



SIMATS
ENGINEERING



SIMATS
Saveetha Institute of Medical And Technical Sciences
(Declared as Deemed to be University under Section 3 of UGC Act 1956)

LAB REPORT

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
IN
COMPUTER SCIENCE AND ENGINEERING

Submitted by

J. Thejeswar Reddy (192210297)

Data Structures For Optimized Memory Usage – CSA0315

Under the Supervisor of

Dr. F. Mary Harin Fernandez
Dr. K. Sashi Rekha

Sep - 2025

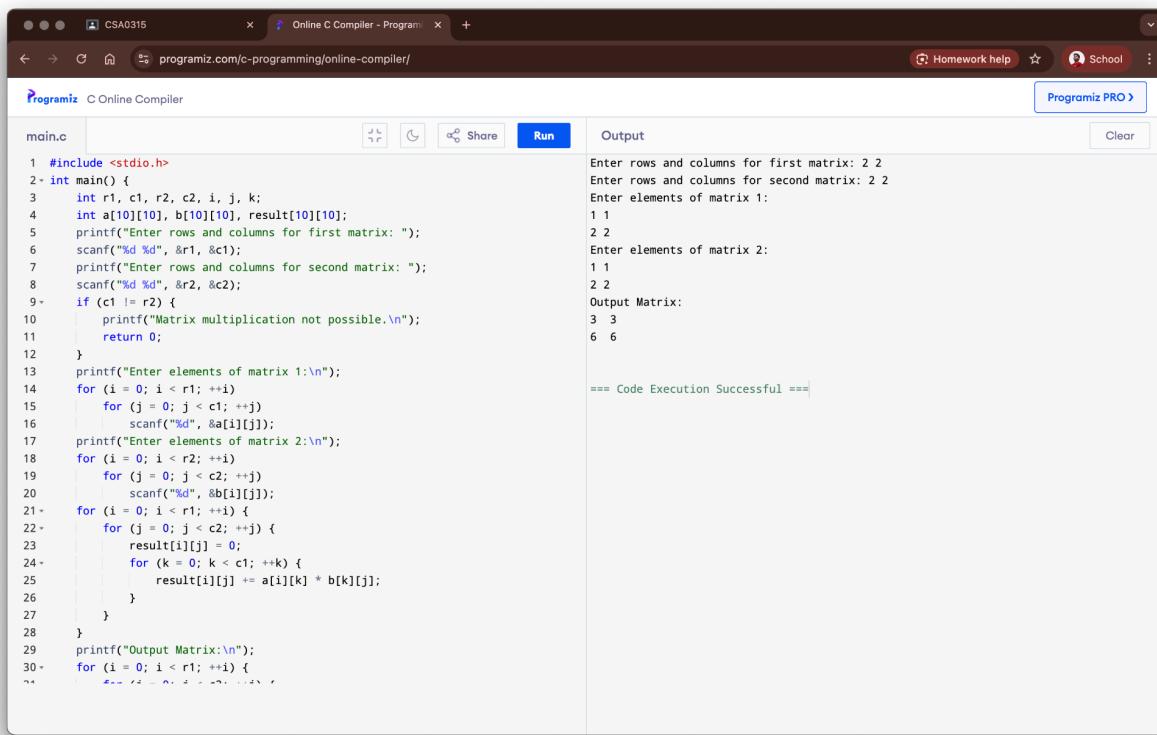
1. Matrix Multiplication

Aim:

To write a C program to perform the multiplication of two matrices.

Algorithm:

1. Start the program.
2. Read the order of the first matrix (r_1, c_1) and the second matrix (r_2, c_2).
3. Check if c_1 is equal to r_2 . If not, multiplication is not possible.
4. Read the elements of both matrices from the user.
5. Multiply the matrices by iterating through rows and columns and store the result in a new matrix.



The screenshot shows a web-based C compiler interface. The code in the editor is:

```
main.c
1 #include <stdio.h>
2 int main() {
3     int r1, c1, r2, c2, i, j, k;
4     int a[10][10], b[10][10], result[10][10];
5     printf("Enter rows and columns for first matrix: ");
6     scanf("%d %d", &r1, &c1);
7     printf("Enter rows and columns for second matrix: ");
8     scanf("%d %d", &r2, &c2);
9     if (c1 != r2) {
10         printf("Matrix multiplication not possible.\n");
11         return 0;
12     }
13     printf("Enter elements of matrix 1:\n");
14     for (i = 0; i < r1; ++i)
15         for (j = 0; j < c1; ++j)
16             scanf("%d", &a[i][j]);
17     printf("Enter elements of matrix 2:\n");
18     for (i = 0; i < r2; ++i)
19         for (j = 0; j < c2; ++j)
20             scanf("%d", &b[i][j]);
21     for (i = 0; i < r1; ++i) {
22         for (j = 0; j < c2; ++j) {
23             result[i][j] = 0;
24             for (k = 0; k < c1; ++k) {
25                 result[i][j] += a[i][k] * b[k][j];
26             }
27         }
28     }
29     printf("Output Matrix:\n");
30     for (i = 0; i < r1; ++i) {
31         for (j = 0; j < c2; ++j)
32             printf("%d ", result[i][j]);
33     }
}
```

The output window shows the interaction with the user:

```
Enter rows and columns for first matrix: 2 2
Enter rows and columns for second matrix: 2 2
Enter elements of matrix 1:
1 1
2 2
Enter elements of matrix 2:
1 1
2 2
Output Matrix:
3 3
6 6

```

At the bottom right, it says "Code Execution Successful".

Result:

The program for matrix multiplication was successfully executed.

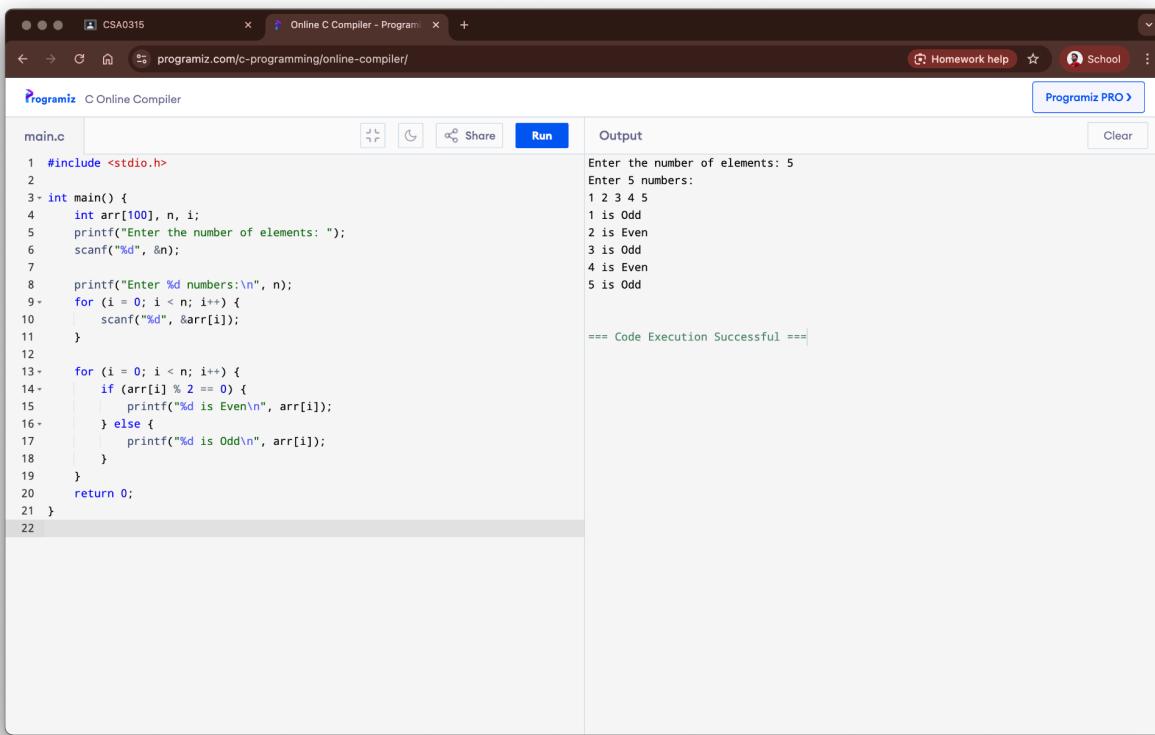
2. Find Odd or Even Number

Aim:

To write a C program to find whether a number is odd or even from a given set of numbers.

Algorithm:

1. Start the program.
2. Read the total number of elements (n).
3. Read n numbers from the user and store them in an array.
4. Iterate through each number in the array.
5. For each number, check if it is divisible by 2. If yes, it's even; otherwise, it's odd.



The screenshot shows a web-based C compiler interface. The code in the editor is:

```
main.c
1 #include <stdio.h>
2
3 int main() {
4     int arr[100], n, i;
5     printf("Enter the number of elements: ");
6     scanf("%d", &n);
7
8     printf("Enter %d numbers:\n", n);
9     for (i = 0; i < n; i++) {
10         scanf("%d", &arr[i]);
11     }
12
13     for (i = 0; i < n; i++) {
14         if (arr[i] % 2 == 0) {
15             printf("%d is Even\n", arr[i]);
16         } else {
17             printf("%d is Odd\n", arr[i]);
18         }
19     }
20     return 0;
21 }
```

The output window shows the following interaction:

```
Enter the number of elements: 5
Enter 5 numbers:
1 2 3 4 5
1 is Odd
2 is Even
3 is Odd
4 is Even
5 is Odd

==== Code Execution Successful ====
```

Result:

The program to find odd or even numbers from a set was successfully executed.

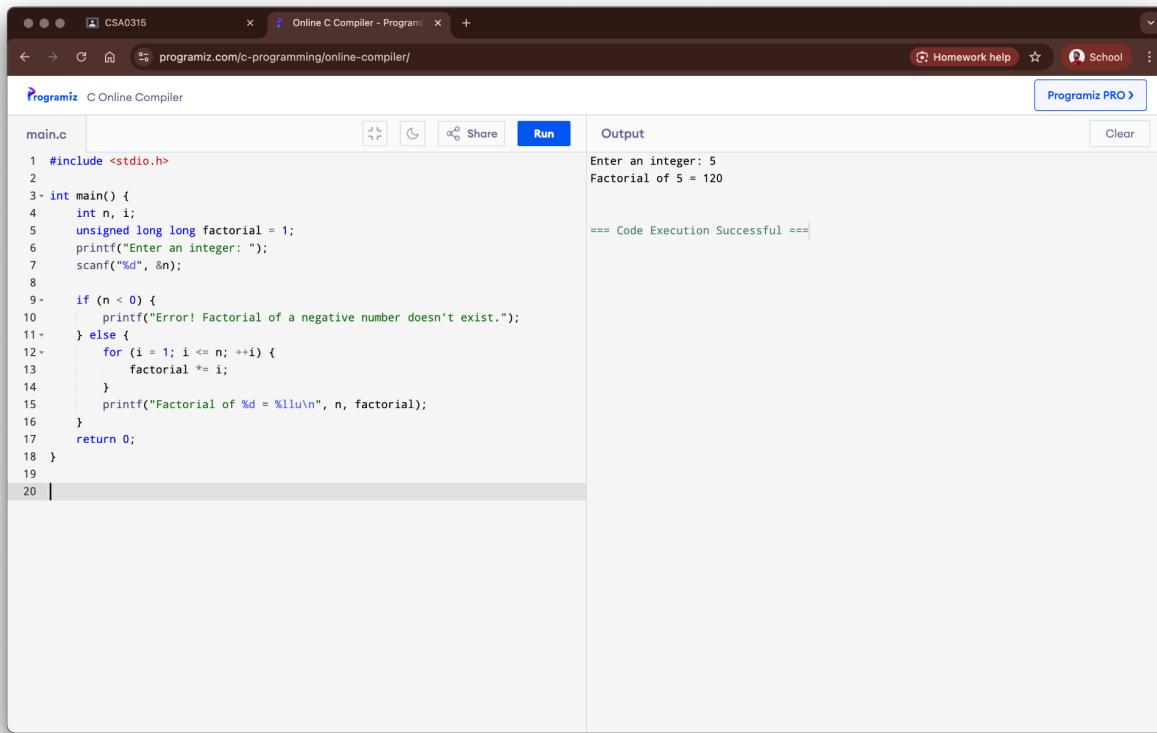
3. Factorial of a Given Number (without Recursion)

Aim:

To write a C program to find the factorial of a given number without using recursion.

Algorithm:

1. Start the program.
2. Read an integer 'n' from the user.
3. Initialize a variable 'factorial' to 1.
4. Use a loop to iterate from 1 to n.
5. In each iteration, multiply 'factorial' by the loop counter.



The screenshot shows a web-based C compiler interface. The code editor contains a file named 'main.c' with the following content:

```
1 #include <stdio.h>
2
3 int main() {
4     int n, i;
5     unsigned long long factorial = 1;
6     printf("Enter an integer: ");
7     scanf("%d", &n);
8
9     if (n < 0) {
10         printf("Error! Factorial of a negative number doesn't exist.");
11     } else {
12         for (i = 1; i <= n; ++i) {
13             factorial *= i;
14         }
15         printf("Factorial of %d = %llu\n", n, factorial);
16     }
17     return 0;
18 }
19
20 }
```

The output window shows the results of running the program. It prompts the user to enter an integer (5), displays the factorial calculation (Factorial of 5 = 120), and concludes with a success message (Code Execution Successful).

Result:

The program to find the factorial iteratively was successfully executed.

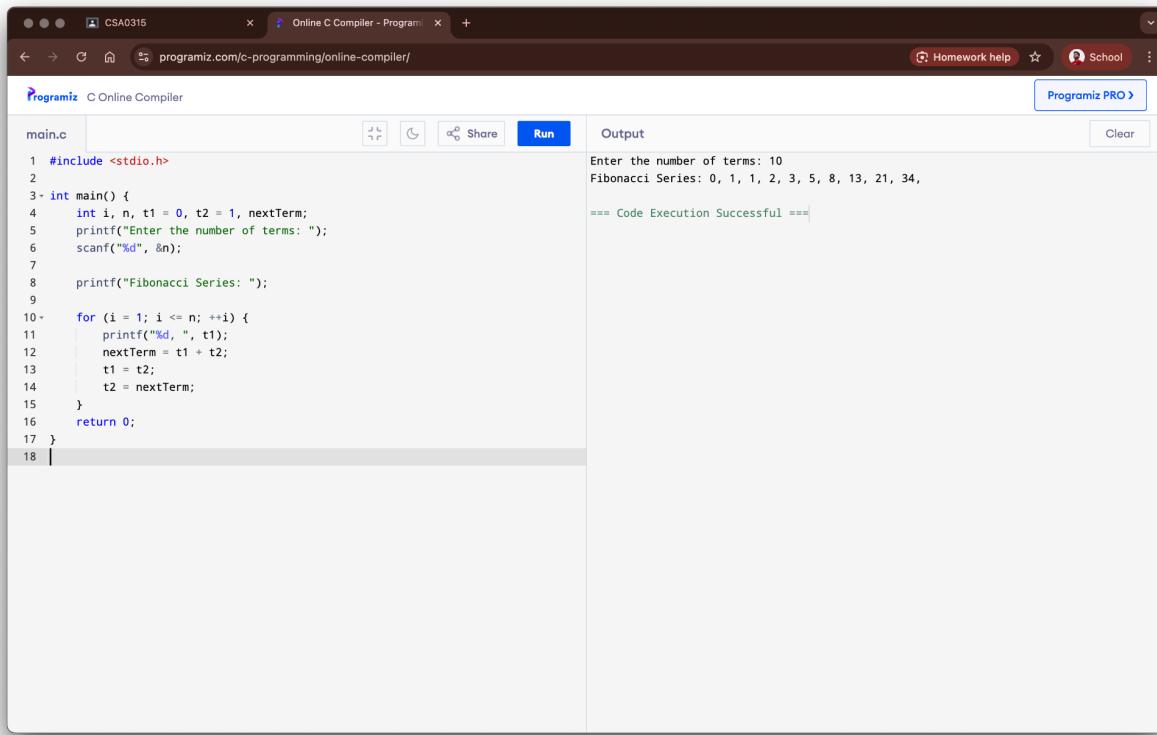
4. Fibonacci Series (without Recursion)

Aim:

To write a C program to generate the Fibonacci series up to a given number of terms without using recursion.

Algorithm:

1. Start the program.
2. Read the number of terms 'n' from the user.
3. Initialize the first two terms, $t_1 = 0$ and $t_2 = 1$.
4. Use a loop to iterate from 1 to n.
5. Print the current term and calculate the next term by summing the previous two.



The screenshot shows a web-based C compiler interface. The code editor contains a file named 'main.c' with the following content:

```
1 #include <stdio.h>
2
3 int main() {
4     int i, n, t1 = 0, t2 = 1, nextTerm;
5     printf("Enter the number of terms: ");
6     scanf("%d", &n);
7
8     printf("Fibonacci Series: ");
9
10    for (i = 1; i <= n; ++i) {
11        printf("%d, ", t1);
12        nextTerm = t1 + t2;
13        t1 = t2;
14        t2 = nextTerm;
15    }
16    return 0;
17 }
18 }
```

The output window shows the results of running the program. It prompts the user to enter the number of terms (10), then displays the Fibonacci series: 0, 1, 2, 3, 5, 8, 13, 21, 34. Below the output, a message indicates successful code execution.

Result:

The program to generate the Fibonacci series iteratively was successfully executed.

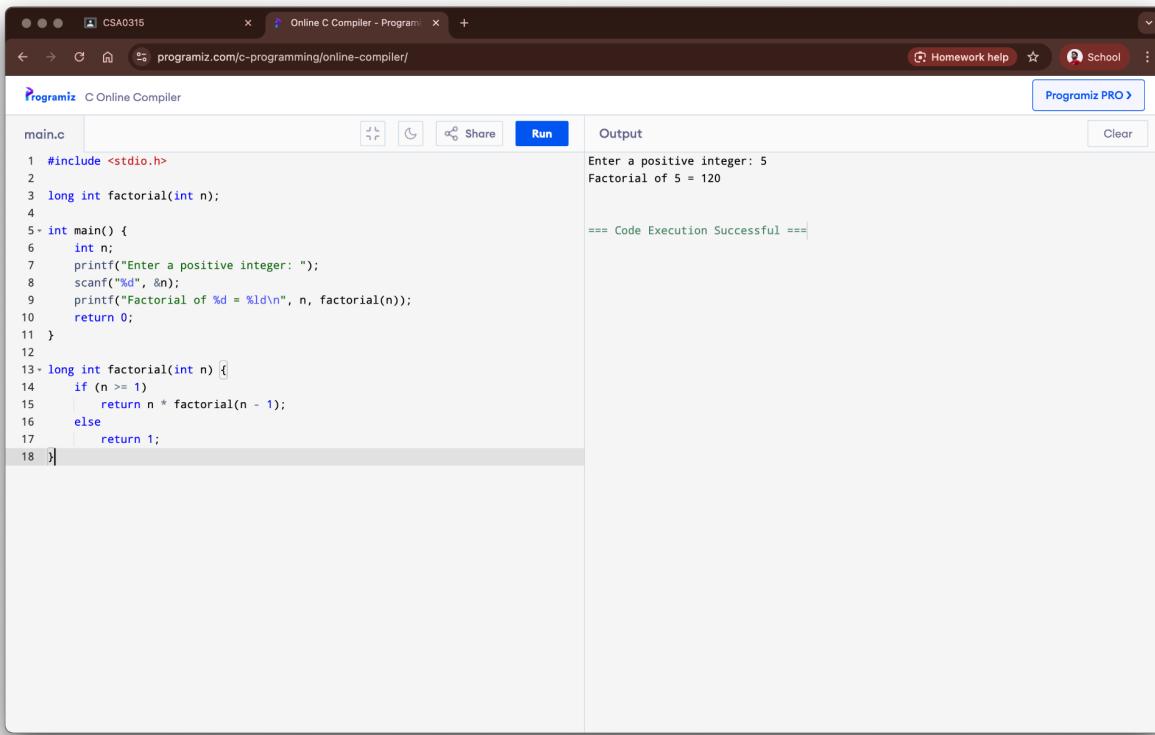
5. Factorial of a Given Number (using Recursion)

Aim:

To write a C program to find the factorial of a given number using recursion.

Algorithm:

1. Start the program.
2. Read an integer 'n' from the user.
3. Define a recursive function factorial(num).
4. The base case for the recursion is when num is 0 or 1, return 1.
5. Otherwise, return num * factorial(num - 1).



The screenshot shows a web-based C compiler interface. The code editor contains a file named 'main.c' with the following content:

```
1 #include <stdio.h>
2
3 long int factorial(int n);
4
5 int main() {
6     int n;
7     printf("Enter a positive integer: ");
8     scanf("%d", &n);
9     printf("Factorial of %d = %ld\n", n, factorial(n));
10    return 0;
11 }
12
13 long int factorial(int n) {
14     if (n >= 1)
15         return n * factorial(n - 1);
16     else
17         return 1;
18 }
```

The output window shows the results of running the program. It prompts the user to enter a positive integer (5), displays the factorial calculation (Factorial of 5 = 120), and concludes with a success message (Code Execution Successful).

Result:

The program to find the factorial recursively was successfully executed.

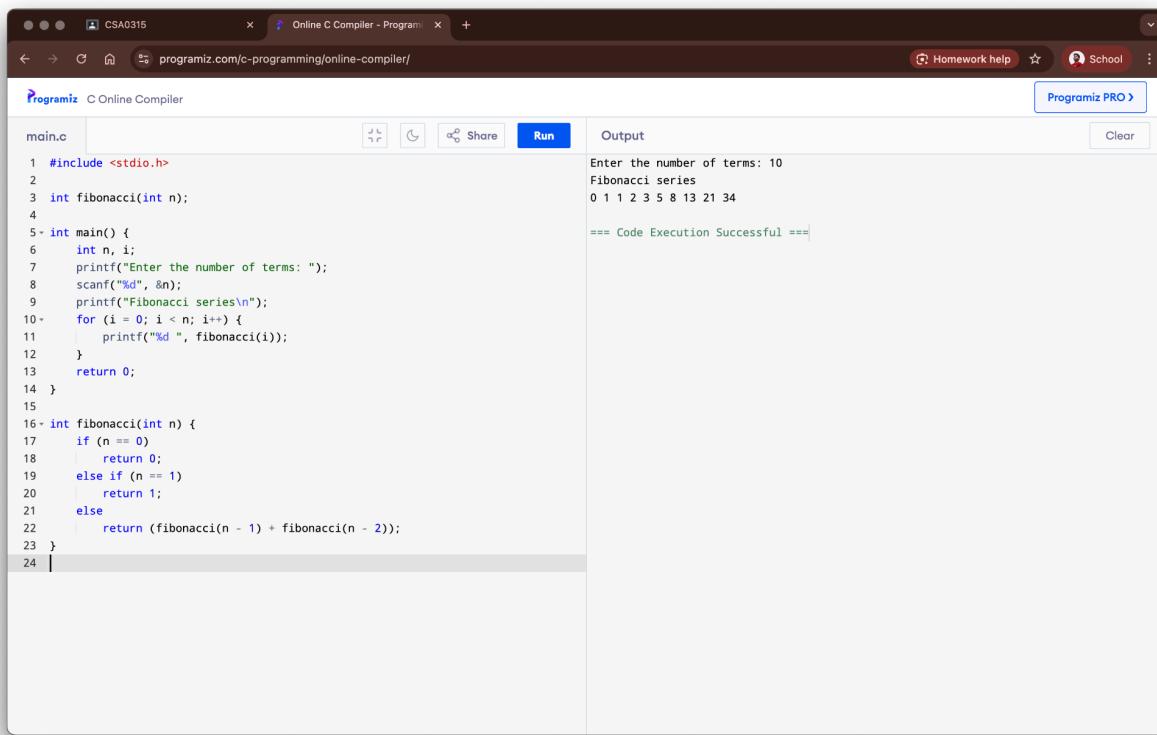
6. Fibonacci Series (using Recursion)

Aim:

To write a C program to generate the Fibonacci series using recursion.

Algorithm:

1. Start the program.
2. Read the number of terms 'n' from the user.
3. Define a recursive function fibonacci(num).
4. The base cases are: if num is 0, return 0; if num is 1, return 1.
5. Otherwise, return fibonacci(num - 1) + fibonacci(num - 2).



The screenshot shows a web-based C compiler interface. The code editor contains a file named 'main.c' with the following content:

```
1 #include <stdio.h>
2
3 int fibonacci(int n);
4
5 int main() {
6     int n, i;
7     printf("Enter the number of terms: ");
8     scanf("%d", &n);
9     printf("Fibonacci series\n");
10    for (i = 0; i < n; i++) {
11        printf("%d ", fibonacci(i));
12    }
13    return 0;
14 }
15
16 int fibonacci(int n) {
17     if (n == 0)
18         return 0;
19     else if (n == 1)
20         return 1;
21     else
22         return (fibonacci(n - 1) + fibonacci(n - 2));
23 }
```

The output window shows the execution results:

```
Enter the number of terms: 10
Fibonacci series
0 1 1 2 3 5 8 13 21 34
== Code Execution Successful ==
```

Result:

The program to generate the Fibonacci series recursively was successfully executed.

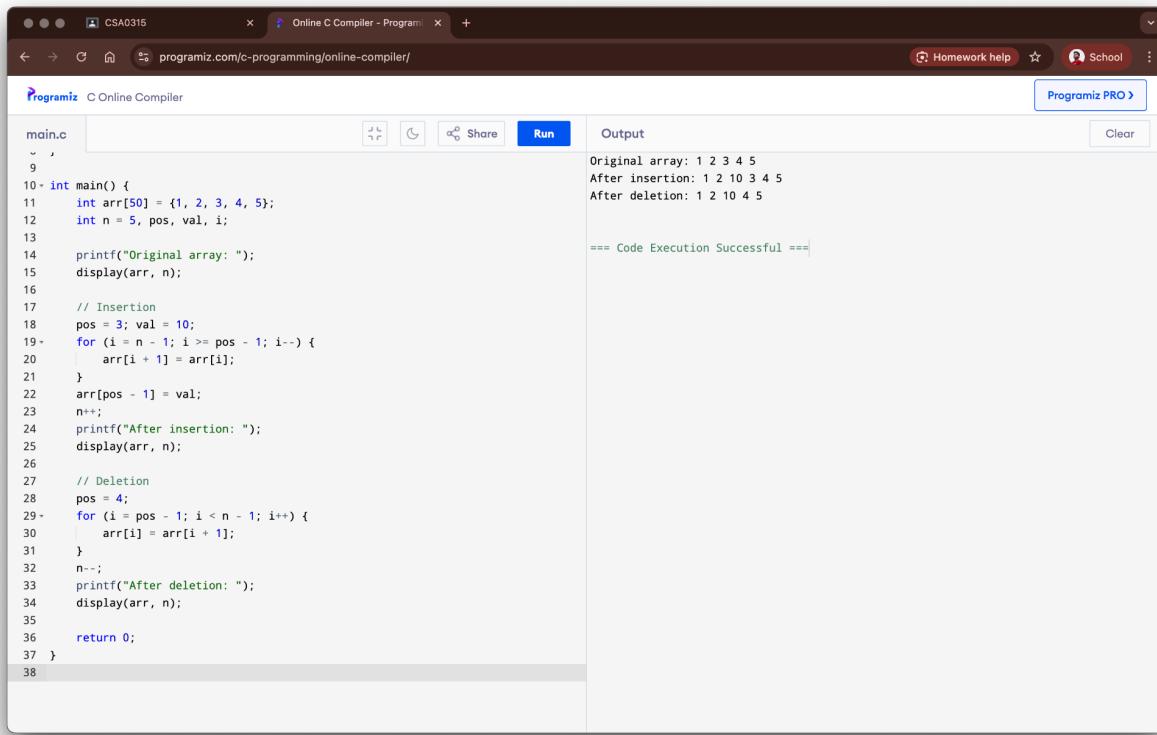
7. Array Operations (Insert, Delete, Display)

Aim:

To write a C program to perform insertion, deletion, and display operations on an array.

Algorithm:

1. Start the program.
2. Create an array and initialize its size.
3. For insertion, shift elements to the right from the given position and insert the new element.
4. For deletion, shift elements to the left from the position of the element to be deleted.
5. For display, iterate through the array and print all its elements.



The screenshot shows a web-based C compiler interface. The code in the editor is as follows:

```
main.c
9
10 int main() {
11     int arr[50] = {1, 2, 3, 4, 5};
12     int n = 5, pos, val, i;
13
14     printf("Original array: ");
15     display(arr, n);
16
17     // Insertion
18     pos = 3; val = 10;
19     for (i = n - 1; i >= pos - 1; i--) {
20         arr[i + 1] = arr[i];
21     }
22     arr[pos - 1] = val;
23     n++;
24     printf("After insertion: ");
25     display(arr, n);
26
27     // Deletion
28     pos = 4;
29     for (i = pos - 1; i < n - 1; i++) {
30         arr[i] = arr[i + 1];
31     }
32     n--;
33     printf("After deletion: ");
34     display(arr, n);
35
36     return 0;
37 }
38
```

The output window shows the execution results:

```
Original array: 1 2 3 4 5
After insertion: 1 2 10 3 4 5
After deletion: 1 2 10 4 5
== Code Execution Successful ==
```

Result:

The program for array operations was successfully executed.

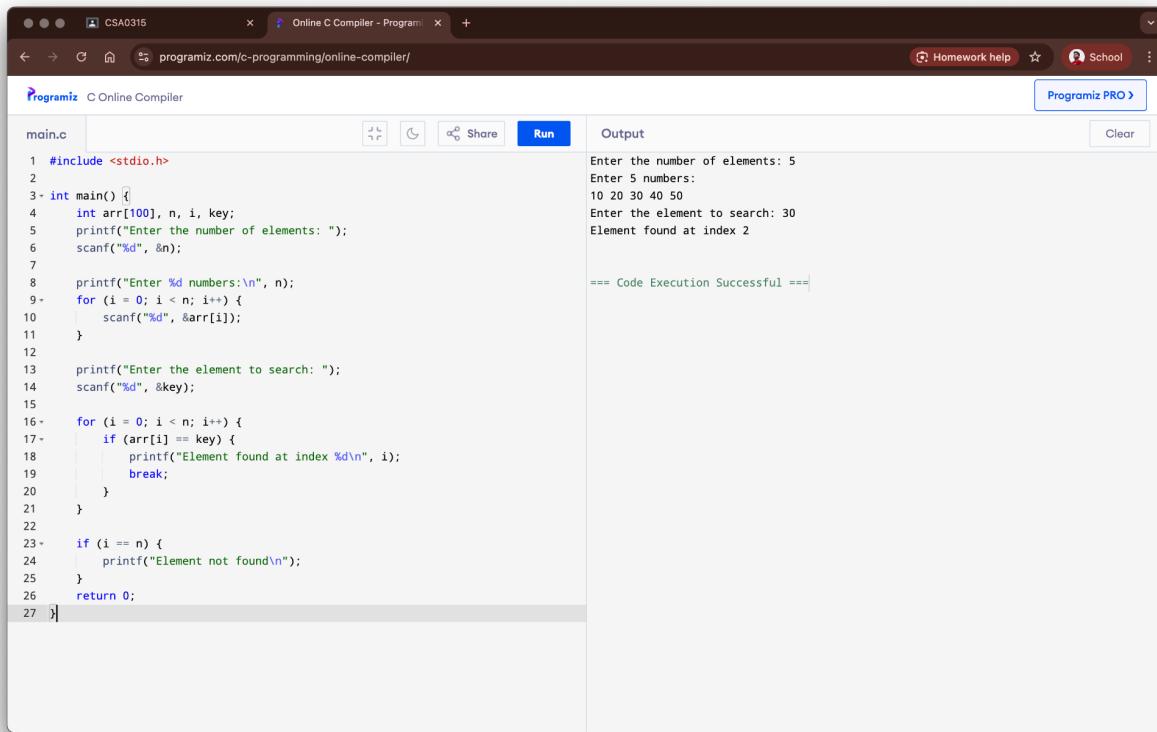
8. Linear Search

Aim:

To write a C program to search for an element in an array using the linear search method.

Algorithm:

1. Start the program.
2. Read the array elements and the search key.
3. Start from the first element of the array.
4. Compare the search key with each element of the array.
5. If a match is found, return the index; otherwise, continue until the end of the array.



The screenshot shows a web-based C compiler interface. The code editor contains a file named 'main.c' with the following content:

```
1 #include <stdio.h>
2
3 int main() {
4     int arr[100], n, i, key;
5     printf("Enter the number of elements: ");
6     scanf("%d", &n);
7
8     printf("Enter %d numbers:\n", n);
9     for (i = 0; i < n; i++) {
10         scanf("%d", &arr[i]);
11     }
12
13     printf("Enter the element to search: ");
14     scanf("%d", &key);
15
16     for (i = 0; i < n; i++) {
17         if (arr[i] == key) {
18             printf("Element found at index %d\n", i);
19             break;
20         }
21     }
22
23     if (i == n) {
24         printf("Element not found\n");
25     }
26     return 0;
27 }
```

The output window shows the following interaction:

```
Enter the number of elements: 5
Enter 5 numbers:
10 20 30 40 50
Enter the element to search: 30
Element found at index 2
== Code Execution Successful ==
```

Result:

The program for linear search was successfully executed.

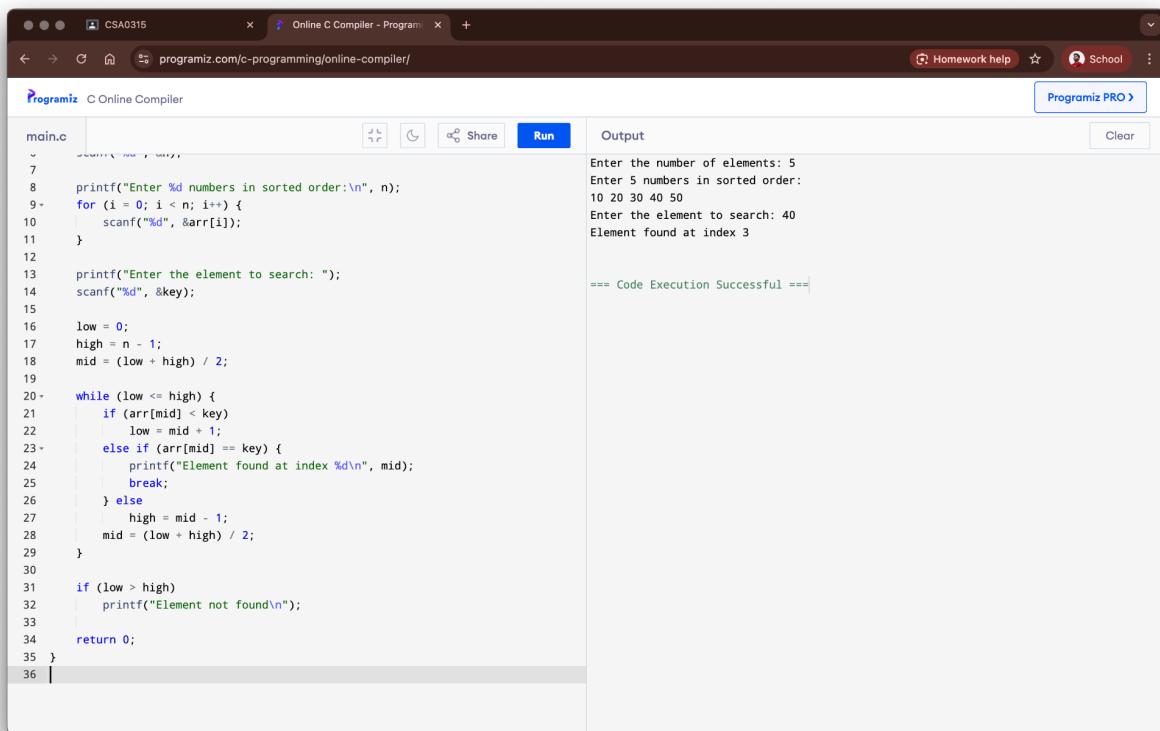
9. Binary Search

Aim:

To write a C program to search for an element in a sorted array using the binary search method.

Algorithm:

1. Start the program.
2. Read the sorted array elements and the search key.
3. Set low = 0 and high = n-1.
4. Find the middle element: mid = (low + high) / 2.
5. If key equals mid, the element is found. If key is less than mid, set high = mid - 1. If key is greater than mid, set low = mid + 1. Repeat until found or low > high.



The screenshot shows a web-based C compiler interface. The code editor contains a C program named main.c. The program prompts the user for the number of elements, reads them into an array, and then performs a binary search for a specified key. The output window shows the execution results: it asks for 5 numbers, receives 10, 20, 30, 40, 50, then asks for a search key of 40, and outputs that the element was found at index 3. A success message follows.

```
main.c
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

Output

```
Enter the number of elements: 5
Enter 5 numbers in sorted order:
10 20 30 40 50
Enter the element to search: 40
Element found at index 3
==== Code Execution Successful ===
```

Result:

The program for binary search was successfully executed.

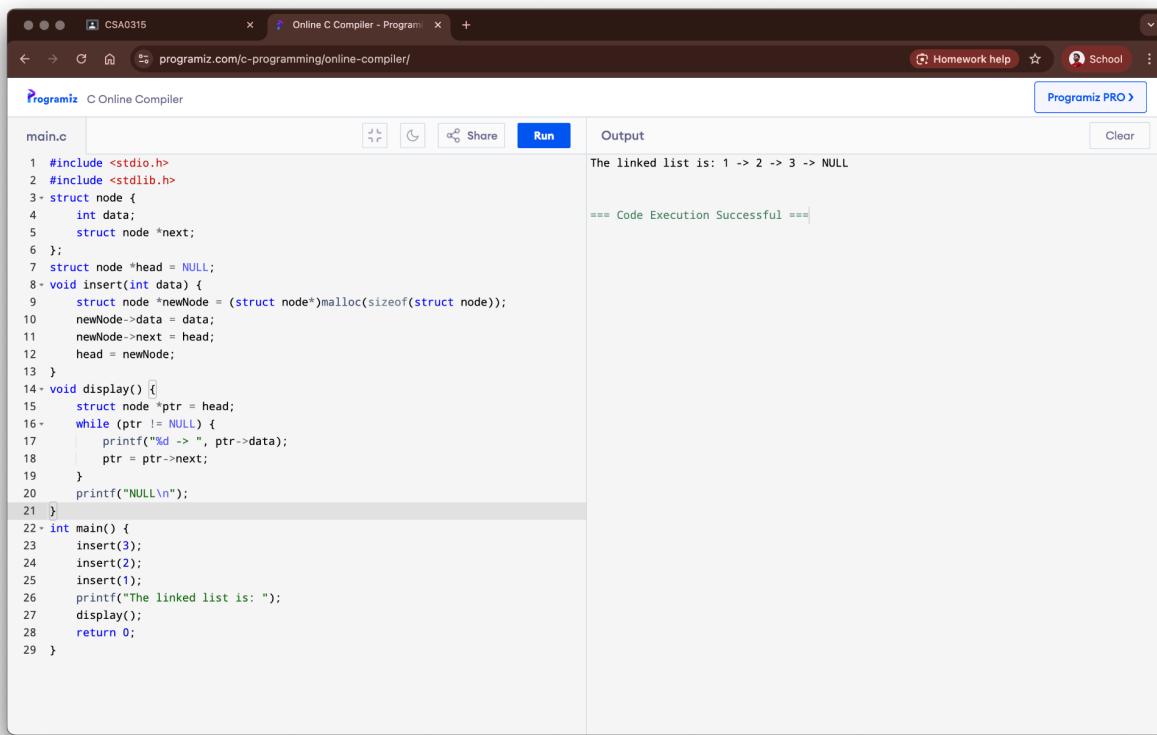
10. Linked List Operations

Aim:

To write a C program to implement basic operations (insert, delete, display) on a singly linked list.

Algorithm:

1. Define a structure for a node with data and a next pointer.
2. For insertion, create a new node, allocate memory, and adjust pointers to add it to the list.
3. For deletion, find the node to be deleted, adjust the pointers of the previous node, and free the memory.
4. For display, traverse the list from the head node and print the data of each node.
5. Implement a menu-driven program to perform these operations.



The screenshot shows a web-based C compiler interface. The code in the editor is as follows:

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct node {
4     int data;
5     struct node *next;
6 };
7 struct node *head = NULL;
8 void insert(int data) {
9     struct node *newNode = (struct node*)malloc(sizeof(struct node));
10    newNode->data = data;
11    newNode->next = head;
12    head = newNode;
13 }
14 void display() {
15     struct node *ptr = head;
16     while (ptr != NULL) {
17         printf("%d -> ", ptr->data);
18         ptr = ptr->next;
19     }
20     printf("NULL\n");
21 }
22 int main() {
23     insert(3);
24     insert(2);
25     insert(1);
26     printf("The linked list is: ");
27     display();
28     return 0;
29 }
```

The output window shows the execution results:

```
The linked list is: 1 -> 2 -> 3 -> NULL
==== Code Execution Successful ====
```

Result:

The program to implement linked list operations was successfully executed.

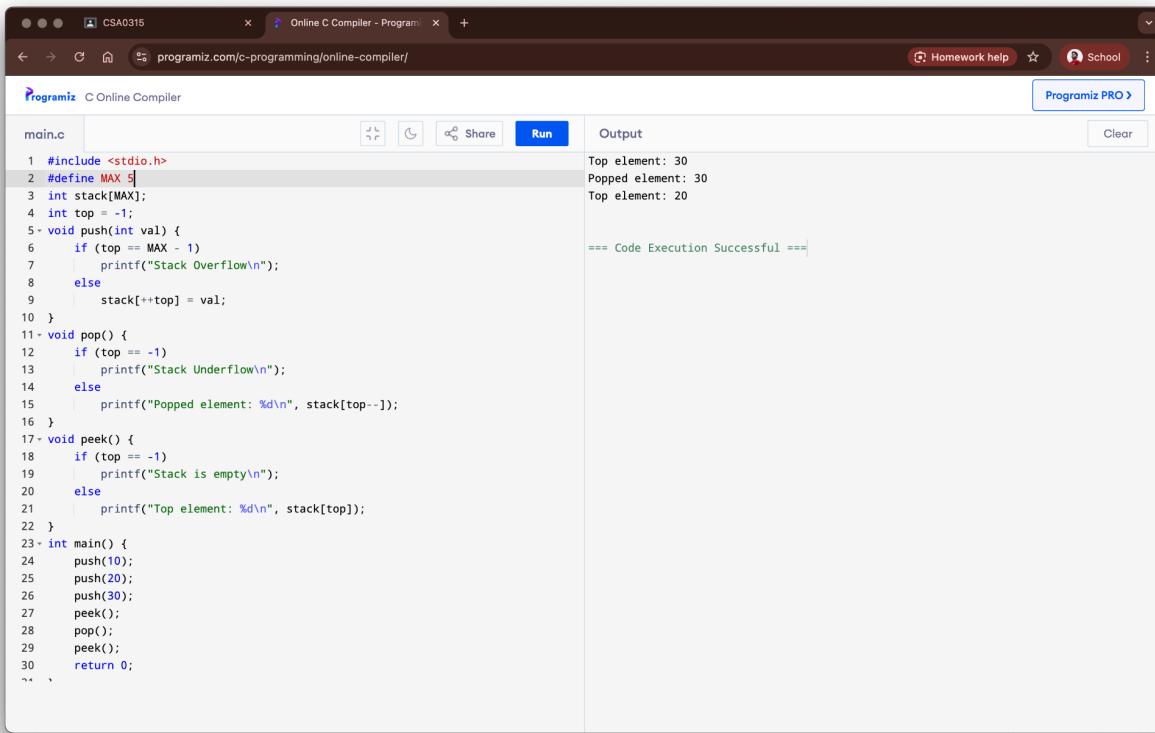
11. Stack Operations (PUSH, POP, PEEK)

Aim:

To write a C program to implement stack operations (PUSH, POP, PEEK) using an array.

Algorithm:

1. Declare a global array for the stack and a variable top initialized to -1.
2. For PUSH, check for overflow. If not, increment top and add the element.
3. For POP, check for underflow. If not, return the element at top and decrement top.
4. For PEEK, check for underflow. If not, return the element at top without decrementing.
5. Implement a menu-driven program to perform these operations.



The screenshot shows a web-based C compiler interface. The code editor contains a file named 'main.c' with the following content:

```
1 #include <stdio.h>
2 #define MAX 5
3 int stack[MAX];
4 int top = -1;
5 void push(int val) {
6     if (top == MAX - 1)
7         printf("Stack Overflow\n");
8     else
9         stack[++top] = val;
10 }
11 void pop() {
12     if (top == -1)
13         printf("Stack Underflow\n");
14     else
15         printf("Popped element: %d\n", stack[top--]);
16 }
17 void peek() {
18     if (top == -1)
19         printf("Stack is empty\n");
20     else
21         printf("Top element: %d\n", stack[top]);
22 }
23 int main() {
24     push(10);
25     push(20);
26     push(30);
27     peek();
28     pop();
29     peek();
30     return 0;
31 }
```

The output window displays the results of running the program:

```
Top element: 30
Popped element: 30
Top element: 20
==== Code Execution Successful ===
```

Result:

The program to implement stack operations was successfully executed.

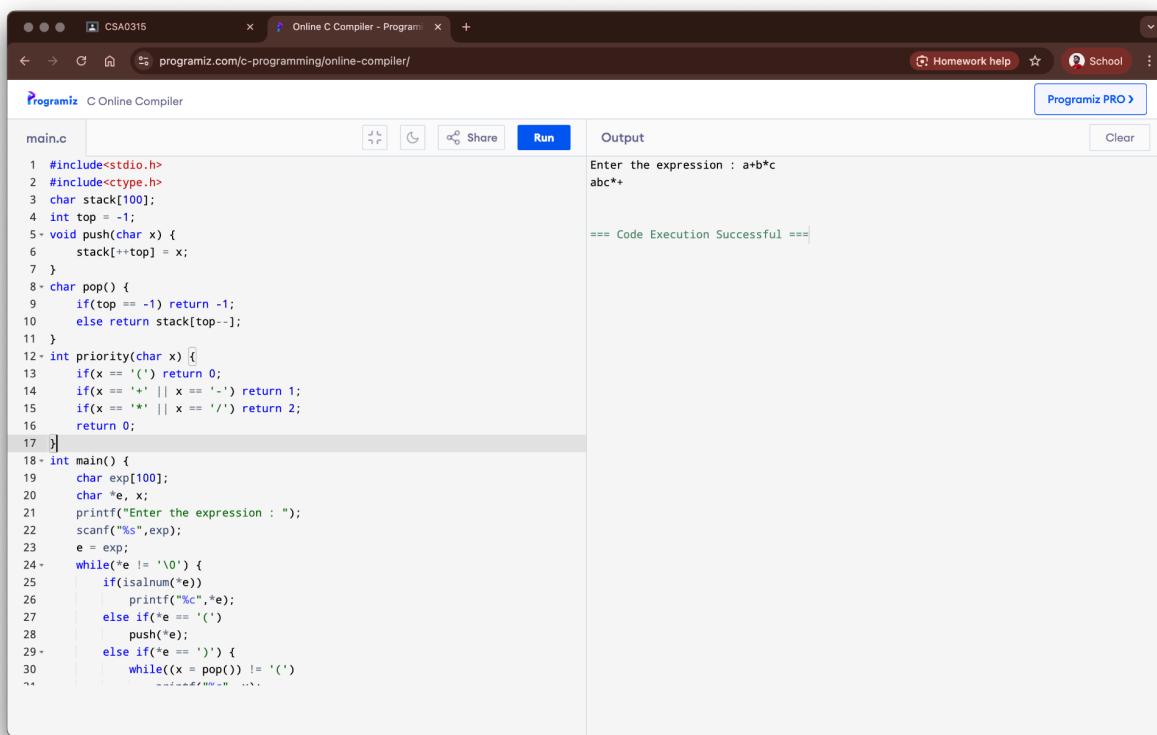
12. Application of Stack (Infix to Postfix)

Aim:

To write a C program to convert an infix expression to a postfix expression.

Algorithm:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. If the scanned character is an operator, and stack is empty or contains a lower precedence operator, push it.
4. If the scanned character is an operator and has lower or equal precedence than the top of the stack, pop the stack and output. Then test the incoming operator against the new top.
5. After scanning all characters, pop and output any remaining operators from the stack.



The screenshot shows a web-based C compiler interface. In the code editor (main.c), the following C code is written:

```
1 #include<stdio.h>
2 #include<ctype.h>
3 char stack[100];
4 int top = -1;
5 void push(char x) {
6     stack[++top] = x;
7 }
8 char pop() {
9     if(top == -1) return -1;
10    else return stack[top--];
11 }
12 int priority(char x) {
13    if(x == '(') return 0;
14    if(x == '+' || x == '-') return 1;
15    if(x == '*' || x == '/') return 2;
16    return 0;
17 }
18 int main() {
19     char exp[100];
20     char *e, x;
21     printf("Enter the expression : ");
22     scanf("%s",exp);
23     e = exp;
24     while(*e != '\0') {
25         if(isalnum(*e))
26             printf("%c",*e);
27         else if(*e == '(')
28             push(*e);
29         else if(*e == ')') {
30             while((x = pop()) != '(')
31                 printf("%c",x);
32         }
33     }
34 }
```

In the output window, the user enters the expression "a+b*c" and the compiler outputs "abc*". Below the output, a message says "Code Execution Successful".

Result:

The program for infix to postfix conversion was successfully executed.

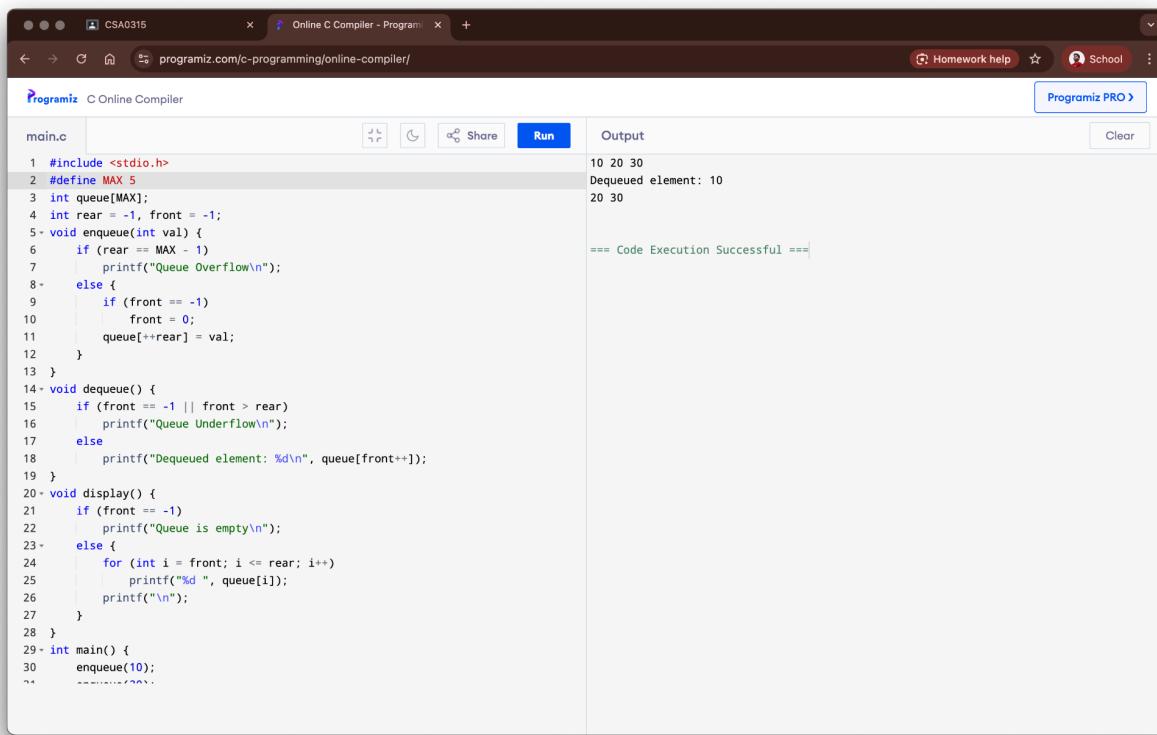
13. Queue Operations (ENQUEUE, DEQUEUE, Display)

Aim:

To write a C program to implement queue operations (ENQUEUE, DEQUEUE, Display) using an array.

Algorithm:

1. Declare a global array for the queue and variables front and rear initialized to -1.
2. For ENQUEUE, check for overflow. If not, increment rear and add the element. Set front to 0 if it's the first element.
3. For DEQUEUE, check for underflow. If not, return the element at front and increment front.
4. For Display, check if the queue is empty. If not, print elements from front to rear.
5. Implement a menu-driven program for these operations.



The screenshot shows a web-based C compiler interface. The code in the editor is as follows:

```
main.c
1 #include <stdio.h>
2 #define MAX 5
3 int queue[MAX];
4 int rear = -1, front = -1;
5 void enqueue(int val) {
6     if (rear == MAX - 1)
7         printf("Queue Overflow\n");
8     else {
9         if (front == -1)
10             front = 0;
11         queue[++rear] = val;
12     }
13 }
14 void dequeue() {
15     if (front == -1 || front > rear)
16         printf("Queue Underflow\n");
17     else
18         printf("Dequeued element: %d\n", queue[front++]);
19 }
20 void display() {
21     if (front == -1)
22         printf("Queue is empty\n");
23     else {
24         for (int i = front; i <= rear; i++)
25             printf("%d ", queue[i]);
26         printf("\n");
27     }
28 }
29 int main() {
30     enqueue(10);
31 }
```

The output window shows the results of running the program:

```
10 20 30
Dequeued element: 10
20 30
==== Code Execution Successful ===
```

Result:

The program to implement queue operations was successfully executed.

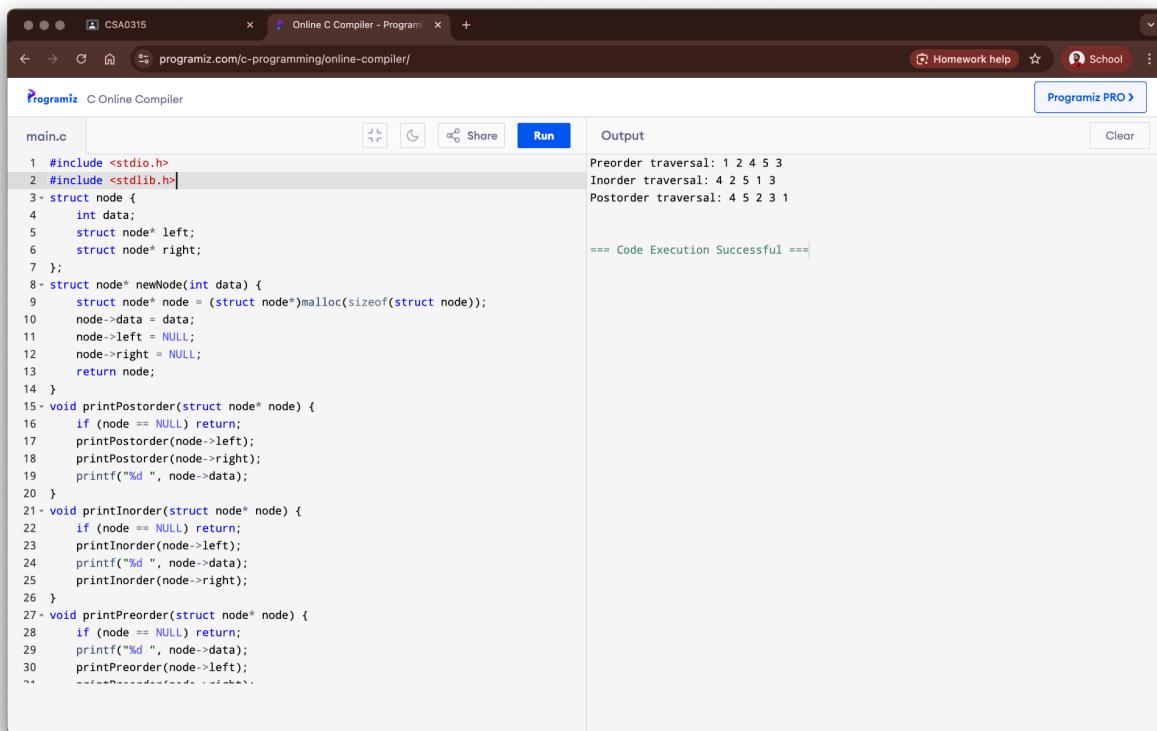
14. Tree Traversals (Inorder, Preorder, Postorder)

Aim:

To write a C program to implement inorder, preorder, and postorder traversals of a binary tree.

Algorithm:

1. Define a structure for a tree node with data, left child, and right child pointers.
2. **Preorder:** Visit the root, traverse the left subtree, then traverse the right subtree (Root-Left-Right).
3. **Inorder:** Traverse the left subtree, visit the root, then traverse the right subtree (Left-Root-Right).
4. **Postorder:** Traverse the left subtree, traverse the right subtree, then visit the root (Left-Right-Root).
5. Create a sample binary tree and call the traversal functions.



The screenshot shows a web-based C compiler interface. In the code editor (main.c), the following C code is written:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct node {
4     int data;
5     struct node* left;
6     struct node* right;
7 };
8 struct node* newNode(int data) {
9     struct node* node = (struct node*)malloc(sizeof(struct node));
10    node->data = data;
11    node->left = NULL;
12    node->right = NULL;
13    return node;
14 }
15 void printPostorder(struct node* node) {
16     if (node == NULL) return;
17     printPostorder(node->left);
18     printPostorder(node->right);
19     printf("%d ", node->data);
20 }
21 void printInorder(struct node* node) {
22     if (node == NULL) return;
23     printInorder(node->left);
24     printf("%d ", node->data);
25     printInorder(node->right);
26 }
27 void printPreorder(struct node* node) {
28     if (node == NULL) return;
29     printf("%d ", node->data);
30     printPreorder(node->left);
31 }
```

The output window displays the results of the code execution:

```
Preorder traversal: 1 2 4 5 3
Inorder traversal: 4 2 5 1 3
Postorder traversal: 4 5 2 3 1
*** Code Execution Successful ***
```

Result:

The program for tree traversals was successfully executed.

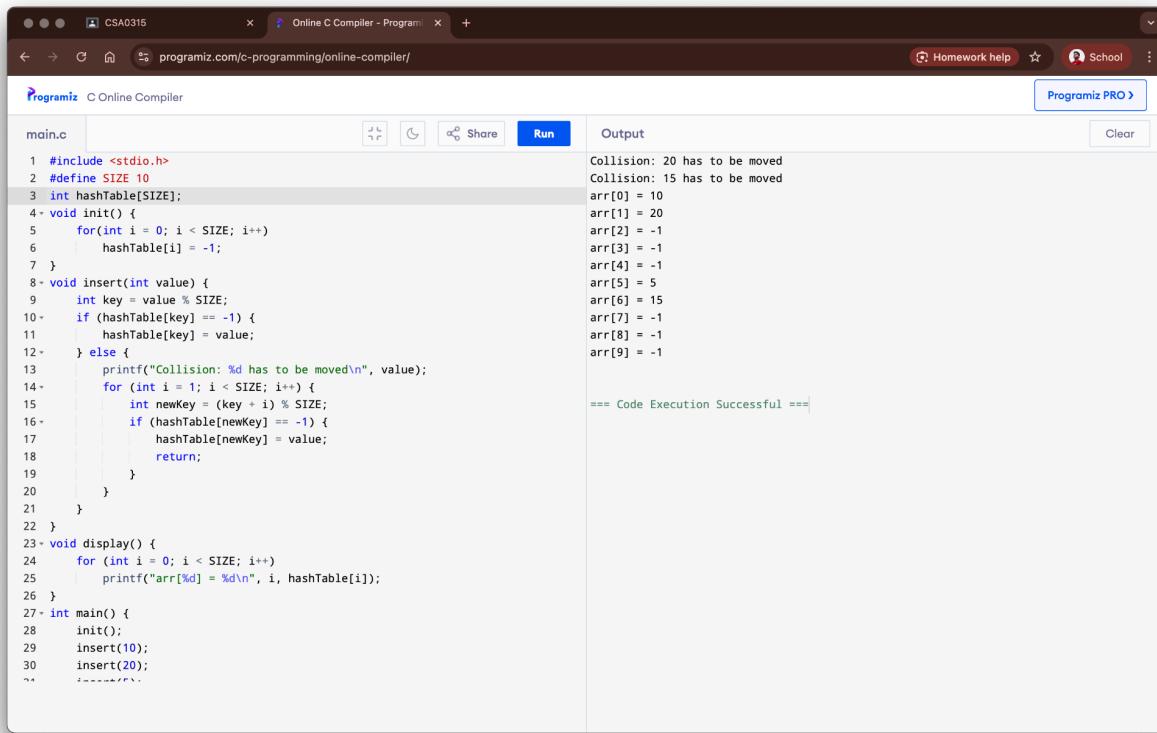
15. Hashing using Linear Probing

Aim:

To write a C program to implement hashing with the linear probing method for collision resolution.

Algorithm:

1. Create a hash table (an array) and initialize all its slots to a sentinel value (e.g., -1).
2. Define a hash function, $h(key) = key \% \text{size}$.
3. To insert a key, compute the hash index. If the slot is empty, place the key.
4. If the slot is occupied (collision), probe subsequent slots linearly ($(\text{index} + 1) \% \text{size}$) until an empty slot is found.
5. To search for a key, compute the hash index and probe linearly until the key is found or an empty slot is encountered.



The screenshot shows a web-based C compiler interface. The code in the editor is as follows:

```
main.c
1 #include <stdio.h>
2 #define SIZE 10
3 int hashTable[SIZE];
4 void init() {
5     for(int i = 0; i < SIZE; i++)
6         hashTable[i] = -1;
7 }
8 void insert(int value) {
9     int key = value % SIZE;
10    if (hashTable[key] == -1) {
11        hashTable[key] = value;
12    } else {
13        printf("Collision: %d has to be moved\n", value);
14        for (int i = 1; i < SIZE; i++) {
15            int newKey = (key + i) % SIZE;
16            if (hashTable[newKey] == -1) {
17                hashTable[newKey] = value;
18                return;
19            }
20        }
21    }
22 }
23 void display() {
24     for (int i = 0; i < SIZE; i++)
25         printf("arr[%d] = %d\n", i, hashTable[i]);
26 }
27 int main() {
28     init();
29     insert(10);
30     insert(20);
31 }
```

The output window shows the execution results:

```
Collision: 20 has to be moved
Collision: 15 has to be moved
arr[0] = 10
arr[1] = 20
arr[2] = -1
arr[3] = -1
arr[4] = -1
arr[5] = 5
arr[6] = 15
arr[7] = -1
arr[8] = -1
arr[9] = -1
==== Code Execution Successful ===
```

Result:

The program for hashing using linear probing was successfully executed.

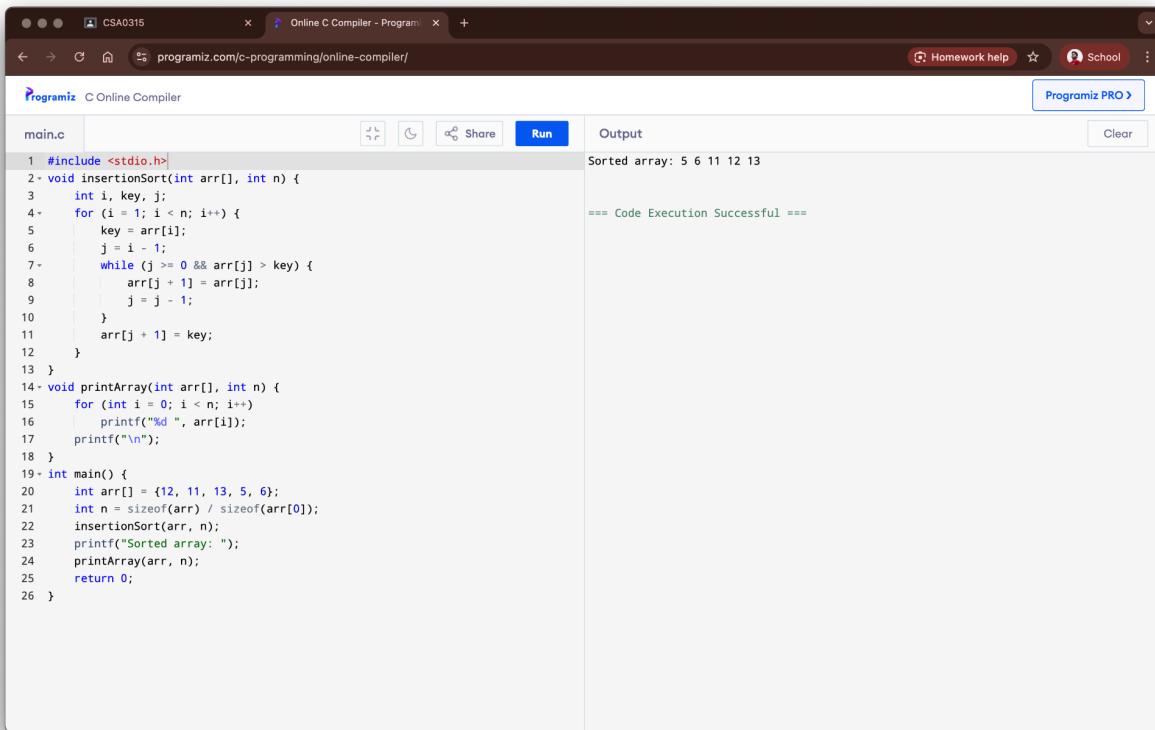
16. Insertion Sort

Aim:

To write a C program to sort an array of numbers using the insertion sort algorithm.

Algorithm:

1. Start from the second element of the array.
2. Compare the current element (key) with its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before.
4. Move the greater elements one position up to make space for the swapped element.
5. Repeat this process for all elements in the array.



The screenshot shows a web-based C compiler interface. The code in the editor is:

```
main.c
1 #include <stdio.h>
2 void insertionSort(int arr[], int n) {
3     int i, key, j;
4     for (i = 1; i < n; i++) {
5         key = arr[i];
6         j = i - 1;
7         while (j >= 0 && arr[j] > key) {
8             arr[j + 1] = arr[j];
9             j = j - 1;
10        }
11        arr[j + 1] = key;
12    }
13 }
14 void printArray(int arr[], int n) {
15     for (int i = 0; i < n; i++)
16         printf("%d ", arr[i]);
17     printf("\n");
18 }
19 int main() {
20     int arr[] = {12, 11, 13, 5, 6};
21     int n = sizeof(arr) / sizeof(arr[0]);
22     insertionSort(arr, n);
23     printf("Sorted array: ");
24     printArray(arr, n);
25     return 0;
26 }
```

The output window shows the sorted array and a success message:

Sorted array: 5 6 11 12 13
==== Code Execution Successful ===

Result:

The program for insertion sort was successfully executed.

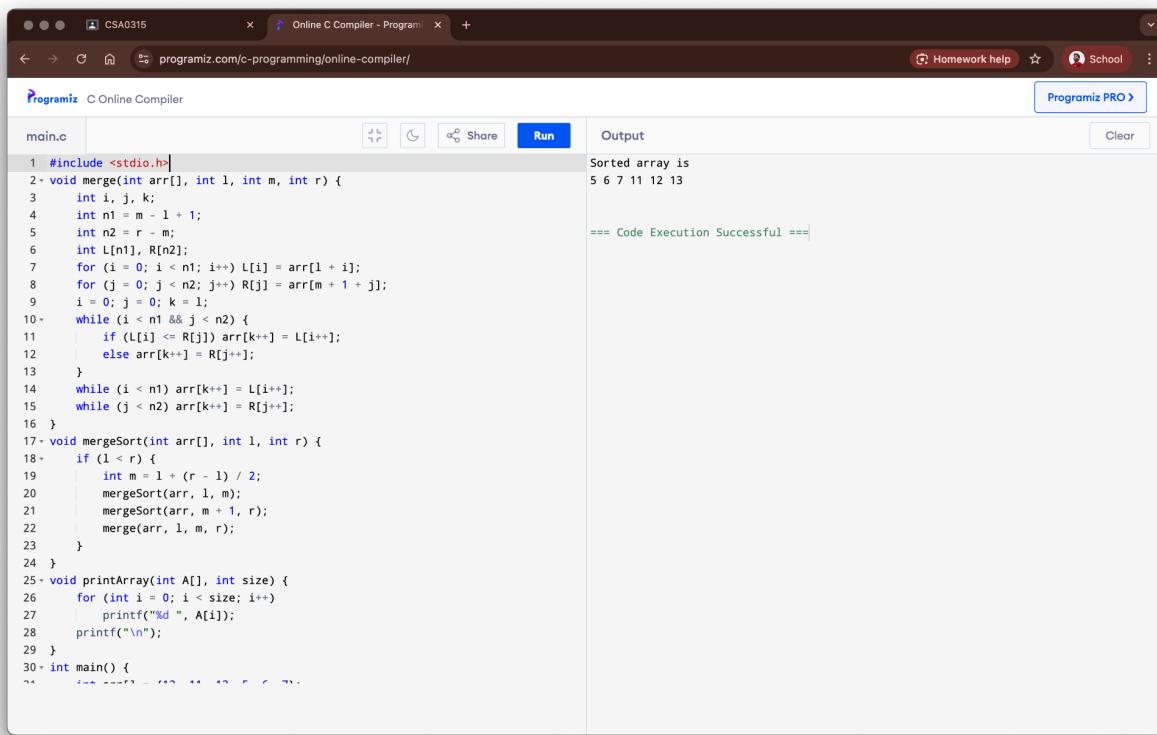
17. Merge Sort

Aim:

To write a C program to sort an array of numbers using the merge sort algorithm.

Algorithm:

1. Divide the unsorted list into n sublists, each containing one element.
2. Repeatedly merge sublists to produce new sorted sublists.
3. The merge step involves comparing elements from two sublists and placing the smaller one into the merged list.
4. Continue merging until there is only one sublist remaining.
5. This will be the sorted list.



The screenshot shows a web-based C compiler interface. The code in the editor is as follows:

```
main.c
1 #include <stdio.h>
2 void merge(int arr[], int l, int m, int r) {
3     int i, j, k;
4     int n1 = m - l + 1;
5     int n2 = r - m;
6     int L[n1], R[n2];
7     for (i = 0; i < n1; i++) L[i] = arr[l + i];
8     for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
9     i = 0; j = 0; k = l;
10    while (i < n1 && j < n2) {
11        if (L[i] <= R[j]) arr[k++] = L[i++];
12        else arr[k++] = R[j++];
13    }
14    while (i < n1) arr[k++] = L[i++];
15    while (j < n2) arr[k++] = R[j++];
16 }
17 void mergeSort(int arr[], int l, int r) {
18    if (l < r) {
19        int m = l + (r - 1) / 2;
20        mergeSort(arr, l, m);
21        mergeSort(arr, m + 1, r);
22        merge(arr, l, m, r);
23    }
24 }
25 void printArray(int A[], int size) {
26     for (int i = 0; i < size; i++)
27         printf("%d ", A[i]);
28     printf("\n");
29 }
30 int main() {
31     // Your code here
32 }
```

The output window shows the sorted array and a success message:

```
Sorted array is
5 6 7 11 12 13
==== Code Execution Successful ===
```

Result:

The program for merge sort was successfully executed.

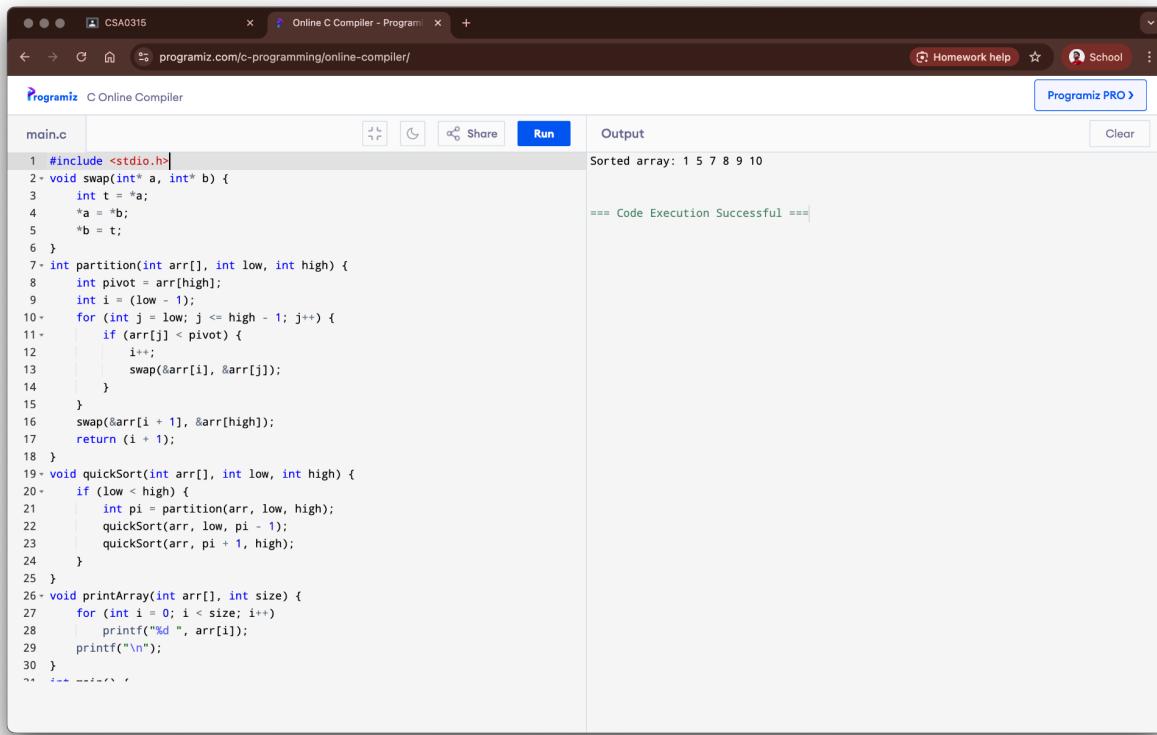
18. Quick Sort

Aim:

To write a C program to sort an array of numbers using the quick sort algorithm.

Algorithm:

1. Pick an element as a pivot (e.g., the last element).
2. Partition the array around the pivot by placing all smaller elements before it and all greater elements after it.
3. The pivot is now in its final sorted position.
4. Recursively apply the above steps to the sub-array of elements with smaller values.
5. Recursively apply the steps to the sub-array of elements with greater values.



The screenshot shows a web-based C compiler interface. In the code editor (main.c), there is a C program for quick sort. The output window shows the sorted array [1, 5, 7, 8, 9, 10] and a success message.

```
main.c
1 #include <stdio.h>
2 void swap(int* a, int* b) {
3     int t = *a;
4     *a = *b;
5     *b = t;
6 }
7 int partition(int arr[], int low, int high) {
8     int pivot = arr[high];
9     int i = (low - 1);
10    for (int j = low; j <= high - 1; j++) {
11        if (arr[j] < pivot) {
12            i++;
13            swap(&arr[i], &arr[j]);
14        }
15    }
16    swap(&arr[i + 1], &arr[high]);
17    return (i + 1);
18 }
19 void quickSort(int arr[], int low, int high) {
20     if (low < high) {
21         int pi = partition(arr, low, high);
22         quickSort(arr, low, pi - 1);
23         quickSort(arr, pi + 1, high);
24     }
25 }
26 void printArray(int arr[], int size) {
27     for (int i = 0; i < size; i++)
28         printf("%d ", arr[i]);
29     printf("\n");
30 }
```

Output:
Sorted array: 1 5 7 8 9 10
==== Code Execution Successful ===

Result:

The program for quick sort was successfully executed.

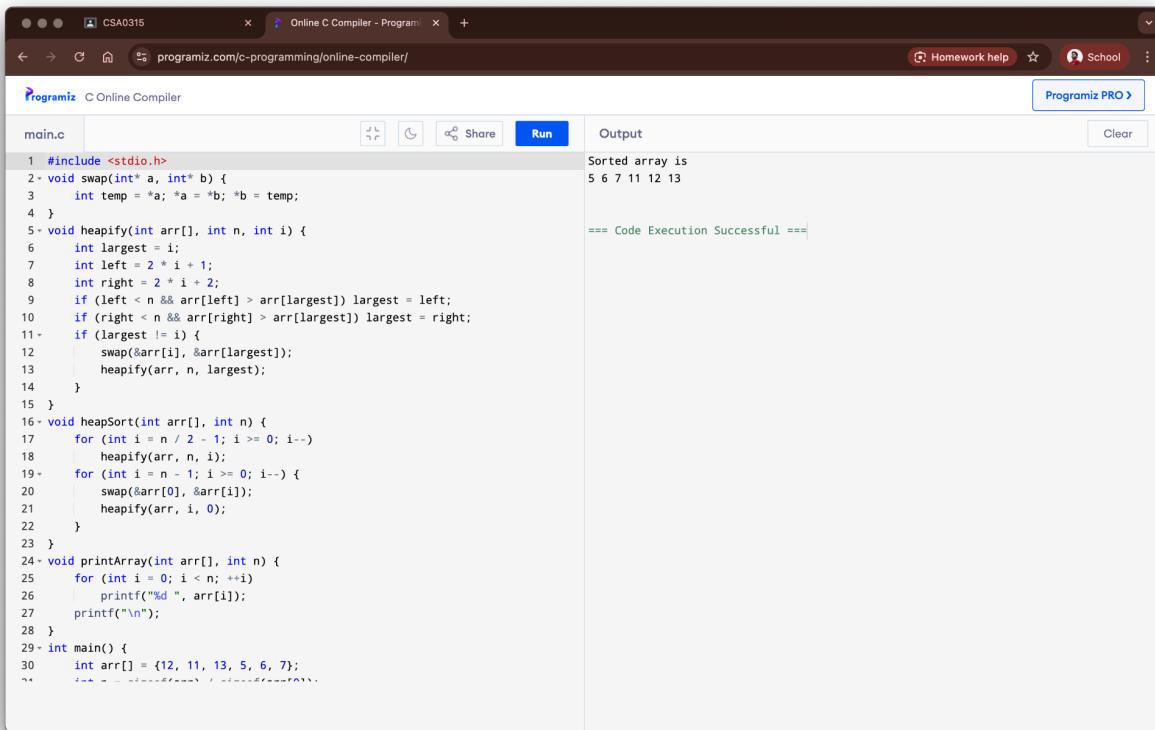
19. Heap Sort

Aim:

To write a C program to sort an array of numbers using the heap sort algorithm.

Algorithm:

1. Build a max heap from the input data.
2. The largest item is stored at the root of the heap.
3. Replace the root with the last item of the heap and reduce the size of the heap by one.
4. Heapify the root of the tree to maintain the heap property.
5. Repeat steps 3 and 4 while the size of the heap is greater than 1.



The screenshot shows a web-based C compiler interface. The code in the editor is a C program for heap sort, which includes functions for swapping elements, heapifying a subtree, performing a full heap sort, and printing the array. The output window shows the sorted array [5, 6, 7, 11, 12, 13] and a success message.

```
main.c
1 #include <stdio.h>
2 void swap(int* a, int* b) {
3     int temp = *a; *a = *b; *b = temp;
4 }
5 void heapify(int arr[], int n, int i) {
6     int largest = i;
7     int left = 2 * i + 1;
8     int right = 2 * i + 2;
9     if (left < n && arr[left] > arr[largest]) largest = left;
10    if (right < n && arr[right] > arr[largest]) largest = right;
11    if (largest != i) {
12        swap(&arr[i], &arr[largest]);
13        heapify(arr, n, largest);
14    }
15 }
16 void heapSort(int arr[], int n) {
17     for (int i = n / 2 - 1; i >= 0; i--) {
18         heapify(arr, n, i);
19     }
20     for (int i = n - 1; i >= 0; i--) {
21         swap(&arr[0], &arr[i]);
22         heapify(arr, i, 0);
23     }
24 }
25 void printArray(int arr[], int n) {
26     for (int i = 0; i < n; ++i)
27         printf("%d ", arr[i]);
28     printf("\n");
29 }
30 int main() {
31     int arr[] = {12, 11, 13, 5, 6, 7};
32 }
```

Sorted array is
5 6 7 11 12 13
== Code Execution Successful ==

Result:

The program for heap sort was successfully executed.

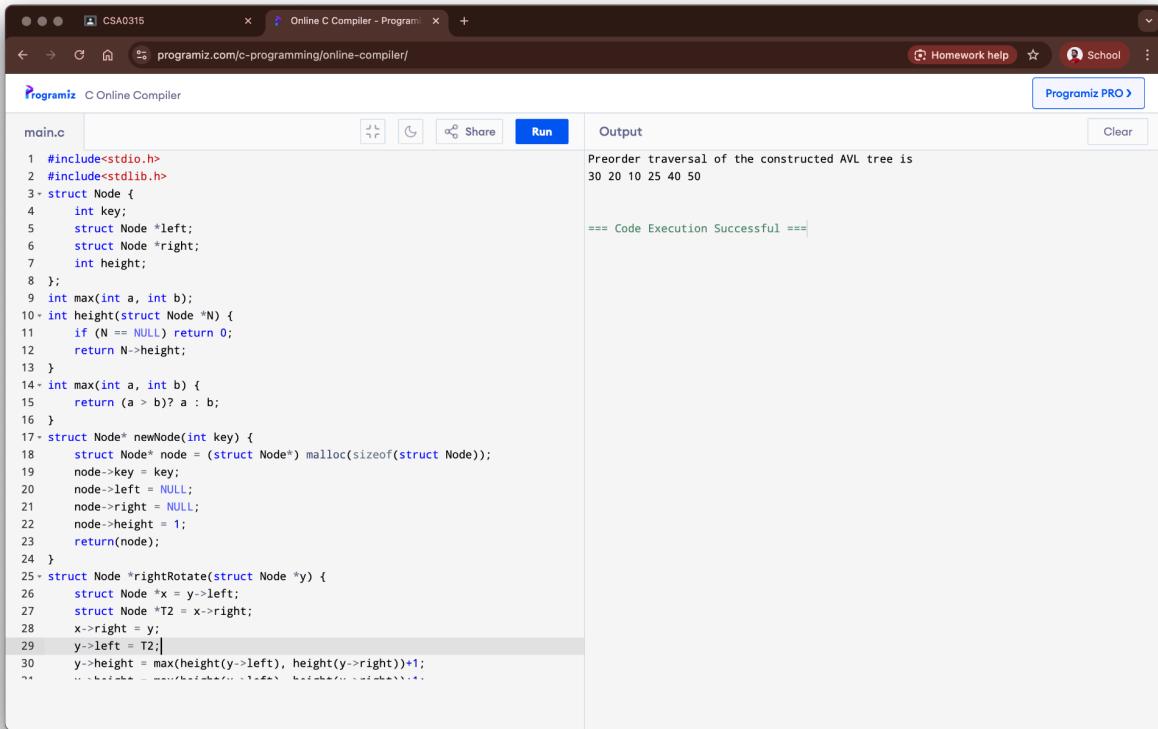
20. AVL Tree Operations

Aim:

To write a C program to perform insertion, deletion, and searching operations on an AVL tree.

Algorithm:

1. Define a node structure with data, height, and left/right child pointers.
2. **Insertion:** Perform a standard BST insert. Then, trace back to the root, updating heights and performing rotations (LL, RR, LR, RL) to balance the tree if the balance factor is > 1 or < -1 .
3. **Deletion:** Perform a standard BST delete. Then, trace back to the root, updating heights and performing rotations to rebalance the tree.
4. **Search:** Perform a standard BST search.
5. The balance factor of a node is the height of its left subtree minus the height of its right subtree.



The screenshot shows a web-based C compiler interface. The code editor contains a file named 'main.c' with the following content:

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 struct Node {
4     int key;
5     struct Node *left;
6     struct Node *right;
7     int height;
8 };
9 int max(int a, int b);
10 int height(struct Node *N) {
11     if (N == NULL) return 0;
12     return N->height;
13 }
14 int max(int a, int b) {
15     return (a > b)? a : b;
16 }
17 struct Node* newNode(int key) {
18     struct Node* node = (struct Node*) malloc(sizeof(struct Node));
19     node->key = key;
20     node->left = NULL;
21     node->right = NULL;
22     node->height = 1;
23     return(node);
24 }
25 struct Node *rightRotate(struct Node *y) {
26     struct Node *x = y->left;
27     struct Node *T2 = x->right;
28     x->right = y;
29     y->left = T2;
30     y->height = max(height(y->left), height(y->right))+1;
31 }
```

The output window displays the results of the program execution:

```
Preorder traversal of the constructed AVL tree is
30 20 10 25 40 50
==== Code Execution Successful ===
```

Result:

The program for AVL tree operations was successfully executed.

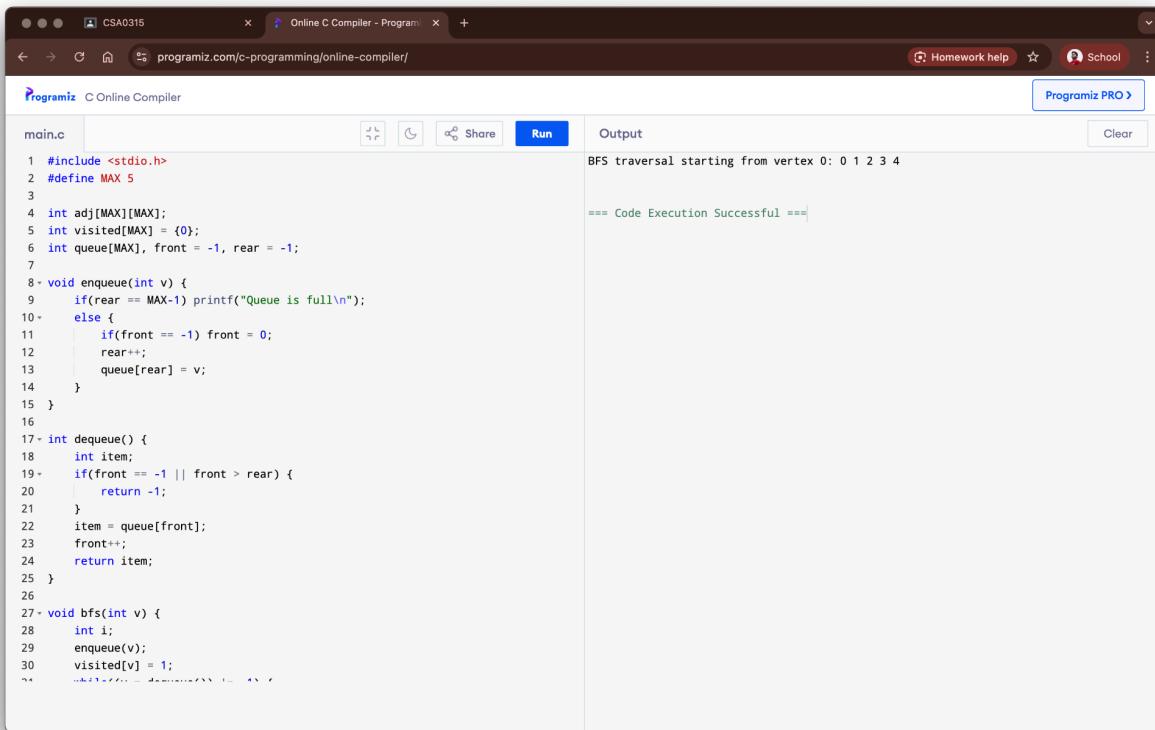
21. Graph Traversal using Breadth First Search (BFS)

Aim:

To write a C program to traverse a graph using the Breadth First Search (BFS) algorithm.

Algorithm:

1. Start at a source vertex and add it to a queue. Mark it as visited.
2. While the queue is not empty, dequeue a vertex.
3. Visit all its unvisited adjacent vertices.
4. Mark them as visited and enqueue them.
5. Repeat until the queue is empty.



The screenshot shows a web-based C compiler interface. In the code editor (main.c), there is a C program for Breadth-First Search (BFS). The output window shows the execution results: "BFS traversal starting from vertex 0: 0 1 2 3 4" followed by "==== Code Execution Successful ===".

```
main.c
1 #include <stdio.h>
2 #define MAX 5
3
4 int adj[MAX][MAX];
5 int visited[MAX] = {0};
6 int queue[MAX], front = -1, rear = -1;
7
8 void enqueue(int v) {
9     if(rear == MAX-1) printf("Queue is full\n");
10    else {
11        if(front == -1) front = 0;
12        rear++;
13        queue[rear] = v;
14    }
15 }
16
17 int dequeue() {
18     int item;
19     if(front == -1 || front > rear) {
20         return -1;
21     }
22     item = queue[front];
23     front++;
24     return item;
25 }
26
27 void bfs(int v) {
28     int i;
29     enqueue(v);
30     visited[v] = 1;
31 }
```

Result:

The program for BFS graph traversal was successfully executed.

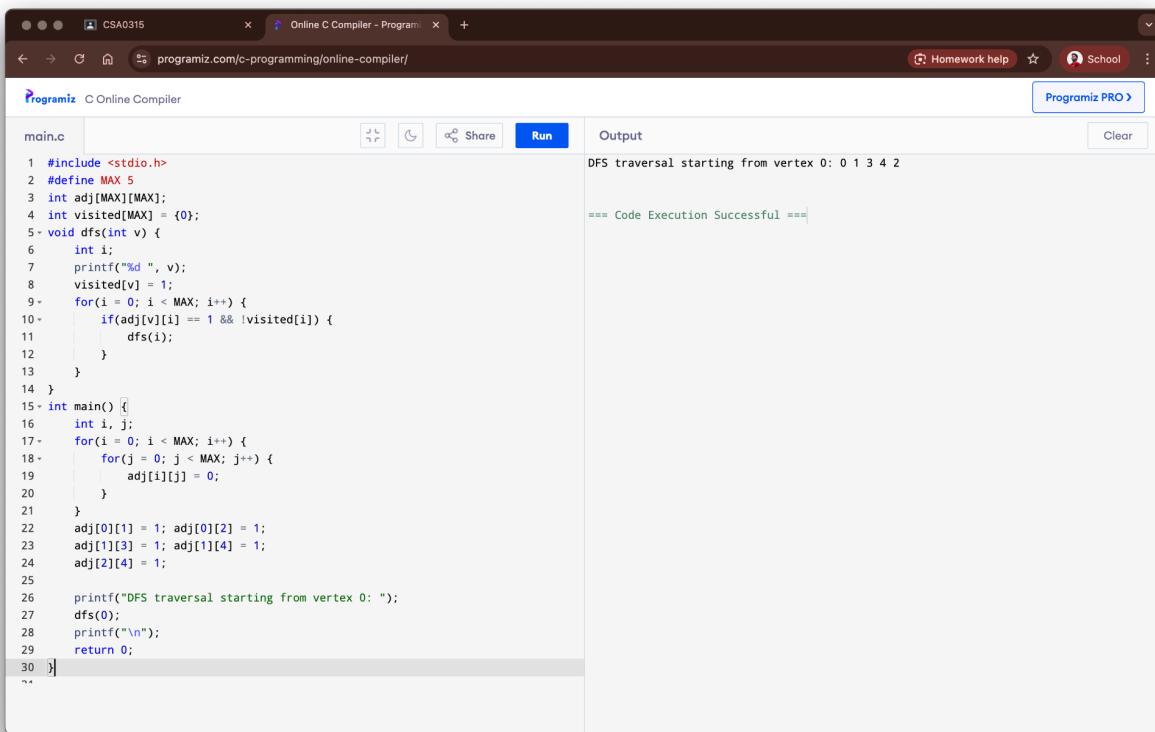
22. Graph Traversal using Depth First Search (DFS)

Aim:

To write a C program to traverse a graph using the Depth First Search (DFS) algorithm.

Algorithm:

1. Start at a source vertex. Mark it as visited.
2. Explore as far as possible along each branch before backtracking.
3. Recursively visit an unvisited adjacent vertex of the current vertex.
4. Mark it as visited and repeat the process.
5. If there are no unvisited adjacent vertices, backtrack.



The screenshot shows a web-based C compiler interface. The code in the editor is as follows:

```
main.c
1 #include <stdio.h>
2 #define MAX 5
3 int adj[MAX][MAX];
4 int visited[MAX] = {0};
5 void dfs(int v) {
6     int i;
7     printf("%d ", v);
8     visited[v] = 1;
9     for(i = 0; i < MAX; i++) {
10        if(adj[v][i] == 1 && !visited[i]) {
11            dfs(i);
12        }
13    }
14 }
15 int main() {
16     int i, j;
17     for(i = 0; i < MAX; i++) {
18         for(j = 0; j < MAX; j++) {
19             adj[i][j] = 0;
20         }
21     }
22     adj[0][1] = 1; adj[0][2] = 1;
23     adj[1][3] = 1; adj[1][4] = 1;
24     adj[2][4] = 1;
25
26     printf("DFS traversal starting from vertex 0: ");
27     dfs(0);
28     printf("\n");
29     return 0;
30 }
```

The output window displays the results of the program execution:

```
DFS traversal starting from vertex 0: 0 1 3 4 2
== Code Execution Successful ==
```

Result:

The program for DFS graph traversal was successfully executed.

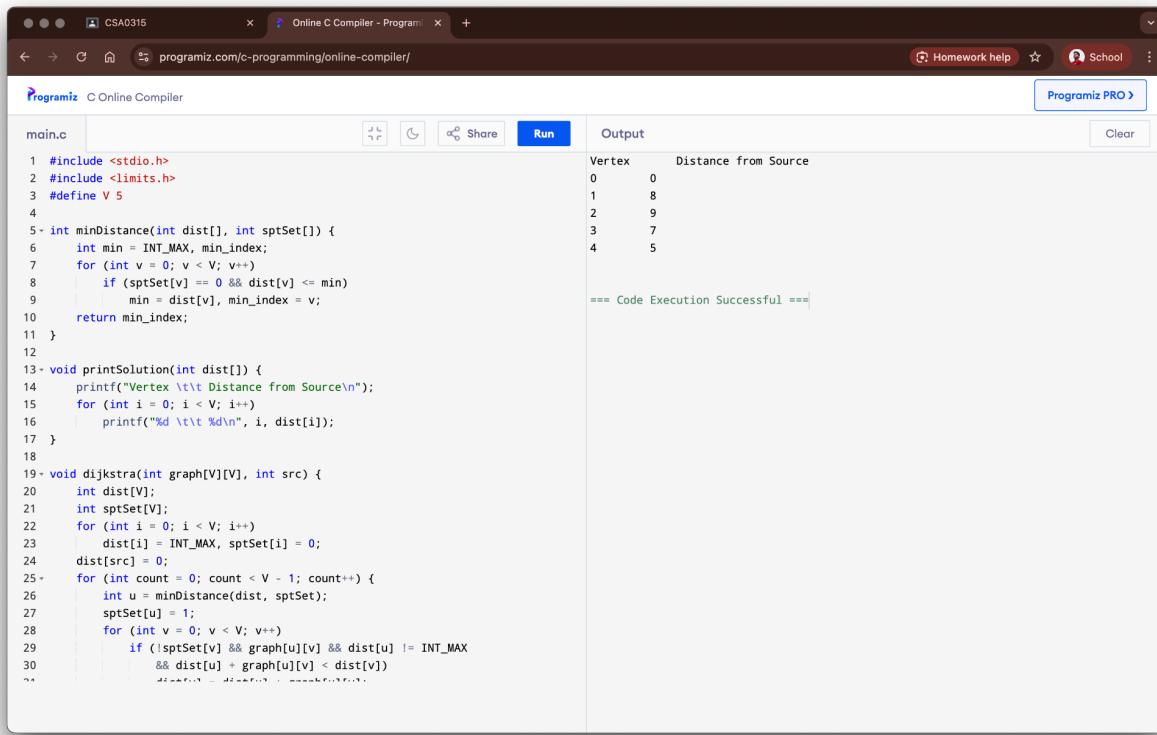
23. Shortest Path using Dijkstra's Algorithm

Aim:

To implement Dijkstra's algorithm to find the shortest path from a source vertex to all other vertices in a weighted graph.

Algorithm:

1. Initialize distances to all vertices as infinite and the source vertex as 0.
2. Create a set of unvisited vertices.
3. While the unvisited set is not empty, select the vertex u with the smallest distance.
4. For each neighbor v of u , update its distance if the path through u is shorter.
5. Remove u from the unvisited set. Repeat until all vertices are visited.



The screenshot shows a web-based C compiler interface. The code in the editor is for Dijkstra's algorithm, implemented in main.c. The output window displays the shortest distances from the source vertex (0) to all other vertices (1, 2, 3, 4). The output is as follows:

Vertex	Distance from Source
0	0
1	8
2	9
3	7
4	5

Below the table, a message indicates successful execution: "==== Code Execution Successful ===".

Result:

The program for Dijkstra's algorithm was successfully executed.

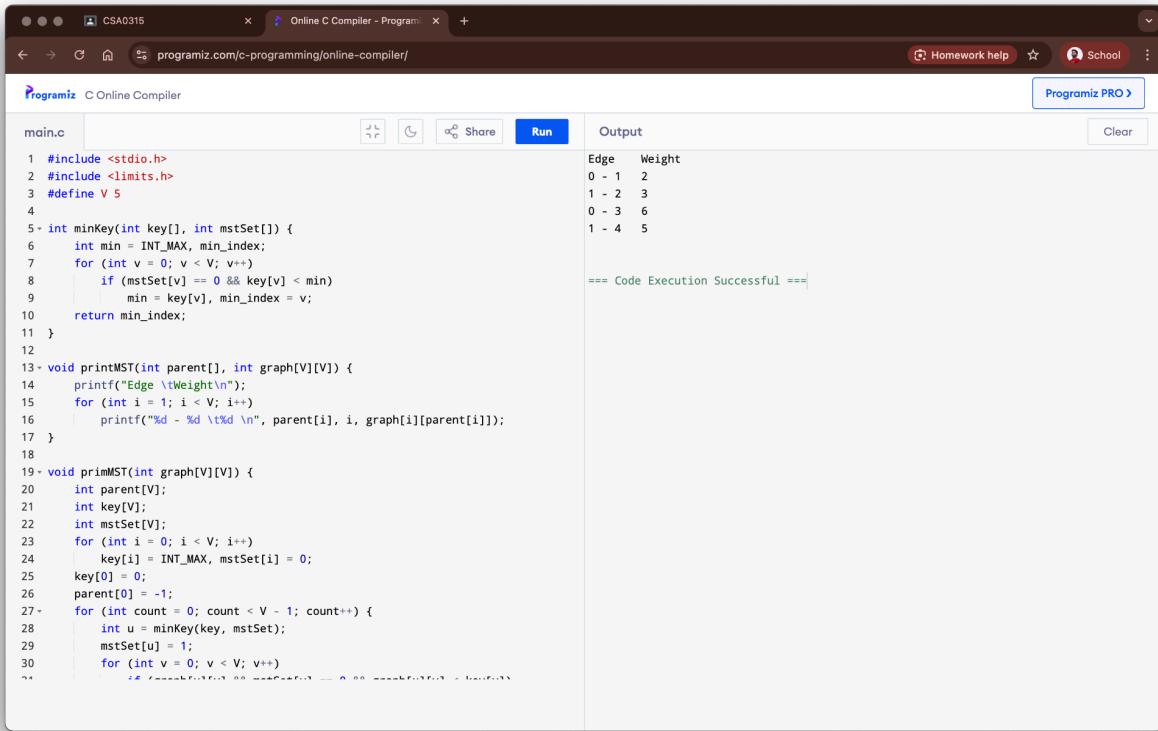
24. Minimum Spanning Tree using Prim's Algorithm

Aim:

To implement Prim's algorithm to find the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph.

Algorithm:

1. Initialize the MST with a single vertex (e.g., the first one).
2. Maintain a set of vertices already included in the MST.
3. In each step, find the edge with the minimum weight that connects a vertex in the MST to a vertex outside the MST.
4. Add this edge and the corresponding vertex to the MST.
5. Repeat until all vertices are included in the MST.



The screenshot shows a web-based C compiler interface. The code in the editor is for Prim's algorithm, which finds the Minimum Spanning Tree (MST) of a weighted graph. The output window displays the edges and their weights that were selected during the execution of the algorithm. The output is as follows:

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
1 - 4	5

Below the table, a message indicates the execution was successful.

```
main.c
1 #include <stdio.h>
2 #include <limits.h>
3 #define V 5
4
5 int minKey(int key[], int mstSet[]) {
6     int min = INT_MAX, min_index;
7     for (int v = 0; v < V; v++)
8         if (mstSet[v] == 0 && key[v] < min)
9             min = key[v], min_index = v;
10    return min_index;
11 }
12
13 void printMST(int parent[], int graph[V][V]) {
14     printf("Edge \tWeight\n");
15     for (int i = 1; i < V; i++)
16         printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
17 }
18
19 void primMST(int graph[V][V]) {
20     int parent[V];
21     int key[V];
22     int mstSet[V];
23     for (int i = 0; i < V; i++)
24         key[i] = INT_MAX, mstSet[i] = 0;
25     key[0] = 0;
26     parent[0] = -1;
27     for (int count = 0; count < V - 1; count++) {
28         int u = minKey(key, mstSet);
29         mstSet[u] = 1;
30         for (int v = 0; v < V; v++)
31             if (graph[u][v] > 0 && graph[u][v] < key[v])
```

Result:

The program for Prim's algorithm was successfully executed.

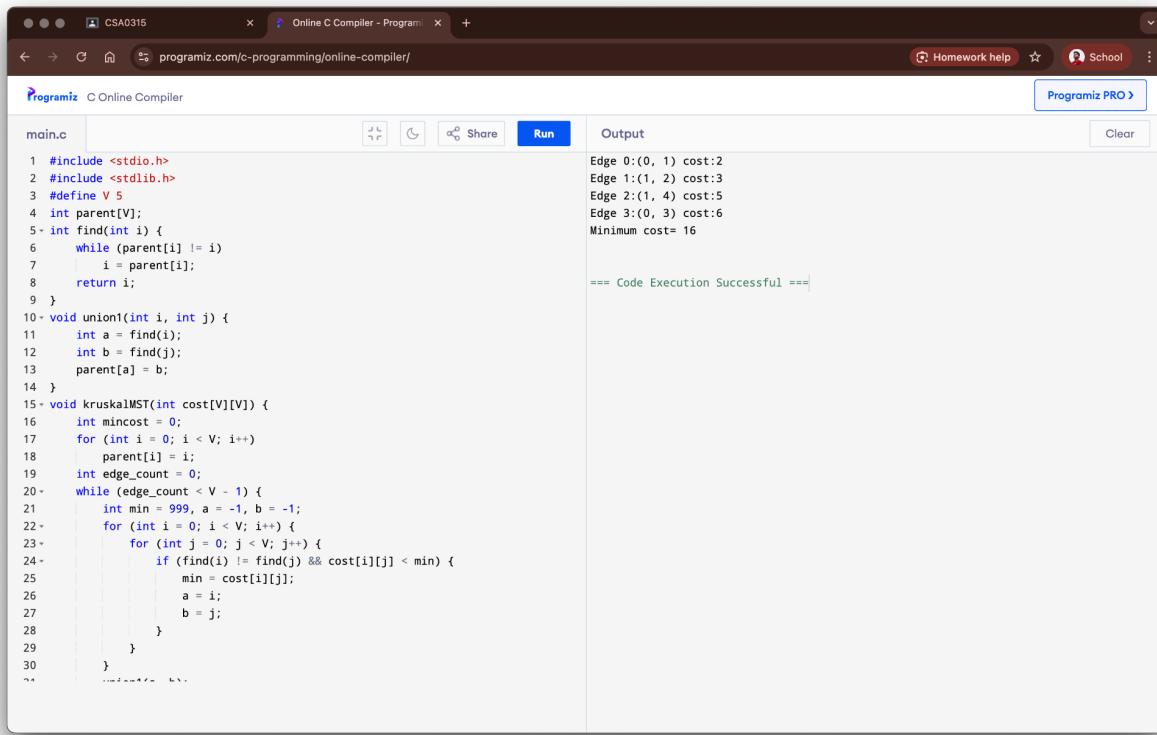
25. Minimum Spanning Tree using Kruskal's Algorithm

Aim:

To implement Kruskal's algorithm to find the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph.

Algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
3. If no cycle is formed, include this edge. Else, discard it.
4. Repeat step 2 and 3 until there are $(V-1)$ edges in the spanning tree.
5. The 'find' and 'union' operations are used to detect cycles.



The screenshot shows a web-based C compiler interface. In the code editor (main.c), the following C code is written:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define V 5
4 int parent[V];
5 int find(int i) {
6     while (parent[i] != i)
7         i = parent[i];
8     return i;
9 }
10 void union(int i, int j) {
11     int a = find(i);
12     int b = find(j);
13     parent[a] = b;
14 }
15 void kruskalMST(int cost[V][V]) {
16     int mincost = 0;
17     for (int i = 0; i < V; i++)
18         parent[i] = i;
19     int edge_count = 0;
20     while (edge_count < V - 1) {
21         int min = 999, a = -1, b = -1;
22         for (int i = 0; i < V; i++) {
23             for (int j = 0; j < V; j++) {
24                 if (find(i) != find(j) && cost[i][j] < min) {
25                     min = cost[i][j];
26                     a = i;
27                     b = j;
28                 }
29             }
30         }
31     }
32 }
```

In the output window, the results of the execution are displayed:

```
Edge 0:(0, 1) cost:2
Edge 1:(1, 2) cost:3
Edge 2:(1, 4) cost:5
Edge 3:(0, 3) cost:6
Minimum cost= 16

==== Code Execution Successful ====
```

Result:

The program for Kruskal's algorithm was successfully executed.