Insertion Sort

→ The main idea behind Insertion Sort is that it inserts each item into its proper place in the final list. To save memory, most implementation of the Insertion sort algorithm works by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding values until it is in its corred place.

- Insertion Sort is less efficient as compared to other more advanced algorithms such as $\boxed{\text{QUICK SORT, HEAP SORT AND MERGE SORT.}}$.

Techniques :→

Insertion Sorts works :→ One that stores

→ The array of values to be sorted is divided into two sets.
sorted values and another that contains unsorted values.

→ Sorting proceeds until there are elements in the unsorted set.

→ Suppose there are ~~x~~ n elements in the array. Initially, the elements with index 0 (assuming LB=0) is in the sorted set. Rest of the elements are in the unsorted set.

→ The first elements of the unsorted partition has array index 1 (if LB=0).

→ During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

```
INSERTION-SORT (ARR, N)

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.
```

COMPLEXITY OF INSERTION SORT

For Insertion Sort, $\boxed{\text{the Best case}}$ occur when the array is already sorted. In this case, the running time of the algorithm has a linear Running Time i.e **O(n)**. This is because, during each iteration, the first elements from the unsorted set is compared only with the last element of the sorted set of the array.

The WORST CASE → occurs when array is in Reverse Sorted Order. In this scenerio, The first element of the Unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before Inserting the next element.

Therefore in the Worst Case Scenerio → INSERTION SORT has a QUADRATIC RUNNING TIME i.e $O(n^2)$.

Even in the Average Case → the Insertion Sort algorithm will have to make atleast $(K-1)/2$ comparisons.
Thus the average case also has a Quadratic Running Time.

ADVATAGES OF INSERTION SORT

→ efficient and easy to implement to use on small sets of data.

→ efficiently implemented on data sets that are already substaintially sorted.

→ it requires less memory space only $O(1)$ of additional memory space.

→ it is said to be ONLINE, → As it can sorted a list as and when it receives new elements.

→ insertion Sort is faster than Bubble Sort and selection sort.

- → ~~faster~~ Insertion Sort 2Times faster than Bubble Sort.
- → Insertion Sort 40% faster than Selection Sort.

→ Insertion Sort is simpler than Shell Sort, with only a small trade off in efficiency.

INSERTION SORT.

```
# include <stdio.h>
# include <conio.h>
# define size 5
void insertion-sort (int arr[], int n);
void main()
{
  int arr[size], i, n;
  printf("\n Enter the number of elements in the array:");
  scanf ("%d", &n);
  printf ("\n Enter the elements of the array: ");
  for(i=0 ; i<n; i++)
     {
       scanf ("%d", &arr [i]);
     }
void insertion-sort (int arr[], int n)
  {
    int i, j, temp;
    for(i=1 ; i<n; i++)
      {  temp = arr [i];
         j= i-1;
       while ((temp < arr [j] && (j>=0))
         {
           arr [j+1] = arr[j];
           j--;
         }
        arr[j+1] = temp;
      }
```

Ex.

39	9	45	63	18	81	108	54	72	36

Unsorted list

39	9	45	63	18	81	108	54	72	36

A[0] is the only element in sorted list.

PASS 1:→

9	39	45	63	18	81	108	54	72	36

Sorted Unsorted List

PASS 2:→

9	39	45	63	18	81	108	54	72	36

Sorted unsorted list

PASS 3:→

9	39	45	63	18	81	108	54	72	36

Sorted unsorted list

PASS 4:→

9	18	39	45	63	81	108	54	72	36

Sorted unsorted.

PASS 5:→

9	18	39	45	63	81	108	54	72	36

SORTED UNSORTED

PASS 6:→

9	18	39	45	63	81	108	54	72	36

SORTED UNSORTED

PASS 7 :→

9	18	39	45	54	63	81	108	72	36

SORTED UNSORTED

PASS 8 : →

9	18	39	45	54	63	72	81	108	36

PASS 9 : →

9	18	36	39	45	54	63	72	81	108

To insert an element A[K] in a sorted list A[0], A[1], ..., A[K-1] we need to compare A[K] with A[K-1], then with A[K-2], A[K-3], and so on until we meet an element A[J] such that A[J] <= A[K].

In order to insert A[K] in its correct position, we need to move elements A[K-1], A[K-2], ..., A[J] by one position and then A[K] is inserted at the $(J+1)^{th}$ location.

STRAIGHT INSERTION SORT

→ Suppose there are N keys in an input list.
→ It requires N iterations to sort the entire list.

Straight insertion sort involves the following basic operations in any iteration j $(0 \le j \le N-1)$ when there are j number keys in the output list such that $K_1 \le K_2 \le K_3 \le \dots \le K_j$.

(1). Select the $(j+1)$th key K from the input, which has to be inserted into the output list.

(2). Compare the key K with, K_j, K_{j-1}, .. etc. in the output list successively (from right to left) until discovering that K should be inserted b/w K_i and K_{i+1}, that is, $K_i \le K \le K_{i+1}$

(3) Move keys $K_{i+1}, K_{i+2}, \dots K_j$ in the output list up one space to the right.

(4) Put the key K into the position $i+1$.

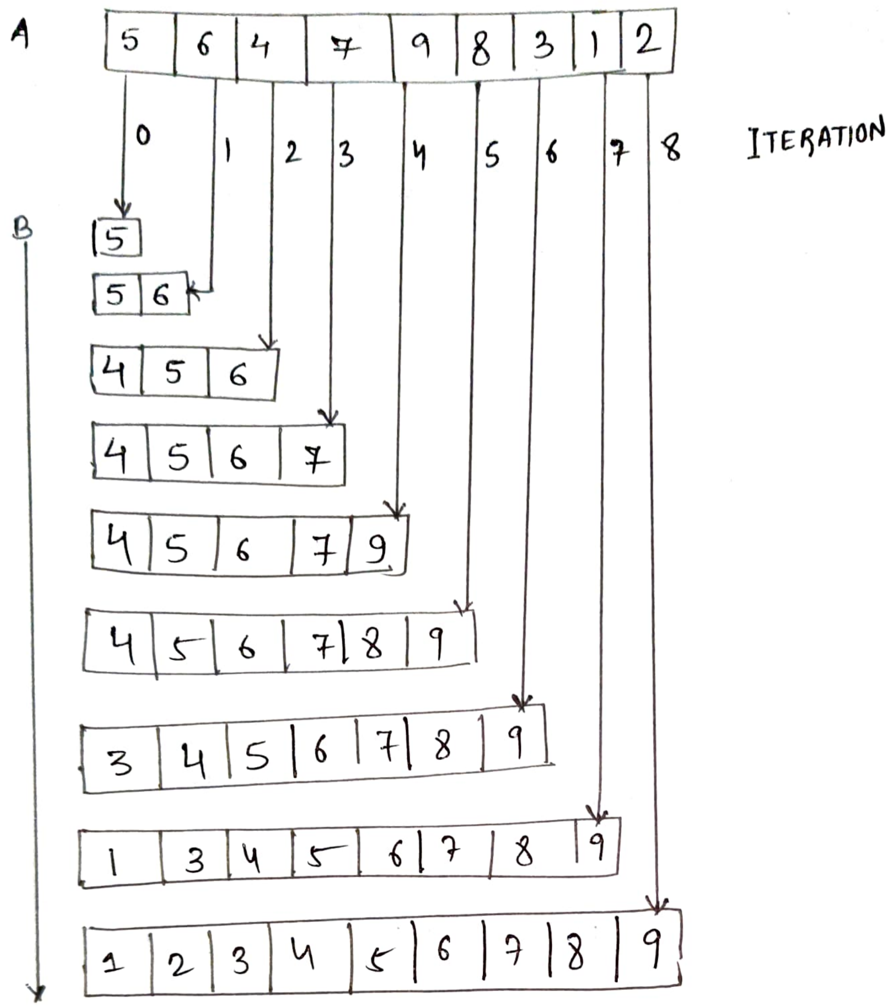


Illustration of Straight Insertion Sort

Analysis of the algorithm StraightInsertionSort

Case 1:→ The input list already in sorted order.

No. of comparisons

→ In this case the number of comparison in each iteration is 1.

→ The total number of iterations required is $n-1$, where n is the size of the list.

Let $C(n)$ denote the total number of comparisons to sort a list of size n. Therefore

$$C(n) = 1+1+1+\dots+1 \text{ upto } (n-1)^{th} \text{ iterations}$$

$$= n-1.$$

* Number of movements

In this, data movement is not there. in any iteration.

Hence, the number of movement $M(n)$ is

$$M(n) = 0.$$

* Memory Requirement

In this, case, an additional storage space other than the storage space for the Input list is required to store the output list.

Hence, the size of storage space required to run the StraightInsertion Sort is

$$S(n) = n.$$

Case 2: The input list is stored but in reverse order.

→ Number of comparisons

It is observed that the ith iteration requires i numbers of comparisons. Hence considering the total $n-1$ iteration, we have

$$C(n) = 1+2+3+ \cdots + (n-1)$$
$$= \frac{n(n-1)}{2}$$

→ Number of movement

The number of keys that needs to be moved in ith iteration is i. Therefore the total number of key movement is

$$\boxed{M(n) = 1+2+3+ \cdots + (n-1)}$$
$$\boxed{= \frac{n(n-1)}{2}}$$

→ Memory Requirement

Here also we do not need any extra storage space other than to store the output list. Hence

$$\boxed{S(n) = n}$$

Case 3: Input list is in random order.

→ Number of comparisons.

- Let us consider the $(i+1)^{th}$ iteration, when there are i keys in the o/p list and we want to insert the $(i+1)$th key into the list.

- Note that in this case there are i number of elements in the o/p list.

- Noted that there are $(i+1)$ locations where the key may go.

- If P_j be the probability that the key may go to the j^{th} location $(0 \le j \le i+1)$ then the number of comparisons will be $j \cdot P_j$.

Hence average number of comparisons in the $(i+1)$th iteration is

$$A_{i+1} = \sum_{j=1}^{i+1} j \cdot P_j$$

To simplify the analysis, let us assume that all keys are distinct and all permutation of keys are equally likely. Then

$$\boxed{P_1 = P_2 = P_3 = \cdots = P_{i+1} = \frac{1}{i+1}}$$

Therefore with this assumption, we have the average number of comparisons in the $(i+1)$th iteration as

$$A_{i+1} = \frac{1}{i+1} \sum_{j=1}^{i+1} j$$

$$= \frac{1}{i+1} \cdot \frac{(i+1) \cdot (i+2)}{2} = \frac{(i+2)}{2}.$$

Hence the total number of comparisons for all $(n-1)$ iteration is

$$C(n) = \sum_{i=0}^{n-1} A_{i+1}$$

$$= \sum_{i=0}^{n-1} \left(\frac{i}{2}+1\right)$$

$$= \frac{1}{2}\sum_{i=0}^{n-1} i + (n-1)$$

$$= \frac{1}{2} \cdot \frac{n(n-1)}{2} + (n-1)$$

NUMBER OF MOVEMENTS

On the average, the number of movements in the i^{th} iteration can be calculated with the same assumptions as in the case of calculation of number of comparisons, i.e

$$M_i = \frac{i+(i-1)+(i-2)+ \cdots +2+1}{i} = \frac{i+1}{2}$$

Hence, the total number of movements $M(n)$ to sort a list of n numbers in random order is

$$M(n) = \sum_{i=1}^{n-1} M_i$$

$$= \frac{1}{2}\sum_{i=1}^{n-1} i + \left(\frac{n-1}{2}\right)$$

$$= \frac{1}{2} \cdot \frac{n(n-1)}{2} + \frac{n-1}{2}$$

Time Complexity of Straight Insertion Sort.

	Run Time	Complexity	Remark
Case 1	$T(n) = c(n-1)$	$T(n) = O(n)$	Best Case
Case 2	$T(n) = cn(n-1)$	$T(n) = O(n^2)$	Worst Case
Case 3	$T(n) = c\frac{(n-1)(n+3)}{2}$	$T(n) = O(n^2)$	Average case.

The time complexity ($T(n)$) of the algorithm Straight Insertion Sort can be calculated considering No. of comparisons and no. of movements, i.e

$$\boxed{T(n) = t_1 \cdot C(n) + t_2 \cdot M(n)}$$

where t_1 and t_2 denotes the time required for a unit comparison and movement respectively.

Analysis of Algorithm Straight Insertion Sort.

Case	Comparison	Movement	Memory	Remarks
Case 1	$C(n) = n-1$	$M(n) = 0$	$S(n) = n$	Input list in sorted order.
Case 2	$C(n) = \frac{n(n-1)}{2}$	$M(n) = \frac{n(n-1)}{2}$	$S(n) = n$	Input list in sorted order but reverse.
Case 3	$C(n) = \frac{(n-1)(n+4)}{4}$	$M(n) = \frac{(n-1)(n+2)}{4}$	$S(n) = n$	Input list in random order.

LIST INSERTION SORT

→ Two major operation in Insertion Sort →

(a) Scan the list to find the location of Insertion.

(b) move keys to accomodate the key.

The straight insertion sort assumes that the list of keys is ~~sorted~~ stored in an array, that is, using the Sequential storage allocation.

The problem with this data structure is that it is necessary to move a large number of keys in order to accomplish each insertion operation.

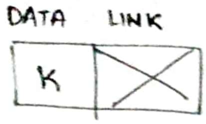
→ In contrast to this, we know that linked allocation --> Ideally suited for Insertion, since only a few links needs to be changed and other operation that is scanning with linked allocation is as easy as with sequential allocation.

To Insert a node, a Iteration in the list Insertion Sort comprises the following Steps →

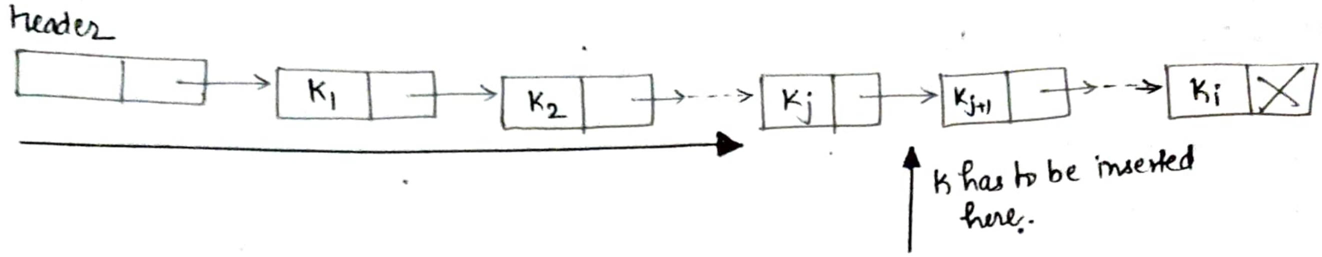
(1). Allocate memory for a node to be inserted.

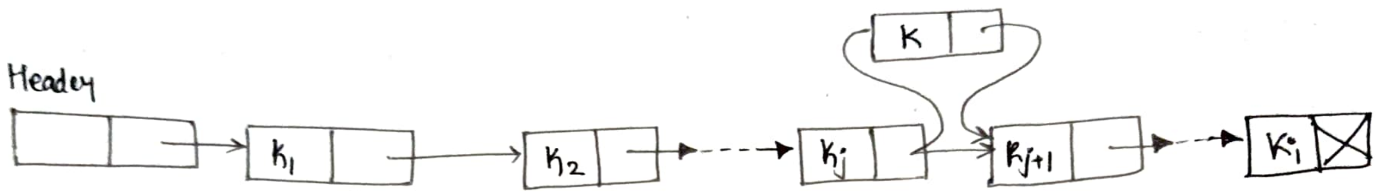
(2). Scan the list to the right starting from the header node in the o/p list to find its place of Insertion.

(3) Insert the node.



Allocate memory for a node to store k





Analysis of the Algorithm ListInsertionSort

CASE 1: The input list is already in sorted order

NUMBER OF COMPARISON

→ The number of comparisons in the ith iteration is i (because there are i number of elements in the o/p list).

→ Now the total number of iterations required in the algo is n-1, where n is the size of the input list.

→ Let C(n) denote the total number of comparisons to sort a list of size n.

$$C(n) = 1+2+3+\cdots+(n-1)$$
$$= \frac{n(n-1)}{2}$$

NUMBER OF MOVEMENTS.

This technique does not require any data movement unlike the straight insertion sort. However, the algorithm requires changes in two link fields. With this consideration, the number of movements $M(n)$ stands as

$$\boxed{M(n) = 2(n-1)}$$

MEMORY REQUIREMENT.

This methods needs to maintain $(n+1)$ nodes including the header node to maintain the o/p list with n key values.

Let us consider that a node needs three units (one unit for key and two units for the link field) memory to hold a key.

Then the size of storage space required is

$$\boxed{S(n) = 3(n+1)}$$

CASE 2: THE INPUT LIST IS SORTED BUT IN REVERSE ORDER

Number of Comparisons

In this case any iteration requires exactly one comparison. Hence, considering a total $(n-1)$ iterations, we have

$$C(n) = 1+1+1+\cdots+1 \text{ upto } (n-1) \text{ terms}$$

$$= n-1$$

Number of Movements

Therefore, the total number of key movements is

$$\boxed{M(n) = 2(n-1)}$$

Memory Requirement

The Storage space required in this case is

$$\boxed{S(n) = 3(n+1)}$$

CASE 3: INPUT LIST IS RANDOMLY ORDERED

Number of comparisons

let us consider the ith iteration, when there are i keys in the output list and we want to insert the $(i+1)$th key into the list.

This key can be inserted in any location such as in the front, at the end or after the jth node from the front.

- There is only one Comparison, if the key is inserted in the front.
- There are i number of comparisons when the key is inserted at the end.
- Similarly, j numbers of comparisons are required to insert the key after the jth node $(1 < j \leq i)$.
- Assuming that all keys are distinct and all permutations of keys are equally likely, then number of comparisons required to find the place of insertion of $(i+1)$th key at any ith iteration, on the avg is,

$$A_i = \frac{\text{Sum of all comparisons for all possible locations}}{\text{Number of possible locations}}$$

$$A_i = \frac{1+2+3+\cdots+i}{i} = \frac{\cancel{i}(i+1)}{2\times\cancel{i}} = \frac{i+1}{2}$$

Total number of comparisons for all $(n-1)$ iterations is

$$C(n) = \sum_{i=1}^{n-1} A_i$$

$$= \frac{1}{2}\sum_{i=1}^{n-1}(i+1)$$

$$= \frac{1}{2}(n-1) + \frac{1}{2}\sum_{i=1}^{n-1} i$$

$$\boxed{C(n) = \frac{(n-1)}{2} + \frac{1}{2}\cdot\frac{n(n-1)}{2} \Rightarrow \frac{(n-1)(n+2)}{4}}$$

NUMBER OF MOVEMENTS

The number of key movement is same as in Case1 and Case 2 of this technique. Therefore the total number of key movement is

$$\boxed{M(n) = 2(n-1)}$$

MEMORY REQUIREMENT

$$\boxed{S(n) = 3(n+1)}$$

ANALYSIS OF ALGORITHM LIST INSERTION SORT

Case	Comparison	Mort.	Memory	Remark
Case 1:	$C(n) = \frac{n(n-1)}{2}$	$M(n) = 2(n-1)$	$S(n) = 3(n+1)$	Input list is in sorted order.
Case 2:	$C(n) = n-1$	$M(n) = 2(n-1)$	$S(n) = 3(n+1)$	I/p list is sorted in reverse order.
Case 3:	$C(n) = \frac{(n-1)(n+2)}{4}$	$M(n) = 2(n-1)$	$S(n) = 3(n+1)$	I/p list is in random order.

TIME COMPLEXITY OF ALGORITHM LIST INSERTION SORT

Case	RunTime T(n)	Complexity	Remark
Case 1:	$T(n) = c\left[\frac{n(n-1)}{2} + 2\right]$	$T(n) = O(n^2)$	Worst case
Case 2:	$T(n) = c(n+1)$	$T(n) = O(n)$	Best case
Case 3:	$T(n) = c\left[\frac{(n-1)(n+2)}{4} + 2\right]$	$T(n) = O(n^2)$	Average case.