Assignment 1- CS110

1) Three-way merge sort

#sort Implement three-way merge sort in Python. It should at a minimum accept lists off integers as input.

In [103]:
```
from math import ceil

def three_way_merge_sort(a):
    if len(a)<2:
        return a #checks if already sorted

    m= ceil(len(a)/3) # division point


    b=[] # empty list to store sorted lists for merging
    for i in range(0,len(a),m): # iterate through a in segments of size m
        b.append(three_way_merge_sort(a[i:i+m])) # recursively calls the merge so

    c=[] # making a new list to hold the merged list
    while len(b)>0: # we need to empty each of the lists in b into c
        minimum=b[0][0] # initializing the minimum (first of first list)
        index=0 # setting the index to 0
        for i in b:  # for each of the lists in b
            if i[0]<minimum: # check if new minimum is needed
                minimum=i[0] # change minimum
                index=b.index(i) # set index to position of list
        c.append(minimum) # append the minimum value to c
        b[index].pop(0) # remove minimum value from b
        if len(b[index])==0: # remove empty lists
            b.remove(b[index])
    return c

x=[32,4,13,15,1,2,35,1,2,5,1,23]
three_way_merge_sort(x)


#comment: I made the k-way merge sort first and this is just an adaptation of it
```

Out[103]:
```
[1, 1, 1, 2, 2, 4, 5, 13, 15, 23, 32, 35]
```

2) Insertion-merge-sort

#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.

In [109]:
```
 from math import ceil

def insertion_sort(a):
    for j in range(1,len(a)): # for each element in the list
        key= a[j] # define a key that is equal to the element in position j
        i = j-1 # and an i which is the index of the element before the one in pos
        while i>=0 and a[i]>key: # checks if insertion should happen
            a[i+1]=a[i] # insert the new value into the correct position
            i-=1
        a[i+1]=key #move on to the next step
    return a #return the sorted list

def three_way_merge_insertion_sort(a):
    if len(a)<2:
        return a #checks if a is smaller than two elements, in which case it is a
    if len(a)<6:
        insertion_sort(a) # for lists shorter than 6 elements, use insertionsort
    else:
        m= ceil(len(a)/3) # determine break points

        b=[] # empty list to store sorted lists for merging
        for i in range(0,len(a),m): # iterate through a in segments of size m
            b.append(three_way_merge_sort(a[i:i+m])) # recursively calls the merg

        c=[] # making a new list to hold the merged list
        while len(b)>0: # we need to empty each of the lists in b into c
            minimum=b[0][0] # initializing the minimum (first of first list)
            index=0 # setting the index to 0
            for i in b:  # for each of the lists in b
                if i[0]<minimum: # check if new minimum is needed
                    minimum=i[0] # change minimum
                    index=b.index(i) # set index to position of list
            c.append(minimum) # append the minimum value to c
            b[index].pop(0) # remove minimum value from b
            if len(b[index])==0: # remove empty lists
                b.remove(b[index])
        return c

x=[32,4,13,15,1,2,35,1,2,5,1,23]
three_way_merge_insertion_sort(x)
```

Out[109]: [1, 1, 1, 2, 2, 4, 5, 13, 15, 23, 32, 35]

3) k-way merge sort

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

I believe that the best value of k will be two, since that is the stage at which there will be the least comparisons.

In [111]:
```
from math import ceil


def  k_way_merge_sort(a,k):
    if len(a)<2:
        return a #checks if already sorted

    m= ceil(len(a)/k) # division point


    b=[] # empty list to store sorted lists for merging
    for i in range(0,len(a),m): # iterate through a in segments of size m
        b.append(three_way_merge_sort(a[i:i+m])) # recursively calls the merge so

    c=[] # making a new list to hold the merged list
    while len(b)>0: # we need to empty each of the lists in b into c
        minimum=b[0][0] # initializing the minimum (first of first list)
        index=0 # setting the index to 0
        for i in b:  # for each of the lists in b
            if i[0]<minimum: # check if new minimum is needed
                minimum=i[0] # change minimum
                index=b.index(i) # set index to position of list
        c.append(minimum) # append the minimum value to c
        b[index].pop(0) # remove minimum value from b
        if len(b[index])==0: # remove empty lists
            b.remove(b[index])
    return c

x=[32,4,13,15,1,2,35,1,2,5,1,23]
k_way_merge_sort(x,4)
```

Out[111]:
```
[1, 1, 1, 2, 2, 4, 5, 13, 15, 23, 32, 35]
```

In [99]:
```
import time
import numpy as np
import random

# this part generates a list of 5000 lists of length 0 to 1999 and random integers
list_of_lists=[[random.randint(0,1000) for i in range(random.randint(0,2000))] fo

times=[] # list to store the average times of each of the k_way_merge_sort
for i in range(2,100): # we start with the 2-way merge sort and go up to 100
    start=time.time() # starts the timer
    [k_way_merge_sort(list_of_lists[j],i) for j in range(2000)] # sorts each of t
    end=time.time() # stops the timer
    times.append((end-start)/2000) # calculates the average time
```

In [100]:
```
import matplotlib.pyplot as plt

plt.plot(range(2,100),times) # plots the sorting algorithms against their average
plt.show()
```

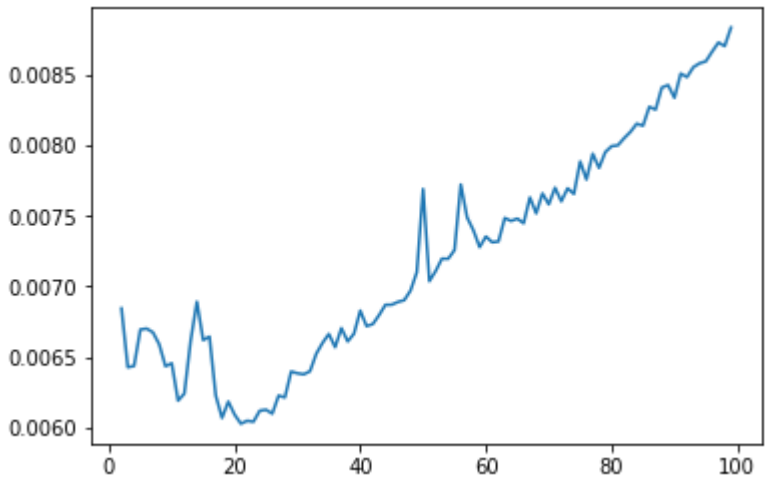


Figure 1 k plotted against the average time taken to sort a list of between 0 and 1999 elements.

As the graph above shows, there is a lot of random variation in the actual average running time of the algorithm. I could design more experiments to see if the length of the list, the ratio of the average length of the lists to k or many more factors have a significant influence, etc.
I could also decrease the random variation by generating more lists than the 2000 I am currently giving it.
Nonetheless, we see that the running time does not keep decreasing but increases after the first 20 values of k. This is due to the fact that the algorithm needs to merge many more individual lists at that point. The number of elements didn't change, but it needs to do many more comparisons. This leads to the higher running time.
The minimum of the running time appears to be around 20, which would contradict my hypothesis of 2. I believe this may be only the case for this specific length of list and computer I am running on. To overcome these limitations, I would need more resources and time.
While the experiment did not find the "best" value of k with absolute certaintry, since it is hard to make any generalizations between 2 and 20 going from the plot, it showed me that there is more to it than keeping on dividing the list into more and more pieces.
The details of the optimization will probably depend on hardware (like how many cores my machine has to run the code) as well as the length of the list, etc.

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 result 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 benchmark with the theoretical result we have discussed in class.

Practical run times: to calculate this, I recycled my code from (3)
The results came out as 0.00658, 0.00621 an 0.00618 We see that that they are not much different, but that both the three-way merge sort and augmented three-way merge sort win over the regular merge sort.

In [101]:
```
comparisons=[]

start=time.time() # starts the timer
[k_way_merge_sort(list_of_lists[j],2) for j in range(2000)] # sorts each of the 5
end=time.time() # stops the timer
comparisons.append((end-start)/2000) # append the average running time to "compar

start=time.time() # starts the timer
[k_way_merge_sort(list_of_lists[j],3) for j in range(2000)] # sorts each of the 5
end=time.time() # stops the timer
comparisons.append((end-start)/2000) # append the average running time to "compar

start=time.time() # starts the timer
[three_way_merge_insertion_sort(list_of_lists[j]) for j in range(2000)] # sorts e
end=time.time() # stops the timer
comparisons.append((end-start)/2000) # append the average running time to "compar

print("The average running time of regular merge sort is:",comparisons[0])
print("The average running time of three-way-merge-sort is:",comparisons[1])
print("The average running time of the augmented three-way-merge-sort is:",compar
```

```
The average running time of regular merge sort is: 0.006582396626472473
The average running time of three-way-merge-sort is: 0.006206901669502258
The average running time of the augmented three-way-merge-sort is: 0.0061829655
17044068
```

In [105]:
```
from IPython.display import Image
Image("Table.JPG")
```

Out[105]:

Algorithm	Time complexity	Complexity	Merging complexity with number of comparisons
Two-way merge sort	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	O (n log2(n))	T (n log 2)
Three-way merge sort	$T(n) = 3T\left(\frac{n}{3}\right) + O(n)$	O (n log3(n))	T (n log 3)
Augmented three-way merge sort	$T(n) = 3T\left(\frac{n}{18}\right) + O(n) + O(6n)$	O (6n+nlog18(n))	

Figure 2: A table comparing the complexity of the three algorithms.

Complexity: All of the three algorithms above are recursive algorithms that use a divide-and-conquer approach. Their complexity will depend on their depth as well as the amount of work that needs to be done at each level (and at the very last level).
For the three-way merge sort, there are n/3 levels and at each level there are 3 lists. The time-complexity of the algorithm therefore is
$T(n) = 3T(n/3) + O(n) -> O(nlog3(n))$

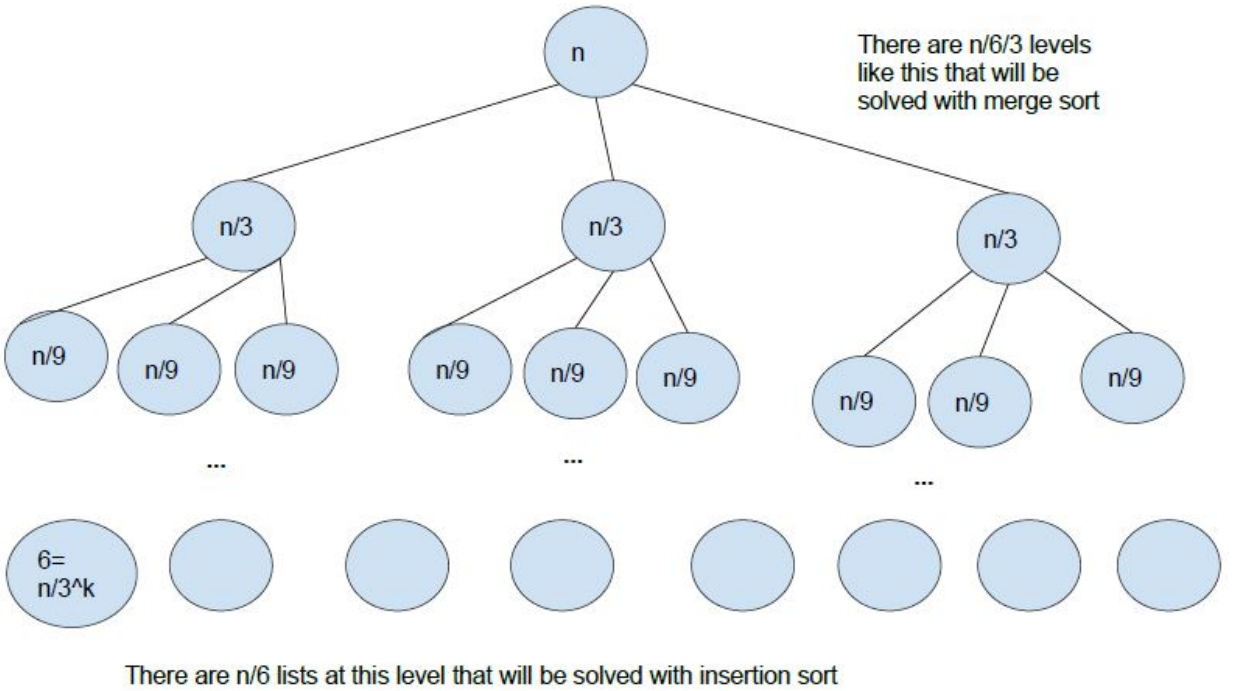
Similarily, the time complexity of a basic merge sort is
$T(n) = 2T(n/2) + O(n) -> O(nlog2(n))$
This would suggest that the three-way merge sort is better than the two-way merge sort, however it does not consider the time it takes for comparisons to be executed. Pracically, the three-way merge sort should only be better for short lists and the two-way merge sort is generally preferred.
The three_way_insertion_sort algortithm reduces the depth of the algorithm. After that level, it follows the complexity of insertion sort (which O(n^2)). At first, this seems slower, but for very short lists, insertion sort actually performs better than merge sort.
To find the complexity, we have to add the new complexity of the merge sort at it's new depth to the complexity of the insertion sort. To visualize this better, refer to the graph below.
New three-way-merge-sort complexity: It still needs O(n) on each level to merge them back together, but we are effectively dividing n into groups of 6, so $T(n) = 3T(n/6/3) + O(n)$.
Therefore, the new complexity of the merge sort part is $O(nlog18(n))$. To this we add the complexity of the insertion sort. It will have n/6 lists to sort and a complexity of O(n) in the best case and O(n^2) in the worst case. for lists of length 6 this is 6^2 operations. The total complexity is therefore $O(6n + nlog18(n))$

The insertion sort can augment merge sort because it has a lower running time for lower values of n. The results of my experiments confirm this hypothesis.

Figure 3:visual representation of the augmented merge sort steps

In [106]:
```
from IPython.display import Image
Image("diagram.JPG")
```

Out[106]:



Appendix

HC-Applications:
Algorithms: In this assignment, I designed an algorithm that I was able to implement a working algorithm for k-way merge sort, which I commented well.
Responsibility: In section 3, I took the challenge to work on a more difficult task that I completed to the best of my ability, even though it was not required. I still finished the assignment in time and put a lot of work into it.

In [ ]: