Kathmandu University

Dhulikhel, Kavre

LAB-II
Merge Sort and Quick Sort
[Code No: COMP314]

Submitted by

Sudip Bhattarai (11)

Submitted to:-

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submission Date:

22nd MAY

Quick Sort

Quick sort is an internal sorting algorithm which works by divide and conquer method .It works by selecting one of the elements of array as pivot element and compares each element with pivot element to keep number smaller then pivot element to left and number larger the pivot element to right of an pivot element and finally the pivot element is placed in correct position and the process continues until whole array is sorted by repeated dividing the array into.

The time complexity of Quick sort is O(nlog(n)) in best case and $0(n^2)$ in worst case.

The code implementation of Quick sort is shown below:

```
def swap(arr,i,j):
    arr[i],arr[j]=arr[j],arr[i]

def partition(arr,p,r):
    pivot=arr[r]
    i=p-1
    for  j in range(p,r):
        if(arr[j]<=pivot):
            i+=1
            swap(arr,i,j)
    swap(arr,i+1,r)
    return  i+1

def quick_sort(arr,p,r):
    if(p<r):
        q=partition(arr,p,r)
        quick_sort(arr,p,q-1)
        quick_sort(arr,q+1,r)
```

Merge Sort

Merge sort is an external sorting algorithm which works by divide and conquer method This is because merge sort always divides the array into two halves and then merges them, ensuring consistent performance across a range of input sizes and conditions.

It works by dividing the array into two equal halves until it can no more be divided. Then each subarray is sorted individually using the merge sort algorithm and the sorted subarrays are merged back together in sorted order.

The time complexity of Merge sort is O(nlog(n)) in all cases.

The code implementation of Quick sort is shown below:

```
def merge_sort(arr,p,r):
    if(p<r):
        q=math.floor((p+r)/2)   #calculates the midpoint of the
array   merge_sort(arr,p,q)
        merge_sort(arr,q+1,r)
        merge(arr,p,q,r)
```

```
def merge(arr,p,q,r):
    n1=q-p+1
    n2=r-q
    #initailzing left and right array with all zeros
    L=[0]*(n1+1)
    R=[0]*(n2+1)
    for i  in range(0,n1):
        L[i]=arr[p+i]
    for j in range(0,n2):
        R[j]=arr[q+j+1]
    L[n1]=math.inf
    R[n2]=math.inf
    i=0
    j=0
    for k in range(p,r+1):
        if(L[i]<=R[j]):
            arr[k]=L[i]
            i+=1
        else:
            arr[k]=R[j]
            j+=1
```

Test Case

Here we have written the text cases for both sorting algorithm which include the 4 different cases:

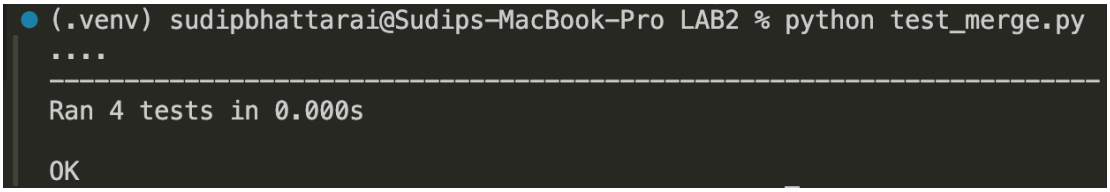
- Best case : Already sorted i.e. ascending order
- Word case: Reverse list i.e. descending order
- Normal Case: Random list
- Empty case: Empty list

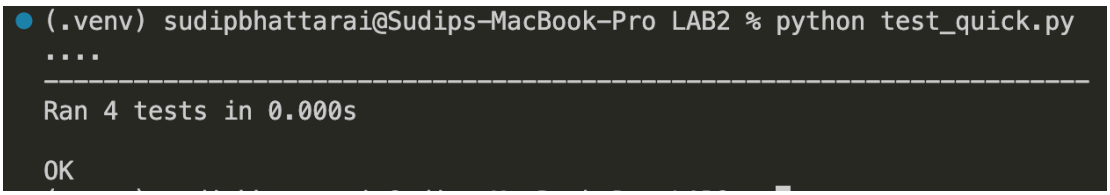
```python
import unittest
from merge import merge_sort

class TestMerge(unittest.TestCase):
    def test_descending(self):
        input_data=[5,4,3,2,1,8]
        merge_sort(input_data,0,len(input_data)-1)
        self.assertListEqual(input_data,[1,2,3,4,5,8])
    def test_ascending(self):
        input_data=[1,2,3,4,5]
        merge_sort(input_data,0,len(input_data)-1)
        self.assertListEqual(input_data,[1,2,3,4,5])

    def test_empty(self):
        input_data=[];
        merge_sort(input_data,0,len(input_data)-1)
        self.assertListEqual(input_data,[])
    def test_sorting(self):
        input_data=[1,10,5,2,3,8,6,9];
        merge_sort(input_data,0,len(input_data)-1)
        self.assertListEqual(input_data,
[1,2,3,5,6,8,9,10])
if __name__=='__main__':
    unittest.main()
```

```python
import unittest
from quick import quick_sort

class TestMerge(unittest.TestCase):
    def test_descending(self):
        input_data=[5,4,3,2,1,8]
        quick_sort(input_data,0,len(input_data)-1)
        self.assertListEqual(input_data,[1,2,3,4,5,8])
    def test_ascending(self):
        input_data=[1,2,3,4,5]
        quick_sort(input_data,0,len(input_data)-1)
        self.assertListEqual(input_data,[1,2,3,4,5])
    def test_empty(self):
        input_data=[];
        quick_sort(input_data,0,len(input_data)-1)
        self.assertListEqual(input_data,[])
    def test_sorting(self):
        input_data=[1,10,5,2,3,8,6,9];
        quick_sort(input_data,0,len(input_data)-1)
        self.assertListEqual(input_data,
[1,2,3,5,6,8,9,10])
if __name__=='__main__':
    unittest.main()
```

Test Result:

Here on running both of the file we have successfully passed all the test and the output can be seen below





Graph (Time Vs Array Size):

Here we have plotted the graph for complexity of both quick sort and merge sort for array of size increasing 10 at each iteration until the maximum array size. At each iteration the time is calculated before and after sorting and its difference is recorded in an array .
The code snippet can be seen below :

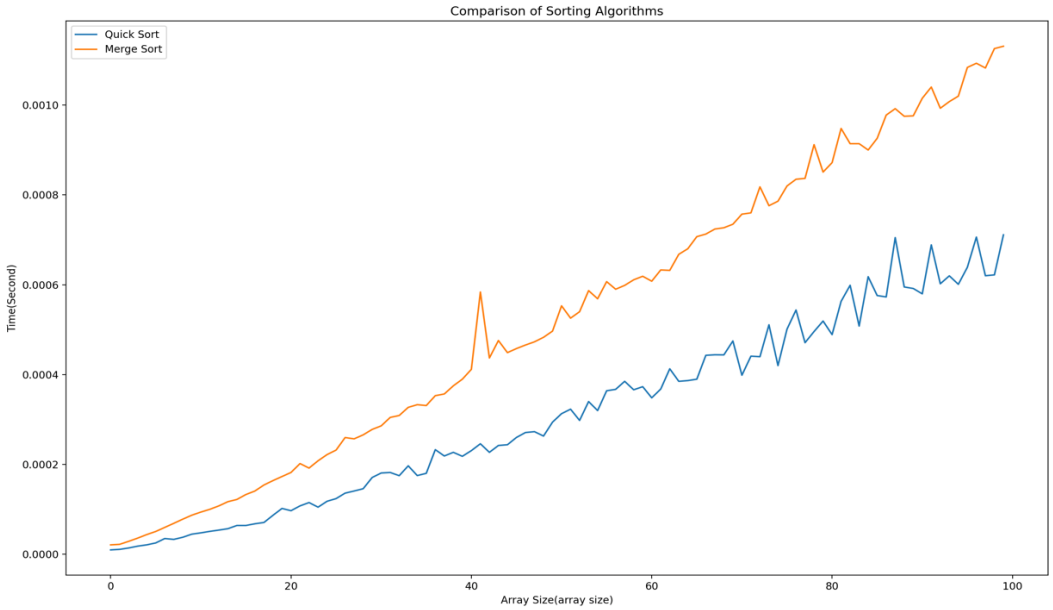
```
from quick import quick_sort
from merge import merge_sort
from timecalculation import cal_time,current_time
from randomarray import generate_random_array
import matplotlib.pyplot as plt
start_time = current_time()
time_array_insertion = []
time_array_selection = []
increment_size = 10
MAX_ARRAY_SIZE=500
for i in range(MAX_ARRAY_SIZE):
    increment_size += 10
    array = generate_random_array(0,increment_size)
    start_time = current_time()
    quick_sort(array,0,len(array)-1)
    end_time = current_time()
    time_array_insertion.append(cal_time(start_time, end_time))
    start_time = current_time()
    merge_sort(array,0,len(array)-1)
    end_time = current_time()
    time_array_selection.append(cal_time(start_time, end_time))
plt.plot(time_array_insertion, label='Quick Sort')
plt.plot(time_array_selection, label='Merge Sort')
plt.xlabel('Array Size(array size)')
plt.ylabel('Time(Second)')
plt.title('Comparison of Sorting Algorithms')
plt.legend()
plt.show()
```

```
import time

def current_time():
    return time.time()

def cal_time(start_time,end_time):
    result=end_time-start_time;
    return result
```

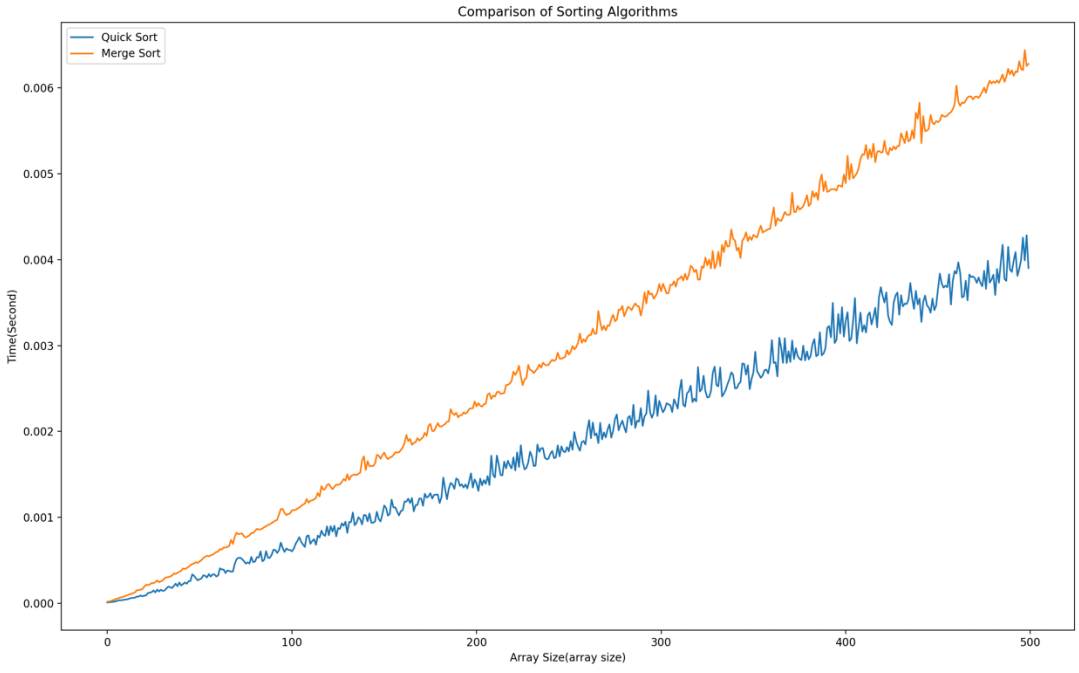
```
import random
def generate_random_array(start_range, end_range):
    if end_range == 0 or start_range >= end_range:
        return []
    return [random.randint(start_range, end_range) for _ in range(end_range-
start_range)]
```

The graph then obtained can be seen below for different range of array size:

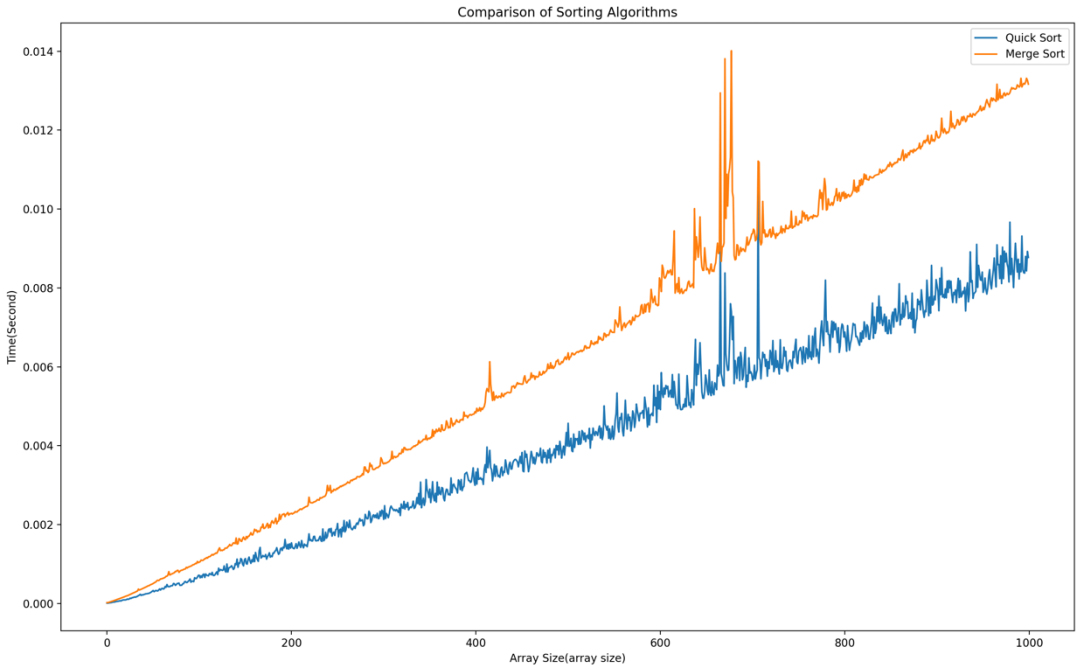
- Max Array Size(100):



- Max Array Size(500):



- Max Array Size(1000):



The graph above shows the performance curves of quick sort and merge sort. Initially, both algorithms perform similarly, but as the array size increases, there is difference in performance. Quick sort typically takes less time than merge sort when sorting the same data. Quick sort's average time complexity is $O(n\log n)$ and its worst-case time complexity is $O(n^2)$. The worst case scenario is when the pivot selection is poor, such as always choosing the smallest or largest element as the pivot in a sorted or reverse-sorted array. In contrast, merge sort has a consistent time complexity of O(nlogn) in both the best and worst cases. This is because merge sort always divides the array into two halves and then merges them, ensuring similar performance across a different size of input. When comparing these two sorting algorithms, quick sort is generally preferred for its average performance and lower overhead, whereas merge sort is more reliable and stable in terms of time complexity, particularly for large datasets.