Bubble Sort

In Bubble sort, Each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires n-1 passes for sorting. Consider an array A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.

1. In Pass 1, A[0] is compared with A[1], A[1] is compared with A[2], A[2] is compared with A[3] and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.
3. In pass n-1, A[0] is compared with A[1], A[1] is compared with A[2] and so on. At the end of this pass. The smallest element of the list is placed at the first index of the list.

Algorithm :

- **Step 1**: Repeat Step 2 For i = 0 to N-1
- **Step 2**: Repeat For J = i + 1 to N - I
- **Step 3**: IF A[J] > A[i]
  SWAP A[J] and A[i]
  [END OF INNER LOOP]
  [END OF OUTER LOOP
- **Step 4**: EXIT

Heap Sort

Heap sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

- Build a heap H, using the elements of ARR.
- Repeatedly delete the root element of the heap formed in phase 1.

Algorithm

HEAP_SORT(ARR, N)

- **Step 1**: [Build Heap H]
  Repeat for i=0 to N-1
  CALL INSERT_HEAP(ARR, N, ARR[i])
  [END OF LOOP]
- **Step 2**: Repeatedly Delete the root element
  Repeat while N > 0
  CALL Delete_Heap(ARR,N,VAL)
  SET N = N+1
  [END OF LOOP]
- **Step 3**: END

Insertion Sort

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array. This is less efficient than the other sort algorithms like quick sort, merge sort, etc.

Technique

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k-1], A[k-2], and so on until we find an element A[j] such that, A[j]<=A[k].

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

Algorithm

- **Step 1**: Repeat Steps 2 to 5 for K = 1 to N-1
- **Step 2**: SET TEMP = ARR[K]
- **Step 3**: SET J = K ? 1
- **Step 4**: Repeat while TEMP <=ARR[J]
  SET ARR[J + 1] = ARR[J]
  SET J = J ? 1
  [END OF INNER LOOP]
- **Step 5**: SET ARR[J + 1] = TEMP
  [END OF LOOP]
- **Step 6**: EXIT

Merge sort

Merge sort is the algorithm which follows divide and conquer approach. Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.

1. If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
2. Conquer means sort the two sub-arrays recursively using the merge sort.
3. Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

   The main idea behind merge sort is that, the short list takes less time to be sorted.

Example :

Consider the following array of 7 elements. Sort the array by using merge sort.

1. A = {10, 5, 2, 23, 45, 21, 7}

Algorithm

- **Step 1**: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
- **Step 2**: Repeat while (I <= MID) AND (J<=END)
  IF ARR[I] < ARR[J]
  SET TEMP[INDEX] = ARR[I]
  SET I = I + 1
  ELSE
  SET TEMP[INDEX] = ARR[J]
  SET J = J + 1
  [END OF IF]
  SET INDEX = INDEX + 1
  [END OF LOOP]
  Step 3: [Copy the remaining
  elements of right sub-array, if
  any]
  IF I > MID
  Repeat while J <= END
  SET TEMP[INDEX] = ARR[J]
  SET INDEX = INDEX + 1, SET J = J + 1
  [END OF LOOP]
  [Copy the remaining elements of

left sub-array, if any]
ELSE
Repeat while I <= MID
SET TEMP[INDEX] = ARR[I]
SET INDEX = INDEX + 1, SET I = I + 1
[END OF LOOP]
[END OF IF]

- **Step 4**: [Copy the contents of TEMP back to ARR] SET K = 0
- **Step 5**: Repeat while K < INDEX
  SET ARR[K] = TEMP[K]
  SET K = K + 1
  [END OF LOOP]
- **Step 6**: Exit

MERGE_SORT(ARR, BEG, END)

- **Step 1**: IF BEG < END
  SET MID = (BEG + END)/2
  CALL MERGE_SORT (ARR, BEG, MID)
  CALL MERGE_SORT (ARR, MID + 1, END)
  MERGE (ARR, BEG, MID, END)
  [END OF IF]
- **Step 2**: END

Quick Sort

Quick sort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach. The algorithm processes the array in the following way.

1. Set the first index of the array to left and loc variable. Set the last index of the array to right variable. i.e. left = 0, loc = 0, en d = n ? 1, where n is the length of the array.
2. Start from the right of the array and scan the complete array from right to beginning comparing each element of the array with the element pointed by loc.

   Ensure that, a[loc] is less than a[right].

   1. If this is the case, then continue with the comparison until right becomes equal to the loc.
   2. If a[loc] > a[right], then swap the two values. And go to step 3.
   3. Set, loc = right

1. start from element pointed by left and compare each element in its way with the element pointed by the variable loc. Ensure that a[loc] > a[left]
    1. if this is the case, then continue with the comparison until loc becomes equal to left.
    2. [loc] < a[right], then swap the two values and go to step 2.
    3. Set, loc = left.

Algorithm

PARTITION (ARR, BEG, END, LOC)

- **Step 1**: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =
- **Step 2**: Repeat Steps 3 to 6 while FLAG =
- **Step 3**: Repeat while ARR[LOC] <=ARR[RIGHT]
  AND LOC != RIGHT
  SET RIGHT = RIGHT - 1
  [END OF LOOP]
- **Step 4**: IF LOC = RIGHT
  SET FLAG = 1
  ELSE IF ARR[LOC] > ARR[RIGHT]
  SWAP ARR[LOC] with ARR[RIGHT]
  SET LOC = RIGHT
  [END OF IF]
- **Step 5**: IF FLAG = 0
  Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
  SET LEFT = LEFT + 1
  [END OF LOOP]
- **Step 6**:IF LOC = LEFT
  SET FLAG = 1
  ELSE IF ARR[LOC] < ARR[LEFT]
  SWAP ARR[LOC] with ARR[LEFT]
  SET LOC = LEFT
  [END OF IF]
  [END OF IF]
- **Step 7**: [END OF LOOP]
- **Step 8**: END

QUICK_SORT (ARR, BEG, END)

- **Step 1**: IF (BEG < END)
  CALL PARTITION (ARR, BEG, END, LOC)
  CALL QUICKSORT(ARR, BEG, LOC - 1)
  CALL QUICKSORT(ARR, LOC + 1, END)
  [END OF IF]
- **Step 2**: END

Radix Sort

Radix sort processes the elements the same way in which the names of the students are sorted according to their alphabetical order. There are 26 radix in that case due to the fact that, there are 26 alphabets in English. In the first pass, the names are grouped according to the ascending order of the first letter of names.

In the second pass, the names are grouped according to the ascending order of the second letter. The same process continues until we find the sorted list of names. The bucket are used to store the names produced in each pass. The number of passes depends upon the length of the name with the maximum letter.

In the case of integers, radix sort sorts the numbers according to their digits. The comparisons are made among the digits of the number from LSB to MSB. The number of passes depend upon the length of the number with the most number of digits.

Example

Consider the array of length 6 given below. Sort the array by using Radix sort.

A = {10, 2, 901, 803, 1024}

Pass 1: (Sort the list according to the digits at 0's place)

10, 901, 2, 803, 1024.

Pass 2: (Sort the list according to the digits at 10's place)

02, 10, 901, 803, 1024

Pass 3: (Sort the list according to the digits at 100's place)

02, 10, 1024, 803, 901.

Pass 4: (Sort the list according to the digits at 1000's place)

02, 10, 803, 901, 1024

Therefore, the list generated in the step 4 is the sorted list, arranged from radix sort.

Algorithm

- **Step 1**:Find the largest number in ARR as LARGE

- **Step 2**: [INITIALIZE] SET NOP = Number of digits in LARGE
- **Step 3**: SET PASS =0
- **Step 4**: Repeat Step 5 while PASS <= NOP-1
- **Step 5**: SET I = 0 and INITIALIZE buckets
- **Step 6**:Repeat Steps 7 to 9 while I<n-1< li=""> </n-1<>
- **Step 7**: SET DIGIT = digit at PASSth place in A[I]
- **Step 8**: Add A[I] to the bucket numbered DIGIT
- **Step 9**: INCREMENT bucket count for bucket numbered DIGIT
  [END OF LOOP]
- **Step 10**: Collect the numbers in the bucket
  [END OF LOOP]
- **Step 11**: END

Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

- In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
- In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
- In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].

Therefore, by following the above explained process, the elements A[0], A[1], A[2],...., A[n-1] are sorted.

Example

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

A = {10, 2, 3, 90, 43, 56}.

Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

- In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
- In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
- In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].

Therefore, by following the above explained process, the elements A[0], A[1], A[2],...., A[n-1] are sorted.

Example

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

A = {10, 2, 3, 90, 43, 56}.

Pass	Pos	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
1	1	2	10	3	90	43	56
2	2	2	3	10	90	43	56
3	2	2	3	10	90	43	56
4	4	2	3	10	43	90	56
5	5	**2**	**3**	**10**	**43**	**56**	**90**

Sorted A = {2, 3, 10, 43, 56, 90}

Algorithm

SELECTION SORT(ARR, N)

- **Step 1**: Repeat Steps 2 and 3 for K = 1 to N-1
- **Step 2**: CALL SMALLEST(ARR, K, N, POS)
- **Step 3**: SWAP A[K] with ARR[POS]
  [END OF LOOP]
- **Step 4**: EXIT

SMALLEST (ARR, K, N, POS)

- **Step 1**: [INITIALIZE] SET SMALL = ARR[K]
- **Step 2**: [INITIALIZE] SET POS = K
- **Step 3**: Repeat for J = K+1 to N -1
  IF SMALL > ARR[J]
  SET SMALL = ARR[J]
  SET POS = J
  [END OF IF]
  [END OF LOOP]
- **Step 4**: RETURN POS

Shell Sort

Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions. In general, Shell sort performs the following steps.

- **Step 1**: Arrange the elements in the tabular form and sort the columns by using insertion sort.
- **Step 2**: Repeat Step 1; each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted.

Algorithm

Shell_Sort(Arr, n)

- **Step 1**: SET FLAG = 1, GAP_SIZE = N
- **Step 2**: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1
- **Step 3**:SET FLAG = 0

- **Step 4**:SET GAP_SIZE = (GAP_SIZE + 1) / 2
- **Step 5**:Repeat Step 6 for I = 0 to I < (N -GAP_SIZE)
- **Step 6**:IF Arr[I + GAP_SIZE] > Arr[I]
  SWAP Arr[I + GAP_SIZE], Arr[I]
  SET FLAG = 0
- **Step 7**: END