1. Give the asymptotic complexity for the worst case run-time of the following function `foo()` with respect to its input N. Function `bar()` runs in constant time with respect to N. For full credit you should show how you arrived at your answer. An answer that solely provides a value in Big O notation without reason will receive zero points. Answers that include variables other than N will receive fewer points.

```
1  void foo(unsigned int N) {
2      unsigned int S = 0;
3
4      for (unsigned int i = 1; i <= N; i++) {
5          S += i;
6      }
7
8      for (unsigned int j = 1; j <= S; j++) {
9          bar();
10     }
11 }
```

the first for loop has $O(N)$ running time

S become $\frac{N(1+N)}{2}$

Then the second for loop has $O(S) = O(\frac{N}{2} + \frac{N^2}{2}) = O(N^2)$ running time

the total running time $O(N + N^2) = O(N^2)$

2. Suppose we have a counter that stores an arbitrary number of bits and counts in binary. It always begin at 0. The only mutator operation it can perform is to increment, adding one to the current count. This changes one or more bits. **Show** that if we start at 0 and perform k increment operations, a total of $\mathcal{O}(k)$ bits will change. Correct answers will reason about the **asymptotic** behavior of the total number of bit changes.

   Hint: There are multiple ways to approach this problem. One approach could use a credit argument, and you should refer to the array expansion of array-based Stacks and Queues for inspiration here. Another approach is numerical analysis. Try to find patterns in the total number of bit changes as k increases and inductively reason about the overall asymptotic behavior.

L-

bit 0 change every increment

bit 1 change every 2^{nd} increment

bit 2 change every 4^{th} increment

bit n change very 2^{nth} increment

let $n \in \mathbb{Z}$ s.t. $2^n < k$ and 2^n is the closet to k

running time: $k + \frac{k}{2} + \frac{k}{4} + \cdots \frac{k}{2^n}$

$< k/(1-1/2) = 2k = O(k)$

3. An undirected graph is called d-regular if it is a simple graph and, for every vertex v, $\delta(v) = d$. For example, in a 3-regular graph, every vertex has three neighbors.

   (a) Either draw a 3-regular graph that has exactly nine (9) vertices, or explain why it cannot be done.

   (b) Either draw a 3-regular graph that has exactly twelve (12) vertices, or explain why it cannot be done.

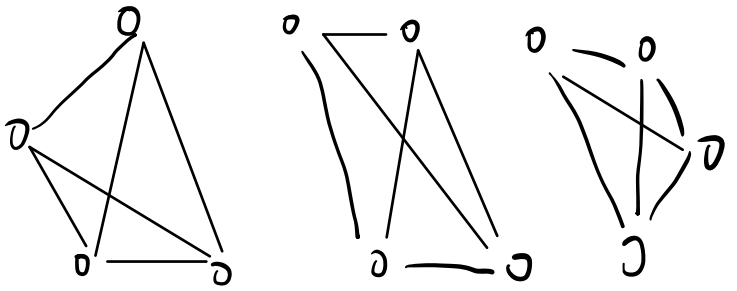
   (c) Suppose I have a d-regular graph and I want to find a path that contains $d+1$ (or more) vertices. It turns out this is easy to do: pick an arbitrary start vertex. Until I have a path with $d+1$ vertices, pick an adjacent vertex to my current one that I have not yet visited. Add that edge to my growing path and set my current vertex to that one.

   Explain in 1-2 sentences why this will always produce a path with at least $d+1$ vertices. How do I know I won't "get stuck" at a vertex until it is at least the $d+1$th vertex I visit?

(a) $9 \times 3 = 27 \mod 2 = 1$ since it is a simple graph

we cannot have odd number of total edges

(b)



(c) since it is d-regular, every vertex has a neighbor vertex to go to. if there is no vertex to go to, then it means we have already visited at least d vertices and the current one can be view at d+1 at least.

4. Suppose I have a `std::vector<unsigned> A`. This vector has only values in the range of 0 to `A.size()`, inclusive. For example, if there are 5 values in A, then the only values it can have are $\{0, 1, 2, 3, 4, 5\}$, although not necessarily in that order. Duplicate values might be present too.

   Obviously, at least one value is missing. Using $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space, where n is `A.size()`, determine which value(s) are missing from the vector. You may not use any standard library data structure other than `std::vector`.

```
std::vector<bool> B(A.size()+1, false);
for (int i=0; i<A.size(); i++)
{
    B[A[i]] = True;
}
for (int j=0; j<A.size()+1; j++)
{
```

```
1   if not B[j]
        cout << j << "is not in the vector"
        <<endl;
}
```