1. (a) $T(n)$ is $O(n^2)$ because the loop at line 2 runs for at most n times and the loop in line 5 which is nested in the first loop runs at most n times. The line inside both for loops takes constant time therefore $T(n) \in O(n^2)$

   (b) Consider the loop at line 2 which runs at least from $A.size/2..A.size$, as well consider the condition is met so that the for loop in line 5 runs. The for loop in line 5 then runs from $1..A.size/2$ at least. Again the line in the for loop takes constant time so the loop has run at least $c(A.size/2) \times (A.size/2)$ which is $\Omega(n^2)$

2. Tony's claim is true.

```
LIST(H, k):
if H[1] > k HALT
print H[1]
up = False
current = 1
while current < H.size:
   if up == True:
       current = current/2
   else if current*2 < k:
       print H[current*2]
       current = current*2
       continue
   if current*2+1 < k:
       print H[current*2+1]
       current = current*2 +1
       up = False
       continue
   else: up = True
```

3. (a) This algorithm uses a min-heap. input is a stream that has a function Next which is null if the stream has ended. If the input is query it prints the first m numbers in the min-heap, which will be the smallest because of the min-heap property. When there is a number, it appends it and then rearranges the elements to satisfy the heap property.

```
QUERY(m, input)
H = [0] // This is a empty heap
while in = input.Next():
    if in == 'query':
        for i = 1..m:
            print H[i]
    else:
       H.append(in)
       curr = H.size
       while (curr > 1 and H[curr] < H[curr/2]):
           H[curr], H[curr/2] = H[curr/2], H[curr]
           curr = curr/2
```

(b) It is $O(\log(m))$ for key input because append is constant time and the rearrangement takes log(m) swaps. The query operation is $O(m)$ because it outputs the first m elements, each output takes constant time.

4. (a) If k > x.key then compare k to the key of x's parent, if k is bigger than that swap, else we're done, continue this until k $\leq$ x.parent.key. Then change x's key. This is $O(\log(n))$ because there at most $\log(n)$ levels to swap and swapping keys takes a constant amount of time.

   (b) First move x to the root of its S_k tree treating x as positive infinity. Do this by swapping x with it's parent until it is at the top, similar to part a. Then rearrange pointers so that the roots of the trees on either side point to each other, and x's children no longer point to x, so that x is removed from the tree. Then merge the two binomial max heaps, one of all the other S_k trees and the other consisting of x's children, following the algorithm we learned in class.

5. (a) The underlying data structure I am using is a Binomial Heap. To implement the extra operation there are two sets of pointers for each key. One that implements a max-heap, and another that implements a min-heap.

   (b) Insert is done the normal way except it is done twice, once for each set of pointers. Merge is done similarly as well, one set of operations for each set of pointers. ExtractMin and ExtractMax are done differently. ExtractMin starts out like a normal ExtractMin on the min-heap pointers, then a delete operation is performed on the max-heap pointers as per what was outlined in 4(b) ExtractMax is similar to ExtractMin, it starts out with a ExtractMax on the max pointers, then a delete node on the min-pointers. All these operations are $O(\log(n))$ because Insert and Merge are the same except done twice so it differs from a regular heap by a constant factor which is 2. ExtractMin and ExtractMax are the original operations plus a delete operation, these operations are both $O(\log(n))$, so the total operation is also $O(\log(n))$.