Sheetal Parikh
Foundations of Algorithms
EN.605.621.81.SU19
Homework #3

1.) *The following problem is known as the Dutch Flag Problem. In this problem, the task is to rearrange an array of characters R, W, and B (for red, white, and blue, which are the colors of the Dutch national flag) so that all the R's come first, all the W's come next, and all the B's come last. Design a linear, in-place algorithm that solves this problem.*

We want to rearrange the characters of the array A[0...n-1] so that the same colors are next to each other. We can assume R = 0, W = 1, and B = 2. There will be 4 sections of the array and we will use pointers Left(l), Middle(m) and Right(r).

Section Red	= A[0...l-1] and contains the Red characters or 0s
Section White	= A[0...m-1] and contains the White characters or 1s
Section Unknown	= A[m...r] and contains unknown characters
Section Blue	= A[r+1...n-1] and contains Blue characters or 2s

While determining the unknown characters in the unknown section, we need to maintain the order of the other sections. Similar to quicksort, we will check the middle element. We increment the pointers depending on the element.

3 Situations:

1.) If the element is a 0, we swap the current middle element with the left element. We also increment the middle pointer by 1 and the left pointer by 1.
2.) If the element is a 1, we increment the middle pointer one space to the right.
3.) If the element is a 2, we swap the current middle element with the right element. We also decrement the right pointer by 1.

The algorithm will continue until the right and middle pointer are in the same position.

Pseudocode:

```
Function DutchFlag(A[0...n-1])
    Initialize l = 0
    Initialize m = 0
    Initialize r = n-1

    while(m ≤ r) do
        if A[m] = 0 then
            swap(A[m],A[l])
            l = l+1
            m = m+1
        elseif A[m] = 1 then
            m = m +1
        else A[m] = 2
            swap(A[m],A[r])
```

```
                r = r - 1
            end if
    end while
return A
```

2.) *Give an O(n lg k)-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hints: Use a heap for k-way merging.)*

If we were merging 2 sorted lists into one sorted list, we would be able to pick the smallest element in each list having *k-1* comparisons per element. However, with *k* sorted lists, it would be more difficult to compare the *k* lists to determine the minimum.

Therefore, we can use a min-heap to help merge the *k* sorted lists. Using the similar idea of merging we used for sorting 2 lists, we will take the first element of each list (presumably the smallest element of each list) and place it in a min-heap. We will extract the smallest value from the min-heap and then insert the next smallest element from the corresponding list. The extraction will consist of swapping the root element with rightmost leaf element on the lowest level of the heap. The size of the heap will reduce by one and then the new heap must heapify downwards. Insertion will consist of inserting the element in leftmost lowest position of the heap. The new element will swap with the smaller elements until heap order is established or heapify upwards. The process of extracting from heap and inserting to the heap will continue until the *k* sorted lists are empty and all the elements have been sorted.

Building the heap will take O(*k*) time since we have *k* sorted lists. The insertion and extraction from the heap will both take O(*nlogk*) time. Extraction and deletion, as discussed in our textbook, require logk time because every time we insert or extract the root must bubble down the size of the heap. Since we are given that there are n elements in all the input lists, the insertion and extraction would both take O(*nlogk*) time. Therefore the worst case time for the algorithm would be O(*nlogk*).

3.) *parts (a) through (h)*

(a) *Argue that A[q] > A[p] so that B[p] = 0 and B[q] = 1.*

We are given that A[*p*] is the smallest value in A that algorithm X puts in the wrong location and that A[*q*] is the value that is put in the wrong position. Therefore, A[q] must be a value greater than A[*p*]. Furthermore, we also know that B[*i*] = 0 if A[*i*] ≤ A[*p*] and B[*i*] = 1 if A[*i*] > A[*p*]. Since we just established that A[*q*] > A[*p*], we can say that B[*q*] = 1. Also, since A[*p*] is equal to itself, we can say that B[*p*] = 0.

(b) *To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to sort the array B correctly.*

In part a, we proved that if A[*q*] > A[*p*] then B[*p*] = 0 and B[*q*] = 1 so overall B[*q*] > B[*p*]. Therefore we know that if array A is not sorted properly that array B will have a dirty area of 1s and 0s and will also not be sorted. Overall algorithm X does not change how it operates based on the values in the array. If A[*p*] is in the wrong location in the output of A, then we know algorithm X failed to compare it to other elements in the array properly such as A[*q*]. Algorithm

X will operate similarly for array B and so if algorithm X failed to sort array A properly, it will also fail to sort array B properly.

(c) *Argue that we can treat columnsort as an oblivious compare-exchange algorithm, even if we do not know what sorting method the odd steps use.*

As proven in parts a and b and given in the problem description, we know that the 0-1 Lemma states that if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values. The even step of column sort performs all the shifts, transpositions, and inverse operations on the array. The odd steps of column sort will sort the data in each column using an unknown sorting method. Even though the method is unknown, we know that if overall columnsort sorts the array column wise, then the oblivious compare-exchange algorithm would produce a clean area of only 1s or 0s. Therefore, regardless of the values of the array and the sorting method, we will have the same result as if we had used the oblivious compare-exchange algorithm. Therefore we can treat columns sort as an oblivious compare-exchange algorithm.

(d) *Prove that after the first three steps, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most s dirty rows between them.*

We are given that the input array contains only 0s and 1s as well as *r* rows and *s* columns. Also, we know that the even steps performs all the shifts, transpositions, and inverse operations on the array whereas the odd steps sort each column.

In the first step, which is odd, each column will be sorted with all 0s at the top and all 1s at the bottom. Therefore, each column will have 0s followed by 1s and so there will be only one switch in each column from 0s to 1s. Because, the elements are only sorted within each column and not the entire array, each column will have a different location for the 0 to 1 switch.

In the second step, which is even, the array is transposed but reshaped back to *r* rows and *s* columns. For example, the leftmost column will become the top r/s rows and the rightmost column will become the bottom r/s rows. Since in the first step each column has a 0 to 1 switch at different positions, each row after the second step will either be clean, will have at most one switch from 0 to 1 or will have at most one switch from 1 to 0. The switch from 1 to 0 can occur from switching from one old column to another. For example the switch from 1 to 0 can occur from a series of 1s at the end of a row and a series of 0s beginning in another row. Due to the transposition, the end of the old columns or now rows *r/s* will be dirty. As we have *s* groups of these rows in total, we can have at most *s* dirty rows.

Lastly in the third step, which is odd, each column is sorted. We will have 0s at the top and 1s at the bottom. The s dirty rows from step 2 will now be in the middle between the clean 0s at the top and clean 1s at the bottom.

(e) *Prove that after the fourth step the array, read in column-major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most s^2 elements in the middle.*

We saw from part d, that we will have a dirty block of *s* rows and we already have *s* columns. Therefore we would have at most an area of s^2 elements. Step 4, which is an even step, will shift the clean 0s and 1s to the right leaving a dirty block of elements in the middle.

(f) *Prove that steps five through eight produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.*

We are given in columnsort that the array must satisfy r ≤ $2s^2$ where *r* is the number of rows and *s* is the number of columns. This equation can also viewed $\frac{r}{2} \leq s^2$ if we solve for s^2 . As per part e of the problem, we found that the array has a dirty area of at most s^2 elements in the middle. Therefore, if the dirty area has at most s^2 elements, then the dirty area also has a size of at most $\frac{r}{2}$ elements or half a column in the array. If we have a dirty area of at most $\frac{r}{2}$ elements then either the dirty elements will be all in one column or will be split over two columns. When the dirty elements are all in one column, then step 5, an odd step, will sort the column and the following steps 6 to 8 will eventually sort the entire array. When the dirty elements are split over two columns, the dirty elements will be held at the bottom of a column and the top of the next column. Step 5 will sort the column and step 6 will move each column down $\frac{r}{2}$ or half a column. This will move the dirty portions of the two columns all into one column. After the dirty elements are all in one column then step 7 will sort the columns and step 8 will put all columns back into the correct positions producing a sorted 0-1 output. Overall, columnsort correctly sorts all inputs of the array which contained arbitrary values.

(g) *Now suppose that s does not divide r. Prove that after steps one through three, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most 2s − 1 dirty rows between them. How large must r be, compared with s, for columnsort to correctly sort when s does not divide r.*

As given by the problem, if s does not divide r, one of the array restrictions for columns sort, then we will have at most $2s-1$ dirty rows between clean rows of 0s at the top and clean rows of 1s at the bottom after steps 1 through 3. Because we know we have *s* columns, the area of the dirty elements becomes $(2s-1)s$ or $2s^2 - s$. In part f we proved that the area of the dirty elements would be a size of at most s^2 *or* $\frac{r}{2}$. To maintain that assumption and determine how large *r* must be, we can set $2s^2 - s \leq \frac{r}{2}$ and solve for *r*. We will get $r \geq 2(2s^2 - s)$ which simplifies to $r \geq 4s^2 - 2s$. Therefore, *r* must be a size of at least $4s^2 - 2s$ for columnsort to correctly sort when *s* does not divide *r*.

(h) *Suggest a simple change to step one that allows us to maintain the requirement that r ≥ ws^2 even when s does not divide r, and prove that with your change, columnsort correctly sorts.*

A simple change that could be made to maintain the requirement that *r* ≥ *w*s^2 even when *s* does not divide *r*, would be to sort every other column in reverse order in step 1. When *s* does divide *r,* then we would have no more than one switch between 0s and 1s or 1s and 0s in step 2 ensuring that no new dirty rows are created in the following steps. However, when *s* does not divide *r,* we could have *2s-1* dirty rows meaning we could be creating new dirty rows after step 2 due to more switches between a column. If this occurs, we would need more steps to sort the array. By sorting every other column in reverse order in step 1, we would reduce the number of switches in the rows after step 2 because a switch not occur when we move to a new column unless a column had all 0s or 1s. This would limit the dirty rows to be at most *s* and the remaining steps of the algorithm would proceed normally (as if *s* did divide *r*).

4.) *Consider the following algorithm for doing a postorder traversal of a binary tree with root vertex root. Prove that this algorithm run in time Θ(n) when the input is an n-vertex binary tree.*

The postorder traversal of a binary tree is similar to an inorder traversal in that all nodes of the subtree are visited and that the postorder traversal prints the root after the values in its subtree (rather than printing between the values of the left subtree and right subtree). Similar to the proof of the in order subtree on page 288 of textbook, let T(n) denote the time taken by the postorder traversal when it is called on an n-vertex subtree. Because the postorder traversal visits all n vertices of the subtree, we have T(n) = Ω(n). However, we need to show that the algorithm runs in time Θ(n).

As given by the problem, we have n vertices. If we are at a vertice *x* on the left side of the subtree with *k* vertices, then the right subtree has *n-k-1* vertices. Therefore the time to perform the postorder traversal at vertice *x* is bounded by $T(n) \leq T(k) + T(n-k-1) + d$, for some constant d > 0 and exclusive of any recursive calls.

We can consider two cases, case 1 in which we have an empty subtree and case 2 in which we have an equal number of vertices for both subtrees. If we have an empty subtree, then we would only visit the vertice once. For case 2, if both subtree's are of equal size than we can say k = $\frac{n}{2}$ and we have:

$$\begin{aligned} T(n) &\leq T(\tfrac{n}{2}) + T(n - \tfrac{n}{2} - 1) + d \\ &\leq T(\tfrac{n}{2}) + T(\tfrac{n}{2} - 1) + d \\ &\leq T(\tfrac{n}{2}) + T(\tfrac{n}{2}) - T(1) + d \end{aligned}$$

If we ignore the constants we get $T(n) \leq 2T(\frac{n}{2})$ which resembles the format needed to use the master's theorem. Using the master's theorem we have the following steps:

Step 1: a = 2
b = 2
f(n) = 0

Step 2: $n^{\log_b a} = n^{\log_2 2} = n^1 = n$

Step 3: n = n
f(n) = 0
f(n) < n

Step 4: since f(n) < n, T(n) is Case 1
Therefore, f(n) = $O(n^{1-\varepsilon})$ and T(n) = $\Theta(n^{\log_b a}) = n^{\log_2 2}$ = Θ(n)

Therefore, we can say that the post order traversal algorithm runs in time Θ(n).

5.) *We define an AVL binary search tree to be a tree with the binary search tree property where, for each node in the tree, the height of its children differs by no more than 1. For this problem, assume we have a* *team of biologists that keep information about DNA sequences in an AVL binary search tree using the specific weight (an integer) of the structure as the key. The biologists routinely ask questions of the type, “Are there any structures in the tree with specific weight between a and b, inclusive?” and they hope to get an answer as soon as possible. Design an efficient algorithm that, given integers a and b, returns true if there exists a key x in the tree such that a ≤ x ≤ b, and false if no such tree exists in the tree. Describe your algorithm in pseudocode and English. What is the time complexity of your algorithm? Explain.*

Pseudocode:

```
Function AVLWeightSearch(Node, a, b)  //the function will accept a node, integer a, integer b
    x = Node                          // assigning the node to the key variable x
    if x = null then                  // checking whether then node exists
        return false                  // if the node does not exist, return false
    end if
    if x ≥ a and x ≤ b then           //returning true if the node is between a and b
        return true
    elseif x < a then                 // if node less than a, we move right
        return AVLWeightSearch(right-x, a, b)
    else x > b then             // if node greater than b, we move left
        return AVLWeightSearch(left-x, a, b)
    endif
```

Overall the algorithm assigns the node to the key variable *x*. If the node does not exist than false is returned. Next, we check whether the node is between a and b (*a ≤ x ≤ b).* If this is satisfied, then true is returned. However, if the node is not between a and b, we check whether the node is less than a. If this is satisfied, then we move to search the right side of the current node because the values towards the right subtree would be greater than the current node. However, if the node is not less than a, we check whether x is greater than b. If this is the case, we move to search the left of the current node because the values towards the left subtree would be smaller than the current node. The algorithm will continue until there are no longer any nodes remaining or if we find a node between a and b.

The time complexity of the algorithms would be O(logn). Until we find a match, at every step the number of nodes we need to check is reduced in half. This would produce a time complexity of O(logn).