KATHMANDU UNIVERSITY

DHULIKHEL, KAVREPALANCHOWK

SUBJECT CODE: COMP 314

In the partial fulfilment of "Introduction to Sorting Algorithm-Insertion and Merge Sort"

Lab Report #2

Submitted To:

Rajani Chulyadyo, PhD

Department of Computer Science and Engineering (DoCSE)

Submitted By:

Abhay Raut
Roll No.:43
3rd Year, 2nd Semester
B.E. Computer Engineering

Submission Date: 13th May 2019

Click Here (link to github)

Lab 2: Sorting Algorithm (Insertion Sort and Merge Sort)

Insertion Sort Algorithm:

Features:

- It is efficient for smaller data sets, but very inefficient for larger lists. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
- It is better than Selection Sort and Bubble Sort algorithms.
- Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
- It is a **stable** sorting technique, as it does not change the relative order of elements which are equal.

Algorithm:

Step 1 − If it is the first element, it is already sorted. Return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the Value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

Pseudo Code:

```
procedureinsertionSort( A : array of items )
intholePosition
intvalueToInsert

fori=1 to length(A) inclusive do:

/* select value to be inserted */
valueToInsert= A[i]
holePosition=i

/*locate hole position for the element to be inserted */
whileholePosition>0and A[holePosition-1]>valueToInsertdo:
      A[holePosition]=A[holePosition-1]
holePosition=holePosition-1
endwhile
```

```
/* insert the number at hole position */
      A[holePosition]=valueToInsert
endfor

end procedure
```

Python Code:

```python
def insertionsort(arr):

    for i in range (1, len(arr)):
        currentnum = arr[i]
        j = i-1
        while j>=0 and currentnum < arr[j] :
            arr[j+1] = arr[j]
            j=j-1
        arr[j+1] = currentnum
    return arr

```

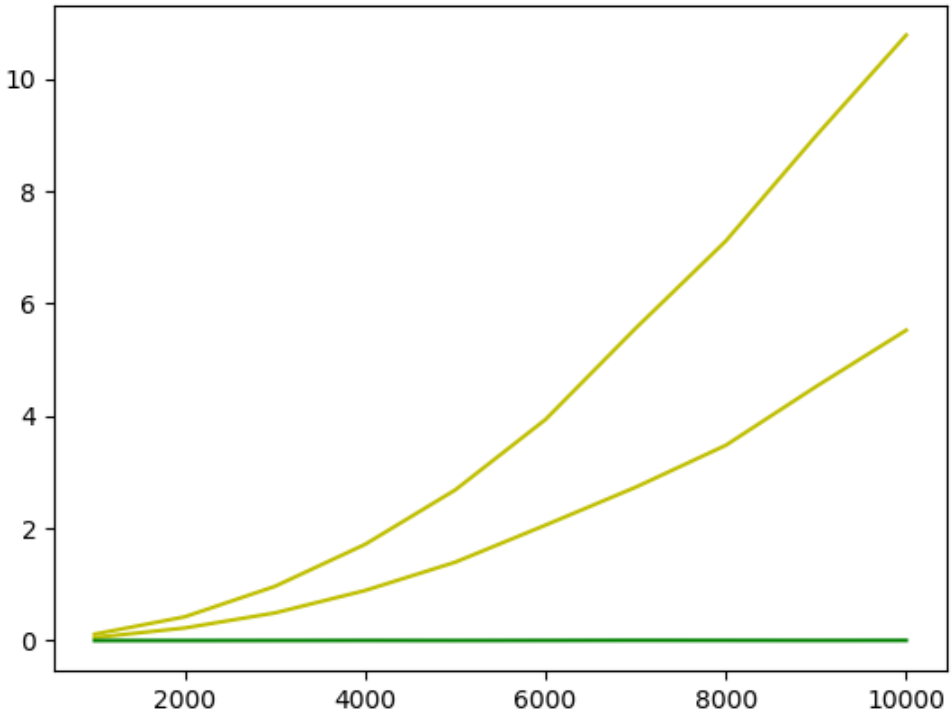
```python
from insertionsorting import insertionsort
import random
from time import time
import matplotlib.pyplot as plt

randomnum= random.sample(range(100000),10000)
sortednum=sorted(randomnum)
insertion_unsorted=[]
insertion_bestcase=[]
insertion_worstcase=[]
xdomain=[]
j=0
stepsize=1000
print("normal case scenario")
for i in range(1000,11000,stepsize):
    start_time=time()
    insertionsort(randomnum[0:i])
    end_time=time()
    print(end_time-start_time)
    insertion_unsorted.insert(j,end_time-start_time)
    xdomain.insert(j,i)
    j+=1
print("BEST CASE SCENARIO")
for i in range (1000,11000,stepsize):
    start_time=time()
    insertionsort(sortednum[0:i])
    end_time=time()
    print(end_time-start_time)
    insertion_bestcase.insert(j,end_time-start_time)
    j+=1
```

```
31  print("WORST CASE SCENARIO")
32  for i in range (1000,11000,stepsize):
33      start_time=time()
34      insertionsort(sortednum[i:0:-1])
35      end_time=time()
36      print(end_time-start_time)
37      insertion_worstcase.insert(j,end_time-start_time)
38      j+=1
39
40  print("*****************")
41  plt.plot(xdomain,insertion_unsorted,'y',label="Normal Case")
42  plt.plot(xdomain,insertion_bestcase,'g',label="Best Case")
43  plt.plot(xdomain,insertion_worstcase,'y',label="Worst Case")
44  plt.show()
45
```

Output:

Input Size	Normal Case	Best Case	Worst Case
1000	0.08395099639892578	0.0010006427764892578	0.1599104404449463
2000	0.2788405418395996	0.0019996166229248047	0.625640869140625
3000	0.6806111335754395	0.0019981861114501953	1.335254192352295
4000	1.147357702255249	0.0029969215393066406	3.163175344467163
5000	1.9129068851470947	0.003998756408691406	4.023701429367065
6000	2.683483600616455	0.001997232437133789	5.981588363647461
7000	4.000710964202881	0.0019989013671875	8.226300716400146
8000	5.708739519119263	0.0029990673065185547	10.973678588867188
9000	6.714166164398193	0.0049970149993896484	13.931021451950073
10000	8.52612566947937	0.0029964447021484375	17.432392835617065

Graph:



Interpretation of Observations:

From the above graph we observe that the time complexities for insertion sorting algorithm are:

For Normal and Worst Case scenario: O (n^2)

For Best Case scenario: O(n)

Merge Sort Algorithm:

Divide and Conquer in Merge Sort:

- **Divide** by finding the number q of the position midway between p and r Do this step the same way we found the midpoint in binary search: add p and r divide by 2, and round down.
- **Conquer** by recursively sorting the subarrays in each of the two sub problems created by the divide step. That is, recursively sort the subarray `array[p..q]` and recursively sort the subarray `array[q+1..r]`.
- **Combine** by merging the two sorted subarrays back into the single sorted subarray `array[p..r]`.

Algorithm:

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

Pseudo Code:

```
proceduremergesort(var a as array )
if( n ==1)return a

var l1 as array = a[0]... a[n/2]
var l2 as array = a[n/2+1]... a[n]

  l1 =mergesort( l1 )
  l2 =mergesort( l2 )

return merge( l1, l2 )
end procedure

procedure merge(var a as array,var b as array )

var c as array
while( a and b have elements )
if( a[0]> b[0])
add b[0] to the end of c
remove b[0]from b
else
add a[0] to the end of c
remove a[0]from a
endif
endwhile
```

```
while( a has elements )
add a[0] to the end of c
remove a[0]from a
endwhile

while( b has elements )
add b[0] to the end of c
remove b[0]from b
endwhile

return c

end procedure
```

Python Code:

```
def mergesort(alist):
    if len(alist)>1:
            mid = len(alist)//2
            lefthalf = alist[:mid]
            righthalf = alist[mid:]

            mergesort(lefthalf)
            mergesort(righthalf)

            i=0
            j=0
            k=0
            while i < len(lefthalf) and j < len(righthalf):
                if lefthalf[i] < righthalf[j]:
                    alist[k]=lefthalf[i]
                    i=i+1
                else:
                    alist[k]=righthalf[j]
                    j=j+1
                k=k+1

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

            while j < len(righthalf):
                alist[k]=righthalf[j]
                j=j+1
                k=k+1
    return alist
```

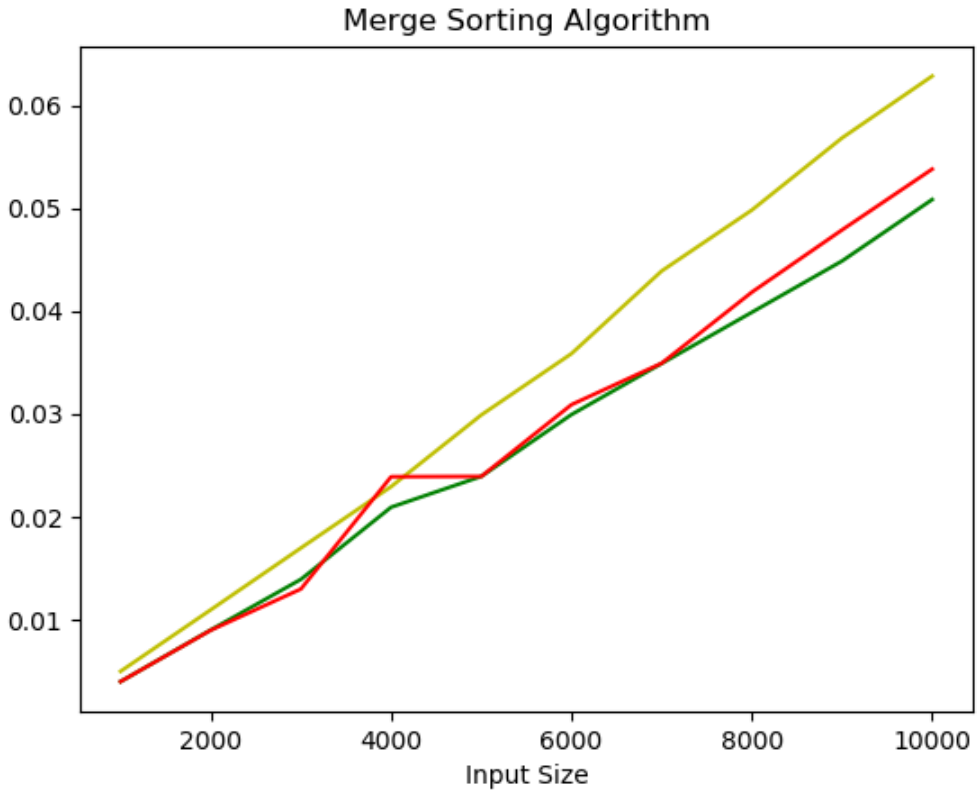
```
from mergesorting import mergesort
import random
from time import time
import matplotlib.pyplot as plt

randomnum=random.sample(range(100000),10000)
sortednum=sorted(randomnum)
merge_unsorted=[]
merge_bestcase=[]
merge_worstcase=[]
xdomain=[]
j=0
stepsize=1000
print("NORMAL CASE SCENARIO")
for i in range(1000,11000,stepsize):
    start_time=time()
    mergesort(randomnum[0:i])
    end_time=time()
    print(end_time-start_time)
    merge_unsorted.insert(j,end_time-start_time)
    xdomain.insert(j,i)
    j+=1

print("BEST CASE SCENARIO")
for i in range(1000,11000,stepsize):
    start_time=time()
    mergesort(sortednum[0:i])
    end_time=time()
    print(start_time-end_time)
    merge_bestcase.insert(j,end_time-start_time)
    j+=1
```

```
print("WORST CASE SCENARIO")
for i in range(1000,11000,stepsize):
    start_time=time()
    mergesort(sortednum[i:0:-1])
    end_time=time()
    print(end_time-start_time)
    merge_worstcase.insert(j,end_time-start_time)
    j+=1

print("***************")
plt.plot(xdomain,merge_unsorted,'y',label="Normal Case")
plt.plot(xdomain,merge_bestcase,'g',label="Best Case")
plt.plot(xdomain,merge_worstcase,'r',label="Worst Case")
plt.show()
```

Output:

Input Size	Normal Case	Best Case	Worst Case
1000	0.004985809326171875	-0.003988981246948242	0.003988742828369141
2000	0.010969161987304688	-0.009009599685668945	0.008980989456176758
3000	0.016980886459350586	-0.013930559158325195	0.012991189956665039
4000	0.02294468879699707	-0.02094292640686035	0.0239102840423584
5000	0.02991461753845215	-0.02393817901611328	0.02393937110900879
6000	0.035877227783203125	-0.029952287673950195	0.03091740608215332
7000	0.04391336441040039	-0.03491997718811035	0.03494453430175781
8000	0.049837350845336914	-0.03989601135253906	0.041852474212646484
9000	0.05684947967529297	-0.04488396644592285	0.047907114028930664
10000	0.06286859512329102	-0.05086350440979004	0.05382084846496582

Graph:



Interpretation of Observations:

From the above graph we observe that the time complexities for merge sorting algorithm are is O(n logn) for all the cases i.e. Normal, Best and Worst Case scenario.

Test Cases:

```python
import unittest
import random
from mergesorting import mergesort
from insertionsorting import insertionsort

number_of_test_cases=10000
class SearchTestCase(unittest.TestCase):

    def test_insertionsort(self):

        randomnum=random.sample(range(100000),number_of_test_cases)
        sortednum=sorted(randomnum)
        self.assertEqual(insertionsort(randomnum),sortednum)

    def test_mergesort(self):

        randomnum=random.sample(range(100000),number_of_test_cases)
        sortednum=sorted(randomnum)
        self.assertEqual(mergesort(randomnum),sortednum)

if __name__=='__main__':
    unittest.main()
```

Output: