C++ Algorithms Practice

# Merge two sorted arrays without using extra space

void mergeArrays(int arr1[], int arr2[], int n, int m) {  
 int i = n - 1;  
 int j = 0;  
 while (i >= 0 && j < m && arr1[i] > arr2[j]) {  
 std::swap(arr1[i], arr2[j]);  
 i--;  
 j++;  
 }  
 std::sort(arr1, arr1 + n);  
 std::sort(arr2, arr2 + m);  
}

# Check if array is sorted

bool isSorted(int arr[], int n) {  
 for (int i = 1; i < n; i++) {  
 if (arr[i] < arr[i-1]) {  
 return false;  
 }  
 }  
 return true;  
}

# Count subarrays with at most k unique characters

int countSubarraysWithAtMostKUniqueChars(const std::vector<int>& arr, int k) {  
 std::unordered\_map<int, int> freq;  
 int left = 0, count = 0;  
 for (int right = 0; right < arr.size(); right++) {  
 freq[arr[right]]++;  
 while (freq.size() > k) {  
 freq[arr[left]]--;  
 if (freq[arr[left]] == 0) freq.erase(arr[left]);  
 left++;  
 }  
 count += right - left + 1;  
 }  
 return count;  
}

# Count subarrays where max - min <= k

int countSubarraysWithMaxMinDiffLessThanOrEqualToK(const std::vector<int>& arr, int k) {  
 int count = 0;  
 for (int i = 0; i < arr.size(); i++) {  
 int minElem = arr[i], maxElem = arr[i];  
 for (int j = i; j < arr.size(); j++) {  
 minElem = std::min(minElem, arr[j]);  
 maxElem = std::max(maxElem, arr[j]);  
 if (maxElem - minElem <= k) count++;  
 else break;  
 }  
 }  
 return count;  
}

# First negative number in every window of size k

std::vector<int> firstNegativeInWindow(const std::vector<int>& arr, int k) {  
 std::deque<int> dq;  
 std::vector<int> result;  
 for (int i = 0; i < arr.size(); i++) {  
 if (arr[i] < 0) dq.push\_back(i);  
 if (i >= k - 1) {  
 if (!dq.empty()) result.push\_back(arr[dq.front()]);  
 else result.push\_back(0);  
 if (!dq.empty() && dq.front() <= i - k) dq.pop\_front();  
 }  
 }  
 return result;  
}

# Largest subarray where sum <= S

int largestSubarrayWithSumLessThanOrEqualToS(const std::vector<int>& arr, int S) {  
 int left = 0, sum = 0, maxLen = 0;  
 for (int right = 0; right < arr.size(); right++) {  
 sum += arr[right];  
 while (sum > S) {  
 sum -= arr[left];  
 left++;  
 }  
 maxLen = std::max(maxLen, right - left + 1);  
 }  
 return maxLen;  
}

# Largest subarray with K distinct characters

int largestSubarrayWithKDistinct(const std::string& str, int K) {  
 std::unordered\_map<char, int> freq;  
 int left = 0, maxLen = 0;  
 for (int right = 0; right < str.size(); right++) {  
 freq[str[right]]++;  
 while (freq.size() > K) {  
 freq[str[left]]--;  
 if (freq[str[left]] == 0) freq.erase(str[left]);  
 left++;  
 }  
 maxLen = std::max(maxLen, right - left + 1);  
 }  
 return maxLen;  
}

# Length of largest subarray with no repeating characters

int lengthOfLongestSubarrayWithNoRepeatingChars(const std::string& str) {  
 std::unordered\_map<char, int> indexMap;  
 int left = 0, maxLen = 0;  
 for (int right = 0; right < str.size(); right++) {  
 if (indexMap.find(str[right]) != indexMap.end() && indexMap[str[right]] >= left) {  
 left = indexMap[str[right]] + 1;  
 }  
 indexMap[str[right]] = right;  
 maxLen = std::max(maxLen, right - left + 1);  
 }  
 return maxLen;  
}

# Minimum window substring

std::string minWindowSubstring(const std::string& s, const std::string& t) {  
 if (t.size() > s.size()) return "";  
 std::unordered\_map<char, int> freqT, windowFreq;  
 for (char c : t) freqT[c]++;  
   
 int left = 0, right = 0, minLen = s.size() + 1, start = 0, matches = 0;  
 for (right = 0; right < s.size(); right++) {  
 if (freqT.find(s[right]) != freqT.end()) {  
 windowFreq[s[right]]++;  
 if (windowFreq[s[right]] == freqT[s[right]]) matches++;  
 }  
   
 while (matches == freqT.size()) {  
 if (right - left + 1 < minLen) {  
 minLen = right - left + 1;  
 start = left;  
 }  
 if (freqT.find(s[left]) != freqT.end()) {  
 windowFreq[s[left]]--;  
 if (windowFreq[s[left]] < freqT[s[left]]) matches--;  
 }  
 left++;  
 }  
 }  
 return minLen > s.size() ? "" : s.substr(start, minLen);  
}

# Kadane's algorithm

int maxSubarraySum(const std::vector<int>& arr) {  
 int maxSum = arr[0], currentSum = arr[0];  
 for (int i = 1; i < arr.size(); i++) {  
 currentSum = std::max(arr[i], currentSum + arr[i]);  
 maxSum = std::max(maxSum, currentSum);  
 }  
 return maxSum;  
}

# Moore's voting algorithm (majority element)

int majorityElement(const std::vector<int>& arr) {  
 int candidate = arr[0], count = 1;  
 for (int i = 1; i < arr.size(); i++) {  
 if (arr[i] == candidate) count++;  
 else count--;  
 if (count == 0) {  
 candidate = arr[i];  
 count = 1;  
 }  
 }  
 count = 0;  
 for (int num : arr) {  
 if (num == candidate) count++;  
 }  
 return count > arr.size() / 2 ? candidate : -1;  
}