Brute Force Apporach: Selection Sort:-

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last n − 1 elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the ith pass through the list, which we number from 0 to n − 2, the algorithm searches for the smallest item among the last n − i elements and swaps it with Ai:

After n − 1 passes, the list is sorted.

Here is pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array:

ALGORITHM SelectionSort(A[0..n − 1])

//Sorts a given array by selection sort

//Input: An array A[0..n − 1] of orderable elements //Output: Array A[0..n − 1] sorted in nondecreasing order for i ← 0 to n − 2 do

min ← i

for j ← i + 1 to n − 1 do

if A[j] < A[min] min ← j swap A[i] and A[min]

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison A[j] < A[min]. The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

89	45	68	90	29	34	**17**
17 |	45	68	90	**29**	34	89
17	29 |	68	90	45	**34**	89
17	29	34 |	90	**45**	68	89
17	29	34	45 |	90	**68**	89
17	29	34	45	68 |	90	**89**
17	29	34	45	68	89 |	90

Brute Force Apporach: Bubble Sort:-

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n - 1$ passes the list is sorted. Pass i ($0 \le i \le n - 2$) of bubble sort can be represented by the following diagram:

Here is pseudocode of this algorithm.

ALGORITHM BubbleSort(A[0..n − 1])

//Sorts a given array by bubble sort

//Input: An array A[0..n − 1] of orderable elements //Output: Array A[0..n − 1] sorted in nondecreasing order for i ← 0 to n − 2 do

for j ← 0 to n − 2 − i do

if A[j + 1] < A[j] swap A[j] and A[j + 1]

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort:

89 $\overset{?}{\leftrightarrow}$ 45 68 90 29 34 17
45 89 $\overset{?}{\leftrightarrow}$ 68 90 29 34 17
45 68 89 $\overset{?}{\leftrightarrow}$ 90 $\overset{?}{\leftrightarrow}$ 29 34 17
45 68 89 29 90 $\overset{?}{\leftrightarrow}$ 34 17
45 68 89 29 34 90 $\overset{?}{\leftrightarrow}$ 17
45 68 89 29 34 17 | 90

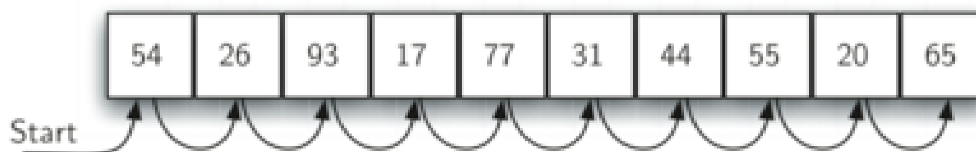
45 $\overset{?}{\leftrightarrow}$ 68 $\overset{?}{\leftrightarrow}$ 89 $\overset{?}{\leftrightarrow}$ 29 34 17 | 90
45 68 29 89 $\overset{?}{\leftrightarrow}$ 34 17 | 90
45 68 29 34 89 $\overset{?}{\leftrightarrow}$ 17 | 90
45 68 29 34 17 | 89 90

etc.

Sequential Search:-

When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship. Each data item is stored in a position relative to the others. In Python lists, these relative positions are the index values of the individual items. Since these index values are ordered, it is possible for us to visit them in sequence. This process gives rise to our first searching technique, the sequential search.

Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items. If we run out of items, we have discovered that the item we were searching for was not present.



String Matching:-

String Matching Algorithm is also called "String Searching Algorithm." This is a vital class of string algorithm is declared as "this is the method to find a place where one is several strings are found within the larger string."

Given a text array, T [1.....n], of n character and a pattern array, P [1......m], of m characters. The problems are to find an integer s, called valid shift where $0 \leq s < n\text{-}m$ and T [s+1......s+m] = P [1......m]. In other words, to find even if P in T, i.e., where P is a substring of T. The item of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b..... z}.

Given a string T [1......n], the substrings are represented as T [i......j] for some $0 \leq i \leq j \leq n-1$, the string formed by the characters in T from index i to index j, inclusive. This process that a string is a substring of itself (take i = 0 and j =m).

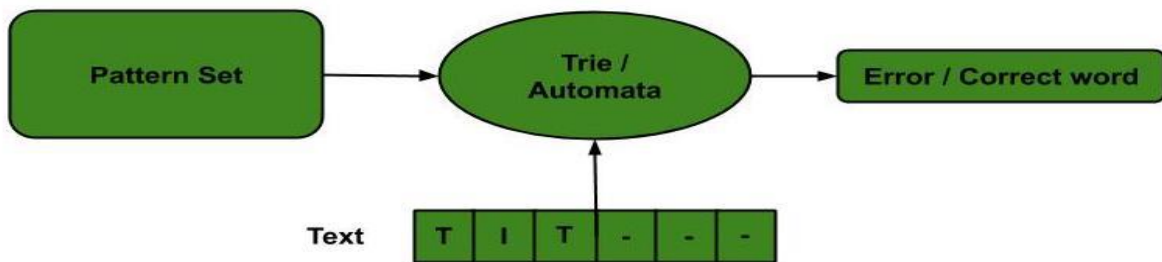
The proper substring of string T [1......n] is T [1......j] for some $0 < i \leq j \leq n-1$. That is, we must have either $i > 0$ or $j < m-1$.

Using these descriptions, we can say given any string T [1......n], the substrings are

T [i.....j] = T [i] T [i +1] T [i+2]......T [j] **for** some $0 \leq i \leq j \leq n-1$.

And proper substrings are

T [i.....j] = T [i] T [i +1] T [i+2]......T [j] **for** some $0 \leq i \leq j \leq n-1$.



If $i > j$, then T [i.....j] is equal to the empty string or null, which has length zero.