1. To solve this problem, you have to move to the left sometimes and to the right sometimes, and you need to increase the distance traveled after each change of direction. There are two basic ideas: increase the distance by a constant amount each time you change direction, or multiply the distance by some constant amount each time. To make the analysis easier, here are basic versions of the two algorithms:

Algorithm 1. Phase 1: Move the right one step, and back, then move to the left one step and back. Phase 2: Move to the right two steps and back, then move to the left two steps and back. Phase 3: Move to the right three steps and back, then move to the left three steps and back. Phase 4: Move to the right four steps and back, then move to the left four steps and back. And so on. (That is, increase the distance traveled by 1 step each time you complete a phase.) Stop as soon as you find the treasure.

Analysis: If the treasure is N steps away from the starting point, you will do Phases 1 through $N-1$ without finding the treasure. That's a total of $4*(1+2+\cdots(N-1))$ steps (since phase K takes $4*K$ steps.) This sum is equal to $4*(N-1)*N/2$, or $2N^2-2N$. If the treasure is to the right of the starting point, you find it after an additional N steps; if it is to the left, you find it after an additional $3*N$ steps. So, the number of steps to find the treasure is either $2N^2-N$ or $2N^2+N$. These are exact values. In any case, the number of steps is $\Theta(N^2)$.

Algorithm 2. Phase 0: Move the right one step, and back, then move to the left one step and back. Phase 1: Move to the right 2 steps and back, then move to the left 2 steps and back. Phase 2: Move to the right 2^2 steps and back, then move to the left 2^2 steps and back. Phase 3: Move to the right 2^3 steps and back, then move to the left 2^3 steps and back. And so on. (That is, *double* the distance traveled each time you complete a phase.) Stop as soon as you find the treasure.

Analysis: If the treasure is more than 2^K steps away from the starting point and less than or equal to 2^{K+1} steps away, you will do Phases 0 through K without finding the treasure. That's a total of $4*(2^0+2^1+2^2\cdots 2^K)$ steps. This sum is, in fact, $4*(2^{K+1}-1)$. (Add up the powers of 2 in binary if you don't believe that.) You will then find the treasure in the next phase, adding somewhere between between 2^K+1 and $4*2^{K+1}$. In any case, we see that the run time is $\Theta(2^K)$, and since N is between 2^K and 2^{K+1}, N is essentially a constant multiple of 2^K, and $\Theta(2^K)$ is the same as $\Theta(N)$. So, Algorithm 2 is much more efficient than Algorithm 1.

2. I can think of two algorithms that might work.... The first is a simple "iterative" algorithm. Assume that the vertices are numbered 1, 2, ..., N. The algorithm is:

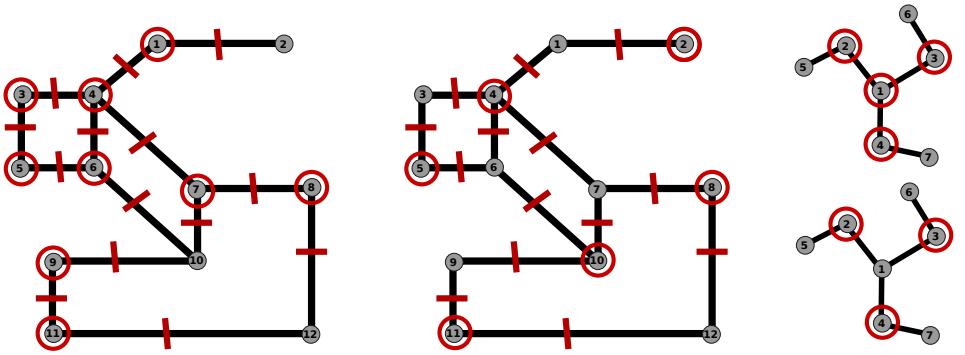
```
for i from 1 to N:
    if there is a vertex connected to vertex i that is not yet covered:
        post a guard at vertex i
```

This might seem reasonable because it only posts a guard at a vertex if that guard is needed. The result of this algorithm applied to the sample graph is shown on the left below. As I ran the algorithm on this graph, I crossed out covered edges as the guards were posted. The vertices where guards are posted are circled. In fact, this algorithm doesn't work very well. It uses nine guards. (We might try to improve the algorithm by making a second pass and deleting unnecessary guards that can be deleted without leaving any corridor uncovered. In this case, that would only remove the guard at intersection number 3.)

The second algorithm is a "greedy" algorithm. The idea is to always post a guard at a vertex that connects to a maximal number of uncovered corridors. Repeat that until all the corridors

are covered. The result of applying this algorithm to the sample graph is shown in the middle image below. The first guard is posted at vertex 4, since that will cover four corridors, and there are no vertices that cover more than four. The next guard is placed at vertex 10, which covers three vertices that are not yet covered. Then there are three vertices that cover two new edges, vertices number 5, 8, and 11. Guards can be placed on these vertices in any order (or numerical order, to make it more definite). Finally, we need a guard at vertex 2 to cover the one remaining uncovered corridor. Note that the greedy algorithm only uses six guards.

Neither of these algorithms finds an optimal solution in all cases. A counterexample that works for both algorithms is shown on the right below. Both algorithms start by posting a guard at vertex 1, but they must then post three additional guards to cover all the corridors. However, as the picture at bottom right shows, it is possible to cover all the corridors with just three guards.



Iterative Algorithm Greedy Algorithm Counterexample

3. Algorithm:

```
int i = 0;
while (i < A.length && A[i] != v) {
      i = i + 1;
}
if ( i == A.length )
      return -1;
else
      return i
```

A loop invariant for the *while* loop is the statement, "For all $j < i$, $A[j]$ is not equal to v." Or, in plain English, "v is not among the first i elements of A."

Initialization: This is certainly true before the loop starts, when i is zero, since v certainly can't be among the first zero elements of A. (There are no such elements to consider.)

Maintenance: Suppose that the loop invariant is true at the start of the loop. If i is $A.length$, or if $A[i]$ is v, then the loop exits without changing anything, so the loop invariant is still true (since nothing has changed!). If $A[i]$ is not v, then the loop body executes. That is, i goes up by one, and the loop invariant is now a statement about one additional element of A. But since we tested $A[i] \neq v$, we know that that additional element is not v. So, even with the new, larger value of i, it's still true that v is not among the first i elements of A.

Termination: So, the loop invariant is still true when the loop ends. There are two reasons for the loop to end: Either i is $A.length$ or $A[i]$ is v. In the case $i = A.length$, the loop invariant says v is not equal to any element of A, and the algorithm can correctly return -1. In the case where $A[i] = v$, it can correctly return i. (Note that in the second case, the loop invariant implies that i is actually the *first* index for which $A[i] = v$.)