1. Find the worst-case time complexity for the OddMedian algorithm below. Show all work.

   You may use the facts that you can add an element to an array in $\Theta(1)$, remove an element from an array in $\Theta(1)$, find the size of an array in $\Theta(1)$, and find the minimum or maximum of an array of size x in $\Theta(x)$ time. You may assume that lo and hi both have $O(i)$ elements during every iteration of the for loop.

```
Input: data: an array with an odd number of integers
Input: n: the length of data (odd)
Output: element m in data such that (n − 1)/2 elements are smaller
        and (n − 1)/2 elements are larger
1  Algorithm: OddMedian
2  med = data[1]
3  lo = {}
4  hi = {}
5  for i = 1 to n − 1 do
6      if data[i + 1] < med then
7          if lo has less than i/2 elements then
8              Add data[i + 1] to lo
9          else
10             Add med to hi
11             Add data[i + 1] to lo
12             med = max(lo)
13             Remove med from lo
14         end
15     else
16         if hi < i/2 then
17             Add data to hi
18         else
19             Add med to lo
20             Add data[i + 1] to hi
21             med = min(hi)
22             Remove med from hi
23         end
24     end
25 end
26 return med
```

Line 5 For
Since the outer loop steps by 1 until it reaches n-1 it is Θ(n)

Line 6 and Line 15 If Else statement
Since this is a comparison both are in constant time Θ(1)

Line 7 and Line 16 If statements
These would be the BEST case since they only add elements to an array at constant time Θ(1)

Line 9 and Line 18 Else statements
However we want the worst case for the time complexity. Each would have a total of 3 commands to add/remove the lo array and add/remove the hi array, all of which happen in constant time Θ(1), but then they have to find a new med as either the max in lo or min in hi which happens in Θ(n) time when the size of the respective array is n. Therefore the min max functions are O(n).

Analysis
This means that the worst case run time is [Θ(n) loops of Θ(1)*Θ(1)*Θ(1)*O(n)*Θ(1)], or Θ(n)*O(n). Therefore the worst case run time is $O(n^2)$

2. Find the worst-case time complexity for the StrangeSum algorithm below. Show all work.

```
Input: data: an array of n integers
Input: n: the length of data
Output: ∑_{i=1}^{n} data[i]
1 Algorithm: StrangeSum
2 d = 2
3 while d < n do
4     for i = 1 to n step d do
5         data[i] = data[i] + data[i + d/2]
6     end
7     d = 2d
8 end
9 return data[1]
```

Line 7 d = 2d
doubles the value of the d iterator on each iteration (at Θ(1))

Line 3 While
Due to d starting at 2 and doubling with each iteration of the loop each loop will cause the time it reaches n to be halved, therefore it is Θ(n/d), or log n.

Line 4 For
The for loop iterates from 1 to n but the iterator increments by the value of d each time. That means that, similar to the while loop, with each increase of d the time it takes to reach n is halved making it also Θ(n/d), or log n.

Line 5
Adds array elements and stores into array in constant time Θ(1)

Analysis
This means that the worst case run time is [log n loops of (log n * Θ(1)) * Θ(1)], or log n * log n. Therefore the worst case run time is $O((\log n)^2)$

3. Develop a recurrence for the worst-case time complexity for the abSearch algorithm below. Show all work. When describing the recurrence, you may assume that n is even; that is, that $\lfloor n/2 \rfloor = \lceil n/2 \rceil = n/2$.

```
   Input: str: a string of length n
   Input: n: the length of str
   Output: the first index i such that str[i] = a and str[i + 1] = b, or −1
           if str doesn't contain the substring "ab"
1  Algorithm: abSearch
2  if n < 2 then
3  |  return −1
4  else if n = 2 then
5  |  if str = "ab" then
6  |  |  return 1
7  |  else
8  |  |  return −1
9  |  end
10 else
11 |  midpt = ⌊n/2⌋
12 |  left = abSearch(str[1..midpt])
13 |  center = abSearch(str[midpt..midpt + 1])
14 |  right = abSearch(str[midpt + 1..n])
15 |  if left ≠ −1 then
16 |  |  return left
17 |  else if center ≠ −1 then
18 |  |  return midpt
19 |  else if right ≠ −1 then
20 |  |  return right + midpt
21 |  else
22 |  |  return −1
23 |  end
24 end
```

Since the function outputs the first index that contains ab or -1 if ab is not found the worst case run time will occur when ab is not present in the input, meaning the function must check every index value.

Line 2 If to Line 3
Comparison that returns -1 if true, constant time $\Theta(1)$

Line 4 Else If to Line 9
Comparison to see if ab is present in size 2 string, returns 1 or -1 and happens in constant time $\Theta(1)$

Line 10 Else to Line 23
Line 11 chooses a midpoint at constant time $\Theta(1)$ and then makes three recursive calls. Left is the range from 1...n/2 and right is the range from n/2+1...n while center is at most 2 and will therefore be handled by the constant If Else statements on lines 2 to 9.

Lines 15 to 20 are comparisons which will happen in constant time $\Theta(1)$, as will the final else statement at 21.

Analysis
Therefore each iteration will produce $T(n/2)+T(n/2)+\Theta(1)$ for left+right+center so we can say the recurrence is $T(n)=2T(n/2)+\Theta(1)$ for $n > 2$ (since if $n \leq 2$ it will be handled in constant time)