CS110 Assignment 1

September 30, 2018

1/ (#sort) Implement three-way merge sort in Python. It should at a minimum accept lists of integers as input.

General idea: 3-way merge sort is a recursive algorithm that continually splits a list in 3 parts (left, middle, and right). If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on the 3 sublists. Once the 3 sublists are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking 3 smaller sorted lists and combining them together into a single, sorted, new list (the combine phase).

```
In [123]: # define the function for 3-way merge sort
          # call it merge_sort_3
          # the list takes 1 input: a list need to be sorted
          step = 0
          def merge_sort_3(alist):
              global step
              # If the length of the alist is 0 or 1
              ### then we can consider it is sorted
              ### then the "return" at the end of the code will return the
              ### sorted list, which is the list of 0 or 1 given
              # these lists are the base case for merge sort

              # Hence, we only consider the list with more than 1 element
              if len(alist) > 1:
                  step += 1
                  # As this is the 3-way merge sort. We will divide the initial
                  ### list into 3 sublists using following:
                  # The first boundery is len(alist)//3: we use //: the floor
                  ### division in Python. We use floor division to make sure it is
                  ### an interger as the location must be an integer.
                  # And we divide the list into 3 parts, so divide by 3
                  # mid_1 is the left part
                  mid_1 = len(alist)//3
                  step += 1
                  # The second boundery is (2*len(alist)//3): we use //: the floor
                  ### division in Python. We use floor division to make sure it is
                  ### an interger as the location must be an integer.
                  # And we divide the list into 3 parts, so divide by 3
# mid_2 is the middle part
mid_2 = (2*len(alist))//3
step += 1
# There are 2 more bounderies which are the beginning and
### the end of the list

# The 1st part is the list from the beginning to the mid_1 (Excluded)
left_third = alist[:mid_1]
step += 1
# The 2nd part is the list from mid_1(Included) to the mid_2 (Excluded)
mid_third = alist[mid_1:(mid_2)]
step += 1
# The 3rd part is the list from mid_2(Included) to the end of the list
right_third = alist[mid_2:]
step += 1

# As we divide and conquer
# We divide the list into 3 sublists above
# This is sthe recursion in the algorithm
# Now we conquer each sublist through the following codes:
merge_sort_3(left_third)
step += 1
merge_sort_3(mid_third)
step += 1
merge_sort_3(right_third)
step += 1

# After conquer, we need to implement the final step: combine
# i, j, k are the tracking numbers for 3 sublists
i = 0
step += 1
j = 0
step += 1
k = 0
step += 1
# l is the tracking number for the list
l = 0
step += 1

# Now the comparison -> compare and then combine into a single
### sorted list

# This while loop ensure that there is no sublists has appended
### all its element
while i < len(left_third) and j < len(mid_third) and k < len(right_third):
    step += 1
    # We will have the current_min variables to be the smallest
    # using current_min, we will find the smallest number between
```

```
### the 3 sublists, to merge into the big list

# Find the smallest between the first element of sublist left and mid
if left_third[i] < mid_third[j]:
    step += 1
    current_min = left_third[i]
    step += 1
    # list tracking is to see which sublist has the smallest element
    # we temporarily store the value list_tracking as 0 as
    ### the left_third has the smallest value
    list_tracking = 0
    step += 1
else:
    step += 1
    current_min = mid_third[j]
    step += 1
    list_tracking = 1
    step += 1
    # we temporarily store the value list_tracking as 1 as
    ### the mid_third has the smallest value
if current_min > right_third[k]:
    step += 1
    # list tracking continues to see which sublist has the smallest elem
    # As the right_third now has the smallest value
    # We store the list_tracking to be 3
    # Else, the current_min is the min of all 3 lists
    # And we don't need to change the list_trackking and current_min
    current_min = right_third[k]
    step += 1
    list_tracking = 2
    step += 1

# if the list has the smallest element is the left_third:
if list_tracking == 0:
    step += 1
    alist[l] = left_third[i]
    step += 1
    i += 1
    step += 1
    l += 1
    step += 1
# if the list has the smallest element is the mid_third:
elif list_tracking == 1:
    step += 1
    alist[l] = mid_third[j]
    step += 1
    j += 1
    step += 1
```

```
        l += 1
        step += 1
    # if the list has the smallest element is the right_third:
    else:
        alist[l] = right_third[k]
        step += 1
        k += 1
        step += 1
        l += 1
        step += 1

# As the previous while loop ends when 1 list ran out of element
# We have 3 chances:
# right_third ran out of element
while i < len(left_third) and j < len(mid_third):
    step += 1
    # begin to compare and merge, similar to merge sort 2 ways
    if left_third[i] < mid_third[j]:
        step += 1
        alist[l] = left_third[i]
        step += 1
        i += 1
        step += 1
    else:
        alist[l] = mid_third[j]
        step += 1
        j += 1
        step += 1
    l += 1
    step += 1
# mid_third ran out of element
while i < len(left_third) and k < len(right_third):
    step += 1
    # begin to compare and merge, similar to merge sort 2 ways
    if left_third[i] < right_third[k]:
        step += 1
        alist[l] = left_third[i]
        step += 1
        i += 1
        step += 1
    else:
        alist[l] = right_third[k]
        step += 1
        k += 1
        step += 1
    l += 1
    step += 1
# left_third ran out of element
```

```
while j < len(mid_third) and k < len(right_third):
    step += 1
    # begin to compare and merge, similar to merge sort 2 ways
    if mid_third[j] < right_third[k]:
        step += 1
        alist[l] = mid_third[j]
        step += 1
        j += 1
        step += 1
    else:
        alist[l] = right_third[k]
        step += 1
        k += 1
        step += 1
    l += 1
    step += 1


# As the previous while loop ends when 1 more list ran out of element
# Hence, 2 lists have ran out of element to merge
# We have 3 chances:
# With each type of remainding list, we append everything into the big list
# left_third still have elements
while i < len(left_third):
    step += 1
    alist[l:(l + len(left_third)-i)] = left_third[i:]
    step += 1
    l += len(left_third)-i
    step += 1
    i = len(left_third)
    step += 1
# mid_third still have elements
while j < len(mid_third):
    step += 1
    alist[l:(l + len(mid_third)-j)] = mid_third[j:]
    step += 1
    l += len(mid_third)-j
    step += 1
    j = len(mid_third)
    step += 1
# right_third still have elements
while k < len(right_third):
    step += 1
    alist[l:(l + len(right_third)-k)] = right_third[k:]
    step += 1
    l += len(right_third)-k
    step += 1
    k = len(right_third)
```

```
                step += 1

            return alist


        # import a library to compute a randomize list

        import numpy as np
        alist = list(np.random.permutation(100))
        print('Input:', alist)
        print("")
        print('Output:',merge_sort_3(alist))
        print("")
        print('Steps: ', step)
```

```
Input: [16, 55, 58, 39, 15, 50, 85, 61, 59, 4, 87, 32, 67, 2, 60, 41, 78, 92, 99, 25, 47, 70,

Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,

Steps:  3799
```

2/ (#sort) Implement a second version of three-way merge sort that calls insertion sort when sublists are below a certain length (of your choice) rather than continuing the subdivision process.

General idea: 3-way merge sort + insertion sort is a recursive algorithm that continually splits a list in 3 parts (left, middle, and right) if the length of the sublist > a predetermined threshold (called "k"). If the length of the sublist < a predetermined threshold (called "k"), we will use insertion sort. If the list is empty or has one item, it is sorted by definition (the base case). Once the sublists are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking smaller sorted lists and combining them together into a single, sorted, new list (the combine phase).

Note: We are curious which value of the threshold will yield the best running time and least steps. The data visualization will be conducted after the code.

```
In [141]: # define the function for 3-way merge sort
          # call it merge_sort_3_insertion
          # the function takes the input: a list and a threshole
          # the threshole: calls insertion sort when sublists are below a certain length
          ### , which is the threshole (of your choice) rather than continuing the subdivision
          step = 0
          def merge_sort_3_insertion(alist, threshole):
              global step
              # If the length of the alist is 0 or 1
              ### then we can consider it is sorted
              ### then the "return" at the end of the code will return the
              ### sorted list, which is the list of 0 or 1 given
              # these lists are the base case for merge sort
```

```
        # Hence, we only consider the list with more than 1 element
        if len(alist) > 1:
            step += 1
            # As this is the 3-way merge sort. We will divide the initial
            ### list into 3 sublists using following:
            # The first boundery is len(alist)//3: we use //: the floor
            ### division in Python. We use floor division to make sure it is
            ### an interger as the location must be an integer.
            # And we divide the list into 3 parts, so divide by 3
            # mid_1 is the left part
            mid_1 = len(alist)//3
            step += 1
            # The second boundery is (2*len(alist)//3): we use //: the floor
            ### division in Python. We use floor division to make sure it is
            ### an interger as the location must be an integer.
            # And we divide the list into 3 parts, so divide by 3
            # mid_2 is the middle part
            mid_2 = (2*len(alist))//3
            step += 1
            # There are 2 more bounderies which are the beginning and
            ### the end of the list

            # The 1st part is the list from the beginning to the mid_1 (Excluded)
            left_third = alist[:mid_1]
            step += 1
            # The 2nd part is the list from mid_1(Included) to the mid_2 (Excluded)
            mid_third = alist[mid_1:(mid_2)]
            step += 1
            # The 3rd part is the list from mid_2(Included) to the end of the list
            right_third = alist[mid_2:]
            step += 1

    # Application of insertion Sort starts here

    # Algorithm for insertion sort
            def insertion_sort(input_list):
                global step
                # It always maintains a sorted sublist in the lower positions of the lis
                # Each new item is then inserted back into the previous sublist such
                ### that the sorted sublist is one item larger.
                # We begin by assuming that a list with one item
                ### (position 0) is already sorted
                # Hence, we start with 1 going to len(input_list)
                for index in range(1,len(input_list)):
                    step += 1
                    # The current_value variable is new inserted item
                    current_value = input_list[index]
                    step += 1
            position = index
            step += 1
            #the current item is checked against those in the already sorted sub
            # As we look back into the already sorted sublist, we shift those it
            ### that are greater to the right. When we reach a smaller item or t
            ### of the sublist, the current item can be inserted.
            while position > 0 and input_list[position - 1] > current_value:
                step += 1
                input_list[position] = input_list[position-1]
                step += 1
                position = position - 1
                step += 1
            input_list[position]=current_value
            step += 1
        return input_list
    # As we divide and conquer
    # We divide the list into 3 sublists above
    # Now we conquer each sublist through the following codes:
    # This is sthe recursion in the algorithm
    # If the length of the list larger than the threshold preselected
    ### Do the divide and conquer again
    # Else (the length of the listsmaller than the threshold preselected)
    ### Do the insertion_sort defined above
    if len(left_third) > threshole:
        step += 1
        merge_sort_3_insertion(left_third, threshole)
        step += 1
    else:
        insertion_sort(left_third)
        step += 1

    if len(mid_third) > threshole:
        step += 1
        merge_sort_3_insertion(mid_third, threshole)
        step += 1
    else:
        insertion_sort(mid_third)
        step += 1

    if len(right_third) > threshole:
        step += 1
        merge_sort_3_insertion(right_third, threshole)
        step += 1
    else:
        insertion_sort(right_third)
        step += 1
```

```
# After conquer, we need to implement the final step: combine
# i, j, k are the tracking numbers for 3 sublists
i = 0
step += 1
j = 0
step += 1
k = 0
step += 1
# l is the tracking number for the list
l = 0
step += 1

# Now the comparison -> compare and then combine into a single
### sorted list

# This while loop ensure that there is no sublists has appended
### all its element
while i < len(left_third) and j < len(mid_third) and k < len(right_third):
    step += 1
    # We will have the current_min variables to be the smallest
    # using current_min, we will find the smallest number between
    ### the 3 sublists, to merge into the big list

    # Find the smallest between the first element of sublist left and mid
    if left_third[i] < mid_third[j]:
        step += 1
        current_min = left_third[i]
        step += 1
        # list tracking is to see which sublist has the smallest element
        # we temporarily store the value list_tracking as 0 as
        ### the left_third has the smallest value
        list_tracking = 0
        step += 1
    else:
        current_min = mid_third[j]
        step += 1
        list_tracking = 1
        step += 1
        # we temporarily store the value list_tracking as 1 as
        ### the mid_third has the smallest value
    if current_min > right_third[k]:
        step += 1
        # list tracking continues to see which sublist has the smallest elem
        # As the right_third now has the smallest value
        # We store the list_tracking to be 3
        # Else, the current_min is the min of all 3 lists
        # And we don't need to change the list_trackking and current_min
        current_min = right_third[k]
        step += 1
        list_tracking = 2
        step += 1

    # if the list has the smallest element is the left_third:
    if list_tracking == 0:
        step += 1
        alist[l] = left_third[i]
        step += 1
        i += 1
        step += 1
        l += 1
        step += 1
    # if the list has the smallest element is the mid_third:
    elif list_tracking == 1:
        step += 1
        alist[l] = mid_third[j]
        step += 1
        j += 1
        step += 1
        l += 1
        step += 1
    # if the list has the smallest element is the right_third:
    else:
        step += 1
        alist[l] = right_third[k]
        step += 1
        k += 1
        step += 1
        l += 1
        step += 1

# As the previous while loop ends when 1 list ran out of element
# We have 3 chances:
# right_third ran out of element
while i < len(left_third) and j < len(mid_third):
    # begin to compare and merge, similar to merge sort 2 ways
    if left_third[i] < mid_third[j]:
        step += 1
        alist[l] = left_third[i]
        step += 1
        i += 1
        step += 1
    else:
        step += 1
        alist[l] = mid_third[j]
        step += 1
        j += 1
        step += 1
    l += 1
# mid_third ran out of element
while i < len(left_third) and k < len(right_third):
    step += 1
    # begin to compare and merge, similar to merge sort 2 ways
    if left_third[i] < right_third[k]:
        step += 1
        alist[l] = left_third[i]
        step += 1
        i += 1
        step += 1
    else:
        alist[l] = right_third[k]
        step += 1
        k += 1
        step += 1
    l += 1
    step += 1
# left_third ran out of element
while j < len(mid_third) and k < len(right_third):
    step += 1
    # begin to compare and merge, similar to merge sort 2 ways
    if mid_third[j] < right_third[k]:
        step += 1
        alist[l] = mid_third[j]
        step += 1
        j += 1
        step += 1
    else:
        alist[l] = right_third[k]
        step += 1
        k += 1
        step += 1
    l += 1
    step += 1


# As the previous while loop ends when 1 more list ran out of element
# Hence, 2 lists have ran out of element to merge
# We have 3 chances:
# With each type of remainding list, we append everything into the big list
# left_third still have elements
while i < len(left_third):
    step += 1
    alist[l:(l + len(left_third)-i)] = left_third[i:]
    step += 1
    l += len(right_third)-i
```

```
                    step += 1
                    i = len(left_third)
                    step += 1
                # mid_third still have elements
                while j < len(mid_third):
                    step += 1
                    alist[l:(l + len(mid_third)-j)] = mid_third[j:]
                    step += 1
                    l += len(mid_third)-j
                    step += 1
                    j = len(mid_third)
                    step += 1
                # right_third still have elements
                while k < len(right_third):
                    step += 1
                    alist[l:(l + len(right_third)-k)] = right_third[k:]
                    step += 1
                    l += len(right_third)-k
                    step += 1
                    k = len(right_third)
                    step += 1

        return alist

#####################################3
# test code

import numpy as np
alist = list(np.random.permutation(100))
print("Input: ", alist)
print()
print('Output: ',merge_sort_3_insertion(alist,20))
print()
print('steps: ',step)
```

```
Input:  [43, 54, 46, 49, 66, 80, 58, 45, 3, 55, 67, 41, 2, 97, 21, 42, 10, 79, 28, 30, 90, 85,

Output:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23

steps:  2733
```

We have 2 desired outcomes: The first is to compare between merge-sort 3 way and the augmented merge sort. The second is that we are also trying to see which value of the threshold can bring the least steps and the best running time. These code below are targetted to identify such outcome. As for the scope of this assignment, we will only test on the case: list of 1000 elements. As we do not want to only test on the worst case scenarios, we compute the random permutation using numpy library. As we have random elements, we will conduct 10 randomly ordered list of

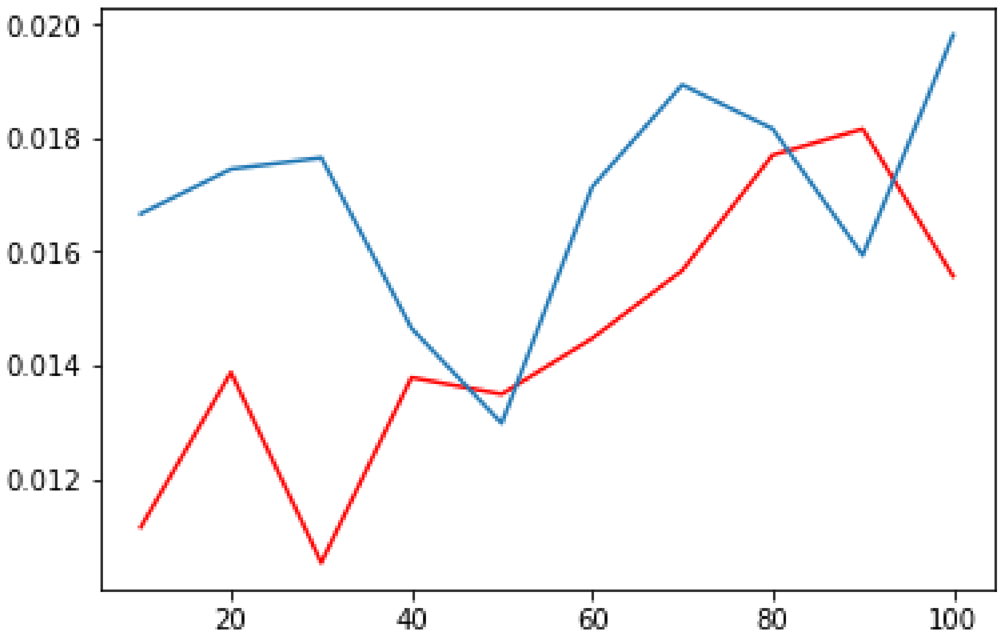
1000 elements and take the mean of those values. Note: I wanted to compute the same testing cases for 2 algorithms, but list in Python are shallow copies. Also, at this stage, I don't want to create a deep copy of a list using library. Hence, I allows evey testing cases to be random. Therefore, can occurs fluctuation in the graph.

Furthermore, we test on 10 different values of k: 10,20,30,...,100.

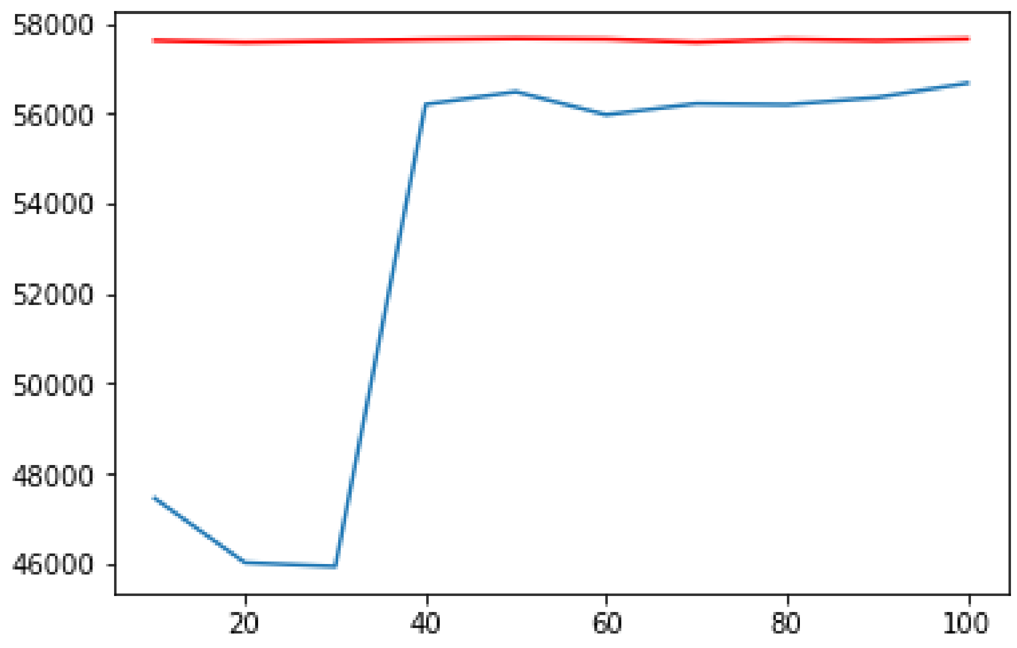
```
In [190]: import numpy as np
          import matplotlib.pyplot as plt
          import time

          # Compare the 2 codes above merge_sort_3 and merge_sort_3_insertion

          alist = []
          alist_1 = []
          index = []
          time_record = []
          time_record_1 = []
          step_record = []
          step_1_record = []
          for x in range(10):
              time_record.append([])
              time_record_1.append([])
              step_record.append([])
              step_1_record.append([])

          alist = list(np.random.permutation(1000))
          for k in range(10,110,10):
              index.append(k)
              for i in range(10):
                  step=0
                  start_time = time.time()
                  merge_sort_3_insertion(alist,k)
                  time_record_1[index.index(k)].append((time.time() - start_time))
                  step_record[index.index(k)].append(step)

                  alist = list(np.random.permutation(1000))

                  step=0
                  start_time = time.time()
                  merge_sort_3(alist)
                  time_record[index.index(k)].append((time.time() - start_time))
                  step_1_record[index.index(k)].append(step)
                  alist = list(np.random.permutation(1000))

          avg_time_record = []
          avg_time_record_1 = []
          avg_step_record = []
          avg_step_1_record = []
```

```
print(step_record)

for i in range(10):
    avg_time_record.append(np.mean(time_record[i]))
    avg_time_record_1.append(np.mean(time_record_1[i]))
    avg_step_record.append(np.mean(step_record[i]))
    avg_step_1_record.append(np.mean(step_1_record[i]))

print("k yields the best running time:", index[avg_time_record_1.index(min(avg_time_
plt.plot(index, avg_time_record_1,color='red')
plt.plot(index, avg_time_record)
plt.show()

print("k yields the least steps:", index[avg_step_1_record.index(min(avg_step_1_reco
plt.plot(index, avg_step_record)
plt.plot(index, avg_step_1_record,color='red')
plt.show()
```

```
[[47606, 47733, 47312, 47745, 47411, 47501, 47308, 47433, 47458, 47061], [46085, 45986, 45799,
k yields the best running time: 30
```



```
k yields the least steps: 20
```



The red-line represents the augmented merge sort. The blue line represent the 3-way merge sort. The x-axis are the values of k, the y-axis represents the running time and the steps.

We can see that k=20 and k=30 gives the best outcome for k-way merge sort for a list of 1000. We can use these value to evaluate in section 4.

Briefly comparing the 3-way merge sort and augmented merge sort: the augmented merge sort seems taking less time but more steps.

```
In [ ]:
```

3/ (Optional challenge) (#sort, #optimalalgorithm) Implement k-way merge sort, where the user specifies k. Develop and run experiments to support a hypothesis about the "best" value of k.

General idea: k-way merge sort is a recursive algorithm that continually splits a list in k parts. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on the k sublists. Once the k sublists are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking 3 smaller sorted lists and combining them together into a single, sorted, new list (the combine phase).

The difficult parts are storing sublists and merging. The storing is conducted in a list in list type. Also, the conducted merging has a lot of steps as we cannot make a detailed decision tree such as 2-way or 3-way. Here is the code:

```
In [149]: # import a function for later use
          import math

          # define the function for 3-way merge sort
          # call it merge_sort_k
          # the function takes the input: a list and a value k

          step = 0
          def merge_sort_k(alist,k):
              global step
              # If the length of the alist is 0 or 1
              ### then we can consider it is sorted
              ### then the "return" at the end of the code will return the
              ### sorted list, which is the list of 0 or 1 given
              # these lists are the base case for merge sort
              if len(alist) > 1:
                  step += 1
                  # We are storing data as a list of k sublists
                  sub_list = []
                  step += 1
                  # We create k sublists using the floor function as follow
                  # First, create k-1 lists
                  for i in range(k-1):
                      step += 1
                      sub_list.append(alist[(i*(len(alist))//k):(((i+1)*len(alist))//k)])
                      step += 1
                  # The last list has all the remainding elements
                  sub_list.append(alist[((k-1)*len(alist))//k:])
                  step += 1

                  # As we divide and conquer
                  # We divide the list into k sublists above
                  # Now we conquer each sublist through the following codes:
                  # This is the recursion in the algorithm
                  for i in range(k):
                      step += 1
                      merge_sort_k(sub_list[i],k)
                      step += 1

                  # We have a tracking number for each of the sublist
                  tracking = []
                  step += 1
                  # Initially, all are 0, represent the 1st element of each list
                  for i in range(k):
                      step += 1
                      tracking.append(0)
                      step += 1

                  # This is the tracking number for the list
                  l = 0
                  step += 1

                  # We will the sumlist function, which will be describe later
                  # sum_list is the total number of elements of all sublists
                  sum_list = sumlist(sub_list)
                  step += 1

              # This is the condition for the while-loop to ensure all lists have to run
              ### through all its elements
                  while sum(tracking) < sum_list:
                      step += len(tracking)
                      # Using infinity will ensure the whenever an number exists,
                      ### current_min will be changes into that value
                      current_min = math.inf
                      step += 1
                      # Use a for-loop to find the smallest element each time
                      for i in range(k):
                          step += 1
                          if tracking[i] < len(sub_list[i]):
                              step += 1
                              if current_min > sub_list[i][tracking[i]]:
                                  step += 1
                                  current_min = sub_list[i][tracking[i]]
                                  step += 1
                                  current_min_location = i
                                  step += 1


                      # This is the merging part
                      # merge the smallest value to the big list
                      # As we use 1 element of 1 sublist, we add 1 to to tracking number of th
                      alist[l] = current_min
                      step += 1
                      l += 1
                      step += 1
                      tracking[current_min_location] += 1
                      step += 1


              return alist

          # sumlist is a function to calculate the total length of all elements of a list
          def sumlist(input_list):
              global step
              step += 1
              sum_list = 0
              step += 1
              for i in range(len(input_list)):
                  step += 1
                  sum_list += len(input_list[i])
                  step += 1
```

```
            return sum_list

        #####################################3
        # test code

        import numpy as np
        alist = list(np.random.permutation(100))
        print("Input: ", alist)
        print()
        print('Output: ',merge_sort_k(alist,5))
        print()
        print('steps: ',step)
```

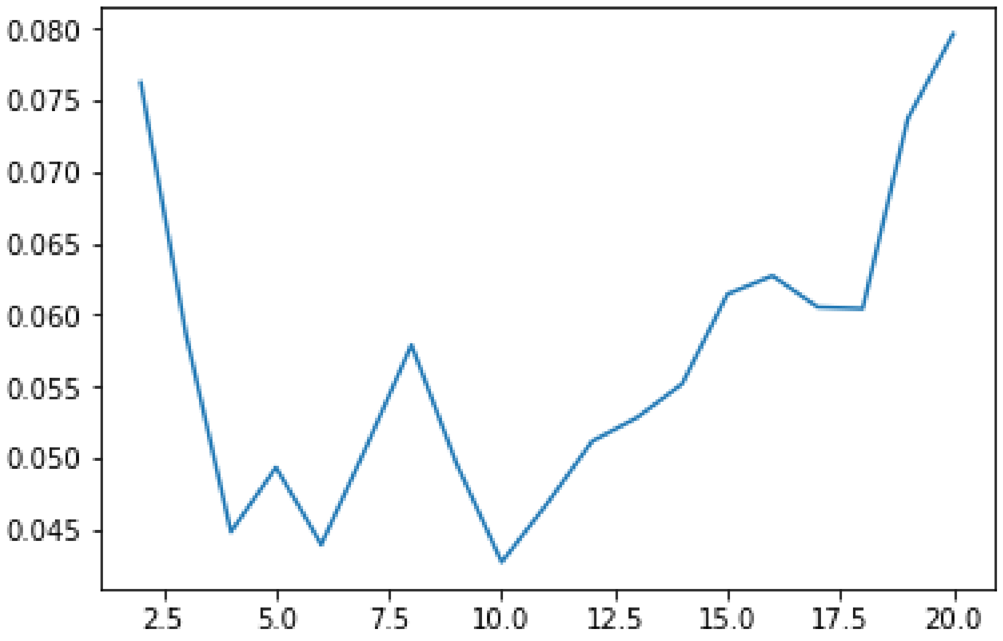
```
Input:  [37, 23, 17, 6, 74, 10, 48, 78, 93, 99, 28, 25, 84, 53, 15, 19, 79, 21, 85, 42, 70, 31

Output:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23

steps:  8541
```

We are interested in which value of k is the best. We conduct tests on different value of k: from 2 to 20. Each value of k, we test them using 30 randomly ordered list of 1000 elements. (The reason choosing 30 is in that in experiments, 30 is the threshold for normal aproximation to mitigate the possible biases caused by random sampling).

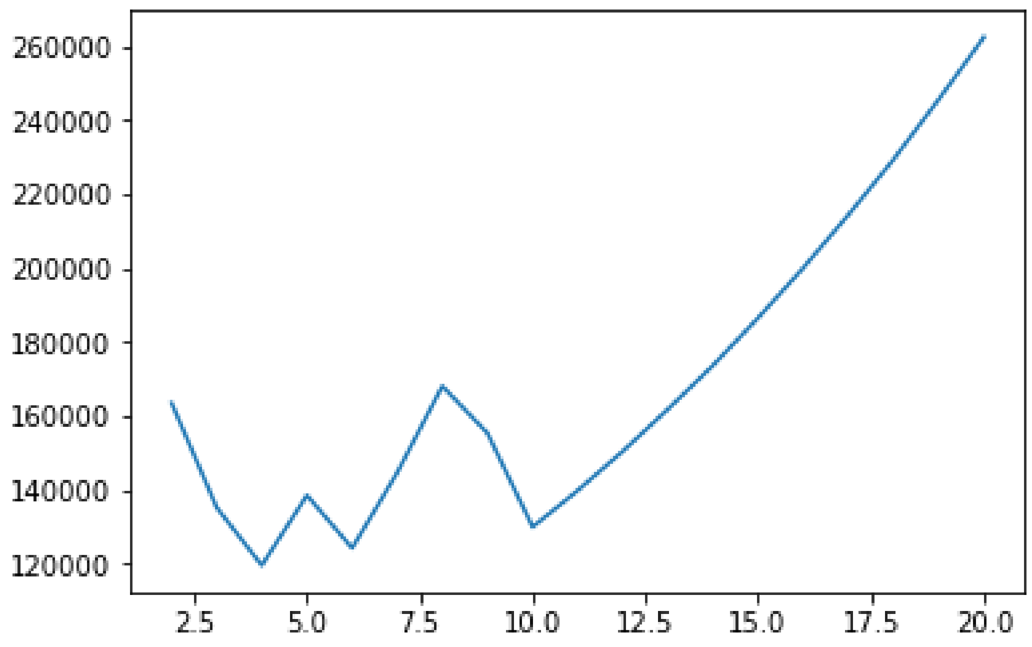
```
In [192]: import numpy as np
          import matplotlib.pyplot as plt
          import time

          # Compare the 2 codes above merge_sort_3 and merge_sort_3_insertion

          index = []
          time_record = []
          step_k_record = []
          for x in range(19):
              time_record.append([])
              step_k_record.append([])

          for k in range(2,21):
              index.append(k)
              for i in range(30):
                  alist = list(np.random.permutation(1000))
                  step = 0
                  start_time = time.time()
                  merge_sort_k(alist,k)
                  time_record[index.index(k)].append((time.time() - start_time))
                  step_k_record[index.index(k)].append(step)
```

```
avg_time_record = []
avg_step_k_record = []

for i in range(19):
    avg_time_record.append(np.mean(time_record[i]))
    avg_step_k_record.append(np.mean(step_k_record[i]))

print("k yields best running time: ", index[avg_time_record.index(min(avg_time_recor
plt.plot(index, avg_time_record)
plt.show()

print("k yields least steps: ", index[avg_step_k_record.index(min(avg_step_k_record)
plt.plot(index, avg_step_k_record)
plt.show()
```

```
k yields best running time:  10
```



```
k yields least steps:  4
```



The graph represents the fluctuation of the running time and steps of the k-way merge sort depends on k. The x-axis is the value of k, whereas the y-axis is the running time and number of steps

We can see that k=4 and k=10 gives the best outcome for k-way merge sort for a list of 1000. We can use these value to evaluate in section 4.

In addition, when we yield such result, we can further compare the efficiency of k=4 and k=10 using p-value testing. As we tested over 30 sets, we can plot all 30 data points, compute their mean and standard deviation, and assume they are normal distribution. Then, we have 2 normal distributions with different mean and standard deviations. We can use p-value testing with the confidence interval = 95%. From that, we can evaluate whether we can plausibly conclude that k=4 or k=10 is better. For example, if the mean of k=4 stands outside the 95% confidence interval of the normal distribution of k=10, we can plausibly conclude that k=10 is better than k=4. However, "better" here only applies to certain sets of values and input. For example, we can test on the case of 1000 elements list. However, for a different list (100000 elements), we need to conduct the test again.

```
In [ ]:
```

4/ (#complexity, #optimalalgorithm) Analyze and compare the practical run times of regular merge sort, three-way merge sort, and the augmented merge sort from (2). Make sure to define what each algorithm's complexity is and to enumerate the explicit assumptions made to assess each algorithm's run time. Your results should be presented in a table, along with an explanatory paragraph and any useful graphs or other charts to document your approach. Part of your analysis should indicate whether or not there is a "best" variation. Compare your benchmarks with the theoretical result we have discussed in class

We first need to compute the normal merge sort.

```
In [152]: #normal merge sort 2-way mergesort
          step = 0
          def merge_sort_2(alist):
              global step
              step += 1
              if len(alist)>1:
                  step += 1
                  mid = len(alist)//2
                  step += 1
                  left_half = alist[:mid]
                  step += 1
                  right_half = alist[mid:]
                  step += 1

                  merge_sort_2(left_half)
                  step += 1
                  merge_sort_2(right_half)
                  step += 1

                  i=0
                  step += 1
                  j=0
                  step += 1
                  k=0
                  step += 1
                  while i < len(left_half) and j < len(right_half):
                      step += 1
                      if left_half[i] < right_half[j]:
                          step += 1
                          alist[k]=left_half[i]
                          step += 1
                          i=i+1
                          step += 1
                      else:
                          step += 1
                          alist[k]=right_half[j]
                          step += 1
                          j=j+1
                          step += 1
                      k=k+1
                      step += 1

                  while i < len(left_half):
                      step += 1
                      alist[k:(k + len(left_half)-i)] = left_half[i:]
                      step += 1
                      k += len(left_half)-i
                      step += 1
```

```
                i = len(left_half)
                step += 1
            while j < len(right_half):
                step += 1
                alist[k:(k + len(right_half)-j)] = right_half[j:]
                step += 1
                k += len(right_half)-j
                step += 1
                j = len(right_half)
                step += 1

        return alist

    ##########################################
    # Tests

    import numpy as np
    alist = list(np.random.permutation(100))
    print("Input: ", alist)
    print()
    print('Output: ',merge_sort_2(alist))
    print()
    print('steps: ',step)
```

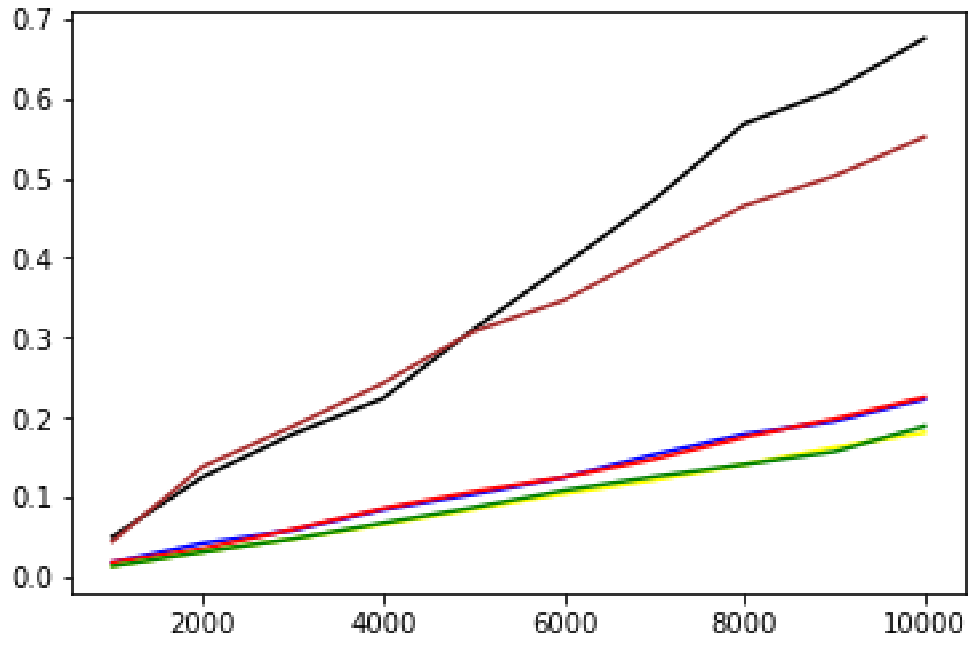
```
Input:  [7, 76, 19, 58, 34, 56, 15, 62, 46, 93, 99, 44, 9, 63, 50, 51, 68, 69, 67, 53, 97, 72,

Output:  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23

steps:  4206
```

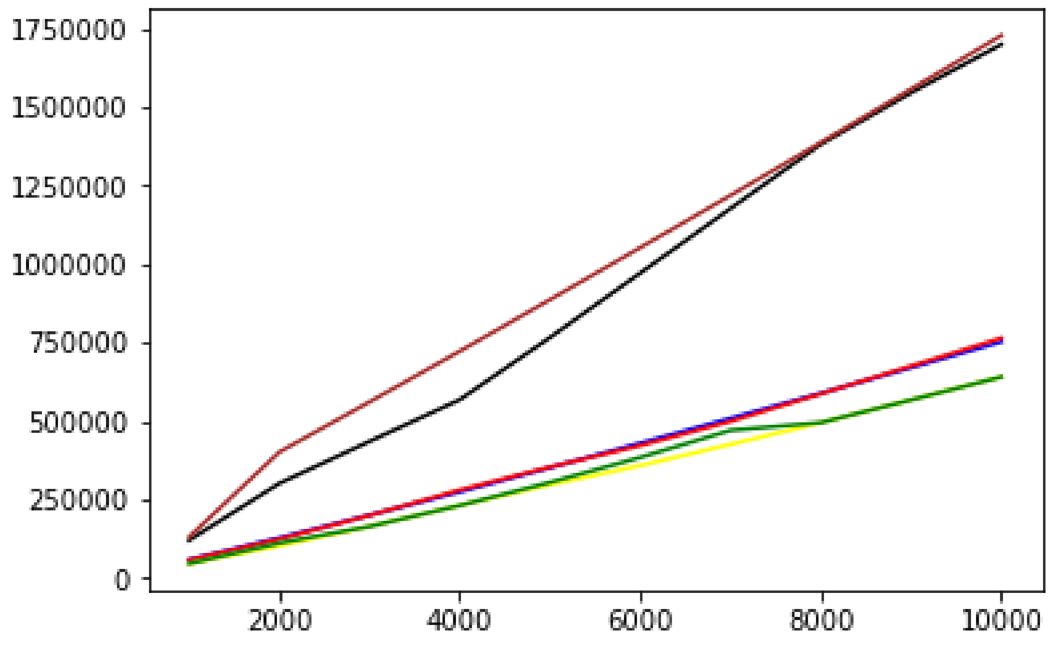
We are comparing the different codes. As this is a rigorous comparisons, we are conducting 300 tests on these algorithms: 10 value of the length of the list: 1000, 2000, 3000, ..., 10000. Each value of the len(test_list), we conduct 30 randomly ordered lists. (The reason choosing 30 is in that in experiments, 30 is the threshold for normal aproximation to mitigate the possible biases caused by random sampling).

The 6 algorithms we are comparing are 1/ 2-way merge sort 2/ 3-way merge sort 3/ augmented 3-way merge sort for the threshold = 30 (As tested above for having good running time) 4/ augmented 3-way merge sort for the threshold = 20 (As tested above for having the best number of steps) 5/ k-way merge sort for the threshold = 10 (As tested above for having good running time) 6/ k-way merge sort for the threshold = 4 (As tested above for having the least number of steps)

```
In [193]: import numpy as np
          import matplotlib.pyplot as plt
          import time

          # Compare the 2 codes above merge_sort_3 and merge_sort_3_insertion
```

```
index = []

time_2_record = []
time_3_record = []
time_3_insertion_10_record = []
time_3_insertion_20_record = []
time_4_k_record = []
time_6_k_record = []
time_record = []

step_2_record = []
step_3_record = []
step_3_insertion_record = []
step_4_k_record = []
step_6_k_record = []
step_record = []

for i in range(6):
    time_record.append([])
    step_record.append([])
    for x in range(10):
        time_record[i].append([])
        step_record[i].append([])


for k in range(1000,11000,1000):
    index.append(k)
    for i in range(30):
        for t in range(6):
            alist = list(np.random.permutation(k))
            step = 0
            if t == 0:
                start_time = time.time()
                merge_sort_2(alist)
                time_record[t][index.index(k)].append((time.time() - start_time))
                step_record[t][index.index(k)].append(step)
            if t == 1:
                start_time = time.time()
                merge_sort_3(alist)
                time_record[t][index.index(k)].append((time.time() - start_time))
                step_record[t][index.index(k)].append(step)
            if t == 2:
                start_time = time.time()
                merge_sort_3_insertion(alist,20)
                time_record[t][index.index(k)].append((time.time() - start_time))
                step_record[t][index.index(k)].append(step)
            if t == 3:
                start_time = time.time()
                merge_sort_3_insertion(alist,30)
                time_record[t][index.index(k)].append((time.time() - start_time))
                step_record[t][index.index(k)].append(step)
            if t == 4:
                start_time = time.time()
                merge_sort_k(alist,4)
                time_record[t][index.index(k)].append((time.time() - start_time))
                step_record[t][index.index(k)].append(step)
            if t == 5:
                start_time = time.time()
                merge_sort_k(alist,10)
                time_record[t][index.index(k)].append((time.time() - start_time))
                step_record[t][index.index(k)].append(step)

avg_time_record = [[],[],[],[],[],[]]
avg_step_record = [[],[],[],[],[],[]]




for i in range(6):
    for x in range(10):
        avg_time_record[i].append(np.mean(time_record[i][x]))
        avg_step_record[i].append(np.mean(step_record[i][x]))

colorsss = ['blue','red','yellow','green', 'black', 'brown']
for i in range(6):
    plt.plot(index, avg_time_record[i], color = colorsss[i])
plt.show()

for i in range(6):
    plt.plot(index, avg_step_record[i], color = colorsss[i])
plt.show()
```





The 2 graphs above represents the running time and number of steps for the 6 algorithms above The x-axis is the length of the testing lists; the y-axis is the running time and the number of steps

The line are color coded: 'blue': 2-way merge sort 'red': 3-way merge sort 'yellow': augmented 3-way merge sort, k = 20 'green': augmented 3-way merge sort, k = 30 'black': 4-way merge sort 'brown': 10-way merge sort

```
In [196]: print("For list of 1000 elements")
          print("avg time for 2-way merge sort:", avg_time_record[0][0])
          print("avg time for 3-way merge sort:", avg_time_record[1][0])
          print("avg time for augmented 3-way merge sort with k=20:", avg_time_record[2][0])
          print("avg time for augmented 3-way merge sort with k=30:", avg_time_record[3][0])
          print("avg time for 4-way merge sort:", avg_time_record[4][0])
          print("avg time for 10-way merge sort:", avg_time_record[5][0])
          print()
          print("avg step for 2-way merge sort:", avg_step_record[0][0])
          print("avg step for 3-way merge sort:", avg_step_record[1][0])
          print("avg step for augmented 3-way merge sort with k=20:", avg_step_record[2][0])
          print("avg step for augmented 3-way merge sort with k=30:", avg_step_record[3][0])
          print("avg step for 4-way merge sort:", avg_step_record[4][0])
          print("avg step for 10-way merge sort:", avg_step_record[5][0])
```

```
For list of 1000 elements
avg time for 2-way merge sort: 0.0178378423055
avg time for 3-way merge sort: 0.0181090116501
avg time for augmented 3-way merge sort with k=20: 0.0135378599167
avg time for augmented 3-way merge sort with k=30: 0.0134078343709
avg time for 4-way merge sort: 0.0501440684001
avg time for 10-way merge sort: 0.0445953925451

avg step for 2-way merge sort: 58531.8333333
avg step for 3-way merge sort: 57622.4666667
avg step for augmented 3-way merge sort with k=20: 45967.2666667
avg step for augmented 3-way merge sort with k=30: 46013.6
avg step for 4-way merge sort: 119697.266667
avg step for 10-way merge sort: 130064.466667
```

As we can see from the graph: 1/ The augmented 3-way merge-sort is having the least running time and number of steps, both for the threshold = 20 and 30 (green and yellow line). These 2 algorithms have the relatively same running time and running steps.

2/ The 2-way and 3-way merge sort have trlatively similar number of steps and running time.

3/ The 4-way merge sort and 10-way merge sort have the biggest number of steps and the longest running time. The 10-way merge sort seems to have lower running time than the 4-way merge sort when the length of the testing list > 5000. Also, the 10-way merge sort has less steps than the 4-way merge sort in the tested cases.

We have: The complexity of the algorithms are: (as we discussed in class already, this is an induction):

$$2 - way - merge - sort : O(n \log_2 n)$$

$$3 - way - merge - sort : O(n \log_3 n)$$

$$4 - way - merge - sort : O(n \log_4 n)$$

$$10 - way - merge - sort : O(n \log_{10} n)$$

As in here, we choose a low threshold for k: k = 20 and 30: much smaller than the input list (1000 elements and above). Hence:

$$augmented - 3 - way - merge - sort, k = 20 : O(n \log_3 n)$$

$$augmented - 3 - way - merge - sort, k = 30 : O(n \log_3 n)$$

Assumptions and limitations: 1/ As we know: running time is highly dependable on different computers. Hence, besides running time, I also incorporate steps counting as the method of measuring complexity. Also, as conting steps is depends on the coder. Each coder might have different way of defining "steps". The steps metrics is subjective to the coder Besides, there are other metrics those we can measure (storage, etc.) In the scope of this assignment, I choose to measure only steps and time.

2/ Our test cases are only for lists have the length between 1000 and 10000. In real life, the lists usually have bigger length. For example, an app have 1000000 data points might need to sort a list over 1000000 elements. This is a limitation of this tests and measurements. However, in regards of this assignment and running time measurement (as we test over 300 cases), 1000 to 10000 is appropriate.

3/ One limitation of the k-way merge sort. As we do not know what is the value of k. Hence, we cannot code a customized code for a specific k. Therefore, the code is slightly generalize, which can causes more steps and running time. This is one factor making k-way merge sort has larger steps and longer running time. Also, if we consider the recursive tree, then the tree with big k will have less levels, but more recursive calls at each levels. The comparison also is more complicated and time-consuming than 2-way merge sort. If we consider big-O-notation, then we can see that the larger the value k, the smaller the big-O-notation:

$$k - way - merge - sort : O(n \log_k n)$$

However, if we use the formula:

$$O(n \log_k n) = O(n \log_2 n) * \frac{1}{\log_2 k}$$

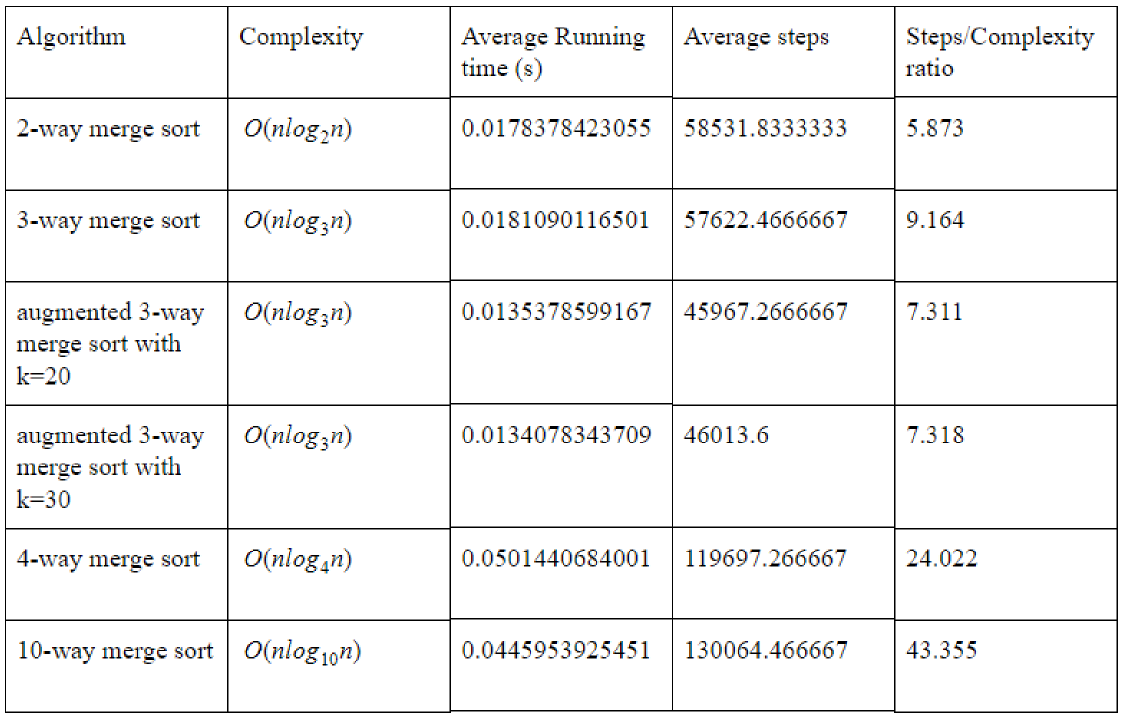
As $\frac{1}{\log_2 k}$ is a constant, $O(n \log_k n)$ is equivalent to $O(n \log_2 n)$ in complexity term for a specific k, which making 2-way and k-way having similar complexity.

In my opinion, there is no best algorithms for all cases. Each algorithm has its own strength and weaknesses. For example, merge sort has a great big-O-notation. However, when the list has small length, insertion sort work just as well. In our assignment, as we only tests for a small number of cases, even though augmented 3-way merge sort works really well, but we cannot guaranteed those assumptions for further cases. However, through this, we can see that the merge sort can be modified and improved using different ways.

I have a hypothesis: with big input list, the running time for 2-way merge sort, 3-way merge sort and augmented 3-way merge sort will be approximately the same. However, for k-way, if it want to reach similar running time, we should modify the steps for a customized k rather than a general code working for all values of k like the code above.

```
In [202]: from IPython.display import Image

          Image("comparison.png")
```

```
Out[202]:
```



Algorithm	Complexity	Average Running time (s)	Average steps	Steps/Complexity ratio
2-way merge sort	$O(nlog_2n)$	0.0178378423055	58531.8333333	5.873
3-way merge sort	$O(nlog_3n)$	0.0181090116501	57622.4666667	9.164
augmented 3-way merge sort with k=20	$O(nlog_3n)$	0.0135378599167	45967.2666667	7.311
augmented 3-way merge sort with k=30	$O(nlog_3n)$	0.0134078343709	46013.6	7.318
4-way merge sort	$O(nlog_4n)$	0.0501440684001	119697.266667	24.022
10-way merge sort	$O(nlog_{10}n)$	0.0445953925451	130064.466667	43.355

This is the table of comparison between 6 types of algorithms in term of complexity, running time, steps, and also the ratio between steps and complexity $= \frac{number-of-steps}{complexity}$ Note: this is only applicable for the case the input list has 1000 elements.

In general, for complexity, we don't sepcific log_xn, we only state: $O(n\log n)$ One reason is because the constant $\frac{1}{\log_k n}$ as we discussed above, making all these complexity be merely $n \log n$. The other parts belong to the constant coefficient.

```
In [ ]:
```

Appendix

1 **organization: This is a well-organized presentation. There is a coherence, consistency throughout the assignment, with linking between solving each subproblem in a big problem. The important findings are clearly noted and marked (e.g. function A.1.1), with supports later used of similar findings. This presentation also utilize the similarity between problems to reduce the length of presentation but still keep the quality. This is a highly effective way to organize a coding-work.**

2 **dataviz: This work contains multiple appropriate data visualization with effective data storage, color coding, and variables chosen. I also effectively analyzes and interprets a data visualization and provide appropriate justification and details to back up my arguments.**

3 **hypothesisdevelopment: This assignment uses hypothesisdevelopment to match the link between patterns and initial data and the generation of hypotheses and clearly explains the connection: using data, graphs, sampling method, sampling sizie to back up (choosing optimal k for augmented merge sort and k-way merge sort). This is a deep application of #hypothesisdevelopment**

4 **sampling: In this work, the student understand that he need to create appropriate sampling to test the codes and compare between different algorithms. Hence, the student decided to use random sampling (using numpy) and choose a sampling size of 30 to mitigate biases (and can plausibly approximate normal distribution). This is a deep knowledge of #sampling in testing algorithms.**