Solutions to Challenges in *Calculus*

Dayeol Choi
https://dayeolchoi.com/calculus

2022

1 Reviewing Arithmetic

1.1 Units

Challenge 1 (a)

$$5^5 \times \frac{3}{5^3} = (5 \times 5 \times \cancel{5} \times \cancel{5} \times \cancel{5}) \times \frac{3}{\cancel{5} \times \cancel{5} \times \cancel{5}} = 5 \times 5 \times 3 = 5^2 \times 3,$$

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

(b) Since density is **M**ass/**L**ength^3, we have

$$\frac{\textbf{L}\text{ength}^5 \times \text{density}}{\textbf{T}\text{ime}^2} = \frac{\textbf{L}\text{ength}^5 \times \textbf{M}\text{ass}/\textbf{L}\text{ength}^3}{\textbf{T}\text{ime}^2} = \frac{\textbf{L}\text{ength}^2 \times \textbf{M}\text{ass}}{\textbf{T}\text{ime}^2}.$$

This is the same as Energy.

(c) Only the second expression is valid.

Challenge 2 (a) We do a few check to verify that the equation $E = R^5\rho/t^2$ makes sense. Indeed, an increase in blast radius R is associated with an increase in energy E. If R and time t were the same, but density ρ was higher, then E must have been higher as well. It time t to reach radius R was smaller, then E must have been smaller. These facts agree with the equation $E = R^5\rho/t^2$.

(b) The radius R of the blast looks to be about 150 meters to me. The time t is given as 0.025 seconds. With $\rho = 1.2\text{kg/m}^3$, we have

$$E = \frac{1.5^5 \cdot 100^5\text{m}^5 \cdot 1.2\text{kg/m}^3}{0.025^2\text{s}} = \frac{1.5^5 \cdot 10^{10}\text{m}^5 \cdot 1.2\text{kg/m}^3}{2.5^2 \cdot 10^{-4}\text{s}}$$

and a value E of about $1.5 \times 10^{14}\text{kg}\cdot\text{m}^2/\text{s}^2$.

(c)

$$\frac{1.5 \times 10^{14}\text{ joule}}{4.2 \cdot 10^9} = 0.36 \cdot 10^5\text{ tons of TNT}.$$

Divide by a thousand (10^3) to get that E is about 36 kilotons.

(d) Looking up the yield at https://en.wikipedia.org/wiki/Trinity_(nuclear_test), we can see that the yield was actually about 25 kilotons.

1.2 Exponentiation

Challenge 3 (a) Using the fact that $(a+b)(c+d) = ac + ad + bc + bd$, we get

$$(10+x)(10+y) = 10 \cdot 10 + 10 \cdot y + 10 \cdot x + x \cdot y.$$

The first three terms are multiples of 10. Therefore,

$$(10+x)(10+y) = 10 \cdot 10 + 10 \cdot y + 10 \cdot x + x \cdot y = 10 \cdot (10 + x + y) + xy.$$

(b) Using the formula obtained from the previous part, with $x = 6$ and $y = 4$, we need only add a zero to the sum $10 + 6 + 4$, then add 24 to get 224.
In order to do the multiplication $116 \cdot 114$, we use the same formula from part b, except we switch the '10' with '100':

$$(10+x)(10+y) = 10 \cdot (10+x+y) + xy \rightarrow (100+x)(100+y) = 100 \cdot (100+x+y) + xy.$$

Hence we need only add two zeros to the sum $100 + 16 + 14$, and add $16 \cdot 14$, which we already know to be 224, to get 13224.

Challenge 4 (a) If we allocated 0 boxes per song, then 0 unique signatures are possible. If we allocated 1 box per song, then 3 unique signatures are possible: $\boxed{0}$, $\boxed{1}$, and $\boxed{2}$. If we allocated 2 boxes per song, then the possible signatures are $\boxed{0}\boxed{0}$, $\boxed{0}\boxed{1}$, $\boxed{0}\boxed{2}$, $\boxed{1}\boxed{0}$, $\boxed{1}\boxed{1}$, $\boxed{1}\boxed{2}$, $\boxed{2}\boxed{0}$, $\boxed{2}\boxed{1}$, $\boxed{2}\boxed{2}$. That's $3^2 = 9$ unique signatures. Therefore, if we allocated 5 boxes per song, 3^5 unique signatures are possible; if we allocated 8 boxes per song, then 3^8 unique signatures are possible.

(b) This is a matter of taking a calculator to calculate various powers of 3. If I use a calculator to calculate 3^{16}, I get about 43 million, which is not enough. If I use a calculator to calculate 3^{17}, I get about 128 million, which is enough. So 17 boxes is the minimum number of boxes needed.

(c) Let us examine how 127 is expressed in base 10. In base 10, we need to find what is the largest power of 10 whose multiple fits in 127: that's $10^2 = 100$. Thus $127 = 1 \cdot 10^2 + \ldots$. Next, we see what is the largest power of 10 whose multiple fits in $127 - 100 = 27$: the answer is 10^1. Thus $127 = 1 \cdot 10^2 + 2 \cdot 10 + \ldots$. The remainder is $127 - 100 - 20 = 7$, which is less than 10, so we have

$$127 = 1 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0.$$

How about expressing 120 in base 60? 60^2 is much larger than 120, but $2 \cdot 60^1 \leq 120$. Thus the first digit of 120 is base 60 is 2. In fact, we are done, because $2 \cdot 60^1 = 120$. Thus all powers of 60 that follow, in this case 60^0 contributes 0. Therefore,

$$120 = 2 \cdot 60^1 + 0 \cdot 60^0.$$

In part (b), we needed to know what is the smallest power of 3 that is greater than 100 million. This time, we need to know what is the largest power of 3 whose multiple is fits in 12. Now, $3^2 = 9 < 12$ but $3^3 = 27 > 12$. Thus 3^2 is the largest power whose multiple fits in 12. Thus $12 = 1 \cdot 3^2 + \cdots$. The remainder is $12 - 3^2 = 3$, and so

$$12 = 1 \cdot 3^2 + 1 \cdot 3^1 + 0 \cdot 3^0.$$

(d) From part (c), we found that $12 = 1 \cdot 3^2 + 1 \cdot 3^1 + 0 \cdot 3^0$. Thus 3 is the fewest number of boxes required to record the 12 different pitches.

(e) This time, unlike part (b), we want to know what is the smallest power of 3, when multiplied by 12, will give us a number larger than 100 million. In a calculator, I get $12 \cdot 3^{14}$ is about 57 million, which is not enough. However, $12 \cdot 3^{15}$ is about 172 million. Therefore, we need 3 boxes (to store the starting pitch) and 15 additional boxes: $3 + 15 = 18$ boxes.

(f) In part (b), we found that 17 boxes per song is enough in the original scheme. In part (e), we found that 18 boxes per song is needed for the modified scheme. Therefore, in terms of boxes needed, the original algorithm is better, by a total of 100 million boxes.

Even if there were 200 millions songs, the answer would not change; we'd need 18 boxes per song in the original scheme, compared to the 19 boxes per song in the modified scheme.

(g) Suppose we wanted to find a song by humming the tunes. It would be very difficult to get the exact pitch right. The same goes for getting the exact meter correct. This algorithm doesn't require either such information. Less is more.

1.3 Arrays and Datatypes

Challenge 5 (a) This is the same question as Challenge 4 part (a), with 2 instead of 3. A byte is 2^8, so 8 "boxes" of bits are necessary. Therefore, a byte is 8 bits.

(b) Since $8 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$, 4 bits are needed to store the number 8. Since $7 = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$, 3 bits are needed to store the number 7.

(c) Since $32768 = 1 \cdot 2^{15}$, the number 32768 in binary is 1 followed by 15 zeros. Just as $8 = 1 \cdot 2^3$ needs 4 bits to store, the number $32768 = 1 \cdot 2^{15}$ needs 16 bits to store. Now 16 bits is 2 bytes ($8 \cdot 2 = 16$), so 2 bytes is sufficient to store 32768.

Since $65536 = 1 \cdot 2^{16}$, the number 65536 in binary is 1 followed by 16 zeros. The number $65536 = 1 \cdot 2^{16}$ needs 17 bits to store. Now, 2 bytes is 16 bits, so 2 bytes is not enough to store the number 65536. We need 3 bytes to store the number 65536.

(d) Because we start counting from 0 (32 0's), and not from 1. For example, if we started counting from 0 to 9, that's ten numbers, not nine. On the other hand, if we started counting from 1 to 9, that's nine numbers.

(e) If we want to store the number 2^{31}, then we need to store it as an `unsigned int`. The number $-2^{31} - 1$ is a singed number (negative sign) so an `unsigned int` is out of the question. However, the number $-2^{31} - 1$ is too small to store as a `signed int`, it is not possible to store this number, unless we can allocated more boxes (bytes) to store it. It turns out that a `long` datatype does this, with 64 bits (usually).

(f) The number `0x10000` is $1 \cdot 16^4$. Since each byte can store 16 possible values, we need 5 bytes minimum. Each byte is 8 bits, so 40 bits is necessary to store `0x10000`.

(g) Since a gibibyte is 2^{30} bytes, 8 GiB is $8 \cdot 2^{30} = 2^3 \cdot 2^{30} = 2^{33}$. A 32 bit address can store 2^{32} possible addresses, so only half of 2^{33} address in a 8 GiB memory is reachable: 50%. A 64 bit address can store 2^{64} possible addresses, so a 64bit operating system will have no trouble addressing all 2^{33} address: 100%.

Challenge 6 (a) This wouldn't work because [_|c|a|t|_|d|o|g|_|r|a|m|_] could mean the strings "cat", "dog", and "ram", or it could mean the string "_cat_dog_ram_". A space "_" is a natural part of a string.

(b)

$$|\underset{\texttt{0x100000}}{65}|\underset{\texttt{0x100001}}{0}|\underset{\texttt{0x100002}}{33}|\underset{\texttt{0x100003}}{0}|.$$

(c) The number 65 corresponds to "A" and the number 33 corresponds to "!". So the array is an array of two strings: ["A"|"!"].

(d) The array of three `chars` [72|97|116] might be stored as:

$$|\underset{\texttt{0x100000}}{72}|\underset{\texttt{0x100001}}{97}|\underset{\texttt{0x100002}}{116}|.$$

But what about memory addresses `0x100003` and beyond? We have no idea, it could be anything. It might be:

$$|\underset{\texttt{0x100000}}{72}|\underset{\texttt{0x100001}}{97}|\underset{\texttt{0x100002}}{116}|\underset{\texttt{0x100003}}{0}|,$$

in which case, the device would read "Hat". Or it could be

$$|\underset{\texttt{0x100000}}{72}|\underset{\texttt{0x100001}}{97}|\underset{\texttt{0x100002}}{116}|\underset{\texttt{0x100003}}{116}|\underset{\texttt{0x100004}}{101}|\underset{\texttt{0x100005}}{114}|\underset{\texttt{0x100006}}{0}|,$$

which would read as the string "Hatter". The key idea is that we don't know what lies outside, so the result is undefined.

Challenge 7 The statements in (i), (ii), (v), and (vi) are always true. In (iii), there is no way to know a priori what address array `a` is stored in. In (iv), we have done a variable declaration for `b`, but we haven't assigned any value to `b`; thus there is no way we can know what the value of `b` is.

Challenge 8 (a) From (i) and (iv), we see that both "`x++;`" and "`++x;`" has the effect of increasing the value of x by 1. However, in (ii), we see that the expression "`++x`" is resolved before a value assignment to `y`, whereas in (iv), we see that the expression "`x++`" is resolved after a value assignment to `y`.

(b) We know that the expression "`++x`" occurs before an assignment, but we don't know whether it occurs before or after arithmetic operations like `-`, `+`, or `*`. This exact result of this expression is undefined and depending on the compiler, the result might be `0` or `-1`. Trying compile this code in my compiler, I get a warning, and a value of -1 for `y`. Because results of some expressions in C are undefined in the standard, and left at the hands of compiler developers, one needs to be careful.

Challenge 9 (a) `numbers1[3]` is 2 and `numbers2[1]` is `0x0030004`.

(b)

$$|\underset{\texttt{0x100000}}{\texttt{0x00}}|\underset{\texttt{0x100001}}{\texttt{0x01}}|\underset{\texttt{0x100002}}{\texttt{0x00}}|\underset{\texttt{0x100003}}{\texttt{0x02}}|\underset{\texttt{0x100004}}{\texttt{0x00}}|\underset{\texttt{0x100005}}{\texttt{0x03}}|\underset{\texttt{0x100006}}{\texttt{0x00}}|\underset{\texttt{0x100007}}{\texttt{0x04}}|$$

$$|\underset{\texttt{0x200000}}{\texttt{0x00}}|\underset{\texttt{0x200001}}{\texttt{0x01}}|\underset{\texttt{0x200002}}{\texttt{0x00}}|\underset{\texttt{0x200003}}{\texttt{0x02}}|\underset{\texttt{0x200004}}{\texttt{0x00}}|\underset{\texttt{0x200005}}{\texttt{0x03}}|\underset{\texttt{0x200006}}{\texttt{0x00}}|\underset{\texttt{0x200007}}{\texttt{0x04}}|$$

(c) If `a` was created by "`char *a`" and points to a valid address, then "`a++`" means increment the address stored in `a` by 1. Remember, the smallest address a processor can access is 1 byte, so a byte is the smallest unit (thus 1).

`pptr` is a pointer to an address. If your device has a 64 bit operating system (8 bytes), then `pptr++` increments `pptr` by 8. If your device has a 32 bit operating system (4 bytes), then `pptr++` increments `pptr` by 4.

(d) "`*(ptr1+1)`" is 1, "`*(ptr1+2)`" is 0, "`*ptr2`" is `0x00010002`, and "`*(ptr2+1)`" is `0x00030004`.

`numbers1[i]` is exactly the same as `*(ptr1 + i)` and `numbers2[i]` is the same as `*(ptr2 + i)`. More generally, "`array[i]`" is equivalent to "`*(ptr + i)`", where `ptr` is a pointer to the `array`.

In order for this to work without creating a separate pointer, `array` actually works as a pointer. Thus `array[i]` is equivalent to `*(array + i)`, which equals `*(i + array)`, which means `i[array]`.

Challenge 10 (a) We may multiply 1 as many times as we want to a number without any change. Therefore 1 is a multiplicative identity.

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

(a) If $j > k$, then $\prod_{i=j}^{k} A[i] = 1$.

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

Notice that we define the base case to equal 1. If we set the base case to equal 0, the subroutine will always return a 0.

(c) Below is a straightforward modification.

```
int product(int j, int k, int array[]) {
  if (j > k)
     return 1;
  else
     return array[k] * product(j, k-1, array);
 }
```

This code can give garbage results if we pass in an empty array, say, using the expression "`product(0, 0, array)`", where `int array[] = {};`. We cannot predict what the result will be, as it will depend on what is stored on the memory address of `array`. For us, this is fine, as this problem occurs only when the user uses the subroutine incorrectly. The expression "`product(0, 0, array)`" implicitly assumes `array` contains an element at its 0th index. For an empty array this is false, so this is a case of *garbage in, garbage out*. Nevertheless, as a programmer, it is much better to check for this case. For example, a straightforward fix is to require that the user pass in the length of `array`, then do a check to see if the length is 0.

Challenge 11 As I commented in the solution to Challenge 10c, the subroutine `summation` can return garbage if its input is an empty array. But the root of the problem is *not* that someone might try to use the subroutine on an `array` that is empty.

In Challenge 6, part (d), we saw that there is no telling what lies in an array outside what we have defined. If the user were to use the subroutine using invalid indexes j and/or k, we'd be in trouble. For example, if the user passed an `array` with length 5, but set $j = 0$ and $k = 6$, the subroutine will be computing garbage.

Thus we see the root of the problem is that we have zero checks to make sure that the indexes are within the bounds of the array.

In order to verify that an index is not going outside the length of an array, we need to know the length of the array. Thus the first line of the function (the function header) should be changed to " `int summation(int j, int k, int array[], int n) {` " where `n` is the length of `array`. A subroutine should always require the user to pass in the length of each array, in addition to any array that is being passed on.

In between the function header and the next line of code, there should be a check to verify that `j` and `k` are between 0 and `n` - 1. If this check is not satisfied, we are trespassing on memory space we should not have access to. In a case of memory access violation, the operating system should be notified; for example, by including the instruction "`raise(SIGSEGV);`" (a **segmentation fault**). Once a signal is raised, the programmer can handle this signal. One way to do this is notifying the user of the problem, by printing our some message on screen, and terminating the program, using the instruction "`exit(EXIT_SUCCESS);`". This instructs the program to return the `int` 0 and terminate.

Challenge 12 For the C code on the right, `y` takes the value of 0. However, for the C code on the left, `y` takes the value of 1, and not 0. The first time we saw the keywords "`else if`" is in `fibonacci`. For each natural number k, only three of one cases are possible, analogous to trichotomy for pairs of numbers. So replacing line `3` with `k++;` should never lead to both lines `3` and `5` being run.

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

Thus, if we have

```
if (...)
   ...;
else if (...)
  ...;
else
  ....;
```

only exactly one of the three cases are run.[1]

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

Therefore, in the left code, exactly one of lines `4` and `6` can be run. Since `x == 0` is true, line `4` is run, but not line `6`, and the final value of `y` is 1.

1.4 Searching and Sorting

Challenge 13 (a) (i) is false, "A" < "a" is correct. (ii) is true. (iii) is false, "abjad" > "123" is correct. (iv) is false, "aact" > "a123" is correct. (v) is false, "abba" > "abb?"

(b) Since "C++" < "Perl", we divide up the array into two, then look into the lower half of the array: ["C"$_0$|"C++"$_1$|"Java"$_2$|"Javascript"$_3$|"Lisp"$_4$]. The item in the middle is "Java"; since "C++" < "Java", once again we divide the array into two halves, then look into the lower half: ["C"$_0$|"C++"$_1$]. This array does not have a middle element, so we choose the highest index in the first half of the array. In this case, this is the index 0. Since "C" < "C++", we divide the array into two halves, and look at the upper half: ["C++"$_1$]. Finally, "C++" = "C++", so we have found the string "C++" in index 1. The binary search algorithm is done.

Challenge 14 ASCII characters form an ordered set. The goal of this Challenge is to show that twoples of ASCII characters also form an ordered set (twoples are lists of length 2). That is, twoples of ASCII characters obey both trichotomy and transitivity.

First, we show that trichotomy is satisfied. Let a and b be twoples of ASCII characters. We need to compare the first characters $a[0]$ and $b[0]$. Since $a[0]$ and $b[0]$ are ASCII characters, they are elements of an ordered set, and they will obey trichotomy. Thus there are three possibilities: (i) $a[0] < b[0]$ (ii) $a[0] = b[0]$ (iii) $a[0] > b[0]$. Let us consider case (i). By the definition of lexicographical order, $a < b$, no matter what $a[1]$ and $b[1]$ are. Similarly, in the case (iii), $a > b$. Let us consider case (ii): we now need to look at how $a[1]$ and $b[1]$ compare. Once again, $a[1]$ and $b[1]$ are both ASCII characters, so there are three possibilities: (1) $a[1] < b[1]$ (2) $a[1] = b[1]$ (3) $a[1] > b[1]$. In the first case, $a < b$; in the second case, $a = b$; in the third case $a > b$. We have exhausted all possibilities and we see that trichotomy is satisfied.

Next, we show that transitivity holds. To prove this, we need to show that for twoples of ASCII characters a, b, c, if $a < b$ and $b < c$, then $a < c$. Assume that $a < b$, which means either (i) $a[0] < b[0]$ or (ii) $a[0] = b[0]$ with $a[1] < b[1]$. Let us analyze case (i); since $b < c$ either (1) $b[0] < c[0]$ or (2) $b[0] = c[0]$ with $b[1] < c[1]$. Since the set of ASCII characters obeys

[1] We are allowed to add any number of "`else if (...)`" conditions, depending on how many cases we need to handle.

transitivity, case (1) implies that $a[0] < c[0]$, and so $a < c$. In case (2), $a[0] < b[0] = c[0]$, and so $a < c$; this completes our analysis of case (i). Let us check case (ii), where $a[0] = b[0]$ with $a[1] < b[1]$. Once again, since $b < c$ either (1') $b[0] < c[0]$ or (2') $b[0] = c[0]$ with $b[1] < c[1]$. In the case (1'), $a[0] = b[0] < c[0]$, so $a < c$. In case (2'), by transitivity of S, $a[1] < b[1] < c[1]$ and $a[0] = b[0] = c[0]$. Therefore, $a < c$. In all possibilities, $a < c$, and we see that twoples of ASCII characters are transitive.

Since the set of twoples of ASCII characters is both trichotomous and transitive, it is an ordered set. The proof is complete.

Challenge 15 First a few comments. We can associate each twople of ASCII characters as a string of length 2 by assuming a NUL is always placed at the end. Additionally, the string "string" is a string of length 6 (since it has 6 characters), but it could also be interpreted as a string of length 10 (or really any length) by extending it witn 4 copies of a fictional character with value -1.[2]

The proof from the previous challenge doesn't use any property of ASCII characters besides the fact that they form an ordered set. Therefore, the proof from the previous challenge with all reference to set of ASCII characters replaced with an arbitrary nonempty ordered set suffices. So we can assume that set A consisting of all lists of length 2 whose entries are elements from any ordered set is itself an ordered set.

Here is the proof that ASCII strings of length 4 form an ordered set. By the previous challenge, ASCII strings of length 2 form an ordered set (since twoples of ASCII characters can be thought of as strings of length 2),[3] and so the collection of twoples of ASCII strings of length 2 is an ordered set. Done!

Since ASCII strings of length 4 form an ordered set, twoples of those form an ordered set. Hence ASCII strings of length 8 form an ordered set. Use the previous Challenge once more to see that twoples with first entry of ASCII strings of length 8 and second entry of ASCII strings of length 2 form an ordered set. Therefore, ASCII strings of length 10 form an ordered set.

Challenge 16 When you are about to write a subroutine, you would have an idea of what kind of outputs the subroutine will return, given various inputs. The very first step to writing a subroutine is to gather a number of such input, output pairs, so you can check that the subroutine works on some examples. If you writing a subroutine for a computer, then you should be able to test the subroutine on more complex inputs; the first step to writing a subroutine for a computer is to write a set of *test suites* so that you can thoroughly test the subroutine.

(Step 0) It is time-consuming for us to manually go through the code for `binary_search` on very large arrays, so we will have to make do with some simple arrays: [1], [1|2], [1|2|3], and [1|2|4|5]. Normally, we would also include the empty array [ ]. However, as I have discussed in the solution to Challenge 10 part (c), there is no correct way for a user to call `binary_search` on an empty array (this is also true for `linear_search`). For now, we will assume that the user will not call on either search algorithms on empty arrays.

[2] In this case we'd be extending the extended ASCII encoding system, but we've already been referring ti the encoding system without the word extended, so removing one more is not a big issue.

[3] Technically, there would have to be a mull terminator at the end, but this point is unimportant.

- Searching for 1 in [1]: since both `left` and `right` is 0, `left > right` is false, and we skip to line `5`. Then `mid` is 0, and line `6`, `array[mid] == item` is true, so we return index 0 in line `7`.
- Searching for 1 in [1|2]: since `left` is 0 and `right` is 1, `left > right` is false, and we skip to line `5`. Then `mid` is $\lfloor (0+1)/2 \rfloor = 0$, and line `6`: `array[mid] == item` is true, so we return index 0 in line `7`.
- Searching for 1 in [1|2|3]: since `left` is 0 and `right` is 2, `left > right` is false, and we skip to line `5`. Then `mid` is $\lfloor (0+2)/2 \rfloor = 1$, and line `6`: `array[mid] == item` is false, so we go to line `8`. Since $1 < 2$, we go to line `9`, where we do a recursive call `binary_search(3, 0, array, 0)`. From here, the steps are analogous to searching for 1 in [1].
- Searching for 1 in [1|2|4|5]: since `left` is 0 and `right` is 3, `left > right` is false, and we skip to line `5`. Then `mid` is $\lfloor (0+3)/2 \rfloor = 1$, and line `6`: `array[mid] == item` is false, so we go to line `8`. Since $1 < 2$, we go to line `9`, where we do a recursive call `binary_search(3, 0, array, 0)`. From here, the steps are analogous to searching for 1 in [1].
- Searching for 3 in [1]: since both `left` and `right` is 0, `left > right` is false, and we skip to line `5`. Then `mid` is 0, and line `6`, `array[mid] == item` is false, so we go to line `8`. Since $3 \not< 1$, we go to line `10`, and then do a recursive call in line `11` of `binary_search(3, 1, array, 0)`. Then we go to line `2` for `left` $= 1$ and `right` $=$ 0. Evidently `left > right`, and so we go to line `3`, where we return -1.
- Searching for 3 in [1|2]: since `left` is 0 and `right` is 1, `left > right` is false, and we skip to line `5`. Then `mid` is $\lfloor (0+1)/2 \rfloor = 0$, and line `6`: `array[mid] == item` is false, so we go to line `8`. Since $3 \not< 1$, we go to line `10`, and then do a recursive call in line `11` of `binary_search(3, 1, array, 1)`. From here on the steps are similar to the ones when searching for 3 in [1].
- Searching for 3 in [1|2|3]: since `left` is 0 and `right` is 2, `left > right` is false, and we skip to line `5`. Then `mid` is $\lfloor (0+2)/2 \rfloor = 1$, and line `6`: `array[mid] == item` is false, so we go to line `8`. Since $3 \not< 2$, we go to line `10`, and then do a recursive call in line `11` of `binary_search(3, 2, array, 2)`. We start again from line `2`, which is false. We go to line `5` where `mid` is $\lfloor (2+2)/2 \rfloor = 2$. The condition `array[mid] == item` is true, so we return the index 2.
- Searching for 3 in [1|2|4|5]: since `left` is 0 and `right` is 3, `left > right` is false, and we skip to line `5`. Then `mid` is $\lfloor (0+3)/2 \rfloor = 1$, and line `6`: `array[mid] == item` is false, so we go to line `8`. Since $3 \not< 2$, we go to line `10`, and then do a recursive call in line `11` of `binary_search(3, 2, array, 3)`. From here on the steps are similar to the ones searching for 3 in [1|2], except that the recursive calls are in line `9`, rather than line `8`.

(Step 1) As in `linear_search`, the first thing the search algorithm needs to check is if `left` $\leq$ `right` (done in line `2`). If not, we must return -1 and terminate immediately. Otherwise, we check if the middle element of the array is equal to the item we are looking for. This is done in line `6`. If there is a match, then we return the index (this is done in line `7`). If there is no match, depending on the order of the item in relation to the middle element, we must search the appropriate half of the array. These cases are handled correctly in `binary_search`. Step 1 is complete.

(Step 2a) There are two recursive calls. If `item < array[mid]`, then `left` stays the same, but `right` becomes `(left + right)/2 - 1`. Therefore, the input array we must search in the recursive call is reduced. The same reasoning applies to the case when `item > array[mid]`, and Step 2a is done.

(Step 2b) Although there are two recursive cases, the two cases are so similar that only one needs checking. Suppose that `item < array[mid]` and the recursive call `binary_search(item, left, array, mid - 1);` works correctly. There are two cases, either the recursive cases returned `-1`, or an index. Assume that the recursive call returned `-1`. That means `[array[left]|array[left+1]|...|array[mid-1]]` does not contain `item`. This recursive call occurred because `item < array[mid]` is true. By transitivity, `[array[left]|array[left+1]|...|array[mid-1]|array[mid]|...|array[right]]` does not contain `item`, which is correct.

Assume that the recursive case returned an index. The recursive call is assumed to be correct, so this case is correct by assumption. This completes our analysis of `binary_search`.

Challenge 17 (a) For storing small counting numbers, either would be fine, however an `unsigned int` might be better if we are expecting to store larger numbers. If we expect to store very large numbers, it might call for a `long` data type.

(b) The problem with this program is that it will never end! An unsigned int will always be greater than equal to zero, so the condition "`n >= 0`" is always true, no matter what the value of `n` is. The fix is to change the counter `n` into an `int`, so deleting the word `unsigned` in line `6` will fix this bug. This perhaps explains why we've preferred using `int`'s over `unsigned int`'s in our subroutines.

(c) Earlier in Challenge 8, we saw the importance of the order in which operators are evaluated. The problem with our `binary_search` is that in line `5`, we calculate "`left + right`", then divide by 2. This sum could well be above $2^{31} - 1$ (assuming `int` uses 4 bytes).

(d) An array of length $2^{31} + 1$ will cause problem for sure, because $0 + 2^{31} > 2^{31} - 1$.

(e) Picture (or don't picture) an `array` of length $2^{30} + 1$ where the item is in index 2^{30}, dead last. In order to reach this final index, `binary_search` must calculate "`(left + right)`" where `right` is $2^{30} + 1$ and `left` is either $2^{30} - 1$ or 2^{30}.[4] In either case, $2^{30} - 1 + 2^{30} + 1 > 2^{31} - 1$ and $2^{30} - 1 + 2^{30} + 1 > 2^{31} - 1$, so the bug is triggered. This is not a problem for an array of length 2^{30} or lower.[5]

The bug is triggered when using arrays of length 1,073,741,825 or higher. It was only in the mid 2000s that such array sizes needed to be used in the industry.

Are you getting tired of all the $+1$'s and -1's? I'm afraid this is the price of admission for getting into computing...

(f) Suppose that `array` has length $2^{31}+1$, and that `left` is 0 and `right` is 2^{31}. Then "`(left + right)`" exceeds the limit that `int` can store. On the other hand, "`left + (riht - left)/2`" calculates just fine, since each calculation stays within the limits of what `int` can store.

(g) Just like part (d), an array of length $2^{31} + 1$ will still cause problems.

[4] For the truncated division `/`, both $(2^{30} - 1 + 2^{30} + 1)/2$ and $(2^{30} + 2^{30} + 1)/2$ evaluates to the last index of `array`: 2^{30}. Any other value of `left` will not give `mid` equal to 2^{30}.

[5] Because $2^{30} - 2 + 2^{30} - 1 \leq 2^{31} - 1$, everything fits inside an `int`.

(h) The input to the algorithm `binary_search` includes `int right`, where a user provides the length of `array` minus 1. Using an array of length $2^{31}+1$ violates the precondition that the length of the array fits inside an `int`, and is a misuse of the algorithm. On the other hand, our original `binary_search` fails to provide the correct behavior even when the preconditions are satisfied. An example is the case described in part (e). This is faulty programming.

Challenge 18 (a) No, if `x` has a value of 3, then `x/2 + x/2` gives the value of 2, which is less than 3.

(b) `digitlen(5)` returns 1, `digitlen(100)` returns 3, and `digitlen(199)` also returns a 3. Thus the subroutine `digitlen` calculates the number of digits in a decimal `unsigned int`.

(c) This requires translating the code snippet for breaking a number into two halves, into math. Since $n = 5$, we have $\lfloor n/2 \rfloor = 2$. Thus, we need to find integers a, b such that $25621 = 10^2 a + b$, where $b < 10^2$. Then $a = 256$ and $b = 21$. Similarly, $c = 301$ and $d = 71$.

(d) Since $x = 10^{\lfloor n/2 \rfloor} a + b$ and $y = 10^{\lfloor n/2 \rfloor} c + d$, their product is

$$\begin{aligned} x \cdot y &= (a \cdot c)10^{\lfloor n/2 \rfloor} 10^{\lfloor n/2 \rfloor} + (a \cdot d)10^{\lfloor n/2 \rfloor} + (b \cdot c)10^{\lfloor n/2 \rfloor} + b \cdot d \\ &= (a \cdot c)10^{\lfloor n/2 \rfloor + \lfloor n/2 \rfloor} + (a \cdot d + b \cdot c)10^{\lfloor n/2 \rfloor} + b \cdot d. \end{aligned}$$

The only difference between the algebraic manipulations here and Challenge 3 is that we used an exponentiation rule $a^b a^c = a^{b+c}$.

(e) There are four multiplications: $a \cdot c$, $a \cdot d$, $b \cdot c$, and $b \cdot d$. An identity to remove one of the multiplications can be found by expanding out the product $(a - b) \cdot (c - d)$:

$$(a - b) \cdot (c - d) = a \cdot c - a \cdot d - b \cdot c + (-b) \cdot (-d).$$

Multiply both sides by -1, then add $a \cdot c + b \cdot d$ to both sides to get

$$a \cdot d + b \cdot c = a \cdot c + b \cdot d - (a - b) \cdot (c - d).$$

Plug this into the formula from part (d) to get

$$\begin{aligned} x \cdot y &= (a \cdot c)10^{\lfloor n/2 \rfloor + \lfloor n/2 \rfloor} + (a \cdot d + b \cdot c)10^{\lfloor n/2 \rfloor} + b \cdot d \\ &= (a \cdot c)10^{\lfloor n/2 \rfloor + \lfloor n/2 \rfloor} + (a \cdot c + b \cdot d - (a - b) \cdot (c - d))\, 10^{\lfloor n/2 \rfloor} + b \cdot d. \end{aligned}$$

This formula only involves three multiplications: $a \cdot c$, $b \cdot d$, and $(a - b) \cdot (c - d)$.

(f) Because there are four multiplications in the formula, four recursive calls are required.

```
 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;
12        return rec_prod(a, c) * e1 +
13               (rec_prod(a, d) + rec_prod(b, c)) * e2 + rec_prod(b, d);
14     }
15   }
```

I have broken up lines `12` and `13` into 2 lines because the statement went on for too long. Since there is only one "`;`", the compiler understands that lines `12` and `13` are a single statement. In some programming languages, one needs to be careful when breaking up long statements into multiple lines. This is not the case in C.

(g) The whole point of `fast_prod` is to reduce the number of multiplication required. Notice that the product $a \cdot c$ and $b \cdot d$ are both used twice. Instead of doing each of these

multiplications twice, we will save each calculation as a variable, then reuse the results.

```
 1  int fast_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;
12        int ac = fast_prod(a, c);
13        int bd = fast_prod(b, d);
14        return ac * e1 + (ac + bd - fast_prod(a - b, c - d)) * e2 + bd;
15     }
16  }
```

There are three recursive calls. If we did not have the variables `ac` and `bd` to save $a \cdot c$ and $b \cdot d$, respectively, there would have been five recursive calls.

This is actually a modified version of what Karatsuba proposed. Karatsuba used the identity $a \cdot d + b \cdot c = (a + b) \cdot (c + d) - a \cdot c - b \cdot d$. Thus it would be more faithful to the original algorithm to replace line `14` with `return ac * e1 + (fast_prod(a + b, c + d) - ac - bd) * e2 + bd;`

Challenge 19

(a) $k = 6$; In binary, $a =$ `100010` and $b =$ `100011`. In big-endian order, the bit arrays are `a = [1|0|0|0|1|0]` and `b = [1|0|0|0|1|1]`.

(b) After a logical right shift, `a` becomes `[0|1|0|0|0|1]` and `b` becomes `[0|1|0|0|0|1]`. Both new numbers are `10001` in binary, which is $2^4 + 1$ in decimal.

(c) The new numbers are $\lfloor \frac{a}{2} \rfloor$, and $\lfloor \frac{b}{2} \rfloor$, which happen to be equal.

(d) Applying another logical shift gives `[0|0|1|0|0|0]`, which is `1000` in binary, or 2^3 in decimal. This is equal to $\lfloor \frac{a}{2^2} \rfloor$ and $\lfloor \frac{b}{2^2} \rfloor$.

(e) We can use the statement "`int mid = left + ((right - left) >> 1);`".

(f) An alternative way is to use type casting, where we force the device to change the `int`'s `left` and `right` into `unsigned int`'s:

"`int mid = ( (unsigned int)left + (unsigned int)high ) >> 1;`

By the preconditions, the absolute maximum values of `left` and `right` are $2^{31} - 1$, which is the maximum possible value of an `int` of 4 bytes. Casting each into `unsigned int`'s, we have $2^{31} - 1 + 2^{31} - 1 = 2^{32} - 2$, which is a valid `unsigned int`. Doing a right shift gives $2^{31} - 1$, which is once again, a valid `int` value. Therefore, no overflow occurs.

(g) When stored in 32 bits, the bit arrays are

`a` = `[0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|0|1|0]`,
`b` = `[0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|0|1|1]`.

Applying a left shift results in the bit arrays:

`[0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|0|1|0|0]`,
`[0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|0|1|1|0]`,

respectively. In decimal, these numbers are $2^7 + 2^2$ and $2^7 + 2^2 + 2^1$, which are twice the values of a and b. We see that each right shift on a nonnegative number has the effect of multiplying a number by 2, as long as we do not exceed the limits of what a datatype can hold. If we apply another right shift, the decimal numbers are $2^8 + 2^3$ and $2^8 + 2^3 + 2^2$.