Data Structures and Algorithm

Moazzam Ali Sahi

Lecture # 8

Searching an Array

Last Lecture

- List implementation using Arrays
- Memory management
- Basic terminologies
- Basic operations on Arrays

This Lecture

- Searching Methods
  - Sequential search
  - Binary search
  - Interpolation search

Searching

- Searching is an operation or a technique that helps finds the place of a given element or value in the list.
- Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not.
- Some of the standard searching technique that is being followed in the data structure is listed below:
  - Linear Search or Sequential Search
  - Binary Search

Linear Search

- Linear search is a very simple search algorithm.
- In this type of search, a sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Linear Search [Algorithm]

```
Linear Search ( Array A, Value x)

Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Linear Search [Algorithm]

```
procedure linear_search (list, value)

   for each item in the list
      if match item == value
         return the item's location
      end if
   end for

end procedure
```

Time Complexity

- Best Case Time
- Average Case
- Worst Case

Time Complexity [Best Case Time]

- The Best Case will take place if:
  - The element to be search is on the first index.
  - The number of comparisons in this case is 1.
- Therefore, Best Case Time Complexity of Linear Search is O(1).

Time Complexity [Average Case Time]

- Let there be N distinct numbers: a_1, a_2, …, $a_{(N-1)}$, a_N. We need to find element P.
- There are two cases:
  - Case 1: The element P can be in N distinct indexes from 0 to N-1.
  - Case 2: There will be a case when the element P is not present in the list.
- There are N case 1 and 1 case 2.
- So, there are N+1 distinct cases to consider in total.
- If element P is in index K, then Linear Search will do K+1 comparisons.
- Number of comparisons for all cases in Case 1

= Comparisons if element is in index 0 +
Comparisons if element is in index 1 +
… + Comparisons if element is in index N-1
= 1 + 2 + … + N
= $\frac{N*(N+1)}{2}$ comparisons

Time Complexity [Average Case Time]

- If element P is not in the list, then Linear Search will do N comparisons.
- Number of comparisons for all cases in case 2 = N
- Therefore, total number of comparisons for all N+1 cases

$$= \frac{N*(N+1)}{2} + N$$

$$= N * (\frac{N*(N+1)}{2} + 1)$$

Average number of comparisons = (N * ((N+1)/2 + 1)) / (N+1)

= N/2 + N/(N+1)

- The dominant term in "Average number of comparisons" is N/2.
- So, the Average Case Time Complexity of Linear Search is O(N)

Time Complexity [Worst Case Time]

- The worst case will take place if:
  - The element to be search is in the last index
  - The element to be search is not present in the list
- In both cases, the maximum number of comparisons take place in Linear Search which is equal to N comparisons.

- Hence, the Worst-Case Time Complexity of Linear Search is O(N).

- Number of Comparisons in Worst Case: N

Time Complexity [Summary]

- Best Case Time Complexity of Linear Search: O(1)
- Average Case Time Complexity of Linear Search: O(N)
- Worst Case Time Complexity of Linear Search: O(N)
- Space Complexity of Linear Search: O(1)
- Number of comparisons in Best Case: 1
- Number of comparisons in Average Case: N/2 + N/(N+1)
- Number of comparisons in Worst Case: N

Binary Search

- Binary search is a fast search algorithm with run-time complexity of O(log n).
- This search algorithm works on the principle of divide and conquer.
- For this algorithm to work properly, the data collection should be in the sorted form.
- Binary search looks for a particular item by comparing the middle most item of the collection.
- If a match occurs, then the index of item is returned.
- If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item.
- This process continues on the sub-array as well until the size of the subarray reduces to zero.

Binary Search [Algorithm]

- algorithm Binary_Search(list, item)
- Set L to 0 and R to n: 1
- if L > R, then Binary_Search terminates as unsuccessful
- else
- Set m (the position in the mid element) to the floor of (L + R) / 2
- if Am < T, set L to m + 1 and go to step 3
- if Am > T, set R to m: 1 and go to step 3
- Now, Am = T,
- the search is done; return (m)

Binary Search
[Pseudocode]

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1

      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while

end procedure
```

Binary Search working

- The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

- First, we shall determine half of the array by using this formula

```
mid = low + (high - low) / 2
```

- Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.

Binary Search working

- Now we compare the value stored at location 4, with the value being searched, i.e. 31.
- As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

Binary Search working

- We change our low to mid + 1 and find the new mid value again.

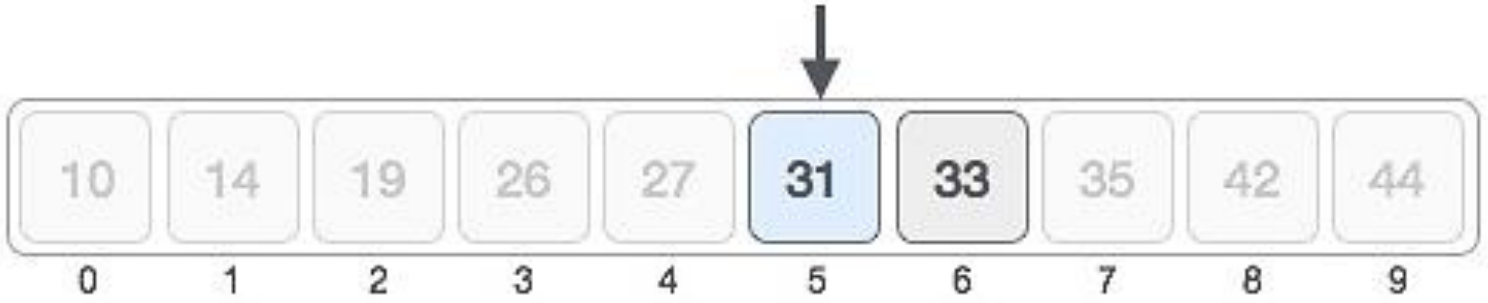
```
low = mid + 1
mid = low + (high - low) / 2
```

- Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

Binary Search working

- The value stored at location 7 is not a match, rather it is more than what we are looking for.
- So, the value must be in the lower part from this location.

- Hence, we calculate the mid again. This time it is 5.



Binary Search working

- We compare the value stored at location 5 with our target value.
- We find that it is a match.

- We conclude that the target value 31 is stored at location 5.
- Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers

Interpolation Search

- Interpolation search is an improved variant of binary search.
- This search algorithm works on the probing position of the required value.
- For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.
- Binary search has a huge advantage of time complexity over linear search.
- Linear search has worst-case complexity of O(n) whereas binary search has O(log n).

Interpolation Search

- There are cases where the location of target data may be known in advance.
- For example, in case of a telephone directory, if we want to search the telephone number of Morphius.
- Here, linear search and even binary search will seem slow as we can directly jump to memory space where the names start from 'M' are stored.

Positioning in Binary Search

- In binary search, if the desired data is not found then the rest of the list is divided in two parts, lower and higher.
- The search is carried out in either of them

Even when the data is sorted, binary search does not take advantage to probe the position of the desired data

Position Probing in Interpolation Search

- Interpolation search finds a particular item by computing the probe position.
- Initially, the probe position is the position of the middle most item of the collection.

Position Probing in Interpolation Search

- If a match occurs, then the index of the item is returned.
- To split the list into two parts, we use the following method –

```
mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

where –
   A    = list
   Lo   = Lowest index of the list
   Hi   = Highest index of the list
   A[n] = Value stored at index n in the list
```

Position Probing in Interpolation Search

- If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item.
- Otherwise, the item is searched in the subarray to the left of the middle item.
- This process continues on the sub-array as well until the size of subarray reduces to zero.
- Runtime complexity of interpolation search algorithm is **O(log (log n))** as compared to **O(log n)** of BST in favorable situations.

Interpolation Search [Algorithm]

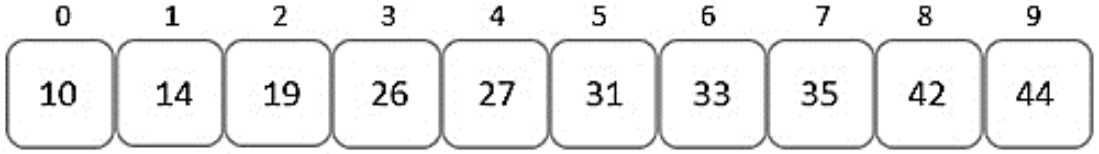
- Step 1 – Start searching data from middle of the list.
- Step 2 – If it is a match, return the index of the item, and exit.
- Step 3 – If it is not a match, probe position.
- Step 4 – Divide the list using probing formula and find the new middle.
- Step 5 – If data is greater than middle, search in higher sub-list.
- Step 6 – If data is smaller than middle, search in lower sub-list.
- Step 7 – Repeat until match.

Interpolation Search

[Pseudocode]

```
Procedure Interpolation_Search()

   Set Lo  →  0
   Set Mid → -1
   Set Hi  →  N-1

   While X does not match

      if Lo equals to Hi OR A[Lo] equals to A[Hi]
         EXIT: Failure, Target not found
      end if

      Set Mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

      if A[Mid] = X
         EXIT: Success, Target found at Mid
      else
         if A[Mid] < X
            Set Lo to Mid+1
         else if A[Mid] > X
            Set Hi to Mid-1
         end if
      end if
   End While

End Procedure
```

Interpolation Search [Example]



Search 19

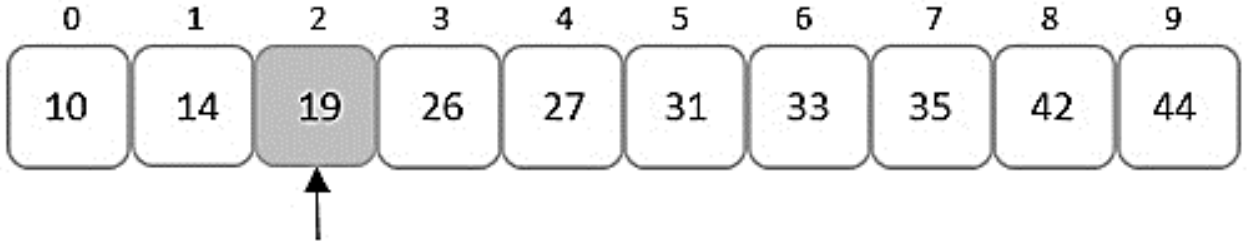
$$mid = Lo + \frac{(Hi - Lo) * (X - A[Lo])}{A[Hi] - A[Lo]}$$

Lo = 0, A[Lo] = 10
Hi = 9, A[Hi] = 44
X = 19

$$mid = 0 + \frac{(9 - 0) * (19 - 10)}{44 - 10}$$

$$mid = \frac{9 * 9}{34}$$

$$mid = \frac{81}{34} = 2.38$$



Sorting

Sorting

- Sorting refers to arranging data in a particular format.
- Sorting algorithm specifies the way to arrange data in a particular order.
- Most common orders are in numerical or lexicographical order.
- The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner.
- Sorting is also used to represent data in more readable formats.
- Following are some of the examples of sorting in real-life scenarios −
  - **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
  - **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

Type of Sorting

- Sorting algorithms may require some extra space for comparison and temporary storage of few data elements.
- These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself.
- This is called in-place sorting.
- Bubble sort is an example of in-place sorting.
- However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted.
- Sorting which uses equal or more space is called not-in-place sorting.
- Merge-sort is an example of not-in-place sorting.

Sorting [Important Terms]

- **Increasing Order**
  - A sequence of values is said to be in increasing order, if the successive element is greater than the previous one.
  - For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.
- **Decreasing Order**
  - A sequence of values is said to be in decreasing order, if the successive element is less than the current one.
  - For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.
- **Non-Increasing Order**
  - A sequence of values is said to be in non-increasing order, if the successive element is less than or equal to its previous element in the sequence.
  - This order occurs when the sequence contains duplicate values.
  - For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.
- **Non-Decreasing Order**
  - A sequence of values is said to be in non-decreasing order, if the successive element is greater than or equal to its previous element in the sequence.
  - This order occurs when the sequence contains duplicate values.
  - For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

Bubble Sort [Algorithm]

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Bubble Sort

[Pseudocode]

```
procedure bubbleSort( list : array of items )

   loop = list.count;

   for i = 0 to loop-1 do:
      swapped = false

      for j = 0 to loop-1 do:

         /* compare the adjacent elements */
         if list[j] > list[j+1] then
            /* swap them */
            swap( list[j], list[j+1] )
            swapped = true
         end if

      end for

      /*if no number was swapped that means
      array is sorted now, break the loop.*/

      if(not swapped) then
         break
      end if

   end for

end procedure return list
```

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array.

Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

Bubble Sort

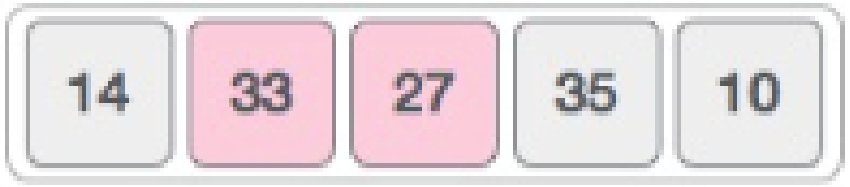
- We take an unsorted array for our example.
- Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

- Bubble sort starts with very first two elements, comparing them to check which one is greater.

Bubble Sort

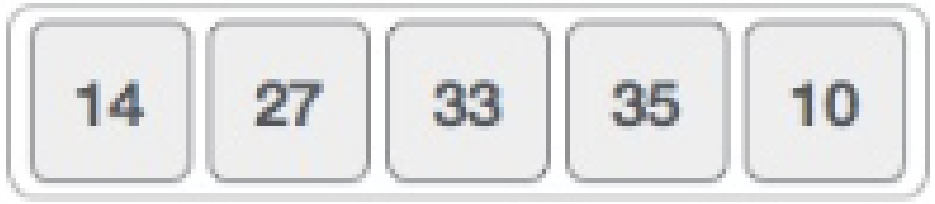
- In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

- We find that 27 is smaller than 33 and these two values must be swapped.

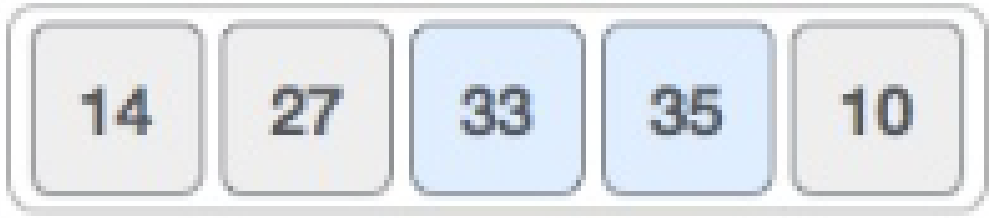


Bubble Sort

- The new array should look like this −



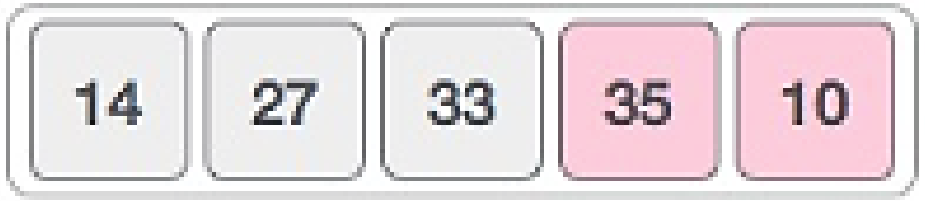
- Next, we compare 33 and 35. We find that both are in already sorted positions.



Bubble Sort

- Then we move to the next two values, 35 and 10

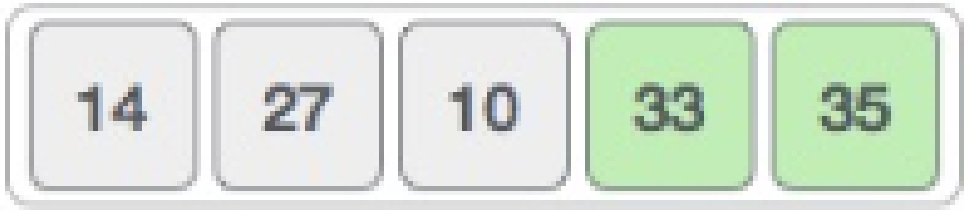
- We know then that 10 is smaller 35. Hence, they are not sorted.



Bubble Sort

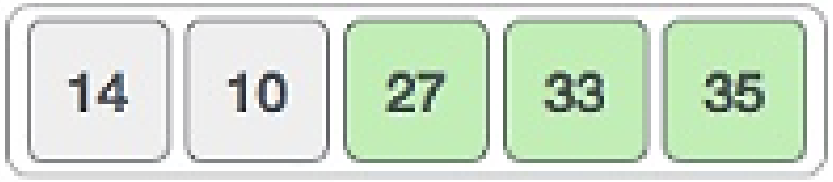
- We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

- To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Bubble Sort

- Notice that after each iteration, at least one value moves at the end.



- And when there's no swap required, bubble sorts learns that an array is completely sorted.

Insertion Sort

- This is an in-place comparison-based sorting algorithm.
- Here, a sub-list is maintained which is always sorted.
  - For example, the lower part of an array is maintained to be sorted.
  - An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there.
- Hence the name, insertion sort.
- The array is searched sequentially, and unsorted items are moved and inserted into the sorted sub-list (in the same array).
- This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$, where n is the number of items.

Insertion Sort [Working]

- We take an unsorted array for our example.

- Insertion sort compares the first two elements.

- It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort [Working]

- Insertion sort moves ahead and compares 33 with 27.

- And finds that 33 is not in the correct position.

- It swaps 33 with 27. It also checks with all the elements of sorted sub-list.

Insertion Sort [Working]

- Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

- By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

- These values are not in a sorted order.

Insertion Sort [Working]

- So, we swap them.

- However, swapping makes 27 and 10 unsorted.

- Hence, we swap them too.

Insertion Sort [Working]

- Again, we find 14 and 10 in an unsorted order.

- We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

- This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort

Insertion Sort [Algorithm]

```
Step 1 - If it is the first element, it is already sorted. return 1;
Step 2 - Pick next element
Step 3 - Compare with all elements in the sorted sub-list
Step 4 - Shift all the elements in the sorted sub-list that is greater than the
         value to be sorted
Step 5 - Insert the value
Step 6 - Repeat until list is sorted
```

Insertion Sort [Pseudocode]

```
procedure insertionSort( A : array of items )
   int holePosition
   int valueToInsert

   for i = 1 to length(A) inclusive do:

      /* select value to be inserted */
      valueToInsert = A[i]
      holePosition = i

      /*locate hole position for the element to be inserted */

      while holePosition > 0 and A[holePosition-1] > valueToInsert do:
         A[holePosition] = A[holePosition-1]
         holePosition = holePosition -1
      end while

      /* insert the number at hole position */
      A[holePosition] = valueToInsert

   end for

end procedure
```

Insertion Sort [Demo]

```
               ----------------Printing Array---------------------
                      20 : 5 : 41 : -45 : 87 : 3 : 2 : 1 : 0 : 9 :
 item moved : 20 Item inserted : 5, at position : 0
Iteration : 1[5 : 20 : 41 : -45 : 87 : 3 : 2 : 1 : 0 : 9 : ]
Iteration : 2[5 : 20 : 41 : -45 : 87 : 3 : 2 : 1 : 0 : 9 : ]
```

```
 item moved : 41
 item moved : 20
 item moved : 5 Item inserted : -45, at position : 0
Iteration : 3[-45 : 5 : 20 : 41 : 87 : 3 : 2 : 1 : 0 : 9 : ]
Iteration : 4[-45 : 5 : 20 : 41 : 87 : 3 : 2 : 1 : 0 : 9 : ]
```

```
 item moved : 87
 item moved : 41
 item moved : 20
 item moved : 5 Item inserted : 3, at position : 1
Iteration : 5[-45 : 3 : 5 : 20 : 41 : 87 : 2 : 1 : 0 : 9 : ]
```

```
 item moved : 87
 item moved : 41
 item moved : 20
 item moved : 5
 item moved : 3 Item inserted : 2, at position : 1
Iteration : 6[-45 : 2 : 3 : 5 : 20 : 41 : 87 : 1 : 0 : 9 : ]
```

```
 item moved : 87
 item moved : 41
 item moved : 20
 item moved : 5
 item moved : 3
 item moved : 2 Item inserted : 1, at position : 1
Iteration : 7[-45 : 1 : 2 : 3 : 5 : 20 : 41 : 87 : 0 : 9 : ]
```

```
 item moved : 87
 item moved : 41
 item moved : 20
 item moved : 5
 item moved : 3
 item moved : 2
 item moved : 1 Item inserted : 0, at position : 1
Iteration : 8[-45 : 0 : 1 : 2 : 3 : 5 : 20 : 41 : 87 : 9 : ]
```

```
 item moved : 87
 item moved : 41
 item moved : 20 Item inserted : 9, at position : 6
Iteration : 9[-45 : 0 : 1 : 2 : 3 : 5 : 9 : 20 : 41 : 87 : ]
```

Insertion Sort

[Demo 2]

```
                ----------------Printing Array--------------------
                        4 : 6 : 3 : 2 : 1 : 9 : 7 :
Iteration : 1           [4 : 6 : 3 : 2 : 1 : 9 : 7 : ]
Iteration : 2
        Item moved : 6
        Item moved : 4
        Item inserted : 3, at position : 0
                [3 : 4 : 6 : 2 : 1 : 9 : 7 : ]
```

```
Iteration : 3
        Item moved : 6
        Item moved : 4
        Item moved : 3
        Item inserted : 2, at position : 0
                [2 : 3 : 4 : 6 : 1 : 9 : 7 : ]
```

```
Iteration : 4
        Item moved : 6
        Item moved : 4
        Item moved : 3
        Item moved : 2
        Item inserted : 1, at position : 0
                [1 : 2 : 3 : 4 : 6 : 9 : 7 : ]
```

```
Iteration : 5           [1 : 2 : 3 : 4 : 6 : 9 : 7 : ]
Iteration : 6
        Item moved : 9
        Item inserted : 7, at position : 5
                [1 : 2 : 3 : 4 : 6 : 7 : 9 : ]
```

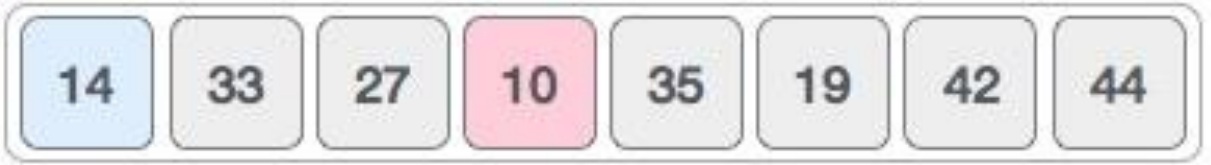
Selection Sort

- Selection sort is a simple sorting algorithm.
- This sorting algorithm is an in-place comparison-based algorithm
- In which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end.
- Initially, the sorted part is empty, and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.
- This process continues moving unsorted array boundary by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$, where n is the number of items.

Selection Sort [Working]

- Consider the following depicted array as an example.

- For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



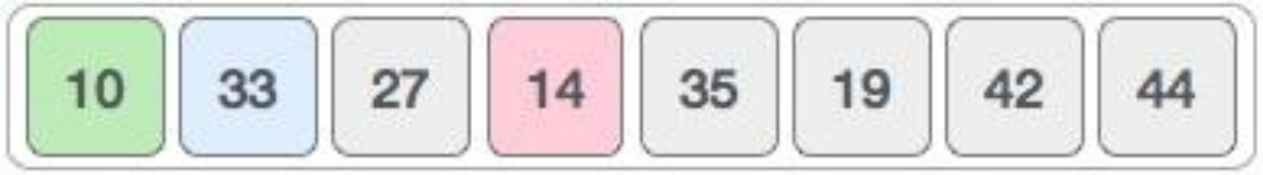
Selection Sort [Working]

- So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

- For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort [Working]

- We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



- After two iterations, two least values are positioned at the beginning in a sorted manner.

- The same process is applied to the rest of the items in the array.

Selection Sort

[Working]

Selection Sort [Algorithm]

```
Step 1 - Set MIN to location 0
Step 2 - Search the minimum element in the list
Step 3 - Swap with value at location MIN
Step 4 - Increment MIN to point to next element
Step 5 - Repeat until list is sorted
```

Selection Sort [Pseudocode]

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```