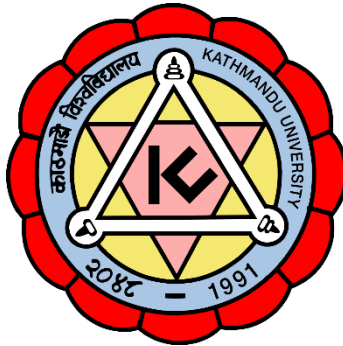


Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre



Algorithms and Complexity (COMP 314) – Lab 2

Submitted To:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submitted By:

Mission Shrestha(54)


CE-2019 Year/Sem: 3rd Year/2nd Sem

Submission Date: 25th April, 2023

Implementation, Testing and Performance measurement of Linear Search and Binary Search Algorithms.

Implementation


Insertion Sort Algorithm:




```
1  def insertion_sort(arr):
2      for i in range(1, len(arr)):
3          key = arr[i]
4          j = i - 1
5          while (j >= 0 and arr[j] > key):
6              arr[j + 1] = arr[j]
7              j = j - 1
8          arr[j + 1] = key
9
```

The array is split into sorted and unsorted portions using this technique. The unsorted array's key component is then inserted into the appropriate spot in the sorted portion of the array.

Merge Sort Algorithm:



```
1  from math import floor
2  from sys import maxsize
3
4
5  def merge_sort(arr, lb, rb):
6      if (lb < rb):
7          mid = floor((lb+rb)/2)
8          merge_sort(arr, lb, mid)
9          merge_sort(arr, mid+1, rb)
10         merge(arr, lb, mid, rb)
11     return arr
12
```



```

1
2
3 def merge(arr, lb, mid, rb):
4     n1 = mid-lb+1
5     n2 = rb-mid
6     # left_arr = [n1]
7     # right_arr = [n2]
8     left_arr = [0] * (n1+1)
9     right_arr = [0] * (n2+1)
10
11
12     for i in range(0, n1):
13         left_arr[i] = arr[lb+i]
14
15     for j in range(0, n2):
16         right_arr[j] = arr[mid+j+1]
17
18     left_arr[n1] = maxsize
19     right_arr[n2] = maxsize
20     # left_arr.append(maxsize)
21     # right_arr.append(maxsize)
22
23     i = 0
24     j = 0
25
26     for k in range(lb, rb+1):
27         if (left_arr[i] <= right_arr[j]):
28             arr[k] = left_arr[i]
29             i = i+1
30         else:
31             arr[k] = right_arr[j]
32             j = j+1
33

```

This data sorting technique is based on the divide and conquer strategy. A single element is present in each sub array of an array that is divided into n subarrays, where n is the length of the array. The separated sub arrays are then combined once more in sorted order.

Testing

Testing of Sorting Algorithm:

```
1  import unittest
2  from insertion_sort import insertion_sort
3  from merge_sort import merge_sort
4
5
6  class SearchTestCase(unittest.TestCase):
7      def test_insertion_sort(self):
8          required_output = [1, 2, 3, 4, 5]
9          input = [3, 2, 1, 5, 4]
10         insertion_sort(input)
11         self.assertEqual(input, required_output)
12
13         # input = [5, 4, 3, 2, 1]
14         # output = [1, 2, 3, 4, 5]
15         # insertion_sort(input)
16         # self.assertEqual(input, required_output)
17
18     def test_merge_sort(self):
19         required_output = [1, 2, 3, 4, 5]
20         input = [3, 2, 1, 5, 4]
21         merge_sort(input, 0, len(required_output)-1)
22         self.assertEqual(input, required_output)
23
24         # input = [5, 4, 3, 2, 1]
25         # output = [1, 2, 3, 4, 5]
26         # merge_sort(input)
27         # self.assertEqual(input, required_output)
28
29
30 if __name__ == '__main__':
31     unittest.main()
32
```


Output of Testing Both Sorts

```
PS E:\6th Sem\Algorithms\LabWorks\LW2> python -u "e:\6th Sem\Algorithms\LabWorks\LW2\tempCodeRunnerFile.py"
..
-----
Ran 2 tests in 0.001s

OK
PS E:\6th Sem\Algorithms\LabWorks\LW2> █
```

Performance

Code for generating random data and applying both Insertion and Merge Sort algorithm to check the time required.



```
1  from random import sample
2  from insertion_sort import insertion_sort
3  from merge_sort import merge_sort
4  from time import time_ns
5  from time import time
6
7
8  def run(n):
9      data = sample(range(1, n+1), n)
10     # print(data)
11     # start_time = time()
12     start_time = time_ns()
13     insertion_sort(data)
14     # sortedData = merge_sort(data, 0, n-1)
15     # print(sortedData)
16     # end_time = time()
17     end_time = time_ns()
18
19     time_taken = end_time-start_time
20     print(time_taken)
21
22
23  for i in range(1000, 10001, 1000):
24      run(i)
25  # for i in range(100000, 1000001, 100000):
26  #     run(i)
27
```

Output for Insertion Sort :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS E:\6th Sem\Algorithms\LabWorks\LW2> python -u "e:\6th Sem\Algorithms\LabWorks\LW2\main.py"
31253200
124958800
203114400
312428200
468637500
765794500
920928500
1186629400
1404727200
1841438700
○ PS E:\6th Sem\Algorithms\LabWorks\LW2> █
```

Graph for Insertion Sort :



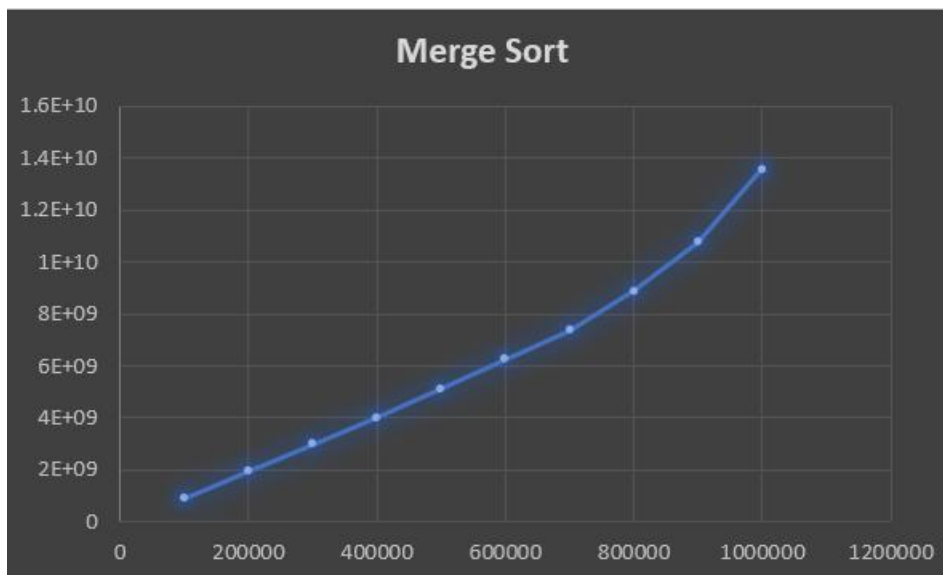
We can see from this graph that the Insertion Sort algorithm produces a quadratic route when showing the number of data vs. time graph. This indicates that the time complexity is $O(n^2)$.

Output for Merge Sort:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

● PS E:\6th Sem\Algorithms\LabWorks\LW2> python -u "e:\6th Sem\Algorithms\LabWorks\LW2\main.py"
913886500
1934203100
2985827400
4008611700
5114505200
6259242200
7367374200
8868400100
10749934300
13553468500
○ PS E:\6th Sem\Algorithms\LabWorks\LW2> █
```

Graph for Merge Sort:



We can see from this graph that the Merge Sort algorithm produces a somewhat linear route when showing the number of data vs. time graph. Its not exactly linear but close to the graph of $n \log n$. Hence, its time complexity is $O(n \log n)$.

Conclusion:

We implemented both Insertion Sort as well as Merge Sort. We checked if the algorithm were correct. We also tested some random data. Finally, we sketched Data Vs Time graph. We found the time complexity of Insertion sort to be $O(n^2)$ whereas $O(n \log n)$ in case of Merge Sort. Hence, we can conclude saying, Merge Sort works more efficiently in sorting huge volume of data compared to Insertion Sort.