Gurumanie Singh Dhiman (COMS3110 HW2)

1.

```
findLargestIndex(A, i) {
      left = 0, right = len(A)−1, lastIndex = −1
      while (left ≤ right) {
            i = (left + right) / 2
            if A[i] == 0 {
                  lastIndex = i
                  left = i + 1      # search right half
            } else {
                  right = i − 1     # search left half
            }
      }
      return lastIndex
}
```

The above code returns -1 if the lastIndex is not found or there is no 0's. It also checks the case where the i might land on the lastIndex (in the middle) and returns it as the lastIndex.

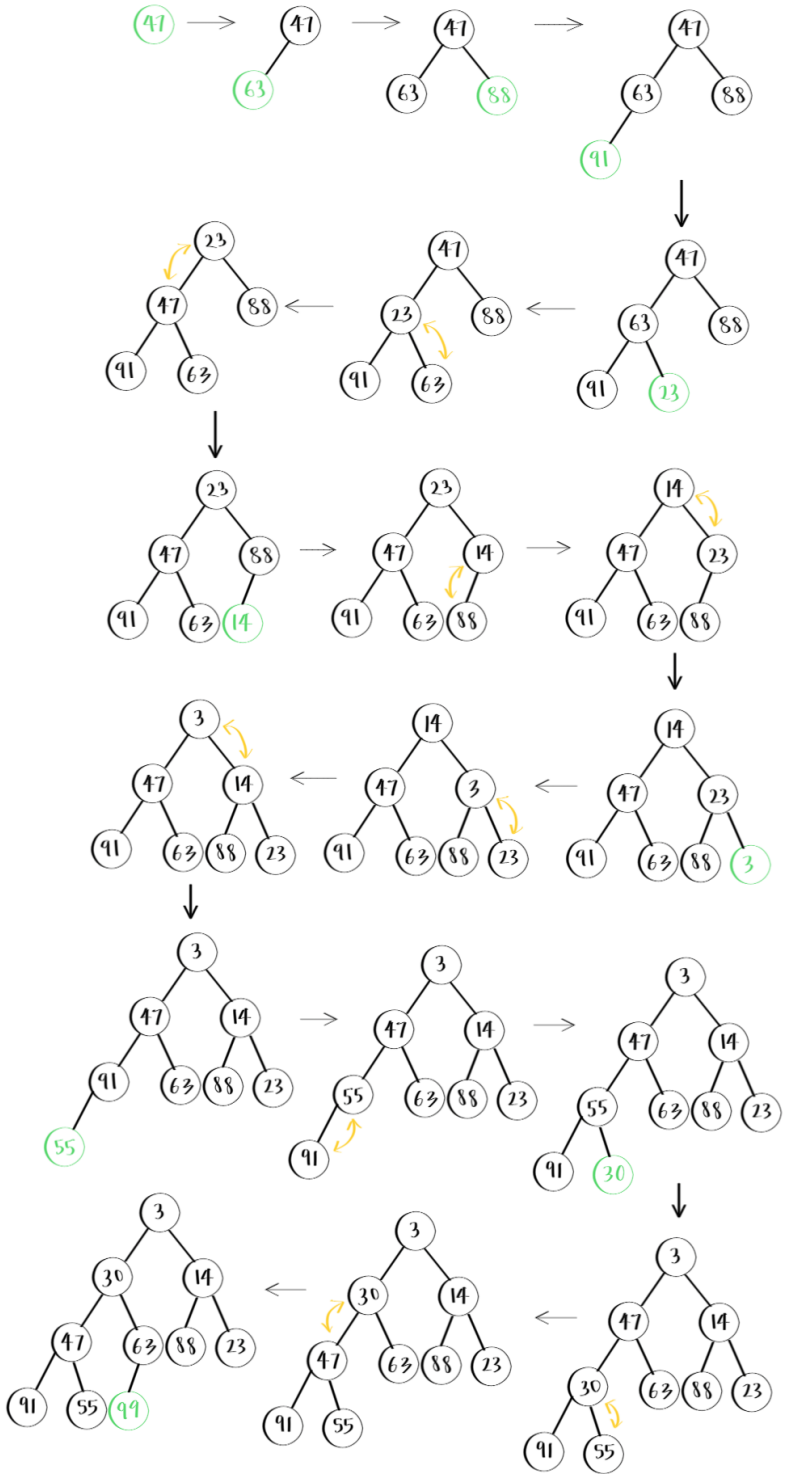
Runtime = O(log n)

2.

```
largestSubarray(array A) {
HashTable T<sum, index>, T[0] = 0, prefixSum = 0, currLen = 0, subLen = 0
      for (i = 1 to len(A)) {
            prefixSum += A[i]
            if (prefixSum exists in T) {
                  currLen = i − T[prefixSum]
                  if (currLen > subLen) {
                        subLen = currLen
                  }
            } else {
                  T[prefixSum] = i
            }
      }
      return subLen
}
```

The hashtable search has expected runtime of O(1), all other functions have a runtime of O(1) as well. Since we loop over this n times. **Expected: O(n).**

3.



Final Order: 3, 30, 14, 47, 63, 88, 23, 91, 55, 99.

4.

```
kAlmostSorter(A,k){
   // Let mH be a minHeap
   temp = 0
   for i = 1 to k + 1 {
      mH[i] = A[i]
   }
   for i = k + 1  to n + k {
      mH.heapifyUp()
      OutputArr[] = mH.extractMin() // extract smallest element from heap
                                    (and append to output) then replace with
                                    leaf
      if (i < n) {
            mH.add(A[i])
      }
   }
}
```

Runtime for the above O(n log k) where k < n for a nearly sorted array.