Homework #5 - 2.19 / 2.23

2.19) Suppose we have k sorted arrays with n-elements each, & we will combine them into 1 kn element array

a) Using merge (pg 51), complexity in k & n

Linear ← function merge (x[1...k], y[1...l]) (pg. 51)

```
if k=0 return y[1...l]
if l=0 return x[1...k]
if x[1] ≤ y[1]:
    return x[1] ∘ merge(x[2..k], y[1..l])
else
    return y[1] ∘ merge(x[1..k], y[2...l])
```

Merging 2 arrays of k length has time complexity $O(n+n)$. Because the next array is merged after the previous one finishes, the third array would merge at $O(2n+n)$ or $O(3n)$. A total of $k-1$ merges will occur, so the final merge will take $O((k-1)n)$

b) The linear performance happens because the algorithm iterates over each array more than once until they are combined into one large array. A better solution would take the total K arrays & recursively divide K arrays by 2 until arrays are in groups of 2. These groups can be merged & returned, then merged again until the array is completed. The merge still takes the same time, but happen much less, log k times, so the time complexity would be $O(n \log k)$, which is improved over the original example.

```
merge(x[1...k], y[1...l])
 if k = 0: return y[1...l]
 if l = 0: return x[1...k]
 if x[1] ≤ y[1]:
   return x[1] ∘ merge(x[2...k], y[1...l])
 else:
   return y[1] ∘ merge(x[1..k], y[2..l])

mergelog(arrays[0...k-1]
   resultarrays[ ]
 for i = 0 < (k-1)/2
   mergearray merge(arrays[i], arrays[i+1])
   resultarray[i] = mergearray
 if resultarray length = 1
    return resultarrays[0]
 else
   mergelog(resultarrays[])
```

2.23 a)

```
majority value ( A.[1..n])
  if A length = 1
    return A[i]
  A_1 = A[1.. n/2]
  A_2 = [n/2+1 ... n]
  A1majority = majorityvalue(A_1)
  A2majority = majorityvalue(A_2)
  if A1majority == A_2 majority
    return A1 majority
  A1hasmajority = ismajority(A_1, A1majority)
  A2hasmajority = ismajority(A_2, A2majority)
  if A1hasmajority is true
    return A1majority
  else if A2has majority is true
    return A2majority
  else
    return null
```

Split Array in 1/2 each iteration until length = 1. This allows for $\log n$ iterations.

$O(n)$ linear search for majority value from previous iteration

If either half of the array has a majority value return it, if not then return null. If both halves dont have a majority value the whole cannot.

```
ismajority(A, majority)
  for i in A
    if A[i] = majority
      count ++
  if count > A length/2
    return true
  else
    return false
```

$O(n)$, only iterates once through array to count a value

majorityvalue(A) = $O \log n$

is majority(A_1, majority) = $O(n)$

Total time complexity = $\boxed{O(n \log n)}$

23.3 b) Majoritylinear(A[1..n])

```
if A length = 2
    if A[1] = A[2]          ] if final 2 values are the same, A[1]/A[2]
        return A[1]         ] is the majority value
    else
        return null         ] Else there is none
create majorityvalue array
for i in A length/2         ] Iterate half through array to look for
    if A[i] == A[i+1]       ] matching pairs to test for possible majority
        add A[i] to majorityvalue  ] Add to new array if pair matches
return majoritylinear(majorityvalue)  ] Recursive call on array that is now
                                        n/2 size
```

Using the master theorem, $a=1$ $b=2$ $d=1$ because the pre work before the recursive call is $O(n/2)$ or $O(n)$ to iterate through the array. Plugging into the master Theorem we get

$$T(n) = 1T\left(\frac{n}{2}\right) + O(n^1) = T\left(\frac{n}{2}\right) + O(n)$$

If we plug in for $\frac{a}{b^d}$ we get $\frac{1}{2^1} = \frac{1}{2} < 1$, so our complexity is $O(n^d)$, which is also $O(n)$, confirming the linear time complexity.