**Roll No…………….. Total No. of Pages:……**

**ST-1 (SET-IV)**

**6th SEMESTER 2023-24**

**CS192- Advanced Data Structures**

**Time allowed: 90 Minutes Max. Marks: 40**

**General Instructions:**

* **Follow the instructions given in each section.**
* **Make sure that you attempt the questions in order.**

**SECTION-A (10\*1 mark=10 marks)**

***(All questions are compulsory)***

1. Given two strings, "ABCD" and "DCBA," what will be the output of their longest common prefix length?
   1. **0**
   2. 1
   3. 2
   4. 4
2. Which algorithm uses a suffix array to efficiently search for patterns in a text?
   1. Rabin-Karp Algorithm
   2. Boyer-Moore Algorithm
   3. Knuth-Morris-Pratt Algorithm
   4. **Aho-Corasick Algorithm**
3. In which scenario is the Heap Sort algorithm most suitable?
   1. When the dataset is already partially sorted
   2. When the dataset contains a large number of duplicates
   3. **When the dataset is distributed uniformly at random**
   4. When the dataset is stored in a binary heap
4. The process of dividing the array into two subarrays in Merge Sort is known as:
   1. Partitioning
   2. Merging
   3. Selection
   4. **Dividing**
5. The prime sieve algorithm works by \_\_\_\_\_\_\_\_\_\_\_finding and eliminating multiples of prime numbers.
   1. Dividing the given limit by all numbers
   2. **Checking divisibility of numbers by all primes**
   3. Factorizing all numbers up to the given limit
   4. Multiplying all numbers up to the given limit
6. In the prime sieve algorithm, vectors are commonly used to \_\_\_\_\_\_\_\_\_\_\_.
   1. **Store the prime numbers**
   2. Store the composite numbers
   3. Perform arithmetic operations on numbers
   4. Implement dynamic programming algorithms
7. In the Pigeonhole Sort algorithm, the range of elements determines the size of the:
   1. **Pigeonholes**
   2. Buckets
   3. Swaps
   4. Inversions
8. Which sorting algorithm is not an in-place sorting algorithm?
   1. Bubble Sort
   2. **Heap Sort**
   3. Quick Sort
   4. Selection Sort
9. The Z function algorithm is particularly efficient for:
   1. **Short texts and long patterns**
   2. Long texts and short patterns
   3. Texts with repeating characters
   4. Texts containing only lowercase letters
10. The time complexity of the Rabin-Karp algorithm becomes O(N) in the best-case scenario when:
    1. **The pattern occurs only once in the text**
    2. The hash function has a low collision rate
    3. The pattern and text are of the same length
    4. The pattern and text have the same characters

**SECTION-B (5\*2 mark=10 marks)**

***(All questions are compulsory)***

**SECTION-C(Coding Question) (2x5 marks=5 marks)**

Q16) Given an array arr of N integers, write a function that returns true if there is a triplet (a, b, c) that satisfies a2 + b2 = c2, otherwise false.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | N = 5  Arr[] = {3, 2, 4, 6, 5} | N = 3  Arr[] = {3, 8, 5} | N = 5  Arr[] = { 5, 43, 12, 3, 13} |
| **Output** | Yes | No | Yes |

Solution :

**#include <bits/stdc++.h>**

**using namespace std;**

**class Solution {**

**public:**

**// Function to check if a Pythagorean triplet exists in the given array.**

**bool checkTriplet(int arr[], int n) {**

**// Sort the input array in ascending order.**

**sort(arr, arr + n);**

**// Iterate through all possible triplets in the array using nested loops.**

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

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

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

**// Check if the current triplet forms a Pythagorean triplet.**

**if ((arr[i] \* arr[i] + arr[j] \* arr[j]) == arr[k] \* arr[k]) {**

**return true; // If a Pythagorean triplet is found, return true.**

**}**

**}**

**}**

**}**

**return false; // If no Pythagorean triplet is found, return false.**

**}**

**};**

**int main() {**

**int n, i;**

**cin >> n; // Input the size of the array.**

**int arr[n]; // Declare an array to store the elements of the array.**

**// Input the elements of the array.**

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

**cin >> arr[i];**

**}**

**Solution ob; // Create an object of the Solution class.**

**// Call the checkTriplet function to check if a Pythagorean triplet exists.**

**auto ans = ob.checkTriplet(arr, n);**

**// Output the result.**

**if (ans) {**

**cout << "Yes\n"; // If a Pythagorean triplet exists, print "Yes".**

**} else {**

**cout << "No\n"; // If no Pythagorean triplet exists, print "No".**

**}**

**return 0;**

**}**

Q17) Given a string, find the length of the longest substring that contains at most K distinct characters.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | Enter the input string: macronipasta  Enter the value of K: 8 | Enter the input string: lemonade  Enter the value of K: 4 | Enter the input string: fortran  Enter the value of K: 5 |
| **Output** | The length of the longest substring with at most 8 distinct characters is: 9 | The length of the longest substring with at most 4 distinct characters is: 4 | The length of the longest substring with at most 5 distinct characters is: 6 |

Solution :

**#include <iostream>**

**#include <unordered\_map>**

**using namespace std;**

**int longestSubstringLengthWithKDistinctChars(const string& s, int K) {**

**int n = s.length();**

**if (n == 0 || K == 0)**

**return 0;**

**unordered\_map<char, int> charFrequency; // Map to store character frequency in the current window**

**int left = 0; // Left pointer of the sliding window**

**int maxLen = 0; // Maximum length of the substring with at most K distinct characters**

**for (int right = 0; right < n; right++) {**

**charFrequency[s[right]]++; // Expand the window by adding the right character**

**// Contract the window from the left if we have more than K distinct characters**

**while (charFrequency.size() > K) {**

**charFrequency[s[left]]--; // Decrease the frequency of the left character**

**if (charFrequency[s[left]] == 0) {**

**charFrequency.erase(s[left]); // Remove character if its frequency becomes zero**

**}**

**left++; // Move the left pointer to contract the window**

**}**

**// Update the maximum length encountered so far**

**maxLen = max(maxLen, right - left + 1);**

**}**

**return maxLen;**

**}**

**int main() {**

**string input;**

**int K;**

**cout << "Enter the input string: ";**

**cin >> input;**

**cout << "Enter the value of K: ";**

**cin >> K;**

**int result = longestSubstringLengthWithKDistinctChars(input, K);**

**cout << "The length of the longest substring with at most " << K << " distinct characters is: " << result << endl;**

**return 0;**

**}**

**SECTION-D (Coding Question)(1x10 mark=10 mark)**

Q18) Given an array arr[] denoting heights of N towers and a positive integer K.

For each tower, you must perform exactly one of the following operations exactly once.

Increase the height of the tower by K

Decrease the height of the tower by K

Find out the minimum possible difference between the height of the shortest and tallest towers after you have modified each tower.

You can find a slight modification of the problem here.

Note: It is compulsory to increase or decrease the height by K for each tower. After the operation, the resultant array should not contain any negative integers.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | K = 2, N = 4  Arr[] = {1, 5, 8, 10} | K = 1, N = 5  Arr[] = {4, 6, 3, 2, 3} | K = 3, N = 4  Arr[] = {12, 26, 19, 18} |
| **Output** | 5 | 2 | 8 |

Solution :

**#include <bits/stdc++.h>**

**using namespace std;**

**class Solution {**

**public:**

**// Function to find the minimum possible difference between the maximum and minimum elements in the array**

**// after adding or subtracting 'k' from each element.**

**int getMinDiff(int arr[], int n, int k) {**

**sort(arr, arr + n); // Sort the array in ascending order.**

**int diff = arr[n - 1] - arr[0]; // Initialize the difference as the range between max and min elements.**

**int a\_min = 0; // Initialize variables to store potential new min and max elements.**

**int a\_max = 0;**

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

**if (arr[i] - k < 0)**

**continue; // Skip elements for which 'k' will cause a negative result.**

**// Calculate the new potential max and min elements.**

**a\_max = max(arr[n - 1] - k, arr[i - 1] + k);**

**a\_min = min(arr[0] + k, arr[i] - k);**

**diff = min(diff, a\_max - a\_min); // Update the minimum difference if needed.**

**}**

**return diff;**

**}**

**};**

**int main() {**

**int n, k;**

**cin >> k; // Input the value of 'k'.**

**cin >> n; // Input the number of elements in the array.**

**int arr[n];**

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

**cin >> arr[i]; // Input the elements of the array.**

**}**

**Solution ob; // Create an object of the Solution class.**

**auto ans = ob.getMinDiff(arr, n, k); // Call the getMinDiff function to get the result.**

**cout << ans << "\n"; // Output the minimum possible difference.**

**return 0;**

**}**