PROJECT 1

Comparison-based Sorting Algorithms

Algorithms and Data Structures

ITCS 6114/8114 - Fall 2021

UNIVERSITY OF NORTH CAROLINA AT CHARLOTTE

SUBMITTED BY:

SAI PRATHYUSHA POTU
Student Id: 801261324
Email id: spotu1@uncc.edu

THARANI KUMARESAN
Student Id: 801265490
Email id: tkumares@uncc.edu

1. INSERTION SORT:

Insertion sort is a simple and efficient comparison sort. In this algorithm, each iteration removes an element from the input data and inserts it into the correct position in the list being sorted. The choice of the element being removed from the input is random, and this process is repeated until all input elements have gone through.

Algorithm:

- Iterate through the array from array [0] to array[n].
- Compare the current element to the previous element on the left.
- Compare the items from left to zero if the current element is less than its immediate left element. Shift the larger parts to the right one position.

Time Complexity: O (n^2)

Auxiliary Space: O (1)

Code:

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

2. MERGE SORT:

Merge sorting is a sorting method based on the divide and conquer strategy. Merge sort divides the array into equal halves before sorting it. Each n/2 sub array is sorted in a recursive fashion in O(log(n)) time before being combined in a sorted manner.

Algorithm:

- To divide the array into two halves, find the center point.
- Call merge Sort function for first half: Call merge Sort (array, 1, m)
- Call merge Sort for second half: Call merge Sort (array, m+1, r)
- Merge two halves in step 2 and 3: Call merge (array, l, m, r)

Time complexity: O (nlogn)

Auxiliary Space: O (n)

Code:

```
def MergeSort(arr):
    if len(arr)>1:
        mid = len(arr)//2
        left = arr[:mid]
        right = arr[mid:]

        left_sorted = MergeSort(left)
        right_sorted = MergeSort(right)
        arr =[]
        while len(left_sorted)>0 and len(right_sorted)>0:
            if left_sorted[0] < right_sorted[0]:
                arr.append(left_sorted[0])
                left_sorted.pop(0)
            else:
                arr.append(right_sorted[0])
                right_sorted.pop(0)
        for i in left_sorted:
            arr.append(i)
        for i in right_sorted:
            arr.append(i)
    return arr
```

3. HEAP SORT:

Heap sorting is a comparison-based sorting approach with a Tree-like structure. The binary heap data structure is used to build the tree. This sorting method is similar to selection sorting, in which I discover the maximum element first and then place it at the end. For the remaining components, I repeat the heapify process.

Algorithm:

- Create a maximum heap using the provided data.
- The largest item is placed at the top of the stack at this point. Replace it with the heap's last item, then reduce the heap's size by one. Finally, heapify the tree's root
- Step 2 should be repeated as long as the size of the heap is bigger than 1.

Time complexity: O (nlogn)

Auxiliary complexity: O (1)

Code:

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

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

    return arr
```

4. QUICK SORT:

Quicksort algorithm uses Divide and Conquer method. It is also called partition exchange sort. It selects a pivot element and partitions the specified array around that pivot. There are a variety of quicksort versions that choose pivot in various ways, such as Always pick first element as pivot, Always pick last element as pivot, Pick a random element as pivot, and Pick median as pivot. Partition () is the most important operation in quicksort. Given an array and a pivot element p, the goal of partition is to place p in the correct position in the sorted array, with all smaller elements (less than p) placed before p and all larger elements (greater than p) placed after p. All of this should be completed in a linear manner.

Algorithm:

- If there are one or no elements in the array to be sorted, return.
- Pick an element in the array to serve as the "pivot" point. (Usually the left-most element in the array is used.)
- Split the array into two parts one with elements larger than the pivot and the other with elements
- Recursively repeat the algorithm for both halves of the original array.

Time complexity: O (nlogn)

Auxiliary Complexity: O (n)

Code:

```
def partition(arr, f, last):
    #f = first, l = left, r = right
    if last - f > 0:
        pivot, l, r = arr[f],f, last
        while l <= r:
            while arr[l] < pivot:
                l += 1
            while arr[r] > pivot:
                r -= 1
            if l <= r:
                arr[l], arr[r] = arr[r], arr[l]
                l += 1
                r -= 1
        partition(arr, f, r)
        partition(arr, l, last)

def QuickSort(arr):
    partition(arr, 0, len(arr) - 1)
    return arr
```

5. MODIFIED QUICK SORT:

In the Modified Quick sort, I utilized the median-of-three as the pivot. That is, in this approach, I can choose the pivot by taking the median of the array's leftmost, middle, and rightmost members. Compare three elements and, if necessary, switch them. Insertion sort is employed if the input size is less than or equal to 10.

Time complexity: O (nlogn)

Code:

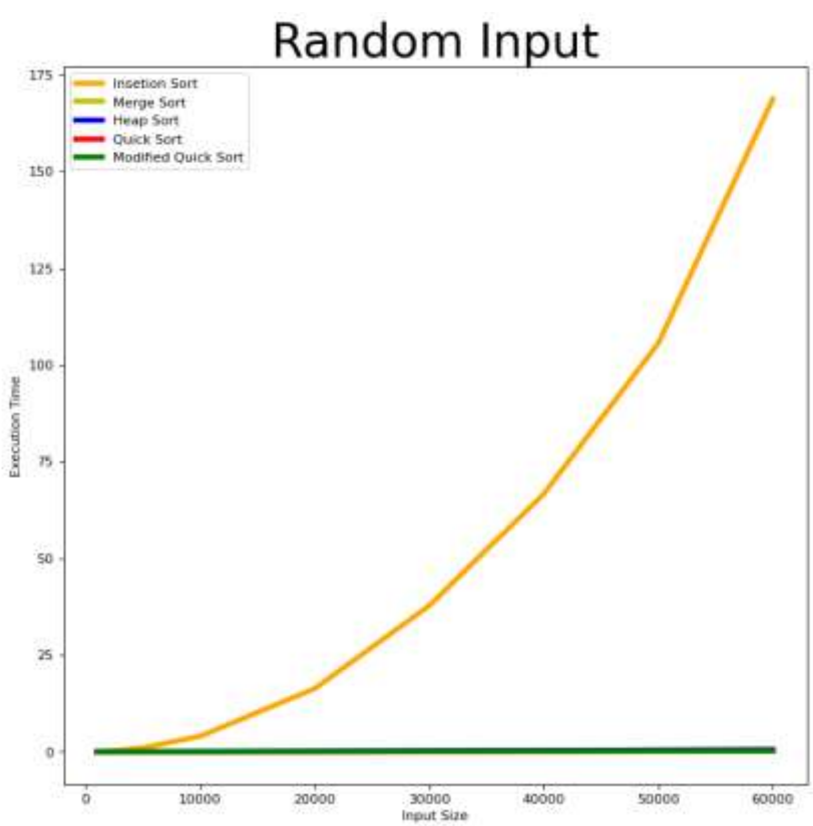
```
def MedianOfThree(array, l, r):
    #l = left, r = right
    mid = (l + r)//2
    if array[r] < array[l]:
        arr[l],arr[r] = arr[r],arr[l]
    if array[mid] < array[l]:
        arr[l],arr[mid] = arr[mid],arr[l]
    if array[r] < array[mid]:
        arr[r],arr[mid] = arr[mid],arr[r]
    return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

```python
def ModifiedPartition(arr, f, last):
    #f = first, l = left, r = right
    if last - f > 0:
        l, r = f, last
        pivot = MedianOfThree(arr,f,last)
        while l <= r:
            while arr[l] < pivot:
                l += 1
            while arr[r] > pivot:
                r -= 1
            if l <= r:
                arr[l], arr[r] = arr[r], arr[l]
                l += 1
                r -= 1
        ModifiedPartition(arr, f, r)
        ModifiedPartition(arr, l, last)

def ModifiedQuicksort(arr):
    if len(arr) <= 15:
        for i in range(1, len(arr)):
            key = arr[i]
            j = i-1
            while j >= 0 and key < arr[j]:
                arr[j + 1] = arr[j]
                j -= 1
            arr[j + 1] = key
    else:
        ModifiedPartition(arr, 0, len(arr) - 1)

    return arr
```

Analysis of Random Input:

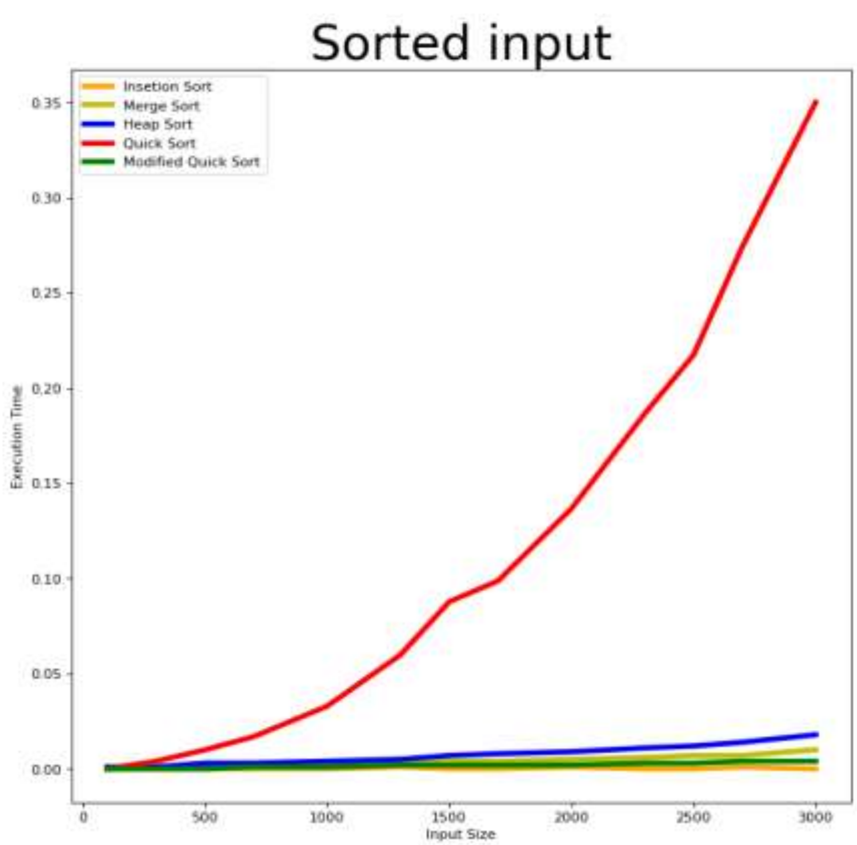
Time in seconds

Length of Array	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Modified Quick Sort
1000	0.03789854049682617	0.0029921531677246094	0.003989696502685547	0.0009965896606445312	0.0009973049163818336
2000	0.14860248565673828	0.006976127624511719	0.008975744247436523	0.00399017333984375	0.003966331481933594
3000	0.3430821895599365	0.010971784591674805	0.013993024826049805	0.005983829498291016	0.005984067916870117
5000	0.9773688316345215	0.02094411849975586	0.025930404663085938	0.010988712310791016	0.009973287582397461
10000	3.96239972114563	0.0468745231628418	0.05585122108459473	0.0219423770904541	0.02194046974182129
20000	16.285434246063232	0.11269712448120117	0.12865591049194336	0.049866676330566406	0.0468747615814209
30000	37.73115420341492	0.19449901580810547	0.21346044540405273	0.0797872543334961	0.07679557800292969
40000	66.64971375465393	0.2991974353790283	0.27227282524108887	0.10671329498291016	0.09773945808410645
50000	105.59825897216797	0.4089062213897705	0.3759937286376953	0.14261841773986816	0.1356368064880371
60000	168.74261951446533	0.5964062213897705	0.49268198013305664	0.20246171951293945	0.16954493522644043



Analysis of Sorted Input:

Length of Array	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Modified Quick Sort
100	0.0	0.0	0.000997304916381836	0.0	0.0
300	0.0	0.00099945068359375	0.000997304916381836	0.003987312316894531	0.0
500	0.0	0.0019948482513427734	0.0029916763305664062	0.00997471809387207	0.0
700	0.0	0.0019931793212890625	0.0030198097229003906	0.016927003860473633	0.0009970664978027344
1000	0.0	0.0029921531677246094	0.004006624221801758	0.03289294242858887	0.000997304916381836
1300	0.000997781753540039	0.0029916763305664062	0.004986763000488281	0.059841156005859375	0.001994609832763672
1500	0.0	0.003989458084106445	0.0069811344146728516	0.08776545524597168	0.0019943714141845703
1700	0.0	0.003989458084106445	0.007978677749633789	0.09873485565185547	0.0019943714141845703
2000	0.0009970664978027344	0.004986763000488281	0.008975744247436523	0.13663673400878906	0.0019936561584472656
2300	0.0	0.0059871673583984375	0.010967493057250977	0.18650102615356445	0.0030040740966796875
2500	0.0	0.006982088088989258	0.012001514434814453	0.21738481521606445	0.003016233444213867
2700	0.0009982585906982422	0.006981372833251953	0.013962745666503906	0.2742648124694824	0.0039899349212646484
3000	0.0	0.01002955436706543	0.017906904220581055	0.350053071975708	0.0039899349212646484



Analysis of Reverse Sorted Input:

Length of Array	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Modified Quick Sort
100	0.0009980201721191406	0.0	0.0009970664978027344	0.0	0.0
300	0.006981372833251953	0.0009984970092773438	0.0010099411010742188	0.0049746036529541016	0.0
500	0.018949508666992188	0.000997781753540039	0.0019943714141845703	0.032912254333496094	0.0009975433349609375
700	0.0359044075012207	0.0019948482513427734	0.0029916763305664062	0.01595759391784668	0.0009975433349609375
1000	0.06881523132324219	0.003023862838745117	0.003962993621826172	0.032906532287597656	0.0009980201721191406
1300	0.11768412590026855	0.0029921531677246094	0.005983829498291016	0.05884265899658203	0.000997304916381836
1500	0.16057133674621582	0.003994941711425781	0.006975650787353516	0.07679462432861328	0.002011537551879883
1700	0.22539830207824707	0.003988981246948242	0.007977724075317383	0.10471987724304199	0.002992868423461914
2000	0.29418110847473145	0.004987001419067383	0.008975505828857422	0.13464117050170898	0.0019948482513427734
2300	0.40491580963134766	0.005984783172607422	0.010986089706420898	0.18349409103393555	0.002991914749145508
2500	0.46475648880004883	0.006981849670410156	0.01296377182006836	0.2294158935546875	0.002988576889038086
2700	0.5684537887573242	0.0069811344146728516	0.013962745666503906	0.2593073844909668	0.003988504409790039
3000	0.7031190395355225	0.007978439331054688	0.01595759391784668	0.31914710998535156	0.0039882659912109375