Sorting - UNIT V

* One of the most Common applications in Computer Science is sort, the process through which data are arranged according to their values.

* Sorteng is One of the most Common data-processing applications

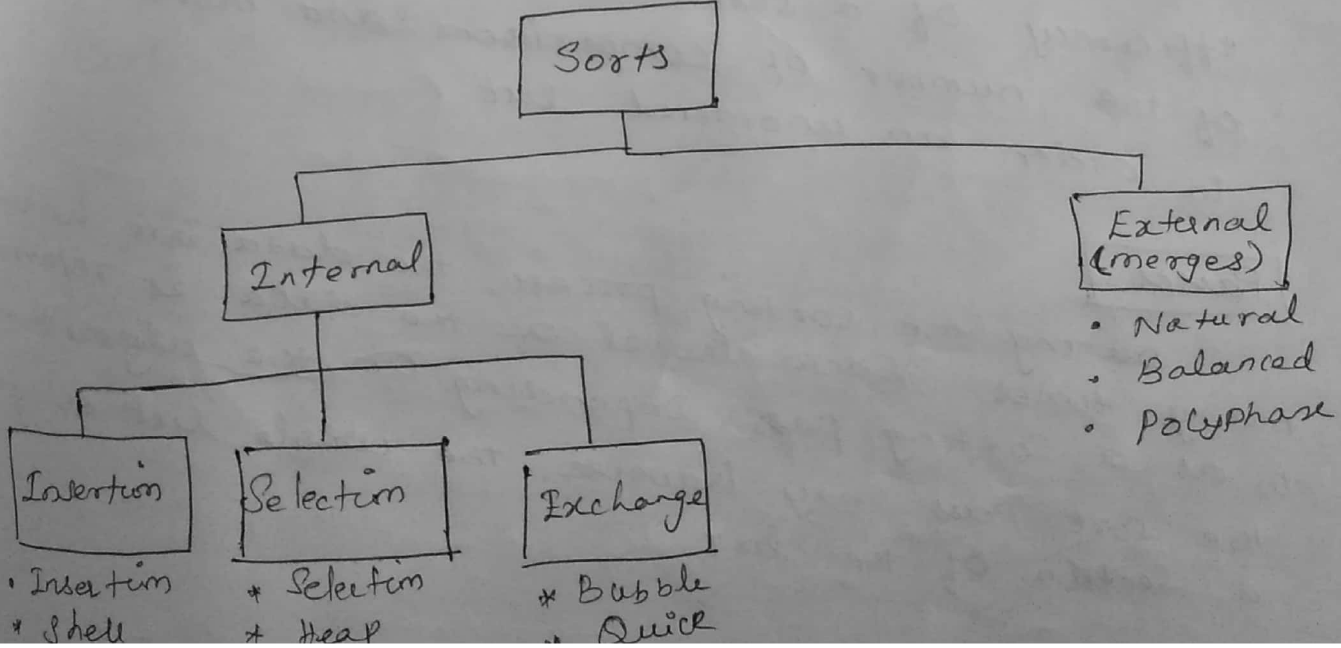
Sort classification

* Sorts are generally classifed as either internal or external sorts

* Internal Sort → all of the data are held in primary memory during the sorting process

* External Sort uses primary memory for the data currentey being sorted and secondary storage for any data that does not fit in primary memory

eg: a file of 20,000 records may be sorted Using an array that holds only 1000 records. During the sorting process, only 1000 records are therefore in memory at any one time, the other 19,000 are kept in a file in Secondary storage.



Internal sorting algorithms have been grouped into several different classifications depending on their general approach to sorting.

Sort order

- Data may be sorted in either ascending sequence or descending sequence.

- The sort order identifies the sequence of the sorted data, ascending or decending.

- If the order of the sort is not specified, it is assumed to be ascending.

- ~~Eg~~ Examples of common data sorted is ascending sequence are the dictionary and the telephone book

- Examples of common decending data are percentages of games won in the sporting event such as baseball or grade point averages for honor students.

Sort Efficiency

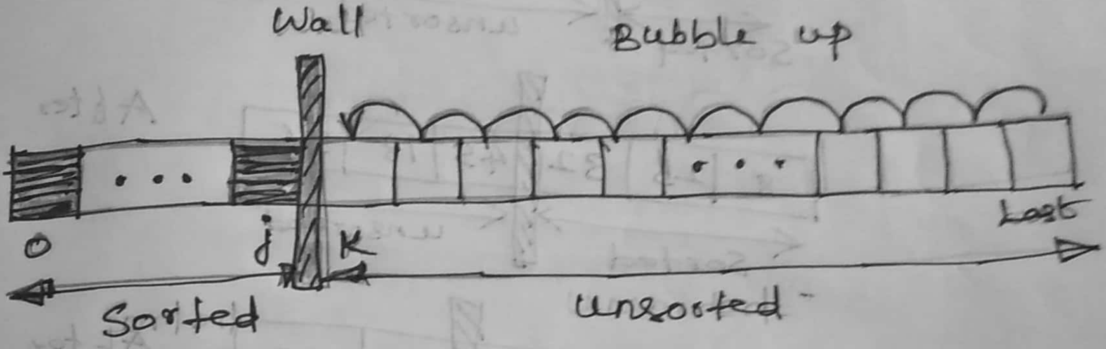
It is a measure of the relative efficiency of a sort. It is usually an estimate of the number of comparisons and moves required to order an unordered list.

Passes

During the sorting process, the data are traversed many times. Each travel of the data is referred to as a sorting pass. Depending on the algorithm, the sort pass may traverse the whole list or just a section of the list.

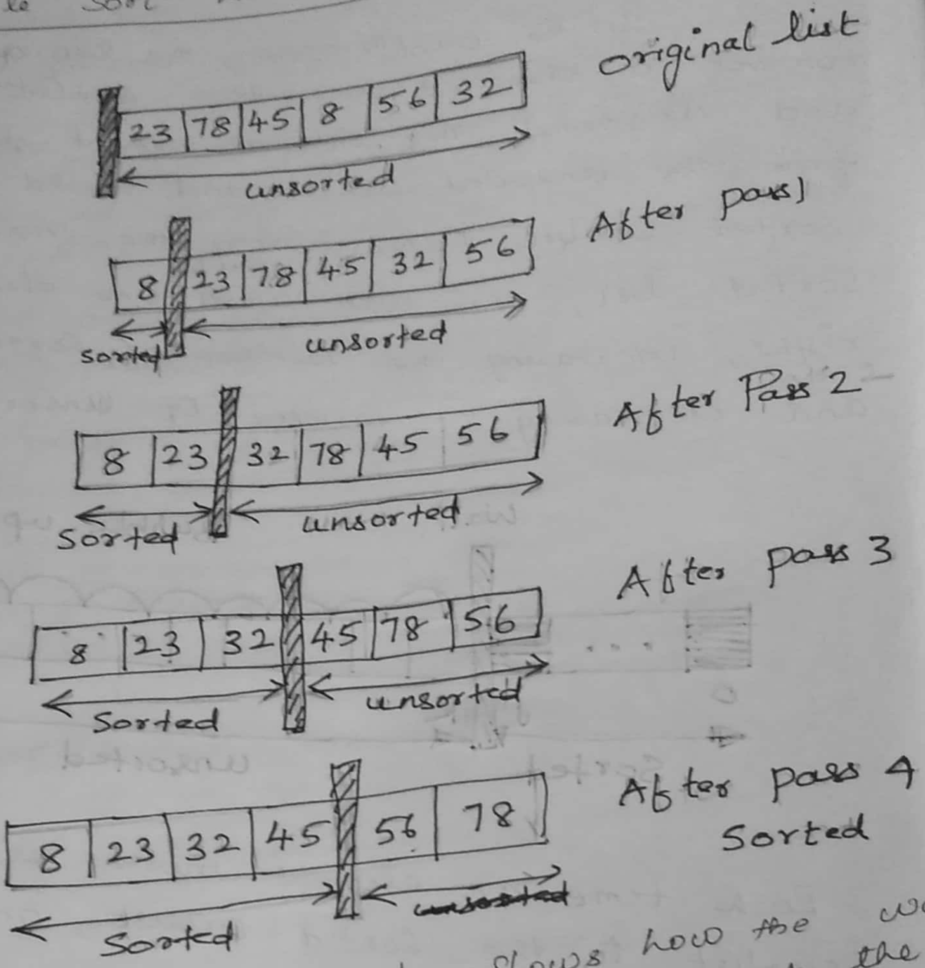
Bubble Sort

In the bubble sort, the list at any moment is divided into two sublists: sorted and unsorted. The smallest element is bubbled from the unsorted sublist and moved to the sorted sublist. After moving the smallest to the sorted list, the wall moves one element to the right, increasing the number of sorted elements and decreasing the number of unsorted ones.



Each time an element moves from the unsorted sublist to the sorted sublist, one sort pass is completed. Given a list of n elements, the bubble sort requires up to n-1 passes to sort the data.

Bubble Sort Example



The above example shows how the wall moves one element in each pass. Looking at the first pass, we start with 32 and compare it with 56. Because 32 is less than 56, we exchange the two and step down one element. We then compare 32 and 8. Because 32 is not less than 8, we do not exchange these elements. We step down one element and compare 45 and 8. They are out of sequence, so we exchange them and step down again. Because we moved 8 down, it is now compared with 78, and these two elements are exchanged. Finally, 8 is compared with 23 and exchanged. this series of exchanges places 8 is the first location, and the wall is moved up one position.

Bubble Sort Algorithm Bubble up smallest element

```
void   bubble_sort (int arr[], int n)
{
  int i, j;
  for(i=0; i<n-1; i++) till unsorted position
  {          last
    for (j=n-1; j>=i; j--)
     {
       if (arr[j]<= arr[j-1])
        {
           int temp= arr[j];
           arr[j] = arr[j-1];
           arr[j-1]= temp;
        }
     }
}  }
```

Optimized Implementation

The above algorithm runs $\Theta(n^2)$ time even if the array is sorted. It can be optimized by Stopping the algorithm if inner loop did not cause any swap i/p: 64, 34, 25, 12, 22, 11, 90

```
void bubble_sort(int arr[], int n)
{ int i,j;
  int sorted=0;
  for(i=0; i<n-1; i++)
  {
    sorted=1;
    for(j=n-1; j>=i; j--)
     {
       if (arr[j] <= arr[j-1])
        {
           int temp = arr[j];
           arr[j] = arr[j-1];
           arr[j-1] = temp;
           sorted=0;
        }
     }
    if (sorted == 1)
    break;
  }
}
```

bubble down

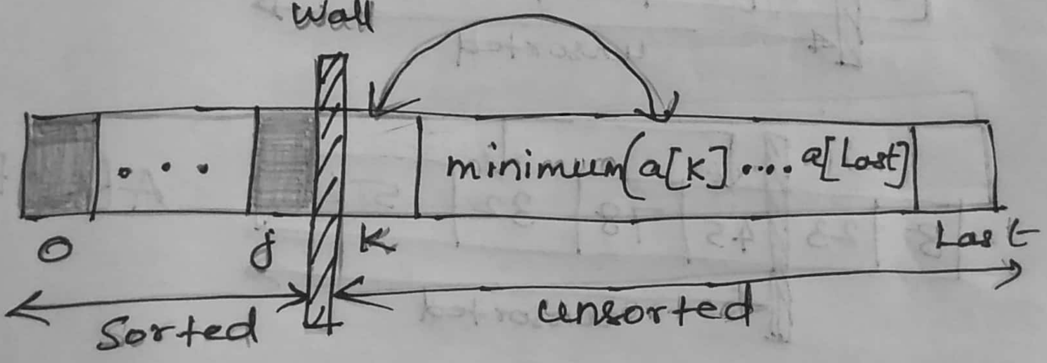
Bubble sort algorithm

```
void bubble-sort(int arr[], int n)
 {
   int i,j;
   for (i=0; i<n-1; i++)
   {
     for(j=0; j<n-i-1; j++)
      {
        if (arr[j]>= arr[j+1])
         {
           int temp = arr[j];
           arr[j] = arr[j+1];
         } arr[j+1] = temp;
      }
   }
}
```

Selection Sort

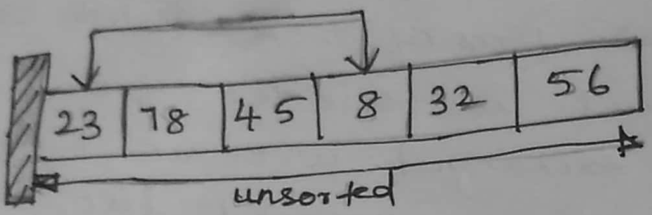
Selection Sorts are among the most intuitive of all sorts. Given a list of data to be sorted, we simply select the smallest item and place it in a sorted list. These steps are then repeated until we have sorted all of the data.

In each pass of the ~~the~~ selection sort, the smallest element is selected from the unsorted sublist and exchanged with the element at the begining of the unsorted list.

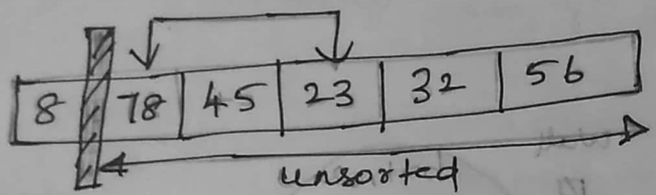


In the selection sort, the list at any moment is divided into two sublists, sorted and unsorted, which are divided by an imaginary wall. we select the smallest element from the unsorted sublist and exchange it with the element at the begining of the unsorted data. After each selection and exchange, the wall moves one element, between the two sublists increasing the number of sorted elements and decreasing the number of unsorted ones. Each time we move one element from the unsorted Sublist to the Sorted Sublist, we say that we have Completed one sort pass.

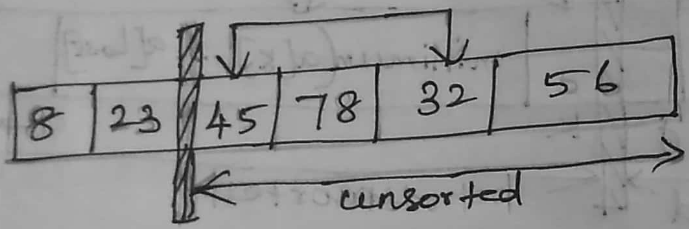
If we have a list of n elements, therefore, we need n-1 passes to completely rearrange the data.



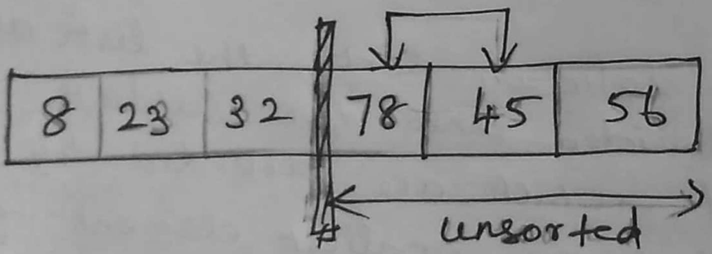
Original list



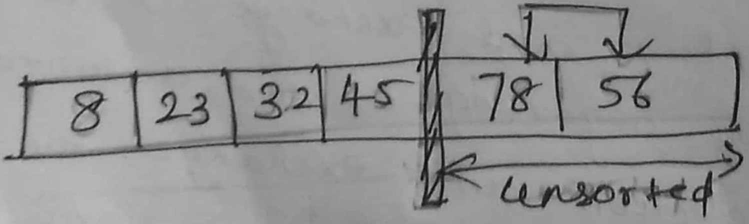
After Pass 1



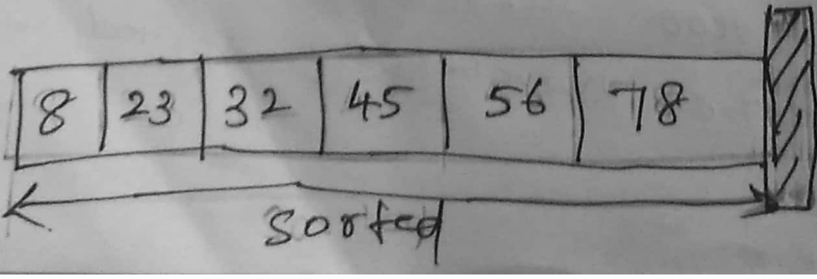
After Pass 2



After Pass 3



After Pass 4



After Pass 5

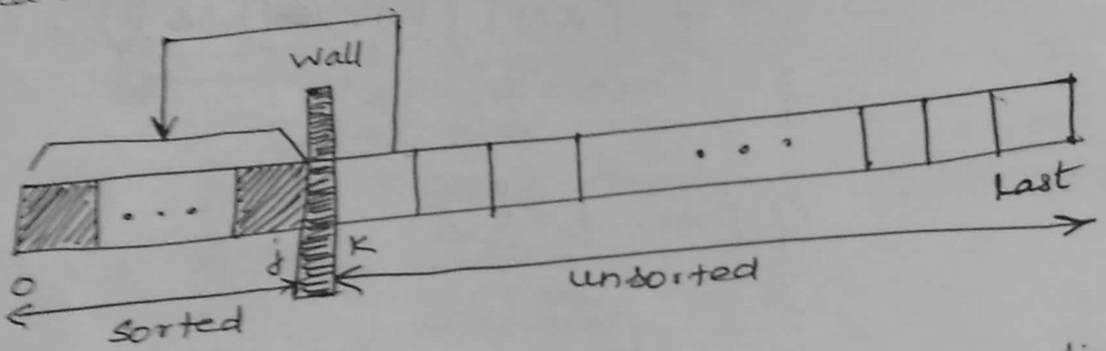
Selection Sort Algorithm

```
void  Selection_Sort (int arr[], int n)
{
     int  i, j;
     for(i=0;  i<n-1; i++)
      {
            int  min = i;
            for (j=i+1 ; j<n; j++)
              {
                   if (arr[j]< =arr[min])
              }      min=j;

              if (min != i)
              {
               int temp = arr[min];
                arr[min] = arr[i];
                 arr[i] = temp;
              }
      }
}
```

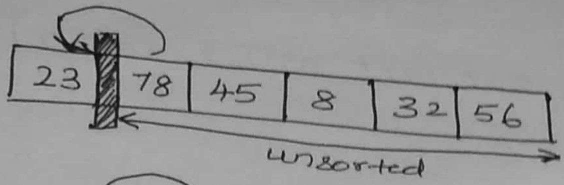
Insertion Sort

Insertion sorting is one of the most common sorting techniques used by card players. As they pick up each card, they insert it into the proper sequence in their hand. The concept extends well into computer sorting. In each pass of an insertion sort, one or more pieces of data are inserted into their ~~correcta~~ correct location in an ordered list.

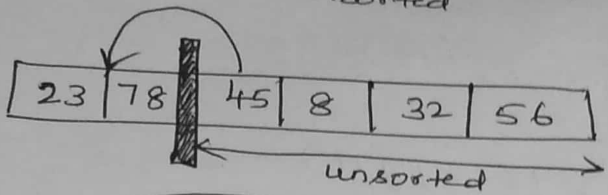


In the [straight] insertion sort, the list is divided into two parts: sorted and unsorted. In each pass the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place. If we have a list of n elements, it will take at most n-1 passes to sort the data.

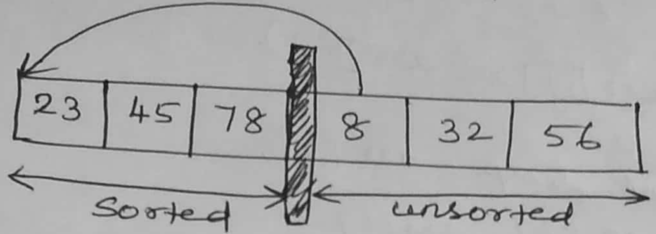
Insertion Sort Example



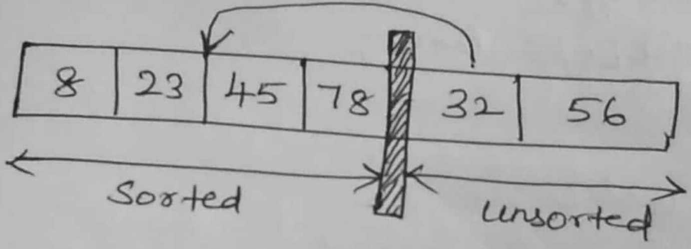
Original list



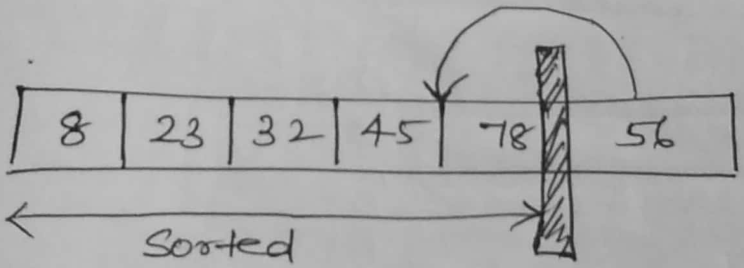
After Pass 1



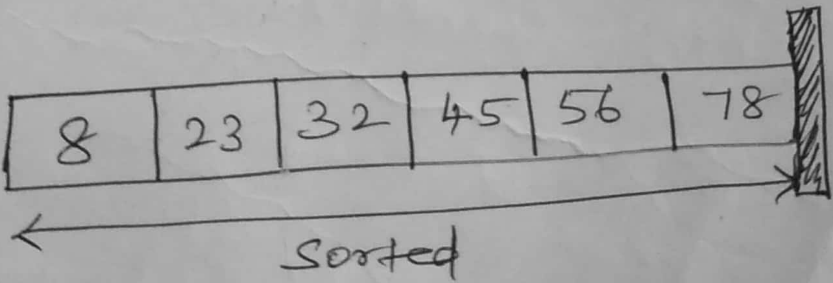
After pass 2



After pass 3



After pass 4



After pass 5

Insertion Sort Algorithm

```
void  insertion_sort (int arr[ ], int n)
{
  int i,j;
  for(i=0; i<n-1; i++)
  {
     for(j=0; j<=i; j++)
     {
        if(arr[j]>=arr[i])
        {
          int temp = arr[j];
          arr[j]= arr[i];
          arr[i]= temp;
        }
     }
  }
}
```

Counting Sort

Counting sort is a sorting technique based on keys between a specific range.

It works by counting the number of objects having distinct key values, and using arithmetic on these counts to determine the positions of each key value in the output sequence.

Consider the data in the range 0 to 9.

Input data : 1, 4, 1, 2, 7, 5, 2

Step 1 Take a count array ~~such that each element~~ to store the count of each unique object.

Index →	0	1	2	3	4	5	6	7	8	9
Count	0	2	2	0	1	1	0	1	0	0

The modified count array indicates the position of each object in the output sequence.

Steps output each object from the input sequence followed by decreasing its count by 1

	0	1	2	3	4	5	6	7	8	9
output	1	1	2	2	4	5	7			

Counting Sort.

Algorithm

```
void Countsort (int arr[N])
{
     int count[10], i, output[N];
// Intialize Count array to Zero
   for(i=0; ~~count[i~~ i<N; i++)
       ++count [arr[i]];

   for(i=[illegible] ; i<=9 ; i++)
       count[i] = count[i] + count [i-1];

   for(i=0; i<N; i++)
   {
       output [count [arr[i]]-1] = arr[i];
       --count [arr[i]];
   }
  for (i=0; i<N; i++)
     arr[i] = output[i];
}
```

Radix Sort

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

Example

Original, unsorted list

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives.

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives.

2, 24, 45, 66, 75, 90, ~~802~~ 170, 802

```
void  countsort (int arr[], int n, int place)
{
   int i, freq[range] = {0}; // range for integers is 10 as digits range from 0-9
   int output[n];
   for(i=0; i<n; i++)
        freq[(arr[i]/place)%range]++;

   for(i=1; i<range; i++)
        freq[i] = freq[i] + freq[i-1];

   for (i=n-1; i>=0; i--)
   {
     output [freq [arr[i] / place) %range] -1] = arr[i];
     freq [(arr[i] / place)% range]--;

   }
  for (i=0; i<n; i++)
       arr[i] = output[i]
}

Void radixsor(int arr[], int n, int maxx)
{
   int mul = 1;
   while (maxx) // maxx is the maximum ele in the array.
   {
      countsort(arr, n, mul);
      mul = mul*10;
      maxx = maxx/10;
   }
}
```