PROJECT – 1

Comparison-based Sorting Algorithms

UNIVERSITY OF NORTH CAROLINA AT CHARLOTTE

DEPARTMENT OF INFORMATION TECHNOLOGY

ITCS-6114

Algorithms and Data Structures

Submitted By

HEMANTH CHOWDARY GADDIPATI

Student ID: 801169527

Email ID: hgaddipa@uncc.edu

1. INSERTION SORT:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm:

To sort an array of size n using Insertion Sort:

1: Iterate from array[0] to array[n] over the array.

2: Compare the current element to its most recent left element.

3: If the current element is smaller than its immediate left element then compare it with the elements from left to 0. Move the greater elements one position to the right.

Time Complexity: O(n*2) for large arrays and O(n) for sorted arrays.

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time O(n) when elements are already sorted.

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

CODE:

```
def insertionsort(array):
    # Loop from arr[1] to arr[n]
    for i in range(1, len(array)):
        current_element = array[i]
        # Now we iterate backwords from i to 0 and compare the values with the current element
        j = i-1
        while j >= 0 and current_element < array[j]:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = current_element
    return array
```

2. MERGE SORT

Merge sort is the sorting technique which based on the divide and conquer technique. Merge sort first divides the array into equal halves. Each sub array of size n/2 is sorted in a recursive manner in O(log(n)) time and then combines them in a sorted manner.

Algorithm:

1: Find the middle point to divide the array into two halves

2: Call merge Sort function for first half: Call merge Sort (array, 1, m)

3: Call merge Sort for second half: Call merge Sort (array, m+1, r)

4: Merge two halves sorted in step 2 and 3: Call merge (array, l, m, r)

Time Complexity: O(n*log(n)) for best, average and worst cases.

Uses: Merge Sort is useful for sorting linked lists in O(n*log(n)) time.

CODE:

```
def mergeSort(array):
    if len(array)>1:
        mid_pos = len(array)//2
        left = array[:mid_pos]
        right = array[mid_pos:]

        sorted_left = mergeSort(left)
        sorted_right = mergeSort(right)
        array =[]
        while len(sorted_left)>0 and len(sorted_right)>0:
            if sorted_left[0] < sorted_right[0]:
                array.append(sorted_left[0])
                sorted_left.pop(0)
            else:
                array.append(sorted_right[0])
                sorted_right.pop(0)
        for i in sorted_left:
            array.append(i)
        for i in sorted_right:
            array.append(i)
    #print("Time taken to execute Merge Sort is : ",datetime.now()-starttime,"\n")
    return array
```

3. HEAP SORT

Heap sort is a comparison-based sorting technique whose structure is of Tree type. The tree is based on Binary heap data structure. This sorting technique is similar to selection sort where I first find the maximum element and place the maximum element at the end. I repeat the heapify process for remaining elements.

Algorithm:

1: Build a max heap from the input data.
2: At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3: Repeat step 2 while size of heap is greater than 1.

Time Complexity: The Running time for heapify operation takes O(log(n)) which is the height of the binary tree. After that, a sorted array is created by repeatedly removing the largest element from the heap and adding the element to the array. In this way, Heapsort takes O(n*log(n)) time.

Uses: Priority Queues, Order Statistics

CODE:

```
def heapify(array, length, index):
    max_element = index
    left = 2 * index + 1
    right = 2 * index + 2
    # See if left child of root exists and is greater than root
    if left < length and array[index] < array[left]:
        max_element = left
    # See if right child of root exists and is greater than root
    if right < length and array[max_element] < array[right]:
        max_element = right
    if max_element != index:
        array[index],array[max_element] = array[max_element],array[index]
        heapify(array, length, max_element)

def heapSort(array):
    length = len(array)
    for i in range(length//2 - 1, -1, -1):
        heapify(array, length, i)
    for i in range(length-1, 0, -1):
        array[i], array[0] = array[0], array[i]
        heapify(array, i, 0)

    return array
```

4. QUICK SORT

Quicksort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways such as Always picking first element as pivot, always pick last element as pivot, Pick a random element as pivot, Pick median as pivot. The key process in quicksort is partition (). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Algorithm:

/* low --> Starting index, high --> Ending index */

quicksort(array[], low, high){if (low < high) {/* pi is partitioning index, array[pi] is now at right place */ pi = partition(array, low, high); quicksort(array, low, pi - 1); // Before pi quicksort (array, pi + 1, high); // After pi}}

Time Complexity: O(n^2) is the worst case and O(N*log(n)) is the average case complexity.

Uses: Quick Sort is a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

CODE:

```
def partition(array, first, last):
    if last - first > 0:
        pivot, left, right = array[first],first, last
        while left <= right:
            while array[left] < pivot:
                left += 1
            while array[right] > pivot:
                right -= 1
            if left <= right:
                array[left], array[right] = array[right], array[left]
                left += 1
                right -= 1
        partition(array, first, right)
        partition(array, left, last)

def quicksort(array):
    partition(array, 0, len(array) - 1)
    return array
```

5. MODIFIED QUICK SORT

I used median-of-three as pivot in Modified Quick sort. Which means, in this algorithm, I can pick the pivot by taking the median of left most, middle and the right most elements from the array. Compare three elements and swap these elements if necessary. If the input size is less than or equal to 15, Insertion sort algorithm is used.

CODE:

```python
def MedianOfThree(arr, left, right):
    mid = (left + right)//2
    if arr[right] < arr[left]:
        array[left],array[right] = array[right],array[left]
    if arr[mid] < arr[left]:
        array[left],array[mid] = array[mid],array[left]
    if arr[right] < arr[mid]:
        array[right],array[mid] = array[mid],array[right]
    return arr[mid]

def modifiedpartition(array, first, last):
    if last - first > 0:
        left, right = first, last
        pivot = MedianOfThree(array,first,last)
        while left <= right:
            while array[left] < pivot:
                left += 1
            while array[right] > pivot:
                right -= 1
            if left <= right:
                array[left], array[right] = array[right], array[left]
                left += 1
                right -= 1
        modifiedpartition(array, first, right)
        modifiedpartition(array, left, last)

def modified_quicksort(array):
    if len(array) <= 15:
        for i in range(1, len(array)):
            current_element = array[i]
            j = i-1
            while j >= 0 and current_element < array[j]:
                array[j + 1] = array[j]
                j -= 1
            array[j + 1] = current_element
    else:
        modifiedpartition(array, 0, len(array) - 1)

    return array
```

EXECUTION CODE:

Function to generate array of random numbers for a given range “n”.

```
from time import time
import random

def random_number_generator(n):
    array = set()
    while len(array) < n:
        array.add(random.randint(1,n))
    array = list(array)
    random.shuffle(array)
    return array
```

Executing for shuffled Input:

```
for i in input_size_of_array:
    n= i
    array = random_number_generator(n)

    starttime1 = time()
    res = quicksort(array)
    executiontime1 = (time()-starttime1)
    quicksorttime[n] = executiontime1

    random.shuffle(array)
    starttime2 = time()
    res = insertionsort(array)
    executiontime2 = (time()-starttime2)
    insertionsorttime[n] = executiontime2

    random.shuffle(array)
    starttime3 = time()
    res = mergeSort(array)
    executiontime3 = (time()-starttime3)
    mergesorttime[n] = executiontime3

    random.shuffle(array)
    starttime4 = time()
    res = heapSort(array)
    executiontime4 = (time()-starttime4)
    heapsorttime[n] = executiontime4

    random.shuffle(array)
    starttime5 = time()
    res = modified_quicksort(array)
    executiontime5 = (time()-starttime5)
    modifiedquicksorttime[n] = executiontime5
```

```
print("Insertion Sort runtimes =\n",list(insertionsorttime.values()),"\n")
print("Heap Sort runtimes = \n", list(heapsorttime.values()),"\n")
print("Merge Sort runtimes = \n", list(mergesorttime.values()),"\n")
print("Quick Sort runtimes = \n", list(quicksorttime.values()),"\n")
print("Modified Quick Sort runtimes = \n", list(modifiedquicksorttime.values()))
```

Executing for Sorted input:

```python
input_sorted_array = [10,50,100,500,1000,1500,2000,2500,3000]
for i in input_sorted_array:
    n= i
    array = random_number_generator(n)
    array.sort()

    starttime1 = time()
    res = quicksort(array)
    executiontime1 = (time()-starttime1)
    sortedquicksorttime[n] = executiontime1

    starttime2 = time()
    res = insertionsort(array)
    executiontime2 = (time()-starttime2)
    sortedinsertionsorttime[n] = executiontime2

    starttime3 = time()
    res = mergeSort(array)
    executiontime3 = (time()-starttime3)
    sortedmergesorttime[n] = executiontime3

    starttime4 = time()
    res = heapSort(array)
    executiontime4 = (time()-starttime4)
    sortedheapsorttime[n] = executiontime4

    starttime5 = time()
    res = modified_quicksort(array)
    executiontime5 = (time()-starttime5)
    sortedmodifiedquicksorttime[n] = executiontime5
```

Executing for Reversely Sorted input:

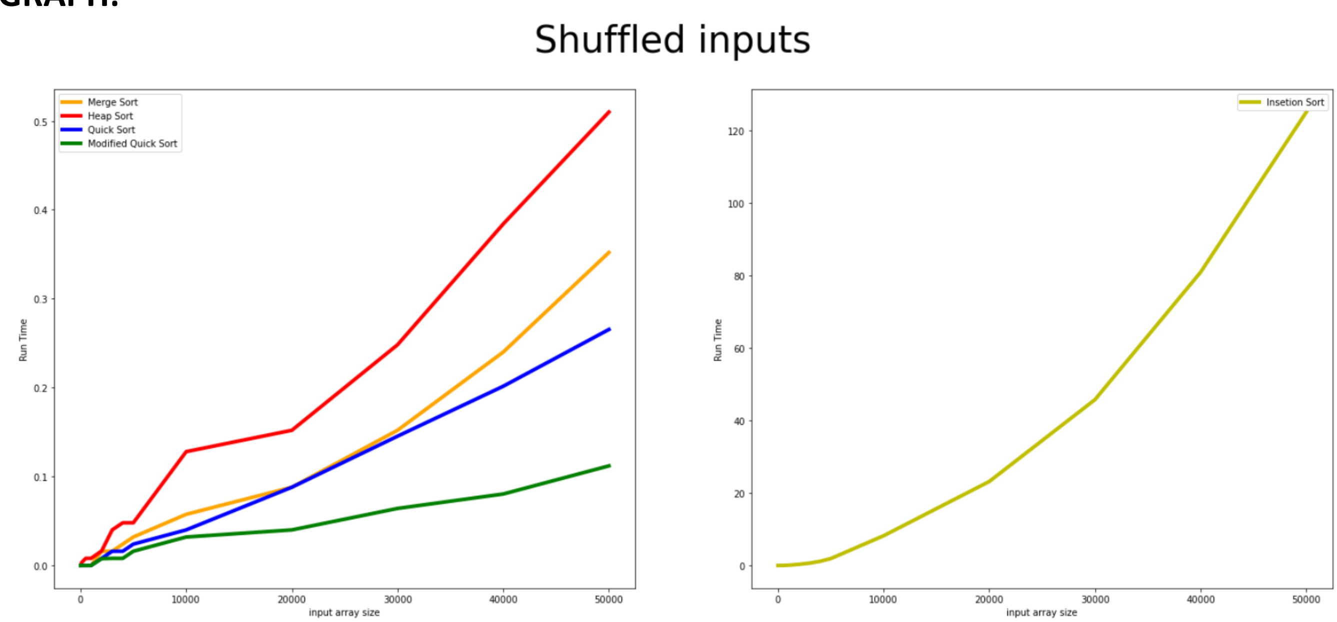
```python
input_sorted_array = [10,50,100,500,1000,1500,2000,2500,3000]
for i in input_sorted_array:
    n= i
    array = random_number_generator(n)
    array.sort(reverse = True)

    starttime1 = time()
    res = quicksort(array)
    executiontime1 = (time()-starttime1)
    reversesortedquicksorttime[n] = executiontime1

    starttime2 = time()
    res = insertionsort(array)
    executiontime2 = (time()-starttime2)
    reversesortedinsertionsorttime[n] = executiontime2

    starttime3 = time()
    res = mergeSort(array)
    executiontime3 = (time()-starttime3)
    reversesortedmergesorttime[n] = executiontime3

    starttime4 = time()
    res = heapSort(array)
    executiontime4 = (time()-starttime4)
    reversesortedheapsorttime[n] = executiontime4

    starttime5 = time()
    res = modified_quicksort(array)
    executiontime5 = (time()-starttime5)
    reversesortedmodifiedquicksorttime[n] = executiontime5
```

ANALYSIS FOR SHUFFLED INPUT: (Time in Seconds)

Len(array)	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Modified Quick Sort
1000	0.08133721351623535	0.008001327514648438	0.008000612258911133	0.0030663013458251953	0.008000612258911133
2000	0.2960543632507324	0.01600170135498047	0.01598334312438965	0.008000373840332031	0.01602005958557129

3000	0.7516045570373535	0.02401876449584961	0.03998589515686035	0.007996559143066406	0.007983684539794922
4000	1.2575297355651855	0.03200364112854004	0.03998541831970215	0.015999555587768555	0.016002178192138672
5000	1.8544583320617676	0.04802370071411133	0.047986507415771484	0.015982866287231445	0.016000986099243164
10000	6.502464532852173	0.06249356269836426	0.07813596725463867	0.048005104064941406	0.015613794326782227
20000	18.00629734992981	0.14401602745056152	0.16801834106445312	0.04687905311584473	0.04800534248352051
30000	40.652886390686035	0.24803924560546875	0.24802756309509277	0.09374856948852539	0.08000850677490234
40000	79.38648629188538	0.41416168212890625	0.3279118537902832	0.12801313400268555	0.16954922676086426
50000	136.04107475280762	0.5430471897125244	0.472414493560791	0.16002917289733887	0.15645718574523926

GRAPH:



Observations:

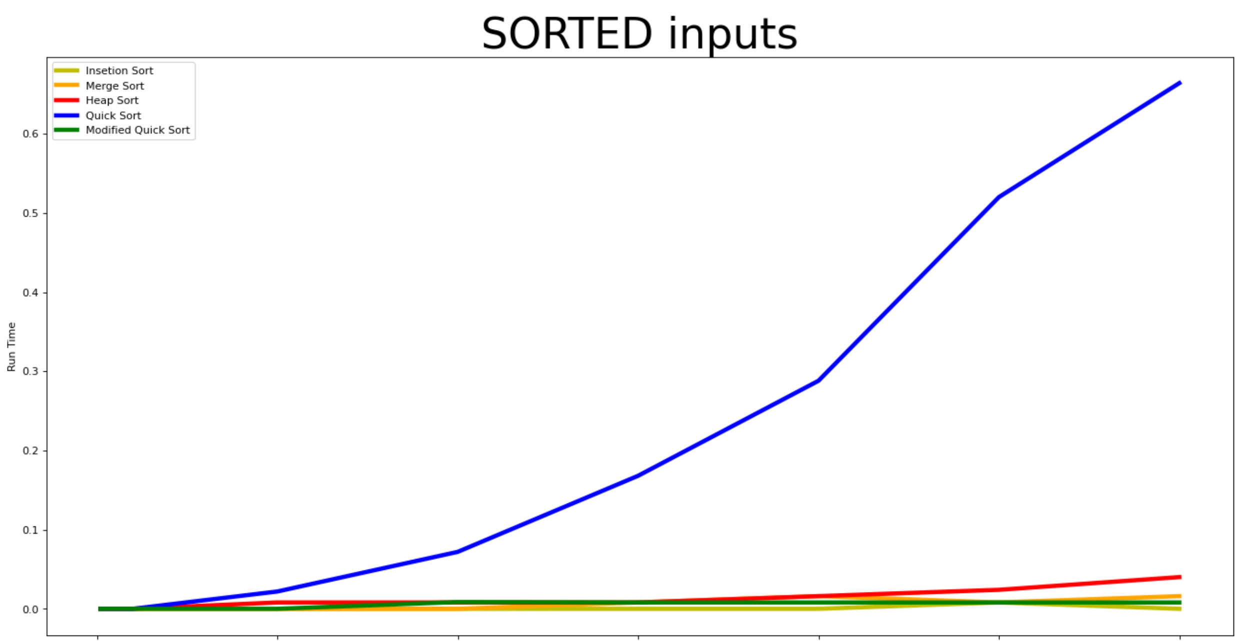
- The quick sort algorithm running time depends on the selection of pivot in the given random data. If pivot is around the median value, less time is taken, or else algorithm takes more time.
- In our case modified quick sort took more time as the size of the input increases.
- Time taken by Insertion Sort also increased as the input size increases.
- Time taken by Merge Sort and heap sort is less if the input is random data.

ANALYSIS FOR SORTED INPUT: (Time in Seconds)

```
print("Insertion Sort runtimes =\n",list(sortedinsertionsorttime.values()),"\n")
print("Heap Sort runtimes = \n", list(sortedheapsorttime.values()),"\n")
print("Merge Sort runtimes = \n", list(sortedmergesorttime.values()),"\n")
print("Quick Sort runtimes = \n", list(sortedquicksorttime.values()),"\n")
print("Modified Quick Sort runtimes = \n", list(sortedmodifiedquicksorttime.values()))
```

```
Insertion Sort runtimes =
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.007995128631591797, 0.0]

Heap Sort runtimes =
 [0.0, 0.0, 0.0, 0.008006095886230469, 0.008000850677490234, 0.007999897003173828, 0.016002416610717773, 0.023990869522094727, 0.0400240421295166]

Merge Sort runtimes =
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.008001089096069336, 0.01598525047302246, 0.008000612258911133, 0.015999794006347656]

Quick Sort runtimes =
 [0.0, 0.0, 0.0, 0.021834850311279297, 0.07188296318054199, 0.1679990291595459, 0.28805065155029297, 0.520054817199707, 0.664074182510376]

Modified Quick Sort runtimes =
 [0.0, 0.0, 0.0, 0.0, 0.008541345596313477, 0.008001327514648438, 0.007997751235961914, 0.008001565933227539, 0.007982254028320312]
```

GRAPH:



ANALYSIS FOR REVERSLY SORTED INPUT: (Time in Seconds)

```
print("Insertion Sort runtimes =\n",list(reversesortedinsertionsorttime.values()),"\n")
print("Heap Sort runtimes = \n", list(reversesortedheapsorttime.values()),"\n")
print("Merge Sort runtimes = \n", list(reversesortedmergesorttime.values()),"\n")
print("Quick Sort runtimes = \n", list(reversesortedquicksorttime.values()),"\n")
print("Modified Quick Sort runtimes = \n", list(reversesortedmodifiedquicksorttime.values()))
```

```
Insertion Sort runtimes =
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Heap Sort runtimes =
 [0.0, 0.0, 0.0, 0.008009195327758789, 0.008199214935302734, 0.016001224517822266, 0.016002893447875977, 0.024003982543945312, 0.039994001388549805]

Merge Sort runtimes =
 [0.0, 0.0, 0.0, 0.008006811141967773, 0.008000373840332031, 0.0, 0.015984535217285156, 0.008001089096069336, 0.016013383865356445]

Quick Sort runtimes =
 [0.0, 0.0, 0.0, 0.013830423355102539, 0.07200312614440918, 0.16802167892456055, 0.2800486087799072, 0.4480552673339844, 0.7295355796813965]

Modified Quick Sort runtimes =
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.008000612258911133, 0.00800013542175293, 0.01135563850402832, 0.007999897003173828]
```

GRAPH: