III B.TECH I SEM	Employability Skills-1	L	T	P	C
		2	0	0	0

Course Objectives:

- The development and implementation of advanced algorithms, as well as the skills required for programming competitions.

Course Outcomes:

By the end of the course students will be able to

1. select appropriate algorithms for a given problem
2. integrate multiple algorithms for solving a complex problem
3. Design new algorithms, and implement them in Python or Java.
4. Learn skills required for participation in programming contests, which include evaluation of problem difficulty, solving problems in teams, and work under time pressure.

UNIT I

Basics of Array, String, Greedy and Bit Manipulation:

Sum of array elements, Reverse of an array, Maximum and minimum element of an array, counting frequencies of array elements, prefix sum, Kadane algorithm, Activity Selection problem, Sliding Window, Bit manipulation.

UNIT II

Number Theory and Combinatorics:

Prime Number, Sieve of Eratosthenes, Find all divisors of a natural number, Least prime factor of numbers upto N, All prime factors of a number, Prime factorization using Sieve, Sum of all factors of a number, GCD and LCM of two numbers, Euclidean algorithms.

UNIT III

Searching, Sorting, Basic Data Structures:

Linear Search, Binary Search, Merge Sort, Quick Sort, Stack, Queue, Deque, Priority Queue.

UNIT IV

Trees and Graphs:

Tree Traversals, BFS, DFS, Dijkstra's Shortest Path algorithm, Bell-man Ford Algorithm, Floyd's algorithm

UNIT V

Recursion and Dynamic Programming:

Recursion and problems, Backtracking, N-Queens Problem, Dynamic Programming, Minimum-Edit Distance Problem.

Text Books :

1. Fundamentals of computer algorithms E. Horowitz S. Sahni, University Press

Reference Books:

1. Competitive Programming in Python: 128 Algorithms to Develop your Coding Skills by by Christoph Dürr, Jill-Jênn Vie
2. Guide to Competitive Programming: Learning and Improving Algorithms Through Contests (Undergraduate Topics in Computer Science) by Antti Laaksonen.

Arrays:

- An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an *index* (also known as the *subscript*).
- In C, arrays are declared using the following **syntax: datatype name[size];**

Example: int marks[10];

1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

- ***ONE – DIMENSIONAL ARRAYS:-***
  - A list of items can be given one variable index is called single subscriptedvariable or a one-dimensional array.
  - The subscript value starts from 0. If we want 5 elements the declarationwill be int number[5];
  - The elements will be number[0], number[1], number[2], number[3], number[4]There will not be number[5]

Declaration of One - Dimensional Arrays: ***Type variable – name [sizes];***

Type – data type of all elements Ex: int, float etc.,

Variable – name – is an identifier

Size – is the maximum no of elements that can be stored

Ex: - float avg[50]

This array is of type float. Its name is avg. and it can contains 50 elements only.The range starting from 0 – 49 elements.

Initialization of Arrays:-

Initialization of elements of arrays can be done in same way as ordinaryvariables are done when they are declared. **Type array name[size] = {List of Value};**

Ex:- int number[3]={0,0,0};

If the number of values in the list is less than number of elements then only that elements will be initialized. The remaining elements will be set tozero automatically.

- ***TWO – DIMENSIONAL ARRAYS:-***

To store tables we need two dimensional arrays. Each table consists of rowsand columns.

Two dimensional arrays are declare as **type array name [row-size][col-size];**

INITIALIZING TWO DIMENSIONAL ARRAYS:-

They can be initialized by following their declaration with a list of initialvalues enclosed in braces.

Ex: - int table[2][3] = {0,0,0,1,1,1};

Initializes the elements of first row to zero and second row to one. The initialization is done by row by row. The above statement can be written as **int table[2][3] = {{0,0,0},{1,1,1}};**

When all elements are to be initialized to zero, following short-cut methodmay be used.

int m[3][5] = {{0},{0},{0}};

- ***Multidimensional Arrays:-***

C allows for arrays of two or more dimensions. A two-dimensional(2D) array is an array of arrays. A three-dimensional (3D) array is an array of arrays of arrays.

In C programming an array can have two, three, or even ten or more dimensions. The maximum dimensions a C program can havedepends on which compiler is being used.

Declare a Multidimensional Array in C:-

A multidimensional array is declared using the following syntax:

type array_name[d1][d2][d3][d4]…….[dn];

Where each **d** is a dimension, and **dn** is the size of final dimension.

Examples:

1. **int table[5][5][20];**
   - **int** designates the array type integer.
   - **table** is the name of our 3D array.
   - Our array can hold 500 integer-type elements. This number isreached by multiplying the value of each dimension. In this case: **5x5x20=500**.
2. **float arr[5][6][5][6][5];**

a. Array **arr** is a five-dimensional array.

b. It can hold 4500 floating-point elements (**5x6x5x6x5=4500**).

Limitations:

- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from theirpositions.

Strings:-

A String is an array of characters. Any group of characters (except doublequote sign) defined between double quotes is a constant string.

Ex: “C is a great programming language”. If we want to include double quotes.

Ex: “\”C is great \” is norm of programmers “.

Declaring and initializing strings:-

A string variable is any valid C variable name and is always declared asan array.

char string name [size];

Size determines number of characters in the string name. When the compiler assigns a character string to a character array, it automaticallysupplies a null character (‘\0’) at end of String.

Therefore, size should be equal to maximum number of character inString plus one.

There are two ways to declare a string in c language.

1. By char array: char city*10+= ,‘N’,’E’,’W’,’ ‘,’Y’,’O’,’R’,’K’,’\0’-;
2. By string literal: char city*10+= “NEW YORK’;

C also permits us to initializing a String without specifying size.

Ex:- char Strings*+= ,‘G’,’O’,’O’,’D’,’\0’-;

String Input / Output Functions:-

- C provides two basic ways to read and write strings
- First we can read and write strings with the formatted input/output functions,scanf/fscanf and prinf/fprinf.
- Second we can use a special set of strin only functions ,getstring(gets/fgets) and put string(puts/fputs).

Formatted string Input/Output:

- Formatted String Input:scanf/fscanf:
- **int fscanf(FILE **stream*, const char **format*, ...);**
- ***int scanf(const char *format, ...);***
- The ..scanf functions provide a means to input formatted informationfrom a stream.
- **fscanf** reads formatted input from a stream
- **scanf** reads formatted input stdin

These functions take input in a manner that is specified by the format argument and store each input field into the following argumentsin a left to right fashion.

String Input/Output

In addition to the Formatted string functions,C has two sets of string functions that read and write strings without reformatting any data.These functions convert text file lines to strings and strings to textfile lines

gets(): *Declaration:* ***char *gets(char *str);***

Reads a line from **stdin** and stores it into the string pointed to by *str*. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first. The newline character is not copied to the string. A null character is appended to the end of the string.

Puts(): *Declaration:* ***int puts(const char *str);***

Writes a string to **stdout** up to but not including the null character. A newline character is appended to the output. On success a nonnegative value is returned. On error **EOF** is returned.

STRING HANDLING/MANIPULATION FUNCTIONS:-

- strcat( ) Concatenates two Strings
- strcmp( ) Compares two Strings
- strcpy( ) Copies one String Over another
- strlen( ) Finds length of String

✓ ***strcat() function:*** This function adds two strings together.

Syntax: char *strcat(const char *string1, char *string2);

Ex: strcat(string1,string2);

string1 = VERY

string2 = FOOLISH

strcat(string1,string2);

string1=VERYFOOLISH

string2 = FOOLISH

Strncat: Append n characters from string2 to string l.
char *strncat(const char *string1, char *string2, size_t n);

✓ ***strcmp() function :***

This function compares two strings identified by arguments and has a value 0 if they are equal. If they are not, it has the numeric difference betweenthe first non-matching characters in the Strings.

Syntax: int strcmp (const char *string1,const char *string2);
strcmp(string1,string2);

Ex:- strcmp(name1,name2);
strcmp(name1,"John");
strcmp("ROM","Ram");

Strncmp: Compare first n characters of two strings.
int strncmp(const char *string1, char *string2, size_t n);

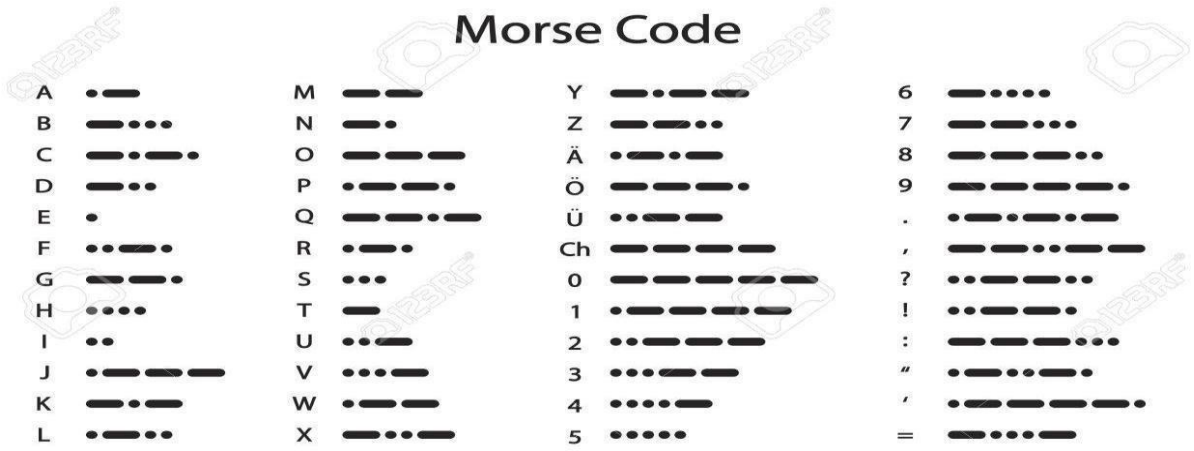
✓ ***strcpy() function :*** It works almost as a string assignment operators.
It takes the form Syntax: char *strcpy(const char *string1,const char *string2);
strcpy(string1,string2);
string2 can be array or a constant.

Strncpy: Copy first n characters of string2 to stringl .
char *strncpy(const char *string1,const char *string2, size_t n);

✓ ***strlen() function : Counts and returns the number of characters in a string.***

Syntax: int strlen(const char *string);
n= strlen(string);
n integer variable which receives the value of length of string.

A Programming Example – Morse Code:- Morse code is a method of transmitting text information as a series of on-off tones, lights, or clicks that can be directly understoodby a skilled listener or observer without special equipment. It is named for Samuel F. B. Morse, an inventor of the telegraph.



Sum of Array of Elements

Given an array of integers, find the sum of its elements.

Examples:
Input: *arr[] = {1, 2, 3}*
Output : *6*
Explanation: *1 + 2 + 3 = 6*
Code:

```
#include <stdio.h>
int main()
{
  int sum = 0, arr[] = {1, 2, 3, 4, 5};        //Initialize array
  int length = sizeof(arr)/sizeof(arr[0]);  //Calculate length of array arr
    for (int i = 0; i < length; i++)          //Loop through the array to calculate sum of elements
      {
        sum = sum + arr[i];
       }
  printf("Sum of all the elements of an array: %d", sum);
  return 0;
}
```

Reverse an Array

Reversing an array means changing the order of elements so that the first element becomes the last element and the second element becomes the second last element and so on. The task is to reverse the elements of the given array.

For example, Suppose given an array:

1	2	3	4	5	6

6	5	4	3	2	1

Possible Approaches:

1. Declaring another array
2. Iteration and Swapping
3. Using pointers

1. Declaring another array

We can declare another array, iterate the original array from backward and send them into the new array from the beginning.

Code:

```
#include<stdio.h>
int main()
{
    int n, arr[n], i;
   printf("Enter the size of the array: ");
    scanf("%d", &n);
    printf("Enter the elements: ");
    for(i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    int rev[n], j = 0;
    for(i = n-1; i >= 0; i--)
    {
        rev[j] = arr[i];
        j++;
    }
    printf("The Reversed array: ");
    for(i = 0; i < n; i++)
   {
        printf("%d ", rev[i]);
    }
}
```

Output:

```
Enter the size of the array: 5
Enter the elements: 1 2 3 4 5
The Reversed array: 5 4 3 2 1
```

Here is an example:

If the array is:

1	2	3	4	5	6

We create another array rev with the same size = 6:

First iteration: i = n - 1 = 5, i >= 0, j = 0

rev[0] = arr[5]

In the same way:

Second iteration: rev[1] = arr[4]

Third iteration: rev[2] = arr[3]

Fourth iteration: rev[3] = arr[2]

Fifth iteration: rev[4] = arr[1]

Sixth iteration: rev[5] = arr[0]

Hence, the resultant reversed array:

6	5	4	3	2	1

2. Iteration and swapping

Iterating half of the array:

We iterate the array till size/2, and for an element in index i, we swap it with the element at index (size - i - 1).

Here is an example:

If the array is:

1	2	3		4	5	6

size = 6, size/2 = 3

First iteration: i = 0, i < 3

We swap arr[i] with arr[n - i - 1]

arr[0] <-> arr[5]

6	2	3	4	5	1

Second iteration: i = 1, i < 3

We swap arr[i] with arr[n - i - 1]

arr[1] <-> arr[4]

6	5	3	4	2	1

Third iteration: i = 2, i < 3

We swap arr[i] with arr[n - i - 1]

arr[2] <-> arr[3]

Termination when i = 3 as i == n/2. **The resultant reversed array:**

6	5	4	3	2	1

Code:

```
#include<stdio.h>
int main()
{
    int i, n, temp;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for(i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    int end = n - 1;
    for(i = 0; i < n/2; i++)
    {
        temp = arr[i];
        arr[i] = arr[end];
        arr[end] = temp;
        end--;
    }
    printf("The reversed array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
}
```

Output:

```
Enter the size of the array: 6
Enter the elements: 1 2 3 4 5 6
The reversed array: 6 5 4 3 2 1
```

Iterating from two ends of the array:

We can keep iterating and swapping elements from both sides of the array till the middle element.

Code:

```
#include<stdio.h>
int main()
{
    int n, i;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for(i = 0; i < n; i++)
            scanf("%d", &arr[i]);
    int l = 0;
    int h = n - 1;
    int temp;
    while(l < h)
    {
        temp = arr[l];
        arr[l] = arr[h];
        arr[h]= temp;
        l = l + 1;
        h = h - 1;
    }
    printf("The Reversed array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
}
```

Output:

```
Enter the size of the array: 6
Enter the elements: 1 2 3 4 5 6
The Reversed array: 6 5 4 3 2 1
```

Here is an example:

If the array is:

1	2	3	4	5	6

size = 6

l = 0, h = size - 1 = 5

First iteration: l = 0, h = 5

We swap arr[l] with arr[h]

arr[0] <-> arr[5]

6	2	3	4	5	1

Second iteration: l = 1, h = 4

We swap arr[l] with arr[h]

arr[1] <-> arr[4]

6	5	3	4	2	1

Third iteration: l = 2, h = 3

We swap arr[l] with arr[h]

arr[2] <-> arr[3]

Termination when l = h = 3 as l !< h

The resultant reversed array:

6	5	4	3	2	1

Recursive Way:

```
#include<stdio.h>
void reverse(int l, int h, int arr[]);
void display(int arr[], int n);
int main()
{
    int n, i;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for(i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
```

```
    }
    int l = 0, h = n - 1;
    reverse(l, h, arr);
    printf("The reversed array: ");
    display(arr, n);
    return 0;
}
void reverse(int l, int h, int arr[])
{
    if(l >= h)
    {
        return;
    }
    int temp;
    temp = arr[l];
    arr[l] = arr[h];
    arr[h] = temp;
    reverse(l + 1, h - 1, arr);  //recursive call
}
void display(int arr[], int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
}
```

Output:

```
Enter the size of the array: 6
Enter the elements: 1 2 3 4 5 6
The reversed array: 6 5 4 3 2 1
```

3. Using pointers

A pointer is like a special variable that can hold the addresses of other variables and pointers. We can swap the array elements using pointers rather than traditional array indexes.

Important points about pointers:

1. A pointer has to be declared as **data type* name**. The pointer will be able to store the address of variables of only the specified data type.
2. Suppose we declared a pointer ***ptr**; if we want the pointer to refer to variable **a**, we need to declare **ptr = &a**
3. ***ptr** gives the value of the variable ptr is holding, **&ptr** gives the address of the pointer, **ptr** gives the address of the variable it is holding.
4. When we say **ptr + 1** or ptr - 1, the **pointer arithmetic** happens concerning the data type: If the data type is int, the size of int = 4. Hence, **ptr + 1 -> ptr +1(4)** will be the next address ptr will hold.

Code:

```
#include <stdio.h>
int main()
{
   int *ptr1, *ptr2;
   int n, i, temp;
   printf("Enter the size of the array: ");
   scanf("%d", &n);
   int arr[n];
   printf("Enter the elements into the array: ");
   for(i = 0; i < n; i++)
         scanf("%d", &arr[i]);
   ptr1 = &arr[0];
   ptr2 = &(arr[n - 1]);
        while(ptr1 < ptr2)
   {
      temp = *ptr1;
      *ptr1 = *ptr2;
      *ptr2 = temp;
      ptr1 = ptr1 + 1;
      ptr2 = ptr2 - 1;
   }
   printf("The reversed array: ");
   for(i = 0; i < n; i++)
   {
      printf("%d ", arr[i]);  } }
```

Output:

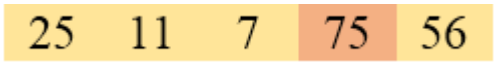
```
Enter the size of the array: 6
Enter the elements into the array: 1 2 3 4 5 6
The reversed array: 6 5 4 3 2 1
```

Understanding:

- We declared two integer pointers, ptr1 and ptr2.
- ptr1 points to the first element of the array, and ptr2 points to the last element of the array.
- When we say ptr1 and ptr2 refer to the addresses of the first and last elements of the array.
- An array stores elements in contiguous memory locations. The memory allocations for an array from the first element to the last element will be in increasing order; hence the condition ptr1 < ptr2 is used in the while loop.
- Now, using *ptr1 and *ptr2, we swap the values at ptr1 and ptr2 with a temporary variable-temp.
- We increment ptr1 to its succeeding element and decrement ptr2 to its preceding element using simple pointer arithmetic.
- The swapping continues till both ptr1 and ptr2 reach the middle element or when ptr1 and ptr2 overtake the middle element of the array in their iterations.

Largest element of an array:

We need to find out the largest element present in the array and display it. This can be accomplished by looping through the array from start to end by comparing max with all the elements of an array. If any of element is greater than max, then store a value of the element in max. Initially, max will hold the value of the first element. At the end of the loop, max represents the largest element in the array.



In the above array, initially, max will hold the value 25. In the 1st iteration, max will be compared with 11, since 11 is less than max. Max will retain its value. In the next iteration, it will be compared to 7, 7 is also less than max, no change will be made to the max. Now, max will be compared to 75. 75 is greater than max so that max will hold the value of 75. Continue this process until the end of the array is reached. At the end of the loop, max will hold the largest element in the array.

ALGORITHM:

- **STEP 1:** START
- **STEP 2:** INITIALIZE arr[] = {25, 11, 7, 75, 56}
- **STEP 3:** length= sizeof(arr)/sizeof(arr[0])
- **STEP 4:** max = arr[0]
- **STEP 5:** SET i=0. REPEAT STEP 6 and STEP 7 i<length

- **STEP6:** if(arr[i]>max)
  max=arr[i]
- **STEP 7:** i=i+1.
- **STEP 8:** PRINT "Largest element in given array:" assigning max.
- **STEP 9:** RETURN 0
- **STEP 9:** END.

PROGRAM:

```
#include <stdio.h>
 int main()
{
      int arr[] = {25, 11, 7, 75, 56};          //Initialize array
      int length = sizeof(arr)/sizeof(arr[0]);       //Calculate length of array arr
      int max = arr[0];       //Initialize max with first element of array.
        for (int i = 0; i < length; i++)             //Loop through the array
        {
           if(arr[i] > max)              //Compare elements of array with max
           max = arr[i];
         }
   printf("Largest element present in given array: %d\n", max);
   return 0;
}
```

Output:

```
Largest element present in given array: 75
```

Minimum Element of Array

We need to find out the smallest element present in the array. This can be achieved by maintaining a variable min which initially will hold the value of the first element. Loop through the array by comparing the value of min with elements of the array. If any of the element's value is less than min, store the value of the element in min.

Consider above array. Initially, min will hold the value 25. In the 1st iteration, min will be compared with 11. Since 11 is less than 25. Min will hold the value 11. In a 2nd iteration, 11 will be compared with 7. Now, 7 is less than 11. So, min will take the value 7. Continue this process until the end of the array is reached. At last, min will hold the smallest value element in the array.

ALGORITHM:

- **STEP 1:** START
- **STEP 2:** INITIALIZE arr[] = {25, 11, 7, 75, 56}
- **STEP 3:** length= sizeof(arr)/sizeof(arr[0])
- **STEP 4:** min = arr[0]
- **STEP 5:** SET i=0. REPEAT STEP 6 and STEP 7 UNTIL i<length

- **STEP 6:** if(arr[i]<min) min=arr[i]
- **STEP 7:** i=i+1.
- **STEP 8:** PRINT "Smallest element present in given array:" by assigning min
- **STEP 9:** RETURN 0.
- **STEP 10:** END.

PROGRAM:

```
#include <stdio.h>
 int main()
{
   int arr[] = {25, 11, 7, 75, 56};        //Initialize array
    int length = sizeof(arr)/sizeof(arr[0]);      //Calculate length of array arr
     int min = arr[0];       //Initialize min with first element of array.
       for (int i = 0; i < length; i++) //Loop through the array
         {
           if(arr[i] < min)    //Compare elements of array with min
          min = arr[i];
         }
   printf("Smallest element present in given array: %d\n", min);
   return 0;
}
```

Output:

```
Smallest element present in given array: 7
```

Counting frequencies of array elements

We have an array of elements to count the occurrence of its each element. One of the approaches 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 as frequency and store it in another array fr.

```
1  2  8  3  2  2  2  5  1
```

In the given array, 1 has appeared two times so its frequency is 2 and 2 has appeared four times so have frequency 4 and so on.

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

Algorithm

- **STEP 1:** START
- **STEP 2:** INITIALIZE arr[] ={1, 2, 8, 3, 2, 2, 2, 5, 1 }.
- **STEP 3:** CREATE fr[] of arr[] length.
- **STEP 4:** SET visited = -1.
- **STEP 5:** REPEAT STEP 6 to STEP 9 for(i=0;i<arr.length;i++)
- **STEP 6:** SET count = 1
- **STEP 7:** REPEAT STEP 8 for(j=i+1;j<arr.length;j++)
- **STEP8:** if(arr[i]==arr[j])then
  count++
  fr[j] =visited
- **STEP9:** if(fr[i]!=visited)then
  fr[i]=count
- **STEP 10:** PRINT "----------"
- **STEP 11:** PRINT "Element | Frequency"
- **STEP 12:** PRINT "----------"
- **STEP 13:** REPEAT STEP 14 for(i=0;i<fr.length;i++)
- **STEP14:** if(fr[i]!=visited)then
  PRINT arr[i] and fr[i]
- **STEP 15:** PRINT "----------"
- **STEP 16:** END

CODE:

```
#include <stdio.h>
int main()
{
int arr[] = {1, 2, 8, 3, 2, 2, 2, 5, 1};   //Initialize array

   //Calculate length of array arr
   int length = sizeof(arr)/sizeof(arr[0]);

   //Array fr will store frequencies 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++;
            //To avoid counting same element again
            fr[j] = visited;
         }
      }
      if(fr[i] != visited)
```

```
            fr[i] = count;
        }
        printf("---------------------\n");       //Displays the frequency of each element present in array
        printf(" Element | Frequency\n");
        printf("---------------------\n");
        for(int i = 0; i < length; i++)   {
            if(fr[i] != visited)  {
                printf("    %d", arr[i]);
                printf("    |  ");
                printf("  %d\n", fr[i]);
            }
        }
        printf("---------------------\n");
        return 0;
}
```

Output:

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

Kadane's Algorithm

Kadane's algorithm is a dynamic programming approach used to solve the maximum sub-array problem, which involves finding the contiguous sub-array with the maximum sum in an array of numbers. The algorithm was proposed by Jay Kadane in 1984 and has a time complexity of O(n).

History of Kadane's algorithm:

Kadane's algorithm is named after its inventor, Jay Kadane, a computer science professor at Carnegie Mellon University. He first described the algorithm in a paper titled "Maximum Sum Sub-array Problem" published in the Journal of the Association for Computing Machinery in 1984.

The problem of finding the maximum sub-array has been studied by computer scientists since the 1970s. It is a well-known problem in the field of algorithm design and analysis and has applications in a wide range of areas, including signal processing, finance, and bioinformatics.

Prior to Kadane's algorithm, other algorithms had been proposed for solving the maximum sub-array problem, such as the brute-force approach that checks all possible sub-arrays and the divide-and-conquer algorithm. However, these algorithms have higher time complexities and are less efficient than Kadane's algorithm.

Kadane's algorithm is widely used in computer science and has become a classic example of dynamic programming. Its simplicity, efficiency, and elegance have made it a popular solution to the maximum sub-array problem and a valuable tool in algorithm design and analysis.

Working of Kadane's Algorithm:

The algorithm works by iterating over the array and keeping track of the maximum sum of the sub-array ending at each position. At each position i, we have two options: either add the element at position i to the current maximum sub-array or start a new sub-array at position i. The maximum of these two options is the maximum sub-array ending at position i.

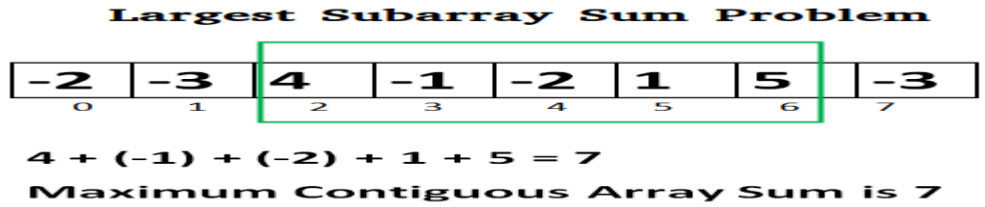
We maintain two variables, max_so_far and max_ending_here, to keep track of the maximum sum seen so far and the maximum sum ending at the current position, respectively. The algorithm starts by setting both variables to the first element of the array. Then, we iterate over the array from the second element to the end.

At each position i, we update max_ending_here by taking the maximum of the current element and the current element added to the previous maximum sub-array. We then update max_so_far to be the maximum of max_so_far and max_ending_here.

The algorithm returns max_so_far, which is the maximum sum of any sub-array in the array.

Here's the step-by-step process of Kadane's Algorithm:

Given an array **arr[]** of size **N**. The task is to find the sum of the contiguous sub-array within a **arr[]** with the largest sum.



The idea of **Kadane's algorithm** is to maintain a variable **max_ending_here** that stores the maximum sum contiguous sub-array ending at current index and a variable **max_so_far** stores the maximum sum of contiguous sub-array found so far, Everytime there is a positive-sum value in **max_ending_here** compare it with **max_so_far** and update **max_so_far** if it is greater than **max_so_far**.

So the main Intuition behind Kadane's algorithm is,

– The sub-array with negative sum is discarded (*by assigning max_ending_here = 0 in code*).

– We carry sub-array till it gives positive sum.

Pseudocode: *Initialize:*

max_so_far = INT_MIN

max_ending_here = 0

Loop for each element of the array

(a) max_ending_here = max_ending_here + a[i]

(b) if(max_so_far < max_ending_here)

max_so_far = max_ending_here

(c) if(max_ending_here < 0)

max_ending_here = 0

return max_so_far

Illustration:

Lets take the example: {-2, -3, 4, -1, -2, 1, 5, -3}
max_so_far = INT_MIN
max_ending_here = 0
for i=0, a[0] = -2
max_ending_here = max_ending_here + (-2)
Set max_ending_here = 0 because max_ending_here < 0
and set max_so_far = -2
for i=1, a[1] = -3
max_ending_here = max_ending_here + (-3)
Since max_ending_here = -3 and max_so_far = -2, max_so_far will remain -2
Set max_ending_here = 0 because max_ending_here < 0
for i=2, a[2] = 4
max_ending_here = max_ending_here + (4)
max_ending_here = 4
max_so_far is updated to 4 because max_ending_here greater
than max_so_far which was -2 till now
for i=3, a[3] = -1
max_ending_here = max_ending_here + (-1)
max_ending_here = 3
for i=4, a[4] = -2
max_ending_here = max_ending_here + (-2)
max_ending_here = 1
for i=5, a[5] = 1
max_ending_here = max_ending_here + (1)
max_ending_here = 2
for i=6, a[6] = 5
max_ending_here = max_ending_here + (5)
max_ending_here = 7
max_so_far is updated to 7 because max_ending_here is
greater than max_so_far
for i=7, a[7] = -3
max_ending_here = max_ending_here + (-3)
max_ending_here = 4

Follow the below steps to Implement the idea:

- Initialize the variables **max_so_far** = INT_MIN and **max_ending_here** = 0
- Run a for loop from **0** to **N-1** and for each index **i**:
  - Add the arr[i] to max_ending_here.

- If max_so_far is less than max_ending_here then update max_so_far to max_ending_here.
- If max_ending_here < 0 then update max_ending_here = 0

- Return max_so_far

Example: **Let's see at an example of how Kadane's algorithm works:**

Suppose we have the following array of integers: **arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]**

We want to find the maximum sub-array sum of this array. We can apply Kadane's algorithm to solve this problem.

We start by initializing two variables:

- **max_so_far:** This variable will keep track of the maximum sub-array sum we have seen so far.
- **max_ending_here:** This variable will keep track of the maximum sum ending at the current index.

max_so_far = INT_MIN;

max_ending_here = 0;

- Then, we iterate through the array, starting from the second element:
  for i in range(1, len(arr)):
- Update the current sum by adding the current element to the previous sum:
  max_ending_here = max(arr[i], max_ending_here + arr[i])
- Update the maximum sum seen so far:
  max_so_far = max(max_so_far, max_ending_here)

At each iteration, we update the current sum by either adding the current element to the previous sum or starting a new sub-array at the current element. We then update the maximum sum seen so far by comparing it with the current sum.

After iterating through the entire array, the value of max_so_far will be the maximum sub-array sum of the given array.

In this example, the maximum sub-array sum is 6, which corresponds to the sub-array [4, -1, 2, 1].

Code Implementation in C++:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
   int a[] = { -2, -3, 4, -1, -2, 1, 5, -3 };
  int n = sizeof(a) / sizeof(a[0]);
 int max_so_far = INT_MIN, max_ending_here = 0;   //  Kadane's algorithm
   for (int i = 0; i < n; i++)
 {
      max_ending_here = max_ending_here + a[i];
  if (max_so_far < max_ending_here)
```

```
        max_so_far = max_ending_here;
 if (max_ending_here < 0)
       max_ending_here = 0;
   }
cout << "Maximum contiguous sum in the array is : "<<max_so_far<<endl;
   return 0;
}
```

Output

```
Maximum contiguous sum in the array is : 7
```

Advantages of Kadane's Algorithm:

- **Efficiency:** Kadane's Algorithm has a time complexity of O(n), which makes it very efficient for solving the maximum sub-array problem. This makes it a great solution for large datasets.
- **Simplicity:** Kadane's Algorithm is relatively easy to understand and implement compared to other algorithms for solving the maximum sub-array problem, such as the divide-and-conquer algorithm.
- **Space Complexity:** Kadane's Algorithm has a space complexity of O(1), which means it uses a constant amount of memory irrespective of the size of the input array.
- **Dynamic Programming:** Kadane's Algorithm is a classic example of dynamic programming, a technique that breaks down a problem into smaller sub problems and stores the solutions to these sub problems to avoid redundant computation.

Disadvantages of Kadane's Algorithm:

- **Only finds sum and not the sub-array itself:** Kadane's Algorithm only finds the maximum sum of the sub-array and not the actual sub-array itself. If you need to find the sub-array that has the maximum sum, you will need to modify the algorithm accordingly.
- **Does not handle negative numbers well:** If an input array has only negative numbers, the algorithm will return the maximum negative number instead of 0. This can be overcome by adding an additional step to the algorithm to check if the array has only negative numbers.
- **Not suitable for non-contiguous sub-arrays:** Kadane's Algorithm is specifically designed for contiguous sub-arrays and may not be suitable for solving problems that involve non-contiguous sub-arrays.

Applications of Kadane's algorithm:

There are some of its applications like the following:

- **Maximum sub-array sum:** As we saw in the example above, Kadane's algorithm is used to find the maximum sub-array sum of an array of integers. This is a common problem in computer science and has applications in data analysis, financial modeling, and other fields.
- **Stock trading:** Kadane's algorithm can be used to find the maximum profit that can be made by buying and selling a stock on a given day. The input to the algorithm is an array of stock

prices, and the output is the maximum profit that can be made by buying and selling the stock at different times.

- **Image processing:** Kadane's algorithm can be used in image processing applications to find the largest contiguous area of pixels that meet a certain condition, such as having a certain color or brightness. This can be useful for tasks such as object recognition and segmentation.
- **DNA sequencing:** Kadane's algorithm can be used in bioinformatics to find the longest subsequence of DNA that meets certain conditions. For example, it can be used to find the longest common subsequence between two DNA sequences or to find the longest subsequence that does not contain certain patterns.
- **Machine learning:** Kadane's algorithm can be used in some machine learning applications, such as reinforcement learning and dynamic programming, to find the optimal policy or action sequence that maximizes a reward function.

Therefore, we can say the advantages of Kadane's Algorithm make it a great solution for solving the maximum sub-array problem, especially for large datasets. However, its limitations must be considered when using it for specific applications.

An Activity Selection Problem

The activity selection problem is a mathematical optimization problem. Our first illustration is the problem of scheduling a resource among several challenge activities. We find a greedy algorithm provides a well designed and simple method for selecting a maximum- size set of manually compatible activities.

Suppose S = {1, 2....n} is the set of n proposed activities. The activities share resources which can be used by only one activity at a time, e.g., Tennis Court, Lecture Hall, etc. Each Activity "i" has **start time** s_i and a **finish time** f_i, where $s_i \leq f_i$. If selected activity "i" take place meanwhile the half-open time interval $[s_i, f_i)$. Activities i and j are **compatible** if the intervals (s_i, f_i) and $[s_i, f_i)$ do not overlap (i.e. i and j are compatible if $s_i \geq f_i$ or $s_i \geq f_i$). The activity-selection problem chosen the maximum- size set of mutually consistent activities.

Algorithm of Greedy- Activity Selector:

GREEDY- ACTIVITY SELECTOR (s, f)

1. n ← length [s]
2. A ← {1}
3. j ← 1.
4. for i ← 2 to n
5. do if $s_i \geq f_i$
6. then A ← A ∪ {i}
7. j ← i
8. return A

Activity Selection Problem

Problem Statement

Problem - given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time

Example :

Activity	A1	A2	A3
Start	12	10	20
Finish	25	20	30

Solution

Greedy Approach -

1) Sort the activities according to their finishing time
2) Select the first activity from the sorted array and print it.
3) Do following for remaining activities in the sorted array.
   -If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

Greedy Approach -

1) Sort the activities according to their finishing time

Activity	A1	A2	A3	A4	A5	A6
Start	0	3	1	5	5	8
Finish	6	4	2	9	7	9

Sorted

Activity	A3	A2	A1	A5	A6	A4
Start	1	3	0	5	8	5
Finish	2	4	6	7	9	9

2) Select the first activity from the sorted array and print it.

Activity	A3	A2	A1	A5	A6	A4
Start	1	3	0	5	8	5
Finish	2	4	6	7	9	9

Answer = A3

3) If the start time of next activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

Activity	A3	A2	A1	A5	A6	A4
Start	1	3	0	5	8	5
Finish	2	4	6	7	9	9

Answer = A3 → A2 → A5 → A6

Example: Given 10 activities along with their start and end time as

S = (A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A9 A10)

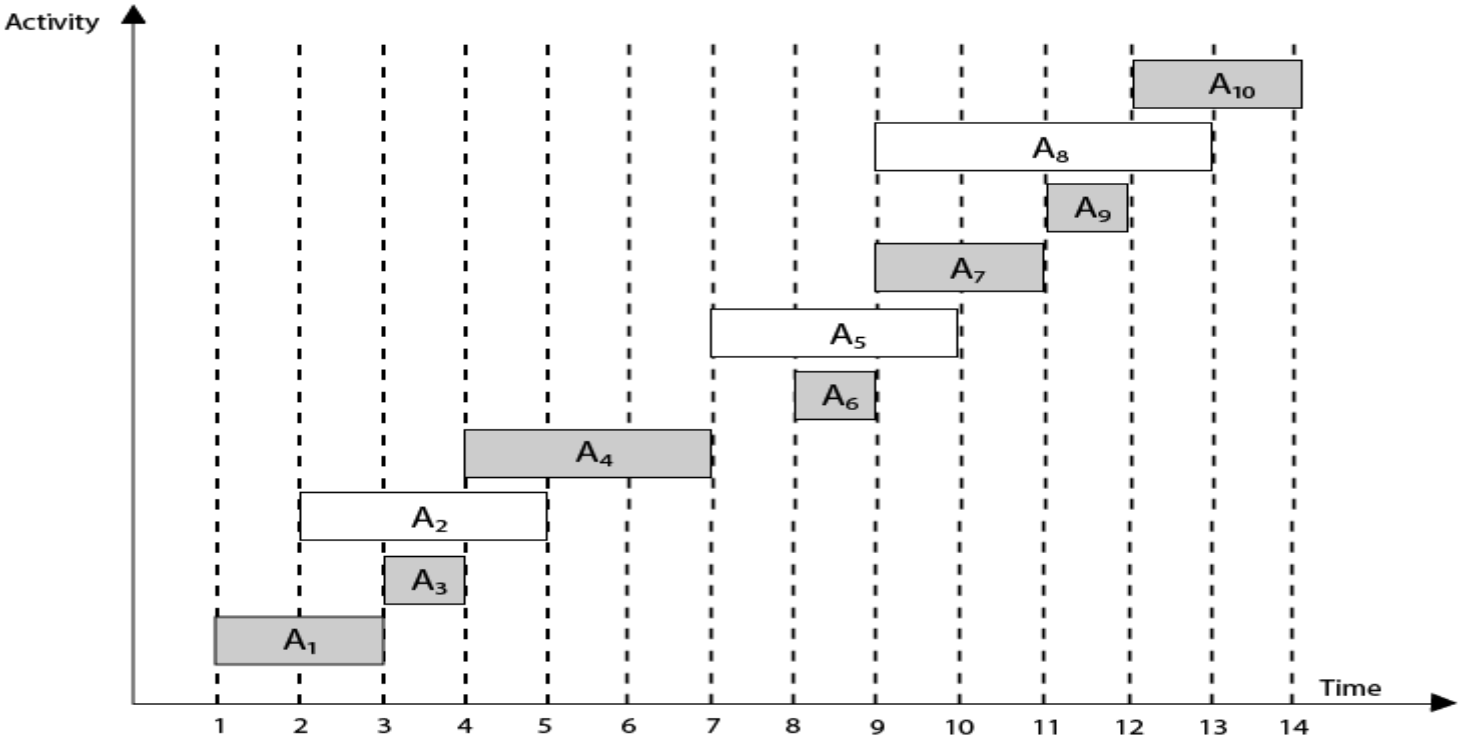
Si = (1,2,3,4,7,8,9,9,11,12)

fi = (3,5,4,7,10,9,11,13,12,14)

Solution: The solution to the above Activity scheduling problem using a greedy strategy is illustrated below:

Arranging the activities in increasing order of end time

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Now, schedule A_1

Next schedule A_3 as A_1 and A_3 are non-interfering.

Next **skip** A_2 as it is interfering.

Next, schedule A_4 as A_1 A_3 and A_4 are non-interfering, then next, schedule A_6 as A_1 A_3 A_4 and A_6 are non-interfering.

Skip A_5 as it is interfering.

Next, schedule A_7 as A_1 A_3 A_4 A_6 and A_7 are non-interfering.

Next, schedule A_9 as A_1 A_3 A_4 A_6 A_7 and A_9 are non-interfering.

Skip A_8 as it is interfering.

Next, schedule A_{10} as A_1 A_3 A_4 A_6 A_7 A_9 and A_{10} are non-interfering.

Thus the final Activity schedule is:

$$(A_1\ A_3\ A_4\ A_6\ A_7\ A_9\ A_{10})$$

CODE:

```
#include <stdio.h>
void printMaxActivities(int s[], int f[], int n)
{
        int i, j;
        printf("Following activities are selected\n");
        i = 0;          // the first activity always gets selected
        printf("%d ", i);
        for (j = 1; j < n; j++)          // Consider rest of the activities
         {
                if (s[j] >= f[i])
                 {
                   printf("%d ", j);
                        i = j;
                }
        }
}
int main()
{
        int s[] = { 1, 3, 0, 5, 8, 5 };
        int f[] = { 2, 4, 6, 7, 9, 9 };
        int n = sizeof(s) / sizeof(s[0]);
        printMaxActivities(s, f, n);          // Function call
        return 0;
}
```

Window Sliding Technique

Window Sliding Technique is a computational technique that aims to reduce the use of nested loops and replace it with a single loop, thereby reducing the time complexity.

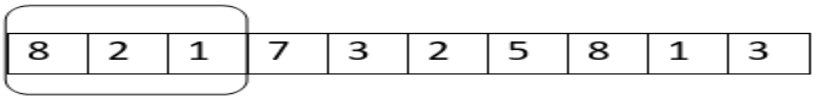
The concept here is that we create a window of size k, and we'll keep sliding it by a unit index. Here, the window isn't any technical term. Rather than using a single value as we do in loops, we use multiple elements simultaneously in each iteration.

For example:

Given an array of size 10:

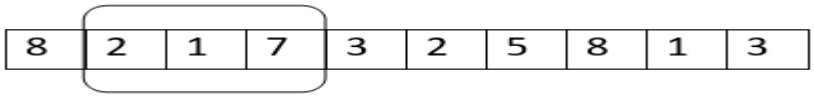
Suppose we need the maximum sum of 3 consecutive indexes, create a 3-sized window, and keep sliding (traversing) it throughout the array. Here is a pictorial representation:

Iteration 1:



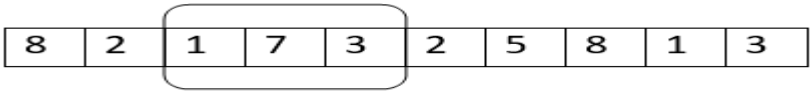
Sum = 8 + 2 + 1 = 11

Iteration 2:



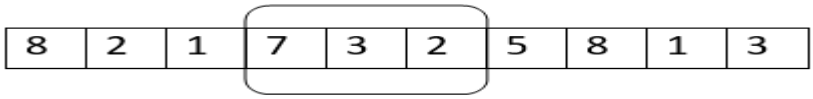
Sum = 2 + 1 + 7 = 10

Iteration 3:



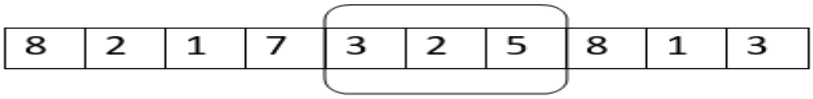
Sum = 1 + 7 + 3 = 11

Iteration 4:



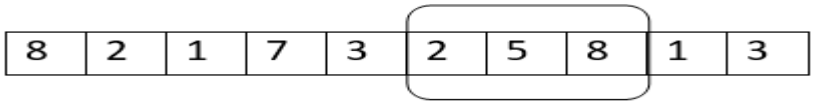
Sum = 7 + 3 + 2 = 12

Iteration 5:



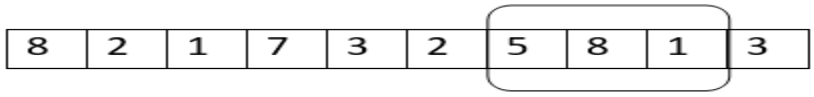
Sum = 3 + 2 + 5 = 10

Iteration 6:



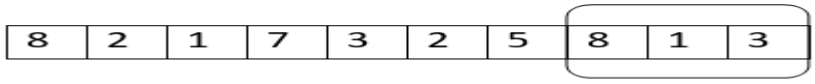
Sum = 2 + 5 + 8 = 15

Iteration 7:



Sum = 5 + 8 + 1 = 14

Iteration 8:



Sum = 8 + 1 + 3 = 12

- Using this method, there will be no inner loop, and the number of iterations of the one single loop will be (n - k + 1), 8 in this case.
- So, a Sliding window is a technique used to reduce the use of nested loops and replace it with a single loop to reduce the total time complexity.
- Observe that in every iteration, when the window is sliding to the next index, **we're deleting the first element from the previous window and adding a new element that is the next succeeding index.**

Code:

```
#include <stdio.h>
int maxsum(int a[], int k, int n);
int main()
{
  int n, i, k;
 printf("Enter the size of the array: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the elements: ");
    for(i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("Enter the value of k: ");
    scanf("%d", &k);
    int max = maxsum(arr, k, n);
    printf("The maximum sum of %d consecutive elements in the array: %d", k,max);
}
int maxsum(int a[], int k, int n)
{
    int i, sum, maxm = 0;
  for(i = 0; i < k; i++)
  {
     maxm = maxm + a[i];
  }
  sum = maxm;
    for(i = k; i < n; i++)
   {
  sum += a[i] - a[i - k]; /*subtract the first element of the previous window and add the next index*/
  if(sum > maxm)
    {
      maxm = sum;
    }
 }
return maxm;
}
```

Output:

```
Enter the size of the array: 10
Enter the elements: 8 2 1 7 3 2 5 8 1 3
Enter the value of k: 3
The maximum sum of 3 consecutive elements in the array: 15
```

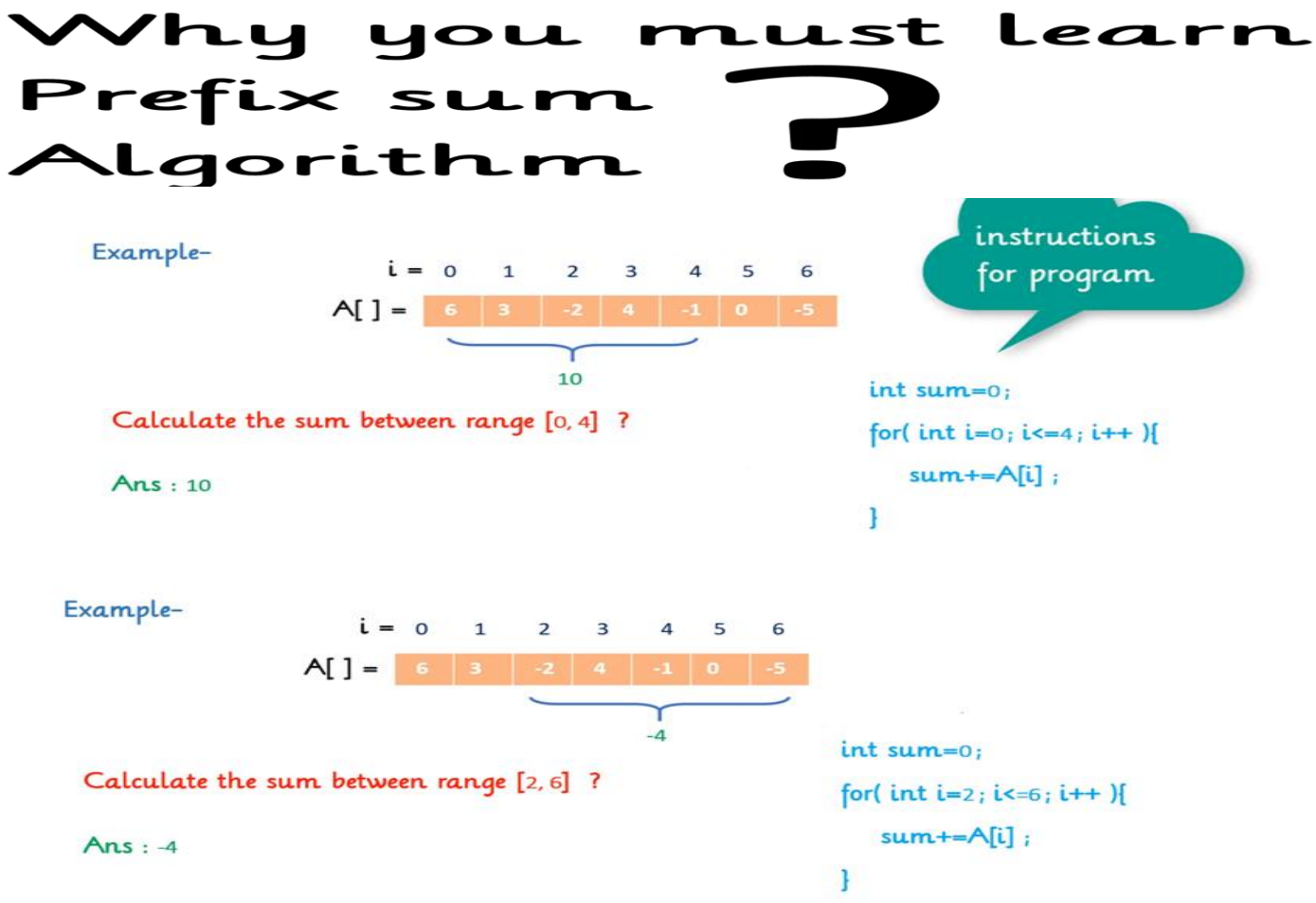
- The naïve approach takes O(k*n) time with two nested loops.
- The time complexity is reduced to O(n) by using the Sliding window technique.

Here are the steps to apply the technique to any problem at hand:

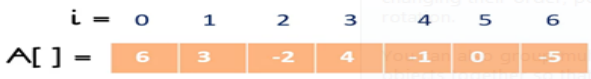
1. First, we must see that the window size is constant and shouldn't change. We can use the technique to only such problems.
2. After you ensure that the window size isn't changing, compute the result of the first window to compare to the computations of the rest of the array.
3. Now, use a loop to slide the window index by index till the end and keep updating the required value.

Prefix Sum Algorithm

Given an array **arr[]** of size **N**, find the prefix sum of the array. A prefix sum array is another array **prefixSum[]** of the same size, such that the value of **prefixSum[i]** is **arr[0] + arr[1] + arr[2] . . . arr[i]**.



Example-



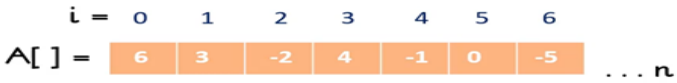
Calculate the sum between range [0, n) ?

```
int sum=0;
for( int i=0; i<n; i++ ){
    sum+=A[i];
}
```

n

Time complexity - O(n)

Example-



Calculate the sum between range [0, 3] ?
Calculate the sum between range [2, 3] ?
Calculate the sum between range [4, 6] ?
.
.
m

m

```
for(int j=1; j<m; j++){
    int sum=0;
    for( int i=start; i<end; i++ ){
        sum+=A[i] ;
    }
}
```

n m

Time complexity - O(m*n)

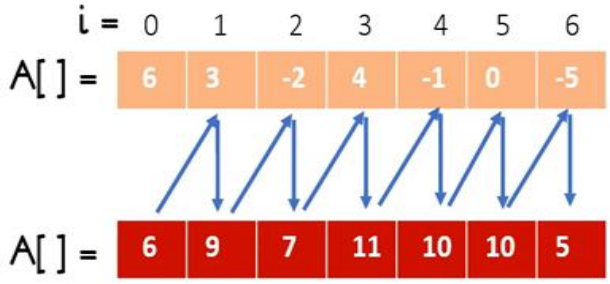
Key takeaway from this lesson -

- A normal algorithm takes O(m*n) time to perform m number of queries to find range sum on n size array.
- Prefix sum algorithm takes O(n) time to perform m number of queries to find range sum on n size array.

Prefix sum

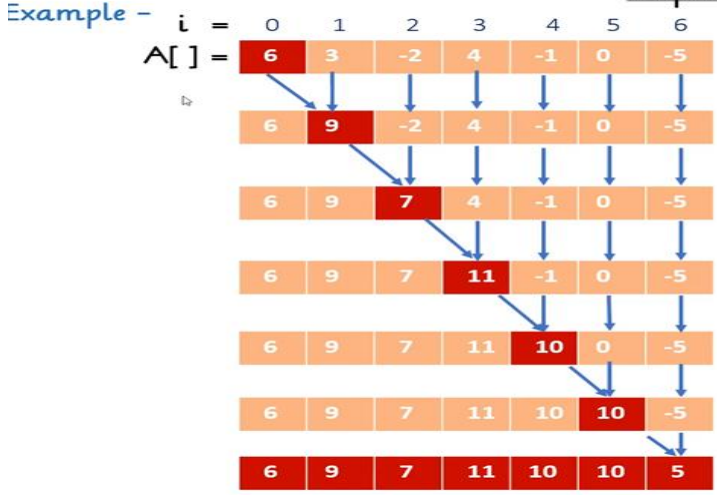
It is a simple yet powerful technique that allows to perform fast calculation on the sum of elements in a given range (called contiguous segments of array).

Example -



Prefix sum

Example –




```
for( int i=1; i<n; i++ ){
    A[i] = A[i]+A[i-1];
}
```

n

Prefix sum

Example - i = 0 1 2 3 4 5 6

A[ ] = 6 3 -2 4 -1 0 -5

Prefix Sum Array -

i = 0 1 2 3 4 5 6

A[ ] = 6 9 7 11 10 10 5

To calculate the sum between range [i, j]

Formula - $A[i, j] = A[j] - A[i-1]$

Calculate the sum between range [3, 5] ?

$A[3, 5] = A[5] - A[3-1]$

$A[3, 5] = A[5] - A[2]$

$A[3, 5] = 10 - 7$

$A[3, 5] = 3$

Calculate the sum between range [0, 4] ?

Ans : 10 (=A[4]) – O(1)

Calculate the sum between range [0, 6] ?

Ans : 5 (=A[6]) – O(1)

Analysis of Algorithm –

- To calculate prefix sum array of n size array
  Time complexity – O(n)
- Time taken to perform range sum query is –
  Time complexity – O(1)
- Total time taken to pre process the n size array and to perform range query is –
  Time complexity – O(n) + O(1)
  ~ O(n)

Key takeaway from this lesson –

- Range sum query formula– $A[i, j] = A[j] - A[i-1]$
- It takes O(n) time to calculate prefix sum array of n size array.
- It takes O(1) time to perform range sum query on n size array.

Examples:

Input: *arr[] = {10, 20, 10, 5, 15}*

Output: *prefixSum[] = {10, 30, 40, 45, 60}*

Explanation: *While traversing the array, update the element by adding it with its previous element.*
prefixSum[0] = 10,
prefixSum[1] = prefixSum[0] + arr[1] = 30,
prefixSum[2] = prefixSum[1] + arr[2] = 40 and so on.

Approach: To solve the problem follow the given steps:

- Declare a new array **prefixSum[]** of the same size as the input array
- Run a for loop to traverse the input array
- For each index add the value of the current element and the previous value of the prefix sum array

Code:

```
#include <stdio.h>
void fillPrefixSum(int arr[], int n, int prefixSum[])          // Fills prefix sum array
{
        prefixSum[0] = arr[0];
        for (int i = 1; i < n; i++)       // Adding present element with previous element
                prefixSum[i] = prefixSum[i - 1] + arr[i];
}
int main()
{
        int arr[] = { 10, 4, 16, 20 };
        int n = sizeof(arr) / sizeof(arr[0]);
        int prefixSum[n];
        fillPrefixSum(arr, n, prefixSum);    // Function call
        for (int i = 0; i < n; i++)
                printf("%d ", prefixSum[i]);
}
```

Applications of Prefix Sum:

Equilibrium index of an array: The equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes.

Find if there is a subarray with 0 sums: Given an array of positive and negative numbers, find if there is a subarray (of size at least one) with 0 sum.

Maximum subarray size, such that all subarrays of that size have a sum less than k: Given an array of n positive integers and a positive integer k, the task is to find the maximum subarray size such that all subarrays of that size have the sum of elements less than k.

Find the prime numbers which can be written as sum of most consecutive primes: Given an array of limits. For every limit, find the prime number which can be written as the sum of the most consecutive primes smaller than or equal to the limit.

Longest Span with same Sum in two Binary arrays: Given two binary arrays, arr1[] and arr2[] of the same size n. Find the length of the longest common span (i, j) where j >= i such that arr1[i] + arr1[i+1] + …. + arr1[j] = arr2[i] + arr2[i+1] + …. + arr2[j].

Maximum subarray sum modulo m: Given an array of n elements and an integer m. The task is to find the maximum value of the sum of its subarray modulo m i.e find the sum of each subarray mod m and print the maximum value of this modulo operation.

Maximum subarray size, such that all subarrays of that size have sum less than k: Given an array of n positive integers and a positive integer k, the task is to find the maximum subarray size such that all subarrays of that size have the sum of elements less than k.

Maximum occurred integer in n ranges : Given n ranges of the form L and R, the task is to find the maximum occurring integer in all the ranges. If more than one such integer exits, print the smallest one.

Minimum cost for acquiring all coins with k extra coins allowed with every coin: You are given a list of N coins of different denominations. you can pay an amount equivalent to any 1 coin and can acquire that coin. In addition, once you have paid for a coin, we can choose at most K more coins and can acquire those for free. The task is to find the minimum amount required to acquire all the N coins for a given value of K.

Random number generator in arbitrary probability distribution fashion: Given n numbers, each with some frequency of occurrence. Return a random number with a probability proportional to its frequency of occurrence.

Bit manipulation

The computer does not understand the high-level language in which we communicate. For these reasons, there was a standard method by which any instruction given to the computer was understood. At an elementary level, each instruction was sent into some digital information known as bits. The series of bits indicates that it is a particular instruction.

Bit

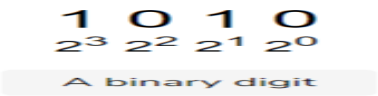
A bit is defined as the basic unit which stores the data in digital notation.

Two values represent it as follows -

1 - It indicates the signal is present or True

0 - It indicates the signal is absent or False

Bits represent the logical state of any instruction. The series of bits have a base which is 2. Thus if we say if we have a series of binary digits, it is read from left to right, and the power of 2 increases.



A binary digit

Bit manipulation

Bit manipulation is defined as performing some basic operations on bit level of n number of digits. Bit manipulation is the process of applying logical operations on a sequence of bits to achieve a required result. Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a word.

Computer programming tasks that require bit manipulation include:

- Low-level device control
- Error detection and correction algorithms
- Data compression
- Encryption algorithms
- Optimization

Bit-level operations

- Sometimes, it becomes mandatory to consider data at the bit level.
- We have to operate on the individual data bit. We must also turn on/off particular data bits during source code drafting. At that time, we must use a bitwise operator to make our task more manageable.
- Programming languages provide us with different bitwise operators for manipulating bits.
- Bitwise operators operate on integers and characters but not on data types float or double.
- Using Bitwise operators, we can easily manipulate individual bits.
- Programming languages support six bitwise operators.

Bitwise Operator in C

The bitwise operators are the operators used to perform the operations on the data at the bit-level. When we perform the bitwise operations, then it is also known as bit-level programming. It consists of two digits, either 0 or 1. It is mainly used in numerical computations to make the calculations faster.

We have different types of bitwise operators in the C programming language. The following is the list of the bitwise operators:

Operator	Meaning of operator
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise exclusive OR operator
~	One's complement operator (unary operator)
<<	Left shift operator
>>	Right shift operator

Let's look at the truth table of the bitwise operators.

X	Y	X&Y	X|Y	X^Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise AND operator

Bitwise AND operator is denoted by the single ampersand sign (&). Two integer operands are written on both sides of the (&) operator. If the corresponding bits of both the operands are 1, then the output of the bitwise AND operation is 1; otherwise, the output would be 0.

For example,

We have two variables a and b.

a =6;

b=4;

The binary representation of the above two variables are given below:

a = 0110

b = 0100

When we apply the bitwise AND operation in the above two variables, i.e., a&b, the output would be:

Result = 0100

As we can observe from the above result that bits of both the variables are compared one by one. If the bit of both the variables is 1 then the output would be 1, otherwise 0.

Let's understand the bitwise AND operator through the program.

```
#include <stdio.h>
int main()
{
   int a=6, b=14;  // variable declarations
 printf("The output of the Bitwise AND operator a&b is %d",a&b);
return 0;
}
```

In the above code, we have created two variables, i.e., 'a' and 'b'. The values of 'a' and 'b' are 6 and 14 respectively. The binary value of 'a' and 'b' are 0110 and 1110, respectively. When we apply the AND operator between these two variables,

Output: **a AND b = 0110 && 1110 = 0110**

Bitwise OR operator

The bitwise OR operator is represented by a single vertical sign (|). Two integer operands are written on both sides of the (|) symbol. If the bit value of any of the operand is 1, then the output would be 1, otherwise 0.

For example,

We consider two variables,

a = 23;

b = 10;

The binary representation of the above two variables would be:

a = 0001 0111

b = 0000 1010

When we apply the bitwise OR operator in the above two variables, i.e., a|b , then the output would b e:

Result = 0001 1111

As we can observe from the above result that the bits of both the operands are compared one by one; if the value of either bit is 1, then the output would be 1 otherwise 0.

Let's understand the bitwise OR operator through a program.

```
#include <stdio.h>
int main()
{
   int a=23,b=10;  // variable declarations
   printf("The output of the Bitwise OR operator a|b is %d",a|b);
   return 0;
}
```

Output: a | b = 0001 1111

Bitwise exclusive OR operator

Bitwise exclusive OR operator is denoted by (^) symbol. Two operands are written on both sides of the exclusive OR operator. If the corresponding bit of any of the operand is 1 then the output would be 1, otherwise 0.

For example,

We consider two variables a and b,

a = 12;

b = 10;

The binary representation of the above two variables would be:

a = 0000 1100

b = 0000 1010

When we apply the bitwise exclusive OR operator in the above two variables (a^b), then the result would be:

Result = 0000 1110

As we can observe from the above result that the bits of both the operands are compared one by one; if the corresponding bit value of any of the operand is 1, then the output would be 1 otherwise 0.

Let's understand the bitwise exclusive OR operator through a program.

```
#include <stdio.h>
int main()
{
   int a=12,b=10;  // variable declarations
  printf("The output of the Bitwise exclusive OR operator a^b is %d",a^b);
  return 0;
}
```

Output: **a** ^ b = 0000 1110

Bitwise complement operator

The bitwise complement operator is also known as one's complement operator. It is represented by the symbol tilde (~). It takes only one operand or variable and performs complement operation on an operand. When we apply the complement operation on any bits, then 0 becomes 1 and 1 becomes 0.

For example,

If we have a variable named 'a',

a = 8;

The binary representation of the above variable is given below:

a = 1000

When we apply the bitwise complement operator to the operand, then the output would be:

Result = 0111

As we can observe from the above result that if the bit is 1, then it gets changed to 0 else 1.

Let's understand the complement operator through a program.

```
#include <stdio.h>
int main()
{
int a=8;  // variable declarations
 printf("The output of the Bitwise complement operator ~a is %d",~a);
  return 0;
}
```

Output: ~a = 0111

Bitwise shift operators

Two types of bitwise shift operators exist in C programming. The bitwise shift operators will shift the bits either on the left-side or right-side. Therefore, we can say that the bitwise shift operator is divided into two categories:

- Left-shift operator
- Right-shift operator

Left-shift operator

It is an operator that shifts the number of bits to the left-side.

Syntax of the left-shift operator is given below: **Operand << n**

Where, Operand is an integer expression on which we apply the left-shift operation.

n is the number of bits to be shifted.

In the case of Left-shift operator, 'n' bits will be shifted on the left-side. The 'n' bits on the left side will be popped out, and 'n' bits on the right-side are filled with 0.

For example,

Suppose we have a statement:

```
int a = 5;
```

The binary representation of 'a' is given below:

a = 0101

If we want to left-shift the above representation by 2, then the statement would be:

a << 2; 0101<<2 = 00010100

Let's understand through a program.

```
#include <stdio.h>
int main()
{
   int a=5; // variable initialization
  printf("The value of a<<2 is : %d ", a<<2);
  return 0;
}
```

Output: **a<<2 = 10100**

Right-shift operator

It is an operator that shifts the number of bits to the right side.

Syntax of the right-shift operator is given below**:** **Operand >> n;**

Where, Operand is an integer expression on which we apply the right-shift operation.

N is the number of bits to be shifted.

In the case of the right-shift operator, 'n' bits will be shifted on the right-side. The 'n' bits on the right-side will be popped out, and 'n' bits on the left-side are filled with 0.

For example,

Suppose we have a statement,

```
int a = 7;
```

The binary representation of the above variable would be:

a = 0111

If we want to right-shift the above representation by 2, then the statement would be:

a>>2;

0000 0111 >> 2 = 0000 0001

Let's understand through a program.

```
#include <stdio.h>
int main()
{
   int a=7; // variable initialization
   printf("The value of a>>2 is : %d ", a>>2);
   return 0;
}
```

Output: a>>2 = 0001