a first examination, this algorithm seems to throw away far too much information about a song to work. For example, why are we not recording the first pitch? Why aren't we recording more fine grained information about each subsequent pitch? On a closer examination, this is more effective than it seems, and there is a reason for the madness.

(a) Each song length varies widely, and it is costly to store too many boxes. If we allocated 0 boxes per song (each song has a signature of length 0), then how many unique signatures are possible? What if we allowed 1 box per song? Repeat for 2, 5, and 8 boxes.

(b) Our algorithm can be thought of placing each song signature (pigeon) into a pigeonhole. We want to ensure that we have enough pigeonholes to make it less likely that pigeons (song signatures) occupy the same hole. What is the minimum number of boxes we need to store the signature of one music, if we wish to distinguish between 100 million songs?

(c) We use a base 10 system, because humans generally have 10 fingers. However, hours and minutes are divided into 60 segments (60 minutes = 1 hour, 60 seconds = 1 minute). Below is an illustration of the base systems:

$$127 = 1 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0 \text{ (base 10)}, \ 120 = 2 \cdot 60^1 + 0 \cdot 60^0 \text{ (base 60)}.$$

Each box is only allowed to store the numbers 0, 1, 2. Thus our boxes operate in base 3. What is the largest power of 3 whose multiple fits in 12? Express the number 12 in base 3.

(d) Suppose we were to classify pitches into 12 different categories. What is the fewest number of boxes required to record the 12 different pitches? We are assuming a box can only store a single counting number between 0 and 2, inclusive.

(e) Suppose we modified the algorithm to keep the starting pitch (by categorizing them into 1 out of 12, and storing the category number into boxes). Thereafter, everything is the same (store 0/1/2 in a box based on the difference in subsequent pitch). Under this scheme, how many boxes would you need to store the signature of one music, given we wish to distinguish between 100 million songs?

(f) In terms of number of boxes needed, is it better to keep the starting pitch or is it worse? How many more/fewer boxes in total would you need to store starting pitches of 100 million songs? Would your answer change if there were 200 million songs in total?

(g) For reasons other than storage space, why might it be better to discard the starting pitch?

1.3 Arrays and Data types

We are now ready for an excursion into computing and precalculus. Essentially all that a digital computing device knows is arithmetic, and your experience spending time with the challenges in the previous sections will prove to be invaluable. Finding solutions to the challenges will turn out to be far less crucial than having struggled with the challenges.

In order to get into interesting material, we will start using some basic terminology from math and computing. Everything we will need is in the three page Appendix.

Arrays

These are times where a tiny SIM card is a faster computer than those used to first send humankind on the moon. With such great power comes great capabilities. Our goal will be to discuss some of the most important considerations that go into utilizing these great capabilities. So far, the most basic objects we dealt with were numbers and units. In computing, the most important basic object is that of an *array*.

An **array** is a realization (or implementation) of a list, whose elements are required to be of the same "type", and stored consecutively.[14] For example, I might have my grocery list of the day written down in three different notepads; that is certainly a list, but not an array, because not every item is listed consecutively. If I have a list of book titles mixed with numbers, we would consider these as lists, but not arrays, because numbers and book titles are not of the same "type".

Signatures of music, as described in Challenge 4, is an array, because a signature is a list of numbers, stored consecutively in sequence. To a (computing) device, a word is also an array, where each character occupies a box:

c	a	m	e	r	a

.

In the notation of the Appendix, an equivalent way to denote this is: [c|a|m|e|r|a]. A sentence is also a list of words, separated by spaces, so a sentence is also an array. For example, [W|h|a|t|_|i|s|_|a|_|c|a|m|e|r|a|?].[15]

Now, a processor in a device doesn't know anything about letters like 'W', or words like 'What', anymore than it knows what a camera is and what a camera looks like. All a processor knows are numbers. Thus all words, sentences, and all others are stored in devices as numbers.

One way to store words and sentences in computers is by agreeing on an *encoding* scheme, where we agree to designate a unique number for each character. For instance, suppose we were to use the ASCII standard for encoding characters into numbers suitable for storage in devices. The extended ASCII standard converts each character into a number between 0 and 255. In such a case, the device would store the sentence "What is a camera?" as:

$$[\underset{W}{87}|\underset{h}{104}|\underset{a}{97}|\underset{t}{116}|\underset{\_}{32}|\underset{i}{105}|\underset{s}{115}|\underset{\_}{32}|\underset{a}{97}|\underset{\_}{32}|\underset{c}{99}|\underset{a}{97}|\underset{m}{109}|\underset{e}{101}|\underset{r}{114}|\underset{a}{97}|\underset{?}{62}].$$ [16]

It is for the person using the device (or more likely, the program the person is using) to tell the device to interpret an array of numbers as an array of ASCII characters. Only then will a device be able to correctly interpret the number 87, for example, as the character 'W', instead of 87.

[14] Lists/arrays of length 0 are permitted, and are called **empty lists/arrays**.

[15] I have used the symbol _ here to denote a space. The sentence is: "What is a camera?"

[16] I have used the decimal representation of each encoding, rather than the binary representation, which is closer to how a device actually stores data.

Since words, sentences, arrays of numbers, and characters are all the same when stored in a device, computer scientists have a special term to encompass all of these; we call them **strings**, where multiple (possibly just one, or even zero, for **empty strings**) characters/numbers, and so on are "strung" together. Thus the sentence, "What is a camera?" is a string, the word "What" is a string, the character "a" is a string, the number "10010" is a string, the number "0" is a string, and so on. Of course, the number 10010 or 0 is often useful to interpret as a number, rather than a string. When a program retrieves the number 10010 stored in a device's memory, usually, the program will instruct the device to interpret it as a number (for example, 10010 could be the number of visitors to a place in a given day). If we wish to emphasize 10010 or 0 as a string, we use the double quotation marks; otherwise, we will not include quotation marks.

Now, suppose we instructed a device to store the strings "cat", "dog", and "ram". The device doesn't know what words are, so it will store each as numbers. Using the ASCII encoding, with the decimal representation of numbers, the device might store

$$[\underset{\text{c}}{99}|\underset{\text{a}}{97}|\underset{\text{t}}{116}|\underset{\text{d}}{100}|\underset{\text{o}}{111}|\underset{\text{g}}{103}|\underset{\text{r}}{114}|\underset{\text{a}}{97}|\underset{\text{m}}{109}].$$

Remember that an ASCII character encoding takes a value between 0 to 255. Thus all ASCII characters (pigeons) fit into $2^8 = 256$ pigeonholes. The number 2^8 is called one **byte**, and is the smallest unit of device memory that a processor can refer to. In this case, to save on space, the device allocates 9 bytes of space. In turn, each byte can store a number between 0 and 255 (ASCII character):

$$|\underset{\texttt{0x100000}}{99}|\underset{\texttt{0x100001}}{97}|\underset{\texttt{0x100002}}{116}|\underset{\texttt{0x100003}}{100}|\underset{\texttt{0x100004}}{111}|\underset{\texttt{0x100005}}{103}|\underset{\texttt{0x100006}}{114}|\underset{\texttt{0x100007}}{97}|\underset{\texttt{0x100008}}{109}|.$$

The numbers below are hexadecimal numbers (numbers represented by base 16) that indicates the unique location of the information in device memory, called an **address** of each byte.[17] Unlike us, a processor has no idea what a cat is, what a dog is, and what English is. Thus if a program instructs the device to retrieve a string from address 0x010000, the device will have no choice but to retrieve "catdograin" as a string. To prevent this, we include a special character to the end of each string, called the **null terminator**. Since you and I are humans we'll use the symbol NUL to indicate the null terminator; for the device, we'll use the number 0. Thus to store the strings "cat", "dog", and "ram", the device will allocate $9 + 3$ bytes of space:

$$|\underset{\texttt{0x100000}}{\overset{\text{c}}{99}}|\underset{\texttt{0x100001}}{\overset{\text{a}}{97}}|\underset{\texttt{0x100002}}{\overset{\text{t}}{116}}|\underset{\texttt{0x100003}}{\overset{\text{NUL}}{0}}|\underset{\texttt{0x100004}}{\overset{\text{d}}{100}}|\underset{\texttt{0x100005}}{\overset{\text{o}}{111}}|\underset{\texttt{0x100006}}{\overset{\text{g}}{103}}|\underset{\texttt{0x100007}}{\overset{\text{NUL}}{0}}|\underset{\texttt{0x100008}}{\overset{\text{r}}{114}}|\underset{\texttt{0x100009}}{\overset{\text{a}}{97}}|\underset{\texttt{0x10000A}}{\overset{\text{m}}{109}}|\underset{\texttt{0x10000B}}{\overset{\text{NUL}}{0}}|.$$

It is only incidental that this array ends with NUL. In general, an array will not end in NUL, because there is no such requirement in the C language. The **length** of a string is the number of characters it includes (not including the null terminator). Thus "string" is a string of length 6.

Challenge 5

(a) A **bit** data type (short for *binary digit*) can hold one of two values: `0` or `1`. How many bits is needed to form a byte? (c.f. Challenge 4, where a box takes a value of: 0, 1, or 2)

[17] Technically this is the address of each byte in *virtual memory* (an abstraction of physical memory provided by the operating system), but from the perspective of the programmer and the program, it suffices to think that the address `0x010000` is a unique address, which can be used to retrieve the number 99, which if interpreted as an ASCII character is 'c'. However, realistically speaking, the address `0x010000` in physical memory is probably occupied by something else. The operating system must therefore translate each virtual address to physical memory address.

(b) The number 8 is represented by `1000` in binary (numbers in base 2) since $8 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$, while 7 is represented as `111` because $7 = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$. What is the minimum number of bits needed to store the number 8? What about 7?

(c) Since $8 = 1 \cdot 2^3$, in binary it is 1 followed by three zeroes. Using the fact that $32768 = 1 \cdot 2^{15}$, represent 32768 in binary. What is the minimum number of bits needed to store the number 32768? How many bytes are needed to store the number 32768? Repeat for $65536 = 1 \cdot 2^{16}$.

(d) A byte takes a value between 0 and 255. This is far too small of a unit of storage to store numbers. The usual unit of storage of an integer for a device is 32 bits. The smallest number representable in this case is the binary number consisting of 32 `0`'s, which represents the number 0. The largest number is 32 `1`'s representing $2^{32} - 1$. Why do we subtract a 1?

(e) The form of storage described in part (e) is called an **unsigned int** data type (or simply `unsigned int`), because it does not allow minus signs (negative numbers). It is much more common to use 32 bits, but to allow signed numbers (both positive and begative numbers). This is called **singed int** data type or `int`.[18] The smallest number that can be stored in `int` is -2^{31} while the largest is $2^{31} - 1$. Based on what we've discussed so far, what should we do if we want to store the number 2^{31}? Is it possible to store the number $-2^{31} - 1$?

(f) The number 10000 could mean $1 \cdot 10^5$ in decimal, $1 \cdot 2^5$ in binary, or $1 \cdot 16^5$ in hexadecimal, and more. By default, we write numbers in decimal. If there is a source of confusion, binary numbers will be written as `10000` and hexadecimal numbers will be written as `0x10000`. What is the minimum amount of bytes required to represent the number `0x10000`? Repeat for bits.

(g) A 32 bit operating system uses memory addresses that are 32 bits long. An address for such systems is thus an `unsigned int` (there is no need for negative addresses). Supposed you had a 32 bit operating system on a device on which you attached a 8 GiB (**gibibyte**) memory (a gibibyte is 2^{30} bytes). What is the maximum percentage of this attached memory that your device's operating system can access? What if your operating system was 64 bits?

Challenge 6

(a) Why bother with null terminators, when we could add spaces in between? For example, what if we have devices store the strings "cat", "dog", and "ram" as [␣|c|a|t|␣|d|o|g|␣|r|a|m|␣]?

(b) A `char` data type typically occupies 8 bits in memory (1 byte). Consider the array of four `char`s: [65|0|33|0]. What might a device save the array [65|0|33|0] on memory as? The address that the arrays begins at is not meaningful for us (if interested, lookup "*memory layout*"). Start the address from `0x100000`.

(c) What would you see if you instructed your device to read and print the aforementioned array as an array of ASCII strings? An ASCII table is available on Wikipedia.

(d) What might a processor save the array of three `char`s [72|97|116] on memory as? Start the address from `0x100000`. What might happen if you asked your device to read *a string* of ASCII characters at address `0x100000`?

Now, we wouldn't want to memorize the address of all our arrays. In the C programming language, **pointers** hold memory addresses for future use. Let us refer to the array [65|0|33|0] using the label "`numbers`". In the C language, Challenge 6 part (b) might look like:

```
char numbers[4] = {65, 0, 33, 0}; char *ptr = numbers;
```

There are quite a few things going on in this short code snippet. First, let us consider the state-

[18] We will always assume that the data types `int` and `unsigned int` are 32 bits, because this is most the common.

ment "`char numbers[4] = {65, 0, 33 0};`". This is really two statements "`char numbers[4]; numbers = {65, 0, 33 0};`" in one. The first is a **variable declaration**, where we tell our device that we need an array of length 4 that will contain data of data type `char`; we will label this array as `numbers`. Writing down the number of elements of an array in a variable declaration is optional, so either "`char numbers[]; numbers = {65, 0, 33 0};`" or "`char numbers[] = {65, 0, 33 0};`" would have been fine too. The second statement "`numbers = {65, 0, 33, 0};`" tells the device to *assign* the array {65, 0, 33, 0} to the array `numbers`; this is an example of **variable assignment**. Notice that the '`=`' symbol means assign; the '`=`' does *not* mean equals. Another example is "`int a = 5; a = 6;`". The first statement "`int a = 5;`" means create an integer, which we'll label as `a`, and assign the value 5 to `a`. The second statement "`a = 6;`" means assign the value of 6 to `a`. Your **compiler** is responsible for turning your human readable code into machine code (`0`'s and `1`'s). The symbol '`;`' tells your compiler that a statement is complete.

Let us turn to the statement: "`char *ptr = numbers;`". This statement is also two statements in one: "`char *ptr; ptr = numbers;`". First, we do a variable declaration, declaring a variable `ptr`—the symbol '`*`' in front of the variable name `ptr` tells your compiler that `ptr` is a pointer (the significance of `char` will be clear later). Second, we do a variable assignment, assigning the memory address of `numbers` into `ptr`. The memory address of an array is the address of its 0th element, in our example, `0x100000`. Thus `ptr` holds the address `0x100000`.

Challenge 7 The symbol "`==`" is used to check whether two objects are equal, while the symbol "`!=`" is used to check whether two objects are *not* equal. Which of the following statements are *always* true?

(i) `int a = 5; int b = 5; a == b;`
(ii) `int a = 5; int b = 6; b = 5; a == b;`
(iii) `int a[2] = {2, 1}; int *ptr = a; int b = 0x100000; a != b;`
(iv) `int a = 0; int b; a != b;`
(v) `int a = 2; int b = 4; b = 2 * a; a == b;` (the symbol `*` means to multiply)
(vi) `int a = 2; int b = 4; b = b + 2; a == b;`

As you may have noticed, a symbol may mean two different things. If the symbol `*` is used between two objects, it is used as a **binary operator**, and means to multiply (for example, `2 * 3` is the number `6`). If the symbol `*` is used in front of a single object (for example, in the expression "`char *ptr`", "`char`" is a data type, not an object), then it is used as a **unary operator**. When '`*`' is used as a unary operator during a variable declaration, it means that the variable is a pointer.

Other familiar examples of operators are `+` and `-`. In the statements "`int a = +2; int b = -2;`", the symbols `+` and `-` are used as unary operators to indicate the sign of `a` and `b`, respectively. In the statements "`int a = 2 + 2; int b = 2 - 2;`", the symbols `+` and `-` are used as the binary operators addition and subtraction, respectively. We need to be careful when using '`-`' as a unary/binary operator with `unsigned int`'s. For example, `2 - 3` results in `-1`, which is not an `unsigned int`, even though `2` and `3` are `unsigned int`'s.

Challenge 8

(a) The following statements are always true. What are the behaviors of the unary operator '`++`' (called the **increment operator**)?[19]
(i) `int x = 2; ++x; x == 3;`

[19] The unary operator '`--`' (called the **decrement operator**) is defined analogously.

(ii) `int x = 1; int y; y = ++x; x == 2; y == 2;`
(iii) `int x = 2; x++; x == 3;`
(iv) `int x = 1; int y; y = x++; x == 2; y == 1;`

(b) What are all the possible values of `y` from the statements "`int x = 1; int y = x - ++x;`"?

Magnitudes of sorts

The most useful property of a number is that they convey a notion of magnitude/size. Because arrays are made up of numbers, it is natural to ascribe some notion of magnitude to arrays.

One simple method for calculating the magnitude of an array might be to simply add all the entries up. For example, the array $[72|97|116|0]$ might have the magnitude $72+97+116+0 = 285$. The array ["*hat*"] could also have the magnitude 285, if we are using ASCII encoding.[20] What about the array $[-2^{32}|2^{50}|2^{32}+1|-2^{50}]$? It doesn't seem right that this array should be ascribed the measly value of 1 for its magnitude, even though each entries are much grander than say, $[72|97|116|0]$.

There are two common ways to approach this. The **absolute value** of a number is the same number, but with negative signs replaced with a positive sign, if applicable. Thus an absolute value is a way of measuring magnitude for numbers. For example, the absolute value of 10 is 10, and the absolute value of -10 is also 10. The first approach to assigning magnitude to an array is to add up the absolute value of each element of the array. In this first approach, the magnitude of $[-2]$ is 2 and the magnitude of $[-2^{32}|2^{50}|2^{32}|-2^{50}]$ is $2^{33}+2^{51}$.

The second approach is to take the square of each element of the array, then add them all up. Now squaring numbers can make the sum too large, so we take the square root of the sum. Taking the square root also has the effect of keeping the magnitude of a number equal with the magnitude of an array of length 1. In the second approach, the magnitude of the magnitude of $[-2]$ is also 2 and the magnitude of $[-2^{32}|2^{50}|2^{32}|-2^{50}]$ is $\sqrt{2^{34}+2^{52}}$. Although both approaches are valid, the second is more common, and is the one we will use.

Let us see this second rule in action, by showing it failing to work. The Schrödinger equation in three dimensions (as you know, we live in three spatial "dimensions") states that

$$i\hbar\frac{\partial\Psi(x,t)}{\partial t} = -\frac{\hbar^2}{2m}\nabla^2\Psi(x,t) + V(x,t)\Psi(x,t).$$

We will have more time later to discuss this equation. In the mean time, let us look at the symbol i. Now, the letter i (stylized in italics to emphasize that it is being used as a mathematical symbol), is typically used to denote some counting number that is being used as an index. Here, the symbol i, defined by the rule: $i^2 = -1$, is an example of a *complex number*. Using the second rule, the array $[i, 1, 0]$ has a magnitude of 0, which what we wanted to avoid. However, all is not lost, because a complex number is actually an array in disguise. To be more precise, a complex number is a twople, where the first element is called the *real part*, and the second element is called the *imaginary part*.[21] Since a twople is an array, we can use the second rule for computing magnitude, and it turns out that all is well.[22] All that we'll need for now is that complex numbers are arrays, and they have a notion of magnitude. In both the case of absolute values of numbers and magnitude of complex

[20] The notion of magnitude will depend on the encoding scheme used, so we will need to be consistent.
[21] The *twople* is defined in the Appendix.
[22] We will save the algebraic intricacies of complex numbers for later.

valued objects, the symbol | | is used. For example, $|-10| = 10$ and $|10| = 10$, and the expression "$|i|$" means the magnitude of the complex number i.

Challenge 9

```
char numbers1[8] = {0, 1, 0, 2, 0, 3, 0, 4}; char *ptr1 = numbers1;
int numbers2[2] = {0x00010002, 0x00030004}; int *ptr2 = numbers2;
```

(a) We access elements of arrays using the element's index with the notation: "array[index]". For example, `numbers1[0]` is 0 and `numbers1[1]` is 1. What are `numbers1[3]` and `numbers2[1]`?

(b) As in Challenge 6, write down how your device might store the arrays `numbers1` and `numbers2` in memory, byte by byte. Assume that the value of `ptr1` is `0x100000` and the value of `ptr2` is `0x200000`. For `numbers2`, use the **big–endian** order, where each byte in memory is stored in order of appearance. For example, the two byte hexadecimal number `0x1011` is broken down into two hexadecimal numbers `0x10` and `0x11`, where `0x10` is stored before `0x11`. For consistency, store each elements of `numbers1` as hexadecimal numbers of length two (just pad each number by adding a `0x0` infront).

(c) Recall that "`a++`" means increment the value of `a` by 1. However, if `a` is a pointer, what "`a++`" means depends on how `a` was created. If `a` was created by the variable declaration "`int *a`" (and points to a valid address), then "`a++`" means increment `a` by 4, assuming `int` data types are held in 4 bytes. Suppose `a` was created by the variable declaration "`char *a`". What does "`a++`" mean? Suppose we add the expression "`char **pptr = &ptr1;`" to the above code, which creates a pointer to a pointer, `pptr`, that points to `ptr1`. What does `pptr++` mean?

(d) Suppose `ptr` is a pointer that stores some address. If we place the unary operator `*` in front, as in: "`*ptr`", this expression refers to the data stored in that address. For example, after `ptr` was created by variable declaration "`int *ptr`", then a subsequent expression "`*ptr`" refers to the data stored in the four bytes starting at address `ptr`. Thus "`*ptr1`" is 0 (or `0x00`). What are "`*(ptr1+1)`", "`*(ptr1+2)`", "`*ptr2`", and "`*(ptr2+1)`"? What is the relationship between `numbers1[i]` and `*(ptr1 + i)`? Repeat for `numbers2[i]` and `*(ptr2 + i)`.

These calculations are examples of **pointer arithmetic**. Pointers are incredibly powerful in the hands of an experienced *and* careful programmer, but dangerous, in general, especially for beginners. Most modern programming language do not even give programmers the option of using pointers.[23]

We end this discussion by making the definition of a magnitude of an array precise. Suppose we have a long array: [1|10|15|2|50|70|31|11|12|150|121]; call it array A. Writing down its magnitude as $\sqrt{1^2 + 10^2 + 15^2 + 2^2 + 50^2 + 70^2 + 31^2 + 11^2 + 12^2 + 150^2 + 121^2}$ is a chore, and we don't want to do this. Notice that the previous expression can be written as

$$\sqrt{A[0]^2 + A[1]^2 + A[2]^2 + A[3]^2 + A[4]^2 + A[5]^2 + A[6]^2 + A[7]^2 + A[8]^2 + A[9]^2}.$$

There is a lot of repetition, the indexes being the main difference. For an array A and counting number n, the expression $\sum_{i=0}^{n} A[i]$ is defined to mean $A[0] + A[1] + A[2] + \cdots + A[n-1] + A[n]$. The Greek letter Σ, capital sigma, indicates summation. Thus, the magnitude of an array A is

$$\sqrt{\sum_{i=0}^{n} (A[i])^2}, \quad \text{where } n = \text{len}(A) - 1.$$

[23] Should a programmer have the power to utilize `ptr++` in part (c) of Challenge 9? Admittedly, this is a silly example. Most pointer problems are very subtle, and greater power and flexibility comes at a price.

Recursive Definitions

Suppose we have a complex number $a = (0|1)$. What is $|a|$? Since a is a twople, $\text{len}(a)$ is 2, and its magnitude is $\sqrt{\sum_{i=0}^{1}(a[i])^2}$. But the sum $\sum_{i=0}^{n} a[i]$ is defined to mean $a[0] + a[1] + a[2] + \cdots + a[n-1] + a[n]$. Blindly applying the definition, we get

$$\sqrt{\sum_{i=0}^{1}(a[i])^2} = \sqrt{(a[0])^2 + (a[1])^2 + (a[2])^2 + (a[0])^2 + (a[1])^2} = \sqrt{0^2 + 1^2 + (???)^2 + 0^2 + 1^2}$$

which is undefined. This example underscores the need for precise definitions.

Definition 1. Let A be a list of numbers and let j, k be counting numbers such that $j \leq \text{len}(A)$ and $k \leq \text{len}(A)$. The expression $\sum_{i=j}^{k} A[i]$ is defined as follows.

(a) If $j > k$, then $\sum_{i=j}^{k} A[i] = 0$.

(b) Otherwise $j \leq k$,[24] in which case, $\sum_{i=j}^{k} A[i] = A[k] + \underline{\sum_{i=j}^{k-1} A[i]}$.

This definition is **recursive** because the definition references itself (underlined part). It is precise enough that a computer can understand it with minimal changes. Part (b) of the definition is called the **recursive case** because it is a case that references itself. Part (a) has no reference to itself and is called the **base case**. A **subroutine**, or a *function*, is a sequence of instructions, organized as one unit, written to perform a definite task. Below is a recursive function `summation` that calculates the summation of `int`s, stored in an array `array`. The keyword "`return`" has two roles: returning the value specified after the keyword and then terminating the subroutine.

```
int summation(int j, int k, int array[]) {
  if (j > k)
     return 0;
  else
     return array[k] + summation(j, k-1, array);
 }
```

Challenge 10

(a) We may add 0 as many times as we want without changing the result of a summation. Thus, we say that 0 is an **additive identity**. What is the analogue of 0 in multiplication (a **multiplicative identity**)?

(b) Make the necessary tweaks to Definition 1 to formulate a definition of a **finite product** "$\prod_{i=j}^{k} A[j]$", where A is a list of numbers.

(c) Write a subroutine `product` that calculates finite products by making the necessary changes to the subroutine `summation`.

Challenge 11 Subroutine `summation` is a faithful reproduction of the Definition 1, and it works. Nevertheless, there is a huge problem with it; what is it? How might you modify the first line of the subroutine (called the **function header**) "`int summation(...)  {`" to help prevent this?

[24] Notice that if j is not greater than k, then $j \leq k$.

Definition 1 is more general than it seems. For example, the magnitude of an array A can be written $\sum_{i=0}^{\text{len}(A')-1} A'[i]$, where A' is the array defined by $A'[i] = (A[i])^2$. I recommend that you go through some examples with various arrays to convince yourself of the correctness of definition 1.

We will be seeing more complicated recursive programs as we go. In preparation for this, here are two rather straightforward recursive subroutines. A **factorial** of a natural number n, written $n!$ is defined as 1 if $n = 0$ (base case) and $n \cdot (n-1)!$ for $n > 0$ (recursive case). Thus $1! = 1$ and

$$5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120.$$

The subroutine `factorial` below calculates `n`! for an `unsigned int`, `n`:

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

A **Fibonacci number** F_k for a natural number k is defined as

(a) If $k = 0$, then 0, (base case 1)
(b) Otherwise, if $k = 1$, then 1, (base case 2)
(c) Otherwise, $F_k = F_{k-1} + F_{k-2}$. (recursive case)

The first few values of the Fibonacci numbers are:

$$F_0 = 0,\ F_1 = 1,\ F_2 = 1,\ F_3 = 2,\ F_4 = 3,\ F_5 = 5,\ F_6 = 8,\ F_7 = 13,\ F_8 = 21,\ F_9 = 34, \ldots.$$

These can be calculated by the subroutine below (the line numbers on the left are not code):

```
1  int fibonacci(unsigned int k) {
2     if (k == 0)
3        return 0;
4     else if (k == 1)
5        return 1;
6     else
7        return fibonacci(k - 1) + fibonacci(k - 2);
8   }
```

Challenge 12 Why is the final value of `y` different in the left and right code below?

```
1  int x = 0;
2  int y = 0;
3  if (x == 0)
4     y++;
5  else if (y == 1)
6     y--;
```

```
1  int x = 0;
2  int y = 0;
3  if (x == 0)
4     y++;
5  if (y == 1)
6     y--;
```

1.4 Search Algorithms

We now have all the basics down, and we can finally discuss how to do things with arrays. Arrays are good for storing information. What good are arrays if we can't find what we've stored?

Linear search

The most important thing about searching for an item in a list would be to go through each element of the list without skipping. If we were to skip an element, we might miss the thing we were looking for. The subroutine `linear_search` searches for item `item` in a nonempty array `array` of `int`'s, by comparing `item` with each element of `array` from index `i` to index `end`, inclusive.

```
1  int linear_search(int item, int i, int array[], int end) {
2     if (i > end)
3        return -1;
4     else if (array[i] == item)
5        return i;
6     else
7        return linear_search(item, i+1, array, end);
8  }
```

For example, let

```
int array[] = {6, 2, 3, 1, 7, 8};
```

A **function call** is an expression that runs a function (subroutine) with all the required inputs specified. The function call "`linear_search(2, 0, array, 5)`" would do a search on the entire array, and would return the index `1`. On the other hand, function call "`linear_search(4, 0, array, 5)`" or "`linear_search(6, 1, array, 5)`" would both return `-1`, as no match is found.

Although we have seen several recursive subroutines, all prior ones were merely translating *definitions* into code. The subroutine `linear_search` is the first that we can call an *algorithm*, and we need to verify that it works correctly. After all, what good is an algorithm if it is incorrect?

To see if a recursive algorithm is correct, we: check that the base cases are correct, check that the recursive (function) calls are correct, and check that the recursive cases are correct. That is:

(0) In order to understand what the algorithm does, we select a few small example inputs. Call the algorithm on each example inputs, and carefully step through each instruction that is run.

(1) Examine the algorithm to check that the base case is correct.

(2a) Verify that the input of the algorithm is strictly *larger* than the input to each recursive call. A correct recursive algorithm will break down a problem into the base case(s).

(2b) *Assuming* all recursive *calls* are correct, check that the recursive *cases* are correct.

Let us go over these steps for `linear_search`. Suppose we wish to find '`2`' in the `array` $[1, 2, 3]$. We want to search the whole array, so `i` is 0 and `end` is 2. Line `2` checks whether `i > end`, that is `0 > 2`, which is false. This means we skip line `3` and we *go to* line `4`, where we check if `array[0]` equals '`2`'. Since `array[0]` is `1` and not `2`, we skip line `5` and run the instructions inside the *else* block, in this case this is just line `7`. Line `7` is a recursive call "`linear_search(2, 1, array, 2)`". This transports us to line `2` once again; since `1 > 2` is false, we go to line `4`. This time, `array[1] == 2` is true, so we run line `5`: return the index 1 and terminate the subroutine. This is Step 0.

Next is Step 1. There are two questions a search algorithm must ask. First, is the index we're looking at within the bounds of the array? In other words, is `i` $\leq$ `end`? If the answer to the first question to search algorithm is a no, we need to terminate immediately. Otherwise, if the answer to the first question is a yes, then the algorithm must ask the second question: is `array[i]` the item we are looking for? If the answer to the second question is a yes, we must return the index `i` and terminate. If the answer to the second question is a no, we must continue searching, via a recursive call. All answers to the questions are handled correctly in `linear_search`, so Step 1 is complete.

Suppose we ran a subroutine in a computer, and 2 hours later, the subroutine has still not completed. Is the subroutine still running because there is a bug in our code? Or is the subroutine still running because there are still instructions left to run? How longer should we continue to wait for the subroutine to run? Step 2a ensures that our recursive algorithm eventually terminates. Indeed, in our `linear_search` code, each recursive call "`linear_search(item, i+1, array, end)`" reduces the length of the array we need to search in by 1. Thus for an array of length n, the subroutine will surely terminate after n recursive calls or less. Step 2a is done.

A recursive algorithms can go wrong in three places: its base cases, its recursive calls, and its recursive cases. Steps 1 and 2a handle the first two, while Step 2b handles the third. There is only one recursive case in `linear_search`: when `array[i] != item`. Assume that the recursive call "`linear_search(item, i+1, array, end)`" is correct. This means that the recursive call will return '`-1`' if `[array[i+1]| array[i+2]|...|array[end]]` does not contain `item`, and will return the index of `item` otherwise. Assume that the recursive call returned '`-1`'. The recursive call occurred because `array[i] != item`, so we conclude that the array does not contain `item` from index `i` to `end`, which is correct. Otherwise, the recursive call returned the index of `item`; since `array[i] != item`, this is the correct index. The recursive case is correct, and Step 2b is done.

Let us conclude our analysis of `linear_search` with some comments. Because `linear_search` is so simple, some of the steps were very easy. Particularly worrisome might be our Step 2b, where it *almost* seemed like we were assuming what we are trying to check! At this stage, I can only assure you that going through these steps for many different algorithms is really the only way to get used to it.

Binary search

In a **linear search**, we go through each item, one by one. Is there any way to speed this up? There is something about going through every single item that is unappealing; after all, there could be billions, or perhaps trillions of items in our array. With any old array, skipping through items might cause problems, so let us make an additional assumption that might help.

What if our array was *sorted*? Suppose we wanted to find the index of string "Racket" in the sorted array of strings:

$$[\underset{0}{\text{“C”}}|\underset{1}{\text{“C++”}}|\underset{2}{\text{“Java”}}|\underset{3}{\text{“Javascript”}}|\underset{4}{\text{“Lisp”}}|\underset{5}{\text{“Perl”}}|\underset{6}{\text{“Prolog”}}|\underset{7}{\text{“Python”}}|\underset{8}{\text{“Racket”}}|\underset{9}{\text{“Rust”}}|\underset{10}{\text{“Scheme”}}]$$

Once an array is sorted, there is no need to search through every single item. If we were searching for a word in a dictionary, we wouldn't go through every single word! Here's a better way: look at the item at the middle of the array: in the above array, that would be the string "Perl".[25] We check whether "Perl" is equal to "Racket": since "Perl" $\neq$ "Racket", we must continue our search.

[25] If the array has a length that is a multiple of two (an **even** length), then there is no "middle" index. In this case, we agree to choose the highest index in the first half of the array as the "middle index".

Since we know "Perl" $\neq$ "Racket", we must determine whether "Racket" is ordered before, or after "Perl" (as a dictionary would). Since 'P' appears after 'R', we write that "Perl" $<$ "Racket".

We now break up the array into two halves: one half consisting of the elements appearing before the middle item "Perl", and the other consisting of the elements appearing after the middle item:

$$\overbrace{[\underset{0}{\text{“C”}}|\underset{1}{\text{“C++”}}|\underset{2}{\text{“Java”}}|\underset{3}{\text{“Javascript”}}|\underset{4}{\text{“Lisp”}}]}^{\text{lower half}} \quad \overbrace{[\underset{0}{\text{“Prolog”}}|\underset{1}{\text{“Python”}}|\underset{2}{\text{“Racket”}}|\underset{3}{\text{“Rust”}}|\underset{4}{\text{“Scheme”}}]}^{\text{upper half}}$$

For each string a in the lower half, the statement $a <$ "Perl" is true. Since $a <$"Perl" $<$ "Racket", we see that no string a in the lower half has any hope of being "Racket". On the other hand, each string b in the upper half satisfies $b >$ "Perl", so there is a possibility that one of b equals "Racket". Thus only the upper half is relevant, so we disregard the lower half. We repeat our steps on the upper half until an item is found, or there is no array left to halve. This is called **binary search**.

The first step of binary search assumes we can always order any two strings alphanumerically. The property that: for any two objects a and b, exactly one of three possibilities: $a < b$, $a = b$, or $a > b$ holds is called **trichotomy**. Thus, we say that the alphanumeric order on the set of strings is *trichotomous*. In the second step of binary search, we needed the property that: for any three strings a, b, and c, if $a < b$ and $b < c$, then $a < c$. This property is called **transitivity**. The alphanumeric order on the set of strings is *transitive*, and this allows us to systematically exclude halves of arrays with binary search. A transitive and trichotomous ordering is called a **total order**.

Challenge 13

(a) Use an ASCII table to determine which the following orderings on strings are true:
(i) "A" $>$ "a" (ii) "C" $\leq$ "C" (iii) "abjad" $<$ "123" (iv) "aact" $<$ "a123" (v) "abba" $\leq$ "abb?"

(b) Referring back to our sorted array, use binary search to find the index of the string "C++".

Challenge 14 A set with a total order is called an **ordered set**. Let A be set of all lists of ASCII characters of length two (e.g. the list ["5", "o"] $\in A$, but ["5o", "6"] $\notin A$ because "5o" is not an ASCII character). For $a \in A$ and $b \in A$, define the **lexicographical order** as $a < b$ if $a[0] < b[0]$, or if $a[0] = b[0]$ with $a[1] < b[1]$. (E.g. ["a"|"b"] $<$ ["b"|"b"] and ["a"|"a"] $<$ ["a"|"b"].) Use the fact that the set of ASCII characters is an ordered set to prove that A is an ordered set.[26]

Challenge 15 Challenge 13 part (b) assumed ASCII strings of length 10 form an ordered set.[27] We prove this fact. Let A be the set of lists of length 2 whose entries are elements of a nonempty ordered set. Prove that A is an ordered set with the lexicographical order. Conclude that ASCII strings of length 4 form an ordered set. Prove that ASCII strings of length 10 form an ordered set.

The usual order on `int`'s, as in `1 < 2`, makes the set of `int`'s an ordered set. Thus binary search works on any array of `int`'s. The subroutine `binary_search` does binary search on a nonempty array of `int`'s. "`/`" is the division binary operator. When dividing two `int`'s, the division operator *truncates* all decimal points. For example, "`10 / 3`" evaluates to `3`, while "`12 / 3`" evaluates to `4`. For a number α, we write $\lfloor \alpha \rfloor$ (the **floor** of α) to denote α with all decimal points truncated. For example, $\lfloor 3.14 \rfloor = 3$ and $\lfloor 3 \rfloor = 3$. The expression "`a / b`" for two `int`'s `a`, `b` means the `int` $\left\lfloor \frac{\texttt{a}}{\texttt{b}} \right\rfloor$.

Challenge 16 Verify that `binary_search` works by going through the four steps we used with `linear_search`. For Step 0, search for the numbers 1 and 3 on the arrays: [1], [1|2], [1|2|3], and [1|2|4|5], each from index 0 to its final element. Do not repeat work you have already done!

[26] Hint: to prove that the lexicographical order on A is trichotomous requires showing that for any $a \in A$ and $b \in A$, exactly one of $a < b$, $a = b$, or $a > b$ holds. Use the fact that $a[0], b[0], a[0], a[1]$ are elements of ordered sets.

[27] Strings of length less than 10 can be extended to 10 characters using a fictional character with value -1.

```
1  int binary_search(int item, int left, int array[], int right) {
2     if (left > right)
3        return -1;
4     else {
5        int mid = (left + right)/2;
6        if (array[mid] == item)
7           return mid;
8        else if (item < array[mid])
9           return binary_search(item, left, array, mid-1);
10       else
11          return binary_search(item, mid+1, array, right);
12    }
13 }
```

Debugging

Consider the following subroutine.

```
1  int enjoy_pizza(int pizza_slices) {
2     while (pizza_slices > 0) {
3        eat();
4        pizza_slices--;
5     }
6     return 0;
7  }
```

Subroutine `enjoy_pizza` is not recursive because its **function body** (everything after the function header), contain no recursive calls. Instead, the keyword "`while`" facilitates continued eating, until all pizza slices run out. The "`while`" keyword is thus an instruction to *iteratively* execute all code inside the curly braces (lines `3` and `4`) while the condition `pizza_slices > 0` continues to be true.

So far, with the exception of `enjoy_pizza` (which uses an undefined subroutine `eat()`), we have been working with subroutines that work. However, it is often the case that one must deal with several versions of subroutines that don't work, before settling on one that works. The process of finding issues in code, called **bugs**, and correcting them, is called **debugging**. Before we get to some debugging, here are two key definitions.

By an **algorithm**, we mean a finite sequence of instructions for solving a well-defined problem. In order to solve a problem we are facing, an algorithm might need "inputs". For example, to search for an item in an array, at the very least, we will require an array to search in, and an item to search for. The **preconditions** of an algorithms are the conditions that the input of the algorithm must satisfy, in order for the algorithm to work. For example, `enjoy_pizza` assumes that its input `pizza_slices` is an `int`, and not a string, or some other data type.

Challenge 17

(a) What datatype would be better for storing a *counting* number, `int` or `unsigned int`?

(b) The program below counts down from a user provided counting number. For example, if a user executes this program and provides a number, say 5, then the program will count down from 5 to 0, with a one second pause after each count down: 5! 4! 3! 2! 1! 0![28] The remarks that follow the "`//`" symbol are comments that are ignored by the compiler. Assume that all preconditions for normal use of this program are satisfied.

```
1  #include <stdio.h> // needed to use printf() subroutine
2  #include <stdlib.h> // needed to use atoi() subroutine
3  #include <unistd.h> // needed to use sleep() subroutine
4
5  int main(int argc, char const *argv[]) {
6     unsigned int n = atoi(argv[1]); // convert argv[1] to unsigned int
7
8     while (n >= 0) {
9        printf("%d!/n", n); // print current countdown number
10       sleep(1); // pause program for 1 second
11       n--;
12    }
13    return 0;
14 }
```

Actually, the above program does *not* do what I claimed it does, due to one subtle bug in one line of code. You know everything needed to identify it! I leave it to you to debug the above program. Make the necessary additions and/or deletions to fix the above program.[29]

(c) In Challenge 16, we verified that `binary_search` works fine. We could develop more mathematics to give a rigorous proof that the subroutine is correct. Nevertheless, there is a subtle bug in `binary_search` that was unnoticed for decades until 2006. Assume an `int` uses 4 bytes. What might cause a problem in line `5` in `binary_search`, where we add two `int`'s?

(d) What array length will trigger the bug in `binary_search`, no matter what `array` contains?

(e) Use a calculator to find the *minimum* array length that will trigger the bug in `binary_search`. Why did it take so long to find this bug?

(f) What is the difference between "`(left + right)/2`" and "`left + ((right - left)/2)`"?

(g) Suppose we corrected the bug in line `5` of `binary_search` by using part (f). What is an array length that will cause a problem in our corrected code?

(h) Why do we say that the expression "`(left + right)/2`" is a bug, when anyone can create arbitrarily large arrays that could trigger the problem in part (c)? (Remember, `binary_search` is an *algorithm*.) Both bugs in part (b) and part (c) are examples of an **(integer) overflow**. The problem in part (b), in which a program never terminates, is called an **infinite loop**.

[28] Upon user input, the number 5 is stored as a string at index 1 in an array of strings `argv`. The string "`5`" is converted safely to a number using the subroutine `atoi`, and stored in variable `n` as an unsigned int (line `6`).

[29] Hint: to debug code, one must first go through the code *carefully* with a simple case.

Divide and conquer

In Challenge 3, we found out how to do multiplication of two numbers between 0 and 20 in our heads, by turning the multiplication of those numbers into three additions, a padding by zero, and one multiplication:

$$(a+b)(c+d) = ac + ad + bc + bd \Longrightarrow (10+x)(10+y) = 10(10+x+y) + xy.$$

Can we can do something similar for general integers?

Suppose we only knew how to multiply two single digit natural numbers, but we wished to multiply two n-digit integers. For example, $2562 \cdot 3017$ is the multiplication of two 4-digit integers, but we only know the multiplication table up to 9×9. We could do the manual multiplication like we learned in grade school, or we could try our trick from binary search.

Similar to binary search, we divide up each n-digit integer into two halves. For example, 26127 can be divided into the first half: 261 and a second half: 27 (as with arrays, not all n-digit integers can be divided evenly into two halves). For an `n`-digit `int x`, we can do this in C by:

```
#include <math.h> // needed to use the pow() subroutine
int p = pow(10, n/2); // p = 10^⌊n/2⌋
x1 = x / p; // x1 = ⌊x/p⌋
x2 = x % p;
```

The `pow(a, b)` subroutine calculates the power a^b. The binary operator `%` returns the remainder of a division. *In programming, the remainder of a division is allowed to be negative.*[30] For example, `5 % 2` is `1`, `26127 % 100` is `27`, and `-26127 % 100` is `-27`. The fact that the expression "`(x / p) * p + (x % p)`" is always equal to `x` for nonzero `p` is what allows us to split an `int` into two halves using the above C code. For example, `(26127 / 100) * 100` is `26100` and `26127 % 100` is `27`, so their sum is `26127`. Similarly, `(-26127 / 100) * 100 + (-26127 % 100)` equals `-26127`.

Challenge 18 (Fast integer multiplication)

(a) For an `unsigned int x`, is the expression "`x/2 + x/2 == x`" always true?

(b) What do the function calls "`digitlen(5)`", "`digitlen(100)`", and "`digitlen(199)`" return?

```
int digitlen(unsigned int n) {
  if (n < 10)
     return 1;
  else
     return 1 + digitlen(n / 10);
}
```

(c) Here is the C code for breaking up an integer to two halves, translated to math. For two integers x and y, both n-digits long, let a, b, c, d be integers that satisfy the formula:

$$x = 10^{\lfloor n/2 \rfloor} a + b \text{ and } y = 10^{\lfloor n/2 \rfloor} c + d,$$

[30] In mathematics, the convention is that the remainder of a division is always nonnegative.

where $0 \le b, d < 10^{\lfloor n/2 \rfloor}$. We call a and c the *first half* of x and y, respectively, and b and d the *second half* of x and y, respectively. Find a, b, c, and d when $x = 25621$ and $y = 30171$.

(d) (C.f. Challenge 3) Let x and y be n-digit integers. Let a and c be the first half of x and y, respectively, and let b and d be the second half of x and y, respectively. Show that

$$x \cdot y = (a \cdot c)10^{\lfloor n/2 \rfloor + \lfloor n/2 \rfloor} + (a \cdot d + b \cdot c)10^{\lfloor n/2 \rfloor} + b \cdot d. \tag{1.3}$$

(e) There are four multiplications in formula 1.3. In 1960, Anatoly Karatsuba discovered that we can swap out two multiplications for one. Use the fact that the product of two negative numbers is *positive* to show that $a \cdot d + b \cdot c = a \cdot c + b \cdot d - (a-b) \cdot (c-d)$. Modify formula 1.3 to use three multiplications. Notice the expression $(a-b) \cdot (c-d)$ involves just one multiplication.

(f) Use formula 1.3 to finish the recursive subroutine `rec_prod`. Assume both `x` and `y` have the same number of digits. According to formula 1.3, how many recursion calls are needed?

```
1  int rec_prod(int x, int y) {
2     int n = ( x >= 0 ?  digitlen(x) :  digitlen(-x) );
3     if (n == 1)
4        return x * y;
5     else {
6        int e1 = pow(10, n/2 + n/2);
7        int e2 = pow(10, n/2);
8        int a = x / e2;
9        int b = x % e2;
10       int c = y / e2;
11       int d = y % e2;
.        // your code here
.     }
.  }
```

Line 2 uses the operator "`a ? b : c`", which means: `b` if `a` is true, otherwise, `c`. We use it to make sure we call `digitlen` with nonnegative `int`'s. The entirety of line 2 is shorthand for

```
int n;
if (x >= 0)
   n = digitlen(x);
else
   n = digitlen(-x);
```

(g) Write a subroutine `fast_prod` using your formula from part (e), and only three recursive calls. This algorithm is called the **Karatsuba algorithm** for fast multiplication (Karatsuba 1962).

A **divide and conquer** algorithm recursively breaks down a problem into several subproblems, each of which can be handled with ease. The Karatsuba algorithm and `rec_prod` are both divide and conquer algorithms, because we break down a long n-digit multiplication, which we don't know how to do, into several single digit multiplications, which we can do easily.

The jump from linear search to binary search was a significant improvement (for sorted arrays). On the other hand, it is not clear that the small substitution trick in the Karatsuba algorithm provides any improvement for integer multiplication. At this stage, we do not have the tools to make quantitative assessments of programs. This marks the point where we need to make the jump from computer *programming* to computer *science*.

Challenge 19 (Odds and ends) I claimed there were four multiplications in formula 1.3. Actually, there are three additional multiplications in the form of '$\lfloor n/2 \rfloor$'. This is because the division $\frac{n}{2}$ is the same as the multiplication $n \times \frac{1}{2}$. More generally, for nonzero b, the division $\frac{a}{b}$ is the same as $a \times \frac{1}{b}$. The goal of this challenge is to see that in computing, the expression '$\lfloor n/2 \rfloor$' for any counting number n is not really a multiplication. Assume an `int` and `unsigned int` occupies 4 bytes each.

(a) Let $a = 2^5 + 2^1$ and $b = 2^5 + 2^1 + 1$. What is the k, the minimum numbers of bits required to store a and b? Convert the decimal numbers a and b into binary, and record each number as an an array of k bits using the big-endian order (see Challenge 9); call these arrays `a` and `b`, respectively.

(b) Regard both a and b as `unsigned int`'s. Perform a **logical right shift** on `a` and `b` by shifting each element of the array to the right. The right most bit is to be discarded, while the leftmost bit is to be replaced by a `0`. Interpret the new array of bits as a binary number, then convert them into decimal numbers.

(c) What is the interpretation of the new decimal numbers, with regards to the original a and b?

(d) Apply another logical right shift. What is the interpretation of the new numbers in relation to a and b?

(e) In C, for an `unsigned int a`, the expression "`a >> n`" is an instruction to apply the logical right shift `n` times on `a`.[31] Change line `5` of the *corrected* `binary_search` to use `>>`.[32]

(f) An alternative way is to do a **type conversion**, also called **type casting**, where we **cast** an `int a` into an `unsigned int` using the expression "`(unsigned int) a`". Fix the *original* `binary_search` code using type conversion and one bit shift. Verify that no overflow occurs.

(g) Regard numbers a and b as two `int`'s, each stored in 32 bits. Apply a **left shift** on the array of bits representing a and b by shifting each element to the left. The rightmost element is to be replaced by a 0 (c.f. Challenge 3, where a multiplication by 10 is not really a multiplication, but just a padding by 0). What decimal numbers do you get? What decimal numbers do you get if you do an additional left shift to each bit array? In C, we write "`a << n`" to apply a left shift `n` times on an `int a`.[33]

Every processor contains an **arithmetic logic unit** (ALU), which supports instant bit shifts and other bit manipulations on integers (represented as bit arrays), whenever you should need them.

[31] *Signed* `int`'s are represented differently in memory using *two's complement* and they use *arithmetic* right shifts. Thus in the expression "`a >> k`", if `a` is an `int`, the operator `>>` means apply an arithmetic right shift k times. If `a` is signed (negative), then arithmetic right shift k times is not necessarily the same as $\lfloor \frac{a}{2^k} \rfloor$. The reason negative integers use a different binary representation is to enable fast bit manipulations by the *arithmetic logic unit*.

[32] Assuming the preconditions are met (each index is less than or equal to $2^{31} - 1$), each midpoint `mid` is positive and has its leftmost bit equal to 0 (this may *not* be true if there is an overflow from the addition of two `int`'s, as the result is undefined), and an arithmetic right shift is equivalent to a logical right shift.

[33] An *arithmetic* left shift is the same as a *logical* left shift, so we just refer to these as left shifts.