1. Implement the Binary search algorithm regarded as a fast search algorithm with run-time complexity of Ο(log n) in comparison to the Linear Search.

#include <iostream> using namespace std;

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

while (low <= high) {

int mid = (low + high) / 2; if (arr[mid] == key)

return mid; // found else if (arr[mid] < key)

low = mid + 1; else

high = mid - 1;

}

return -1; // not found

}

int main() {

int arr[] = {11, 12, 22, 25, 34, 64, 90};

int n = sizeof(arr) / sizeof(arr[0]); int key = 25;

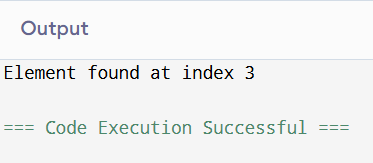
int result = binarySearch(arr, n, key); if (result != -1)

cout << "Element found at index " << result; else

cout << "Element not found";

return 0;

}



1. Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. Code the Bubble sort with the following elements:

64 34 25 12 22 11 90

#include <iostream> using namespace std;

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], arr[j + 1]);

}

}

}

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr) / sizeof(arr[0]);

bubbleSort(arr, n);

cout << "Sorted array: "; for (int i = 0; i < n; i++)

cout << arr[i] << " "; return 0;

}

**Output**

Sorted array: 11 12 22 25 34 64 90

=== Code Execution Successful===

1. Given an array of n-1 distinct integers in the range of 1 to n, find the missing number in it in a Sorted Array
   1. Linear time

#include <iostream> using namespace std;

int findMissingLinear(int arr[], int n) {

int total = (n + 1) \* (n + 2) / 2; // sum of 1 to n+1 for (int i = 0; i < n; i++) {

total -= arr[i];

}

return total;

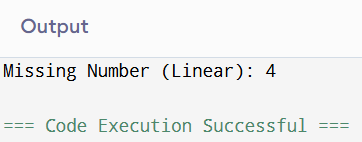
}

int main() {

int arr[] = {1, 2, 3, 5, 6}; // missing 4 int n = sizeof(arr) / sizeof(arr[0]);

cout << "Missing Number (Linear): " << findMissingLinear(arr, n); return 0;

}



3(b) Using binary search.

#include <iostream> using namespace std;

int findMissingBinary(int arr[], int n) { int low = 0, high = n - 1;

while (low <= high) {

int mid = (low + high) / 2; if (arr[mid] != mid + 1) {

if (mid == 0 || arr[mid - 1] == mid) return mid + 1;

high = mid - 1;

} else {

low = mid + 1;

}

}

return n + 1;

}

int main() {

int arr[] = {1, 2, 3, 5, 6};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Missing Number (Binary Search): " << findMissingBinary(arr, n); return 0;

}

**Output**

Missing Number (Binary Search):4

=== Code Execution Successful===

1. String Related Programs
   1. Write a program to concatenate one string to another string.
   2. Write a program to reverse a string.
   3. Write a program to delete all the vowels from the string.
   4. Write a program to sort the strings in alphabetical order.
   5. Write a program to convert a character from uppercase to lowercase.

#include <iostream> #include <algorithm> #include <string> using namespace std;

// (a) Concatenate

void concatenate(string s1, string s2) {

cout << "Concatenated: " << s1 + s2 << endl;

}

// (b) Reverse

void reverseString(string s) { reverse(s.begin(), s.end());

cout << "Reversed: " << s << endl;

}

// (c) Delete vowels

void deleteVowels(string s) { string result = "";

for (char c : s) {

if (c!='a' && c!='e' && c!='i' && c!='o' && c!='u' &&

c!='A' && c!='E' && c!='I' && c!='O' && c!='U') { result += c;

}

}

cout << "Without Vowels: " << result << endl;

}

// (d) Sort alphabetically void sortString(string s) { sort(s.begin(), s.end());

cout << "Sorted: " << s << endl;

}

// (e) Uppercase → Lowercase void toLowerCase(string s) {

for (char &c : s) c = tolower(c); cout << "Lowercase: " << s << endl;

}

int main() {

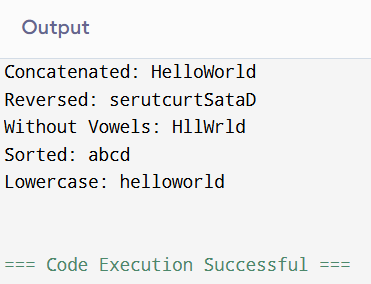
string s1 = "Hello", s2 = "World"; concatenate(s1, s2);

reverseString("DataStructures"); deleteVowels("HelloWorld"); sortString("dcba");

toLowerCase("HeLLoWORLD");

return 0;

}



1. Space required to store any two-dimensional array is number oƒ rows × number oƒ columns. Assuming array is used to store elements of the following matrices,

implement an efficient way that reduces the space requirement.

* 1. Diagonal Matrix.

#include <iostream> using namespace std;

int main() {

int n = 4; // size of matrix

int compact[n] = {1, 5, 9, 7}; // only diagonal stored

cout << "Compact storage (only diagonal elements): "; for (int i = 0; i < n; i++) cout << compact[i] << " ";

cout << "\n\nFull matrix:\n";

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

if (i == j) cout << compact[i] << " "; else cout << 0 << " ";

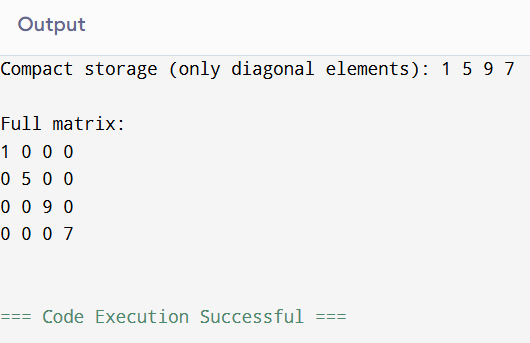
}

cout << "\n";

}

return 0;

}



* 1. Tri-diagonal Matrix.

#include <iostream> using namespace std;

int main() { int n = 4;

// compact storage: main diag + lower diag + upper diag int compact[3 \* n - 2] = {4, 2, 1, 5, 3, 7, 6, 8, 9, 10};

cout << "Compact storage: ";

for (int i = 0; i < 3 \* n - 2; i++) cout << compact[i] << " "; cout << "\n\nFull matrix:\n";

int k = 0;

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

for (int j = 0; j < n; j++) {

if (i == j) cout << compact[n + i - 1] << " "; // main diagonal else if (i == j + 1) cout << compact[i - 1] << " "; // lower diag

else if (j == i + 1) cout << compact[2 \* n + i - 1] << " "; // upper diag else cout << 0 << " ";

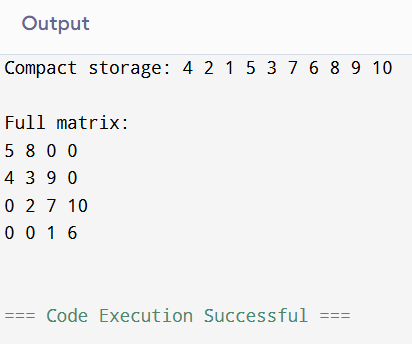
}

cout << "\n";

}

return 0;

}



* 1. Lower triangular Matrix.

#include <iostream> using namespace std;

int main() { int n = 4;

int compact[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// 4x4 → 10 elements stored

cout << "Compact storage: ";

for (int i = 0; i < 10; i++) cout << compact[i] << " "; cout << "\n\nFull matrix:\n";

int k = 0;

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

if (i >= j) cout << compact[k++] << " "; else cout << 0 << " ";

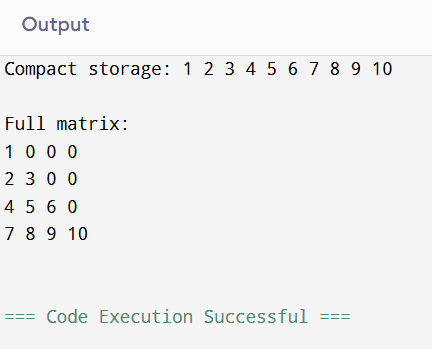
}

cout << "\n";

}

return 0;

}



* 1. Upper triangular Matrix.

#include <iostream> using namespace std;

int main() { int n = 4;

int compact[10] = {11, 12, 13, 14, 15, 16, 17, 18, 19, 20};

cout << "Compact storage: ";

for (int i = 0; i < 10; i++) cout << compact[i] << " "; cout << "\n\nFull matrix:\n";

int k = 0;

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

if (i <= j) cout << compact[k++] << " "; else cout << 0 << " ";

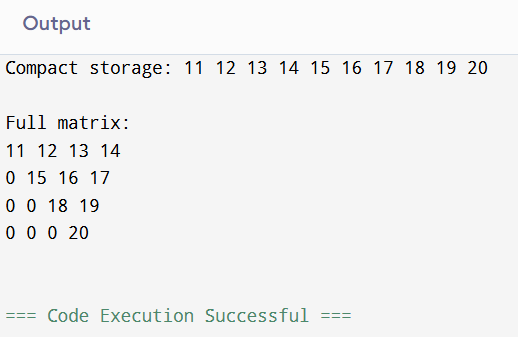
}

cout << "\n";

}

return 0;

}



* 1. Symmetric Matrix

#include <iostream> using namespace std;

int main() { int n = 4;

int compact[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

cout << "Compact storage: ";

for (int i = 0; i < 10; i++) cout << compact[i] << " "; cout << "\n\nFull matrix:\n";

int k = 0;

int A[4][4] = {0};

// fill lower triangular

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

A[i][j] = compact[k++];

A[j][i] = A[i][j]; // mirror to upper

}

}

// print

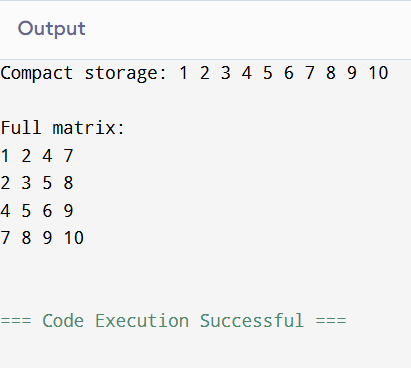
for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) cout << A[i][j] << " "; cout << "\n";

}

return 0;

}



1. Write a program to implement the following operations on a Sparse Matrix, assuming the matrix is represented using a triplet.
   1. Transpose of a matrix.

#include <iostream> using namespace std;

int main() {

int rows = 3, cols = 3; int A[3][3] = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

cout << "Original Matrix:\n"; for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) { cout << A[i][j] << " ";

}

cout << endl;

}

cout << "\nTranspose Matrix:\n"; for (int i = 0; i < cols; i++) {

for (int j = 0; j < rows; j++) { cout << A[j][i] << " ";

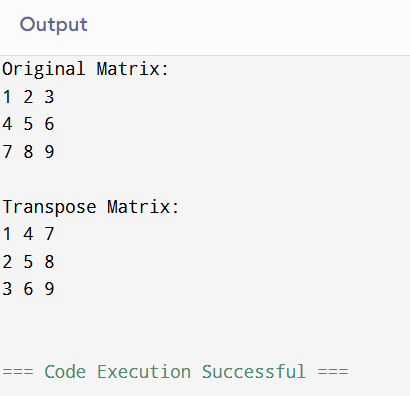
}

cout << endl;

}

return 0;

}



* 1. Addition of two matrices. #include <iostream>

using namespace std;

int main() {

int rows = 2, cols = 3;

int A[2][3] = {{1, 2, 3}, {4, 5, 6}};

int B[2][3] = {{6, 5, 4}, {3, 2, 1}}; int C[2][3];

cout << "Matrix A:\n";

for (int i = 0; i < rows; i++) { for (int j = 0; j < cols; j++) {

cout << A[i][j] << " ";

}

cout << endl;

}

cout << "\nMatrix B:\n"; for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) { cout << B[i][j] << " ";

}

cout << endl;

}

// Addition

for (int i = 0; i < rows; i++) { for (int j = 0; j < cols; j++) {

C[i][j] = A[i][j] + B[i][j];

}

}

cout << "\nMatrix A + B:\n"; for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) { cout << C[i][j] << " ";

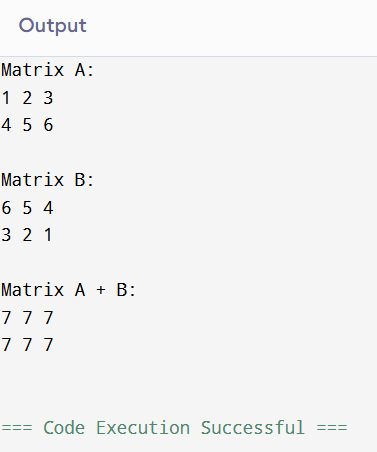
}

cout << endl;

}

return 0;

}



* 1. Multiplication of two matrices. #include <iostream>

using namespace std;

int main() {

int A[2][2] = {{1, 2}, {3, 4}};

int B[2][2] = {{5, 6}, {7, 8}}; int C[2][2] = {0};

cout << "Matrix A:\n"; for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) { cout << A[i][j] << " ";

}

cout << endl;

}

cout << "\nMatrix B:\n"; for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) { cout << B[i][j] << " ";

}

cout << endl;

}

// Multiplication

for (int i = 0; i < 2; i++) { for (int j = 0; j < 2; j++) {

for (int k = 0; k < 2; k++) { C[i][j] += A[i][k] \* B[k][j];

}

}

}

cout << "\nMatrix A x B:\n"; for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) { cout << C[i][j] << " ";

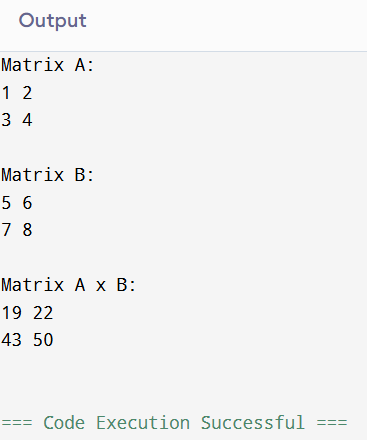
}

cout << endl;

}

return 0;

}



1. Let A[1 n] be an array of n real numbers. A pair (A[i], A[j ]) is said to be an

inversion if these numbers are out of order, i.e., i < j but A[i]>A[j ]. Write a program to count the number of inversions in an array.

#include <iostream> using namespace std;

int main() {

int arr[] = {12, 45, 7, 89, 34, 22, 90, 56};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Array elements: "; for (int i = 0; i < n; i++) {

cout << arr[i] << " ";

}

int largest = arr[0];

for (int i = 1; i < n; i++) { if (arr[i] > largest) { largest = arr[i];

}

}

cout << "\nLargest element in the array = " << largest << endl;

return 0;

}

**Output**

Array elements: 12 45 7 89 34 22 90 56 Largest element in the array= 90

Code Execution Successful



#include <iostream> using namespace std;

int main() {

int arr[] = {5, 3, 5, 2, 8, 2, 8, 9, 1};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Array elements: "; for (int i = 0; i < n; i++) {

cout << arr[i] << " ";

}

int distinctCount = 0;

for (int i = 0; i < n; i++) { bool isDistinct = true;

// Check if arr[i] appeared before for (int j = 0; j < i; j++) {

if (arr[i] == arr[j]) { isDistinct = false; break;

}

}

if (isDistinct) {

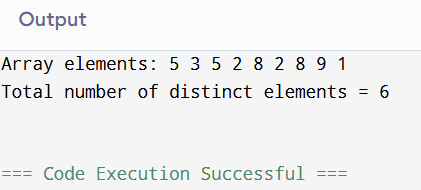
distinctCount++;

}

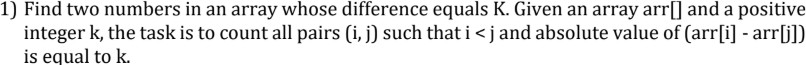
}

cout << "\nTotal number of distinct elements = " << distinctCount << endl;

return 0;

}





#include <iostream> #include <vector>

#include <unordered\_set> using namespace std;

int main() {

vector<int> arr = {1, 5, 3, 4, 2}; int k = 2;

cout << "Array elements: "; for (int x : arr) cout << x << " "; cout << "\nK = " << k << endl;

unordered\_set<int> s(arr.begin(), arr.end()); int count = 0;

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

if (s.find(arr[i] + k) != s.end()) count++;

if (s.find(arr[i] - k) != s.end()) count++;

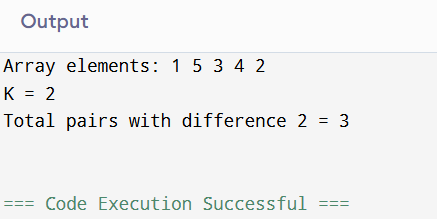
}

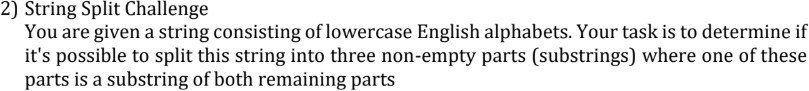
// each pair counted twice, so divide by 2

cout << "Total pairs with difference " << k << " = " << count / 2 << endl;

return 0;

}





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

bool isSubstring(string s, string sub) { return s.find(sub) != string::npos;

}

bool canSplit(string str) { int n = str.size();

// Try all ways to split string into 3 parts for (int i = 1; i < n; i++) {

for (int j = i + 1; j < n; j++) { string part1 = str.substr(0, i); string part2 = str.substr(i, j - i); string part3 = str.substr(j);

// check if any part is substring of other two

if ((isSubstring(part2, part1) && isSubstring(part3, part1)) || (isSubstring(part1, part2) && isSubstring(part3, part2)) || (isSubstring(part1, part3) && isSubstring(part2, part3))) { return true;

}

}

}

return false;

}

int main() {

string str = "abab"; // Example input

cout << "String: " << str << endl;

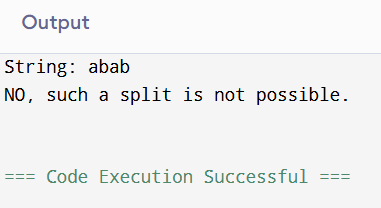
if (canSplit(str))

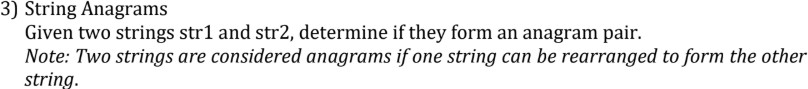
cout << "YES, it can be split into 3 parts with one substring common.\n"; else

cout << "NO, such a split is not possible.\n";

return 0;

}





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

bool areAnagrams(string str1, string str2) {

// If lengths differ, they cannot be anagrams if (str1.length() != str2.length())

return false;

// Sort both strings sort(str1.begin(), str1.end());

sort(str2.begin(), str2.end());

// Compare

return str1 == str2;

}

int main() {

string str1 = "listen"; string str2 = "silent";

cout << "String 1: " << str1 << endl; cout << "String 2: " << str2 << endl;

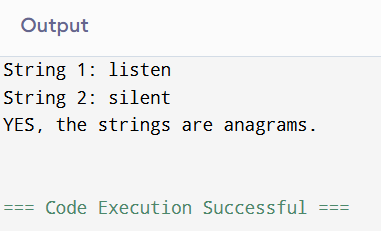
if (areAnagrams(str1, str2))

cout << "YES, the strings are anagrams.\n"; else

cout << "NO, the strings are not anagrams.\n";

return 0;

}





#include <iostream> using namespace std;

void sort012(int arr[], int n) {

int low = 0, mid = 0, high = n - 1;

while (mid <= high) { if (arr[mid] == 0) {

swap(arr[low], arr[mid]); low++;

mid++;

}

else if (arr[mid] == 1) { mid++;

}

else { // arr[mid] == 2 swap(arr[mid], arr[high]); high--;

}

}

}

int main() {

int arr[] = {2, 0, 2, 1, 1, 0};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Original Array: ";

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

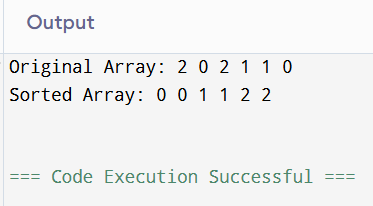
sort012(arr, n);

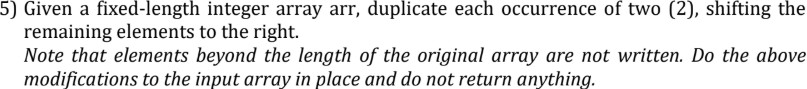
cout << "Sorted Array: ";

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

return 0;

}





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

void duplicateZeros(vector<int>& arr) { int n = arr.size();

int zeros = 0;

// Count how many zeros can be duplicated for (int i = 0; i < n; i++) {

if (arr[i] == 0) zeros++;

}

int i = n - 1; // pointer at original end

int j = n + zeros - 1; // virtual end (after duplicating zeros)

// Traverse backwards while (i < j) {

if (j < n) arr[j] = arr[i]; // Only modify inside array size

if (arr[i] == 0) { j--;

if (j < n) arr[j] = 0; // Duplicate zero

}

i--;

j--;

}

}

int main() {

vector<int> arr = {1,0,2,3,0,4,5,0};

cout << "Original Array: ";

for (int x : arr) cout << x << " "; cout << endl;

duplicateZeros(arr);

cout << "Modified Array: "; for (int x : arr) cout << x << " "; cout << endl;

return 0;

}

