Unit-6
Sorting

The process of ordering elements in ascending or descending or any other specific order on the basis of value, priority order etc is called sorting. Sorting can be categorized as internal sorting and external sorting.

Internal sorting → Internal sorting means we are arranging the numbers within the array only which is in computers primary memory.

External sorting → External sorting is the sorting of numbers from the external file by reading it from secondry memory.

Use of Sorting:

We know that searching a sorted array is much eaiser than searching an unsorted array. The following are some examples where sorting is used so that searching will be much eaiser.

→ Words in a dictionary are sorted.
→ The index of a book is sorted.
→ The files in a directory are often listed in sorted order.
→ A listing of course offerings at university is sorted, first by department then by course number.
→ Many banks provide statements that list checks in increasing order by check number.

⊛. Comparision Sorting Algorithms: (Bubble, Selection, Insertion and Shell):

1) Bubble Sort: It is the simplest sorting algorithm that works by repeatedly swapping the adjcent elements if they are in wrong order. The basic idea of this sort is to pass through the array sequentially several times. Each pass consists of comparing each element in the array with it's adjcent (successor) element (a[i] with a[i+1]) and interchanging the two elements if they are not in the proper order.

Scanned with CamScanner

Characteristics of Bubble Sort:

i) Large values are always sorted first.

ii) It only takes one iteration to detect that a collection is already sorted.

iii) The best time complexity for Bubble Sort is $O(n)$. The average and worst time complexity is $O(n^2)$.

iv) The space complexity for Bubble sort is $O(1)$, because only single additional memory space is required

Algorithm

1. Start
2. For the first iteration, compare all the elements (n). For the subsequent runs, compare (n-1) (n-2) and so on.
3. Compare each element with its right side neighbour.
4. Swap the smaller element to the left.
5. Keep repeating steps 2, 3 and 4 until whole list is covered.
6. Stop.

Pseudo code

```
Bubble Sort(A,n)
 {
     for (i=0; i<n-1; i++)
       {
          for (j=0; j<n-i-1; j++)
            {
               if (A[j] > A[j+1])
                 {
                   temp = A[j];
                   A[j] = A[j+1];
                   A[j+1] = temp;
                 }
            }
       }
 }
```

Time Complexity:- Inner loop executes $(n-1)$ times when $i=0$, $(n-2)$ times when $i=1$ and so on.

Time Complexity $= (n-1)+(n-2)+(n-3)+\dots+2+1$
$= n(n-1)/2 = O(n^2)$.

There is no best case time complexity for this algorithm.

2) Selection Sort:

or selecting

Selection sort is about picking the smallest element from the list and placing it in the sorted portion of list. Initially, the first element is considered the minimum and compared with other elements. During these comparisions, if a smaller element is found then that is considered the new minimum. After completion of one full round, the smallest element found is swapped with the first element. This process continues till all the elements are sorted. The selection sort works as follows:-

Pass1: Find the location loc of the smallest element from the list of n elements a[0], a[1], a[2], .., a[n-1] and then interchange a[loc] and a[0]

Pass2: Find the location loc of the smallest element from the sub-list of n-1 elements a[1], a[2], .., a[n-1] and then interchange a[loc] and a[1] such that a[0], a[1] are sorted and so on.

Then finally we ~~the~~ get the sorted list a[0] <= a[1] <= a[2], .., <= a[n-1].

Algorithm:

1. Start
2. Consider the first element to be sorted and the rest to be unsorted.
3. Assume the first element to be the smallest element.
4. Check if the first element is smaller than each of other elements:
   i) If yes, do nothing
   ii) If no, choose the other smaller element as minimum and repeat step 3.
5. After completion of one iteration through the list, swap the smallest element with the first element of the list.
6. Now consider the second element in the list to be the smallest element and so on till all the elements in the list are covered.
7. Stop.

Pseudo code

```
Selection Sort (A)
{
        for (i=0; i<n; i++)
        {
            least = A[i];
            p = i;
            for (j = i+1; j<n; j++)
            {
                if (A[j] < A[i])
                {
                    least = A[j];
                    p = j;
                }
            }
            Swap (A[i], A[p]);
        }
}
```

Time Complexity: Inner loop executes for $(n-1)$ times when $i=0$, $(n-2)$ times when $i=1$ and so on:

Time complexity $= (n-1) + (n-2) + (n-3) + \ldots + 2 + 1$
$= O(n^2)$

There is no best-case linear time complexity for this algorithm, but the number of swap operations reduced greatly.

3 variable for bubble sort other same

Space Complexity: Since no extra space besides 5 varibles is needed for sorting.

Space Complexity $= O(n)$.

Note: Time Complexity / Analysis is also called Efficiency

3) Insertion Sort:

In this method an element gets compared and inserted into the correct position in the list. To apply this sort, we must consider one part of the list to be sorted and other to be unsorted.
To begin, consider the first element to be the sorted portion and the other elements of the list to be unsorted. Now compare each element from the unsorted portion with the element in the sorted portion. Then insert it in the correct position in the sorted part. This algorithm works best with small number of elements.

The insertion sort works as follows:-

Suppose an array a[n] with n elements.

Pass 1: a[0] by itself is trivially sorted.

Pass 2: a[1] is inserted either before or after a[0] so that a[0], a[1] is sorted.

Pass 3: a[2] is inserted into its proper place in a[0], a[1] that is before a[0], between a[0] and a[1] or after a[1], so that a[0], a[1], a[2] is sorted.

Pass n: a[n-1] is inserted into its proper place in a[0], a[1], a[2], ... a[n-2] so that a[0], a[1], a[2], ... a[n-1] is sorted with n elements.

Algorithm:

1. Start
2. Consider the first element to be sorted and rest to be unsorted.
3. Compare with the second element
   i) If the second element < the first element, insert the element in the correct position of the sorted portion.
   ii) Else, leave it as it is.
4. Repeat 1 and 2 until all elements are sorted.
5. Stop.

Pseudo code:

```
Insertion (A,n)
  {
    for i=1 to n
    { temp=A[i]
      j=i-1
    while (j>=0 && A[j]>temp)
    { A[j+1]=A[j]
      j=j-1
    }
```

}
A[j+1]=temp
}.

Time Complexity:- The worst case runtime complexity of insertion sort is $O(n^2)$ similar to that of Bubble sort. However, insertion sort is considered better than Bubble sort.
The best case time complexity is $O(n)$.

Space Complexity: Similar to of selection sort. [i.e, $O(n)$] Since no, extra space requires besides 5 variables, needed for sorting.

4> Shell Sort:

It is the first algorithm to improve on insertion sort. It's idea was to avoid the large amount of data movement, first by comparing elements that were far apart and then by comparing elements that were less far apart and so on, gradually shrinking toward the basic insertion sort.

The shell sort is a diminishing increament sort. This sort divides the original file into seperate sub-files. These sub-files contain every k^{th} element of the original file. The value of k is called increment. For example, if there are n elements to be sorted and value of k is five then,

Sub-file 1 → a[0], a[5], a[10], a[15], ...
Sub-file 2 → a[1], a[6], a[11], a[16], ...
Sub-file 3 → a[2], a[7], a[12], a[17], ...
Sub-file 4 → a[3], a[8], a[13], a[18], ...
Sub-file 5 → a[4], a[9], a[14], a[19], ...

⊛ Divide-and-Conquer algorithms:

Divide-and-conquer is an important problem-solving technique that makes use of recursion. It is an efficient recursive algorithm that consists of two parts:

i) Divide → In which smaller problems are solved recursively.

ii) Conquer → In which the solution to the original problem is then formed from the solutions to the sub-problems.

Traditionally, routines in which the algorithm contains at least two recursive calls are called divide-and-conquer algorithms, whereas routines whose text contains only one recursive call or not. Consequently, the recursive routines presented so far in this section are not divide-and-conques algorithms. Also, the sub-problems usually must be disjoint (i.e, essentially no overlapping), so as to avoid the excessive costs seen in the sample recursive computation of the Fibonacci numbers. Following are some of the divide-and-conquer algorithms:-

ⓐ. Quick Sort:

Quick Sort is developed by C.A.R Hoare which is unstable sorting. In practice this is the fastest sorting method. It possess very good average case complexity among all the sorting algorithms. This algorithm is based on the divide and conquer paradigm. The main idea behind this sorting is partitioning of the elements.

Steps for Quick sort:

Divide → Partition the array into two non-empty sub arrays.

Conquer → Two sub arrays are sorted recursively.

Combine → Two sub arrays are already sorted in place so no need to combine.

←—— n ——→

Partition 1	key	Partition 2
values < key		Values > key.

Algorithm:

1. Start
2. Choose a pivot.
3. Set a left pointer and right pointer
4. Compare the left pointer element (lelement) with the pivot and the right pointer element (relement) with the pivot.
5. Check if lelement < pivot and relement > pivot:
   i) If yes, increment the left pointer and decrement the right pointer.
   ii) If not, swap the lelement and relement.
6. When left >= right, swap the pivot with either left or right pointer.
7. Repeat step 1-5 on the left half and the right half of the list till the entire list is sorted.
8. Stop.

Pseudo code:

```
QuickSort (A,l,r)
{   if (l<r)
    {   p= Partition (A,l,r);
        QuickSort (A,l,p-1);
        QuickSort (A,p+1,r);
    }
Partition (A,l,r)
{   x=l;
    y=r;
    p=A[l];
  while (x<y)
  {   while (A[x]<=p)
          x++;
      while (A[y]>=p)
          y--;
      if (x<y)
          swap (A[x],A[y]);
  }
}
```

A[l] = A[y];
A[y] = p;
return y; /* return position of pivot */
}

Time Complexity of QuickSort Algorithm:

Best Case → Quick sort gives best time complexity when elements are divided into two partitions of equal size; therefore recurrence relation for this case is;

$$T(n) = 2T(n/2) + O(n).$$

By solving this recurrence, we get

$$T(n) = O(n \log n).$$

Worst case → Quick sort gives worst case when elements are already sorted. In this case one partition contains the n-1 elements and another partition contains no element. Therefore, its recurrence relation is;

$$T(n) = T(n-1) + O(n)$$

By solving this recurrence relation, we get

$$T(n) = O(n^2).$$

Average case → It is the case between best case and worst case. All permutation of the input numbers are equally likely. On a random input array, we will have a balanced and unbalanced splits. Good and bad splits are randomly distributed across throughout the tree. Suppose we are alternate Balanced, Unbalanced, Balanced.

$$B(n) = 2\,UB(n/2) + \Theta(n) \text{ Balanced.}$$
$$UB(n) = B(n-1) + \Theta(n) \text{ Unbalanced.}$$

Solving: $B(n) = 2(B(n/2-1) + \Theta(n/2)) + \Theta(n)$
$$= 2B(n/2-1) + \Theta(n)$$
$$= \Theta(n \log n).$$

(b) Merge Sort:

It is an efficient sorting algorithm which involves merging two or more sorted files into a third sorted file. Merging is the process of combining two or more sorted files into a third sorted file. The merge sort algorithm is based on divide and conquer method.

The process of merge sort can be formalized into three basic operations.

i) Divide the array into two sub arrays.
ii) Recursively sort the two sub arrays.
iii) Merge the newly sorted sub arrays.

Algorithm:

1. Start
2. Split the unsorted list into groups recursively until there is one element per group.
3. Compare each of the elements and then group them.
4. Repeat step 2 until the whole list is merged and sorted in the process.
5. Stop.

Pseudo code:

```
MergeSort (A, l, r)
 {
   If (l<r)
    {
      m=⌊(l+r)/2⌋          //Divide
      MergeSort (A, l, m)   //Conquer
      MergeSort (A, m+1, r) //Conquer
      Merge (A, l, m+1, r)  //Combine
    }
 }

Merge (A, B, l, m, r)
  { x=l;
    y=m;
    k=l;
    while (x<m && y<r)
    { if (A[x] < A[y])
      {
```

```
        B[k] = A[x];
        k++;
        x++;
    }
    else
    {
        B[k] = A[y];
        k++;
        y++;
    }
}
while (x<m)
{
    A[k] = A[x];
    k++;
    x++;
}
while (y<r)
{
    A[k] = A[y];
    k++;
    y++;
}
for (i=l; i<=r; i++)
    A[i] = B[i]
}
```

Time Complexity:

No. of sub-problems = 2

Size of each subproblem = $n/2$

Dividing cost = constant

Merging cost = n.

Thus recurrence relation for Merge sort is;

$T(n) = 1$ if $n = 1$

$T(n) = 2T(n/2) + O(n)$ if $n > 1$

By solving this recurrence relation, we get,

Time Complexity = $T(n) = O(n \log n)$.

Scanned with CamScanner

(c) Heap Sort:

Heap is a special tree-based data structure, that satisfies the following special heap properties:

i) Shape property → Heap data structure is always a complete binary tree, which means all levels of the tree are fully filled.

ii) Heap property → All nodes are either greater than or equal to or less than or equal to each of its children. If the parent nodes are greater than their child nodes, heap is called a Max-Heap, and if the parent nodes are smaller than their child nodes, heap is called Min-Heap.

⇒ Heaps can be used in sorting an array. In Max-heaps, maximum element will always be at the root.

Algorithm:

Consider an array Arr which is to be sorted using Heap Sort.

1. Initially build a max heap of elements in Arr.
2. The root element, that is Arr[1], will contain maximum element of Arr. After that swap this element with the last element of Arr and heapify the max element of Arr excluding the last element which is already in its correct position and then decrease the length of heap by one.
3. Repeat the step 2, until all the elements are in their correct position.

Pseudo Code:

```
heap_sort(Arr[])
{
    heap_size = N;
    build_maxheap(Arr);
    for (i=N; i>=2; i--)
        {
            swap(Arr[1], Arr[i]);
            heap_size = heap_size-1;
            max_heapify(Arr, 1, heap_size);
        }
}
```

Complexity Analysis of Heap Sort (i.e, Efficiency)

Worst case Time Complexity : $O(n \log n)$

Best Case Time Complexity: $O(n \log n)$

Average Case Time Complexity: $O(n \log n)$.

Space Complexity: $O(1)$

⊛ Comparision of various sorting algorithms:

Sorting technique	Worst case	Average case	Best Case	Comment
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$	
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Unstable
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Require extra memory
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Large constant
Quick sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	Small constant