UNIVERSITY OF ENGINEERING AND TECHNOLOGY PESHAWAR

Department of Computer Science

Submitted to :

Sadiq-Ur-Rehman
Data Structure and Algorithm

Submitted by :

Name: Shayan Hassan
Registration No: 24pwbcs1064

Course Name: Data Structure and Algorithm
Semester: 3rd Semester

Assignment No: All Lab Manual
Date of Submission: 22/04/2025

Write the components of function declaration.

int add(int a, int b);

It have return type

Function name

Parenthesis and parameter in parenthesis

Write the components of function definition.

```
int add(int a, int b) {
    return a + b;
}
```

It have return type

Function name

Parenthesis and parameter in parenthesis

Function body and have semi colon;

Function Declaration:

void DisplayFloat();

Function definition

```
void DisplayFloat() {
    float num;
```

```
    cout << "Enter a float: ";
    cin >> num;
    cout << "You entered: " << num << endl;
}
```

There are certain situations where we add fractions such as

1/3 + 2/3 = 3/3 = 1.

2/3 - 1/3 = 1/3.

1/3 × 2/3 = (1 × 2)/ (3 × 3) = 2 / 9

1/3 ÷ 2/3 = 1/3 × 3/2= (1 × 3)/ (3 × 2) = 3/6 = 1/2 .

1/3 +1 = (1 + 3) / 3 = 4/3

1/3 - 1 = (1 - 3) / 3 = -2 /3

Total function that needed for this situation.

- Add Fractions
- Subtract Fractions
- Multiply Fractions
- Divide Fractions
- Add Fraction With Int

Write a program to traverse the array ?

```
#include <iostream>
using namespace std;
```

```
int main() {
    int arr[5] = {10, 20, 30, 40, 50};

    for(int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

Write a program to insert elements at beginning , end and at specified location of array ?

```
#include <iostream>
using namespace std;

int main() {
    int arr[100], n, pos, val;

    cin >> n;
    for(int i = 0; i < n; i++) cin >> arr[i];
cin >> pos >> val;
```

```
    for(int i = n; i > pos; i--) arr[i] = arr[i - 1];
    arr[pos] = val;
    n++;

    for(int i = 0; i < n; i++) cout << arr[i] << " ";
    return 0;
}
```

Write a program to delete elements from beginning

```
#include <iostream>
using namespace std;

int main() {
    int arr[100], n;

    cin >> n;
    for(int i = 0; i < n; i++) cin >> arr[i];

    for(int i = 0; i < n - 1; i++) arr[i] = arr[i + 1];
    n--;
```

```
 for(int i = 0; i < n; i++) cout << arr[i] << " ";


    return 0;
}
```

Write code to merge two arrays.

```
#include <iostream>
using namespace std;

void mergeArrays(int arr1[], int arr2[], int arr3[], int n1, int n2) {
    for(int i = 0; i < n1; i++)
        arr3[i] = arr1[i];

        for(int i = 0; i < n2; i++)
        arr3[n1 + i] = arr2[i];
}

int main() {
    int arr1[] = {1, 2, 3, 4, 5};
```

```
    int arr2[] = {6, 7, 8};
    int n1 = sizeof(arr1) / sizeof(arr1[0]);
    int n2 = sizeof(arr2) / sizeof(arr2[0]);
    int arr3[n1 + n2];  // Array to store merged result

    mergeArrays(arr1, arr2, arr3, n1, n2);

    for(int i = 0; i < n1 + n2; i++)
        cout << arr3[i] << " ";

    return 0;
}
```

```
#include <iostream>
using namespace std;

int sequentialSearch(int arr[], int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i;
```

```
        }
    }
    return -1;
}

int main() {
    int arr[] = {34, 50, 23, 12, 7, 90};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key = 23;

    int index = sequentialSearch(arr, size, key);

    if (index != -1) {
        cout << "Element " << key << " is present at index " << index << endl;
    } else {
```

What will be time complixity of worst case.

The time complexity of worst case will be O(n).

Find the binary search in the given array if the array is sorted.

```
#include <iostream>
using namespace std;

int binarySearch(int arr[], int size, int key) {
    int low = 0;
    int high = size - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key) {
            return mid;
        }
        else if (arr[mid] < key) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
```

```
    }

    return -1;
}

int main() {
    int arr[] = {7, 12, 23, 34, 50, 90};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key = 23;

    int index = binarySearch(arr, size, key);

    if (index != -1) {
        cout << "Element " << key << " is present at index " << index << endl;
    } else {
        cout << "Element " << key << " is not present in the array" << endl;
    }
```

```
    return 0;
}
```

What will be the time complexity of the binary search?

The time complexity of binary search is $O(n^2)$.

Develop an algorithm for checking that the given array is sorted or not?

Start from the first element of the array.

Compare the current element with the next element.

If the current element is greater than the next element, the array is not sorted.

If you reach the end of the array without finding any out-of-order elements, the array is sorted.

Return true if the array is sorted, and false otherwise.

Assume the data stored in the array is not sorted. Develop the function that sorts the array using Bubble Sort.

```
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
```

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

void displayArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr) / sizeof(arr[0]);

    cout << "Original array: ";
    displayArray(arr, size);

    bubbleSort(arr, size);
```

```
    cout << "Sorted array: ";
    displayArray(arr, size);

    return 0;
}
```

What is the complexity of bubble sort for best, worst and average case?

Time complexity for best case is O(n).

Time complexity for average case is $O(n^2)$.

Time complexity for worst case will also be $O(n^2)$.

Assume the data stored in the array is not sorted. Develop the function that sorts the array using selection Sort.

```
#include <iostream>
using namespace std;

void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
```

```
            minIndex = j;
        }
    }
    if (minIndex != i) {
        swap(arr[i], arr[minIndex]);
    }
  }
}

void displayArray(int arr[], int size) {
  for (int i = 0; i < size; i++) {
    cout << arr[i] << " ";
  }
  cout << endl;
}

int main() {
  int arr[] = {64, 34, 25, 12, 22, 11, 90};
  int size = sizeof(arr) / sizeof(arr[0]);

  cout << "Original array: ";
  displayArray(arr, size);
```

```
    selectionSort(arr, size);

    cout << "Sorted array: ";
    displayArray(arr, size);

    return 0;
}
```

Find out number of comparisons and moves in selection sort for an array of size 20.

```
#include <iostream>
using namespace std;

void selectionSortWithCount(int arr[], int size, int &comparisons, int &moves) {
    comparisons = 0;
    moves = 0;

    for (int i = 0; i < size - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            comparisons++;
            if (arr[j] < arr[minIndex]) {
```

```
            minIndex = j;
        }
    }
    if (minIndex != i) {
        swap(arr[i], arr[minIndex]);
        moves += 3; // Each swap involves 3 moves (2 for the elements and 1 for the comparison)
    }
  }
}

int main() {
  int arr[20] = {20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
  int comparisons, moves;

  selectionSortWithCount(arr, 20, comparisons, moves);

  cout << "Number of comparisons: " << comparisons << endl;
  cout << "Number of moves: " << moves << endl;

  return 0;
```

```
}
```

What is the complexity of selection sort for best ,worst and average case ?

Time complexity for best case is $O(n^2)$.

Time complexity for average case is $O(n^2)$.

Time complexity for worst case will also be $O(n^2)$.

Define the term recursion.

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.

Write a recursive function that calculates the Factorial of positive integer.

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);}
```

Write a function that recursively computes the nth Fibonacci number.

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Write a recursive function of Binary search.

```
int binarySearch(int arr[], int low, int high, int key) {
    if (low > high) return -1;
    int mid = low + (high - low) / 2;
    if (arr[mid] == key) return mid;
    if (arr[mid] > key) return binarySearch(arr, low, mid - 1, key);
    return binarySearch(arr, mid + 1, high, key);
}
```

Write a recursive function that check the given string is palindrome or not.

```
bool isPalindrome(string str, int start, int end) {
    if (start >= end) return true;
    if (str[start] != str[end]) return false;
    return isPalindrome(str, start + 1, end - 1);
}
```

Write a recursive function that find out the greatest common divisor of two positive integers.

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}
```

Write a program to solve the Towers of Hanoi puzzle recursively.

```
void towerOfHanoi(int n, char from, char to, char aux) {
    if (n == 1) {
        cout << "Move disk 1 from " << from << " to " << to << endl;
        return;
    }
    towerOfHanoi(n - 1, from, aux, to);
    cout << "Move disk " << n << " from " << from << " to " << to << endl;
    towerOfHanoi(n - 1, aux, to, from);
}
```

Write a program that calculates the nth power of number by recursively.

```
int power(int base, int exp) {
    if (exp == 0) return 1;
    return base * power(base, exp - 1);
}
```

Write a program that reads a string consisting of only parentheses and that determines whether the parentheses are balanced or not.

```
bool isBalanced(string str, int index = 0, int count = 0) {
    if (index == str.length()) return count == 0;
    if (str[index] == '(') count++;
```

```
    else if (str[index] == ')') count--;
    if (count < 0) return false;
    return isBalanced(str, index + 1, count);
}
```

Write a program to implement Merge Sort .

```
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}


void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
```

```
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```

Q1: Define the term Stack

A **Stack** is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle, where elements are added and removed from the **top** only.

Q2: Why are stacks called "LIFO"?

Because the **Last** element that is **Inserted** is the **First** to be **Removed**.

Q3: How many types of operations are implemented in stack?

Basic operations:

- push() – insert an element
- pop() – remove top element
- peek() – return top element
- isEmpty() – check if empty
- isFull() – check if full (in static array implementation)

Q4: Declare static stack of size 10

```
#define SIZE 10
int stack[SIZE];
```

```
int top = -1;
```

Q5: Declare dynamic stack

```
int* stack;
int top = -1;
int size;

void createStack(int s) {
    size = s;
    stack = new int[size];
}
```

Q6: Function to find size of stack

```
int stackSize() {
    return top + 1;
}
```

Q7: Check if stack is empty

```
bool isEmpty() {
    return top == -1;
}
```

Q8: Check if stack is full

```
bool isFull() {
    return top == size - 1;
}
```

Q9: Push element

```
void push(int value) {
    if (!isFull())
        stack[++top] = value;
}
```

Q10: Pop element

```
int pop() {
    if (!isEmpty())
        return stack[top--];
    return -1; // error
}
```

Q11–Q13: Expressions

- **POSTFIX**: Operators follow operands: ab+
- **INFIX**: Operators between operands: a + b
- **PREFIX**: Operators before operands: +ab

Q14: Infix to Postfix

a) a * b + c – d → ab* c + d -
b) (((a + b) – c) * d) – e → ab+ c - d * e -
c) a + b / (c + d) → ab cd + / +

Q15: Postfix to Infix

a) ab* cd- + → (a * b) + (c - d)
b) ab cd + / + → a + (b / (c + d))
c) ab+ c - d * e - → (((a + b) – c) * d) – e

LAB EXERCISE

Q1: Stack Class with All Operations

```
class Stack {
private:
    int *arr, top, size;

public:
    Stack(int s = 10) {
        size = s;
        arr = new int[size];
        top = -1;
    }

    bool isFull() {
        return top == size - 1;
    }

    bool isEmpty() {
        return top == -1;
    }

    void push(int x) {
        if (!isFull())
```

```
        arr[++top] = x;
    }

    int pop() {
        if (!isEmpty())
            return arr[top--];
        return -1;
    }

    int peek() {
        if (!isEmpty())
            return arr[top];
        return -1;
    }

    int stackSize() {
        return top + 1;
    }

    ~Stack() {
        delete[] arr;
    }
};
```

Q2: Decimal to Binary Using Stack

```
void decimalToBinary(int num) {
    Stack s(32);
    while (num > 0) {
        s.push(num % 2);
```

```
        num /= 2;
    }
    while (!s.isEmpty())
        cout << s.pop();
}
```

Q3: Decimal to Any Base

```
void decimalToBase(int num, int base) {
    Stack s(32);
    char digits[] = "0123456789ABCDEF";
    while (num > 0) {
        s.push(num % base);
        num /= base;
    }
    while (!s.isEmpty())
        cout << digits[s.pop()];
}
```

Q4: Infix to Postfix

```
int precedence(char op) {
    if (op == '^') return 3;
    if (op == '*' || op == '/') return 2;
    if (op == '+' || op == '-') return 1;
    return 0;
}

string infixToPostfix(string infix) {
    Stack s(100);
```

```
    string postfix = "";

    for (char c : infix) {
        if (isalnum(c))
            postfix += c;
        else if (c == '(')
            s.push(c);
        else if (c == ')') {
            while (!s.isEmpty() && s.peek() != '(')
                postfix += (char)s.pop();
            s.pop(); // remove '('
        } else {
            while (!s.isEmpty() && precedence((char)s.peek()) >=
precedence(c))
                postfix += (char)s.pop();
            s.push(c);
        }
    }
    while (!s.isEmpty())
        postfix += (char)s.pop();
    return postfix;
}
```

Q5: Postfix to Infix

```
string postfixToInfix(string postfix) {
    Stack s(100);
    for (char c : postfix) {
        if (isalnum(c)) {
            string op(1, c);
```

```
            s.push((int)(new string(op))); // treating pointer as int for
simplicity
        } else {
            string op2 = *(string*)s.pop();
            string op1 = *(string*)s.pop();
            string expr = "(" + op1 + c + op2 + ")";
            s.push((int)(new string(expr)));
        }
    }
    return *(string*)s.pop();
}
```

Q6: Balanced Parentheses Check

```
bool areParenthesesBalanced(string expr) {
    Stack s(100);
    for (char c : expr) {
        if (c == '(')
            s.push(c);
        else if (c == ')') {
            if (s.isEmpty()) return false;
            s.pop();
        }
    }
    return s.isEmpty();
}
```