Comparison-based Sorting Algorithms

PROJECT DOCUMENTATION REPORT

PROGRAMMING PROJECT 1

ITCS 6114 – Algorithm & Data Structures

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO
Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

Harshitha
Keshavaraju Vijaylakshmi
801084151

Shailaja Yadav
801079146

Table of Contents

INTRODUCTION

Statement

In this project, we are executing and comparing different sorting techniques, with different input sizes. We have recorded the execution time for each of these sorting methods for different inputs and plotted the graph. We have used Insertion sort, Merge sort, In place quick sort , and modified quick sort as different sorting techniques.

1) Insertion Sort

Insertion sort is a comparison based sorting algorithm that takes any input, and results in a sorted output.

- The best case for insertion sort is when the array is already sorted the time complexity in this case is given as O(n+d), where d is the number of inversions.
- The worst case for insertion sort is when the array is inversely sorted, the time complexity in this case is given as O(n^2).
- The worst case space complexity in insertion sort is O(n^2) and O(1) for auxiliary
- Insertion sort perform well for smaller input sizes (<1k)

Algorithm for Insertion sort :

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

2) Merge Sort

Merge sort is a comparison-based sorting algorithm which is based on divide and conquer strategy.

- The worst-case time complexity of merge sort is given as O(nlogn).
- It is a stable algorithm; merge sort is insensitive to the initial order of its input.
- It is a fast comparison and sequential sorting technique
- It is preferred for huge data sets of > 1Million
- The worst-case space complexity for merge sort is O(n) for auxiliary.

Algorithm for Merge sort :

```
def merge_sort(a):
    if(len(a)>1):
        mid = len(a)//2
        lefth = a[:mid]
        righth = a[mid:]
        merge_sort(lefth)
        merge_sort(righth)
        i = j = k = 0
        while(i < len(lefth) and j < (len(righth))):
            if(lefth[i] < righth[j]):
                a[k] = lefth[i]
                i +=1
            else:
                a[k] = righth[j]
                j +=1
            k+=1
        while( i < len(lefth)):
            a[k] = lefth[i]
            i +=1
            k += 1
        while( j< len(righth)):
            a[k] = righth[j]
            j +=1
            k +=1
merge_sort(a)
```

3) Quick Sort

Quick sort is a comparison-based sorting algorithm which is based on divide and conquer strategy.

- The best case of quick sort is given as O(n logn)
- Worst case of quick sort is when the array is already sorted, or array is inversely sorted. The time complexity is O(n^2)
- Quick sort is actually the fastest algorithm in real time.

i) In-place Quick Sort

- In-Place quick sort technique does not require auxiliary arrays.
- There are three ways to select pivot in this method:
  a) Rightmost element
  b) Leftmost element and

c) Random selection

Algorithm for In -place quick sort :

```
def quickSort(a,low,high):
    if(low < high):
        pi = partition(a,low,high)
        quickSort(a,low,pi-1)
        quickSort(a,pi+1,high)


def partition(a,low,high):
    i = low-1
    pivot = a[high]
    for j in range(low,high):
        if(a[j] <= pivot):
            i += 1
            a[i],a[j] = a[j],a[i]
    a[i+1],a[high] = a[high],a[i+1]
    return i+1

quickSort(a,0,len(a)-1)
```

ii) Modified Quick sort – Median of 3

- In Modified Quick Sort we select pivot by taking median of three elements (first, last, and middle) which will always partition the array into roughly half .
- We use Insertion sort when the partition is <10 values because Insertion sort is efficient for small arrays.

Algorithm for Modified Quick sort :

```
 def qsHelper(a,low,high):
    if(low + 10 <= high):

        if low<high:
            piv = partition(a,low,high)
            qsHelper(a,low,piv-1)
            qsHelper(a,piv+1,high)
    else:
        insertion_sort(a,low,high)


 def insertion_sort(a,low,high):
```

```
    for i in range(low+1,high+1):
        key = a[i]
        j = i-1
        while(j>=0 and key < a[j]):
            a[j+1] = a[j]
            j -= 1
        a[j+1] = key

def quickSort(a):
   qsHelper(a,0,len(a)-1)


def partition(a,low,high):
   pindex = median(a, low, high, (low + high) // 2)
   a[low], a[pindex] = a[pindex], a[low]
   pivot = a[low]
   lptr = low
   rptr = high

   flag = False
   while not flag:

      while lptr <= rptr and a[lptr] <= pivot:
         lptr = lptr + 1

      while a[rptr] >= pivot and rptr >= lptr:
         rptr = rptr -1

      if rptr < lptr:
         flag = True
      else:
         a[lptr],a[rptr] =a[rptr],a[lptr]

   temp = pivot
   a[a.index(pivot)] = a[rptr]
   a[rptr] = temp
   return rptr

def median(a, low, high, median):
  if a[low] < a[high]:

    return high if a[high] < a[median] else median
  else:
    return low if a[low] < a[median] else median
```

RESULT

We have tested all the above-mentioned sorting techniques for different values (eg : 500, 1000 and so on till 50000 inputs which are randomly sorted) and plotted the graph between Execution Time (milli seconds) and size of input array. The graph is shown below.

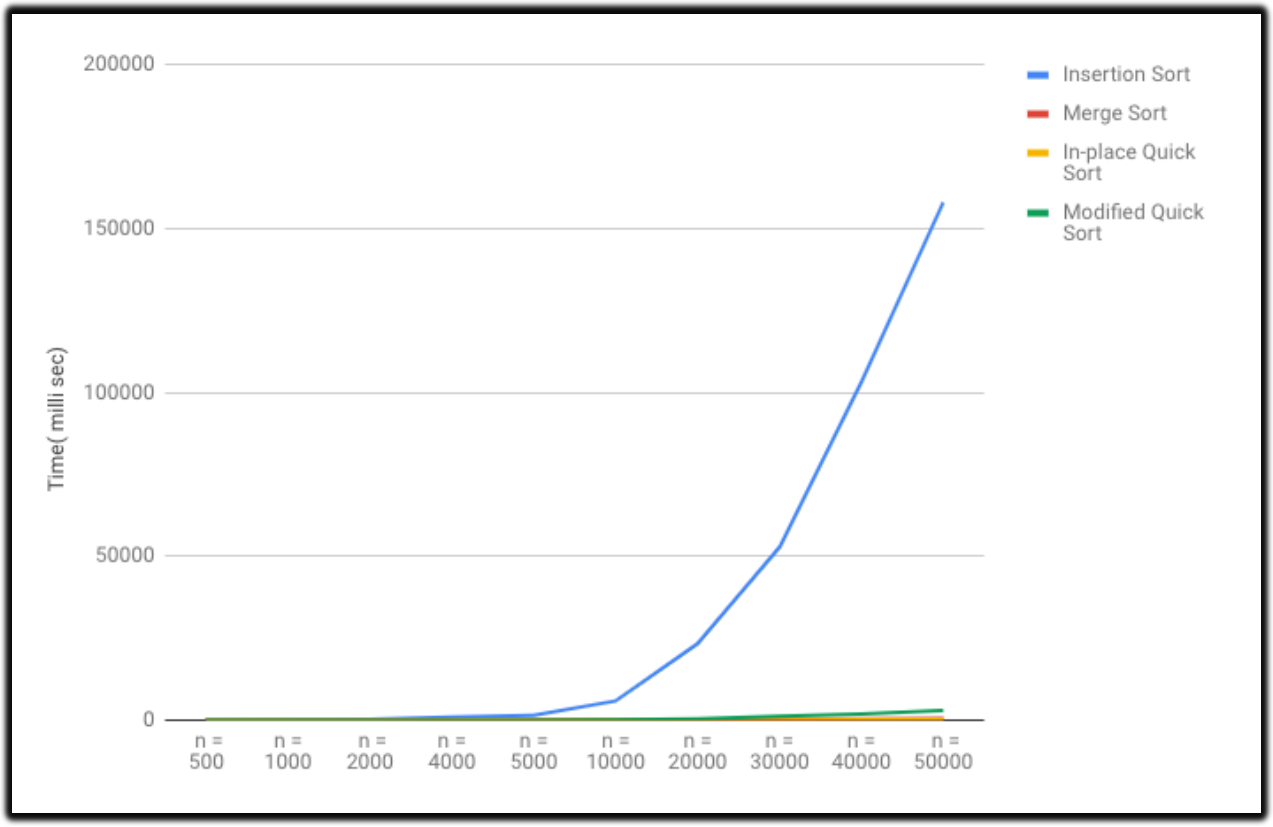

Figure 1: Comparison of sorting techniques

Analysis:

- In Figure 1, we can evidently see that the best performing sorting algorithm has to be merge sort as the input size of randomly sorted data increases, followed by the In-place quick sort, Modified quick sort and lastly insertion sort.
- Merge sort has a worst- and best-case time complexity of O (n logn) irrespective of how the data is sorted or not. Hence, merge sort has a steady graph irrespective of the input size, in fact merge sort performs well with large data sizes.
- Quick sort performance depends on a balanced partition of input, it does not perform well when the partitions are unbalanced. In randomly sorted input we cannot guarantee that the first or last element selected as pivot would be the best choice of pivot.
- Modified Quick sort approach selects pivot by finding the median of input and if the partition input array is less than 10 elements we perform insertion sort. As the sizes of

input array increases we see that due to insertion sorting method and time for finding the median, the time complexity increases by a small degree.

- Insertion sort has a worst case of O(n^2) which is directly proportional to its input size. So as the input size increases, we see that the graph reaches its worst case and increases exponentially.

Special case analysis:

- Insertion sort performs the best when compared to other algorithms, when the input data is in sorted order. But it performs the worst when the input has been inversely sorted.
- Quick sort – in place, it gives us the worst performance comparatively when the input is sorted. When the input size increases weather, it is inversely or sorted order, it throws us an "maximum recursion depth exceeded in comparison" error.
- Merge sort being insensitive to the initial order of inputs, it performs well in both sorted and inversely data is provided.
- Modified quick sort, when provided with sorted or inversely sorted data, the performance of this algorithm is best compared to other algorithms when provided with smaller data. When larger data is provided its performance is good, but not as good as merge sort.

CONCLUSION

From the results observed by all sorting algorithms, we can see that merge sort has the best stable performance when provided with any kind of input. In-place quick sort was found to be the fastest algorithm but in special case analysis we were faced with limitations of stack overflow for larger input for sorted and inversely sorted data. Using Modified quick sort, we are able to stabilize those limitations when in place quick sort fails. Insertion sort algorithm was found to be best when data is more or less sorted.