CS 202
Advance Data Structures and Algorithms
Assignment-1 Report
B17069
(Vipul Sharma)

1. Insertion Sort :-

Best Case - O(n)
Worst Case - O(n^2)
Average Case - O(n^2)

Space Complexity - O(1)

Pseudo Code -

```
for j = 2 to n
    key = A [ j ]
// Insert A[j] into the sorted sequence A[1 ... j - 1]
    i = j - 1
    while i > 0  and A [ i ] > key
        A [ i + 1 ] = A [ i ]
        i = i - 1
    A [ i + 1 ] = key
```

Remarks -

1. Good for sorting smaller arrays. In fact, the smaller the array, the faster insertion sort is compared to any other sorting algorithm.
2. Being O(n^2), it becomes very slow when array size increases.
3. It can detect if the array is already sorted and process it in O(n).
4. In-place, as no extra memory required to sort.
5. Stable, as relative position of similar elements remains the same.

2. Selection Sort :-

Best Case - O(n^2)
Worst Case - O(n^2)
Average Case - O(n^2)

Space Complexity - O(1)

Pseudo Code -

```
sorted = false
j = n
while j > 1  and sorted = false
    pos = 1
    sorted = true
// Find the position of the largest element
    for i = 2 to j
        if A [ pos ] <= A [ i ]
            pos = i
        else
            sorted = false
  // Move A[j] to the position of largest element by swapping
        temp = A [ pos ]
        A [ pos ] =  A [ j ]
        A [ j ] =  temp
        j = j - 1
```

Remarks-

1. It is also inefficient for large array sizes, even more inefficient than Insertion Sort.
2. Rarely used in complex applications.
3. Finds the smallest/largest element in each iteration and places it at the required position.
4. In-place, as no extra memory required to sort.
5. Not Stable, as relative position of similar elements may change.

3. Rank Sort :-

Best Case - O(n^2)
Worst Case - O(n^2)
Average Case - O(n^2)

Space Complexity - O(n)

Pseudo Code -

```
  for j = 1 to n
      R [ j ] = 1
// Rank the n elements in A into R
  for j = 2 to n
      for i = 1 to j - 1
          if A [ i ] <= A [ j ]
              R [ j ] = R [ j ] + 1
          else
              R [ i ] = R [ i ] + 1
  // Move to correct place in U[1 . . n]
  for j = 1 to n
          U [ R [ j ]] = A [ j ]
  // Move the sorted entries into A
  for j = 1  to n
            A [ j ] = U [ j ]
```

Remarks-

1. It's main problem is that it requires a second array with the same size as the array to be sorted, thus doubling the space requirements. Thus, it is not an in-place sorting algorithm.
2. Not used much as it has a very bad time as well as space complexity.
3. Is O(n2) time complexity in every case, which is same as Selection Sort, but also has extra space complexity.
4. Can be regarded as the worst sorting algorithm among all algorithms specified in this document.

4. Bubble Sort :-

Best Case - O(n)
Worst Case - O(n^2)
Average Case - O(n^2)

Space Complexity - O(1)

Pseudo Code -

```
do {
     swapped = false
     for i = 0 to i = n-1
          if arr[i] > arr[i+1]
              swapped = true
              temp = arr[i]
              arr[i] = arr[i+1]
              arr[i+1] = temp
     n--
} while (swapped)
```

Remarks-

1. In-place sorting algorithm. Does not require any extra space to sort.
2. Is stable, as relative position of similar elements remains unchanged.
3. The bubble sort works by iterating down an array to be sorted from the first element to the last, comparing each pair of elements and switching their positions if necessary. This process is repeated as many times as necessary, until the array is sorted.

5. Merge Sort :-

Best Case - O(nlog(n))
Worst Case - O(nlog(n))
Average Case - O(nlog(n))

Space Complexity - O(n)

Pseudo Code -

```
mergeSort(arr[], l, r)
if (l < r)
     m = l+(r-l)/2;
     mergeSort(arr, l, m);
     mergeSort(arr, m+1, r);
     merge(arr, l, m, r);

merge(A, p, q, r);
n1 = q  - p + 1
n2 = r  - q
A1 [ n1 + 1 ] =  ∞
A2 [ n2 + 1 ] =  ∞
i = 1
j = 1
for k = p to r
     if A1 [ i ]  ≤  A2 [ j ]
         A [ k ] = A1 [ i ]
         i = i + 1
     else
         A [ k ] = A2 [ j ]
         j = j + 1
```

Remarks-

1. Stable sorting algorithm (doesn’t change order of equal elements).
2. O(nlogn) Sorting algorithm in every case.
3. It’s main problem is that it requires a second array with the same size as the array to be sorted, thus doubling the space requirements. Thus, it is not an in-place sorting algorithm.

7. Quick Sort :-

Best Case - O(nlog(n))
Worst Case - $O(n^2)$
Average Case - O(nlog(n))

Space Complexity - O(log(n))

Pseudo Code -

```
quickSort(arr[], low, high)
    if (low < high)
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);

partition(A , l , r)
    pivot = A [ r ] //Pivot
    i = p - 1
    j = r + 1
    while TRUE
        do
            j = j - 1
    while A [ j ] > pivot
         do
             i = i + 1
        while A [ i ]  <  pivot
            if j > i
                exchange A [ i ] with A [ j ]
            else if j = i
                return j - 1
            else
                return j
---------------------------------------------------------------
```

Remarks-

1. Is in-place sorting algorithm and is very fast in average case.
2. Not stable, as relative position of similar elements may change.
3. Not trustworthy in every case. Complexity can go upto $O(n^2)$ in worst case.
4. Does not require a second array like Merge Sort. So wins over Merge Sort, in department of space.

8. Heap Sort :-

Best Case - O(nlog(n))
Worst Case - O(nlog(n))
Average Case - O(nlog(n))

Space Complexity - O(1)

Pseudo Code -

```
Heapsort(A)
   1. BuildHeap(A)
   2. for i = length(A) downto 2 {
   3.     exchange A[1] and A[i]
   4.     heapsize = heapsize -1
   5.     Heapify(A, 1)

BuildHeap(A)
   1. heapsize = length(A)
   2. for i = floor( length/2 ) downto 1
   3.     Heapify(A, i)

Heapify(A, i)
   1. le = 2*i+2
   2. ri = 2*i+1
   3. if (le<=heapsize) and (A[le]>A[i])
   4.     largest = le
   5. else
   6.     largest = i
   7. if (ri<=heapsize) and (A[ri]>A[largest])
   8.     largest = ri
   9. if (largest != i)
   10.     exchange A[i] and A[largest]
   11.     Heapify(A, largest)
----------------------------------------------------------------
```

Remarks-

1. In-place non-recursive sorting algorithm.
2. Is trustworthy, takes O(nlog(n)) time complexity in every case.
3. Doesn’t require any extra memory. O(1) space complexity is a major benefit of this sorting.