Name: Mauro Vargas Jr

ID: 107212593

CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Homework 2B (55 points) **Summer 2020, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Verbal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solution:

- The solutions **should be typed**, we cannot accept hand-written solutions. Here's a short intro to **Latex**.
- In this homework we denote the asymptomatic *Big-O* notation by $\mathcal{O}$ and *Small-O* notation is represented as o.
- We recommend using online Latex editor **Overleaf**. Download the **.tex** file from Canvas and upload it on overleaf to edit.
- You should submit your work through **Gradescope** only.
- If you don't have an account on it, sign up for one using your CU email. You should have gotten an email to sign up. If your name based CU email doesn't work, try the identikey@colorado.edu version.
- Gradescope will only accept **.pdf** files (except for code files that should be submitted separately on Canvas if a problem set has them) and **try to fit your work in the box provided**.
- You cannot submit a pdf which has less pages than what we provided you as Gradescope won't allow it.

Name: Mauro Vargas Jr

ID: 107212593

CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Homework 2B (55 points) **Summer 2020, CU-Boulder**

Piazza threads for hints and further discussion

Piazza Threads
Question 1
Question 2
Question 3
Question 4

Recommended reading
Divide Conquer; Recurrence Relations: Ch. 2 2.3; Ch. 4 4.1, 4.2, 4.3, 4.4, 4.5
Quicksort: Chapter 7 complete
Efficient Data Structures: Hash Tables: Ch. 11 Complete

Name: Mauro Vargas Jr

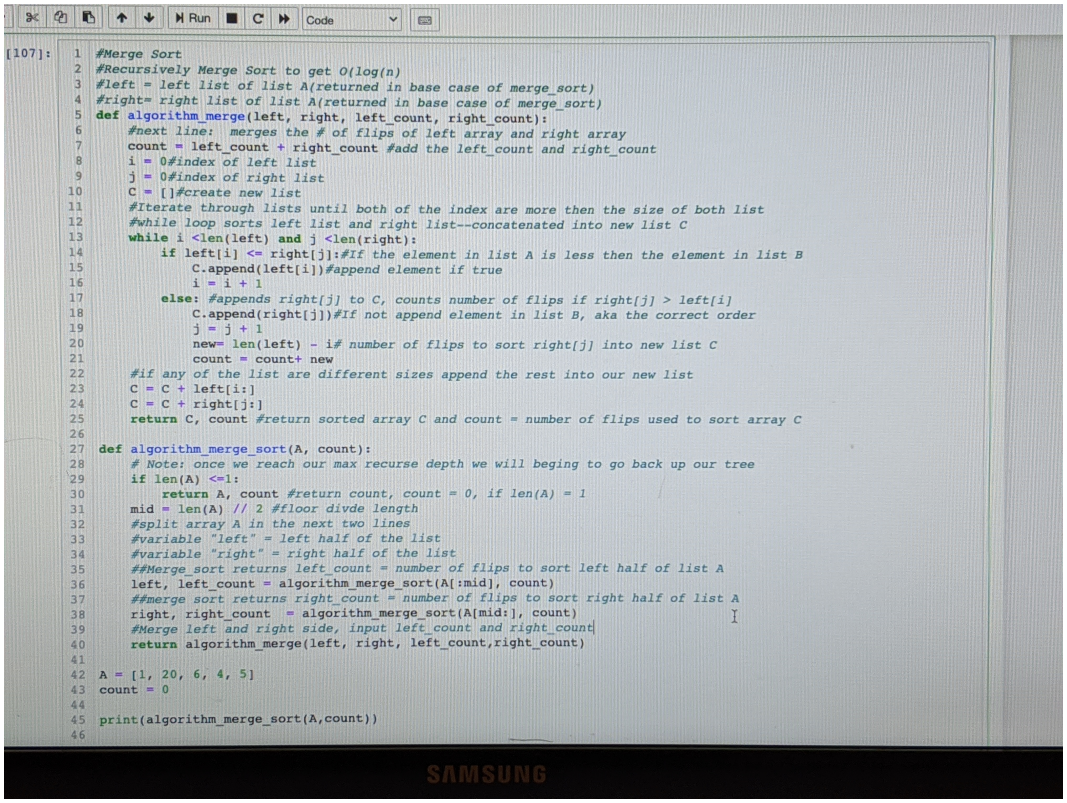
ID: 107212593

CSCI 3104, Algorithms **Escobedo & Jahagirdar**

Homework 2B (55 points) **Summer 2020, CU-Boulder**

1. *(20 pts) Let $A = \langle a_1, a_2, \ldots, a_n \rangle$ be an array of numbers. Let's define a 'flip' as a pair of distinct indices $i, j \in \{1, 2, \ldots, n\}$ such that $i < j$ but $a_i > a_j$. That is, a_i and a_j are out of order.*
For example - In the array A = [1, 3, 5, 2, 4, 6], (3, 2), (5, 2) and (5, 4) are the only flips i.e. the total number of flips is 3. (Note that in this example the indices are the same as the actual values)

   Design a divide-and-conquer algorithm with a runtime of $\mathcal{O}(n \log(n))$ for computing the number of flips. Your algorithm has to be a divide and conquer algorithm that is modified from the Merge Sort algorithm. Explain how your algorithm works, including pseudocode. You can add a picture of your pseudo-code or properly commented code.



```
[107]:
#Merge Sort
#Recursively Merge Sort to get O(log(n)
#left = left list of list A(returned in base case of merge_sort)
#right= right list of list A(returned in base case of merge_sort)
def algorithm_merge(left, right, left_count, right_count):
    #next line:  merges the # of flips of left array and right array
    count = left_count + right_count #add the left_count and right_count
    i = 0#index of left list
    j = 0#index of right list
    C = []#create new list
    #Iterate through lists until both of the index are more then the size of both list
    #while loop sorts left list and right list--concatenated into new list C
    while i <len(left) and j <len(right):
        if left[i] <= right[j]:#If the element in list A is less then the element in list B
            C.append(left[i])#append element if true
            i = i + 1
        else: #appends right[j] to C, counts number of flips if right[j] > left[i]
            C.append(right[j])#If not append element in list B, aka the correct order
            j = j + 1
            new= len(left) - i# number of flips to sort right[j] into new list C
            count = count+ new
    #if any of the list are different sizes append the rest into our new list
    C = C + left[i:]
    C = C + right[j:]
    return C, count #return sorted array C and count = number of flips used to sort array C

def algorithm_merge_sort(A, count):
    # Note: once we reach our max recurse depth we will beging to go back up our tree
    if len(A) <=1:
        return A, count #return count, count = 0, if len(A) = 1
    mid = len(A) // 2 #floor divde length
    #split array A in the next two lines
    #variable "left" = left half of the list
    #variable "right" = right half of the list
    ##Merge_sort returns left_count = number of flips to sort left half of list A
    left, left_count = algorithm_merge_sort(A[:mid], count)
    ##merge sort returns right_count = number of flips to sort right half of list A
    right, right_count  = algorithm_merge_sort(A[mid:], count)
    #Merge left and right side, input left_count and right_count
    return algorithm_merge(left, right, left_count,right_count)

A = [1, 20, 6, 4, 5]
count = 0

print(algorithm_merge_sort(A,count))
```

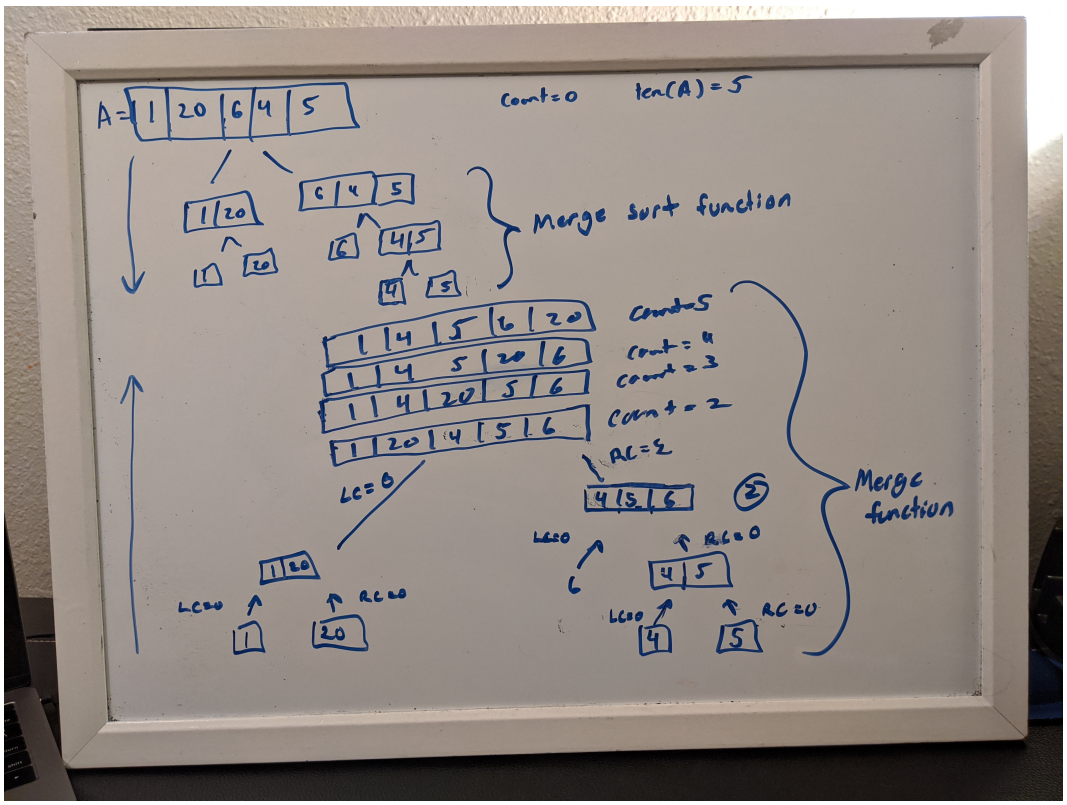
My merge sort algorithm is broken into two functions merge sort and merge. Merge sort is done recursively to break up the list and merge is used to actually merge the separate list and compute the number of flips and put them in ascending order.

Name: Mauro Vargas Jr

ID: 107212593

CSCI 3104, Algorithms **Escobedo & Jahagirdar**

Homework 2B (55 points) **Summer 2020, CU-Boulder**



After the tree hits our base case on line 29, will recurse back up our tree and start to compute our flips.

Name: Mauro Vargas Jr

ID: 107212593

CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Homework 2B (55 points) **Summer 2020, CU-Boulder**

2. *(10 pts) For the following problems, you must use the pseudocode for QuickSort and Partition in Section 7.1 of the Introduction to Algorithms 3rd edition (CLRS) and the array A = [22, 40, 67, 55, 10, 92, 66].*

   (a) *(3 pts) What is the value of the pivot in the call Partition(A; 1; 7)?* (Array indexing starts at 1 as per the convention in the book)

Inside PARTITION(A,p,r)
we have **x** = A[r]
We can see that the function is passing in list $A[p...r]$. The last index in list A is 7 which represents our **r**, therefore the value in x is 66.

(b) *(3 pts) What is the index of that pivot value at the end of that call to Partition?*

In line 8, x = A[r] represents the pivot. In the first call to Partition, the algorithm iterates through all the elements in A and checks if each element ¡= x. If element ¡= x is true, then element is sorted to the left of the element. The algorithm continuously sorts elements less than or equal to x (pivot) to the left and swaps x at the end so elements greater than x are sorted to the right side of x in A. The index of the pivot value at the end of that call to Partition is 4. The resulting array to the first Partition call is A = [22, 40, 55, 10, 66, 92, 67] 66 is the current pivot.

(c) *(4 pts) On the next recursive call to Quicksort, what sub-array does Partition evaluate? (Give the indices specifying the subarray.)?*

In line 4, the next recursive call to quick sort inputs sub-array A[p . . . q-1].
In other words, the sub-array that Partition evaluates is all elements of A that are to the left of the pivot.
For this example, the subarray = [22, 40, 55, 10]

Name: Mauro Vargas Jr

ID: 107212593

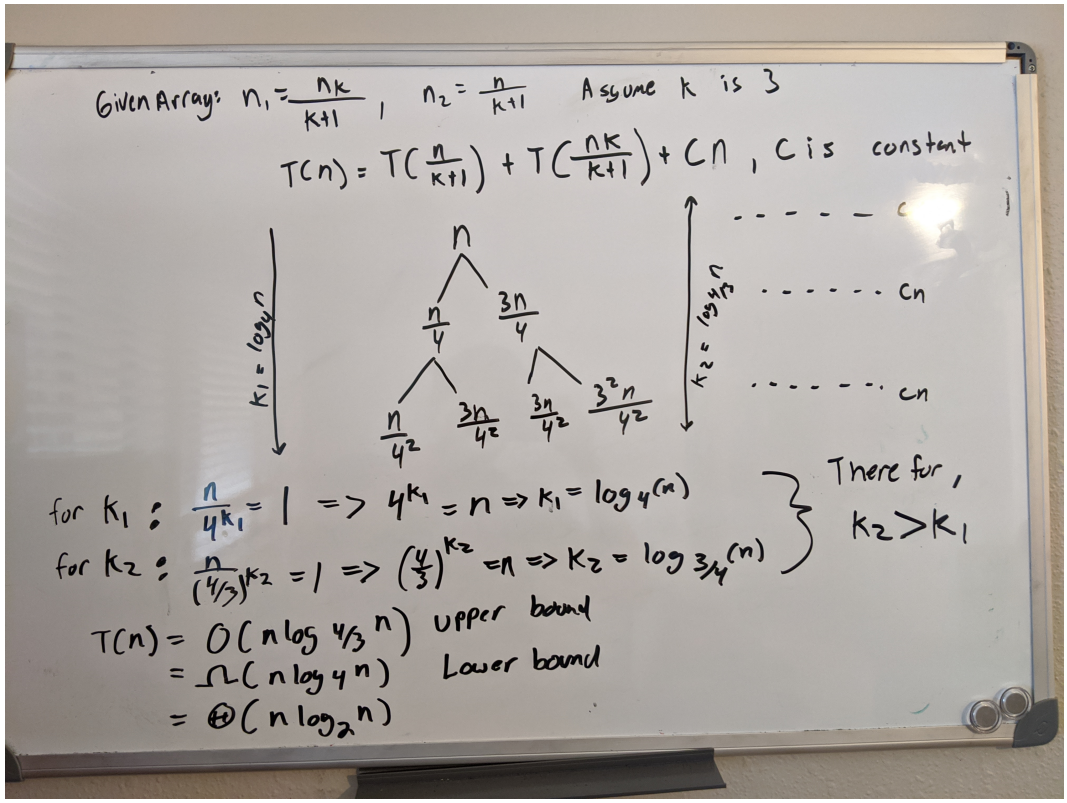
CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Homework 2B (55 points) **Summer 2020, CU-Boulder**

3. *(15 pts) Suppose in quicksort, we have access to an algorithm which chooses a pivot such that, the ratio of the size of the two subarrays divided by the pivot is a* ***constant*** *k. i.e an array of size n is divided into two arrays, the first array is of size $n_1 = \frac{nk}{k+1}$ and the second array is of size $n_2 = \frac{n}{k+1}$ so that the ratio $\frac{n_1}{n_2} = k$ a constant.*

   (a) *(3 pts) Given an array, what value of k will result in the best partitioning?*

The best partition is when the array will be divided into two even sub arrays. Therefore our best partitioning is when we get $\mathbf{k} = 1$

(b) *(9 pts) Write down a recurrence relation for this version of QuickSort, and solve it asymptotically using* ***recurstion tree*** *method. For this part of the question assume $k = 3$. Show your work, write down the first few levels of the tree, identify the pattern and solve. Assume that the time it takes to find the pivot is $\Theta(n)$ for lists of length n.*



(c) *(3 pts) Provide a verbal explanation of how this Partition algorithm affects the running time of QuickSort, in comparison with the case where the best possible pivot is always used. Does the value of k affect the running time?*

The Partition algorithm affects the running time greatly. The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n - 1$ elements and one with 0 elements. If the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. If the tree is equally balanced for at every level of the recursion, we get a asymptotically faster algorithm and the recurrence has a solution of $T(n) = \Theta(nlog(n))$.

Name: Mauro Vargas Jr

ID: 107212593

CSCI 3104, Algorithms **Escobedo & Jahagirdar**

Homework 2B (55 points) **Summer 2020, CU-Boulder**

4. *(10 pts) Consider the hash function* $h(k) = \lfloor 100k \rfloor$ *for all keys* k *for a table of size 100. You have three applications.*

   - ***Application 1****: Keys are generated uniformly at random from the interval* $[0.3, 0.8]$.
   - ***Application 2****: Keys are generated uniformly at random from the interval* $[0.1, 0.4] \cup [0.6, 0.9]$.
   - ***Application 3****: Keys are generated uniformly at random from the interval* $[0, 1]$.

   (a) *(2 pts) For which application does the hash function* $h(k)$ *perform worse? Please explain/justify adequately your answer.*

   Application 1 has the worse performance since it has the smallest range out of the three application with a $\alpha = 50$. Given by using $\alpha = \frac{n}{l}$ and $h(k) = \lfloor 100k \rfloor$.

(b) *(3 pts) In each of the three applications does the hash function satisfy the uniform hashing property? Please explain/justify adequately your answer.*

Simple uniform hashing is only satisfied, if we know that the keys are random real numbers k independently and uniformly distributed in the range $0 <= k < 1$. None of these applications meet the requirements only application 3 meets our second case, but not the first one of knowing that the keys are generated randomly.

(c) *(5 pts) Suppose you have n keys in total for each application. What is the resulting load factor α for each application? Assume that in each application only slots of the hash table that have a chance of being filled are considered when calculating α.*

$Application1 : \alpha = \frac{n}{50}$
$Application2 : \alpha = \frac{n}{60}$
$Application3 : \alpha = \frac{n}{100}$

Name: Mauro Vargas Jr
ID: 107212593

5. ***Extra Credit (5% of total homework grade)*** *For this extra credit question, please refer the leetcode link provided below or click here. Multiple solutions exist to this question ranging from brute force to the most optimal one. Points will be provided based on Time and Space Complexities relative to that of the most optimal solution.*

   Note*: The brute force approach in this question would be to create an array of size* 10^6 *and directly access the key using the index of the array. For this question, we expect you to perform collision handling once you obtain a hash value for a particular key.*

   Please provide your solution with proper comments which carries points as well.

   https://leetcode.com/problems/design-hashmap/

   ```
   Replace this text with your source code inside of the .tex document
   ```