ITCS 6144-Algorithms and Data Structures

Project 1

Comparison Based Sorting Algorithms

Submitted By

Srushti Khot(801203532)

Sanket Revadigar(801203510)

Instructor

Prof. Dewan T. Ahmed, Ph.D.

Project Overview:

In this project, the following comparison – based sorting algorithms are implemented. Performance is observed for different input sizes.

1. Insertion sort

2. Merge sort

3. Heapsort [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, must use insertion sort

Data Structure

Python programing language is used to implement the project.

Complexity Analysis:

1.Insertion Sort:

Insertion sort is a sorting algorithm that helps in building the final sorted list by placing unsorted elements one by one at its suitable place in each iteration. Holding a deck of cards is similar to Insertion sort. It is because of simplicity and low overhead it enjoys its usage.

Running time of Insertion sort:

Worst case (Reversely sorted inputs) :O(n^2)

Average case (Random inputs): O(n^2)

Best case (Sorted inputs) :O(n)

2.Merge Sort

Merge-sort is a sorting algorithm based on the divide-and-conquer rule. It accesses data in sequential manner.

It works as follows:

DIVIDE- Divide a problem into smaller sub problems.

Divide the data into two or more disjoint subsets if the input size is too large to deal with in a straightforward manner.

RECUR- Solve the sub problems .

Use divide and conquer to solve the sub-problems associated with the data subsets

CONQUER- Combine the solutions.

Take the solutions to the sub-problems and "merge" these solutions into a solution for the original problem.

Worst case (Reversely sorted inputs): O(nlogn)

Average case (Random inputs): O(nlogn)

Best case (Sorted inputs): O(nlogn)

3.Heap Sort

Heap-sort is much faster sorting technique than other algorithms such as insertion-sort and selection-sort. Binary heap data structure is used to perform comparison-based sorting algorithm. It can be considered as improved form of selection sort. Heapsort divides the input into a sorted and an unsorted region. Extract the largest element from it and insert it into the sorted region iteratively. The unsorted region is maintained in a heap data structure in order to find the largest element in each step quickly.

Worst case: O(nlogn)

Average case: O(nlogn)

Best case: O(nlogn)

4.

a. In-place Quick Sort

Quicksort is a divide-and-conquer algorithm. Pivot element is selected from the array. Other elements are partitioned into two sub-arrays by checking if they are less than or greater than the pivot element. The sub-arrays are sorted recursively .

The steps for in-place quicksort:

- If the range has less than two elements, return as there is nothing to do.
- Otherwise pick a pivot that is in the range.
- Partition the range.
- Recursively apply the quicksort to the sub-range.

Sorted inputs(worst): O(n^2)

Reversely sorted inputs(worst): O(n^2)

Average inputs(average): O(nlogn)

b. Modified Quick-sort

Choose the pivot element of the array as the median of the array. To determine the median it is necessary to find the middle element, after sorting the array which is takes O(nlog(n)) where n is the size of the array.

Worst case: O(nlog(n))

Average case: O(nlog(n))

Best case: O(nlog(n))

Code

1.Insertion Sort

```
def insertionSort(arr, min, max):
    for i in range(min, max + 1):
        key_ele = arr[i]
        j = i - 1
        while j >= min and key_ele < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key_ele
    return arr
```

2.Merge Sort

```
def merge_sort(arr):
    length= len(arr)
    if length==1:
        return arr
    middle =length//2
    l= merge_sort(arr[:middle])
    r = merge_sort(arr[middle:])
    return merge(l,r)


def merge(l, r):
    i=0
```

```
j=0
k=0

merge_array =[]
while(i<len(l) and j<len(r)):
   if l[i] <= r[j]:
      merge_array.append(l[i])
      i=i+1
   elif l[i] > r[j]:
      merge_array.append(r[j])
      j=j+1
   k=k+1

while(i<len(l)):
   merge_array.append(l[i])
   i+=1
   k+=1
while(j<len(r)):
   merge_array.append(r[j])
   j=j+1
   k=k+1
return merge_array
```

3.Heap Sort

```
def heapify(arr, n, i):
    maximum = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[maximum] < arr[left]:
        maximum = left
    if right < n and arr[maximum] < arr[right]:
        maximum = right
    if maximum != i:
        arr[i], arr[maximum] = arr[maximum], arr[i]

        heapify(arr, n, maximum)



def heapSort(arr):
    n = len(arr)

    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

```
    return arr
```

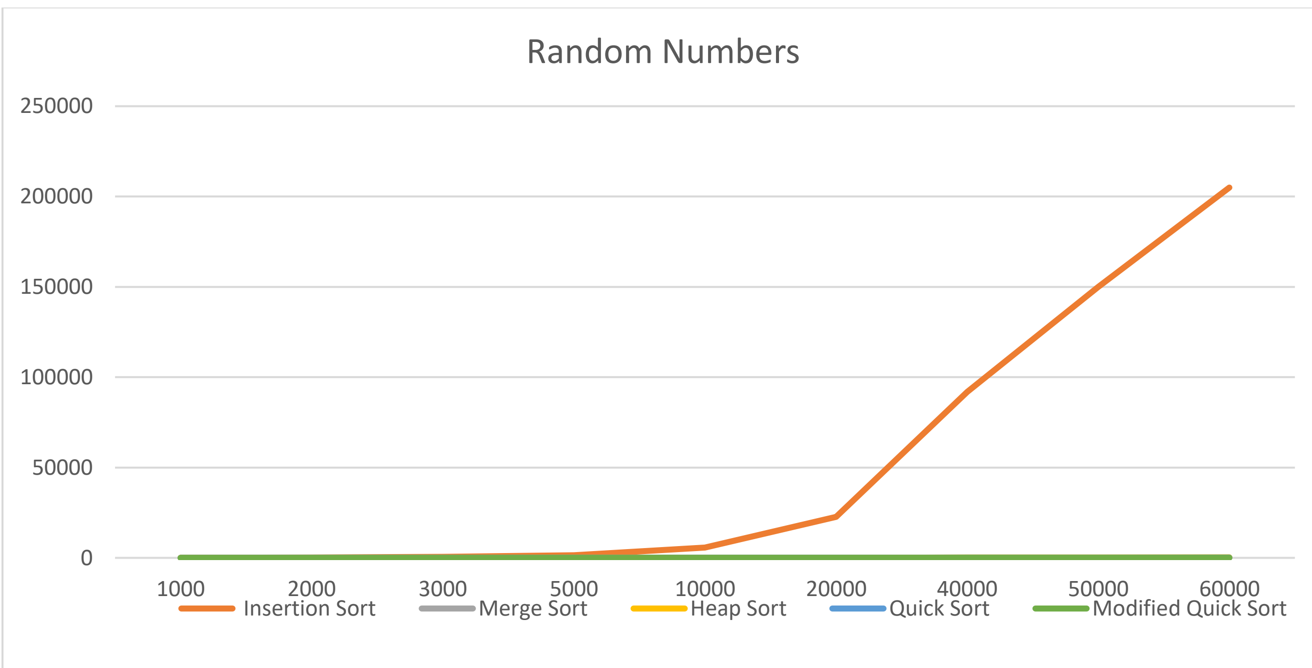
4.In Place Quick Sort

```
def partition(arr, low, high):
    i =low - 1
    n=random.randint(low,high)
    pivot = arr[n]
    for j in range(low, high):
        if arr[j] <= pivot:
            i+= 1
            arr[i] = arr[j]
            arr[j] = arr[i]

    arr[i + 1] = arr[high]
    arr[high] =  arr[i + 1]
    return (i + 1)

def Quick_sort(arr, low, high):
    if low < high:
        p = partition(arr, low, high)
        Quick_sort(arr, low, p - 1)
        Quick_sort(arr, p + 1, high)
```

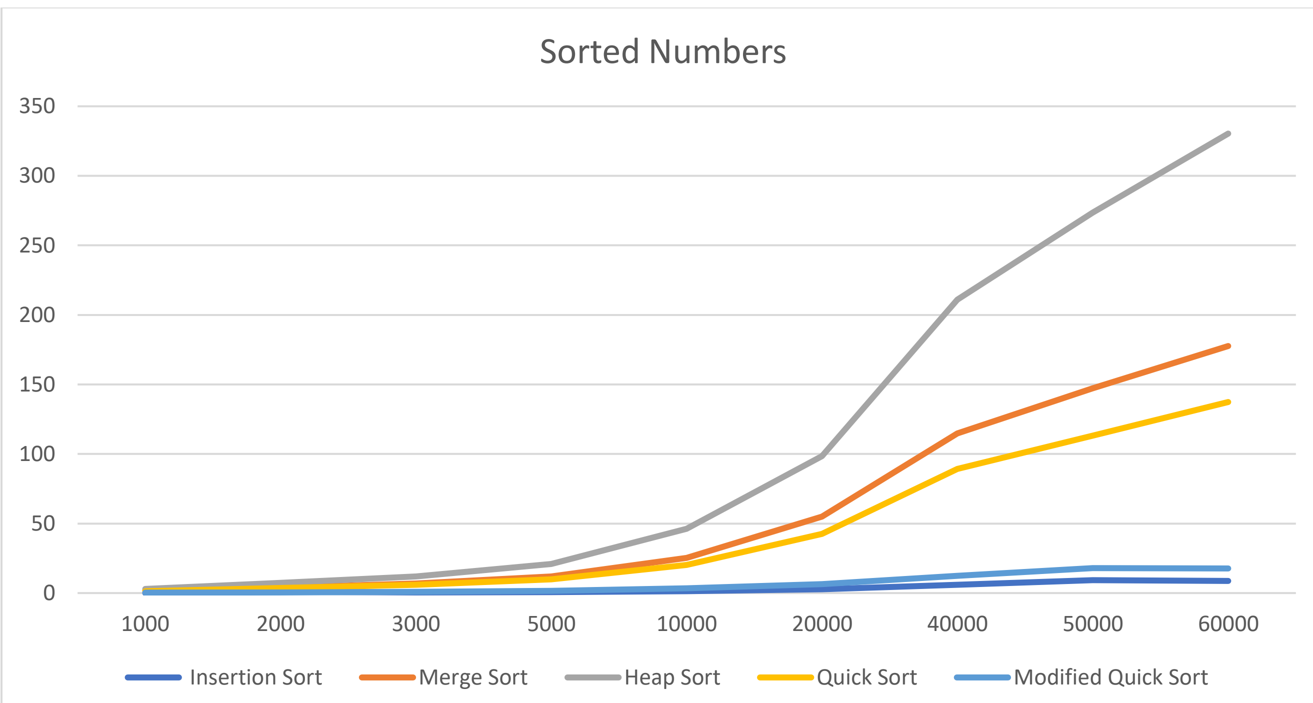
5.Modified Quick Sort

```
def median(arr, minimum, maximum):
   s = int((maximum + minimum))
   middle=int(s/2)
   if arr[minimum] > arr[middle]:
      arr[minimum]= arr[middle]
      arr[middle] = arr[minimum]
   if arr[minimum] > arr[maximum]:
      arr[maximum]= arr[minimum]
      arr[minimum] =arr[maximum]
   if arr[middle] > arr[maximum]:
      arr[maximum] = arr[middle]
      arr[middle] = arr[maximum]

   return middle



def quick_sort_median(arr, low, high):
      if low +10 < high:
         p = median(arr, low, high)
         quick_sort_median(arr, low, p - 1)
         quick_sort_median(arr, p + 1, high)
      else:
         insertionSort(arr, low, high)
```

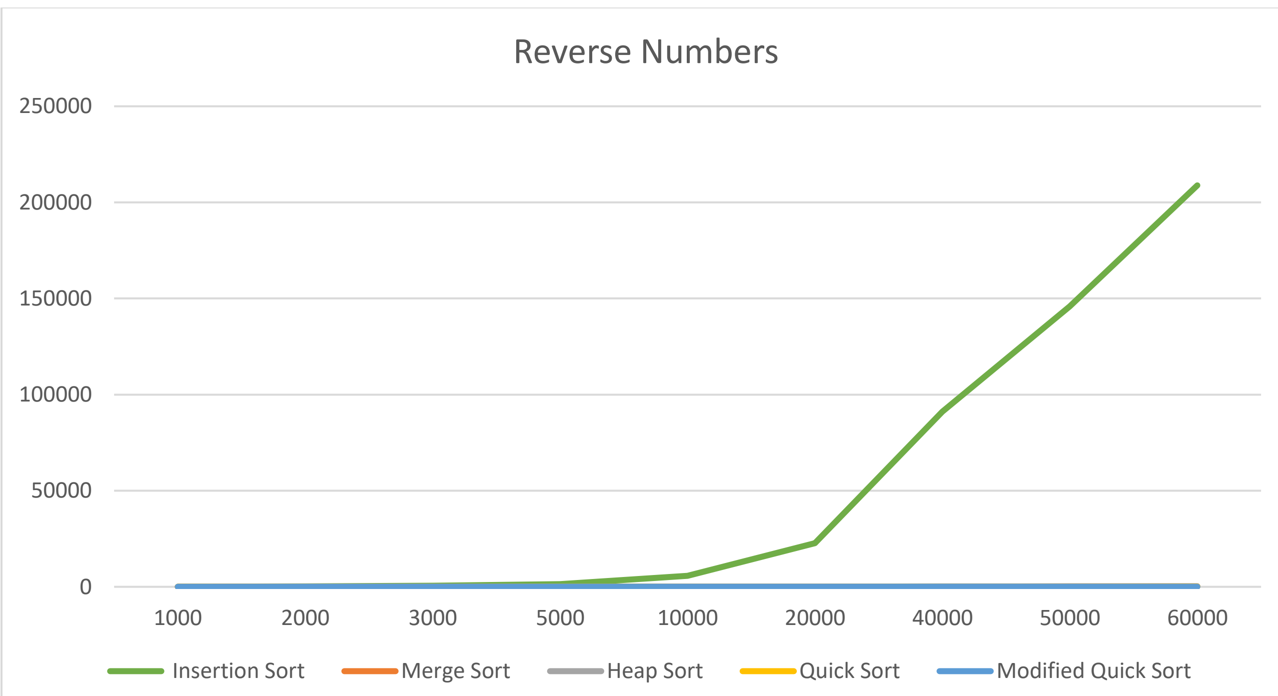
Random Numbers					
Elements	**Insertion Sort**	**Merge Sort**	**Heap Sort**	**Quick Sort**	**Modified Quick Sort**
1000	53.8201	1.9931	2.9807	1.9931	0.9965
2000	221.7757	4.9831	6.9758	2.9902	0.9963
3000	499.7061	7.9734	10.9591	4.9831	0.9965
5000	1400.0763	12.9568	17.9402	9.9666	1.9931
10000	5632.9557	28.9034	40.863	19.9332	3.3752
20000	22750.2698	61.7935	88.7572	40.8854	6.9763
40000	91927.1373	130.2285	190.317	91.6936	12.9566
50000	150013.9493	163.4538	245.256	107.5999	18.9332
60000	204968.4834	203.0844	301.05	135.5466	19.9329



Sorted Numbers					
Elements	**Insertion Sort**	**Merge Sort**	**Heap Sort**	**Quick Sort**	**Modified Quick Sort**
1000	0	1.9931	2.9758	1.3277	0.3322
2000	0.9968	3.9866	7.3089	3.6536	0.3323
3000	0.3548	6.9731	11.9616	5.9808	0.9967
5000	0.6643	12.0486	20.9303	9.7609	1.6612
10000	1.3221	25.2451	46.1761	20.2654	3.3222
20000	2.6562	54.9029	98.3492	42.5179	6.4308
40000	5.9658	114.7561	210.962	89.3699	12.2922
50000	9.2838	147.1684	273.518	113.2878	17.9398
60000	8.6325	177.6312	330.399	137.367	17.6079



Reverse Numbers					
Elements	**Insertion Sort**	**Merge Sort**	**Heap Sort**	**Quick Sort**	**Modified Quick Sort**
1000	53.6552	2.3269	2.6578	1.9932	0.3321
2000	218.4521	4.9832	6.6443	3.3221	0.6644
3000	497.5111	7.9745	10.467	5.6462	0.9966
5000	1401.6144	13.2889	18.2719	9.3021	1.6623
10000	5671.0221	28.9033	40.8655	20.2653	3.3223
20000	22747.9538	62.5344	88.6163	41.8457	6.9771
40000	91120.6982	130.4165	191.617	91.3612	13.9536
50000	145929.2221	164.5348	246.496	111.3077	18.9366
60000	208870.2918	200.4794	301.828	133.5535	20.1005



Observation

- Insertion sort is efficient for already sorted array. Reversely sorted input and random input have worst time complexity of O(n^2).
- Merge sort has same time complexity of O(nlogn) for all inputs(Sorted input, Reversely sorted input, Random input.
- Heap sort has same time complexity of O(nlogn) for all inputs(Sorted input, Reversely sorted input, Random input.
- Quick sort has same time complexity for all inputs(Sorted input, Reversely sorted input, Random input.
- Modified Quick sort is better than In-place Quicksort.

Machine used

RAM- 16 GB 3200Mhz

Processor- i7-11800H

Operating system- Windows 10