setB

1

#include <iostream>

#include <vector>

using namespace std;

int maxSubArraySum(vector<int>& nums) {

int max\_so\_far = nums[0], max\_ending\_here = nums[0];

for (int i = 1; i < nums.size(); i++) {

max\_ending\_here = max(nums[i], max\_ending\_here + nums[i]);

max\_so\_far = max(max\_so\_far, max\_ending\_here);

}

return max\_so\_far;

}

int main() {

vector<int> nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};

cout << "Maximum Subarray Sum: " << maxSubArraySum(nums) << endl;

return 0;

}

2

#include <iostream>

using namespace std;

int mergeAndCount(int arr[], int temp[], int left, int mid, int right) {

int i = left, j = mid, k = left, inv\_count = 0;

while ((i <= mid - 1) && (j <= right)) {

if (arr[i] <= arr[j]) {

temp[k++] = arr[i++];

} else {

temp[k++] = arr[j++];

inv\_count += (mid - i);

}

}

while (i <= mid - 1) temp[k++] = arr[i++];

while (j <= right) temp[k++] = arr[j++];

for (i = left; i <= right; i++) arr[i] = temp[i];

return inv\_count;

}

int countInversions(int arr[], int temp[], int left, int right) {

int mid, inv\_count = 0;

if (right > left) {

mid = (right + left) / 2;

inv\_count += countInversions(arr, temp, left, mid);

inv\_count += countInversions(arr, temp, mid + 1, right);

inv\_count += mergeAndCount(arr, temp, left, mid + 1, right);

}

return inv\_count;

}

int main() {

int arr[] = {7, 2, 6, 3};

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

int temp[n];

cout << "Number of inversions: " << countInversions(arr, temp, 0, n - 1) << endl;

return 0;

}

3

#include <iostream>

#include <limits.h>

using namespace std;

int partition(int arr[], int left, int right, int pivotIndex) {

int pivotValue = arr[pivotIndex];

swap(arr[pivotIndex], arr[right]);

int storeIndex = left;

for (int i = left; i < right; i++) {

if (arr[i] > pivotValue) {

swap(arr[storeIndex], arr[i]);

storeIndex++;

}

}

swap(arr[right], arr[storeIndex]);

return storeIndex;

}

int kthLargest(int arr[], int left, int right, int k) {

while (left <= right) {

int pivotIndex = left + (right - left) / 2;

pivotIndex = partition(arr, left, right, pivotIndex);

if (pivotIndex == k - 1) {

return arr[pivotIndex];

} else if (pivotIndex > k - 1) {

right = pivotIndex - 1;

} else {

left = pivotIndex + 1;

}

}

return -1;

}

int main() {

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

int k = 2;

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

cout << k << "th largest element: " << kthLargest(arr, 0, n - 1, k) << endl;

return 0;

}

4

#include <iostream>

#include <cctype>

using namespace std;

bool isPalindrome(const char\* s) {

int left = 0, right = strlen(s) - 1;

while (left < right) {

while (left < right && !isalnum(s[left])) left++;

while (left < right && !isalnum(s[right])) right--;

if (tolower(s[left]) != tolower(s[right])) return false;

left++;

right--;

}

return true;

}

int main() {

const char\* s = "A man, a plan, a canal: Panama";

cout << (isPalindrome(s) ? "True" : "False") << endl;

return 0;

}

5

#include <iostream>

using namespace std;

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

int totalSum = 0, leftSum = 0;

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

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

totalSum -= arr[i];

if (leftSum == totalSum) return i + 1; // 1-based index

leftSum += arr[i];

}

return -1;

}

int main() {

int arr[] = {-7, 1, 5, 2, -4, 3, 0};

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

cout << "Equilibrium index: " << findEquilibriumIndex(arr, n) << endl;

return 0;

}

#include <iostream>

using namespace std;

bool hasPairWithSumNaive(int arr[], int n, int X) {

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

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

if (arr[i] + arr[j] == X) return true;

}

}

return false;

}

int main() {

int arr[] = {1, 2, 4, 5, 7, 11};

int X = 9;

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

cout << (hasPairWithSumNaive(arr, n, X) ? "Yes" : "No") << endl;

return 0;

}

6 //naive

#include <iostream>

using namespace std;

int maxSumKConsecutive(int arr[], int n, int k) {

if (n < k) {

cout << "Invalid" << endl;

return -1;

}

int max\_sum = 0, window\_sum = 0;

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

window\_sum += arr[i];

}

max\_sum = window\_sum;

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

window\_sum += arr[i] - arr[i - k];

max\_sum = max(max\_sum, window\_sum);

}

return max\_sum;

}

int main() {

int arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20};

int k = 4;

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

cout << "Maximum sum: " << maxSumKConsecutive(arr, n, k) << endl;

return 0;

}

6 //2 pointer

#include <iostream>

using namespace std;

bool hasPairWithSumTwoPointers(int arr[], int n, int X) {

int left = 0, right = n - 1;

while (left < right) {

int sum = arr[left] + arr[right];

if (sum == X) return true;

else if (sum < X) left++;

else right--;

}

return false;

}

int main() {

int arr[] = {1, 2, 4, 5, 7, 11};

int X = 9;

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

cout << (hasPairWithSumTwoPointers(arr, n, X) ? "Yes" : "No") << endl;

return 0;

}

8

#include <iostream>

using namespace std;

int maxSumKConsecutive(int arr[], int n, int k) {

if (n < k) {

cout << "Invalid" << endl;

return -1;

}

int max\_sum = 0, window\_sum = 0;

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

window\_sum += arr[i];

}

max\_sum = window\_sum;

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

window\_sum += arr[i] - arr[i - k];

max\_sum = max(max\_sum, window\_sum);

}

return max\_sum;

}

int main() {

int arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20};

int k = 4;

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

cout << "Maximum sum: " << maxSumKConsecutive(arr, n, k) << endl;

return 0;

}

9

#include <iostream>

using namespace std;

int maxProfit(int prices[], int n) {

int min\_price = INT\_MAX, max\_profit = 0;

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

min\_price = min(min\_price, prices[i]);

max\_profit = max(max\_profit, prices[i] - min\_price);

}

return max\_profit;

}

int main() {

int prices[] = {7, 1, 5, 3, 6, 4};

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

cout << "Maximum profit: " << maxProfit(prices, n) << endl;

return 0;

}

10

#include <iostream>

#include <unordered\_map>

using namespace std;

int longestSubarrayWithSum(int nums[], int n, int target) {

unordered\_map<int, int> sumIndexMap;

int curr\_sum = 0, max\_len = 0;

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

curr\_sum += nums[i];

if (curr\_sum == target) {

max\_len = i + 1;

}

if (sumIndexMap.find(curr\_sum - target) != sumIndexMap.end()) {

max\_len = max(max\_len, i - sumIndexMap[curr\_sum - target]);

}

if (sumIndexMap.find(curr\_sum) == sumIndexMap.end()) {

sumIndexMap[curr\_sum] = i;

}

}

return max\_len;

}

int main() {

int nums[] = {1, -1, 5, -2, 3};

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

int target = 3;

cout << "Length of the longest subarray: " << longestSubarrayWithSum(nums, n, target) << endl;

return 0;

}