**============INSERTION SORT – THEORY-=============**

**Algorithmically, we have the first index of the array as 1. Last index of the array is n.**

**In a program, you may start with 0 as the start index (provided by C programming environment by default ) and** n-1 **as the last index.**

* **Insertion sort is an incremental sorting algorithm. This means it considers only 1 element of the given input array, at a time. The way it does is to consider 1 element of input array (which has not been considered yet) and find it’s correct position.**
* **For this, we need to have a sorted array somewhere. Given such a sorted array, Insertion sort will insert a newly considered element into its proper position in a given sorted array. This way, the size of the sorted array increases by 1.**
* **Finally, when we consider the last untouched element and put it into its correct position in already sorted array of size** n-1**, the entire given input array will be sorted in, say non-decreasing i.e increasing order.**

**Some things to be clarified:-**

* **Which part of the array should be sorted to facilitate incoming new element into it? We can logically/visually partition array A into 2 parts. So called left side of the array which we can call conveniently as LSA(Left sorted array) and keep unconsidered elements into right side and call it conveniently RUSA( Right unsorted array).**
* **Entire strategy rests upon an assumption that there is some sorted array on the left side to start with. How do we make sure we have this assumption CORRECT or TRUE at the start so that we can implement strategy?**

**At the leftmost side, we have a single element of the array i.e A[1]. This forms an array of size 1. Any array of size 1 is by default sorted. Referring to a point mentioned just above, the first LSA to start with will be LSA=A[1].**

**Given above points, the first version of Insertion sort is:-**

|  |  |
| --- | --- |
| Insertion Sort(A) | |
| 1 | LSA=A[1] |
| 2 | while(RUSA is not empty) |
| 3 | { |
| 4 | pick up 1 element , say x, from RUSA |
| 5 | Find correct position of x in LSA |
| 6 | reduce RUSA |
| 7 | } |
| Insertion Sort      --------- strategy algorithm | |

**Further Question:**

* **Which element** x **to pick up to be inserted, in step 4 ?**
* **We can pick up any element from RUSA as a candidate for x.**
* **To be systematic and for making things convenient, we pick up first element of RUSA as x. That is, we pick up leftmost element of RUSA as** x**. But again, we can choose any element.**
* **How to find correct position of** x **?**
* **We can use any method of searching correct position of** x **in LSA. We can use binary search as LSA is already sorted. This will speedup searching process.**
* **In an answer to previous question, suppose LSA is array fragment** A[1…i]**. So, RSA will be** Ai+…n**.  We have picked up** x=A[i+1]**. So we can also search correct position for** x **in** LSA=A[1…i] ***backwards through linear search starting from position*** i **and going down  all the way to** 1**, if at all necessary.**

|  |  |
| --- | --- |
| Insertion Sort(A) | |
| 1 | LSA=A[1] |
| 2 | while(RUSA is not empty) |
| 3 | { |
| 4 | x= pick up first element of RUSA |
| 5 | Find correct position of x in LSA |
| 6 | reduce RUSA |
| 7 | } |
| Insertion Sort      --------- Clear strategy algorithm | |

**Implementing all these as actual programming constructs, we get following algorithms. Choice of an algorithm depends on whether we use swapping or not. Swapping is a costly operation, requiring 3 assignment operations per swap. So whenever we can, we should avoid swapping, as it is a costlier memory write function.**

**A.5 Algorithm:**

**A.5.1:**

|  |  |
| --- | --- |
| **Insertion Sort (A, n)** | |
| 1 | ***For***  k=2 to n |
| 2 | i=k-1;                                 // *we want to fix A[k]* |
| 3 | ***While*** i>0 and A[i]>A[i+1]     //  *Go backwards starting from current position* |
| 4 | Swap(Ai+1,Ai);         // *A[k] will move backwards through swaps,* |
| 5 | i=i-1; |
| **Algo-0    Insertion Sort Pseudocode (Backward search and using swapping)** | |

==============Searching backwards and Using swapping to reach to relatively correct position============

**====Searching backwards and avoiding swapping i.e using copy to right assignments==**

|  |  |
| --- | --- |
| **Insertion Sort (A, n)** | |
| 1 | ***For***  k=2 to n |
| 2 | currentval= A[k];                         //  *Insert A[k] into it’s proper position in A[1…k-1]* |
| 3 | i=k-1; |
| 4 | ***While*** i>0 and A[i]>currentval     //  *Go backwards starting from current position* |
| 5 | Ai+1=Ai;        //  *Shift elements greater than currentval 1 position right,*  *to make place* |
| 6 | i=i-1; |
| 7 | Ai+1=currentval;                   //   *Correct position found…copy currentval to this*  *position* |
| **Algo-1 Insertion Sort Pseudocode (Backward search and no swapping…copy to right instead)** | |

**==============Searching Forward without swapping ========================**

|  |  |
| --- | --- |
| **Insertion Sort (A,n)** | |
| 1 | **for**(i=2; i<=n; i++) |
| 2 | {           position=1; |
| 3 | outside=A[i]; |
| 4 | **for**(j=1;j<=i-1;j++) |
| 5 | { |
| 6 | if( outside > A[j]) |
| 7 | { |
| 8 | position++; |
| 9 | } |
| 10 | Else |
| 11 | break; |
| 12 | }           for(j=i-1; j>position; j--) // can be implemented as a block copy |
| 13 | A[j+1]=A[j]; |
| 14 | A[position+1]=outside; |
| 15 | } |
| **Algo-2   Insertion Sort Pseudocode (Forward search and no swapping…copy to right )** | |

Making some efficiency arrangements i.e removing comparisons which are unnecessary, programmatically, faster algorithm would be as below

|  |  |
| --- | --- |
| **Insertion Sort (A,n)** | |
| 1 | **for**(i=2; i<=n; i++) |
| 2 | { |
| 3 | outside=A[i]; |
| 4 | **for**(j=1;j<=i-1;j++) |
| 5 | { |
| 6 | if( outside > A[j]) |
| 9 | continue; |
| 10 | Else |
| 11 | break; |
| 12 | } |
| 13 | position=j; |
| 14 | for(j=i-1; j>position; j--) // can be implemented as a block copy |
| 15 | A[j+1]=A[j]; |
| 16 | A[position+1]=outside; |
| 17 | } |
| **Algo-3   Insertion Sort Pseudocode (Forward search and no swapping…copy to right )** | |