Rishi Sharma
B17138
CS202 Assignment 1 Report

Sorting Algorithms:

Insertion Sort:

Pseudocode:

```
1  for j <- 2 to length[A]
2       do key <- A[j]
3        // Insert A[j] into the sorted sequence A[1 . . j - 1].
4        i <- j - 1
5        while i > 0 and A[i] > key
6           do A[i + 1] <- A[i]
7              i <- i - 1
8        A[i + 1] <- key
```

Time Complexity:

Best Case: Ω(n)

Worst Case: O(n^2)

Average Case: θ(n^2)

Space Complexity:

Worst Case: O(1) auxiliary

Remarks: It is a divide and conquer algorithm. Divides the array in three parts: sorted, under-process, unsorted. After each iteration, the sorted part remains sorted and the under-process element is inserted in this sorted part.

At each index *j* there can be *j-1* swaps. So if in a machine, swaps are quite costly in terms of resources, we would not Insertion Sort.

Selection Sort:

Pseudocode

```
//Determine the smallest element and move it to A[0], next
    smallest element into A[1] and so on.
1.   j = 1
2.   while  j  <  n
          // Find the position of the smallest element
3.        pos = j
4.        for  i = j+1 to n
5.             if  A[i] < A[j]
```

```
6.                  pos = i
          // Move A[j] to the position of smallest element by
    swapping
7.        temp = A[pos]
8.        A[pos] = A[j]
9.        A[j] = temp
10.       j = j - 1
```

Time Complexity:

Best Case: Ω(n^2)

Worst Case: O(n^2)

Average Case: θ(n^2)

Space Complexity:

Worst Case: O(1) auxiliary

Remarks: In selection sort, we choose the minimum element in each iteration and put it at its correct place. This causes it to make minimum number of swaps to get to sorted array. Hence, if in a machine, swaps(writing to memory) are costly, we would use selection sort.

Rank Sort:
Pseudocode

```
1.   for  j = 1 to n
2.        R[j] = 1
// Rank the n elements in A into R
3.   for  j = 2 to n
4.        for  i = 1 to j - 1
5.             if  A[i] <= A[j]
6.                  R[j] = R[j] + 1
7.             else
8.                  R[i] = R[i] + 1
// Move to correct place in U[1 . . n]
9.   for  j = 1 to n
10.       U[R[j]] = A[j]
// Move the sorted entries into A
11.   for  j = 1 to n
12.       A[j] = U[j]
```

Time Complexity:

Best Case: Ω(n^2)

Worst Case: O(n^2)

Average Case: θ(n^2)

Space Complexity:

Worst Case: O(n) auxiliary

Remarks: Rank sort is costly in terms of memory. But there are no swaps in the O(n^2) part. Thus this can also be effective over Insertion and Bubble Sort on a machine with costly swaps.

Bubble Sort:
Pseudocode

```
1. j = n
2. while  j ≥ 2
          //  Bubble  up  the  smallest  element  to  its  correct  position
3.        for  i = 1 to j - 1
4.             if  A[i] > A[i + 1]
5.                  temp = A[i]
6.                  A[i] = A[i + 1]
7.                  A[i + 1] = temp
8.        j = j - 1
```

Time Complexity:

Best Case: Ω(n^2)

Worst Case: O(n^2)

Average Case: θ(n^2)

Space Complexity:

Worst Case: O(1) auxiliary

Remarks: Bubble sort is the simplest sorting algorithm, quite easy to implement. But the advantage is that it only compares and swaps adjacent elements. Hence in case the structure is not Random Access, bubble sort is advantageous.

Merge Sort:
Pseudocode:

```
MERGE (A, p, q, r)
1.   n1 = q - p + 1
2.   n2 = r - q
3.   A1[n1+1] = ∞
4.   A2[n2+1] = ∞
5.   i = 1
6.   j = 1
7.  for k = p to r
8.        if A1[i] ≤ A2[j]
9.             A[k] = A1[i]
10.            i = i + 1
11.       else
12.            A[k] = A2[j]
13.            j = j + 1
MERGE-SORT (A, p, r)
1.   if p < r then
2.        q = (p+r)/2
3.        MERGE-SORT(A, p, q)
4.        MERGE-SORT(A, mid+1, r)
5.        MERGE(A,p,q,r)
```

Time Complexity:

Best Case: Ω(n.log n)

Worst Case: O(n.log n)

Average Case: θ*(n.log n)*

Space Complexity:

Worst Case: O(n) auxiliary

Remarks: Merge Sort is the fastest - general sorting algorithm and hence it is the most widely used over other Sort functions. But the down-side is that it takes O(n) auxiliary space and hence if considering Memory more than time, we would not use Merge Sort, rather use Insertion/ Selection/Bubble Sort.

Quick Sort:

Pseudocode:

```
PARTITION (A, p, r)
1.   pivot = A[r]     // Pivot
2.   i = p - 1
3.   j = r + 1
4.   while  TRUE
5.              do
6.                   j = j - 1
7.         while A[j] > pivot
8.         do
9.                   i = i + 1
10.             while A[i] < pivot
11.        if j  >  i
12.             exchange A[i] with A[j]
13.        else if  j = i
14.                  return  j - 1
15              else
16                   return  j


QUICK-SORT (A, p, r)
1.   if   p < r  then
2.         q = PARTITION (A, p, r)
3.         QUICK-SORT (A, p, q)
4.         QUICK-SORT (A, q+1, r)
```

Time Complexity:

Best Case: Ω(n.log n)

Worst Case: O(n^2)

Average Case: θ(n.log n)

Space Complexity:

Worst Case: O(log n) auxiliary

Remarks: This is also a fast sorting algorithm that compares to Merge Sort in the best case. It also takes O(log n) auxiliary space, hence more efficient than merge sort when memory is under consideration.

Heap Sort:

Pseudocode:

```
heapify (A, i)
1.  l = left(i)
2.    r = right(i)
3.  if (l ≤ heap_size) and (A[l] < A[i])
4.        smallest = l
5.  else    //smallest = index of smallest key
6.         smallest = i
7.    end
8.    if (r ≤ heap_size) and (A[r] < A[smallest])
9.         smallest = r
10.  if (smallest ≠ i)
11.       exchange A[i] and A[smallest]
12.       heapify (A, smallest)
13. end
heapSort (A)
1.   build_Heap (A)
2.   for i = n down to 2
3.        Exchange A[1] and A[i]
4.        heap_size = heap_size -1
5.        heapify (A, 1)
6.   end
build_Heap (A)
1.   for i = (A.length)/2 down to 1
2.        heapify (A, i)
```

Time Complexity:

Best Case: Ω(n.log n)

Worst Case: O(n.log n)

Average Case: θ(n.log n)

Space Complexity:

Worst Case: O(1) auxiliary

Remarks: Heap sort is in-place sort. Hence it takes O(1) memory. Therefore it is more efficient than merge/quick sort when memory is under

consideration. It is also O(n.log n) time complexity sorting algorithm which makes it best to use.

Problems:

1. In this problem, we first sort the whole array. Now there can be two cases:

n is odd: Then x can take only 1 value to minimise the total sum of distances from all elements, i.e. the middle element.

n is even: Here x can take all values between the middle two elements(included). Sum of these terms would be calculated using AP sum.

Pseudocode:

```
1. Input A
2. MergeSort A
3. if n is odd:
        Output A[n/2] mod 10^9+7 // 0-based indexing
4. else:
        Output ((A[n/2]+A[n/2-1])*(A[n/2]-A[n/2-1]+1)/2) mod 10^9+7
```

Time Complexity:

Best Case: Ω(n.log n)

Worst Case: O(n.log n)

Average Case: θ(n.log n)

Space Complexity:

Worst Case: O(n) auxiliary (Merge Sort)

---

3. This problem is quite simple. After dividing the array into two, we need to make the larger array, larger and smaller array smaller. But this leaves one corner case. What if the sum of both the arrays is equal? To solve this issue, we generalise the problem as two same arrays, before dividing. Divide and Sort them both. In the first array, we make the first partition bigger and second partition smaller. In the second array, we make the first partition smaller and second partition bigger.

Pseudocode:

```
1. Input A
2. Copy A to B
3. Sort(A[0]..A[N/2-1]) AND Sort(A[N/2]..A[N-1])
```

4. Swap k smallest element of first partition and k largest elements of second partition, provided that the first is smaller than the second.
5. Sort(B[0]..B[N/2-1]) AND Sort(B[N/2]..B[N-1])
6. Swap k smallest element of second partition and k largest elements of first partition, provided that the first is smaller than the second.
7. if(max(sumofFirstHalf(A), sumofSecondHalf(A)) > max(sumofFirstHalf(B), sumofSecondHalf(B))):
   print A.saved, A.killed
8. else:
   print B.saved, B.killed

Time Complexity:

Best Case: Ω(n.log n)

Worst Case: O(n.log n)

Average Case: θ(n.log n)

Space Complexity:

Worst Case: O(n) auxiliary

---

4. This problem is nothing but implementation of code. First, we need to insert the students in the line next to their friends as they arrive. Next, we need to count the minimum number of swaps required to sort them in increasing order of their heights. Insertion of students in a line can be done in the worst case of O(n^2) and hence we need not optimise the calculation of minimum number of swaps beyond that. We also know that selection sort makes the minimum number of swaps required to sort the array, hence we sort the array using selection sort and count the number of swaps for each individual line and sum them up and output the time.

Pseudocode:

```
ARE-FRIENDS(x, y):
1.      g=GCD(x,y)
2.      if(g==1 or isPrime(g)) return false
3.      return true
MAIN:
1.      Input heights
2.      For each student s :
```

```
3.          For each x in s.list :
4.              if x==NULL or ARE-FRIENDS(x,s):
5.                   Insert s after x
6.      Ans = 0
7.      For each list l:
5.         Ans +=SelectionSort(l) : Modified to return number of swaps
6.      print(Ans/2)
```

If n is number of students..

Time Complexity:

Best Case: Ω(sqrt(n)*n^2)

Worst Case: O(sqrt(n)*n^2)

Average Case: θ(sqrt(n)*n^2)

Space Complexity:

Worst Case: O(n) auxiliary