CSC13: Algorithmics and Program Design

Dr. S. Zeeshan Hussain
Dept. of Computer Science
Jamia Millia Islamia
New Delhi

Unit-Wise Syllabus

- **Algorithmic Problem Solving**: Algorithms; Problem Solving Aspect: Algorithm Devising, Design and Top-down Design; Algorithm Implementation: Essential and Desirable Features of an Algorithm; Efficiency of an Algorithm, Analysis of Algorithms, Pseudo codes; Algorithm Efficiency, Analysis and Order; Importance of Developing Efficient Algorithms; Complexity Analysis of Algorithms: Every-Case Time Complexity, Worst-Case Time Complexity, Average-Case Time Complexity, Best-Case Time Complexity.
- Basic Algorithms – Exchanging the Values of Two Variables, Counting, Summation of a Set of Numbers, Factorial Computation, Sine Function Computation, Generation of the Fibonacci Sequence, Reversing the Digits of an Integer, Base Conversion, etc. Flowchart. Flowchart – Symbols and Conventions, Recursive Algorithms.
- **Factoring:** Finding the square root of number, Smallest Divisor of an integer, Greatest common divisor of two integers, generating prime numbers, computing prime factors of an integer, Generation of pseudo random numbers, Raising a number to a large power, Computing the *n*th Fibonacci number.
- **Arrays, Searching and Sorting: Single and Multidimensional Arrays, Array Order Reversal, Array counting, Finding the maximum number in a set, partitioning an array, Monotones Subsequence; Searching: Linear and Binary Array Search; Sorting: Sorting by selection, Exchange and Insertion. Sorting by diminishing increment, Sorting by partitioning**.
- **Programming:** Introduction, Game of Life, Programming Style: Names, Documentation and Format, Refinement and Modularity; Coding, Testing and Further Refinement: Stubs and Drivers; Program Tracing, Testing, Evaluation; Program Maintenance: Program Evaluation, Review, Revision and Redevelopment; and Problem Analysis, Requirements Specification, Coding and Programming Principles.
- REFERENCES
- **Dromy: How to Solve by Computer, PE (Unit 1-4)**
- **Kruse: Data Structures and Program Design, PHI (Unit-5)**
- Robertson: Simple Program Design, A Step-by-Step Approach, Thomson

Topics to be covered

Single and Multidimensional Arrays

Array Order Reversal

Array counting

Finding the maximum number in a set

Partitioning an array

Searching: Linear and Binary Array Search

Sorting: Sorting by selection, Exchange and Insertion. Sorting by diminishing increment, Sorting by partitioning

Array- Single and Multi-dimensional

```
int a[10]; int a[]={12,13,14};
int *a;
int a[10][10];          int a[][] ={{123},(4,5,6},{7,8,9}};
int *a[10];
char str[10];           char  vowel[]= “aeiou”;
char *str;              char  vowel[]={‘a’, ’e’, ‘i’, ‘o’, ’u’};
char str[10][10];       char *str=“Akash”;
char *str[10];
```

Reversal of array

Given an array (or string), the task is to reverse the array/string.

Examples :

Input : arr[] = {1, 2, 3}

Output : arr[] = {3, 2, 1}

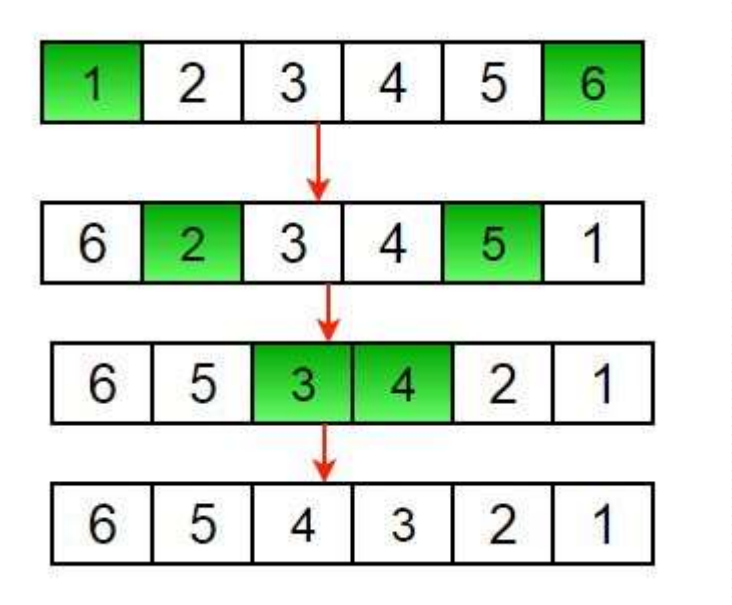
Input : arr[] = {4, 5, 1, 2}

Output : arr[] = {2, 1, 5, 4}

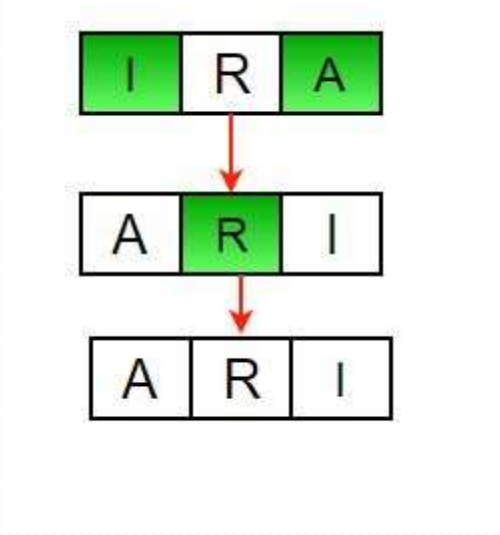
Algorithm-Array Reversal

1) Initialize start and end indexes as start = 0, end = n-1
2) In a loop, swap arr[start] with arr[end] and change start and end as follows : start = start +1, end = end – 1

Reversal of Array



Another example to reverse a string:



C program - array reversal

```
void rvereseArray(int arr[], int start, int
    end)
{
   while (start < end)
   {
      int temp = arr[start];
      arr[start] = arr[end];
      arr[end] = temp;
      start++;
      end--;
   }
}
 /* Utility function to print an array */
void printArray(int arr[], int size)
{
   for (int i = 0; i < size; i++)
   cout << arr[i] << " ";

   cout << endl;
}
```

```
/* Driver function to test above functions */
int main()
{   int arr[] = {1, 2, 3, 4, 5, 6};

   int n = sizeof(arr) / sizeof(arr[0]);
    // To print original array
   printArray(arr, n);
       // Function calling
   rvereseArray(arr, 0, n-1);
       cout << "Reversed array is" << endl;

   // To print the Reversed array
   printArray(arr, n);
       return 0;
}
```

Output :

1 2 3 4 5 6 Reversed array is 6 5 4 3 2 1

Time Complexity : O(n)

Recursive way

1) Initialize start and end indexes as start = 0, end = n-1
2) Swap arr[start] with arr[end]
3) Recursively call reverse for rest of the array.

Recursive Function for array reversal

```
void rvereseArray(int arr[], int start, int
     end)
{
    if (start >= end)
    return;

    int temp = arr[start];
    arr[start] = arr[end];
    arr[end] = temp;

    // Recursive Function calling
    rvereseArray(arr, start + 1, end - 1);
}
  /* Utility function to print an array */
void printArray(int arr[], int size)
{
   for (int i = 0; i < size; i++)
   cout << arr[i] << " ";
    cout << endl;
}
```

```
/* Driver function to test above functions */
int main()
{    int arr[] = {1, 2, 3, 4, 5, 6};

    // To print original array
    printArray(arr, 6);
        // Function calling
    rvereseArray(arr, 0, 5);

    cout << "Reversed array is" << endl;

    // To print the Reversed array
    printArray(arr, 6);
        return 0;
}
```

Output :

1 2 3 4 5 6 Reversed array is 6 5 4 3 2 1

Time Complexity : O(n)

Array Counting

Counting frequencies of array elements

Given an array which may contain duplicates, print all elements and their frequencies.

Examples:

Input : arr[] = {10, 20, 20, 10, 10, 20, 5, 20}

Output :

10 3

20 4

5 1

Input : arr[] = {10, 20, 20}

Output :

10 1

20 2

Algorithm- Counting Array elements

Algorithm

1. Declare and initialize an array arr.
2. Declare another array fr with the same size of array arr. It is used to store the frequencies of elements present in the array.
3. Variable visited will be initialized with the value -1. It is required to mark an element visited that is, it helps us to avoid counting the same element again.
4. The frequency of an element can be counted using two loops. One loop will be used to select an element from an array, and another loop will be used to compare the selected element with the rest of the array.
5. Initialize count to 1 in the first loop to maintain a count of each element. Increment its value by 1 if a duplicate element is found in the second loop. Since we have counted this element and didn't want to count it again. Mark this element as visited by setting fr[j] = visited. Store count of each element to fr.
6. Finally, print out the element along with its frequency.

Program to find frequency of elements in an array

Explanation

In this program, we need to count the occurrence of each unique element present in the array. One of the approach to resolve this problem is to maintain one array to store the counts of each element of the array. Loop through the array and count the occurrence of each element and store it in another array fr.

1	2	8	3	2	2	2	5	1

In the above array, 1 has appeared 2 times, so, the frequency of 1 is 2. Similarly, 2 has appeared 4 times. The frequency of 2 is 4 and so on.

Program to find frequency of elements

```
#include <stdio.h>
  int main()
{
   //Initialize array
   int arr[] = {1, 2, 8, 3, 2, 2, 2, 5
    , 1};
 //Calculate length of array arr
        int length = sizeof(arr)
    /sizeof(arr[0]);
      //Array fr will store frequ
    encies of element
   int fr[length];
   int visited = -1;
       for(int i = 0; i < length; i
    ++){
     int count = 1;
     for(int j = i+1; j < length;
    j++){
         if(arr[i] == arr[j]){
            count++;
ment again
            fr[j] = visited;
        }
     }
     if(fr[i] != visited)
        fr[i] = count;
}
//Displays the frequency of
 each element present in arr
 ay
printf("--------------------
 \n");
printf(" Element | Frequenc
 y\n");
printf("--------------------
 \n");
for(int i = 0; i < length; i++)
 {
   if(fr[i] != visited){
      printf("    %d", arr[i]);
            printf("  %d\n", fr[i]);
       }
   }
   printf("--------------------
    \n");
   return 0;
}
```

Output:

```
------------------------
Element | Frequency
 -------------------------
 1 | 2
 2 | 4
8 | 1
 3 | 1
 5 | 1
 -------------------------
```

Algorithm for finding the maximum number in a set

```
Algorithm straight MaxMin (a, n, max, min)
// Set max to the maximum & min to the minimum of a [1: n]
{
Max = Min = a [1];
For i = 2 to n do
{
If (a [i] > Max) then Max = a [i];
If (a [i] < Min) then Min = a [i];
}}
```

Program to find the Largest value in a set

```
// C program to find maximum
// in arr[] of size n
#include <stdio.h>

int largest(int arr[], int n)
{
    int i;
     // Initialize maximum element
    int max = arr[0];
     // Traverse array elements
    // from second and compare
    // every element with current max
    for (i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];

    return max;
}
 int main()
{
    int arr[] = {10, 324, 45, 90, 9808};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("\nLargest in given array is
     %d " ,largest(arr, n));
    return 0;
}
```

Partitioning an array

```
// A utility function to swap two
    elements
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}
/* This function takes last element as
    pivot, places  the pivot element at its
    correct position in sorted array, and
    places all smaller (smaller than pivot)
    to left of pivot and all greater
    elements to right
  of pivot */
partition (arr[], low, high)
{
// pivot (Element to be placed at right
    position)
   pivot = arr[high];
    i = (low - 1)  // Index of smaller
    element and indicates the
// right position of pivot found so far
    for (j = low; j <= high- 1; j++)
    {
       // If current element is smaller than
    the pivot
       if (arr[j] < pivot)
       {
          i++;    // increment index of
             //smaller element
          swap (&arr[i] , &arr[j]);
       }
    }
    swap (&arr[i + 1] ,&arr[high]);
    return (i + 1);
}
```

Illustration of partition

arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[high] = 70

Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1

j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0

arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

j = 1 : Since arr[j] > pivot, do nothing

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i= 3

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.

Finally we place pivot at correct position by Swapping arr[i+1] and arr[high] (or pivot)

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

Searching: Linear and Binary Array Search

Not even a single day pass, when we do not have to search for something in our day to day life, car keys, books, pen, mobile charger and what not. Same is the life of a computer, there is so much data stored in it, that whenever a user asks for some data, computer has to search it's memory to look for the data and make it available to the user.

What if you have to write a program to search a given number in an array?

How will you do it?

Well, to search an element in a given array, there are two popular algorithms available:

Linear Search

Binary Search

Linear Search

- Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.
- It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1.
- Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

- **Features of Linear Search Algorithm**
- It is used for unsorted and unordered small list of elements.
- It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
- It has a very simple implementation.

Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

The time complexity of Linear search algorithm is **O(n)**, we will analyse the same and see why it is **O(n)** after implementing it.

Implementing Linear Search

Following are the steps of implementation that we will be following:

1. Traverse the array using a for loop.
2. In every iteration, compare the target value with the current value of the array.
   If the values match, return the current index of the array.
   If the values do not match, move on to the next array element.
3. If no match is found, return -1.

To search the number **5** in the array given below, linear search will go step by step in a sequential order starting from the first element in the given array.

8,2,6,3,5

Implementation of Linear Search

```
/* below we have implemented
   a simple function for linear
   search in C - values[] =>
   array with all the values -
   target => value to be found -
   n => total number of
   elements in the array */
int linearSearch(int
   values[], int target, int n)
{
for(int i = 0; i < n; i++)
{ if (values[i] == target) {
return i;
}
}
return -1;
}
```

Inputs:

values[] = {5, 34, 65, 12, 77, 35}
target = 77

Output: 4

Input: values[] = {101, 392, 1, 54, 32, 22, 90, 93}
target = 200

Output: -1 (not found)

Binary Search Algorithm

- Binary Search is applied on the sorted array or list of large size. It's time complexity of **O(log n)** makes it very fast as compared to other sorting algorithms. The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it.

Implementing Binary Search Algorithm

Following are the steps of implementation that we will be following:

Start with the middle element:

If the **target** value is equal to the middle element of the array, then return the index of the middle element.

If not, then compare the middle element with the target value,

If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.

If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.

When a match is found, return the index of the element matched.

If no match is found, then return -1

Implementation of Binary Search(Iterative)

```
#include <stdio.h>
 int iterativeBinarySearch(int
    array[], int start_index, int
    end_index, int element)
{
while (start_index <= end_index)
{
 int middle = start_index +
    (end_index- start_index )/2;
    if (array[middle] ==
    element)       return middle;
   if (array[middle] < element)
    start_index = middle + 1;
 else
  end_index = middle - 1;    }
return -1;
}
```

```
    int main(void)
{    int array[] = {1, 4, 7, 9, 16, 56,
    70};
int n = 7;    int element = 16;
 int found_index =
    iterativeBinarySearch(array, 0,
    n-1, element);
 if(found_index == -1 )
    {      printf("Element not found in
    the array ");
 }   else {
    printf("Element found at index :
    %d",found_index);
}
return 0; }
```

Output

Element found at index : 4

Implementation of Binary Search(Recursive)

```
#include <stdio.h>
int recursiveBinarySearch(int array[], int start_index, int end_index, int element){
  if (end_index >= start_index)
{      int middle = start_index + (end_index - start_index )/2;
    if (array[middle] == element)
return middle;
if (array[middle] > element)
return ecursiveBinarySearch(array, start_index, middle-1, element);
  return recursiveBinarySearch(array, middle+1, end_index, element);
}
return -1;
}
```

```
int main(void){    int array[] = {1, 4, 7, 9, 16, 56, 70};    int n = 7;    int element = 9;
 int found_index = recursiveBinarySearch(array, 0, n-1, element);
if(found_index == -1 )
    {       printf("Element not found in the array ");
  }    else {
     printf("Element found at index : %d",found_index);    }
   return 0;
}
```

Output

Element found at index : 3

Time Complexity of Binary Search (logn)

For n = 8, $\log_2(2^3) = 3$

Output = 3

For n = 256, $\log_2(2^8) = 8$

Output = 8

For n = 2048, $\log_2(2^{11}) = 11$

Output = 11

For n = 8, the output of $\log_2$ n comes out to be 3, which means the array can be halved 3 times maximum, hence the number of steps(at most) to find the target value will be (3 + 1) = 4.

Sorting

Sorting:

Arranging the elements in ascending order or in descending order is called Sorting. Sorting techniques are broadly categorized into two. Internal Sorting and External Sorting.

1. **Internal Sorting** : All the records that are to be sorted are in main memory.
2. **External Sorting**: Some sorts that cannot be performed in main memory and must be done on disk or tape. This type of sorting is known as External Sorting.

Efficiency: One of the major issues in the sorting algorithms is its efficiency. If we can efficiently sort the records then that adds value to the sorting algorithm. We usually denote the efficiency of sorting algorithm in terms of time complexity. The time complexities are given in terms of Big-O notation.

Commonly there are O(n2) and O(n log n) time complexities for various algorithms. Quick sort is the fastest algorithm and bubble sort is the slowest one.

Sorting Methods (Algorithms)

Sorting Method — — — — — — — — — — - Efficiency

Bubble sort/Selection sort/Insertion sort — — — **o(n2)**

Quick / Merge — — — — — — — — — — — **o(n log n)**

Insertion Sort Algorithm

Insertion Sort: Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

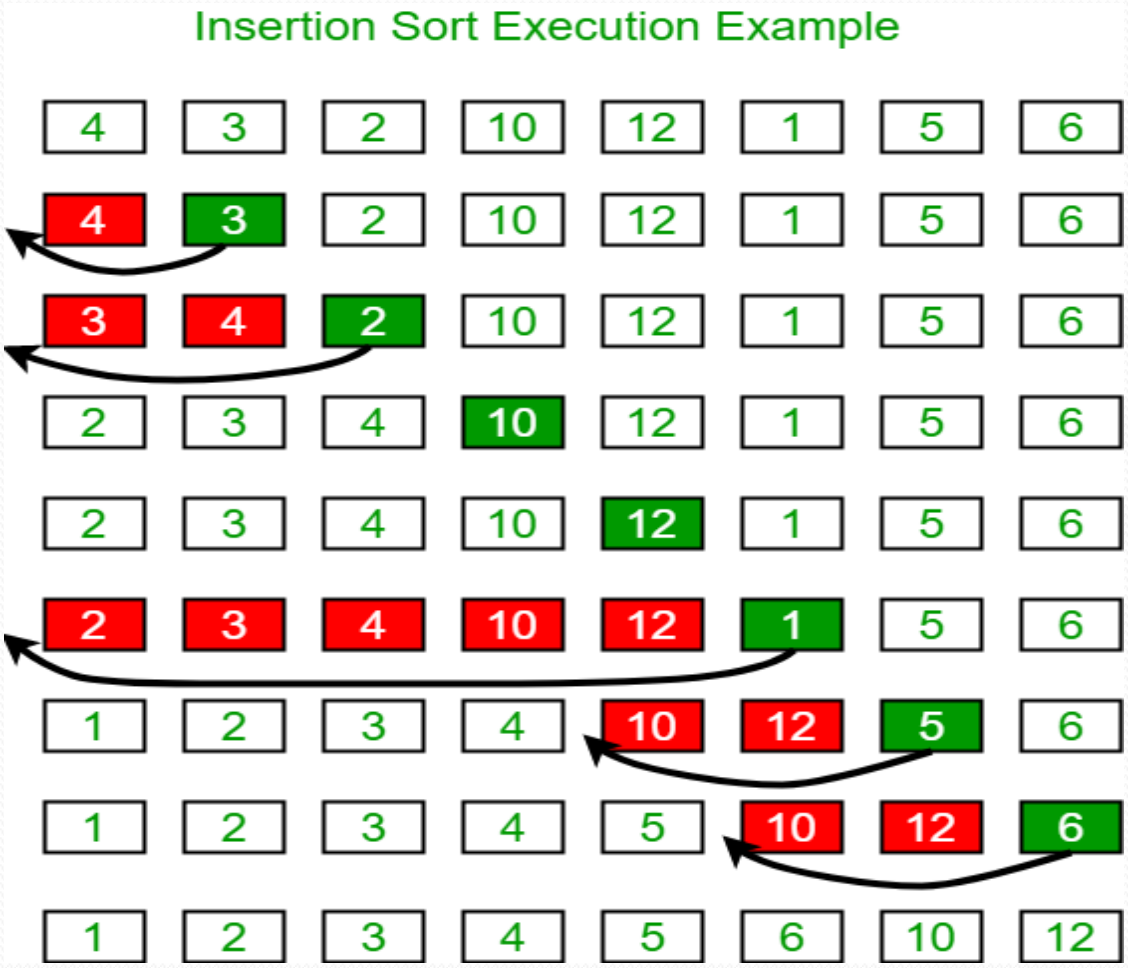
Time Complexity of Insertion Sort

Best Case : O(n) #Means array is already sorted.

Average Case : $O(n^2)$ #Means array with random numbers.

Worst Case : $O(n^2)$ #Means array with descending order.

Insertion Sort Example



Insertion Sort Algorithm

Example:

Let us sort the following numbers using Insertion sort mechanism,

12, 11, 13, 5, 15

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 15

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13

11, 12, 13, 5, 15

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 15

i = 4. 15 will be at position 5 as all the elements are smaller than 15.

5, 11, 12, 13,15

Insertion Sort Implementation in C

```
/* C Program to sort an array in ascending
   order using Insertion Sort */
#include <stdio.h>
int main()
{
int n, i, j, temp;
int arr[64];

printf("Enter number of elements\n");
scanf("%d", &n);

printf("Enter %d integers\n", n);
for (i = 0; i < n; i++)
{
scanf("%d", &arr[i]);
}
for (i = 1 ; i <= n - 1; i++)
{
j = i;
while ( j > 0 && arr[j-1] > arr[j])
{
temp = arr[j];
arr[j] = arr[j-1];
arr[j-1] = temp;
j--;
}
}
printf("Sorted list in ascending order:\n");
for (i = 0; i <= n - 1; i++)
{
printf("%d\n", arr[i]);
}
return 0;
}
```

Program Explanation

1. Using a for loop, we are reading n elements from standard input into an array named arr.
2. Next, we are comparing elements of the array so that we can insert them in the proper position using the insertion sort method.
3. At the end, we are printing/displaying the sorted array.

Runtime Test Cases for Insertion Sort

Here's the run time test cases for Insertion sort algorithm for 3 different input cases.

Test case 1 – Average case: Here, the elements are entered in random order.

/* **Average case** */ Enter number of elements 6

Enter 6 integers 4 6 1 2 5 3 Sorted list in ascending order: 1 2 3 4 5 6

Test case 2 – Best case: Here, the elements are already sorted.

/* **Best case** */ Enter number of elements 3

Enter 3 integers -3 31 66

Sorted list in ascending order: -3 31 66

Test case 3 – Worst case: Here, the elements are reverse sorted.

/* **Worst case** */ Enter number of elements 5

Enter 5 integers 9 8 6 3 1

Sorted list in ascending order: 1 3 6 8 9

Selection Sort algorithm

Selection Sort: The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Time Complexity of Insertion Sort

Best Case : $O(n^2)$ #Means array is already sorted.

Average Case : $O(n^2)$ #Means array with random numbers.

Worst Case : $O(n^2)$ #Means array with descending order.

Selection Sort algorithm

Example:

Following example explains the above steps:

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4] and place it at beginning of arr[1...4]

11 **12** 25 22 64

// Find the minimum element in arr[2...4] and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4] and place it at beginning of arr[3...4]

11 12 22 **25** 64

Implementation of Selection Sort

```
// C program for
    implementation of
    selection sort
#include <stdio.h>
void swap(int *xp, int *yp)  {
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}
 void selectionSort(int
    arr[], int n)  {
   int i, j, min_idx;
    // One by one move
    boundary of unsorted
    subarray
   for (i = 0; i < n-1; i++)
   {
      // Find the minimum
    element in unsorted array
      min_idx = i;
      for (j = i+1; j < n; j++)
      if (arr[j] < arr[min_idx])
         min_idx = j;
  // Swap the found minimum
    element with the first
    element
     swap(&arr[min_idx],
    &arr[i]);
   }
}
/* Function to print an array */
void printArray(int arr[], int
    size)
{
   int i;
   for (i=0; i < size; i++)
      printf("%d ", arr[i]);
   cout << endl;
}
  int main()
{
  int arr[] = {64, 25, 12, 22, 11};
  int n =
   sizeof(arr)/sizeof(arr[0]);
  selectionSort(arr, n);
  printf("\nSorted array: \n");
  printArray(arr, n);
  return 0;
}
```

Output:

Sorted array: 11 12 22 25 64

Time Complexity: $O(n^2)$ as there are two nested loops.

Auxiliary Space: O(1)

Quick Sort

This is the best sort Technique. Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick sort that pick pivot in different ways.
Always pick first element as pivot.
Always pick last element as pivot
You may pick a random element as pivot.
You may pick median as pivot.

Time Complexity :

Best Case : O(nlogn) #Means array is already sorted.
Average Case : **O(nlogn)** #Means array with random numbers.
Worst Case : O(n^2) #Means array with descending order.

Algorithm:

First element is considered as pivot in the implemented code below.
Then we should move all the elements which are less than pivot to one side and greater than pivot to other side.
We divide the array into two arrays in such a way that elements > pivot and elements < pivot.

Quick Sort (Partition Sort)

25,37,12,35,33,87,57,92 (Unsorted List)

Pivot element= 25

Divide the original array around pivot element into two subarrays, one containing values less than the pivot element and the other containing values greater than the pivot element.

(12), 25 ,(37,35,33,87,57,92)

Pivot = 37

12, 25 (35,33) 37 (87,57,92)

Pivot = 35 pivot= 87

12 ,25,(33) ,35 ,37 ,(57) ,87 ,(92)

12,25,33,35,37,57,87,92 (Sorted List)

Quick Sort Algorithm

Step 1 – Make any element as pivot

Step 2 – Partition the array on the basis of pivot

Step 3 – Apply quick sort on left partition recursively

Step 4 – Apply quick sort on right partition recursively

quickSort(arr[], low, high)

```
{ if (low < high)
{ // pivot_index is partitioning index, arr[pivot_index] is now
   at correct place in sorted array
pivot_index = partition(arr, low, high);
quickSort(arr, low, pivot_index - 1); // Before pivot_index
quickSort(arr, pivot_index + 1, high); // After pivot_index
}
}
```

Implementation of Quick sort

Use the partition() and swap() functions defined in earlier slides

```
/* The main function that implements
    QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
    {
   if (low < high)    {
/* pi is partitioning index, arr[p] is now at
    right place */
      int pi = partition(arr, low, high);
 // Separately sort elements before
 // partition and after partition recursively
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}
```

```
{
Int arr[] = {34,25,12,37,57,43};
int n = sizeof(arr)/arr[0];

quicksort(arr,0,n-1);

printf("\nThe sorted array is: \n");

for(i=0; i<n; i++){
printf("%d ", arr[i]);
}
return 0;
}
```

Output:

The sorted array is:

12 25 34 37 43 57

Q & A