1. (Notation for Lists, $3 \times 2 = 6$ points)

a. A predicate ϕ on lists has this definition:

$$\phi(L) := \forall i \in \{0, \ldots, |L| - 1\}. L[|L| - 1 - i] = L[i].$$

Explain in natural language what $\phi(L)$ being true would tell someone about the list L. Do not just translate the math symbol-by-symbol ("for every integer from zero…"); do your best to give an intuitive explanation of what this predicate means for the values in the list.

Reasonable answers include statements like "the list is a palindrome," "the list is its own reverse," or "reversing the list has no effect."

Explanation: Because $|L|$ is the length of the list, the set $\{0, \ldots, |L| - 1\}$ is the set of zero-based indices into the list, so ϕ requires that $L[|L| - 1 - i] = L[i]$ for every index i. When i is 0, 1, 2, etc., then $|L| - 1 - i$ works out to be the last index $|L| - 1$, the second-to-last index $|L| - 2$, the third-to-last index $|L| - 3$, etc., so the first element must be equal to the last, the second element must be equal to the second-to-last, and so on, all of the way through the entire list.

b. A predicate ψ on lists of numbers has this definition:

$$\psi(L) := |\{|L[(i + 1) \,\%\, |L|] - L[i]| \mid 0 \leq i < |L|\}| = 1.$$

Explain in natural language what $\psi(L)$ being true would tell someone about the list L. Do not just translate the math symbol-by-symbol ("the cardinality of…); do your best to give an intuitive explanation of what this predicate means for the values in the list.

Reasonable answers include statements like "thinking of the list as a cyclic sequence, adjacent elements always differ by the same amount" or "the magnitude of the difference between any adjacent elements is the same as the magnitude of the difference between the first and last elements." Informally, such answers are fine; to be fully precise, we might also want to be explicit about the list being nonempty.

Explanation: Because $|L|$ is the length of the list, the set builder notation $\{\cdots \mid 0 \leq i < |L|\}$ constructs a set element for each index i, but because the set ends up with size one, it must be working on a nonempty list and must be constructing the same element every time. Now the index $(i + 1) \,\%\, |L|$ is ordinarily just $i + 1$, the next index after i, but becomes zero when i is already the last index into the list since $(|L| - 1 + 1) \,\%\, |L| = |L| \,\%\, |L| = 0$. That means that $|L[(i + 1) \,\%\, |L|] - L[i]|$ is the absolute value of the difference between adjacent list elements where we think of the first element as being to the right of the last element, as if the list represents a cycle. So each time we advance one index, possibly wrapping around, the element we find is a fixed amount smaller or larger than the element we just saw.

c. Suppose that $\chi(L) := \phi(L) \wedge \psi(L)$. Describe in natural language what lists satisfying χ look like. Do not just combine your descriptions of ϕ and ψ; do your best to give an intuitive explanation of what this predicate means for the values in the list.

Reasonable answers include statements like "the list is nonempty and homogeneous" or "the list contains a single element repeated one or more times."

Explanation: Because $\phi(L)$ holds, the first and last elements are equal to each other, so they have a difference of zero. Then, by $\psi(L)$ holding, we know that the difference between any adjacent elements in the list is also zero, so every element must be equal to every other element; the list is homogeneous. Finally, $\psi(L)$ also tells us that the list is nonempty.

2. (Notation for Sets and Graphs, 3 × 2 = 6 points)

a. A predicate ϕ on simple undirected graphs has this definition:

$$\phi((V, E)) := \forall e \in E. \exists v \in e. \{u \mid \{u, v\} \in E\} = e \setminus \{v\}.$$

Explain in natural language what $\phi(G)$ being true would tell someone about the graph G. Do not just translate the math symbol-by-symbol ("For every undirected edge e, there exists…"); do your best to give an intuitive explanation of what this predicate means for the shape of the undirected graph.

Reasonable answers include statements like "every edge includes a pendant vertex", "every edge is a pendant edge", or "the graph is a disjoint union of stars."

Explanation: The set $\{u \mid \{u, v\} \in E\}$ would be the set of vertices adjacent to v since those are exactly the vertices that share an edge with v. Meanwhile, the set $e \setminus \{v\}$ is the vertices of e without v, or, since e has exactly two elements including v, a singleton holding whichever vertex the edge connects v to. In other words, $\{u \mid \{u, v\} \in E\} = e \setminus \{v\}$ means that the only thing v is adjacent to is whatever is connected by e; the vertex v is pendant. Then, considering the leading quantifiers, ϕ says that every edge includes a pendant vertex, which is just another way of saying that every edge is pendant. A component with only pendant edges is called a "star," so one could also say that the graph's components are all stars.

b. A predicate ψ on simple undirected graphs has this definition:

$$\psi((V, E)) := |\{u \mid \exists v \in V. \exists w \in V. \{\{u, v\}, \{v, w\}, \{w, u\}\} \subseteq E\}| \leq 2.$$

Explain in natural language what $\psi(G)$ being true would tell someone about the graph G. Do not just translate the math symbol-by-symbol ("For every vertex v…"); do your best to give an intuitive explanation of what this predicate means for the shape of the undirected graph.

Reasonable answers include statements like "the graph contains no cycles of length three" or "the graph is triangle-free."

Explanation: The expression $\{\{u, v\}, \{v, w\}, \{w, u\}\} \subseteq E$ tests that all three of the possible edges on $\{u, v, w\}$ appear in the graph, which is the same as saying that those vertices are connected in a triangle (that they induce a three-cycle). The leading quantifiers, $\exists v \in V. \exists w \in V. \cdots$ tell us that v and w can be chosen freely to make the inner condition true, so the whole set-builder condition asks whether u is part of *any* triangle. Then ψ takes the cardinality of the set of all such vertices, essentially counting how many vertices are triangle corners, and requires that cardinality to be at most two. But a graph cannot have any triangles without including at least three triangle corners, so, to satisfy ψ, it must not have any triangles at all.

c. If a graph satisfies ϕ, does it necessarily satisfy ψ? Likewise, if a graph satisfies ψ, does it necessarily satisfy ϕ? Justify your answers.

Yes, $\forall G. \phi(G) \to \psi(G)$; if every edge is pendant, then every edge ends in a dead end, so there is no way it could be part of a triangle.

On the other hand, we can show that $\psi(G)$ does not imply $\phi(G)$ by giving a counterexample, some graph without triangles that still has a non-pendant edge. Some small counterexamples include P_4, the path on four vertices, where the middle edge is not pendant, or C_4, the cycle on four vertices, where no edge is pendant—neither of these graphs contain triangles.

3. (Summations, $3 \times 2 = 6$ points)

Compute the following sums, showing your work:

a. $\sum_{i=0}^{n-1} \left(2 \sum_{j=i}^{2i-1} j\right)$

$$\begin{aligned}
\sum_{i=0}^{n-1} \left(2 \sum_{j=i}^{2i-1} j\right) &= \sum_{i=0}^{n-1} 2 \left.\frac{j(j-1)}{2}\right|_{j=i}^{j=2i} \\
&= \sum_{i=0}^{n-1} j(j-1)|_{j=i}^{j=2i} \\
&= \sum_{i=0}^{n-1} 2i(2i-1) - i(i-1) \\
&= \sum_{i=0}^{n-1} 3i^2 - i \\
&= 3\left(\sum_{i=0}^{n-1} i^2\right) - \left(\sum_{i=0}^{n-1} i\right) \\
&= 3\left(\left.\frac{i(i-1)(2i-1)}{6}\right|_{i=0}^{i=n}\right) - \left(\left.\frac{i(i-1)}{2}\right|_{i=0}^{i=n}\right) \\
&= 3\left(\frac{n(n-1)(2n-1)}{6} - \frac{0(0-1)(2\cdot 0-1)}{6}\right) - \left(\frac{n(n-1)}{2} - \frac{0(0-1)}{2}\right) \\
&= \frac{n(n-1)(2n-1)}{2} - \frac{n(n-1)}{2} \\
&= n(n-1)^2.
\end{aligned}$$

b. $\sum_{i=1}^{2\sum_{j=0}^{n-1} j} 2^i$

$$\begin{aligned}
\sum_{i=1}^{\sum_{j=0}^{n-1} j} 2^i &= \sum_{i=1}^{2^{\frac{j(j-1)}{2}}|_{j=0}^{j=n}} 2^i \\
&= \sum_{i=1}^{j(j-1)|_{j=0}^{j=n}} 2^i \\
&= \sum_{i=1}^{n(n-1)-0(0-1)} 2^i \\
&= \sum_{i=1}^{n(n-1)} 2^i \\
&= \left.\frac{2^i}{2-1}\right|_{i=1}^{i=n(n-1)+1} \\
&= 2^{n(n-1)+1} - 2.
\end{aligned}$$

c. $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{i-1}(i+j)$

$$\begin{aligned}
\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{i-1}(i+j) &= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(i+j)\sum_{k=0}^{i-1}1 \\
&= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(i+j)\ k\Big|_{k=0}^{k=i} \\
&= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(i+j)(i-0) \\
&= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} i^2+ij \\
&= \sum_{i=0}^{n-1}\left(i^2\left(\sum_{j=0}^{n-1}1\right)+i\left(\sum_{j=0}^{n-1}j\right)\right) \\
&= \sum_{i=0}^{n-1}\left(i^2\left(j\Big|_{j=0}^{j=n}\right)+i\left(\frac{j(j-1)}{2}\Big|_{j=0}^{j=n}\right)\right) \\
&= \sum_{i=0}^{n-1}\left(i^2(n-0)+i\left(\frac{n(n-1)}{2}-\frac{0(0-1)}{2}\right)\right) \\
&= \sum_{i=0}^{n-1}\left(ni^2+\frac{n(n-1)}{2}i\right) \\
&= n\left(\sum_{i=0}^{n-1}i^2\right)+\frac{n(n-1)}{2}\left(\sum_{i=0}^{n-1}i\right) \\
&= n\left(\frac{i(i-1)(2i-1)}{6}\Big|_{i=0}^{i=n}\right)+\frac{n(n-1)}{2}\left(\frac{i(i-1)}{2}\Big|_{i=0}^{i=n}\right) \\
&= n\left(\frac{n(n-1)(2n-1)}{6}-\frac{0(0-1)(2(0)-1)}{6}\right)+\frac{n(n-1)}{2}\left(\frac{n(n-1)}{2}-\frac{0(0-1)}{2}\right) \\
&= n\cdot\frac{n(n-1)(2n-1)}{6}+\frac{n(n-1)}{2}\cdot\frac{n(n-1)}{2} \\
&= \frac{n^2(n-1)}{12}\left(2(2n-1)+3(n-1)\right) \\
&= \frac{n^2(n-1)(7n-5)}{12}.
\end{aligned}$$

4. (Recurrences, $2 \times 3 = 6$ points)

Solve the following recurrences for all nonnegative integer values of n either by guess-and-check or by substitution, showing your work:

a.

$$T(n) = \begin{cases} 10T(n-1) + 10^n & \text{if } n > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Preparing a table of $T(n)$ for various small values of n:

n	$T(n)$
0	0
1	10
2	200
3	3000
4	40000
5	500000
6	6000000
⋮	⋮

suggests the formula

$$T(n) = n10^n.$$

To check this guess, we rewrite the recurrence using another variable name:

$$T(x) = \begin{cases} 10T(x-1) + 10^x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Taking $n = x$, the guess becomes $T(x) = x10^x$, which we can use to replace the left-hand side of the recurrence:

$$x10^x = \begin{cases} 10T(x-1) + 10^x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Taking $n = x - 1$, the guess becomes $T(x-1) = (x-1)10^{x-1}$, which we can use to replace the recursive part of the right-hand side of the recurrence:

$$x10^x = \begin{cases} 10(x-1)10^{x-1} + 10^x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Simplifying, we get:

$$x10^x = \begin{cases} (x-1)10^x + 10^x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

and then:

$$x10^x = \begin{cases} x10^x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Because the equality holds for any integral input size $x \geq 0$, the guess is a correct closed form of the recurrence over nonnegative integers.

b.

$$T(n) = \begin{cases} T(n-1) + 2T(n-2) & \text{if } n > 1 \\ n+1 & \text{otherwise.} \end{cases}$$

Preparing a table of $T(n)$ for various small values of n:

n	$T(n)$
0	1
1	2
2	4
3	8
4	16
5	32
$\vdots$	$\vdots$

suggests the formula

$$T(n) = 2^n.$$

To check this guess, we rewrite the recurrence using another variable name:

$$T(x) = \begin{cases} T(x-1) + 2T(x-2) & \text{if } x > 1 \\ x+1 & \text{otherwise.} \end{cases}$$

Taking $n = x$, the guess becomes $T(x) = 2^x$, which we can use to replace the left-hand side of the recurrence:

$$2^x = \begin{cases} T(x-1) + 2T(x-2) & \text{if } x > 1 \\ x+1 & \text{otherwise.} \end{cases}$$

Taking $n = x - 1$, the guess becomes $T(x-1) = 2^{x-1}$, which we can use to replace the first recursive part of the right-hand side of the recurrence:

$$2^x = \begin{cases} 2^{x-1} + 2T(x-2) & \text{if } x > 1 \\ x+1 & \text{otherwise.} \end{cases}$$

Taking $n = x - 2$, the guess becomes $T(x-2) = 2^{x-2}$, which we can use to replace the other recursive part of the right-hand side of the recurrence:

$$2^x = \begin{cases} 2^{x-1} + 2 \cdot 2^{x-2} & \text{if } x > 1 \\ x+1 & \text{otherwise.} \end{cases}$$

Simplifying, we get:

$$2^x = \begin{cases} 2^{x-1} + 2^{x-1} & \text{if } x > 1 \\ x+1 & \text{otherwise.} \end{cases}$$

and then:

$$2^x = \begin{cases} 2^x & \text{if } x > 1 \\ x+1 & \text{otherwise.} \end{cases}$$

Because the equality holds for any integral input size $x \geq 0$, the guess is a correct closed form of the recurrence over nonnegative integers.

5. (Asymptotics, $3 \times 2 = 6$ points)

Use the Limit Method to solve the following problems:

a. Consider the expressions 2^n and 2^{2n}. Indicate whether the first expression has (within a constant multiple) a lower, same, or higher order of growth than the second. Show your work.

Merely by setting up the limit and simplifying, we get

$$\lim_{n\to\infty} \frac{2^n}{2^{2n}} = \lim_{n\to\infty} \frac{2^n}{2^n \cdot 2^n} = \lim_{n\to\infty} \frac{1}{2^n} = 0,$$

Thus, by the limit method, the first expression has a lower order of growth than the second expression.

b. Consider the expressions $n^{2.1} + 10^6$ and $n^2 \ln n$. Indicate whether the first expression has (within a constant multiple) a lower, same, or higher order of growth than the second. Show your work.

With three invocations of L'Hôpital's rule and then simplification:

$$\lim_{n\to\infty} \frac{n^{2.1} + 10^6}{n^2 \ln n} = \lim_{n\to\infty} \frac{2.1n^{1.1}}{n + 2n \ln n} = \lim_{n\to\infty} \frac{2.31n^{0.1}}{3 + 2\ln n} = \lim_{n\to\infty} \frac{0.231n^{-0.9}}{2n^{-1}} = \lim_{n\to\infty} 0.1155n^{0.1} = \infty.$$

Thus, by the limit method, the first expression has a higher order of growth than the second expression.

c. It can be tempting to compare two expressions asymptotics by plotting them, but plots can be misleading. Explain how a plot of the two expressions from Part b could mislead a programmer. Also explain how to vary the bounds on the plot so that it gives the full picture.

The plot shows the first expression substantially ahead, but staying mostly flat, whereas the second expression curls up more as n approaches the given upper bound, 100. As such, one might expect that the second expression will win out in the end (an idea that is worth half points), and extending the plot out further does give what looks like a classic case of one expression overtaking the other. Even looking out to an n of one quadrillion paints the same kind of picture. However, our work in Part b tells us that the first expression should win out in the end, and if we have the perseverance to look even further, to around $n = 10^{16}$, we indeed see it retake the lead.

6. (Iterative Algorithm Analysis, 6 points)

Define the input size n for the following pseudocode and describe how to construct a worst-case input given a value for n. Based on your construction, demonstrate that, for an appropriate choice of basic operations, the algorithm runs in worst-case time $2n^2 + n + 1$. Show your work.

```
   Input: $A$, a list of positive 64-bit integers
   Input: $b$, a positive 64-bit integer
 1 let $r \leftarrow 0$
 2 for $i \in [0, \ldots, |A| - 1]$ do
 3     for $j \in [0, \ldots, |A| - 1]$ do
 4         if $A[i] \leq A[j] + b$ then
 5             let $r \leftarrow r + A[i]$
 6         end
 7     end
 8     for $j \in [i, \ldots, |A| - 1]$ do
 9         if $A[i] + A[j] \leq b$ then
10             let $r \leftarrow r - A[i]$
11             let $r \leftarrow r - A[j]$
12         end
13     end
14 end
15 return $r$
```

Starting with a definition of input size, because the bit-width is fixed everywhere, the only possible way the amount of input can vary is in the list length; we define $n = |A|$.

To form a recipe for a worst-case input, we take n as given and try to choose other aspects of the input to make as much code run as possible. In this case, we are free to vary the elements of A and the value of b. Since only the conditionals care about those values, we focus on a way to make both `if`s follow their "then" branches as often as possible. Fortunately, we can make those branches always taken by doing something like filling A with 1s and setting b to 2, so that would be one construction. But any scheme for making the elements of A small enough compared to b would also be fine.

As for choosing basic operations, it might take some trial and error to hit $2n^2 + n + 1$ exactly, but we can start with an informed guess. To ensure that every loop iteration counts as more than zero steps, it suffices to count at least a basic operation from each of the innermost loops' bodies—if the outer loop was able to run at all, then A must have been nonempty, so the inner loops will also run. But the outer loop is not guaranteed to take iterations on an empty list, and to ensure that every function call counts at least one basic operations, we also have to count something outside of the loops.

Of the choices available, assignments to r are tempting, since they cover all of the necessary program locations—we have shown that we can make all the conditionals taken, and we can already guarantee that these assignments run every loop iteration. We start by trying them.

Working from the inside out, lines 4–6 take one step, so lines 3–7 take $\sum_{j=0}^{n-1} 1$ steps. Similarly, lines 9–12 take two steps, so lines 8–13 take $\sum_{j=i}^{n-1} 2$ steps. Thus, lines 2–14 require $\sum_{i=0}^{n-1} \left(\left(\sum_{j=0}^{n-1} 1 \right) + \left(\sum_{j=i}^{n-1} 2 \right) \right)$ steps, and we add one more step when we bring lines 1 and 15 into the picture.

Now we can try evaluating that sum to see if it gives the desired total or something close:

$$\begin{aligned}
1+\sum_{i=0}^{n-1}\left(\left(\sum_{j=0}^{n-1}1\right)+\left(\sum_{j=i}^{n-1}2\right)\right) &= 1+\sum_{i=0}^{n-1}\left(\left(\sum_{j=0}^{n-1}1\right)+2\left(\sum_{j=i}^{n-1}1\right)\right)\\
&= 1+\sum_{i=0}^{n-1}\left(\left(j\big|_{j=0}^{j=n}\right)+2\left(j\big|_{j=i}^{j=n}\right)\right)\\
&= 1+\sum_{i=0}^{n-1}(n-0+2n-2i)\\
&= 1+3n\left(\sum_{i=0}^{n-1}1\right)-2\sum_{i=0}^{n-1}i\\
&= 1+3n\left(i\big|_{i=0}^{i=n}\right)-2\left(\left.\frac{i(i-1)}{2}\right|_{i=0}^{i=n}\right)\\
&= 1+3n(n-0)-2\left(\frac{n(n-1)}{2}-\frac{0(0-1)}{2}\right)\\
&= 1+3n^2-2\frac{n(n-1)}{2}\\
&= 2n^2+n+1.
\end{aligned}$$

It does match, so we are done. If it had been off, it still could have informed changes to our counting to match what the problem is imagining as a step: basic operations from the inner loop would contribute quadratic terms; basic operations from the outer loop would contribute linear terms, and basic operations outside the loop would contribute constant terms, so we could find the highest-order discrepancy and adjust accordingly.

7. (Recursive Algorithm Analysis and Master Theorem, 6 points)

Use the Master Theorem to demonstrate that, for an appropriate choice of basic operations and input size, the following pseudocode runs in roughly $\Theta(n^{1.585})$ worst-case time. Show your work.

```
 1 function bravo($A, b = |A|, c = |A|$)
       Input: $A$, a nonempty list of numbers
       Input: $b$, an element count
       Input: $c$, an element count
 2     if $c = 0$ then
 3         return $b$
 4     end
 5     let $d \leftarrow 1$
 6     for $i \in [1, 2, 3]$ do
 7         let $d \leftarrow d + \texttt{bravo}(A, \lfloor b/i \rfloor, \lfloor c/2 \rfloor)$
 8     end
 9     for $i \in [0, \ldots, c-1]$ do
10         if $A[i] = d$ then
11             return $b$
12         end
13     end
14     return 0
15 end
```

First, we can measure input size by looking for a quantity that decreases with each recursive call. A never changes, so it is out. b decreases sometimes, but not when i is one, so by itself it cannot be an input size. c, though, does get reduced by about half every recursion. So take $n = c$.

Then, because this problem involves an asymptotic analysis, our choice of basic operations should not matter as long as they occur at least once per iteration and at least once per function call on large inputs. Here we will use equality comparisons as one occurs per function call on line 2, at least one occurs per iteration of the first loop because of the recursive call on line 7, and one occurs per iteration of the second loop because of line 10.

Working from the inside out, line 7 is a recursive call on an input of size $\lfloor n/2 \rfloor$, so it takes $T(\lfloor n/2 \rfloor)$ steps. Lines 6–8 therefore require $\sum_{i=1}^{3} T(\lfloor n/2 \rfloor)$ steps. Looking at the other loop, line 10 requires 1 step, and since we are doing a worst-case analysis, we can suppose that line 11 never returns early, which means that lines 9–13 need $\sum_{i=0}^{n-1} 1$ steps. Along with line 2, that gives us a total of $1 + \left(\sum_{i=1}^{3} T(\lfloor n/2 \rfloor)\right) + \sum_{i=0}^{n-1} 1$ steps normally. However, even a worst-case input can't prevent the `return` on line 3 when $n = 0$, so we treat that as a separate case:

$$T(n) = \begin{cases} 3T(\lfloor n/2 \rfloor) + n + 1 & \text{if } n > 0 \\ 1 & \text{otherwise.} \end{cases}$$

The quantity $n/2$ goes to infinity as n increases, so using the shortcut from class, we can ignore the floors and ceilings for the purposes of applying the Master Theorem to the large-input case:

$$\begin{aligned} T(n) &= \underbrace{3T(\lfloor n/2 \rfloor)}_{\text{about } \Theta(n^{\log_2 3})} + \Theta(n) \\ &= \Theta(n^{\log_2 3}) \\ &\approx \Theta(n^{1.585}). \end{aligned}$$

8. (Practice with JavaScript Arrays, 1 point for the code linting, 5 points for functional correctness)

Consider a problem where an array contains zero-delimited blocks of positive integers, and the goal is compute a new array where each block is flipped. For example, if the array is $[0, 1, 2, 3, 0, 0, 4, 5]$, then flipping the block $[1, 2, 3]$ around would give $[3, 2, 1]$, and flipping the block $[4, 5]$ around would give $[5, 4]$, so the full result after flipping every zero-delimited block would be $[0, 3, 2, 1, 0, 0, 5, 4]$.

Complete the JavaScript function `flipZeroDelimitedBlocks` in `flip.js` in your fork of the homework repository so that it takes one parameter, `list`, and it returns a new array that is `list` after flipping every zero-delimited block. For example, the result of `flipZeroDelimitedBlocks([0, 1, 2, 3, 0, 0, 4, 5])` should be the array `[0, 3, 2, 1, 0, 0, 5, 4]`.

While it is possible to implement this operation using just loops and conditionals, you can save yourself time and simplify parts of the algorithm considerably by taking advantage of some of JavaScript's built-in `Array` methods, possibly in combination with spread syntax.

Unit tests to check functional correctness are provided in `flip.test.js`, and you can run them using the `test-once:flip:correctness` NPM script.

Finally, use the project's built-in linter to check your code style. Any `eslint-disable` directives will be ignored by the autograder, so do not use them to hide warnings in your local copy, or you may miss code style problems that will cost you points.

A straightforward strategy is to process the input from left to right. Whenever we find a zero or the end of the list, we know that we have found the end of a block (though it might be an empty block), so we can reverse that block and append it to a result list. If the block was terminated by a zero, we also have to push the zero onto the result list. In code:

```
export function flipZeroDelimitedBlocks(list) {
  const results = [];
  let blockStartIndex = 0;
  for (let i = 0; i < list.length; ++i) {
    if (list[i] === 0) {
      results.push(...list.slice(blockStartIndex, i).reverse(), 0);
      blockStartIndex = i + 1;
    }
  }
  results.push(...list.slice(blockStartIndex).reverse());
  return results;
}
```

Here, `blockStartIndex` is a loop variable to keep track of where the block we are currently processing began. The other loop variable, `i`, tracks where we are in the input using a standard three-part `for` loop. When we hit a zero element, we slice the input list to get all of the elements from the block's beginning up through but not including the zero we just found, reverse that slice, and spread its elements out as arguments to the `push` method. The zero we found is also given as an additional argument so that it will be pushed on the end. Then we reset `blockStartIndex` knowing that the next block will begin after the just-found zero.

At the end of the loop, we also have the end of block, but this time one not terminated by a zero, so we write the same `push` but omit the final argument. We can then return the results.

9. (Practice with JavaScript Sets and Maps, 1 point for the code linting, 5 points for functional correctness)

A **functional graph** is a directed graph where every vertex has exactly one outgoing edge. Functional graphs can be represented as dictionaries where each key-value pair gives a vertex and the destination of its outgoing edge.

Below is pseudocode for an algorithm that takes a functional graph encoded as a map and returns, for every vertex, the sources that can reach that vertex:

Input: M, a map of vertices to vertices
Output: R, a map of vertices to sets of sources that reach them

```
let $R \leftarrow \varnothing$
for $i \in \{k \mid (k, v) \in M\}$ do
    let $R[i] \leftarrow \varnothing$
end
for $s \in \{k \mid (k, v) \in M\} \setminus \{v \mid (k, v) \in M\}$ do
    let $O \leftarrow \varnothing$
    let $i \leftarrow s$
    while $i \notin O$ do
        let $R[i] \leftarrow R[i] \cup \{s\}$
        let $O \leftarrow O \cup \{i\}$
        let $i \leftarrow M[i]$
    end
end
return $R$
```

Implement this algorithm as the JavaScript function `catalogReachingSources` in `sources.js` in your fork of the homework repository. Do not use any method named `forEach` in your implementation; such methods are only provided for compatibility with JavaScript written prior to 2015, whereas your code should use modern style as described in your readings.

Unit tests to check functional correctness are provided in `sources.test.js`, and you can run them using the `test-once:sources:correctness` NPM script.

Use the project's built-in linter to check your code style. Any `eslint-disable` directives will be ignored by the autograder, so do not use them to hide warnings in your local copy, or you may miss code style problems that will cost you points.

Most of this pseudocode will translate directly to JavaScript, but one big issue is that JavaScript does not have a built-in set difference operator. Therefore, before beginning the translation, we write some code to precompute $\{k \mid (k, v) \in M\} \setminus \{v \mid (k, v) \in M\}$:

```
const sources = new Set(functionalGraph.keys());
for (const value of functionalGraph.values()) {
  sources.delete(value);
}
```

Here $\{k \mid (k, v) \in M\}$ is just `functionalGraph.keys()` converted to a set, which is taken care of on the first line. But then we have to remove all of the dictionary values, which we do by looping over them and calling `delete`.

As for the main translation, lines 1–4 match pretty closely once we know that $\{k \mid (k, v) \in M\}$ can be read as `functionalGraph.keys()` and that JavaScript `Map`s use the `set` method to associate keys to values:

```
const results = new Map();
for (const key of functionalGraph.keys()) {
  results.set(key, new Set());
}
```

And the outer loop is similarly straightforward given the precomputation above:

```
for (const source of sources) {
  const seen = new Set();
  let current = source;
  // ...
}
```

For the inner loop, we have to know that set membership is tested with the `has` method and that elements are added to sets with the `add` method:

```
while (!seen.has(current)) {
  results.get(current).add(source);
  seen.add(current);
  current = functionalGraph.get(current);
}
```

Finally, we can return the results, altogether giving this code:

```
export function catalogReachingSources(functionalGraph) {
  const sources = new Set(functionalGraph.keys());
  for (const value of functionalGraph.values()) {
    sources.delete(value);
  }
  const results = new Map();
  for (const key of functionalGraph.keys()) {
    results.set(key, new Set());
  }
  for (const source of sources) {
    const seen = new Set();
    let current = source;
    while (!seen.has(current)) {
      results.get(current).add(source);
      seen.add(current);
      current = functionalGraph.get(current);
    }
  }
  return results;
}
```

10. (Practice with JavaScript Classes, 1 point for the code linting, 5 points for functional correctness)

A robot, initially facing east at the origin of an infinite two-dimensional grid, has three actions that it can take: it can move forward one unit, it can rotate in place 90° to the left, or it can rotate in place 90° to the right.

Planning and navigation algorithms that the robot might use (and that we will cover later in the course) operate on a graph where the graph's vertices are situations the robot can find itself in, and edges are actions that lead from one situation to the next. In this case the graph is infinite, so it must be encoded by intension, not stored in memory by extension.

One common solution is to write a `Vertex` class and provide getters for each possible action to compute the reached vertex on the fly. For instance, `new Vertex().forward.left.forward.right.forward.forward` would:

- Create the starting vertex as the robot facing east at $(0, 0)$,
- Run the `forward` getter to create another vertex where the robot is facing east at $(1, 0)$,
- Run the `left` getter to create another vertex where the robot is facing north at $(1, 0)$,
- Run the `forward` getter to create another vertex where the robot is facing north at $(1, 1)$,
- Run the `right` getter to create another vertex where the robot is facing east at $(1, 1)$, and
- Run the `forward` getter to create another vertex where the robot is facing east at $(2, 1)$.
- Run the `forward` getter to create another vertex where the robot is facing east at $(3, 1)$,

Complete the JavaScript class `Vertex` in `facing.js` in your fork of the homework repository so that it has a constructor that creates a vertex where the robot facing east at $(0, 0)$ when given no arguments, it has a `forward` getter that returns a new vertex where the robot has moved forward one unit, it has `left` and `right` getters that return new vertices with the robot rotated 90° in the appropriate direction, and it has fields or getters called `x` and `y` to access the robot's coordinates. For example, if code were to set `const vertex = new Vertex().forward.left.forward.right.forward.forward;`, then `vertex.x` would be `3`, and `vertex.y` would be `1`.

Use the project's built-in linter to check your code style. Any `eslint-disable` directives will be ignored by the autograder, so do not use them to hide warnings in your local copy, or you may miss code style problems that will cost you points.

Unit tests in `facing.test.js` are provided to check functional correctness.

The `Vertex` class will need to track two pieces of information: the robot's location (which we can store as an (x, y)-pair) and the direction in which the robot will move if told to move forward (which we can store as an $(\Delta x, \Delta y)$-pair). So we can start by creating a constructor that takes those four values and stores them in instance fields:

```
constructor(x, y, dx, dy) {
  this.x = x;
  this.y = y;
  this.dx = dx;
  this.dy = dy;
}
```

But the contract described in the problem and the test cases both expect to be able to construct a `Vertex` without any arguments and get the robot at the origin facing east, so we need to provide default arguments:

```
constructor(x = 0, y = 0, dx = 1, dy = 0) {
  this.x = x;
  this.y = y;
  this.dx = dx;
  this.dy = dy;
}
```

Then, to implement `.forward`, we define a getter that adds the direction of movement to the robot's current position and creates a new vertex with those updated values. Notice how, unlike in Java, we always use `this.` to refer to instance fields:

```
get forward() {
  return new Vertex(
    this.x + this.dx,
    this.y + this.dy,
    this.dx,
    this.dy,
  );
}
```

The getters for the turn actions are similar, except that we leave the location unchanged and instead transform the directional values:

```
get left() {
  return new Vertex(this.x, this.y, -this.dy, this.dx);
}

get right() {
  return new Vertex(this.x, this.y, this.dy, -this.dx);
}
```