5. (25%) In this problem we will explore an efficient strategy for maintaining *extensible* array-based data structures. The arrays used by these structures have no *a priori* upper limit on their size; rather, they grow as the number of elements in them increases. Java's `ArrayList` type implements an extensible array. An extensible array A starts out empty with some fixed number of slots and grows as elements are inserted into it according to the following **doubling rule**:

   > *If we are about to insert an element into A and find that it is full, double the number of slots in A before inserting the new element.*

   Doubling the array size typically cannot be done in-place; it requires that we allocate a new, larger array, then copy all the elements of the old array into the new array.

   Suppose now that we create an empty extensible array with one slot, then insert 2^k elements into it. Justify your answers to each of the following questions.

   (a) What is the final number of slots in the array after all 2^k elements have been inserted? (*Hint*: work out the answer for a few small values of k to see the pattern, then check your guess with an induction proof.)

   There will be 2^k slots in the array, all filled.

   Proof. By induction, we will start with the case $k = 1$. We have to insert $2^1 =$ elements. Because there are initially $2^0 = 1$ slots in the array, we need to double its size to $2^1 = 2$. There are now $2^1 = 2$ slots in the array. Now, assume this is true for the first k elements, that is that after 2^k elements have been inserted, there are 2^k slots in the array. We now wish to show that the same holds true for $k+1$. Wishing to insert the last $2^{k+1} - 2^k$ elements in the array, which is now full, we must now double the size of the array. Therefore the array that previously had 2^k slots has $2 \cdot 2^k = 2^{k+1}$ slots. Because there are exactly 2^{k+1} elements, this is the exact size of the array we need. Further, using the pigeonhole principle, because we have exactly 2^{k+1} elements, each slot in the array will be filled. □

   (b) How many times did we have to double the array size to insert all 2^k elements?

   We needed to double the array k times.

   Proof. It is clear that we needed to double the array exactly once to insert $2^1 = 2$ elements into the array. Therefore, assume we double the array k times to insert 2^k elements. Clearly, if we wish to insert 2^{k+1} elements, we need to double the array again, so, in total, we must double the array $k+1$ times to insert 2^{k+1} elements. □

   (c) How many times total did we have to copy an element of the array over all these doubling operations? (*Hint*: figure out the number of elements copied during each doubling operation, then sum over all doublings.)

   We will always copy $2^k - 1$ elements to insert 2^k elements into this array.

   Proof. Note that for each array doubling, we need to copy the entirety of the existing array into the new array. Let $f(k)$ be the number of total elements copied in all arrays prior to the k^{th} doubling. Clearly, $f(0) = 0 = 2^0 - 1$, $f(1) = 1 = 2^1 - 1$, and $f(2) = f(1) + 2 = 3 = 2^3 - 1$. We wish to show that the total number of elements copied to insert 2^k elements into an array is is given by $2^k - 1$. As we already have our base case, assume this is true for 2^k elements: that is, we have already copied $f(k) = 2^k - 1$ elements. As we must now double the size of our array, we need to copy another 2^k elements. Therefore, the total number of copies is given by $f(k+1) = f(k) + 2^k = 2^k - 1 + 2^k = 2 \cdot (2^k) - 1 = 2^{k+1} - 1$, thus proving our inductive claim. □

(d) What is the *average* number of elements copied per insertion over all 2^k insertions? In other words, what is the ratio of the number of copies to the number of new insertions? Assuming each copy takes constant time, does extensibility affect the average asymptotic complexity of the `insert` operation?

On average, we copy $1 - \frac{1}{2^k}$ elements for each element we insert. This does not affect the asymptotic complexity of our `insert` operation.

Proof. In total, we have inserted 2^k elements, and copied $2^k - 1$ elements to do so. The average number of elements copied per insertion is given by:

$$\text{Average Copies Per Insertion} = \frac{\text{elements copied}}{\text{elements inserted}} = \frac{2^k - 1}{2^k} = 1 - \frac{1}{2^k}$$

To show that this doesn't affect the asymptotic complexity of our insert operation, note that $\lim_{k\to\infty} 1 - \frac{1}{2^k} = 1$. This means that asymptotically, we make 1 copy for each new item we insert into the array. This means that inserting $n = 2^k$ items into an array will take $n \cdot \Theta(1)$ to insert and $n \cdot \Theta(1)$ to copy elements for the extensible array. However, the aggregate operation to copy for insert is still $\Theta(n)$ for $n = 2^k$ inserting elements, because both insert and copy are constant time. □

(e) How can you use the idea of extensible arrays to implement an *extensible hash table*? Such a table grows dynamically but always maintains its load factor at or below some bound $\alpha < 1$. Assume that you can hash a record and place it into the table in constant time.

Proof. Note that $\alpha = \frac{m}{n}$, with $m < n$, where m is an the maximum number of elements in the table. Note that, because the table doubles in size, with every doubling, α remains constant, and $m \to 2m$ and $n \to 2n$. Let k be the number of elements in the table. As we insert more elements into the array, we need to change the hashing function to take into account the new size of the table. In doing so, we must also rehash the values of all current elements of the table, which may change some of the slot values for some objects in the table. However, this is necessary to keep all operations consistent over the new size of the table. □