4

Q- ① Space Complexity :- It is ~~[illegible]~~ a process of determining formula for prediction how much time will taken for successfull exection of algorithm in memory space generally we consider the space of primary memory.

② Time Complexity :- Time Complexity measure the amount of time which algorithm takes to run as a function of input size. It helps to determine how much running time will increase with input size.

The time required by algorithm in three types

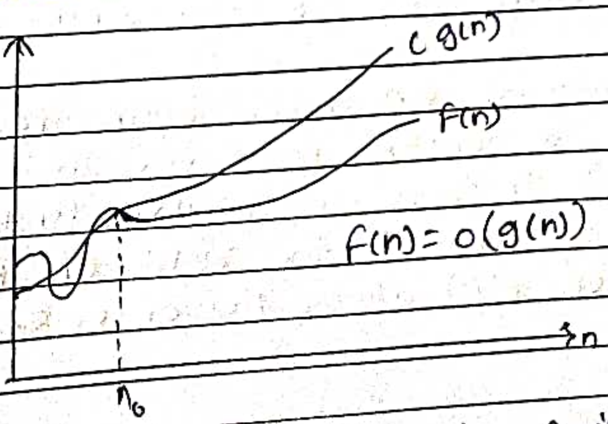
① Best case :- minimum time required for ex. of Prog

② Average : Average time

③ worst :- Maximum time

Asymptotic Notation :- They are used to make a meaningful statement about the efficiency of algorithm there notation helps us to make approximate but meaningful assumption about time and space Complexity

Big-Oh Notation (O)

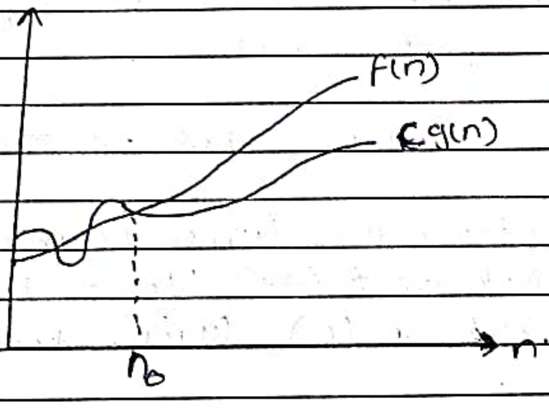


$\Theta(g(n)) = \{ f(n) :$ There exist positive constants c&n such that $0 \leq f(n) \leq c\,g(n)$ for all $n \geq n_0 \}$

Date.....................

O-Notation gives an upper bound for a function to within a constant factor. we write $f(n) = o(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of f(n) always lies on or below c g(n)

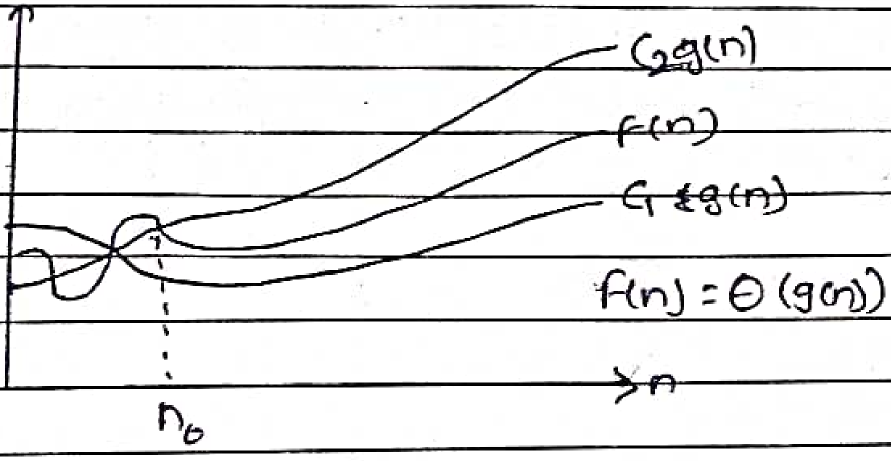
Big-Omega Notation:



$\Omega(g(n)) = \{ f(n) :$ There exist positive constants c and n_0 such that $0 \leq c.g(n) \leq f(n) \; \forall \; n \geq n_0 \}$

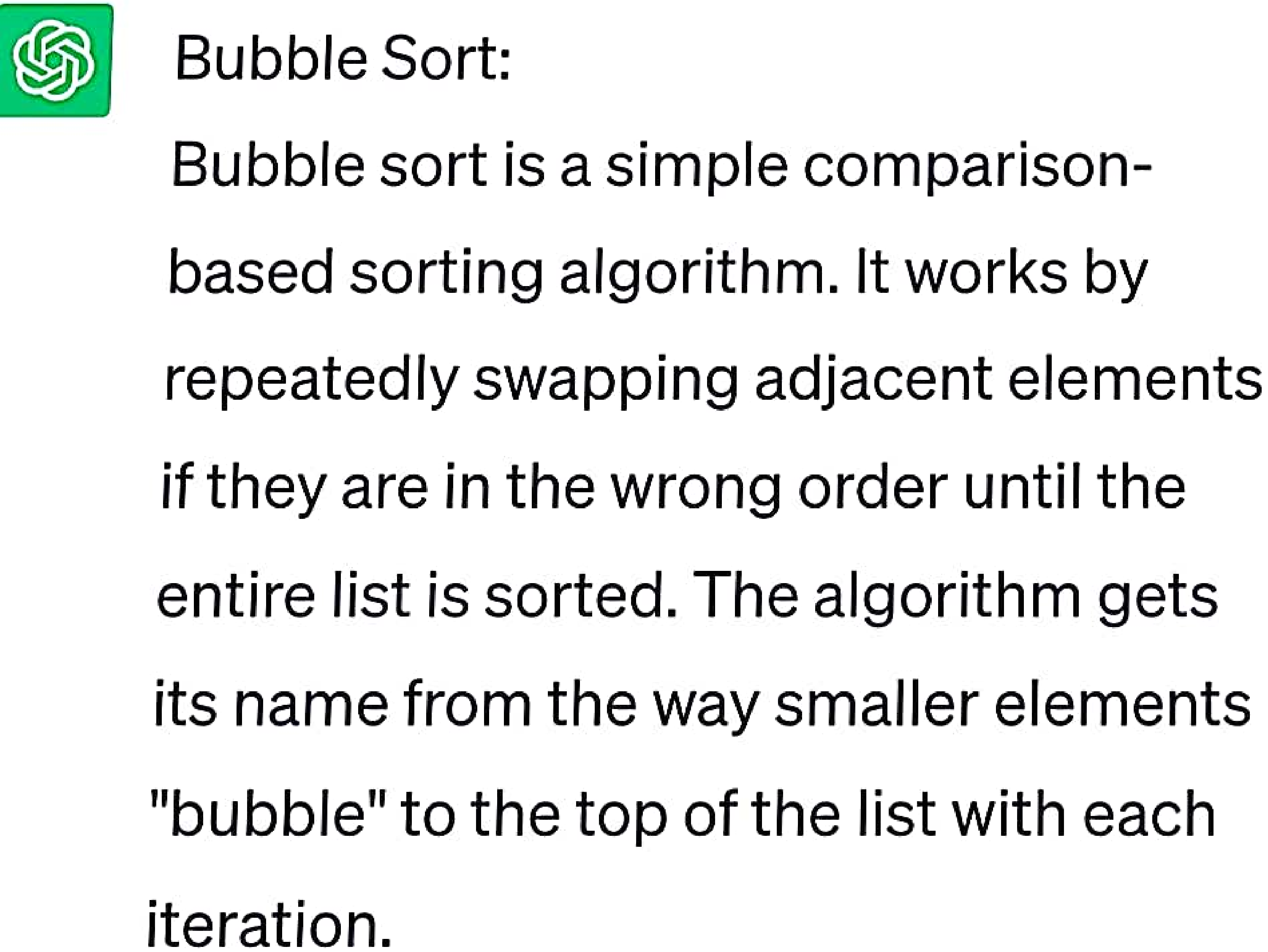
Ω-Notation gives an lower bound for a function to written a constant factor we write. $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of f(n) always lies on or above c g(n).

Big-Theta Notation



$\Theta(g(n)) = \{ f(n)$: There exist positive constants c_1, c_2 and n_0 such that $0 \le c_1g(n) \le f(n) \le c_2g$
$\forall \; n \ge n_0 \}$

Θ-Notation gives tight bounds for a function to written a constant factor. we write $f(n) = \Theta(g(n))$ if there are positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive.



Bubble Sort:

Bubble sort is a simple comparison-based sorting algorithm. It works by repeatedly swapping adjacent elements if they are in the wrong order until the entire list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list with each iteration.

Insertion Sort:

Insertion sort is another comparison-based sorting algorithm that builds the final sorted list one element at a time. It iterates over the list and, for each element, compares it with the elements already sorted to the left. It shifts the larger elements to the right to make room for the current element and inserts it in the correct position.

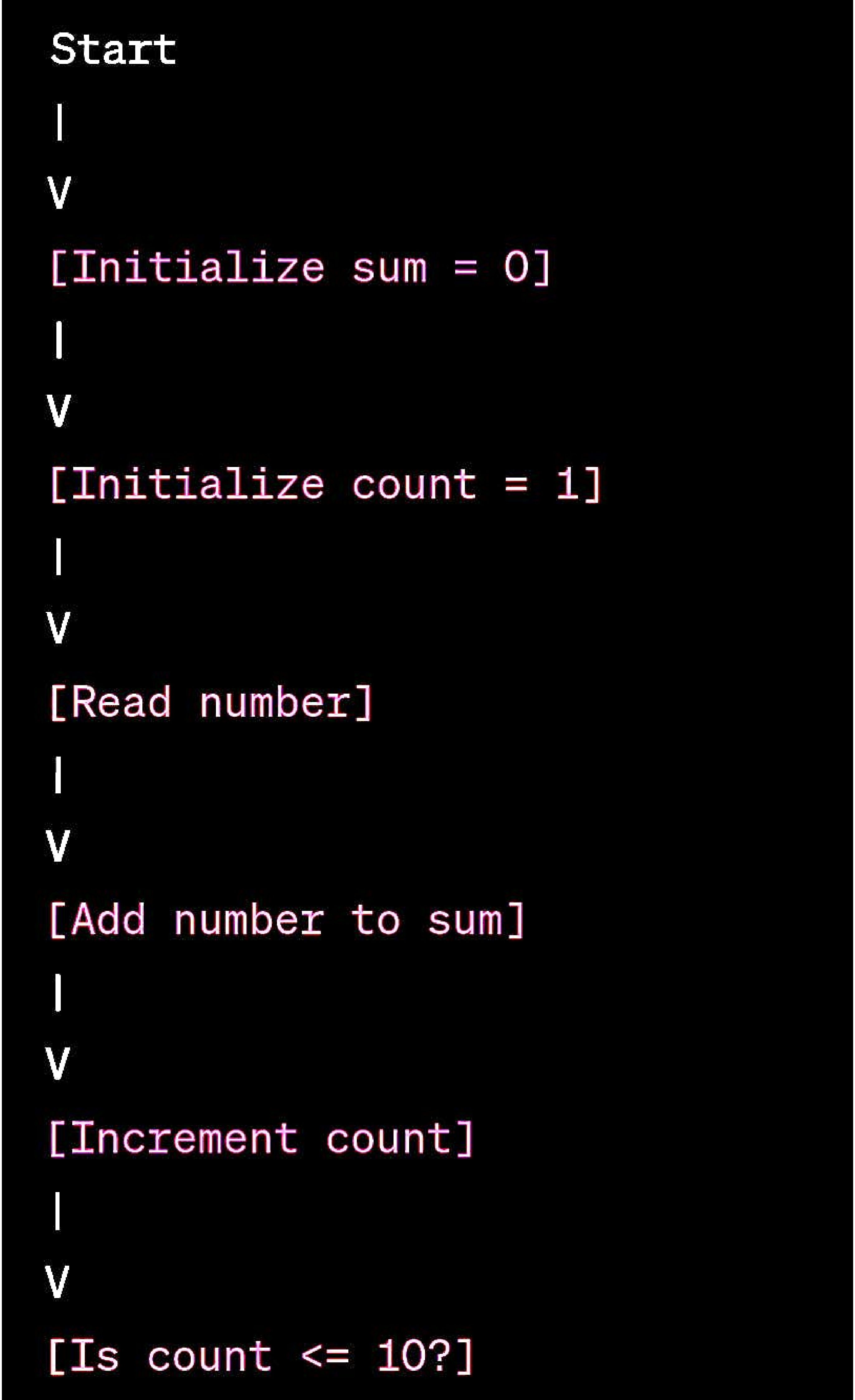
```
# include <stdio.h>

int binary search (int arr[] , int left , int right, int
                                                  target) {
 {
 while ( left <= right)
{~~int~~
  int mid = left + (right - left) / 2 ;
 if (arr [mid] ] = = target)
          return mid ;
~~else~~ if (arr[mid] < target)
          ~~retur~~
        left = mid +1
 else
      right = mid - 1.
 }
 return -1 ;
}
 int main ()   {
 int arr [] = { 1 , 2 , 3 , 4 , 5 , 6 ---} ;
 int n = size of (arr) / sizeof (arr[0]);
 int target = 5 ;
 int index = binary search (arr, target , 0, n-1) ;
if ( index ! = -1)
   printf ("Element %d found at index %.d /n, target,
                                           index);
else
   printf ("Element %.d not found at array \n", target);
}
```

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

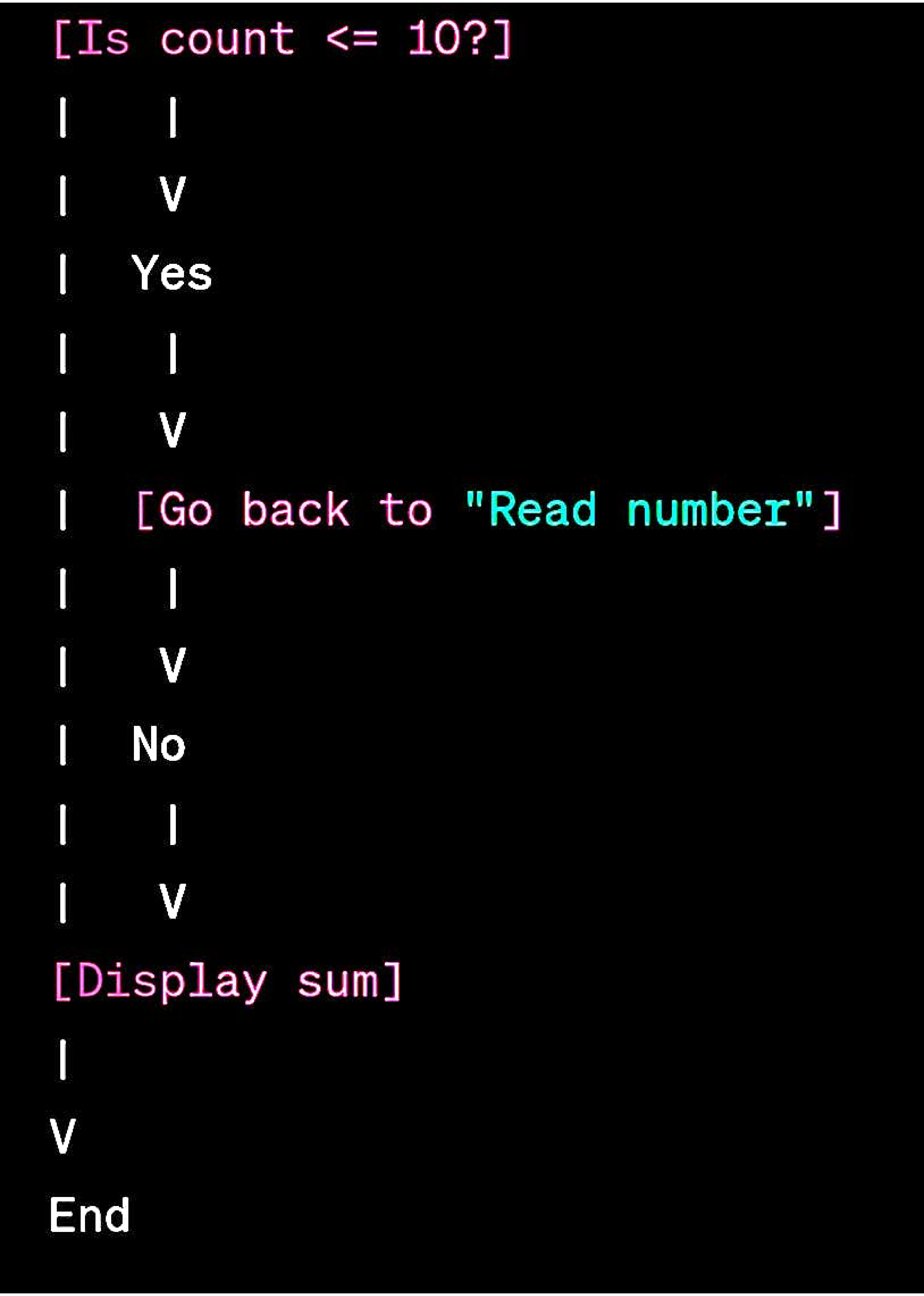
situation when we need operations for our user-defined data type which have to be defined. These operations can be defined only as and when we require them. So, in order to simplify the process of solving problems, we can create data structures along with their operations, and such data structures that are not in-built are known as Abstract Data Type (ADT).

1. Start/End Symbol: Represents the beginning or end of a process. It is usually represented by an oval or rounded rectangle.
2. Process Symbol: Represents a specific action or operation performed in the process. It is represented by a rectangle with rounded corners.
3. Input/Output Symbol: Represents input or output operations in the process. It is usually represented by a parallelogram.
4. Decision Symbol: Represents a decision point where a condition is evaluated and the flowchart branches out based on the result. It is represented by a diamond shape

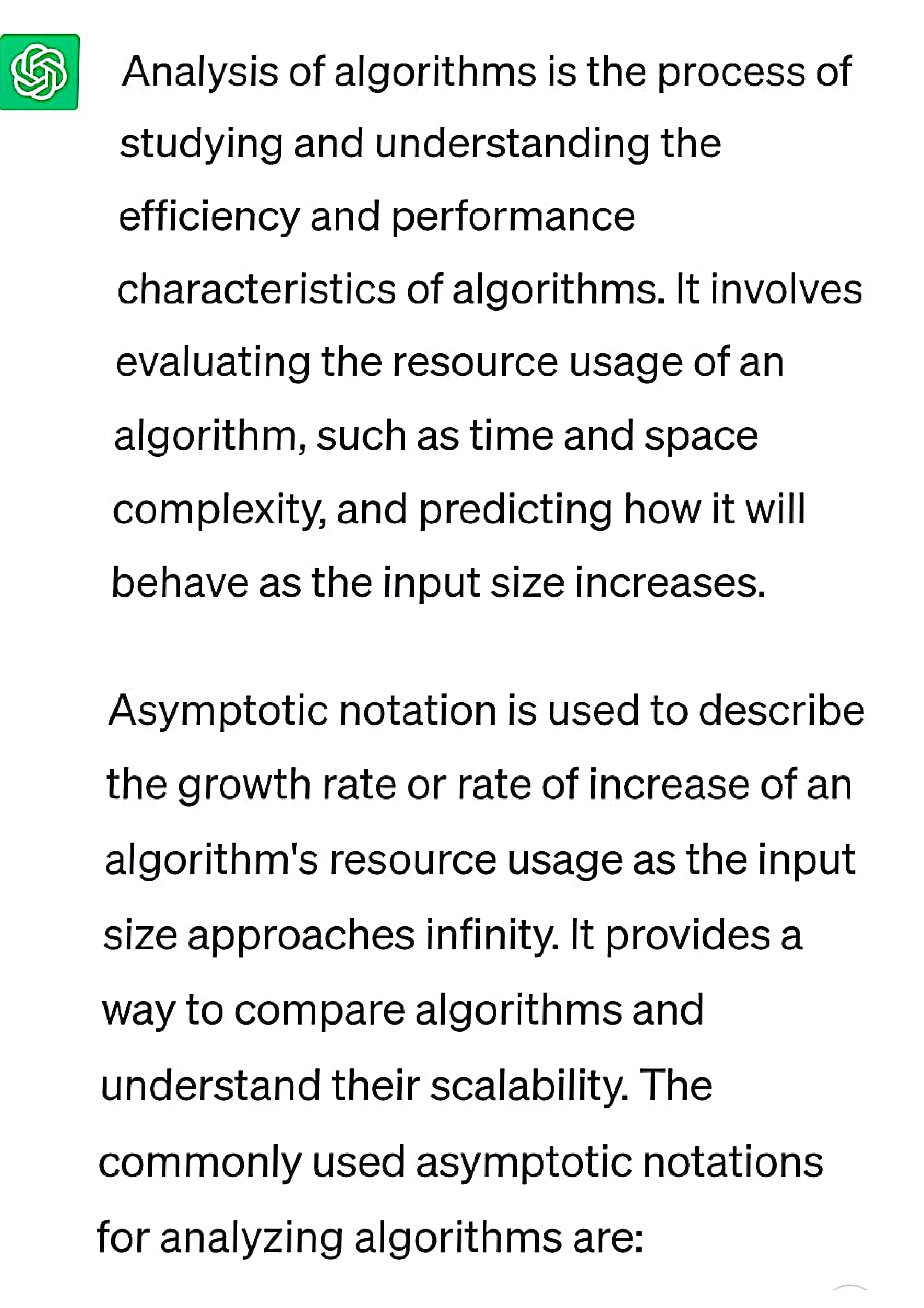
5. Connector Symbol: Represents the point where the flowchart connects to another part of the process on a different page or in a different location. It is represented by a small circle or dot.
6. Flowline/Arrow: Represents the flow or direction of the process. It connects the symbols and shows the sequence of actions.



```
Start
|
V
[Initialize sum = 0]
|
V
[Initialize count = 1]
|
V
[Read number]
|
V
[Add number to sum]
|
V
[Increment count]
|
V
[Is count <= 10?]
```



```
[Is count <= 10?]
|   |
|   V
|  Yes
|   |
|   V
|  [Go back to "Read number"]
|   |
|   V
|  No
|   |
|   V
[Display sum]
|
V
End
```



Analysis of algorithms is the process of studying and understanding the efficiency and performance characteristics of algorithms. It involves evaluating the resource usage of an algorithm, such as time and space complexity, and predicting how it will behave as the input size increases.

Asymptotic notation is used to describe the growth rate or rate of increase of an algorithm's resource usage as the input size approaches infinity. It provides a way to compare algorithms and understand their scalability. The commonly used asymptotic notations for analyzing algorithms are:

Que. write a c program to sort matrix in rowise

```
 # include <stdio.h>.
void sortMatrixRowise (int matrix [][3], int rows, int
 for (int i = 0 ; i< rows ; i++) {                       cols) {
   // Apply bubble sort for each row
 for (int j = 0 ; j< ~~rows~~ cols -1; j++) {
 For (int k =0 ; k< cols -j -1 ; k++) {
   int (matrix [i][k] > matrix [i][k+1]) {
     int temp = matrix[i][k];
    matrix [i][k] = matrix [i][k+1];
     matrix [i][k+1] = temp ;
     }
    }
   }
 }
}

 int main () {
  int matrix [][3] = {
        {9 , 6 , 3},
        {7 , 8 , 6},
        {8 , 1 , 4}
  };
  int rows = sizeof (matrix)/sizeof(matrix [0]);
 int cols = size of ({matrix [0])/ size of (matrix [0][0]);

 printf ("orignal matrix :\n);
 for (int i = 0; i< rows ; i++) {
 for (ij =0 ; j< cols ; j ++) {
   printf (" %d", matrix[i][j]);  }
   printf ("\n");
```

```
 sortMatrixRowise (matrix,
   rows, cols);

 Printf ("\n sorted Matrix
 (Row-wise):\n");
          {
          }
   return 0;
   }
```