**Question 1.** Find the $\mathcal{O}$-notation for the worst-case running time of the following algorithms. You do not need to, nor should you, give the leading constant or the $n_0$ value. For binarySearch, assume the first listed function is called with a *sorted* vector. For linearSearch, does your answer depend on whether or not the vector is sorted?

```
int linearSearch(const std::vector<int> & numbers, int target)
{
    int i;
    int n = numbers.size();
    for(i=0; i < n; i++)
    {
        if( numbers[i] == target )
        {
            return i;
        }
    }
    throw ElementNotFoundException("Element not found by linear search.");
}
```

```
int binarySearch(const std::vector<int> & numbers, int target);
int binarySearch(const std::vector<int> & numbers, int target, int low, int high);
```

```
int binarySearch(const std::vector<int> & numbers, int target)
{
    return binarySearch(numbers, target, 0, numbers.size() - 1);
}

int binarySearch(const std::vector<int> & numbers, int target, int low, int high)
{
    if( low > high )
    {
        throw ElementNotFoundException("Element not found by binary search.");
    }
    int mid = (low + high) / 2;
    int e = numbers[mid];
    if( e == target )
    return mid;
    else if (target < e)
        return binarySearch(numbers, target, low, mid-1);
    else
        return binarySearch(numbers, target, mid+1, high);
}
```

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 5 | 7 | 8 | 9 | 12 | 14 | 17 | 19 | 22 | 25 | 27 | 28 | 33 |

**Question 2.** F. Lake claims to have found an algorithm to solve `find-min` in an arbitrary vector of size $n$ in better than $\mathcal{O}(n)$ time. Do you believe the claim? Why or why not?

**Question 3.** Suppose we have two algorithms to solve the same problem; that problem has, as input, an array $A$ of size $n$. Would it be better to have an algorithm that takes $20n$ operations or one that takes $n^2$ operations? Why?

**Question 4.** What is the running time of the *Depth-First Search* procedure?

**Question 5.** What is the running time of the *Breadth-First Search* procedure?

```
DFS-iterative(s)
  ∀_v discovered[v] = false
  Initialize S to be a stack with s as its only element
  while S is not empty do
    u ← pop(S)
    if discovered[u] = false then
      discovered[u] = true
      for all edges (u, v) do
        push(S, v)
      end for
    end if
  end while
```

```
BFS(s)
  Set discovered[v] = false for all v
  Set discovered[s] = true
  L[0] ← {s}
  i ← 0
  while L[i] is not empty do
    Make L[i + 1] as empty list
    for all vertices u ∈ L[i] do
      for all edges (u, v) do
        if discovered[v] = false then
          discovered[v] ← true
          Add v to list L[i + 1]
        end if
      end for
    end for
    i ← i + 1
  end while
```

Suppose we have a Stack that is implemented as an *array-based stack*. There is a private member `Object * elements`. When we create a new stack, this is initialized to have ten elements and we use `elements[0]` is the *bottom* of the stack (if it is not empty). For simplicity, assume that the elements stored in the stack are always of fixed width (e.g., **int** or **double**).

We also have a private member data `topi` to tell us in which index is the top of the stack. When we push, if `topi+1` is in-bounds for `elements`, we increment `topi` and place the newly pushed element there.

However, if the current top of the stack is the last element before we push, we need to make more room. We do the following:

- Create a new `Object *` of twice the capacity of the current `elements`.
- Copy everything from `elements` to this new array.
- Delete the old `elements`
- Set the array pointer to point to this new one.

In our linked list stack in project zero, push took $\mathcal{O}(1)$ time. Now push takes $\mathcal{O}(1)$ *most of the time*, but sometimes will take $\mathcal{O}(n)$, where $n$ is the number of elements in the structure.

**Question 6.** Is it correct to say this new push function takes $\mathcal{O}(n)$?

**Question 7.** Is it correct to say this new push function takes $\Theta(n)$?

**Question 8.** What is an accurate way to describe the running time of the new push function?

This question will drive the rest of the lecture. We will answer some other questions along the way before we can answer this one.

Our first step is to do some data gathering. We say one unit of work is the act of writing one `Object` to the array `elements`.

**Question 9.** The sequence so far is only pushes. What if we include a pop in the sequence? How does that affect the total work?

**Question 10.** Form a hypothesis for the total work done if we have $k$ push operations, starting from an empty stack.

**Question 11.** Prove the hypothesis by mathematical analysis. That is, prove that the total work done by $k$ pushes is $\mathcal{O}(f(k))$, for the function $f(k)$ you hypothesized earlier.

**Question 12.** Prove the hypothesis by the credit argument.