1. **Describe a recursive algorithm for finding the maximum element in a sequence, *S*, of *n* elements. What is your running time and space usage?**

A function that takes two parameters - the sequence, *S*, and index. The base case will be when the index is equal to *n*, in which case the last element will be returned. In other words, the base case is when the maximum element is at the end of the sequence. If there is no base case, it uses the max() function to compare the current element to a recursive call of the function, but with index+1 as the second parameter.

Running time: $O(n)$
Space usage: $O(n)$

It would look something like:

```
def maxSearch(S, index):
    if index == len(S)-1
        return S[index]
return max(S[index], maxSearch(S[index], index+1)
```

2. **Describe an efficient recursive function for solving the element uniqueness problem, which runs in time that is at most $O(n^2)$ in the worst case without using sorting.**

A function that takes two parameters - the sequence, *S*, and index. The base case is when the index is equal to the length of the sequence, meaning the whole sequence has been checked. Otherwise, the function will compare the current element to all elements in the sequence through the use of a loop. The index is incremented in a recursive call to the function.

Could look something like:

```
def uniqueCheck(S, index):
    if index == len(S)-1:
        return True
    else:
        unique = True
        for i in range(index, len(S)):
            if S[index] == S[i]:
                unique = False
        return unique and uniqueCheck(S, index+1)
```

3. **Write a short recursive Python function that takes a character string *s* and outputs its reverse. For example, the reverse of 'pots&pans' would be 'snap&stop'.**

```
def reverseString(str, index):
    if index == len(str)-1:
        return S[index]
    else:
        str2 = reverseString(str, index+1)
        str2.append(S[index])
        if index == 0:
            str2 = '' + str2
        return str2
```

4. **Execute the experiment from Code Fragment 5.1 and compare the results on your system to those we report in Code Fragment 5.2.**

On my computer, the size in bytes were exactly 8 bytes smaller than the example in the book, but everything adds up.

```
C:\Users\Jake\Documents\UMaine\COS\COS 226>
Length:   0; Size in bytes:   64
Length:   1; Size in bytes:   96
Length:   2; Size in bytes:   96
Length:   3; Size in bytes:   96
Length:   4; Size in bytes:   96
Length:   5; Size in bytes:  128
Length:   6; Size in bytes:  128
Length:   7; Size in bytes:  128
Length:   8; Size in bytes:  128
Length:   9; Size in bytes:  192
Length:  10; Size in bytes:  192
Length:  11; Size in bytes:  192
Length:  12; Size in bytes:  192
Length:  13; Size in bytes:  192
Length:  14; Size in bytes:  192
Length:  15; Size in bytes:  192
Length:  16; Size in bytes:  192
Length:  17; Size in bytes:  264
Length:  18; Size in bytes:  264
Length:  19; Size in bytes:  264
Length:  20; Size in bytes:  264
Length:  21; Size in bytes:  264
Length:  22; Size in bytes:  264
Length:  23; Size in bytes:  264
Length:  24; Size in bytes:  264
Length:  25; Size in bytes:  264
```

5. **Given a set of integers implemented as an array of size $n \geq 2$. Suppose that one - but only one - integer is repeated, though it could be repeated any number of times. Describe (no implementation is needed) a fast ($O(n^2)$ or better) algorithm that removes the duplicates.**

You could implement the uniqueCheck algorithm from question 2, but when a duplicate value is found, instead of returning False, remove that element and shift the rest of the elements down.

6. **Give an implementation of a function, with signature remove_all(data, value), that removes all occurrences of value from the given list, such that the worst-case running time of the function is O(n) on a list with n elements.**

```
def remove_all(data, value):
    toRemove = 0
    for i in range(len(data)):
        if data[i] == value:
            toRemove++
        else:
            data[i-toRemove] = data[i]
```

7. **Suppose we implement a stack as a dynamic array with operations push & pop. Use amortization to find the running time of a sequence of n push and pop operations.**

Wasn't sure how to amortize this. Still having trouble grasping amortization in general, but more so what it's purpose is and what we gain from it. But, I know the running time for adding to/removing from an array is $O(n)$.