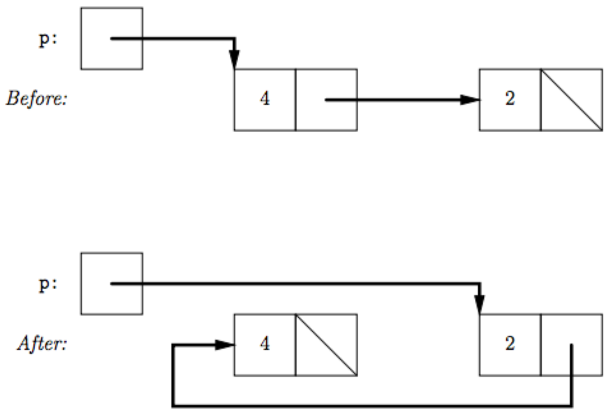
CSE 373 QuickCheck 1 Name: Evil Kevin Student ID: 1234567890

1. The picture shows two states of a `ListNode p`. In the lines below, write Java code to turn *Before* to *After*. **Do not** modify the `data` instance variable, introduce any new variables, or use the `new` keyword.



`p.next.next = p;`

`p = p.next;`

`p.next.next = null;`

Since we can't modify the `data` field, introduce new variables, or use the `new` keyword, we need to modify the pointers in the diagram. When comparing *Before* and *After*, 3 differences stand out.

- `p` points to the node-with-value-2.
- The `next` field of the node-with-value-4 is null.
- The `next` field of the node-with-value-2 points to the node-with-value-4.

We can implement these 3 differences with one assignment statement each.

One of the tricky parts of the problem is that we need to give the correct sequence of operations. If we move `p` too early, we won't have access to the node with value 4! As a result, we need to execute these assignment statements in a certain order.

1. Reassign the `next` field of the node-with-value-2 to the node-with-value-4: `p.next.next = p`
2. Reassign `p` to the node-with-value-2: `p = p.next`

If we reassign the `next` field of the node-with-value-4 to null too early, then we won't have any reference to the node-with-value-2. As a result, we know this operation needs to come after the first two, rather than before or in-between.

3. Reassign the `next` field of the node-with-value-4 to null after reassigning `p` in step 2. We can access the node-with-value-4 with `p.next`, so we can reassign its `next` field: `p.next.next = null`.

2. Give a tight asymptotic runtime bound for `mystery` as a function of N, the length of the array, in the **best case**, **worst case**, and **overall**. Give a $\Theta(\cdot)$ bound if it exists. Otherwise, give both an $O(\cdot)$ and $\Omega(\cdot)$ bound.

```
static boolean mystery(int[] a, int target) {
    int N = a.length;
    for (int i = 0; i < N; i += 1) {
        if (a[i] == target) {
            return true;
        }
    }
    return false;
}
```

Best case $\Theta(1)$

Worst case $\Theta(N)$

Overall $\Omega(1), O(N)$

Similar to our analysis of `dup1` and `dup2`, the fact that `mystery` takes an array as input should clue us into case analysis. Let's break this down following the 3-stage process we saw in lecture: (1) Comprehending the behavior of the program, (2) Modeling the runtime as a function of the input size for the best and worst case, and (3) Formalizing the result with asymptotic notation.

Comprehending. The program `mystery` scans the entire array until it finds an integer equal to the target. The method returns true if an item is found, and false otherwise. The runtime of `mystery` is entirely determined by the content of the array and the target value.

Let's select the number of array accesses as our cost model. The number of equality == operations, `i += 1` increments, or less-than < operations are also good cost models.

Modeling and Formalizing: Best Case. In the best case, `a[0]` is the same as the target value. We can say, then, that the runtime $R_{\text{best}}(N) \approx 1$. This function is in $\Omega(1)$ and also in $O(1)$. Because the lower-bounding and upper-bounding orders of growth coincide, we can say that it is also in $\Theta(1)$.

In general, whenever we restrict our analysis to a single case alone, we can use $\Theta(\cdot)$ notation since there's no way for the runtime to vary once we've locked in every single assumption.

Modeling and Formalizing: Worst Case. In the worst case, the array does not contain the target value at all. We can say, then, that the runtime $R_{\text{worst}}(N) \approx N$. This function is in $\Omega(N)$ and also in $O(N)$. Because the lower-bounding and upper-bounding orders of growth coincide, we can say that it is also in $\Theta(1)$.

Formalizing: Overall. The overall runtime captures all possible scenarios between the best and worst case runtimes. The best case is the lowest possible runtime while the worst case is the highest possible runtime, so the range can be covered by $\Omega(1)$ and $O(N)$.