

**DEERWALK INSTITUTE OF TECHNOLOGY**

**Tribhuvan University**

**Faculties of Computer Science**



**Bachelors of Science in Computer Science and  
Information Technology (BSc. CSIT)**

**Course: Design and Analysis of Algorithms (CSC-314)**  
**Year/Semester: 3/5**

**A Lab report on:  
Iteration Algorithms (I)**

Submitted by:  
Name: Nishant Khadka  
Roll: 1017

Submitted to:  
Sujan Shrestha  
Department of Computer Science

Date: 12-Aug-2023

# GCD

## Theory

The Euclidean algorithm is a way to find the greatest common divisor of two positive integers. GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorize both numbers and multiply common prime factors.

### Basic Euclidean Algorithm for GCD:

The algorithm is based on the below facts.

- If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
- Now instead of subtraction, if we divide the smaller number, the algorithm stops when we find the remainder 0.

The extended Euclidean algorithm is particularly useful when  $a$  and  $b$  are coprime (or gcd is 1). Since  $x$  is the modular multiplicative inverse of " $a$  modulo  $b$ ", and  $y$  is the modular multiplicative inverse of " $b$  modulo  $a$ ". In particular, the computation of the modular multiplicative inverse is an essential step in RSA public-key encryption method.

## Algorithm

- Step 1:** Start
- Step 2:** Set first = 0, second = 1
- Step 3:** Read term of Fibonacci number say  $t$  be  $n$
- Step 4:** Set  $i=3$
- Step 5:** While( $i \leq n$ )
- Step 6:** Set temp = first + second
- Step 7:** Set first = second
- Step 8:** Set second = temp
- Step 9:** Increment  $i$  by 1 as  $i++$

**Step 10:** Print temp as required Fibonacci number

**Step 11:** Stop

## Pseudocode

```
GCD(A,B):  
{  
    If (A = 0)  
        Print "B as GCD"  
    elseif (B = 0)  
        Print "A as GCD";  
    else  
    {  
        While(B!=0)  
        {  
            R = A % B  
            A = B  
            B = R  
        }  
        Print " A as GCD"  
    }  
}
```

## Source Code

```
#include <stdio.h>

int gcd(int a, int b) {
    int r;

    if (a == 0)
        return b;
    else if (b == 0)
        return a;
    else {
        while (b != 0) {
            r = a % b;
            a = b;
            b = r;
        }
        return a;
    }
}

int main() {
    int num1, num2;

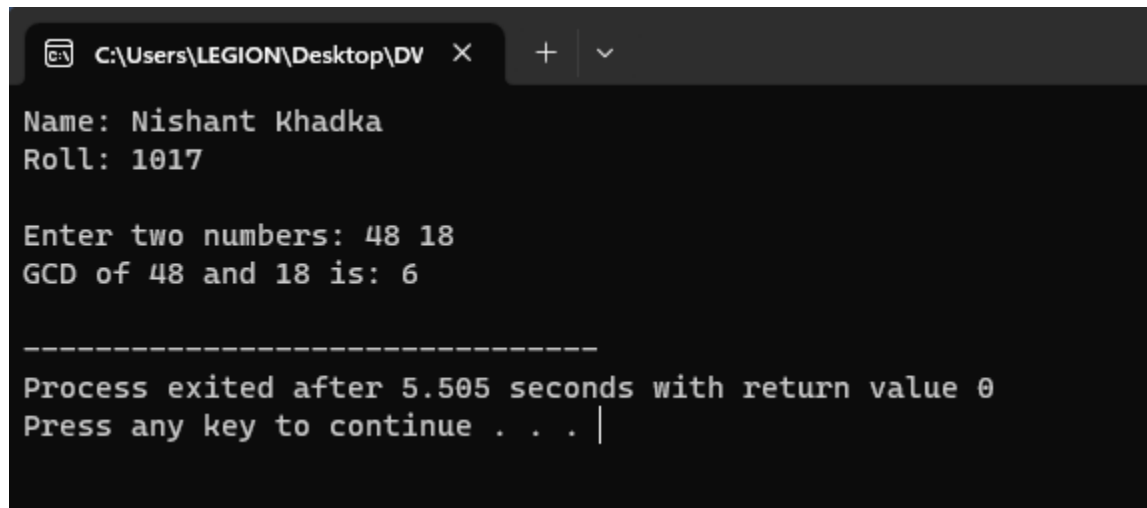
    printf("Name: Nishant Khadka\nRoll: 1017\n\n");
    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);

    int result = gcd(num1, num2);

    printf("GCD of %d and %d is: %d\n", num1, num2, result);

    return 0;
}
```

## Output

A screenshot of a Windows command prompt window. The title bar shows the file path 'C:\Users\LEGION\Desktop\DV' and standard window controls. The command prompt displays the following text: 'Name: Nishant Khadka', 'Roll: 1017', a blank line, 'Enter two numbers: 48 18', 'GCD of 48 and 18 is: 6', a blank line, a separator line of dashes, 'Process exited after 5.505 seconds with return value 0', and 'Press any key to continue . . . |' with a cursor at the end.

```
C:\Users\LEGION\Desktop\DV > Name: Nishant Khadka
Roll: 1017

Enter two numbers: 48 18
GCD of 48 and 18 is: 6

-----
Process exited after 5.505 seconds with return value 0
Press any key to continue . . . |
```

## Discussion

The provided GCD calculation program implements the Euclidean algorithm to efficiently compute the Greatest Common Divisor (GCD) of two given numbers. The program begins with a user-friendly introduction and input prompt displaying the author's name and roll number. Users are then prompted to enter two integers for which the GCD needs to be determined. After inputting the numbers, the program calculates and displays the GCD using the implemented algorithm. The output showcases the result in a clear and comprehensible format. The efficiency of the algorithm is highlighted by its time complexity,  $O(\log \min(A, B))$ , ensuring rapid computation even for large numbers. This GCD calculation program provides a practical tool for finding the GCD of two numbers while offering a seamless user experience and optimal computational performance.

# Nth FIBONACCI NUMBER

## Theory

The numbers in the Fibonacci Sequence don't equate to a specific formula, however, the numbers tend to have certain relationships with each other. Each number is equal to the sum of the preceding two numbers. For example, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377.

- The Fibonacci sequence is a set of steadily increasing numbers where each number is equal to the sum of the preceding two numbers.
- The golden ratio of 1.618 is derived from the Fibonacci sequence.
- Many things in nature have dimensional properties that adhere to the golden ratio of 1.618.
- The Fibonacci sequence can be applied to finance by using four techniques including retracements, arcs, fans, and time zones.

Fibonacci Sequence Rule:

- $x_n = x_{n-1} + x_{n-2}$
- *where:*
- *$x_n$  is term number "n"*
- *$x_{n-1}$  is the previous term (n-1)*
- *$x_{n-2}$  is the term before that (n-2)*

## Algorithm

- Step 1:** Start
- Step 2:** Set first = 0, second = 1
- Step 3:** Read term of Fibonacci number say t be n
- Step 4:** Set i=3
- Step 5:** While(i<=n)
  - a. Set temp = first + second
  - b. Set first = second
  - c. Set second = temp
  - d. Increment i by 1 as i++
- Step 6:** Print temp as required Fibonacci number
- Step 7:** Stop

## Pseudocode

```
fibonacci(n):  
{  
    Set first = 0  
    Set second = 1  
    Read n (the term of the Fibonacci number)  
    Set i = 3  
    While (i <= n)  
    {  
        Set temp = first + second  
        Set first = second  
        Set second = temp  
        Increment i by 1 (i++)  
    }  
    Print temp as the required Fibonacci number
```

```
}
```

## Source Code

```
#include <stdio.h>

int fibonacci(int n) {
    int first = 0, second = 1, temp;

    if (n == 1)
        return first;
    else if (n == 2)
        return second;

    for (int i = 3; i <= n; i++) {
        temp = first + second;
        first = second;
        second = temp;
    }

    return second;
}

int main() {
    printf("Name: Nishant Khadka\nRoll: 1017\n\n");

    int term;
    printf("Enter the term of the Fibonacci number: ");
    scanf("%d", &term);

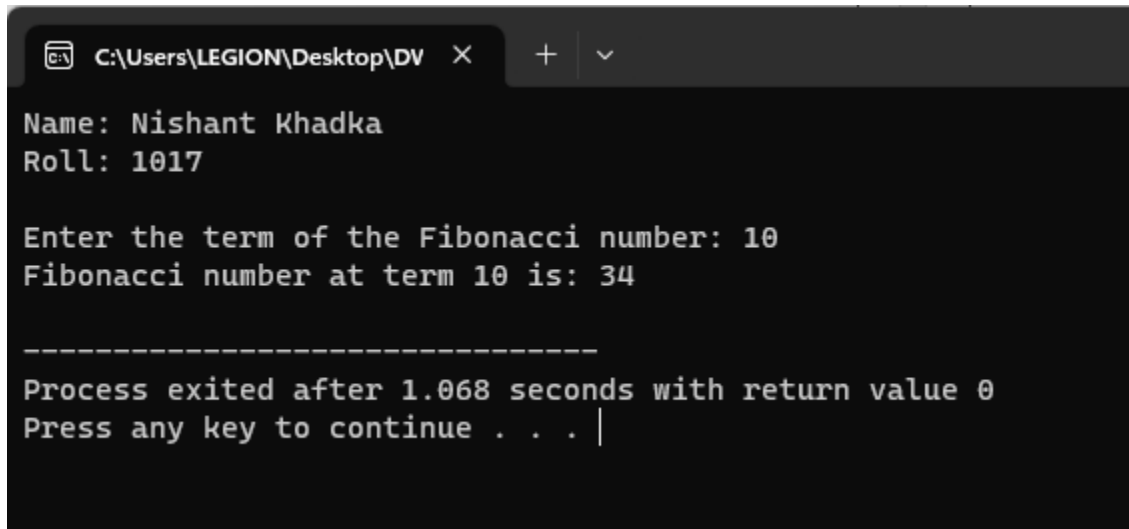
    int result = fibonacci(term);

    printf("Fibonacci number at term %d is: %d\n", term, result);

    return 0;
}
```



## Output

A screenshot of a terminal window with a dark background. The window title bar shows the file path 'C:\Users\LEGION\Desktop\DV' and standard window controls. The output text is as follows:

```
Name: Nishant Khadka
Roll: 1017

Enter the term of the Fibonacci number: 10
Fibonacci number at term 10 is: 34

-----
Process exited after 1.068 seconds with return value 0
Press any key to continue . . . |
```

## Discussion

The provided C program efficiently calculates the Fibonacci number at a given term using an iterative approach. The program begins with an introduction displaying the author's name and roll number. Users are prompted to input the desired term of the Fibonacci number. The fibonacci function uses a loop to iteratively compute the Fibonacci number up to the specified term, ensuring a linear time complexity of  $O(n)$ . The output illustrates the Fibonacci number at the chosen term. This program showcases an effective method for generating Fibonacci numbers, making it a versatile tool for various applications.

# LINEAR SEARCH

## Theory

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

Two popular search methods are Linear Search and Binary Search. So, here we will discuss the popular searching technique, i.e., Linear Search Algorithm.

Linear search is also called as **sequential search algorithm**. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is  **$O(n)$** .

The steps used in the implementation of Linear Search are listed as follows -

- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop**, compare the search element with the current array element, and -
  - If the element matches, then return the index of the corresponding array element.
  - If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return **-1**.

## Algorithm

- Step 1:** Start
- Step 2:** Set  $i = 1$
- Step 3:** Read array A and element x
- Step 4:** While  $i \leq$  last index of A
- Step 5:** If  $A[i]$  equals x
- Step 6:** Print "Element x found at index i"
- Step 7:** Stop
- Step 8:** Increment i by 1
- Step 9:** Print "Element x not found in array A"
- Step 10:** Stop

## Pseudocode

LinearSearch:

```
for i = 1 to last index of A
    if A[i] equals element x
        return i
return -1
```

## Source Code

```
#include <stdio.h>

int linearSearch(int arr[], int size, int x) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == x) {
            return i;
        }
    }
    return -1;
}

int main() {
    printf("Name: Nishant Khadka\nRoll: 1017\n\n");

    int size, x;

    printf("Enter the size of the array: ");
    scanf("%d", &size);

    int arr[size];
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

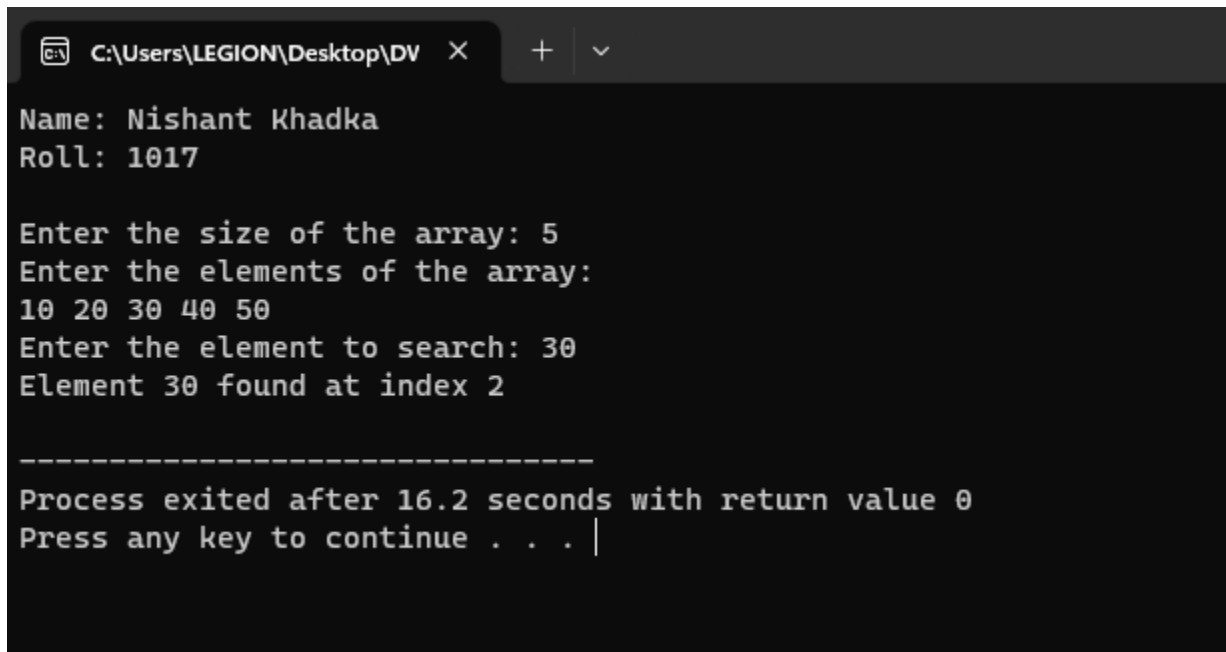
    printf("Enter the element to search: ");
    scanf("%d", &x);

    int index = linearSearch(arr, size, x);

    if (index != -1) {
        printf("Element %d found at index %d\n", x, index);
    } else {
        printf("Element %d not found in the array\n", x);
    }

    return 0;
}
```

## Output

A screenshot of a Windows terminal window showing the execution of a C program. The window title bar indicates the file path 'C:\Users\LEGION\Desktop\DV'. The program output includes the author's name 'Nishant Khadka' and roll number '1017'. It prompts the user to enter the size of the array (5) and the elements (10 20 30 40 50). The user enters '30' as the element to search, and the program outputs 'Element 30 found at index 2'. A separator line of dashes follows, and the program concludes with 'Process exited after 16.2 seconds with return value 0' and a prompt to press any key to continue.

```
C:\Users\LEGION\Desktop\DV X + v
Name: Nishant Khadka
Roll: 1017

Enter the size of the array: 5
Enter the elements of the array:
10 20 30 40 50
Enter the element to search: 30
Element 30 found at index 2

-----
Process exited after 16.2 seconds with return value 0
Press any key to continue . . . |
```

## Discussion

The presented C program demonstrates the Linear Search algorithm for locating an element within an array. It starts with an introduction displaying the author's name and roll number. Users are prompted to input the size of the array and its elements, followed by the element to search. The `linearSearch` function iterates through the array, comparing each element to the target element `x`. If a match is found, it returns the index of the element; otherwise, it returns `-1`. The output showcases whether the target element was found and its corresponding index. Linear Search exhibits a linear time complexity of  $O(n)$ , making it suitable for small to medium-sized arrays where elements are not ordered.

# BUBBLE SORT

## Theory

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is  $O(n^2)$ , where  $n$  is a number of items.

Bubble sort is majorly used where -

- complexity does not matter
- simple and shortcode is preferred

In the bubble sort algorithm, comparisons are made even when the array is already sorted. Because of that, the execution time increases.

To solve it, we can use an extra variable **swapped**. It is set to **true** if swapping requires; otherwise, it is set to **false**.

It will be helpful, as suppose after an iteration, if there is no swapping required, the value of variable **swapped** will be **false**. It means that the elements are already sorted, and no further iterations are required.

This method will reduce the execution time and also optimizes the bubble sort.

## Algorithm

### 1. First Iteration (Compare and Swap)

- a. Starting from the first index, compare the first and the second elements.
- b. If the first element is greater than the second element, they are swapped.
- c. Now, compare the second and the third elements. Swap them if they are not in order.
- d. The above process goes on until the last element.

### 2. Remaining Iteration

The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.

## Pseudocode

Bubble\_Sort(A):

  for  $i \leftarrow 1$  to  $n-1$  do

    for  $j \leftarrow 1$  to  $n-i$  do

      if  $A[j] > A[j+1]$  then

        swap( $A[j]$ ,  $A[j+1]$ )

        temp  $\leftarrow A[j]$

$A[j] \leftarrow A[j+1]$

$A[j+1] \leftarrow temp$

## Source Code

```
#include <stdio.h>

void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

int main() {
    printf("Name: Nishant Khadka\nRoll: 1017\n\n");

    int size;
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    int arr[size];
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    bubbleSort(arr, size);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```



## Output

```
C:\Users\LEGION\Desktop\DV X + v
Name: Nishant Khadka
Roll: 1017

Enter the size of the array: 5
Enter the elements of the array:
45 12 78 34 23
Sorted array: 12 23 34 45 78

-----
Process exited after 14.63 seconds with return value 0
Press any key to continue . . . |
```

## Discussion

The provided C program showcases the Bubble Sort algorithm for sorting an array of integers in ascending order. The program starts with an introduction displaying the author's name and roll number. Users are prompted to input the size and elements of the array. The **bubbleSort** function iterates through the array multiple times, comparing adjacent elements and swapping them if they are out of order. The output illustrates the sorted array. Bubble Sort has a time complexity of  $O(n^2)$ , making it inefficient for large datasets. Despite its simplicity, the algorithm serves as a foundation for understanding more complex sorting algorithms.

# INSERTION SORT

## Theory

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is  $O(n^2)$ , where  $n$  is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

## Algorithm

- Step 1:** Start
- Step 2:** insertionSort(array)
- Step 3:** mark first element as sorted
- Step 4:** for each unsorted element X
  - a. 'extract' the element X
- Step 5:** for  $j \leftarrow \text{lastSortedIndex}$  down to 0
- Step 6:** if current element  $j > X$ 
  - a. move sorted element to the right by 1
- Step 7:** break loop and insert X here
- Step 8:** end insertionSort

## Pseudocode

```
# Input: Array T
# Output: Sorted array T
Algorithm: Insertion_Sort( $T[1, \dots, n]$ )
for  $i \leftarrow 2$  to  $n$  do
     $x \leftarrow T[i]$ ;
     $j \leftarrow i - 1$ ;
    while  $x < T[j]$  and  $j > 0$  do
         $T[j+1] \leftarrow T[j]$ ;
         $j \leftarrow j - 1$ ;
     $T[j+1] \leftarrow x$ ;
```

## Source Code

```
#include <stdio.h>

void insertionSort(int arr[], int size) {
    for (int i = 1; i < size; i++) {
        int x = arr[i];
        int j = i - 1;

        while (j >= 0 && x < arr[j]) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = x;
    }
}

int main() {
    printf("Name: Nishant Khadka\nRoll: 1017\n\n");

    int size;
    printf("Enter the size of the array: ");
    scanf("%d", &size);

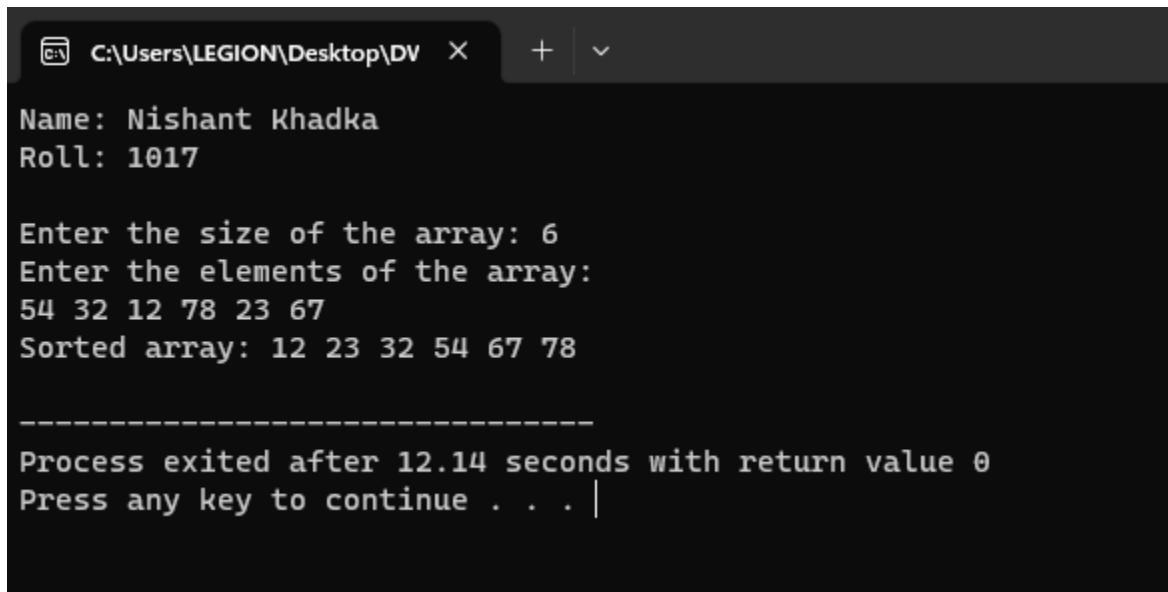
    int arr[size];
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    insertionSort(arr, size);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

## Output

A screenshot of a terminal window showing the output of a C program. The window has a title bar with a file icon, the path 'C:\Users\LEGION\Desktop\DV', and window control buttons. The output text is as follows:

```
Name: Nishant Khadka
Roll: 1017

Enter the size of the array: 6
Enter the elements of the array:
54 32 12 78 23 67
Sorted array: 12 23 32 54 67 78

-----
Process exited after 12.14 seconds with return value 0
Press any key to continue . . . |
```

## Discussion

The provided C program demonstrates the Insertion Sort algorithm for sorting an array of integers in ascending order. The program begins with an introduction displaying the author's name and roll number. Users are prompted to input the size and elements of the array. The insertionSort function iterates through the array, comparing the current element with the previous elements and shifting them to the right until a suitable position is found. This process results in a sorted array. Insertion Sort has a time complexity of  $O(n^2)$ , making it efficient for small datasets and partially sorted arrays. While not as fast as more advanced sorting algorithms, Insertion Sort's simplicity makes it a valuable learning tool.

# SELECTION SORT

## Theory

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is  $O(n^2)$ , where  $n$  is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

## Algorithm

- Step 1:** Start
- Step 2:** Make a variable (say min\_index) and initialize it to the location 0 of the array.
- Step 3:** Traverse the whole array to find the smallest element in the array.
- Step 4:** While traversing the array, if we find an element that is smaller than
- Step 5:** min\_index then swap both these elements.
- Step 6:** After which, increase the min\_index by 1 so that it points to the next element
- Step 7:** of the array.
- Step 8:** Repeat the above process until the whole array is sorted.
- Step 9:** Stop

## Pseudocode

Algorithm: Selection\_Sort(A)

```
for i ← 1 to n-1 do
    minj ← i;
    minx ← A[i];
    for j ← i + 1 to n do
        if A[j] < minx then
            minj ← j ;
            minx ← A[j];
    A[minj] ← A[i];
    A[i] ← minx;
```

## Source Code

```
#include <stdio.h>

void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minj = i;
        int minx = arr[i];

        for (int j = i + 1; j < size; j++) {
            if (arr[j] < minx) {
                minj = j;
                minx = arr[j];
            }
        }

        arr[minj] = arr[i];
        arr[i] = minx;
    }
}

int main() {
    printf("Name: Nishant Khadka\nRoll: 1017\n\n");

    int size;
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    int arr[size];
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

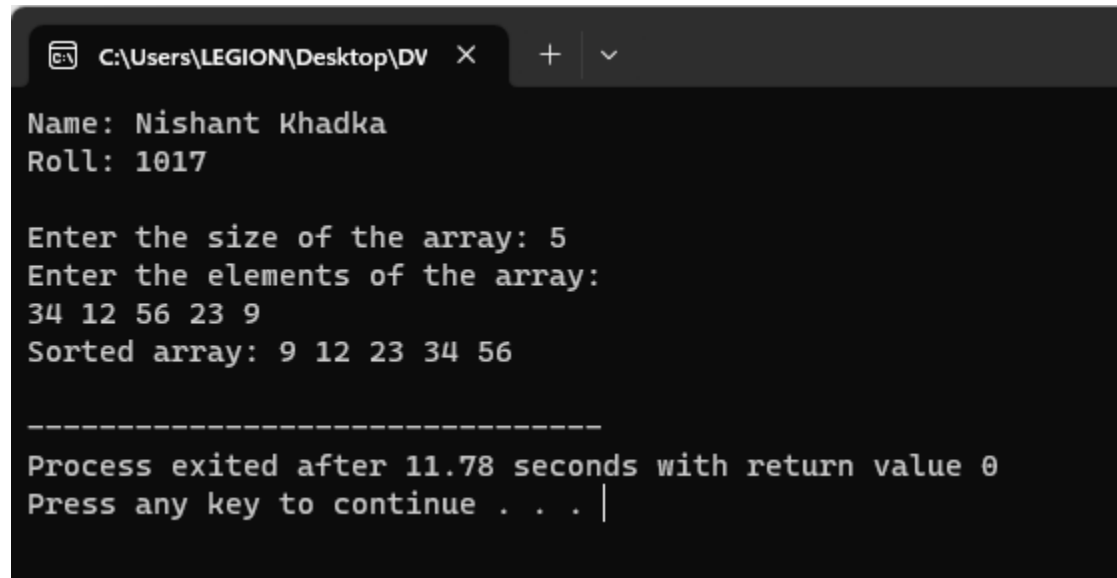
    selectionSort(arr, size);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```



## Output



```
C:\Users\LEGION\Desktop\DV X + v
Name: Nishant Khadka
Roll: 1017

Enter the size of the array: 5
Enter the elements of the array:
34 12 56 23 9
Sorted array: 9 12 23 34 56

-----
Process exited after 11.78 seconds with return value 0
Press any key to continue . . . |
```

## Discussion

The provided C program showcases the Selection Sort algorithm for sorting an array of integers in ascending order. The program starts with an introduction displaying the author's name and roll number. Users are prompted to input the size and elements of the array. The `selectionSort` function iterates through the array, finding the smallest element in the unsorted portion and swapping it with the element at the current position. This process continues until the entire array is sorted. Selection Sort has a time complexity of  $O(n^2)$ , making it simple but inefficient for large datasets. While not as fast as more advanced sorting algorithms, Selection Sort serves as a basic sorting technique that can help build an understanding of sorting algorithms.

## **CONCLUSION**

In this way, various different algorithms such as GCD, nth Fibonacci number, linear search, bubble sort, insertion sort and selection sort were studied and implemented in C programming language.