Problem Set 2

Solution Key

Problem 1

Part (a)

$$H_0 = \begin{bmatrix} 1 \end{bmatrix}, \quad H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad H_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

Part (b)

$$H_0 \cdot (z) = [z], \quad H_1 \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x+y \\ x-y \end{bmatrix}, \quad H_2 \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} a+b+c+d \\ a-b+c-d \\ a+b-c-d \\ a-b-c+d \end{bmatrix}$$

Part (c)

Let $\vec{v}_{\text{top}}$ be a vector of size 2^{k-1} consisting of the first half of $\vec{v}$, and let $\vec{v}_{\text{bot}}$ be a vector of the same size consisting of the second half of $\vec{v}$.

Call our black box on $\vec{v}_{\text{top}}$ and assign the output to a vector $\vec{a}$. Do the same for $\vec{v}_{\text{bot}}$ and assign the output to a vector $\vec{b}$.

Let $p = \vec{a} + \vec{b}$ and $q = \vec{a} - \vec{b}$ be vectors of size 2^{k-1}. We then form $\vec{v'} = H_k \cdot \vec{v}$ by appending $\vec{q}$ to $\vec{p}$.

Proof of Correctness

Consider the following derivation:

$$\begin{aligned}
H_k \cdot \vec{v} &= \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right] \cdot \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{2^k} \end{bmatrix} && \text{definition of } H_k && (1)\\
&= \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right] \cdot \begin{bmatrix} \vec{v}_{\text{top}} \\ \vec{v}_{\text{bot}} \end{bmatrix} && \text{definition of } \vec{v}_{\text{top}} \text{ and } \vec{v}_{\text{bot}} && (2)\\
&= \begin{bmatrix} [H_{k-1} \cdot \vec{v}_{\text{top}} + H_{k-1} \cdot \vec{v}_{\text{bot}}] \\ [H_{k-1} \cdot \vec{v}_{\text{top}} - H_{k-1} \cdot \vec{v}_{\text{bot}}] \end{bmatrix} && \text{matrix multiplication} && (3)\\
&= \begin{bmatrix} \vec{a} + \vec{b} \\ \vec{a} - \vec{b} \end{bmatrix} && \text{definition of } a \text{ and } b && (4)\\
&= \begin{bmatrix} \vec{p} \\ \vec{q} \end{bmatrix} && \text{definition of } p \text{ and } q && (5)\\
&= \vec{v'} && \text{definition of } \vec{v'} && (6)
\end{aligned}$$

The only step that may need a little more justification is (3).

To see why (3) is true, consider the process of computing the matrix multiplication in (2). To get the i^{th} element of the resulting vector, we compute the sum

$$\sum_{j=1}^{2^k} h_{ij} v_j$$

We can break this sum into these two chunks:

$$\left(\sum_{j=1}^{2^{k-1}} h_{ij} v_j \right) + \left(\sum_{j=1+2^{k-1}}^{2^k} h_{ij} v_j \right)$$

Now when $1 \leq i \leq 2^{k-1}$, we see that these two terms are in fact just

$$\left[i^{\text{th}} \text{ row of } H_{k-1} \right] \cdot \vec{v}_{\text{top}} \quad + \quad \left[i^{\text{th}} \text{ row of } H_{k-1} \right] \cdot \vec{v}_{\text{bot}}$$

And when $1 + 2^{k-1} \leq i \leq 2^k$, then the terms become

$$\begin{aligned}
& \left[i^{\text{th}} \text{ row of } H_{k-1} \right] \cdot \vec{v}_{\text{top}} \quad + \quad \left[i^{\text{th}} \text{ row of } -H_{k-1} \right] \cdot \vec{v}_{\text{bot}} \\
= \; & \left[i^{\text{th}} \text{ row of } H_{k-1} \right] \cdot \vec{v}_{\text{top}} \quad - \quad \left[i^{\text{th}} \text{ row of } H_{k-1} \right] \cdot \vec{v}_{\text{bot}}
\end{aligned}$$

Hence the first half of the entries of the resulting vector are given by the following vector sum:

$$[H_{k-1} \cdot \vec{v}_{\text{top}}] + [H_{k-1} \cdot \vec{v}_{\text{bot}}]$$

And the second half is given by

$$[H_{k-1} \cdot \vec{v}_{\text{top}}] - [H_{k-1} \cdot \vec{v}_{\text{bot}}]$$

as claimed in (3). □

Part (d)

Apply-Hadamard($k, \vec{v}$)

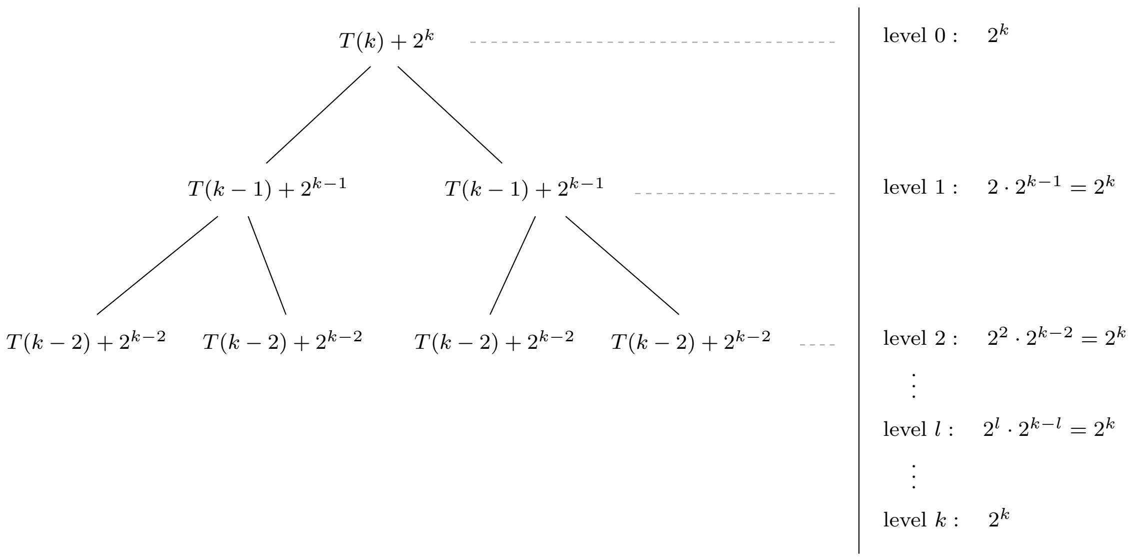
Takes a non-negative integer k and a vector $\vec{v}$ (of the appropriate size) and returns $H_k \cdot \vec{v}$.

```
1   if $k = 0$
2      then return $\vec{v}$
3   else do
4      $\vec{v}_{\text{top}} \leftarrow$ a vector consisting of the first $2^{k-1}$ components of $\vec{v}$
5      $\vec{v}_{\text{bot}} \leftarrow$ a vector consisting of the last $2^{k-1}$ components of $\vec{v}$
6      $\vec{a} \leftarrow$ Apply-Hadamard($k-1, \vec{v}_{\text{top}}$)
7      $\vec{b} \leftarrow$ Apply-Hadamard($k-1, \vec{v}_{\text{bot}}$)
8      $\vec{p} \leftarrow \vec{a} + \vec{b}$
9      $\vec{q} \leftarrow \vec{a} - \vec{b}$
10     $\vec{v'} \leftarrow \vec{q}$ appended to $\vec{p}$
11     return $\vec{v'}$
```

Part (e)

The recursive algorithm given above makes two calls to itself on input $k-1$, and then does 2^k additional work when it does constant-time computations to find each entry of v'. So, the height of the recursive-call tree is k, since after k recursive calls $k = 0$ and the recursion ends. Referring to the diagram of the recurrence tree below (where T denotes the recursive function), we see that the total work done on a given level l of the tree is thus $2^l \cdot 2^{k-l} = 2^k$, where the level of the root is 0. Hence the total work done over all levels of the tree is $k \cdot 2^k$, giving a time complexity of $O(k \cdot 2^k)$ for our algorithm.

Recurrence Tree for Apply-Hadamard(k, v)



□

Problem 2

a) **Give an algorithm and run-time for $k = 2$**

Run-time here is $O(n \log n)$.

With $k = 2$ we can get away with sorting A and then traversing the list once, looking for any pair of adjacent elements with equivalent values. If we find such a set, we've found a 2-repeat.

```
$A \leftarrow$ MergeSort(A)
for all $i = 1$ to $|A| - 1$ do
  if $A[i] = A[i+1]$ then
    return TRUE
  end if
end for
return FALSE
```

b) **Give an algorithm and run-time for $k = n/4$**

Run-time here is $O(n)$. Note this is a special case of the algorithm for general k on the next page.

Algorithm 1 DetectRepeat(A)

```
1:  $m \leftarrow$ PositionOfMedianDeterministic(A)
    Partition list $A$ into 3 sets: elements less than ($L$), equal to ($E$), and greater than ($G$) the median
    Note that this can be done with one traversal of list $A$
2:  $L \leftarrow \{A[j] : A[j] < A[m]$ for $1 \leq j \leq |A|\}$
3:  $E \leftarrow \{A[j] : A[j] = A[m]$ for $1 \leq j \leq |A|\}$
4:  $G \leftarrow \{A[j] : A[j] > A[m]$ for $1 \leq j \leq |A|\}$
    If we've found at least $n/4$ elements in $E$, we're done
5:  if $|E| \geq \frac{n}{4}$: return TRUE
    Otherwise, search recursively in $L$ and $G$, but only if they could contain $\geq \frac{n}{4}$ elements
6:  if $|L| \geq \frac{n}{4}$ then
7:    if DetectRepeat( $L$ ): return TRUE
8:  end if
9:  if $|G| \geq \frac{n}{4}$ then
10:   if DetectRepeat( $G$ ): return TRUE
11: end if
    Reaching this statement implies no $n/4$-repeat was found in $E, L$, or $G$
12: return FALSE
```

c) **Design an algorithm for general k**

Specified in Alg. 2. Note that we rely on a deterministic, worst-case linear time median finding algorithm `PositionOfMedianDeterministic`, which returns the index of the median element in list A. If no single element is the true median, it returns the index of the closest entry less than the median. For example, if $A = [1,\ 2]$, the true median of A is 1.5, and `PositionOfMedianDeterministic`(A) would return 1.

Algorithm 2 DetectKRepeat(A, k)

```
1: $m \leftarrow$ PositionOfMedianDeterministic(A)
   Partition list $A$ into 3 sets: elements less than ($L$), equal to ($E$), and greater than ($G$) the median
   Note that this can be done with one traversal of list $A$
2: $L \leftarrow \{A[j] : A[j] < A[m]$ for $1 \leq j \leq |A|\}$
3: $E \leftarrow \{A[j] : A[j] = A[m]$ for $1 \leq j \leq |A|\}$
4: $G \leftarrow \{A[j] : A[j] > A[m]$ for $1 \leq j \leq |A|\}$
   If we've found at least $k$ elements in $E$, we have detected a $k$-repeat.
5: if $|E| \geq k$ then
6:   return TRUE
7: end if
   Otherwise, search recursively in $L$ and $G$, but only if they could contain at least $k$ elements
8: if $|L| \geq k$ then
9:   if DetectKRepeat( $L$, $k$ ) then
10:     return TRUE
11:   end if
12: end if
13: if $|G| \geq k$ then
14:   if DetectKRepeat( $G$, $k$ ) then
15:     return TRUE
16:   end if
17: end if
   Reaching this statement implies no $k$-repeat was found in $E, L$, or $G$
18: return FALSE
```

d) **Prove correctness of the algorithm**

We represent the execution of `DetectKRepeat(`A, k`)` as a binary tree of recursive calls. Each node is a single execution of the function `DetectKRepeat`. The original (root) call is at depth 0. Each call has at most two possible child nodes (recursive calls in lines 9 and 14 in Alg. 2).

Let A_d be a provided list at some node at depth d in the recursive tree.

Correctness proof via induction

Claim: If a k-repeat exists (does not exist) in A_d, then `DetectKRepeat(`A_d, k`)` will return TRUE (FALSE).

Proof: By induction over the *depth* d of the recursion.

Base Case: Let A_d be the provided list at a leaf node in the recursive tree. This means after partitioning, both $|L| < k$ and $|G| < k$ (which prevents any further recursive calls). We consider all possible conditions for provided list A_d

- A_d has a k-repeat. Both L and G have less than k elements, so the partition must have placed the k-repeat in set E, since a k-repeat element must have been the median. This implies $|E| \geq k$ and we return TRUE (line 6), as desired.
- A_d has no k-repeat Then $|E| < k$ (otherwise contradicts assumption of no k-repeat). So we skip over all `if` statements and return FALSE.

Induction Step: Assume claim true for calls at depth $d+1$, prove for calls at depth d.

Given list A_d at a non-leaf node, we know that either:

- A_d has a k-repeat.

  If the k-repeat is the median element $A_d[m]$, then $|E| \geq k$ (by definition of k-repeat) and we return TRUE. Otherwise, the k-repeat must be in either L or G. If it is in L, then $|L| \geq k$ and we will recurse (line 9) on L and return TRUE via the inductive assumption. If not in L, we skip over line 10 and instead consider G. If the k-repeat is in G (it must be if not in E or L), then necessarily $|G| \geq k$, which implies we will make the recursive call in line 14, it will evaluate to TRUE (by inductive assumption), and then we will return TRUE (line 15).
- A_d has no k-repeat.

  This case implies $|E| < k$, so we skip over line 6. If we perform the recursive call in line 9, by inductive assumption this will evaluate FALSE (since no k-repeat exists), so we skip over line 10. Similarly, we will never reach line 15. Thus, if A_d has no k-repeat we must necessarily reach line 18 and return FALSE.

`DetectKRepeat` **Terminates**

Claim: The depth of the recursion tree is $O(\log \frac{n}{k})$.

Proof: Let A_0 denote the original list at the root call, with $|A_0| = n$. In the worst case, $|E| \leq 1$ and recursion will be necessary for both L and G. We have upper bounds $|L_0| \leq \frac{n}{2}$ and $|G_0| \leq \frac{n}{2}$ by definition of a partition around a median, so $A_1 \leq \frac{n}{2}$. In general, let A_d be the list at depth d in the recursion tree. In the worst case, $|E| \leq 1$ at all previous depths, and we have upper bound on problem size $|A_d| \leq \frac{n}{2^d}$ (by partition around the median). At depth d, the size of lists L_d and G_d is at most $\frac{n}{2^{d+1}}$.

The recursion tree terminates when both L_d and G_d have size less than k. Solving for d, this yields: $|L_d| < k$ and thus $\frac{n}{2^{d+1}} < k$, which implies $d > \log \frac{n}{k} - 1$. Thus, d is finite (setting it to the nearest integer less than or equal to $\log \frac{n}{k}$ suffices) and termination occurs after at a worst-case depth of $d = O(\log \frac{n}{k})$.

e) **Prove running time of** $O(\ n \log(\frac{n}{k})\ )$ We need to simply add up the worst-case cost at each node in the recursion tree. We know this tree has maximum depth $O(\log \frac{n}{k})$.

At a single call (node) at depth d given list A_d, worst case (non-recursive) cost is as follows:

- (Line 1) Finding median:
  Worst case, this takes $O(|A_d|)$ using the deterministic divide-and-conquer algorithm.
- (Line 2-4) Partitioning into L, E, and G:
  Requires $O(|A_d|)$ comparisons and array insertions. Can be done in a single traversal of the array A_d.
- Remaining lines (excluding recursions at lines 9 and 14)
  Checking the size of E, L, and G should take constant time (worst case linear if we don't efficiently store the size). Comparing these sizes to fixed constants as well as returning appropriate boolean values take constant time.

Summing over all these steps, the (non-recursive) cost of a single call at depth d is $O(|A_d|) = O(\frac{n}{2^d})$.

The number of such calls at depth d is as most 2^d (since tree is binary). So the total cost across all possible nodes at depth d is: $O(\ 2^d|A_d|\ )$, which in terms of n is $O(2^d \frac{n}{2^d}) = O(n)$. The worst-case total cost across all *depths* (using the upper bound on recursion depth proved earlier) is given by

$$\sum_{d=0}^{\lfloor \log \frac{n}{k} \rfloor} O(n) \le O(n(1 + \log \frac{n}{k})\ ) = O(n \log \frac{n}{k}) \tag{7}$$

Thus, the worst case run-time meets the desired $O(\ n \log(\frac{n}{k})\ )$ bound.

Problem 3

a. For this problem, since we are just using the general approach, we will pick the most appropriate pairs. If there are an odd number of elements, then we will keep the extra element in our new array. This is *not* in line with the actual algorithm.

(1) $[X, X, X, X, X, Y] \to [(X, X), (X, X), (X, Y)] \to [X, X] \to [(X, X)]$
$\to [X] \Rightarrow X$ majority element

(2) $[X, Y, X, Y, X, Y] \to [(X, Y), (X, Y), (X, Y)] \to [\ ] \Rightarrow$ no majority element

(3) $[X, Y, X, Y, Y] \to [(X, Y), (X, Y), Y] \to [Y] \Rightarrow Y$ majority element

(4) $[Y, Y, X] \to [Y, (Y, X)] \to [Y] \Rightarrow Y$ majority element

b. **Input:** An zero-indexed array A of n elements (elements that can be checked for equality).

Output: The majority element (the element that is present more than $n/2$ times) if one exists, otherwise const null.

Algorithm: This algorithm has two steps: finding a candidate majority element and then, if one is found, checking the candidate to verify that it is indeed a majority element.

```
FindMajorityElt($A$)
1  return CheckElt(FindCandidate($A$), $A$)

CheckElt($x, A$)
1  return (number of times $x$ in $A$) > (length($A$)/2)

FindCandidate($A$)
 1  if length($A$) = 0
 2      then return const null
 3  if length($A$) = 1
 4      then return $A[0]$
 5  if length($A$) odd
 6      then if CheckElt($A[0], A$)
 7               then return $A[0]$
 8               else  remove $A[0]$ from $A$
    ▷ Now length($A$) is even, so we pair up adjacent elements
 9  $i \leftarrow 0, B \leftarrow [\,]$
10  while $i <$ length($A$)
11        do if $A[i] = A[i+1]$
12               then append $A[i]$ to $B$
13           $i \leftarrow i + 2$
14  return FindCandidate($B$)
```

c. Let A be an array of size n.

Claim: The algorithm produces an element when given $A \Leftrightarrow A$ has a majority element.

Proof: We prove both implications:

($\Rightarrow$) If the algorithm produces an element x when given A, then it checked that x was present more than $n/2$ times in A, so by definition A has majority element x.

($\Leftarrow$) We want to show that if A has majority element x, then the algorithm produces x. This is equivalent to showing that FindCandidate produces x. We will use strong induction on n.

If $n = 1$, then x is the only element, so the algorithm returns it.

Assume our claim is true for all k, $0 < k < n$. We have two cases:

- k odd: If the first element is x, then the algorithm returns it. Otherwise, removing the first element decreases the number of non-x elements, so x is still the majority element in the array and we are reduced to the case that k is even.
- k even: Let c_x be the number of times x is in the array. Thus,

$$c_x > n/2$$

Consider the following definitions:

$p_{x,x}$ = # of pairs considered by the algorithm where both elements are x
$p_{x,y}$ = # of pairs where only one element is x
$p_{y,y}$ = # of pairs where neither element is x, but the elements are equal

$p_{y,z} = \#$ of pairs where neither element is x, and the elements are not equal

Clearly $n = 2(p_{x,x} + p_{x,y} + p_{y,y} + p_{y,z})$. Also, $c_x = 2p_{x,x} + p_{x,y}$. Substituting into our inequality, we see,

$$\begin{aligned} 2p_{x,x} + p_{x,y} &> 2(p_{x,x} + p_{x,y} + p_{y,y} + p_{y,z})/2 \\ p_{x,x} &> p_{y,y} + p_{y,z} \end{aligned}$$

Since $p_{x,x}$ is the number of times x occurs in our new array, and $p_{y,y}$ is the number of times non-x elements occur in our new array, this means that x is the majority element of our new array. Clearly our new array has size less than n, so our recursive call to FindCandidate will produce x by the inductive hypothesis.

Thus, by induction, FindCandidate will produce x.

d. **Claim:** Given an array of size n, the algorithm has a running time of O(n).

Proof: The runtime is the sum of the runtimes of CheckElt and FindCandidate. CheckElt counts the number of times an element is in the array, which takes time O(n), and performs constant time arithmetic operations.

In FindCandidate, if the array has odd length, it calls CheckElt, which takes time O(n), and then in the worst case reduces the array by 1 and proceeds to the even case. In the even case, since at most one of every pair of elements is added to the new array, the recursive call is on an array of at most size $n/2$, which takes time O(n) to construct. The runtime of FindCandidate is thus given by the following recurrence:

$$T(n) = T(n/2) + O(n)$$

which can be solved to give a runtime of O(n).

Thus, the total runtime of the algorithm is O($n + n$) = O(n).

Problem 4

Part (a)

We know there are exactly 3 elements in A that will come before a_4 in $A \cup B$. Furthermore, since $a_4 < b_4$, at most b_1, b_2, b_3 will come before a_4 in $A \cup B$. Therefore, the maximum possible rank of a_4 in $A \cup B$ is 7.

By similar reasoning, b_4 must come after exactly 3 elements in B as well as at least 4 elements in A, since $a_4 < b_4$. Therefore, the minimum possible rank of b_4 in $A \cup B$ is 8.

Example: $A = \{4, 5, 6, 7, 13, 14, 15, 16\}$ and $B = \{1, 2, 3, 8, 9, 10, 11, 12\}$.

Part (b)

Input: 2 lists of numbers A and B given as arrays with 1-based indexing, sorted in increasing order with $a_i \neq b_j \ \forall i, j$, and an integer k between 1 and $|A| + |B|$.
Output: The kth element of $A \cup B$, the sorted union of the two lists of numbers.
High Level: Label the two lists as L and G, with pointers s_L, e_L, s_G, e_G to indicate the parts of L

and G being considered (their active regions). We will find the kth element by iteratively moving the pointers inward until one active region is empty, switching L and G as needed to ensure the median of L's active region is lower than that of G ($L[mid_L] < G[mid_G]$). We then subtract the end pointer for the empty active region e from k, and return the element of that rank ($k - e$) from the other list (the list with a non-empty active region).

```
k-In-Union($A, B, k$)
 1  //We assign $L$ and $G$ arbitrarily, but we will fix $L[mid_L] < G[mid_G]$ in (10).
 2  $L \leftarrow A$; $G \leftarrow B$
 3  $s_L \leftarrow 1$; $s_G \leftarrow 1$
 4  $e_L \leftarrow |L|$; $e_G \leftarrow |G|$
 5  //Loop Invariants:
 6  //Every element after its list's end pointer has rank greater than $k$ in $A \cup B$.
 7  //Every element before its list's start pointer has rank lower than $k$ in $A \cup B$.
 8  while $s_L \leq e_L$ and $s_G \leq e_G$
 9        do $mid_L \leftarrow \lfloor \frac{e_L + s_L}{2} \rfloor$, $mid_G \leftarrow \lfloor \frac{e_G + s_G}{2} \rfloor$
10           if $L[mid_L] > G[mid_G]$
11              then switch $L$ and $G$, $s_L$ and $s_G$, and $e_L$ and $e_G$.
12           //There are now 2 possible cases–in both, we halve one of the lists:
13           if $k < mid_L + mid_G$
14              then $e_G \leftarrow mid_G - 1$ //Cut the last half of $G$'s active region.
15      else if $k \geq mid_L + mid_G$
16              then $s_L \leftarrow mid_L + 1$ //Cut the first half of $L$'s active region.
17  if $s_L \leq e_L$, return $L[k - e_G]$
18  if $s_G \leq e_G$, return $G[k - e_L]$
```

Part (c)

We prove a termination claim and a loop invariant, followed by the proof of correctness.

Terms: The two arrays L and G each have a start pointer s (s_L & s_G) and an end pointer e (e_L & e_G). These pointers bound an active region for each list, and their average ($\lfloor \frac{e+s}{2} \rfloor$) is called mid (mid_L & mid_G). (10-11) maintain that ($L[mid_L] < G[mid_G]$).

Claim 1: The while loop terminates. When it does, the start pointer is one more than the end pointer for the list with an empty active region. That is, $s = e + 1$ for that list.
Proof:
At each iteration, one of the active regions is halved, so eventually an active region will reach size 1, such that $s = e = mid$ for that list. Then $e_G \leftarrow mid_G - 1$ and $s_L \leftarrow mid_L + 1$ will result in $s = e + 1$ whether the list is G or L. Then $s > e$, so that list has an empty active region, and the while loop ends. An active region larger than 1 will not halve to an empty one, so only the case $s = e + 1$ will terminate the loop.

Invariant 1: At every loop iteration, each element with a greater rank than its list's end pointer (e) has a rank greater than k in the union $A \cup B$, and each element with a lower rank than its list's start pointer (s) has a rank lower than k in the union $A \cup B$.
Proof:
Initially we set $s_L = s_G = 1$, $e_L = |L|$ and $e_G = |G|$, so no element's rank is greater than its list's end pointer or lower than its list's start pointer and the invariant holds.
Now, the lists are sorted and we ensure $G[mid_G] > L[mid_L]$, so $G[mid_G]$'s rank is at least $mid_L + mid_G$ and $L[mid_L]$'s rank is at most $mid_L + mid_G - 1$ in the union $A \cup B$.
Since e_L and s_G are never directly changed, it suffices to consider just e_G and s_L.
e_G is only changed when $k < mid_L + mid_G$, so by setting e_G to $mid_G - 1$, all elements after e_G in G (starting with $G[mid_G]$) must have rank greater than k.
Similarly, s_L is only changed when $k \geq mid_L + mid_G$, so setting s_L to $mid_L + 1$ means that all elements before s_L in L have rank lower than k.

Theorem: Upon termination, if e is the end pointer for the list with an empty active region, the kth element of $A \cup B$ is the $(k - e)$th element in the other list.
Proof:
The cases in (17) and (18) are symmetric, so w.l.o.g, suppose L is the list with an empty active region upon termination as in (18)–that is, $s_L > e_L$ and $s_G \leq e_G$.
Then Claim 1 shows that $s_L = e_L + 1$ on termination.
By Invariant 1, all elements before the s_Lth in L have rank lower than k in $A \cup B$, and all elements after the e_Lth in L have rank greater than k in $A \cup B$.
But $s_L - 1 = e_L$, so the first e_L elements come before the kth element of $A \cup B$.
Since every element in L is thus accounted for, we know that *exactly* e_L elements in L come before the kth element of $A \cup B$, and that the kth element must be in G.
Thus, the kth element of $A \cup B$ is the $(k - e_L)$th element of G, which is exactly the element we return at the end of the algorithm! In case (17), we can return the $(k - e_G)$th element of L by the same logic, so in both cases, we return the correct element. QED

Part (d)

Proof that the algorithm runs in $O(\log|A| + \log|B|)$:

Lines (2-4) and (17-18) all run in constant time, because they are just constant work.

Turning to the while loop, then:
Each iteration of the loop runs in constant time, because each line consists only of constant time assignments, lookups, and comparisons.

How many times does the loop run?
In each case (13 and 15), the active region of either G or L is halved. Therefore, in every iteration of the loop either the active region of A or B is halved. At most, there can be $(\log|A|)$ halvings of A and $(\log|B|)$ halvings of B.

Thus, the maximum halvings that can be done, or times the loop can run, is in $O(\log|A| + \log|B|)$. Each of these runs is in $O(1)$, so adding the while loop to the rest of the constant time work, we have $O(\log|A| + \log|B|) + O(1) = O(\log|A| + \log|B|)$. Q.E.D.