Assignment 1 - CS110

Tuan Tran - Spring 2018

Part 1: Implementation

In [1]:
```
#test case
import random,time
lis =[random.randint(1,100000) for i in range(100000)]
```

Normal merge sort

In [1]:
```
def mergeSort(array):
    if len(array)>1:
        #slicing the list if its length is greater than 1
        mid = len(array)//2
        lefthalf = array[:mid]
        righthalf = array[mid:]

        mergeSort(lefthalf) #recursion call on the left half
        mergeSort(righthalf) #recursion call on the right half

        #merge
        i=0 #initial index of first half
        j=0 #initial index of second half
        k=0 #initial index of the merged list

        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                array[k]=lefthalf[i]
                i=i+1
            else:
                array[k]=righthalf[j]
                j=j+1
            k=k+1

        #add the remaining of the left half to the merged list
```

```
        while i < len(lefthalf):
            array[k]=lefthalf[i]
            i=i+1
            k=k+1

        #add the remaining of the right half to the merged list
        while j < len(righthalf):
            array[k]=righthalf[j]
            j=j+1
            k=k+1

#try with simple list
sim_lis = [3,5,1,12,45,2,4,7]
mergeSort(sim_lis)
sim_lis
```

Out[1]: `[1, 2, 3, 4, 5, 7, 12, 45]`

In [11]:
```
start_time = time.time()
mergeSort(lis)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.5263962745666504 seconds ---
```

Three-way merge sort

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

In [2]:
```
def three_mergeSort(array):
    if len(array)>1:
        #dividing the list if its length is greater than 1
        mid1 = len(array)//3
        mid2 = len(array)*2//3

        left = array[:mid1] #left subarray
        mid = array[mid1:mid2] #middle subarray
        right = array[mid2:] #right subarray

        #recurssion call on each subarray
        three_mergeSort(left)
        three_mergeSort(mid)
        three_mergeSort(right)

        #Merging
```

```
        i=0 #index of left subarray
        j=0 #index of mid subarray
        l=0 #index of right subarray
        k=0 #index of the merged list

        #set the sentinel value indicating the end of a list
        sentinel = float('inf')
        left.append(sentinel)
        mid.append(sentinel)
        right.append(sentinel)

        """
        Iterating through each subarray until all three subarrays only hav
e
        the sentinel value left. At each iteration, find the smallest valu
e
        among current index of 3 subarrays and add it to the merged array.
        """
        while left[i] < sentinel or right[l] <sentinel or mid[j] <sentinel
:
            value, smallest_from = min([(left[i], "left"), (mid[j], "mid"
), (right[l],"right")])
            # print(value, smallest_from)
            if smallest_from == "left":
                array[k] = left[i]
                i +=1
            elif smallest_from == "mid":
                array[k] = mid[j]
                j +=1
            elif smallest_from == "right":
                array[k] = right[l]
                l +=1
            k+=1

#try with simple list
sim_lis = [3,5,1,12,45,2,4,7]
three_mergeSort(sim_lis)
sim_lis
```

Out[2]: [1, 2, 3, 4, 5, 7, 12, 45]

In [5]:
```
start_time = time.time()
three_mergeSort(lis)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.9565722942352295 seconds ---
```

Augmented merge 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 [3]:
```
threshold = 9
#Insertion sort
def insertionSort(mylist):
    for j in range(1,len(mylist)):
        key = mylist[j]
        i = j -1

        while i>=0 and mylist[i]> key:
            mylist[i+1]=mylist[i]
            i = i-1
            #step +1

        mylist[i+1]=key
    return mylist

def aug_mergeSort(array):
    if len(array)> threshold:
        #dividing the list if its length is greater than the threshold
        mid1 = len(array)//3
        mid2 = len(array)*2//3

        left = array[:mid1] #left subarray
        mid = array[mid1:mid2] #middle subarray
        right = array[mid2:] #right subarray

        #recurssion call on each subarray
        aug_mergeSort(left)
        aug_mergeSort(mid)
        aug_mergeSort(right)

        #Merging
        i=0 #index of left subarray
        j=0 #index of mid subarray
        l=0 #index of right subarray
        k=0 #index of the merged list

        #set the sentinel value indicating the end of a list
```

```
        sentinel = float('inf')
        left.append(sentinel)
        mid.append(sentinel)
        right.append(sentinel)

        """
        Iterating through each subarray until all three subarrays only hav
e
        the sentinel value left. At each iteration, find the smallest valu
e
        among current index of 3 subarrays and add it to the merged array.
        """
        while left[i] < sentinel or right[l] <sentinel or mid[j] <sentinel
:
            value, smallest_from = min([(left[i], "left"), (mid[j], "mid"
), (right[l],"right")])
           # print(value, smallest_from)
            if smallest_from == "left":
                array[k] = left[i]
                i +=1
            elif smallest_from == "mid":
                array[k] = mid[j]
                j +=1
            elif smallest_from == "right":
                array[k] = right[l]
                l +=1
            k+=1

    elif len(array) > 1:
    #call insertion sort if len(array) is below threshold
        insertionSort(array)

#try with simple list
sim_lis = [3,5,1,12,45,2,4,7]
aug_mergeSort(sim_lis)
sim_lis
```

Out[3]: [1, 2, 3, 4, 5, 7, 12, 45]

In [7]:
```
start_time = time.time()
aug_mergeSort(lis)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.7369606494903564 seconds ---
```

Part 2: Analysis

Analyze and compare the practical run times of regular merge sort, three-way merge sort, and the augmented merge sort. Use the tools developed in class to run experiments. Your results should be presented in a table, along with an explanatory paragraph and any useful graphs or other charts. Part of your analysis should indicate whether or not there is a "best" variation.

In theory, normal merge sort recursively breaks down an array into 2 sub-arrays half its size. The recurrence relation of normal merge sort is $T(n) = 2T(n/2) + O(n)$. In three-way merge sort, we break down an array into 3 sub-arrays one-third its size. The recurrence relation of this algorithm is $T(n) = 3T(n/3) + O(n)$. In general, if we ignore the low-order term and the constant c, the running time of both algorithms is $\Theta(n \log n)$. To be more accurate, the recursion tree of three-way merge sort may have fewer levels than normal merge sort, since we are using log base 3. However, at each level the number of sub-problems will be higher. Regarding the augmented version, the running time depends on the threshold. The trade-off is that while we decrease the height trees by using fewer divisions, at the bottom level the running time will be linear or exponential depends on the input because of using insertion sort.

Experimenting with 3 sorting algorithms using a random list of size 100000 results in the following running time.

Sorting Algorithms	Normal	Three-way	Augmented
Running Time (s)	0.52	0.95	0.73

To perform a more rigorous test, I run these algorithms with lists of different sizes and plot the running time accordingly. The figure below shows that normal merge sort has the best running time and three-way merge sort has the worst. As input size increases, the discrepancy also grows bigger. The augmented merge sort performs nearly as good as the normal version.

In [8]:
```
time1 = [] #running time of normal merge sort
time2 = [] #running time of three-way merge sort
time3 = [] #running time of augmented merge sort

the_range = [10**x for x in range (6)]
for n in the_range:
    """
    At each iteration n, create an random list of size 10^1 to 10^5.
    Run different sorting algorithms and record their running time.
    """
```

```
    lis = [random.randint(1,100000) for i in range(n)]

    start_time = time.time()
    mergeSort(lis)
    time1.append((time.time() - start_time))

    start_time = time.time()
    three_mergeSort(lis)
    time2.append((time.time() - start_time))

    start_time = time.time()
    aug_mergeSort(lis)
    time3.append((time.time() - start_time))
```

In [14]:

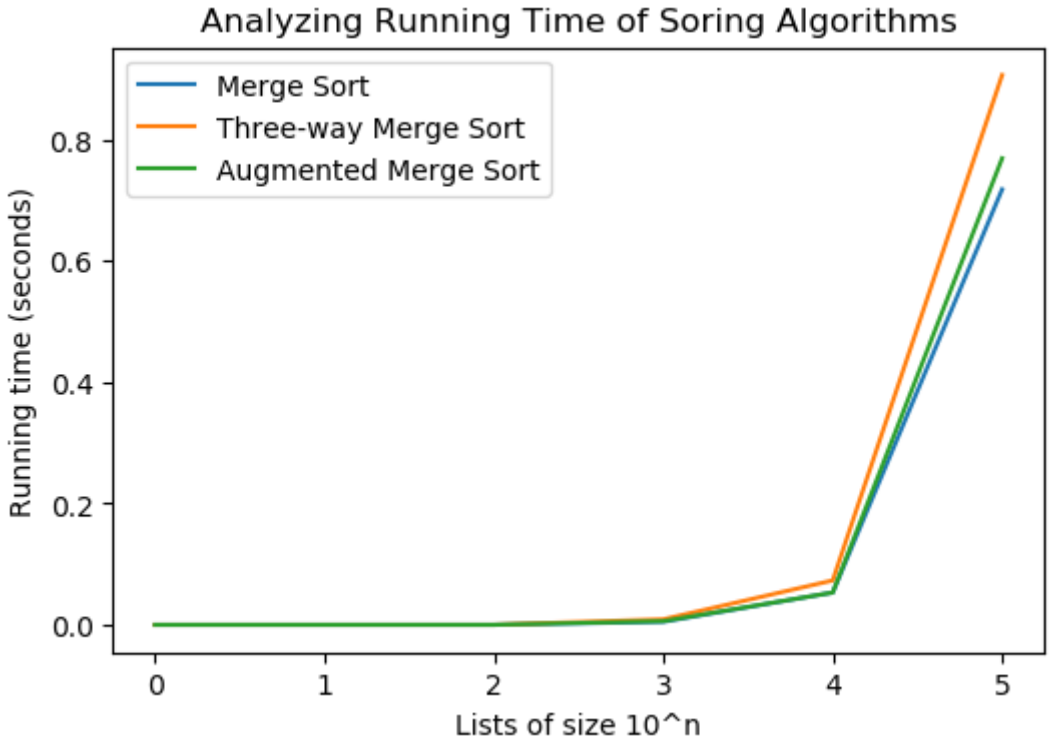
```
from pylab import figure, show
from matplotlib import pyplot as plt

step = [i for i in range(6)]
fig = plt.figure(figsize=(6, 4), dpi=100)
plt.plot(step, time1, label = "Merge Sort")
plt.plot(step, time2, label = "Three-way Merge Sort")
plt.plot(step, time3, label = "Augmented Merge Sort")

#labels
plt.xlabel("Lists of size 10^n")
plt.ylabel("Running time (seconds)")

#title
plt.title("Analyzing Running Time of Soring Algorithms")

plt.legend()
plt.show()
```


Analyzing Running Time of Soring Algorithms
Merge Sort
Three-way Merge Sort
Augmented Merge Sort
Running time (seconds)
0.0
0.2
0.4
0.6
0.8
0
1
2
3
4
5
Lists of size 10^n