In [1]:

```
/*Reches P. Eric K.
18/10/2023
Lab 5 1.0*/
```

In [2]:

```
/*
We start by including the libraries needed to run the program.
You won't need to know all the contents of these, and some libraries
are only needed to run some pre-written code you don't have to handle.
*/
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
```

In [3]:

```
/*
Calculate the factorial of a non-negative integer n. If a negative integer is passe

Factorial of a non-negative integer n, denoted as n!, is the product of all positiv
For example, 5! = 5 * 4 * 3 * 2 * 1 = 120.

param n: The non-negative integer for which to calculate the factorial.
return: The factorial of n.
*/
int flawedFactorial(int n) {
    int result = 1;
    if (n<0){
        result = -1;
        return result;
    }
    // Multiply result by all positive integers from 1 to n.
    for (int i = 1; i <= n; i++) {
        result *= i;
    }

    return result;
}
```

In [4]:

```
/*
Run a test for the flawedFactorial function and compare the result with the expecte
*/
void runTestFlawedFactorial(int num, int expected) {
    // Calculate the factorial using the flawedFactorial function.
    int result = flawedFactorial(num);

    // Check if the result matches the expected value.
    if (result == expected) {
        std::cout << "Test for " << num << " passed. Expected result: " << expected
    } else {
        // If the result doesn't match the expected value, report the test failure.
        std::cerr << "Test for " << num << " failed. Expected result: " << expected
    }
}
```

In [5]:

```
/*
Perform a set of test cases on the flawedFactorial function.

TODO: Add required test cases!

*/
void testFlawedFactorial() {
    // Define a vector of test cases in the format of {input, expected result}.
    // e.g. {{1, 1}, {5, 120}}
    std::vector<std::pair<int, int>> testCases = {
        {1, 1}, {5, 120}, {-5, -1}, {0, 1}                  //TODO: Add test cases
    };

    // Iterate through the test cases and run each test.
    for (const auto& testCase : testCases) {
        runTestFlawedFactorial(testCase.first, testCase.second);
    }
}
```

In [6]:

```
// Call the testFlawedFactorial function to verify if all the test cases pass.
testFlawedFactorial();
```

```
Test for 1 passed. Expected result: 1
Test for 5 passed. Expected result: 120
Test for -5 passed. Expected result: -1
Test for 0 passed. Expected result: 1
```

In [7]:

```
/*
Count the number of vowels in a given string.
param input: The given string.
return: The count of vowels.
*/
int flawedCountVowels(const std::string& input) {
    // Initialize a counter to keep track of the vowel count.
    int count = 0;

    // Iterate through each character (char c) in the input string.
    for (char c : input) {
        // Check if the current character is one of the vowels (a, e, i, o, u).
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u'|| c == 'A' ||
            count++;  // Increment the count if the character is a vowel.
        }
    }
    return count;
}
```

In [8]:

```
/*
Run a test for the flawedCountVowels function and compare the result with the expec
*/
void runTestFlawedCountVowels(const std::string& input, int expected) {
    // Count the vowels using the flawedCountVowels function.
    int result = flawedCountVowels(input);

    // Check if the result matches the expected value.
    if (result == expected) {
        std::cout << "Test for \"" << input << "\" passed. Expected count: " << exp
    } else {
        // If the result doesn't match the expected value, report the test failure.
        std::cerr << "Test for \"" << input << "\" failed. Expected count: " << exp
    }
}
```

In [9]:

```
/*
Perform a set of test cases on the flawedCountVowels function.

TODO: Add required test cases!

*/
void testFlawedCountVowels() {
    // Define a vector of test cases in the format of {input, expected result}.
    // e.g. {{"rhythm", 0}, {"annoy", 2}}
    std::vector<std::pair<std::string, int>> testCases = {
        {"rhythm", 0}, {"annoy", 2}, {"ChEesE", 3}, {"", 0}, {"bnnn", 0}     //TODO
    };

    // Iterate through the test cases and run each test.
    for (const auto& testCase : testCases) {
        runTestFlawedCountVowels(testCase.first, testCase.second);
    }
}
```

In [10]:

```
// Call the testFlawedCountVowels function to verify if all the test cases pass.
testFlawedCountVowels();
```

```
Test for "rhythm" passed. Expected count: 0, Actual count: 0
Test for "annoy" passed. Expected count: 2, Actual count: 2
Test for "ChEesE" passed. Expected count: 3, Actual count: 3
Test for "" passed. Expected count: 0, Actual count: 0
Test for "bnnn" passed. Expected count: 0, Actual count: 0
```

In [11]:

```
/*
Check if an array contains duplicate elements.
param arr: an array of integers.
param size: size of the input array.
return: whether array contains duplicates.

TODO: Debug the code to address incorrect result!

*/
bool containsDuplicates(int arr[], int size) {
    // Nested for loops to compare each element with every other element in the arr
    for (int i = 0; i < size; i++) {
        for (int j = i+1; j < size; j++) {
            // If the same element is found at different positions, return true (in
            if (arr[i] == arr[j]) {
                return true;
            }
        }
    }
    // If no duplicates are found, return false.
    return false;
}
```

In [12]:

```
int arr[] = {1, 3, 2, 4, 5, 8, 6};
// Call the 'containsDuplicates' function to check for duplicates in the array.
bool result = containsDuplicates(arr, sizeof(arr) / sizeof(int));
// Output a message based on the result of the duplicate check.
if (result) {
    // printf(result);
    std::cout << "The array contains duplicates." << std::endl;
} else {
    std::cout << "The array does not contain duplicates." << std::endl;
}
arr
```

```
The array does not contain duplicates.
```

Out[12]:

```
{ 1, 3, 2, 4, 5, 8, 6 }
```

In [13]:

```
int arr[] = {2, 3, 2, 4, 5, 3, 6};
// Call the 'containsDuplicates' function to check for duplicates in the array.
bool result = containsDuplicates(arr, sizeof(arr) / sizeof(int));
// Output a message based on the result of the duplicate check.
if (result) {
    std::cout << "The array contains duplicates." << std::endl;
} else {
    std::cout << "The array does not contain duplicates." << std::endl;
}
arr
```

```
The array contains duplicates.
```

Out[13]:

```
{ 2, 3, 2, 4, 5, 3, 6 }
```

In [14]:

```
/*
Calculate the mean (average) of an array of values
param values: an array of integers.
param numValues: size of the input array.
return: the mean of the array values.

TODO: Debug the code to address incorrect result!

*/
double calculateMean(const double values[], double numValues) {
    // Initialize a variable 'sum' to store the sum of all values, and set it to 0.
    double sum = 0;
    // Iterate through the 'values' array and add each value to the 'sum' variable.
    for (int i = 0; i < numValues; i++) {
        sum += values[i];
    }
    // Calculate the mean by dividing the 'sum' by the total number of values.
    double mean = sum / numValues;
    return sum / numValues;
}
```

In [15]:

```
/*
Calculate the standard deviation of an array of values
param values: an array of integers.
param numValues: size of the input array.
return: the standard deviation of the array values.

TODO: Debug the code to address incorrect result!

*/
double calculateStandardDeviation(const double values[], double numValues) {
    // Calculate the mean (average) of the 'values' array using the 'calculateMean'
    double mean = calculateMean(values, numValues);
    // Initialize a variable 'sumOfSquaredDifferences' to store the sum of squared
    double sumOfSquaredDifferences = 0.0;

    // Iterate through the 'values' array
    for (int i = 0; i <= numValues-1; i++) {
        // Calculate the difference between each value and the mean.
        double difference = values[i] - mean;
        // Add the square of the difference to the 'sumOfSquaredDifferences'.
        sumOfSquaredDifferences += difference * difference;
    }

    // Calculate the variance by dividing the 'sumOfSquaredDifferences' by the numb
    double variance = sumOfSquaredDifferences / numValues;
    // Calculate the standard deviation by taking the square root of the variance.
    double standardDeviation = std::sqrt(variance);

    return standardDeviation;
}
```

In [16]:

```
double data[] = {10.0, 15.0, 12.0, 8.0, 14.0};
double numValues = sizeof(data) / sizeof(double);

// Call the 'calculateStandardDeviation' function to compute the standard deviation
double stdDeviation = calculateStandardDeviation(data, numValues);
std::cout << "Standard Deviation: " << stdDeviation << std::endl;

data
// The correct answer for the standard deviation for the given array is 2.56125.
```

```
Standard Deviation: 2.56125
```

Out[16]:

```
{ 10.000000, 15.000000, 12.000000, 8.0000000, 14.000000 }
```

In [17]:

```
/*
Perform binary search in a sorted array for a given target.
param arr: an array of integers.
param size: size of the input array.
param target: the number to search for in the array.
return: index of the target in the rray.

TODO: Debug the code to address incorrect result!

*/
int binarySearch(int arr[], int size, int target) {
    // Initialize left and right pointers for the binary search.
    int left = 0;
    int right = size - 1;

    // Perform the binary search loop.
    while (left <= right) {

        // Calculate the middle index.
        int mid = left + (right - left) / 2;
        // Check if the middle element matches the target.
        if (arr[mid] == target) {
            return mid;  // Return the index of the target.
        } else if (arr[mid] < target) {  // If the middle element is smaller than t
            left = mid + 1;  // Update the left pointer.
        } else {
            right = mid - 2;  // Update the right pointer
        }
        if (right < 0){
            right = 0;
        }
    }
    return -1;  // Return -1 to indicate that the target is not found in the array.
}
```

In [18]:

```
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int target = 5;

// Perform a binary search for 'target' in the 'arr' array.
int result = binarySearch(arr, sizeof(arr) / sizeof(int), target);
// Check the result of the binary search and provide output accordingly.
if (result != -1) {
    std::cout << "Element " << target << " found at index " << result << std::endl;
} else {
    std::cout << "Element " << target << " not found in the array." << std::endl;
}
arr
```

```
Element 5 found at index 4
```

Out[18]:

```
{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

In [19]:

```
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int target = 1;

// Perform a binary search for 'target' in the 'arr' array.
int result = binarySearch(arr, sizeof(arr) / sizeof(int), target);
// Check the result of the binary search and provide output accordingly.
if (result != -1) {
    std::cout << "Element " << target << " found at index " << result << std::endl;
} else {
    std::cout << "Element " << target << " not found in the array." << std::endl;
}
arr
```

```
Element 1 found at index 0
```

Out[19]:

```
{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```