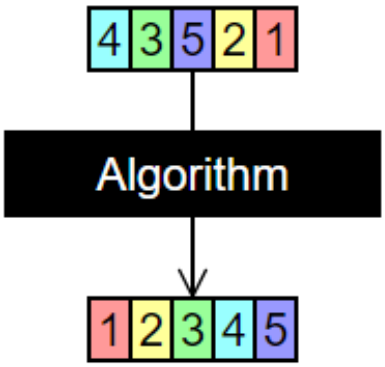
Comparison-based Sorting Algorithms



DOCUMENTATION REPORT

PROJECT 1

ITCS 6114 – Algorithm and Data Structure

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO

Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

Shreeya Gupta

801136226

Sai Kumar Thallada

801154715

UNC CHARLOTTE

College of Computing and Informatics

1. Introduction to Sorting Algorithms

A sorting Algorithm is used to rearrange a given array or list of elements in ascending or descending order using some comparison operator in their algorithm. There are various types of sorting algorithms that uses different approaches to sort a given sequence of numbers. The sorting algorithm takes a list of elements or array as input and performs various operations on the given list of elements and produces the output as a sorted array. There are various factors that needs to be considered while choosing the sorting algorithm.

Following is the list of sorting techniques that we have implemented:

1. Insertion sort
2. Merge sort
3. Heap sort [vector based, and insert one item at a time]
4. In-place quicksort (any random item or the first or the last item of your input can be pivot).
5. Modified quicksort
    - Use median-of-three as pivot.
    - For small sub-problem of size ≤ 10, use insertion sort.

I. Insertion Sort

In Insertion sort, the elements of the input array are inserted one after another in a proper sequence comparing with the rest of the elements. After every iteration the array is divided into two sub-arrays consisting of a sorted array (left) and an unsorted array (right). All the elements of the array are not visited all at once. The unsorted array elements are not visited in the start. As a new element is inserted in the array, it is compared with every element from right to left in the sorted sub array until the element smaller than the new element is found. The comparison will stop as the element is found which is smaller than the new element and the new element is placed in that position.

Time complexity:

In the worst case, all the elements of an array is compared with all the other elements of the array and thus for 'n' elements there will be n^2 comparisons. Thus the running time in worst case and average case is $O(n^2)$. For almost sorted arrays, the running time is O(n).

Code:

```
#------------------ InsertionSort --------------#
def insersionSort(array):
    for i in range(1, len(array)):
        j = i-1
        key = array[i]
        while j>=0 and array[j]>key:
            array[j+1] = array[j]
            j = j-1

    array[j+1]=key
#-----------------------------------------------#
```

II. Merge Sort

The Merge Sort algorithm works in divide and conquer approach. It follows three steps to sort a given list of elements such as:

i. Divide the array in two halves
ii. Repeatedly sort each half until there is only one element in the array
iii. Merge the two halves

While merging the two halves, first element of the two arrays are compared and the element that is smaller is inserted into the sorted array. This process is repeated until both the sub array is empty and a new sorted array is formed. If elements of any one of the sub array gets empty first then all the remaining elements of the other sub array is inserted directly at the end of the sorted array.

Time Complexity:

The list of size 'n' is divided into a max of log (n) parts, and the merging of all sub lists into a single list takes O(n) time, the worst case run time of this algorithm is O(n log n).

Code:

```
#------------------- Merge Sort ----------------#
def merge_sort(arr):
    if len(arr)>1:
        m = len(arr)//2
        leftArr = arr[:m]
        rightArr = arr[m:]

        merge_sort(leftArr)
        merge_sort(rightArr)

        i=0
        j=0
        k=0
        while i < len(leftArr) and j < len(rightArr):
            if leftArr[i] < rightArr[j]:
                arr[k]=leftArr[i]
                i=i+1
            else:
                arr[k]=rightArr[j]
                j=j+1
            k=k+1

        while i < len(leftArr):
            arr[k]=leftArr[i]
            i=i+1
            k=k+1

        while j < len(rightArr):
            arr[k]=rightArr[j]
            j=j+1
            k=k+1
    return (arr)
#-----------------------------------------------------------#
```

III. Heap Sort

The Heap sort algorithm is much faster than insertion sort and selection sort. The main property of a vector based heap sort is that if the parent node is at position i then the left child of that node will be in position 2i and its right child will be at location 2i+1. At the time of insertion, the program checks if there are any elements already present in the heap. The root node in a heap is always the smallest value of the entire array. The property of heap is that the value of root node is always smaller than its child. When inserting a value one by one into the heap, the heap property gets disturbed and to retain the heap order, heapify is done.

Time complexity:

Heapify has complexity O(log n) , Maxheap has complexity O(n) and we run max_heapify n-1 times in heap_sort function, therefore complexity of heap_sort function is O(n log n).

Code:

```
# Heap Sort-----------------------------------------------#
mainHeap=[]
arrSize=0
sortedArr=[]

def create_heap(array):
    global mainHeap
    mainHeap = [0] * (len(array)+1)
    for i in range(0,len(array)):
        insert(array[i]);

def insert(x):
    global arrSize
    arrSize = arrSize +1
    y=arrSize
    mainHeap[y]=x
    bubble_up(y)

def bubble_up(pos):

    PID = pos // 2;

    CID = pos;
    while (CID > 0 and mainHeap[PID] > mainHeap[CID]):

        swap(CID, PID);
        CID = PID;
        PID = PID // 2;


def swap(a,b):
    temp = mainHeap[a];
    mainHeap[a] = mainHeap[b];
    mainHeap[b] = temp;
```

```
def heap_sort1(array):
    create_heap(array);


def extract_min():
    global arrSize
    min = mainHeap[1]
    mainHeap[1]=mainHeap[arrSize]
    mainHeap[arrSize] = 0
    sinkDown(1)
    arrSize=arrSize -1
    return min

def sinkDown(k):
    small = k

    LCID = 2 * k
    RCID = 2 * k + 1
    if (LCID < arrSize  and mainHeap[small] > mainHeap[LCID]):
        small = LCID

    if (RCID < arrSize  and mainHeap[small] > mainHeap[RCID]):
        small = RCID

    if (small != k):
        swap(k, small)
        sinkDown(small)



def heap_sort(array):
    sortedArr=[0]*len(array)
    heap_sort1(array)
    for i in range(0,len(array)):
        sortedArr[i]=extract_min()

    return sortedArr
#-----------------------------------------------------------#
```

IV. Quick Sort

This sorting algorithm is also based on the divide and conquer approach. It reduces the space complexity and removes the use of auxiliary array used in the Merge Sort. In this sorting technique, a pivot is selected at random. After selecting the pivot there are three steps to be followed:

i. Divide the list of elements into three different arrays:
    a. Elements less than the pivot
    b. Elements equal to the pivot
    c. Elements greater than the pivot
ii. Sort the array with elements less than pivot an array with elements greater than pivot
iii. Join the three arrays

The execution of quick sort is depicted by a binary tree.

Time complexity:

If the array is already sorted quick sort runs in $O(n^2)$. In most cases, the expected running time of quick sort algorithm is O(n log n).

A. In place Quick Sort

In place quick sort algorithm uses a single array to sort the elements. It rearranges the elements of the input sequence in such a way that the elements less than the pivot have a rank less than the pivot and the elements greater than the pivot have a rank greater than the pivot.

Recursive calls are made on elements with rank less than and greater than the pivot. Two indexes traverse the array, one from left and the other from right comparing each of the elements until the indexes cross each other.

Code:

```
# In Place Quick Sort ------------------------------------#
import random

def partition(arr, low, high):
    i = (low - 1)

    pivot = arr[random.randint(low,high)]

    for j in range(low, high):

        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return (i + 1)


def quickSort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)

        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)

def quick_sort(numbers):
    arr = numbers
    n = len(arr)
    quickSort(arr, 0, n - 1)
    return arr
#-----------------------------------------------------------#
```

B. Modified Quick Sort

When choosing the pivot, if the pivot is a median then the running time decreases. But selecting a median is a difficult task in an unsorted array. So the "Median of three rule" states that from the leftmost, middle and rightmost element select the one with the median key as the pivot. Comparing just the leftmost, middle and rightmost elements, picking the pivot as the median of three.

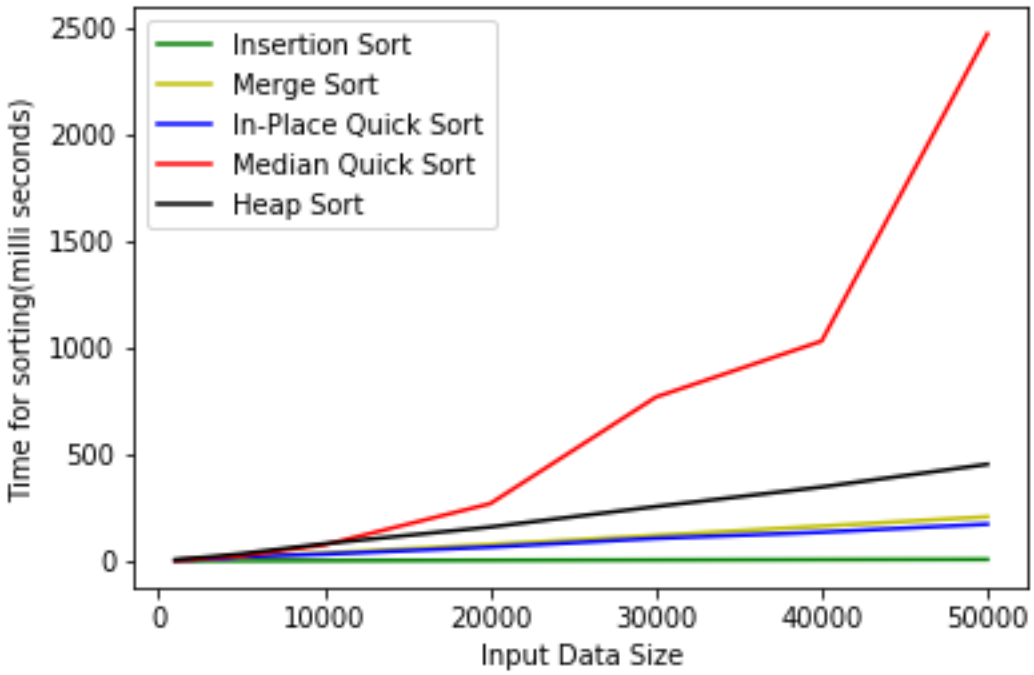
Code:

```
#----------- Median Quick Sort -------------------------#
medianC = 0
def median(a, b, c):
    if ( a - b) * (c - a) >= 0:
        return a

    elif (b - a) * (c - b) >= 0:
        return b

    else:
        return c


def partition_median(array, smallValArr, highValArr):
    small = array[smallValArr]
    high = array[highValArr - 1]
    length = highValArr - smallValArr
    middle = array[smallValArr + length // 2]

    pivot = median(small, high, middle)

    pivotindex = array.index(pivot)

    array[pivotindex] = array[smallValArr]
    array[smallValArr] = pivot

    i = smallValArr + 1
    for j in range(smallValArr + 1, highValArr):
        if array[j] < pivot:
            temp = array[j]
            array[j] = array[i]
            array[i] = temp
            i += 1

    highEndVal = array[smallValArr]
    array[smallValArr] = array[i - 1]
    array[i - 1] = highEndVal
    return i - 1
```

```
def quicksort_median(array, smallIndex, highIndex):
    global medianC
    if smallIndex+ 10  <= highIndex:
        newpivotindex = partition_median(array, smallIndex, highIndex)

        medianC += (highIndex - smallIndex - 1)
        quicksort_median(array, smallIndex, newpivotindex)

        quicksort_median(array, newpivotindex + 1, highIndex)

    else:
        insertion_sortt(array,smallIndex,highIndex)

def insertion_sortt(array,a,b):
    for i in range(a, b):
        j = i
        while j > 0 and array[j] < array[j-1]:
            array[j],array[j-1]=array[j-1],array[j]
            j = j - 1

def mquick_sort(inputArr):
    quicksort_median(inputArr, 0, len(inputArr))
    return inputArr
#---------------------------------------------------------------#
```

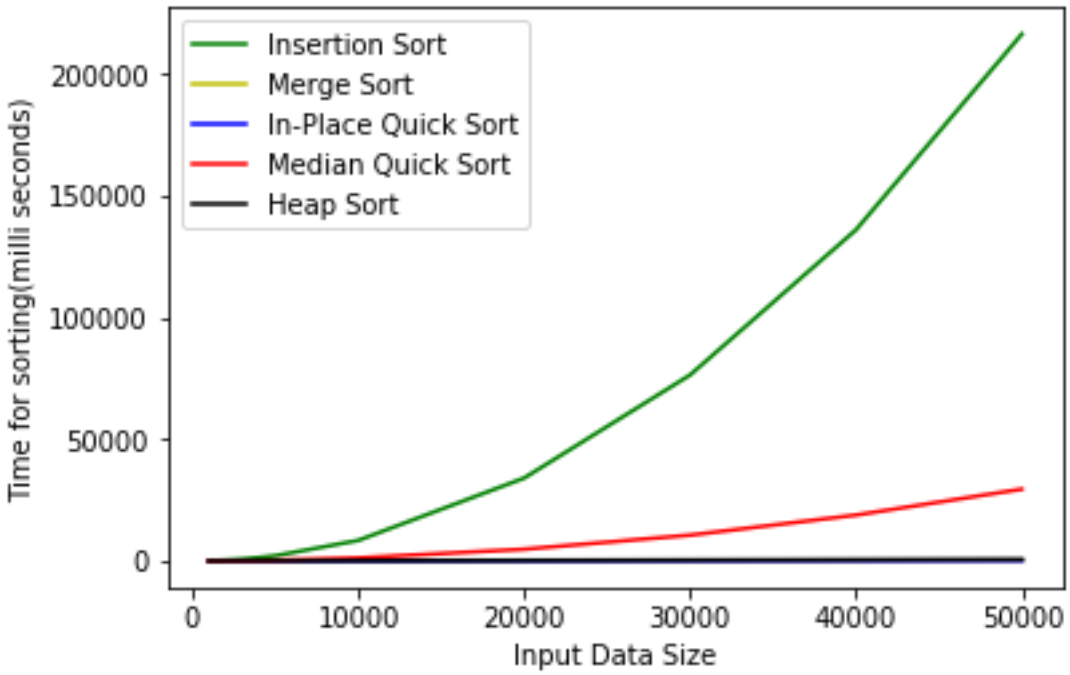
Sorted array Output in milliseconds:

Array size:	Insertion Sort:	Merge Sort:	Heap Sort:	InplaceQuick Sort:	MedianQuick Sort:
1000	0.18954277	2.991278966	5.250136058	2.743244171	1.85918808
2000	0.266869863	5.978902181	11.89017296	5.419015884	5.376895269
3000	0.393390656	9.423494339	18.54006449	8.697032928	13.37234179
4000	0.542322795	13.02925746	25.75826645	11.39052709	17.25069682
5000	0.65056483	17.18711853	33.41166178	19.00458336	23.67417018
10000	1.357952754	36.25146548	80.35588264	31.33789698	74.94584719
20000	2.809127172	76.93401972	158.4997972	64.52377637	267.9437002
30000	4.040559133	119.2208926	254.505078	105.5172284	767.3838933
40000	5.602757136	162.8739834	346.105655	134.2156728	1029.872656
50000	6.729761759	206.5128485	452.5176684	172.1003056	2469.109376



Reverse Sorted array output in milliseconds:

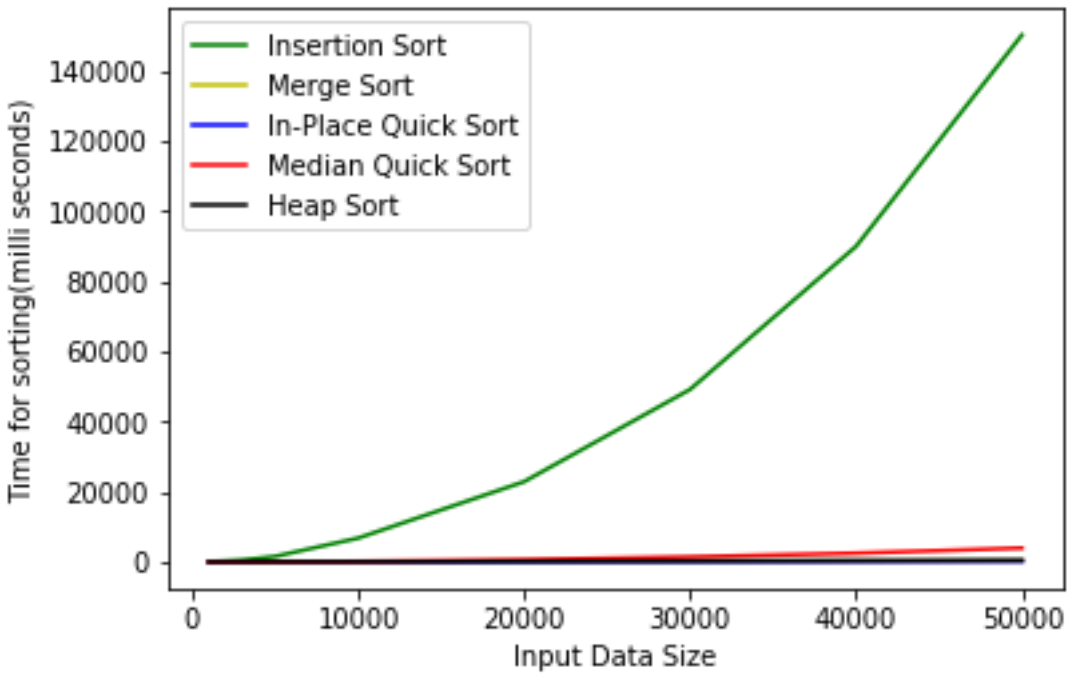
Array size:	Insertion Sort:	Merge Sort:	Heap Sort:	InplaceQuick Sort:	MedianQuick Sort:
1000	73.45629	2.796888	7.265806	2.520084	12.84838
2000	317.2843	6.96063	16.22629	5.402565	48.82526
3000	728.4181	9.789467	26.72696	8.63266	110.2536
4000	1350.157	12.96782	36.24821	11.21092	190.8247
5000	2074.383	16.98065	47.26076	14.48321	296.9656
10000	8380.769	36.03482	105.2105	29.44756	1185.933
20000	34001.41	77.31128	224.8278	66.04028	4768.8
30000	76372.8	119.4286	356.9193	98.50049	10553.59
40000	136035.5	164.9046	492.3162	128.6321	18800.31
50000	216511.4	209.2218	649.8585	164.4425	29468.87



Random Array output in milliseconds:

Array size:	Insertion Sort:	Merge Sort:	Heap Sort:	InplaceQuick Sort:	MedianQuick Sort:
1000	54.53761	3.367503	5.519708	2.827326	2.519766
2000	254.7602	9.426673	12.58898	7.571618	8.350929
3000	518.0411	11.71613	19.62233	10.62965	16.53353
4000	1004.729	16.08316	27.16231	12.17182	23.3651
5000	1551.549	22.11332	35.60702	15.43887	34.33347
10000	6757.569	44.85146	77.50249	30.91296	145.8484
20000	22835.28	98.1067	171.3867	67.52316	619.6049
30000	49106.64	158.6785	271.8091	103.82	1405.956
40000	89836.5	212.6394	377.9009	145.8716	2465
50000	150064.9	273.1417	488.3741	181.5033	3920.885

With Insertion sort:



Without Insertion Sort: