Portfolio Report

Full Name: Abdelrahman Tageldin  
University ID: AA2200780

Table of Contents

[Task 1 3](#_Toc184826442)

[Code: 3](#_Toc184826443)

[Output: 4](#_Toc184826444)

[Analysis: 5](#_Toc184826445)

[Task 2 5](#_Toc184826446)

[Code: 5](#_Toc184826447)

[Analysis 7](#_Toc184826448)

[Output: 7](#_Toc184826449)

[Task 3 8](#_Toc184826450)

[Code: 8](#_Toc184826451)

[Comparison chart 16](#_Toc184826452)

[Analysis 17](#_Toc184826453)

[Task 4: 18](#_Toc184826454)

[Code: 18](#_Toc184826455)

[Output 19](#_Toc184826456)

[Analysis 21](#_Toc184826457)

[Task 5 21](#_Toc184826458)

[Code 21](#_Toc184826459)

[Output 23](#_Toc184826460)

[Analysis 24](#_Toc184826461)

[Task 6 25](#_Toc184826462)

[Code 25](#_Toc184826463)

[Output 27](#_Toc184826464)

[Analysis 27](#_Toc184826465)

[Task 7 28](#_Toc184826466)

[Code 28](#_Toc184826467)

[Output 34](#_Toc184826468)

[Analysis 36](#_Toc184826469)

[Reference: 37](#_Toc184826470)

[Task 8 38](#_Toc184826471)

[Code 38](#_Toc184826472)

[Output 38](#_Toc184826473)

[Analysis 38](#_Toc184826474)

# Task 1

## Code:

**#include <iostream>  
#include <string>  
#include <algorithm>  
*using namespace* std;  
*void* selectionSort(*char* arr[], *int* n) {  
 *for* (*int* i = 0; i < n - 1; i++) {  
*// Find the minimum element in the unsorted part  
 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* swap(arr[i], arr[minIndex]);  
 }  
}  
*// Function to check if two strings are anagrams  
bool* areAnagrams(string A, string B) {  
 *int* SizeA = A.length();  
 *char* arrA[SizeA + 1];  
 *int* SizeB=B.length();  
 *char* arrB[SizeB +1];  
*// Check if both strings have the same size  
 if* (SizeA !=SizeB) {  
 *return false*;  
 }  
*// Fill arr with characters from string A  
 for* (*int* i = 0; i < SizeA; i++) {  
 arrA[i] = A[i];  
 }  
 *for* (*int* i = 0; i < SizeB; i++) {  
 arrB[i] = B[i];  
 }  
 selectionSort(arrA, SizeA);  
 selectionSort(arrB,SizeB);  
*// Compare the sorted versions of both arrays  
 for* (*int* i = 0; i < SizeA; i++) {  
 *if* (arrA[i] != arrB[i]) {  
 *return false*;  
 }  
 }  
 *return true*;  
}  
*int* main() {  
 string A;  
 string B;  
 cout << "Enter first word: ";  
 getline(cin, A);  
 cout << "Enter second word: ";  
 getline(cin, B);  
 transform(A.begin(),A.end(),A.begin(),:: tolower);  
 transform(B.begin(),B.end(),B.begin(),:: tolower);  
*// Call the function to check if the words are anagrams  
 if* (areAnagrams(A, B)) {  
 cout << "The words are anagrams." << endl;  
 } *else* {  
 cout << "The words are not anagrams." << endl;  
 }  
 *return* 0;  
}**

## Output:

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

## Analysis:

To check if 2 words are anagrams of each other we firstly check if the the 2 words have the same length if they are not then they are not anagrams if they are we then insert each character of the first string into an array and we do the same for the other string then we apply any type of sorting algorthim in this case I used selection sort after that we compare the 2 arrays if they are the same then the 2 words are anagrams of each other otherwise they are not.one thing that might cause the program not run correctly is that if one of the words is in capital punction and the other not it would would be different character in the array so the avoid this after the user input we convert both string to lowercase.

The time complexity of the main function is O(n^2) due to the selection sorting process in

the Anagrams function.

Recursive vs iterative:

the primary factor determining complexity is the algorithm itself, not whether it is iterative or recursive. Both iterative and recursive implementations of selection sort have the same O(n²) complexity.

# Task 2

## Code:

**#include <iostream>  
#include <cstdlib>  
#include <ctime>  
  
*using namespace* std;  
  
*// Function to randomly assign weights to coins, with one fake coin  
void* assignRandomWeights(*int* coins[], *int* n) {  
 *int* fakeIndex = rand() % n;  
 *int* fakeWeight = rand() % 2 == 0 ? 9 : 11; *// Either lighter (9) or heavier (11)  
  
 for* (*int* i = 0; i < n; ++i) {  
 coins[i] = 10; *// All "real" coins weigh 10 in this setup* }  
  
 coins[fakeIndex] = fakeWeight; *// Assign the fake coin its weight*}  
  
*void* findFakeCoin(*int* coins[]) {  
 *// First, check if all three coins have the same weight  
 if* (coins[0] == coins[1] && coins[1] == coins[2]) {  
 cout << "All three coins have the same weight. Unable to identify a fake coin." << endl;  
 *return*;  
 }  
  
 cout << "Comparing coin A and coin B..." << endl;  
  
 *if* (coins[0] == coins[1]) {  
 *// If A and B are equal, C must be fake* cout << "A and B are equal. C is the fake coin." << endl;  
 cout << "Comparing coin C with coin A..." << endl;  
  
 *if* (coins[2] > coins[0]) {  
 cout << "C is heavier." << endl;  
 } *else* {  
 cout << "C is lighter." << endl;  
 }  
 } *else* {  
 *// If A and B are different, one of them is fake* cout << "A and B are not equal." << endl;  
 cout << "Comparing coin A with coin C..." << endl;  
  
 *if* (coins[0] == coins[2]) {  
 *// If A and C are equal, B is fake* cout << "A and C are equal. B is the fake coin." << endl;  
  
 *if* (coins[1] > coins[0]) {  
 cout << "B is heavier." << endl;  
 } *else* {  
 cout << "B is lighter." << endl;  
 }  
 } *else* {  
 *// Otherwise, A is fake* cout << "A is the fake coin." << endl;  
  
 *if* (coins[0] > coins[2]) {  
 cout << "A is heavier." << endl;  
 } *else* {  
 cout << "A is lighter." << endl;  
 }  
 }  
 }  
}  
  
*int* main() {  
 srand(time(0)); *// Seed for random number generation  
  
 int* coins[3]; *// Array to store weights of three coins* assignRandomWeights(coins, 3); *// Randomly assign weights to the coins* cout << "Coin weights: A=" << coins[0] << ", B=" << coins[1] << ", C=" << coins[2] << endl;  
  
 findFakeCoin(coins); *// Find and identify the fake coin  
  
 return* 0;  
}**

## Analysis

The main objective of this task is to find a fake coin among a (n) number of coins while maintain a O(1) time complexity,given that the fake coin could be ligher or heavier than the real coins.the program starts by given random weights to the coins then starts to compare their weights.To achine the O(1) we have to avoid using for loops and other functions to achive the constant time complexity.i found that to achive the O(1)we must use 3 as the number of coins otherwise it is not possbile to achive the time complexity required without large amount of comparsian that will take a large amount of time and include a lot of IF condtions.After setting the number of coins to 3 we start by comparing the first 2 coins in this example we would have Coin A,B and C we start by compareing A and B if they are the same then C is the fake coin and we have to then compare A or B with C to check ifts lighter or heavier.

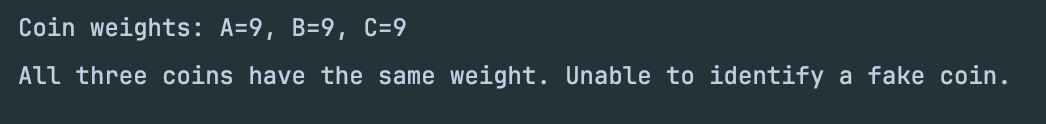
## Output:

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated



# Task 3

## Code:

**#include <iostream>**

**#include <algorithm>**

**#include <chrono>**

**#include <fstream>**

**#include <iomanip>**

**using namespace std;**

**using namespace chrono;**

**// Helper function to maintain the min-heap property**

**void min\_heapify(int arr[], int n, int i, int &count) {**

**int smallest = i;**

**int left = 2 \* i + 1;**

**int right = 2 \* i + 2;**

**if (left < n) {**

**count++; // Count comparison**

**if (arr[left] < arr[smallest])**

**smallest = left;**

**}**

**if (right < n) {**

**count++; // Count comparison**

**if (arr[right] < arr[smallest])**

**smallest = right;**

**}**

**if (smallest != i) {**

**swap(arr[i], arr[smallest]);**

**min\_heapify(arr, n, smallest, count);**

**}**

**}**

**// Min-Heap Sort that counts comparisons**

**int heap\_sort\_count(int arr[], int n) {**

**int count = 0;**

**for (int i = n / 2 - 1; i >= 0; i--)**

**min\_heapify(arr, n, i, count);**

**for (int i = n - 1; i > 0; i--) {**

**swap(arr[0], arr[i]);**

**min\_heapify(arr, i, 0, count);**

**}**

**return count;**

**}**

**// Bubble Sort that counts comparisons**

**int bubble\_sort\_count(int arr[], int n) {**

**int count = 0;**

**for (int i = 0; i < n - 1; i++) {**

**for (int j = 0; j < n - i - 1; j++) {**

**count++; // Count each comparison**

**if (arr[j] > arr[j + 1])**

**swap(arr[j], arr[j + 1]);**

**}**

**}**

**return count;**

**}**

**// Selection Sort that counts comparisons**

**int selection\_sort\_count(int arr[], int n) {**

**int count = 0;**

**for (int i = 0; i < n - 1; i++) {**

**int min\_idx = i;**

**for (int j = i + 1; j < n; j++) {**

**count++; // Count each comparison**

**if (arr[j] < arr[min\_idx])**

**min\_idx = j;**

**}**

**swap(arr[min\_idx], arr[i]);**

**}**

**return count;**

**}**

**// Insertion Sort that counts comparisons**

**int insertion\_sort\_count(int arr[], int n) {**

**int count = 0;**

**for (int i = 1; i < n; i++) {**

**int key = arr[i];**

**int j = i - 1;**

**while (j >= 0) {**

**count++; // Count each comparison**

**if (arr[j] > key)**

**arr[j + 1] = arr[j];**

**else**

**break;**

**j--;**

**}**

**arr[j + 1] = key;**

**}**

**return count;**

**}**

**int random\_arrays[30][30] = {**

**{1}, // size = 1**

**{2, 1}, // size = 2**

**{2, 3, 1}, // size = 3**

**{3, 1, 4, 2}, // size = 4**

**{2, 5, 1, 3, 4}, // size = 5**

**{4, 1, 6, 3, 2, 5}, // size = 6**

**{5, 2, 7, 3, 1, 4, 6}, // size = 7**

**{8, 2, 5, 1, 7, 4, 3, 6}, // size = 8**

**{4, 7, 2, 9, 1, 5, 3, 8, 6}, // size = 9**

**{10, 2, 6, 4, 7, 8, 3, 1, 9, 5}, // size = 10**

**{2, 11, 5, 9, 3, 1, 7, 10, 6, 8, 4}, // size = 11**

**{12, 3, 5, 1, 8, 9, 2, 4, 10, 6, 11, 7}, // size = 12**

**{3, 1, 13, 7, 10, 2, 6, 12, 5, 9, 4, 11, 8}, // size = 13**

**{5, 8, 3, 14, 2, 4, 9, 1, 6, 10, 13, 7, 12, 11}, // size = 14**

**{15, 6, 9, 3, 5, 7, 1, 12, 14, 2, 8, 4, 10, 13, 11}, // size = 15**

**{8, 6, 4, 2, 5, 3, 16, 10, 1, 15, 13, 7, 12, 9, 11, 14}, // size = 16**

**{2, 5, 7, 9, 4, 6, 3, 8, 17, 10, 1, 15, 13, 11, 14, 12, 16}, // size = 17**

**{10, 3, 6, 2, 8, 4, 18, 1, 5, 13, 17, 9, 16, 15, 14, 7, 11, 12}, // size = 18**

**{9, 6, 1, 5, 3, 7, 19, 16, 2, 14, 10, 15, 17, 4, 12, 8, 11, 13, 18}, // size = 19**

**{4, 3, 2, 5, 7, 8, 6, 20, 19, 1, 10, 17, 9, 12, 15, 13, 14, 16, 11, 18}, // size = 20**

**{7, 3, 8, 5, 6, 1, 2, 4, 21, 9, 11, 16, 20, 15, 14, 10, 13, 17, 12, 18, 19}, // size = 21**

**{6, 9, 8, 7, 5, 4, 3, 22, 2, 1, 10, 19, 12, 14, 20, 16, 21, 13, 17, 15, 18, 11}, // size = 22**

**{2, 3, 9, 7, 6, 5, 23, 1, 4, 10, 8, 20, 13, 15, 11, 17, 21, 12, 14, 22, 18, 16, 19}, // size = 23**

**{8, 1, 6, 9, 3, 7, 24, 5, 2, 4, 23, 10, 15, 13, 19, 21, 20, 12, 16, 17, 11, 14, 18, 22}, // size = 24**

**{4, 6, 5, 3, 2, 8, 25, 1, 7, 9, 10, 15, 13, 19, 23, 12, 11, 24, 16, 20, 14, 22, 17, 21, 18}, // size = 25**

**{10, 1, 9, 3, 6, 7, 2, 8, 26, 4, 23, 5, 11, 15, 13, 14, 25, 20, 17, 21, 19, 12, 16, 24, 22, 18}, // size = 26**

**{9, 2, 5, 6, 8, 27, 1, 4, 10, 3, 7, 11, 14, 16, 12, 23, 22, 17, 24, 26, 15, 25, 20, 13, 18, 21, 19}, // size = 27**

**{7, 3, 9, 1, 4, 28, 2, 8, 6, 10, 15, 5, 19, 21, 12, 16, 14, 27, 20, 22, 24, 11, 17, 25, 13, 23, 18, 26}, // size = 28**

**{2, 8, 5, 6, 7, 29, 10, 3, 1, 4, 12, 13, 9, 14, 27, 23, 15, 24, 21, 25, 22, 26, 11, 16, 17, 20, 19, 28, 18}, // size = 29**

**{10, 3, 6, 1, 30, 2, 9, 5, 7, 8, 4, 12, 20, 15, 23, 28, 16, 24, 11, 19, 21, 26, 29, 25, 18, 22, 27, 13, 17, 14} // size = 30**

**};**

**int sorted\_arrays[30][30] = {**

**{1}, // size = 1**

**{1, 2}, // size = 2**

**{1, 2, 3}, // size = 3**

**{1, 2, 3, 4}, // size = 4**

**{1, 2, 3, 4, 5}, // size = 5**

**{1, 2, 3, 4, 5, 6}, // size = 6**

**{1, 2, 3, 4, 5, 6, 7}, // size = 7**

**{1, 2, 3, 4, 5, 6, 7, 8}, // size = 8**

**{1, 2, 3, 4, 5, 6, 7, 8, 9}, // size = 9**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, // size = 10**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}, // size = 11**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}, // size = 12**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}, // size = 13**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}, // size = 14**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}, // size = 15**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}, // size = 16**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}, // size = 17**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}, // size = 18**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19}, // size = 19**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}, // size = 20**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21}, // size = 21**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22}, // size = 22**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23}, // size = 23**

**{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}, // size = 24**

**{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}, // size = 25**

**{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}, // size = 26**

**{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}, // size = 27**

**{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}, // size = 28**

**{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}, // size = 29**

**{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} // size = 30**

**};**

**int inverse\_sorted\_arrays[30][30] = {**

**{1}, // size = 1**

**{2, 1}, // size = 2**

**{3, 2, 1}, // size = 3**

**{4, 3, 2, 1}, // size = 4**

**{5, 4, 3, 2, 1}, // size = 5**

**{6, 5, 4, 3, 2, 1}, // size = 6**

**{7, 6, 5, 4, 3, 2, 1}, // size = 7**

**{8, 7, 6, 5, 4, 3, 2, 1}, // size = 8**

**{9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 9**

**{10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 10**

**{11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 11**

**{12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 12**

**{13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 13**

**{14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 14**

**{15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 15**

**{16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 16**

**{17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 17**

**{18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 18**

**{19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 19**

**{20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 20**

**{21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 21**

**{22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 22**

**{23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 23**

**{24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 24**

**{25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 25**

**{26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 26**

**{27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 27**

**{28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 28**

**{29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}, // size = 29**

**{30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1} // size = 30**

**};**

**// Function to test sorting algorithms and write results to a CSV file**

**void test\_comparisons(const string &test\_type, int size, int arr[], int n, ofstream &outfile) {**

**// Making copies of the array for each sort**

**int arr1[30], arr2[30], arr3[30], arr4[30];**

**copy(arr, arr + n, arr1);**

**copy(arr, arr + n, arr2);**

**copy(arr, arr + n, arr3);**

**copy(arr, arr + n, arr4);**

**// Heap Sort**

**auto start = high\_resolution\_clock::now();**

**int heap\_comps = heap\_sort\_count(arr1, n);**

**auto stop = high\_resolution\_clock::now();**

**auto duration = duration\_cast<nanoseconds>(stop - start);**

**outfile << test\_type << "," << size << ",Heap Sort," << heap\_comps << "," << duration.count() << "\n";**

**cout << left << setw(20) << test\_type**

**<< setw(12) << size**

**<< setw(16) << "Heap Sort"**

**<< setw(12) << heap\_comps**

**<< setw(16) << duration.count()**

**<< "\n";**

**// Bubble Sort**

**start = high\_resolution\_clock::now();**

**int bubble\_comps = bubble\_sort\_count(arr2, n);**

**stop = high\_resolution\_clock::now();**

**duration = duration\_cast<nanoseconds>(stop - start);**

**outfile << test\_type << "," << size << ",Bubble Sort," << bubble\_comps << "," << duration.count() << "\n";**

**cout << left << setw(20) << test\_type**

**<< setw(12) << size**

**<< setw(16) << "Bubble Sort"**

**<< setw(12) << bubble\_comps**

**<< setw(16) << duration.count()**

**<< "\n";**

**// Selection Sort**

**start = high\_resolution\_clock::now();**

**int selection\_comps = selection\_sort\_count(arr3, n);**

**stop = high\_resolution\_clock::now();**

**duration = duration\_cast<nanoseconds>(stop - start);**

**outfile << test\_type << "," << size << ",Selection Sort," << selection\_comps << "," << duration.count() << "\n";**

**cout << left << setw(20) << test\_type**

**<< setw(12) << size**

**<< setw(16) << "Selection Sort"**

**<< setw(12) << selection\_comps**

**<< setw(16) << duration.count()**

**<< "\n";**

**// Insertion Sort**

**start = high\_resolution\_clock::now();**

**int insertion\_comps = insertion\_sort\_count(arr4, n);**

**stop = high\_resolution\_clock::now();**

**duration = duration\_cast<nanoseconds>(stop - start);**

**outfile << test\_type << "," << size << ",Insertion Sort," << insertion\_comps << "," << duration.count() << "\n";**

**cout << left << setw(20) << test\_type**

**<< setw(12) << size**

**<< setw(16) << "Insertion Sort"**

**<< setw(12) << insertion\_comps**

**<< setw(16) << duration.count()**

**<< "\n";**

**}**

**// Main function to run tests and save results to a CSV file**

**int main() {**

**// Open a CSV file for output**

**ofstream outfile("sortingresults.csv");**

**// Write header row to CSV**

**outfile << "Test Type,Array Size,Algorithm,Comparisons,Time (ns)\n";**

**// Print a neat table header to console**

**cout << left << setw(20) << "Test Type"**

**<< setw(12) << "Array Size"**

**<< setw(16) << "Algorithm"**

**<< setw(12) << "Comparisons"**

**<< setw(16) << "Time(ns)"**

**<< "\n";**

**// Testing each category of arrays**

**for (int size = 1; size <= 30; size++) {**

**test\_comparisons("Random Array", size, random\_arrays[size - 1], size, outfile);**

**test\_comparisons("Sorted Array", size, sorted\_arrays[size - 1], size, outfile);**

**test\_comparisons("Inverse Sorted Array", size, inverse\_sorted\_arrays[size - 1], size, outfile);**

**}**

**outfile.close();**

**cout << "Results saved to sortingresults.csv\n";**

**return 0;**

**}**

## Comparison chart

**A graph on a sheet of paper

Description automatically generated**

A screenshot of a graph

Description automatically generated

A graph of a graph of a graph

Description automatically generated with medium confidence

A graph of a graph

Description automatically generated with medium confidence

## Analysis

The code is split into smaller parts firstly, I set up the array and then make copy of each array so that when the sorting algorithm runs on one the other stays the same unchanged. Then I made a function for each sorting type need Heap, bubble, selection and insertion sort that applies the sort algorithms and counts the comparison made for each one. Then a test function that runs all the sorting algorithms on the arrays and records its results in this case the comparison and time taken in nano seconds.

Time complexity

Heap sort: Recursive with a time complexity of O(n log n) In all cases.

Bubble sort: iterative with a time complexity of O(n^2) in worst case.

Selection sort: iterative with a time complexity of O(n^2) in worst case.

Insertion sort: iterative with a time complexity of O(n^2) in worst case.

# Task 4:

## Code:

## **#include <iostream>**

## **using namespace std;**

## **// Global variable to count the total number of moves**

## **int moveCount = 0;**

## **// Recursive function to solve the Tower of Hanoi problem**

## **// n - Number of disks**

## **// start - Starting rod**

## **// target - Target rod**

## **// middle - Intermediate rod**

## **void hanoi(int n, char start, char target, char middle) {**

## **// Base case: if there is only one disk**

## **if (n == 1) {**

## **// Move the disk from the starting rod to the intermediate rod**

## **cout << "Move disk 1 from " << start << " to " << middle << endl;**

## **moveCount++; // Increment the move counter**

## **// Move the disk from the intermediate rod to the target rod**

## **cout << "Move disk 1 from " << middle << " to " << target << endl;**

## **moveCount++; // Increment the move counter**

## **return; // Exit the recursion**

## **}**

## 

## **// Recursive case: solve for n-1 disks**

## **// Move the top n-1 disks from start to middle using target as auxiliary**

## **hanoi(n-1, start, target, middle);**

## **// Move the nth disk (largest) from start to middle**

## **cout << "Move disk " << n << " from " << start << " to " << middle << endl;**

## **moveCount++; // Increment the move counter**

## **// Move the n-1 disks from target back to start using middle as auxiliary**

## **hanoi(n-1, target, start, middle);**

## **// Move the nth disk from middle to target**

## **cout << "Move disk " << n << " from " << middle << " to " << target << endl;**

## **moveCount++; // Increment the move counter**

## **// Move the n-1 disks from start to target using middle as auxiliary**

## **hanoi(n-1, start, target, middle);**

## **}**

## **int main() {**

## **int n; // Variable to store the number of disks**

## **cout << "Enter the number of disks: "; // Prompt user for input**

## **cin >> n; // Read the number of disks from the user**

## **// Call the Tower of Hanoi function**

## **// Start rod: A, Target rod: C, Intermediate rod: B**

## **hanoi(n, 'A', 'C', 'B');**

## **// Print the total number of moves made**

## **cout << "Total moves: " << moveCount << endl;**

## **return 0; // Return 0 to indicate successful execution**

## **}**Output

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

## Analysis

The Tower of Hanoi is defined as a game based on three rods A, B and C and N discs. The main challenge presented by the game is in getting all the discs to a different rod but with a twist from the normal version of towers of hanoi, while having the following simple rules:

• Only a maximum of one disk is to be moved at any one time.

• Only the upper disk of each stack is able to be moved, onto a different stack or on a blank rod. • Of course, no disk may be placed upon another disk which is smaller than it

• Each move has to be from rod B or to Rod b   
Algorithm Used: Recursive Tower of Hanoi Algorithm

Why This Algorithm?

The advantage of the recursive method is that the problem is simplified when solving smaller parts of the large/more complex problem.

Other Algorithms That Apply :

Iterative Tower of Hanoi Algorithm: This kind of algorithm is based on a loop that moves through the steps rather than using recursive calls.

Differences Between Algorithms

Although both these algorithms successfully solve the Tower of Hanoi problem, they differ in the way the operations are carried out. The recursive Tower of Hanoi, as the name implies is recursive that it calls functions to solve sub-parts of the problem, unlike the iterative algorithm that makes use of the moving through the steps of the problem in a loop.

How the Code Works

The code is comprised of a recursive function hanoi, where four parameters are passed to the function; n is the number of disks, start is the start rod, target is the destination rod and middle is the auxiliary rod. The function has a base case for n being equal to 1 and otherwise continues to move top n-1 disks from the source rod to the auxiliary rod, move the nth disk from source rod to destination and finally n-1 disks from auxiliary rod to the destination rod.

Complexity Analysis

The recursive Tower of Hanoi algorithm has a time complexity of O(2^n) where n is the number of disks. This is due to the fact that each recursive call is a problem with a reduced size by one and there are 2^n moves to be made. On the other hand, the iterative Tower of Hanoi algorithm has O(2^n) time complexity, because in this algorithm, all moves are done with the help of loops and not by means of any recursive function calls.

# Task 5

## Code

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int main() {**

**int n; // Variable to store the number of disks**

**cout << "Enter the number of disks (n): "; // Prompt the user to input the number of disks**

**cin >> n; // Read the user input**

**// Ensure the input is valid (n must be greater than 0)**

**while (n <= 0) {**

**cout << "Please enter a number greater than 0: ";**

**cin >> n;**

**}**

**int size = n; // Set the size of the array to n (no need to multiply by 2 as indicated in the comment)**

**string\* disks = new string[size]; // Dynamically allocate an array of strings to hold disk colors**

**// Initialize the disks array with an alternating pattern of "white" and "black"**

**for (int i = 0; i < size; i++) {**

**disks[i] = (i % 2 == 0) ? "white" : "black"; // Use modulus operator to alternate colors**

**}**

**// Display the initial configuration of the disks**

**cout << "Initial configuration: ";**

**for (int i = 0; i < size; i++) {**

**cout << disks[i] << " ";**

**}**

**cout << endl;**

**int swapCount = 0; // Counter for the total number of swaps made**

**bool sorted = false; // Boolean flag to check if the disks are sorted**

**int iteration = 0; // Counter for the number of iterations**

**// Perform sorting using a swapping process until the array is sorted**

**while (!sorted) {**

**iteration++; // Increment the iteration count**

**int iterationSwaps = 0; // Count the number of swaps in the current iteration**

**sorted = true; // Assume the array is sorted initially**

**// Loop through the array to find "white" disks next to "black" disks and swap them**

**for (int i = 0; i < size - 1; i++) {**

**if (disks[i] == "white" && disks[i + 1] == "black") {**

**swap(disks[i], disks[i + 1]); // Swap the disks**

**swapCount++; // Increment the total swap count**

**iterationSwaps++; // Increment the iteration swap count**

**sorted = false; // Set sorted to false, as we performed a swap**

**}**

**}**

**// Print the number of interchanges (swaps) performed in the current iteration**

**cout << "Number of interchanges in iteration " << iteration << ": " << iterationSwaps << endl;**

**}**

**// Display the final configuration of the disks after sorting**

**cout << "Final configuration: ";**

**for (int i = 0; i < size; i++) {**

**cout << disks[i] << " ";**

**}**

**cout << endl;**

**// Display the total number of swaps performed**

**cout << "Total number of swaps: " << swapCount << endl;**

**delete[] disks; // Deallocate the dynamically allocated array to avoid memory leaks**

**return 0; // Indicate successful program termination**

**}**

## Output

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Analysis

In this case I used the bubble sort algorithm in a iteritave approach as its simpler and still has the same time complexity of the recursive approach. Bubble sort compares and swaps white and black disks.Any different type of sorting alorthim can be used in this case like insertion or selection sort both would work in this case.The code first starts by make a dynamic array of disks with alternating white and black values. Then it keeps looping through the array and checks for a alternating white and black that means that they are out of order so its swaps them it keeps runing until the bool value is true.with each step there is a counter to indicate the number of swaps and prints the result of each interchane in each iteration.

Time complexity

The time complexity in the case differs as it depends wither it’s a Worst case or best case.

For worst case it would be O(n^2) because every white disk must bubble through every black disk.

For best case it would be O(n) as that is the best case for bubble sort where it runs through everything once.

# Task 6

## Code

**#include <iostream>**

**#include <vector>**

**#include <sstream>**

**#include <string>**

**#include <limits>**

**using namespace std;**

**int main() {**

**int L;**

**// Input validation for line length**

**cout << "Enter the maximum line length (L): ";**

**if (!(cin >> L) || L <= 0) {**

**cerr << "Error: Line length must be a positive integer." << endl;**

**return 1;**

**}**

**cin.ignore(); // Ignore the newline character after the integer input**

**string text;**

**cout << "Enter the text: ";**

**getline(cin, text);**

**if (text.empty()) {**

**cerr << "Error: Input text cannot be empty." << endl;**

**return 1;**

**}**

**// Split the input text into words**

**vector<string> words;**

**stringstream ss(text);**

**string word;**

**while (ss >> word) {**

**if (word.size() > static\_cast<size\_t>(L)) {**

**cerr << "Error: Word '" << word << "' is longer than the maximum line length (" << L << ")." << endl;**

**return 1;**

**}**

**words.push\_back(word);**

**}**

**if (words.empty()) {**

**cerr << "Error: No valid words found in the input text." << endl;**

**return 1;**

**}**

**int n = words.size();**

**vector<int> dp(n + 1, numeric\_limits<int>::max()); // Minimum cost to format from word `i` to end**

**vector<int> nxt(n + 1, -1); // Tracks the next word index for line breaks**

**dp[n] = 0; // Base case: No cost for formatting an empty line**

**// Dynamic programming to calculate minimum penalty cost**

**for (int i = n - 1; i >= 0; i--) {**

**int total = 0; // Total characters used in the current line**

**for (int j = i; j < n; j++) {**

**total += words[j].size();**

**int used = total + (j - i); // Includes spaces between words**

**if (used > L) break; // Stop if the line exceeds maximum length**

**int cost = (j == n - 1) ? 0 : (L - used) \* (L - used); // Penalty for extra spaces**

**if (dp[j + 1] + cost < dp[i]) {**

**dp[i] = dp[j + 1] + cost; // Update minimum cost**

**nxt[i] = j + 1; // Record where to break the line**

**}**

**}**

**}**

**// Output the formatted text**

**int idx = 0, lineNum = 1;**

**cout << "\nFormatted text:\n";**

**while (idx < n) {**

**cout << "line " << lineNum++ << ":";**

**for (int k = idx; k < nxt[idx]; k++) {**

**cout << " " << words[k];**

**}**

**cout << "\n";**

**idx = nxt[idx];**

**}**

**return 0;**

**}**

## Output

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

## Analysis

How the Code works:

The program starts by dividing words using string stream.Stringstream divides the input string into words by separating the text according to spaces.Then we use dynamic programming arrays

dp[i] to store the min cost of foramtting the words from index I to the end and we use array nxt[i] to track where the it’s the best place for a line break for the optimal cost.Then we use a iterative function to calculate the cost.The function starts from the last word and iterates backward trying to add words to the same line while not exced the max charcter within the line limit. The reason I used the iterative approach instead of the recursive approcach is because the iterative approach is more efficient and avoids stack limitations, making it a better choice for large inputs in this context.Then using the nxt array the prgram prints the text with the best and most optimal line breaks

A different algorithms approach that can be used in this case is greedy algorithm.Greedy algorithms are simpler and faster but they don’t account for the min penalty cost so it might not be the best and optimal unline dynamic programming.

Time complexity:

In this case I used a nested loop so the time complexity is O(n^2).

# Task 7

## Code

**#include <iostream>  
#include <string>  
*using namespace* std;  
  
*// ANSI color codes for coloring output in the terminal*#define COLOR\_RESET "\033[0m"  
#define COLOR\_GREEN "\033[32m"  
#define COLOR\_YELLOW "\033[33m"  
  
*// Define the maze as a 2D array, where 1 represents walls and 0 represents open paths.  
// The maze is 15x15 and we have a start and end point defined below.  
int* maze[15][15] = {  
 {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},  
 {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  
 {1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1},  
 {1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1},  
 {1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1},  
 {1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1},  
 {1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1},  
 {1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1},  
 {1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1},  
 {1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1},  
 {1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1},  
 {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  
 {1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1},  
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  
 {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}  
};  
  
*// Define starting and ending coordinates in the maze  
// startX, startY is the start location  
// endX, endY is the goal location  
int* startX = 13, startY = 0;  
*int* endX = 1, endY = 14;  
  
*// Step counters to measure how many steps DFS and BFS take  
int* dfsSteps = 0, bfsSteps = 0;  
  
*// Visited arrays to keep track of visited positions for each search  
bool* visitedDFS[15][15];  
*bool* visitedBFS[15][15];  
  
*// Depth First Search (DFS) recursive function  
bool* dfs(*int* x, *int* y) {  
 *// Check bounds, wall presence, or if already visited  
 if* (x < 0 || y < 0 || x >= 15 || y >= 15 || maze[x][y] == 1 || visitedDFS[x][y])  
 *return false*;  
  
 *// Increment DFS steps* dfsSteps++;  
  
 *// If we reached the end point, return true  
 if* (x == endX && y == endY)  
 *return true*;  
  
 *// Mark this cell as visited* visitedDFS[x][y] = *true*;  
  
 *// Try moving in all four directions (right, left, down, up)  
 // If any direction leads to a solution, return true  
 return* dfs(x, y + 1) || dfs(x, y - 1) || dfs(x + 1, y) || dfs(x - 1, y);  
}  
  
*// Breadth First Search (BFS) function  
bool* bfs() {  
 *// We will use a simple queue implemented with a 2D array  
 // q will store positions (x, y)  
 int* q[225][2];  
 *int* front = 0, rear = 0;  
  
 *// Initialize visitedBFS to false  
 for* (*int* i = 0; i < 15; i++)  
 *for* (*int* j = 0; j < 15; j++)  
 visitedBFS[i][j] = *false*;  
  
 *// Enqueue the start position and mark it visited* q[rear][0] = startX;  
 q[rear][1] = startY;  
 rear++;  
 visitedBFS[startX][startY] = *true*;  
  
 *// Possible movements: right, left, down, up  
 int* directions[4][2] = {{0,1},{0,-1},{1,0},{-1,0}};  
  
 *// BFS loop  
 while* (front < rear) {  
 *// Dequeue the next position  
 int* x = q[front][0];  
 *int* y = q[front][1];  
 front++;  
  
 *// Increment BFS steps* bfsSteps++;  
  
 *// Check if we reached the end  
 if* (x == endX && y == endY)  
 *return true*;  
  
 *// Explore neighbors in all four directions  
 for* (*int* i = 0; i < 4; i++) {  
 *int* newX = x + directions[i][0];  
 *int* newY = y + directions[i][1];  
  
 *// Check boundaries, walls, and visited status  
 if* (newX >= 0 && newY >= 0 && newX < 15 && newY < 15  
 && maze[newX][newY] != 1 && !visitedBFS[newX][newY]) {  
 *// Enqueue the neighbor and mark as visited* q[rear][0] = newX;  
 q[rear][1] = newY;  
 rear++;  
 visitedBFS[newX][newY] = *true*;  
 }  
 }  
 }  
 *// If we exhaust all possibilities and never reach the end, return false  
 return false*;  
}  
  
*int* main() {  
 *// Initialize the visited arrays for DFS  
 for* (*int* i = 0; i < 15; i++)  
 *for* (*int* j = 0; j < 15; j++)  
 visitedDFS[i][j] = *false*;  
  
 *// Run DFS and BFS on the maze  
 bool* dfsResult = dfs(startX, startY);  
 *bool* bfsResult = bfs();  
  
 *// Print results of searches* cout << "\nResults:\n";  
 cout << "DFS steps: " << dfsSteps << "\n";  
 cout << "BFS steps: " << bfsSteps << "\n";  
  
 *if* (dfsResult && bfsResult)  
 cout << "Both DFS and BFS found the path.\n";  
 *else if* (dfsResult)  
 cout << "Only DFS found the path.\n";  
 *else if* (bfsResult)  
 cout << "Only BFS found the path.\n";  
 *else* cout << "No path found by either DFS or BFS.\n";  
  
 *// Print legend for visualization* cout << "\nLegend:\n";  
 cout << "# = Wall\n";  
 cout << ". = Unvisited Path (" << COLOR\_YELLOW << "yellow" << COLOR\_RESET << ")\n";  
 cout << "D = DFS Visited\n";  
 cout << "B = BFS Visited (" << COLOR\_GREEN << "green" << COLOR\_RESET << ")\n";  
 cout << "S = Start\n";  
 cout << "E = End\n\n";  
  
 *// Print the maze with DFS visited nodes highlighted* cout << "Maze with DFS visited nodes:\n ";  
 cout << "\n " << string(3\*15, '-') << "\n";  
 *for* (*int* i = 0; i < 15; i++) {  
 cout << (i < 10 ? " " : "") << i << "| ";  
 *for* (*int* j = 0; j < 15; j++) {  
 *if* (i == startX && j == startY) {  
 *// Print the start position* cout << "S ";  
 } *else if* (i == endX && j == endY) {  
 *// Print the end position* cout << "E ";  
 } *else if* (maze[i][j] == 1) {  
 *// Print wall* cout << "# ";  
 } *else if* (visitedDFS[i][j]) {  
 *// Print DFS visited node in yellow* cout << COLOR\_YELLOW <<"D "<<COLOR\_RESET;  
 } *else* {  
 *// Print unvisited path* cout << ". ";  
 }  
 }  
 cout << "\n";  
 }  
  
 *// Print the maze with BFS visited nodes highlighted* cout << "\nMaze with BFS visited nodes:\n ";  
 cout << "\n " << string(3\*15, '-') << "\n";  
 *for* (*int* i = 0; i < 15; i++) {  
 cout << (i < 10 ? " " : "") << i << "| ";  
 *for* (*int* j = 0; j < 15; j++) {  
 *if* (i == startX && j == startY) {  
 *// Print the start position* cout << "S ";  
 } *else if* (i == endX && j == endY) {  
 *// Print the end position* cout << "E ";  
 } *else if* (maze[i][j] == 1) {  
 *// Print wall* cout << "# ";  
 } *else if* (visitedBFS[i][j]) {  
 *// Print BFS visited node in green* cout << COLOR\_GREEN << "B " << COLOR\_RESET;  
 } *else* {  
 *// Print unvisited path* cout << ". ";  
 }  
 }  
 cout << "\n";  
 }  
  
 *return* 0;  
}**

## Output

A screenshot of a computer program

Description automatically generated

A screenshot of a computer screen

Description automatically generated

## Analysis

Algorithm Used: Depth-First Search and Breadth-First Search were used for the two algorithms in this code.

With DFS: The first thing to understand about depth-first search is that it's a recursive algorithm. This means we go down one branch of a tree all the way to the bottom, then back up before moving along another branch. So in finding paths through mazes with DFS we can keep trying out every possibility until eitherwe reach our goal or see that no path exists.

BFS: BFS is an iterative algorithm, which check the current floor before it moves to the nodes at next level, explores all of these nodes and their descents simultaneously. It is an example in the maze to find the shortest path.

Why Use These Algorithms?

DFS: is best suited for exploring all paths in order to find a but not necessarily the shortest one.

When the goal is to find the shortest path Bfs is best; since each subtree with a partition into subclasses must be explored before one below it can get explored.

Alternative Algorithms: If we are looking for an optimal path by distance or cost other algorithms such as A\* can also be applied here. They perform better on graphs with weights or where heuristics are beneficial.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **DFS (Depth-First Search)** | **BFS (Breadth-First Search)** |
| Memory Usage | Typically lower; utilizes recursion (call stack). | Typically higher; must store all nodes at the current breadth level. |
| Path Quality | Does not guarantee shortest path by default. | Guarantees shortest path if all edges have equal weight. |
| Speed in Small Mazes | Generally fast to implement and run. | Also fast, but the overhead of storing more nodes can be larger. |
| Efficiency on Large Mazes | More memory-efficient; may be preferable if speed is not critical. | More demanding in memory, but ensures shortest path if needed. |
| Suitability | Better when memory is limited and a shortest path is not mandatory. | Better when shortest path is needed and memory is not a primary concern. |

## Reference:

1. Unacademy. (n.d.). *Difference between BFS and DFS.* <https://unacademy.com/content/gate-cse-it/difference-between-bfs-and-dfs/>

# Task 8

## Code

## Output

## Analysis