In [1]:
```
def insertionSort(alist):
    for index in range(1,len(alist)):
        currentvalue = alist[index]
        position = index
        while position>0 and alist[position-1]>currentvalue:
            alist[position]=alist[position-1]
            position = position-1
            alist[position]=currentvalue
    return(alist)
```

In [2]:
```
def merge2(left, right):
    if not len(left) or not len(right):
        return left or right

    result = []
    i, j = 0, 0
    while (len(result) < len(left) + len(right)):
        if left[i] < right[j]:
            result.append(left[i])
            i+= 1
        else:
            result.append(right[j])
            j+= 1
        if i == len(left) or j == len(right):
            result.extend(left[i:] or right[j:])
            break

    return result

def mergesort_2(array):
    if len(array) < 2:
        return array
    else:
        middle = len(array)/2
        left = mergesort_2(array[:middle])
        right = mergesort_2(array[middle:])
    return merge2(left,right)
```

In [3]:
```
def merge3(left, middle, right):
    array = []
    i=0
    j=0
    k=0
    left.append(float("inf"))
    middle.append(float("inf"))
    right.append(float("inf"))
    #to avoid comparison to check if any list is empty
    while len(array)<(len(left)+len(middle)+len(right))-3:
        array.append(min(left[i],middle[j],right[k]))
        if i<len(left) and j<len(middle) and k<len(right):
            if min(left[i],middle[j],right[k]) == left[i]:
                i += 1
            elif min(left[i],middle[j],right[k]) == middle[j]:
                j += 1
            elif min(left[i],middle[j],right[k]) == right[k]:
                k += 1
    return array
```

In [4]:
```
def merge_sort3(array):
    if len(array)<2:
        return array
    elif len(array)==2:
        return mergesort_2(array)
    else:
        divider3 = len(array)//3
        left = merge_sort3(array[:divider3])
        middle = merge_sort3(array[divider3:2*divider3])
        right = merge_sort3(array[2*divider3:])
    return merge3(left,middle,right)

print merge_sort3([1,4,4,3,3,4,5,6])
```

```
[1, 3, 3, 4, 4, 4, 5, 6]
```

In [5]:
```
def k_mergesort_3(array, k):
    r = len(array)
    divider3 = len(array)//3
    left = array[:divider3]
    middle = array[divider3:2*divider3]
    right = array[2*divider3:]
    if len(array)<2:
        return array
    elif len(array)<k:
        return insertionSort(array)
    else:
        left = k_mergesort_3(left,k)
        middle = k_mergesort_3(middle,k)
        right = k_mergesort_3(right,k)
    return merge3(left,middle,right)

print k_mergesort_3([1,5,6,8,9,56,32,56,8,5,4,65,8,7],6)
```

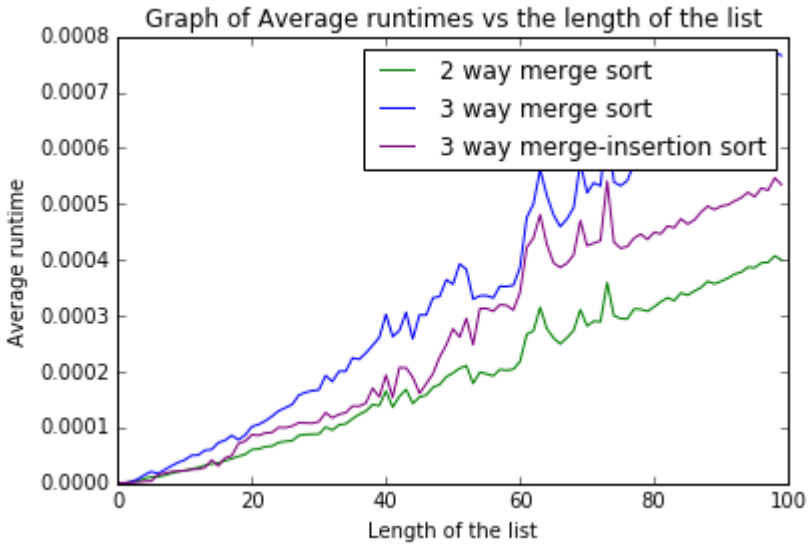
```
[1, 4, 5, 5, 6, 7, 8, 8, 8, 9, 32, 56, 56, 65]
```

In [14]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import random
import timeit
import plotly
import plotly.plotly as py
import plotly.figure_factory as ff
plotly.tools.set_credentials_file(username='mvhrt',api_key='usSJFqlImTcT5bteeLsC
def graphplt(length,k): #the max length of list we want and the threshhold after
    mergesort_2_avg=[]
    mergesort_3_avg=[]
    k_mergesort_3_avg=[]
    tablmrgesrt2 = []
    tablmrgesrt3 = []
    tablmrgesrt3k = []
    for n in range(length):
        mergesort_2_lst = []
        mergesort_3_lst = []
        k_mergesort_3_lst = []
        for i in range (1000): #Higher range/repetations = more accuracy.
            randomlist=random.sample(range(1000),n)
            start = timeit.default_timer()
            mergesort_2(randomlist)
            stop1 = timeit.default_timer()
            merge_sort3(randomlist)
            stop2 = timeit.default_timer()
            k_mergesort_3(randomlist,k)
            stop3 = timeit.default_timer()
            mergesort_2_lst.append(stop1-start)
            mergesort_3_lst.append(stop2-stop1)
            k_mergesort_3_lst.append(stop3-stop2)
        mergesort_2_avg.append(np.mean(mergesort_2_lst))
        mergesort_3_avg.append(np.mean(mergesort_3_lst))
        k_mergesort_3_avg.append(np.mean(k_mergesort_3_lst))

    tablmrgesrt2.append(np.mean(mergesort_2_avg))
    tablmrgesrt3.append(np.mean(mergesort_3_avg))
    tablmrgesrt3k.append(np.mean(k_mergesort_3_avg))
    lengths=range(length) #generates x-axis
    plt.plot(lengths, mergesort_2_avg, color="green", label="2 way merge sort")
    plt.plot(lengths, mergesort_3_avg, color="blue", label="3 way merge sort")
    plt.plot(lengths, k_mergesort_3_avg, color="purple", label="3 way merge-inse
    plt.title("Graph of Average runtimes vs the length of the list")
    plt.xlabel("Length of the list")
    plt.ylabel("Average runtime")
    plt.legend()
    plt.show()

    matrix = []
    print "m.s = mergesort, m.i = mergesort with insertion sort, (s) = time in s
    matrix.append(["Runtime of 2-way m.s (s)","Runtime of 3-way m.s (s).", "Runt
    matrix.append([ tablmrgesrt2,tablmrgesrt3,tablmrgesrt3k])
    matrix.append([ " "," "," "])
    table=ff.create_table(matrix)
    return py.iplot(table, filename='Runtime table')
```

```
graphplt(100,6)
```




```
m.s = mergesort, m.i = mergesort with insertion sort, (s) = time in seconds
```

Out[14]:

Runtime of 2-way m.s (s)	Runtime of 3-way m.s (s).	Runtime of 3-way m.i. sor
[0.00018950372345669168]	[0.0003498861787647048]	[0.00026233551012343469]

EDIT CHART

```
As we see from the graph and the table above, the two way mergesort
outperforms the three way mergesort and the mergesort with insertion sort
called when k = 6. As the elements in the list grow, the slope of the graph
increses. The spikes in the graps are the worst-case scenarios of the sorting
algorithms. But two way mergesort increases the time taken slowly, so with
increasing elements in the list, two way mergesort will out perform other
mergesort types. Hence a two way mergesort is the 'best' among the three
types of mergesort for sorting an unsorted list.
```