Homework #5 Key

2.19

a. Merging the first and second lists will require $2n$ operations. Merging the third list with this result will require an additional $3n$ operations. The fourth, an additional $4n$, and so on up to the kth list, which will require an additional kn operations. Since there are a total of $k-1$ of these varying merge operations, the overall time complexity for this approach is $k(k-1)n = O(k^2n)$.

b. A divide and conquer approach could recursively split the set of k lists into two sets of $k/2$ lists, with the base case being 2 lists, as follows:

function K-MERGE($l_1, l_2, ..., l_k$)
  if $k = 2$ **then**
    return Merge(l_1, l_2)
  else
    m = k-Merge($l_1, l_2, ..., l_{\frac{k}{2}}$)
    q = k-Merge($l_{\frac{k}{2}+1}, l_{\frac{k}{2}+2}, ..., l_k$)
    return Merge(m, q)

where Merge() is a normal 2-way, linear merge of 2 sorted lists that returns a single sorted list. Assuming k is a multiple of two here for simplicity, we can apply the Master Theorem, with a bit of care. Note that we are recursing here on k, the number of lists, rather than on n, the length of the lists. At each level, we have two children, so $a = 2$; and, each child is half the size of its parent, so $b = 2$; the amount of work to combine two sorted lists is linear, so $d = 1$; however, we must be careful to include n in that linearity, as it is not constant, so the amount of work to combine lists at each level is $(kn)^d = kn$. With $a = 2$, $b = 2$ and $d = 1$, we have case 2, with gives a runtime of $\theta((kn)^d \log k) = \theta(nk \log k)$.

2.23

a. Here is a divide and conquer algorithm for finding the majority element of an array A (if it exists):

```
1: function MAJORITY(A)
2:     if |A| = 0 then return NULL
3:     if |A| = 1 then return A[1]
4:     p ← Majority(A_L)                          ▷ left half of array
5:     q ← Majority(A_R)                          ▷ right half of array
6:     if Count(p, A_L A_R) > |A_L A_R|/2 then return p
7:     if Count(q, A_L A_R) > |A_L A_R|/2 then return q
8:     return NULL
 function COUNT(b, A)
    c ← 0
    for all a ∈ A do
        if a=b then c ← c + 1
    return c
```

Claim

For any string of symbols A, `Majority(A)` returns the majority element m of A, if it exists; if A has no majority element, `MajorityElement(A)` returns *null*.

Proof. We proceed by strong induction on the length k of the list A.

Base case: The base case for no majority is an empty string (handled in line 2) and for the case of a majority a string of length 1 (handled in line 3). In both situations, the base case is obviously correct by inspection.

Induction step: Assume that the algorithm returns the majority element m for any list of length $i, 1 \leq i \leq k$ which contains a majority element and returns NULL otherwise. We show that the algorithm correctly returns m or NULL for a list of length $k+1$.

Because $|A_L|, |A_R| = (k+1)/2$, by the inductive assumption p and q returned in the recursive calls of lines 4 and 5 will be the majority element of $|A_L|$ and $|A_R|$, respectively, or NULL if no majority exists in the sub-list.

If A has a majority element m, then either $p = m$ or $q = m$ (or both). Lines 6 and 7 determine which and therefore correctly return m.

Conversely, if A has no majority, then either $p = q =$ NULL (neither sub-list has a majority), and thus the if statements of lines 6 and 7 both fail, and line 8 correctly returns NULL. Otherwise, $p \neq q$ so neither can be a majority in A, and thus the if statements of lines 6 and 7 both fail, and line 8 correctly returns NULL.

□

Claim

The runtime of `Majority` is $O(n \log n)$, where $n = |A|$ for input A.

Proof. Because the runtime of the algorithm can be expressed in the form

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + \mathrm{O}(n^d)$$

we can use the Master Theorem to obtain its time complexity. Since the number of children for each recursive level is two and each child is half the size of its parent, $a = 2, b = 2$. Work for recombination is constant except for calls to the `Count` function, which is linear, so $d = 1$. With $a = 2, b = 2, d = 1$, the geometric ratio $r = \frac{a}{b^d} = \frac{2}{2}$, and the Master Theorem therefore gives a runtime of $\mathrm{O}(n^d \log n) = \mathrm{O}(n \log n)$.

□

b. Here is another divide and conquer algorithm for finding the majority element of an array A (if it exists):

```
1:  function MAJORITY2(A)
2:      if |A| = 0 then return NULL
3:      if |A|%2 ≠ 0 then                                   ▷ ODD
4:          if Count(A[0], A) ≥ |A|/2 then return A[0]
5:          m ← Majority2(A[1 :])
6:      else                                                ▷ EVEN
7:          B ← []
8:          for all i in range(0, |A|/2, 2) do
9:              if A[i] = A[i + 1] then
10:                 B.append(A[i])
11:         m ← Majority2(B)
12:     if Count(m, A) ≥ |A|/2 then return m
13:     return NULL
  function COUNT(b, A)
      c = 0
      for all a ∈ A do
          if a=b then c ← c + 1
      return c
```

Claim

For any string of symbols A, `Majority2(A)` returns the majority element m of A, if it exists; if A has no majority element, `MajorityElement(A)` returns *null*.

Proof. For the case that A has a majority element m, we proceed by strong induction on the length k of the list A.

Base case ($k = 0$): handled in line 2, this is obviously correct as the return value for a *null* list is *null*.

Induction step: Assume that the algorithm returns the majority element m for any list of length $i, 1 \leq i \leq k$. We show that the algorithm returns m for a list of length $k + 1$.

If $k + 1$ is odd (line 3), either $A[0] = m$, or it does not. If so, the call to `Count()` on line 4 will detect this and the algorithm returns m. If $A[0] \neq m$,

the recursive call (line 5) reduces the length of A to k, and by the inductive assumption correctly returns m.

If $k+1$ is even (line 6), we must show that m is also the majority element of B. Let c_m be the total number of m in A, and let $c_{\neg m} = |A| - c_m$ be the total number of all other symbols in A. Note that since $|A|$ is even, $c_m \geq c_{\neg m} + 2$. Let $P_=$ be the set of pairs $(i, i+1)$, such that $A[i] = A[i+1]$ for $0 \leq i \leq |A|-1, i \mod 2 = 0$ and let $P_\neq$ be the set of pairs $(i, i+1)$, such that $A[i] \neq A[i+1]$ for $0 \leq i \leq |A|-1, i \mod 2 = 0$. Let P_a be the set of pairs $(i, i+1)$, such that $A[i] = A[i+1] = a$ for $0 \leq i \leq |A|-1, i \mod 2 = 0$ for any symbol a. Then, $2|P_m| \geq c_m - |P_\neq| \implies 2|P_m| > c_{\neg m} - |P_\neq|$ and $2|P_{\neg m}| \leq c_{\neg m} - |P_\neq|$. So, $|P_m| > |P_{\neg m}|$. Since $|B| = |P_=|$, with B containing $|P_a|$ a's for each symbol a represented in $P_=$ (lines 8-10), B will contain more m's than all other symbols combined. Finally, because $|B| \leq (k+1)/2$, by the inductive assumption the recursive call (line 11) will return m.

For the case that A has no majority element, the algorithm always returns *null* because of the base case (line 2) and the verification performed in lines 12-13. □

Claim

The runtime of `Majority2` is $\mathrm{O}(n)$, where $n = |A|$ for input A.

Proof. Because the runtime of the algorithm can be expressed in the form

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + \mathrm{O}(n^d)$$

we can use the Master Theorem to obtain its time complexity. Here, $a = 1$ because there is only a single recursive call at each level; $b = 2$ because in the worst case, the recursion reduces the length of the list by at least half every other level; and $d = 1$ because `Count()` and computing B are $\mathrm{O}(n)$, and computing $A[1:]$ is not greater than $\mathrm{O}(n)$. With $a = 1, b = 2, d = 1$, the geometric ratio $r = \frac{a}{b^d} = \frac{1}{2}$, and the Master Theorem therefore gives a runtime of $\mathrm{O}(n^d) = \mathrm{O}(n)$. □