## 1. Create and traverse a linked list.

```
#include <stdio.h>
#include <stdlib.h>
struct node{
   int data;
   struct node *next;
};

void LL(struct node *ptr){
   int i=1;
   while (ptr->next!=NULL)
   {
      printf("Value[%d]=%d\n",i++,ptr->data);
      ptr=ptr->next;
   }
}

void main(){
   struct node *list,*start;
   int i,n;
   list = (struct node*)malloc(sizeof(struct node));
   start=list;
   printf("How mmany element do you want:");
   scanf("%d",&n);
   for (i=1;i<=n;i++)
   {
      printf("Data Value =");
      scanf("%d",&list->data);
      list->next=(struct node*)malloc(sizeof(struct node));
      list=list->next;
   }
   list->next=NULL;
   printf("list all the elements of linked list\n");
   LL(start);
}
```

## 2. Insert element at linked list.

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};

// Function to print the linked list
void LL(struct node *ptr) {
   int i = 1;
   while (ptr != NULL) {
```

```c
        printf("Value[%d] = %d\n", i++, ptr->data);
        ptr = ptr->next;
    }
}

// Function to insert a new element at a specific position
void insertAtPosition(struct node **head, int data, int position) {
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = data;

    // If inserting at the head (position 0)
    if (position == 0) {
        newNode->next = *head;
        *head = newNode;
        return;
    }

    struct node *current = *head;
    for (int i = 0; current != NULL && i < position - 1; i++) {
        current = current->next;
    }

    // If the position is greater than the number of nodes
    if (current == NULL) {
        printf("Position %d is out of bounds. Inserting at the end instead.\n", position);
        free(newNode);
        return;
    }

    newNode->next = current->next;
    current->next = newNode;
}

// Main function
int main() {
    struct node *list, *start;
    int i, n, data, position;

    // Initialize the linked list
    list = (struct node *)malloc(sizeof(struct node));
    start = list;

    printf("How many elements do you want: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        printf("Data Value = ");
        scanf("%d", &list->data);
        if (i < n) {
            list->next = (struct node *)malloc(sizeof(struct node));
            list = list->next;
        }
```

```
    }
    list->next = NULL;

    printf("List all the elements of the linked list:\n");
    LL(start);

    // Ask user for a new element and position to insert
    printf("Enter a new data value to insert: ");
    scanf("%d", &data);
    printf("Enter the position to insert the new element (0-based index): ");
    scanf("%d", &position);

    // Insert the new element at the specified position
    insertAtPosition(&start, data, position);

    // Print the updated linked list
    printf("Updated linked list:\n");
    LL(start);

    // Free the allocated memory
    struct node *current = start;
    struct node *nextNode;
    while (current != NULL) {
        nextNode = current->next;
        free(current);
        current = nextNode;
    }

    return 0;
}
```

## 3. Delete an item from linked list.

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *next;
};

// Function to print the linked list
void LL(struct node *ptr) {
    int i = 1;
    while (ptr != NULL) {
        printf("Value[%d] = %d\n", i++, ptr->data);
        ptr = ptr->next;
    }
}

// Function to delete a node at a specific index
```

```c
void deleteNodeAtIndex(struct node **head, int index) {
    if (*head == NULL) {
        printf("The linked list is empty. Cannot delete from index %d.\n", index);
        return;
    }

    struct node *current = *head;

    // If the head needs to be removed
    if (index == 0) {
        *head = current->next; // Move head to the next node
        free(current); // Free the memory of the deleted node
        printf("Deleted node at index %d.\n", index);
        return;
    }

    // Find the previous node of the node to be deleted
    for (int i = 0; current != NULL && i < index - 1; i++) {
        current = current->next;
    }

    // If the position is more than the number of nodes
    if (current == NULL || current->next == NULL) {
        printf("Index %d does not exist in the linked list.\n", index);
        return;
    }

    // Node current->next is the node to be deleted
    struct node *nextNode = current->next->next;
    free(current->next); // Free the memory of the deleted node
    current->next = nextNode; // Unlink the deleted node from the list
    printf("Deleted node at index %d.\n", index);
}

// Main function
int main() {
    struct node *list, *start;
    int i, n, index;

    // Initialize the linked list
    list = (struct node *)malloc(sizeof(struct node));
    start = list;

    printf("How many elements do you want: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        printf("Data Value = ");
        scanf("%d", &list->data);
        if (i < n) {
            list->next = (struct node *)malloc(sizeof(struct node));
            list = list->next;
```

```
        }
    }
    list->next = NULL;

    printf("List all the elements of the linked list:\n");
    LL(start);

    // Ask user for an index to delete
    printf("Enter the index to delete from the linked list (0-based index): ");
    scanf("%d", &index);
    deleteNodeAtIndex(&start, index); // Delete the specified element

    // Print the linked list after deletion
    printf("Updated linked list:\n");
    LL(start);

    // Free the allocated memory
    struct node *current = start;
    struct node *nextNode;
    while (current != NULL) {
        nextNode = current->next;
        free(current);
        current = nextNode;
    }

    return 0;
}
```

## 4. Create circular linked list and print all them .

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *next;
};

// Function to create a new node
struct node* createNode(int data) {
    struct node* newNode = (struct node*)malloc(sizeof(struct node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a new node at the end of the circular linked list
void insertEnd(struct node** head, int data) {
    struct node* newNode = createNode(data);
```

```c
    if (*head == NULL) {
        *head = newNode;
        newNode->next = newNode; // Point to itself, making it circular
    } else {
        struct node* current = *head;
        while (current->next != *head) {
            current = current->next; // Traverse to the last node
        }
        current->next = newNode; // Link the new node
        newNode->next = *head; // Make it circular
    }
}

// Function to print the circular linked list
void traverse(struct node* head) {
    if (head == NULL) {
        printf("The circular linked list is empty.\n");
        return;
    }
    struct node* current = head;
    do {
        printf("Value = %d\n", current->data);
        current = current->next;
    } while (current != head);
}

// Main function
int main() {
    struct node* head = NULL; // Initialize the head of the circular linked list
    int n, data;

    printf("How many elements do you want to insert into the circular linked list? ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Data Value = ");
        scanf("%d", &data);
        insertEnd(&head, data); // Insert the new element
    }

    printf("List all the elements of the circular linked list:\n");
    traverse(head); // Traverse and print the circular linked list

    // Free the allocated memory
    struct node* current = head;
    struct node* nextNode;
    if (current != NULL) {
        do {
            nextNode = current->next;
            free(current);
            current = nextNode;
        } while (current != head);
```

```
    }

    return 0;
}
```

## 5. Sorting 2 polynomial equation in linked list and find out their sum in another linked list.

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a polynomial term
struct Term {
    int coeff; // Coefficient
    int exp;   // Exponent
    struct Term* next;
};

// Function to create a new term
struct Term* createTerm(int coeff, int exp) {
    struct Term* newTerm = (struct Term*)malloc(sizeof(struct Term));
    newTerm->coeff = coeff;
    newTerm->exp = exp;
    newTerm->next = NULL;
    return newTerm;
}

// Function to insert a term in sorted order
void insertTerm(struct Term** head, int coeff, int exp) {
    struct Term* newTerm = createTerm(coeff, exp);
    if (*head == NULL || (*head)->exp < exp) {
        newTerm->next = *head;
        *head = newTerm;
    } else {
        struct Term* current = *head;
        while (current->next != NULL && current->next->exp >= exp) {
            current = current->next;
        }
        newTerm->next = current->next;
        current->next = newTerm;
    }
}

// Function to print the polynomial
void printPolynomial(struct Term* head) {
    struct Term* current = head;
    while (current != NULL) {
        printf("%dx^%d ", current->coeff, current->exp);
```

```
        if (current->next != NULL) {
            printf("+ ");
        }
        current = current->next;
    }
    printf("\n");
}

// Function to add two polynomials
struct Term* addPolynomials(struct Term* poly1, struct Term* poly2) {
    struct Term* result = NULL;
    struct Term* current1 = poly1;
    struct Term* current2 = poly2;

    while (current1 != NULL || current2 != NULL) {
        int coeff = 0;
        int exp = 0;

        if (current1 != NULL && (current2 == NULL || current1->exp > current2->exp)) {
            coeff = current1->coeff;
            exp = current1->exp;
            current1 = current1->next;
        } else if (current2 != NULL && (current1 == NULL || current2->exp > current1->exp)) {
            coeff = current2->coeff;
            exp = current2->exp;
            current2 = current2->next;
        } else {
            coeff = current1->coeff + current2->coeff;
            exp = current1->exp;
            current1 = current1->next;
            current2 = current2->next;
        }

        if (coeff != 0) {
            insertTerm(&result, coeff, exp);
        }
    }

    return result;
}

// Main function
int main() {
    struct Term* poly1 = NULL;
    struct Term* poly2 = NULL;

    // Input for first polynomial
    printf("Enter terms for the first polynomial (coeff exp), end with -1 -1:\n");
    while (1) {
        int coeff, exp;
        scanf("%d %d", &coeff, &exp);
```

```
        if (coeff == -1 && exp == -1) break;
        insertTerm(&poly1, coeff, exp);
    }

    // Input for second polynomial
    printf("Enter terms for the second polynomial (coeff exp), end with -1 -1:\n");
    while (1) {
        int coeff, exp;
        scanf("%d %d", &coeff, &exp);
        if (coeff == -1 && exp == -1) break;
        insertTerm(&poly2, coeff, exp);
    }

    // Print the polynomials
    printf("First Polynomial: ");
    printPolynomial(poly1);
    printf("Second Polynomial: ");
    printPolynomial(poly2);

    // Add the polynomials
    struct Term* sum = addPolynomials(poly1, poly2);
    printf("Sum of Polynomials: ");
    printPolynomial(sum);

    // Free allocated memory
    struct Term* current;
    while (poly1 != NULL) {
        current = poly1;
        poly1 = poly1->next;
        free(current);
    }
    while (poly2 != NULL) {
        current = poly2;
        poly2 = poly2->next;
        free(current);
    }
    while (sum != NULL) {
        current = sum;
        sum = sum->next;
        free(current);
    }

    return 0;
}
```

Output: Enter terms for the first polynomial (coeff exp), end with -1 -1:
3 2
5 1
2 0
-1 -1

Enter terms for the second polynomial (coeff exp), end with -1 -1:
4 2
1 1
3 0
-1 -1

First Polynomial: 3x^2 + 5x^1 + 2x^0
Second Polynomial: 4x^2 + 1x^1 + 3x^0
Sum of Polynomials: 7x^2 + 6x^1 + 5x^0

## 6. Write a program for multiplication of two numbers using recursion function.

```
#include <stdio.h>
int mul(int a , int b){
   if(a==0)
      return 0;
   else
      return( b + mul ( a - 1, b ));
}

int main(){
   int a,b;
   printf("Enters two numbers to multiply:");
   scanf("%d %d",&a,&b);
   int result = mul(a,b);
   printf("The result of %d * %d is : %d \n",a,b,result);
   return 0;
}
```

## 7. WAP in c to implement a stack with the function of push,pop,peek,print stack.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100 // Maximum size of the stack

// Stack structure
struct Stack {
   int arr[MAX];
   int top;
};

// Function to initialize the stack
```

```c
void initStack(struct Stack* stack) {
    stack->top = -1; // Stack is initially empty
}

// Function to check if the stack is full
int isFull(struct Stack* stack) {
    return stack->top == MAX - 1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

// Function to push an element onto the stack
void push(struct Stack* stack, int value) {
    if (isFull(stack)) {
        printf("Stack Overflow! Cannot push %d\n", value);
        return;
    }
    stack->arr[++stack->top] = value; // Increment top and add value
    printf("Pushed %d onto the stack\n", value);
}

// Function to pop an element from the stack
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow! Cannot pop from an empty stack\n");
        return -1; // Return -1 to indicate an error
    }
    return stack->arr[stack->top--]; // Return the top value and decrement top
}

// Function to peek at the top element of the stack
int peek(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty! Cannot peek\n");
        return -1; // Return -1 to indicate an error
    }
    return stack->arr[stack->top]; // Return the top value
}

// Function to print the stack
void printStack(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = stack->top; i >= 0; i--) {
        printf("%d ", stack->arr[i]);
```

```
    }
    printf("\n");
}

// Main function
int main() {
    struct Stack stack;
    initStack(&stack); // Initialize the stack

    int choice, value;

    do {
        printf("\nStack Operations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Print Stack\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(&stack, value);
                break;
            case 2:
                value = pop(&stack);
                if (value != -1) {
                    printf("Popped %d from the stack\n", value);
                }
                break;
            case 3:
                value = peek(&stack);
                if (value != -1) {
                    printf("Top element is %d\n", value);
                }
                break;
            case 4:
                printStack(&stack);
                break;
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 5);

    return 0;
```

```
}
```

## 8. WAP for sorting and binary search.

```
#include <stdio.h>

// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            // Swap arr[j] and arr[j + 1]
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}

// Function to perform Binary Search
int binarySearch(int arr[], int n, int target) {
   int left = 0;
   int right = n - 1;

   while (left <= right) {
      int mid = left + (right - left) / 2;

      // Check if target is present at mid
      if (arr[mid] == target) {
         return mid; // Target found
      }
      // If target is greater, ignore left half
      else if (arr[mid] < target) {
         left = mid + 1;
      }
      // If target is smaller, ignore right half
      else {
         right = mid - 1;
      }
   }
   return -1; // Target not found
}

// Main function
int main() {
   int n;

   // Input the number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &n);
```

```
    int arr[n];

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

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

    // Input the target value to search
    int target;
    printf("Enter the value to search: ");
    scanf("%d", &target);

    // Perform binary search
    int result = binarySearch(arr, n, target);
    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found in the array.\n", target);
    }

    return 0;
}
```

# 9. WAP in c for implement bubble sort, insertion sort , selection sort to sort an unsorted array .

```
#include <stdio.h>

// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
```

```
        }
    }
}

// Function to perform Insertion Sort
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1] that are greater than key
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Find the minimum element in unsorted array
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap the found minimum element with the first element
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int n;

    // Input the number of elements
    printf("Enter the number of elements: ");
```

```
    scanf("%d", &n);

    int arr[n];

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

    // Bubble Sort
    int bubbleArr[n];
    for (int i = 0; i < n; i++) {
        bubbleArr[i] = arr[i]; // Copy original array
    }
    bubbleSort(bubbleArr, n);
    printf("Sorted array using Bubble Sort: ");
    printArray(bubbleArr, n);

    // Insertion Sort
    int insertionArr[n];
    for (int i = 0; i < n; i++) {
        insertionArr[i] = arr[i]; // Copy original array
    }
    insertionSort(insertionArr, n);
    printf("Sorted array using Insertion Sort: ");
    printArray(insertionArr, n);

    // Selection Sort
    int selectionArr[n];
    for (int i = 0; i < n; i++) {
        selectionArr[i] = arr[i]; // Copy original array
    }
    selectionSort(selectionArr, n);
    printf("Sorted array using Selection Sort: ");
    printArray(selectionArr, n);

    return 0;
}
```

# 9. Perform count sort.

```
#include <stdio.h>
#include <stdlib.h>

// Function to perform Counting Sort
void countingSort(int arr[], int n) {
    // Find the maximum element in the array
```

```
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }

    // Create a count array to store the count of each unique element
    int* count = (int*)calloc(max + 1, sizeof(int));

    // Store the count of each element
    for (int i = 0; i < n; i++) {
        count[arr[i]]++;
    }

    // Build the sorted output array
    int index = 0;
    for (int i = 0; i <= max; i++) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }

    // Free the allocated memory for the count array
    free(count);
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int n;

    // Input the number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int* arr = (int*)malloc(n * sizeof(int));

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

```
    // Perform Counting Sort
    countingSort(arr, n);

    // Print the sorted array
    printf("Sorted array using Counting Sort: ");
    printArray(arr, n);

    // Free the allocated memory for the array
    free(arr);

    return 0;
}
```

# 10. Perform quick sort.

```
#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function to place the pivot element at the correct position
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as the pivot
    int i = (low - 1); // Index of the smaller element

    for (int j = low; j < high; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++; // Increment index of smaller element
            swap(&arr[i], &arr[j]); // Swap
        }
    }
    swap(&arr[i + 1], &arr[high]); // Swap the pivot element with the element at i + 1
    return (i + 1); // Return the partitioning index
}

// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partitioning index
```

```
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Main function
int main() {
    int n;

    // Input the number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

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

    // Perform Quick Sort
    quickSort(arr, 0, n - 1);

    // Print the sorted array
    printf("Sorted array using Quick Sort: ");
    printArray(arr, n);

    return 0;
}
```

# 11. WAP in c to traverse graph using DFS and BFS.

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#define MAX 100

// Function to perform DFS
void DFS(int graph[MAX][MAX], int visited[], int vertex, int numVertices) {
    visited[vertex] = 1; // Mark the current vertex as visited
    printf("%d ", vertex); // Print the visited vertex

    // Recur for all the vertices adjacent to this vertex
    for (int i = 0; i < numVertices; i++) {
        if (graph[vertex][i] == 1 && !visited[i]) {
            DFS(graph, visited, i, numVertices);
        }
    }
}

// Function to perform BFS
void BFS(int graph[MAX][MAX], int visited[], int startVertex, int numVertices) {
    int queue[MAX], front = 0, rear = 0;

    visited[startVertex] = 1; // Mark the starting vertex as visited
    queue[rear++] = startVertex; // Enqueue the starting vertex

    while (front < rear) {
        int currentVertex = queue[front++]; // Dequeue a vertex
        printf("%d ", currentVertex); // Print the visited vertex

        // Get all adjacent vertices of the dequeued vertex
        for (int i = 0; i < numVertices; i++) {
            if (graph[currentVertex][i] == 1 && !visited[i]) {
                visited[i] = 1; // Mark as visited
                queue[rear++] = i; // Enqueue the vertex
            }
        }
    }
}

// Main function
int main() {
    int numVertices;
    int graph[MAX][MAX];

    // Input the number of vertices
    printf("Enter the number of vertices: ");
    scanf("%d", &numVertices);

    // Input the adjacency matrix
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            scanf("%d", &graph[i][j]);
        }
```

```
    }

    // Input the starting vertex
    int startVertex;
    printf("Enter the starting vertex (0 to %d): ", numVertices - 1);
    scanf("%d", &startVertex);

    // Perform DFS
    int visited[MAX] = {0}; // Initialize visited array for DFS
    printf("DFS Traversal: ");
    DFS(graph, visited, startVertex, numVertices);
    printf("\n");

    // Reset visited array for BFS
    for (int i = 0; i < numVertices; i++) {
        visited[i] = 0;
    }

    // Perform BFS
    printf("BFS Traversal: ");
    BFS(graph, visited, startVertex, numVertices);
    printf("\n");

    return 0;
}
```

**Output**

```
Enter the number of vertices: 4
Enter the adjacency matrix:
0 1 1 0
1 0 0 1
1 0 0 1
0 1 1 0
Enter the starting vertex (0 to 3): 0
DFS Traversal: 0 1 3 2
BFS Traversal: 0 1 2 3


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