CODING PROBLEMS [illegible] minute read

Subarray Sum Equals K

November 12, 2021

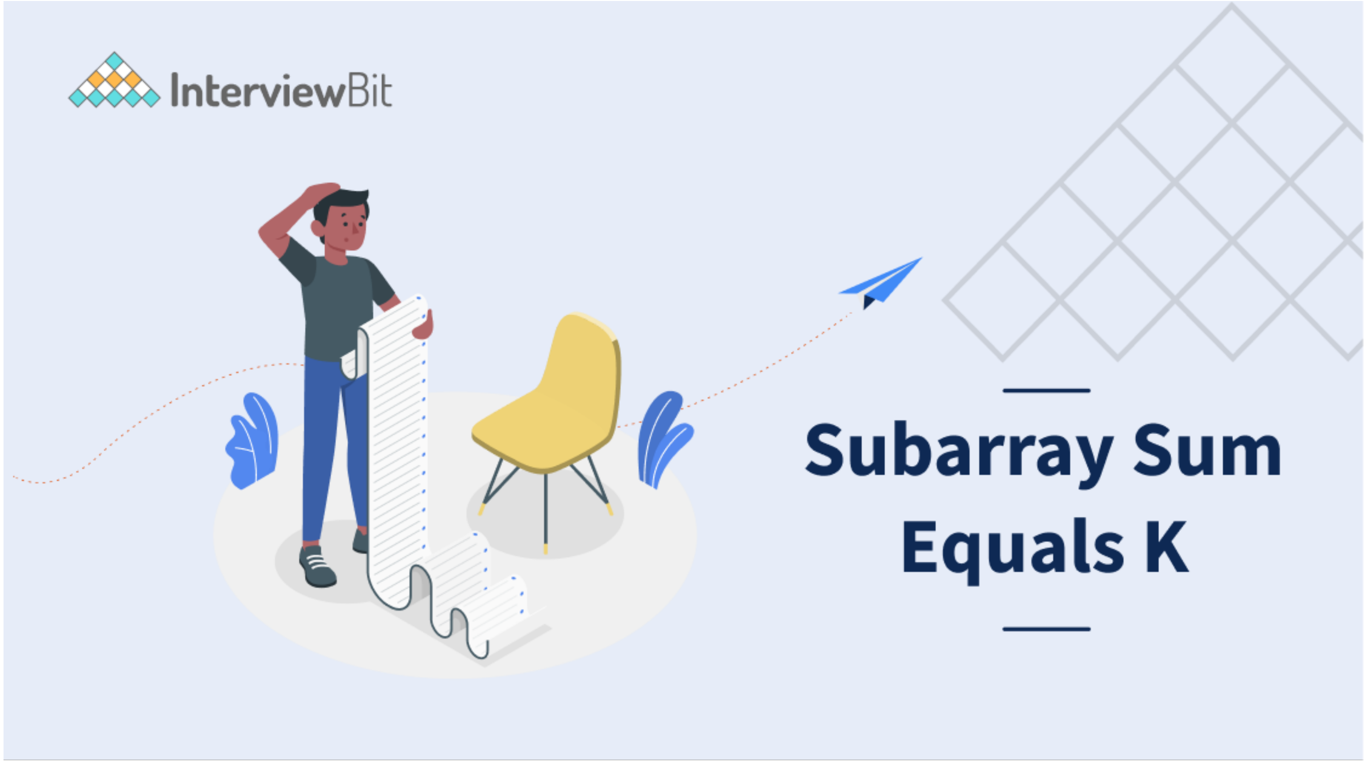
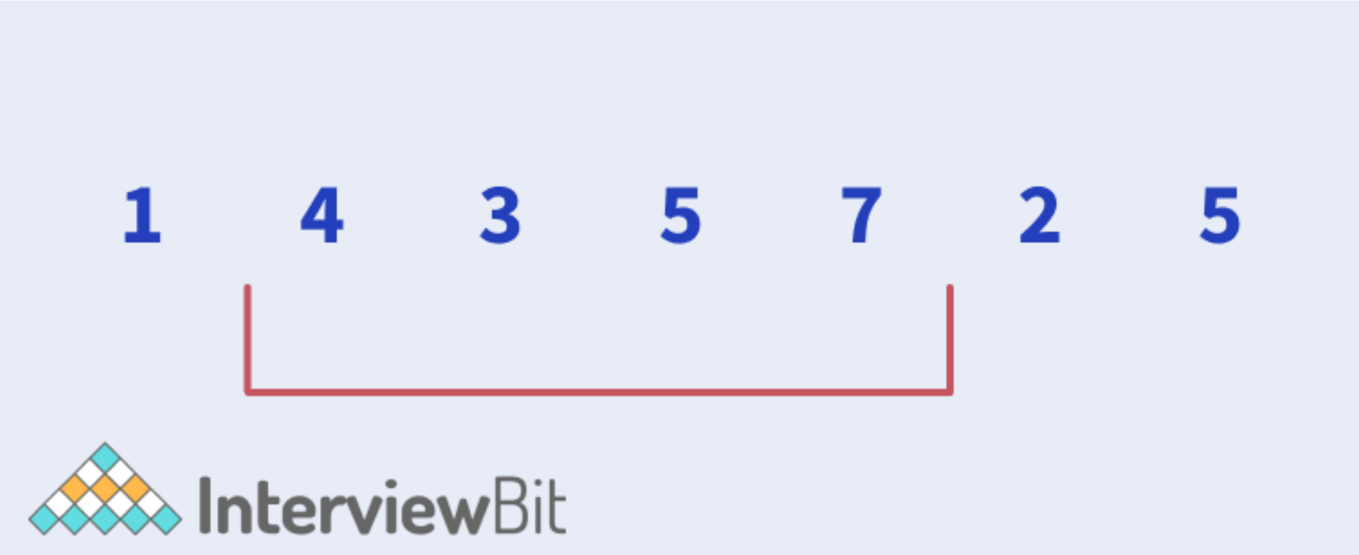


Table Of Contents

Problem Statement

Given an array **a[]**, find the number of subarrays in it, which have a sum of **k**.

Subarray: A subarray of an array is formed by deleting some(possibly zero) elements from the beginning or end of the array.



The red region shows a subarray of the original array.

Sample Test Cases

Input 1: a = [10, 2, -2, -20, 10], k = -10
Output 1: 3

Explanation 2: All subarrays of length 2 are valid subarrays in this case, and there are a total of 2 such subarrays.

Naive Approach

The naive approach is to generate all the subarrays of the array and calculate their sum. Whenever we find a subarray with a sum equal to **k,** we increment our counter by 1. Finally, we return the count which keeps track of the number of subarrays with a sum equal to **k**.

Since there are a total of **(n * (n + 1)) / 2** subarrays of an array, and each subarray will take **O(n)** time to traverse and calculate their sum, the required time complexity of this approach will be cubic in nature.

C++ Code

```
int countSubarraysWithSumK(vector < int > & a, int K) {
  int n = a.size();
  int count = 0;
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      int sum = 0;
      for (int k = i; k <= j; k++) {
```

Java Code

```
public static int countSubarraysWithSumK(int[] a, int K) {
  int n = a.length;
  int count = 0;
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
      int sum = 0;
      for (int k = i; k <= j; k++) {
        sum += a[k];
      }
      count += (sum == K ? 1 : 0);
    }
  }
  return count;
}
```

Python Code

```
def countSubarrayswithSumK(a, K):
    n = len(a)
    count = 0
```

Complexity Analysis

- Time Complexity: O(n^3)
- Space Complexity: O(1)

Optimal Approach

We can solve this problem in linear time complexity using a **hashmap-based** approach. The algorithm is described as follows:

- Traverse the array, and keep track of the current running sum up to the **ith** index in a variable, say **sum.**
- Also, hash the different values of the **sum** obtained so far, into a hashmap.
- If the **sum** equals **k** at any point in the array, increment the count of subarrays by 1.
- If this value of **sum** has exceeded **k** by a value of **sum – k**, we can find the number of subarrays, found so far with sum = **sum – k,** from our hashmap**.** Observe that if these subarrays are deleted from our current array, we will again obtain a sum of **k.** So, we add to our answer, the **number of subarrays with sum = sum – k found so far** from our hashmap.
- After traversing through the entire array once and applying the above steps, return the calculated result.

C++ Implementation

```
int countSubarraysWithSumK(vector < int > & a, int K) {
  int n = a.size();
  unordered_map < int, int > hash;
  int count = 0, sum = 0;
  for (int i = 0; i < n; i++) {
    sum += a[i];
    if (sum == K) {
      count++;
    }
    if (hash.find(sum - K) != hash.end()) {
      count += hash[sum - K];
    }
    hash[sum]++;
  }
  return count;
}
```

Java Implementation

```
public static int countSubarraysWithSumK(int[] a, int K) {
  int n = a.length;
```

```
      count += hash.get(sum - K);
    }
    if (hash.get(sum) != null) {
      hash.put(sum, hash.get(sum) + 1);
    } else {
      hash.put(sum, 1);
    }
  }
  return count;
}
```

Python Implementation

```
from collections import defaultdict


def countSubarrayswithSumK(a, K):
    n = len(a)
    hash = defaultdict(lambda: 0)
    count = 0
    sum = 0
    for i in range(n):
        sum += a[i]
        if sum == K:
            count += 1
```

- Time Complexity: O(n)
- Space Complexity: O(1)

Practice Problem

Subarray With Given XOR

FAQs

Q. What is the time complexity of lookup in a hashmap?
A. The time complexity of lookup in a hashmap is **O(1)** amortized.

Q. Why is the number of subarrays of an array given by (n * (n + 1)) / 2?
A. The number of subarrays of an array can be calculated as there are,

- 1 subarray of length n
- 2 subarrays of length n – 1
- ……
- n subarrays of length 1

So, the total number of subarrays count out to a total of **1 + 2 + 3 + … n = (n * (n + 1)) / 2.**

CODING PROBLEMS

Graph Coloring Problem

November 12, 2021

NEXT POST

CODING PROBLEMS

Reverse Words in a String

November 12, 2021

Categories

Applications

Architecture

Books

Careers

Characteristics

Coding Problems

Commands

Compare

Components

Courses

November 12, 2021

Libraries

Methodologies

Table Of Contents [show]

Problem Statement

Given a **sentence** of the form of words separated by spaces, return a new sentence that consists of the words of the original sentence in the reverse order.

Sample Test Cases

Input 1:

s = “Hello World”

Output 1:

World Hello

Input 2:

s = “This is a good day”

Output 2:

day good a is This

The naive approach for this problem is to **split** the string into individual words using the **spaces** as delimiters, and then print the words in reverse order.

```
  vector < string > words;
  string word = "";
  for (char c: s) {
    if (c == ' ') {
      words.push_back(word);
      word = "";
    } else {
      word += c;
    }
  }
  words.push_back(word);
  string ans = "";
  reverse(words.begin(), words.end());
  for (auto x: words) {
    ans += x;
    ans += " ";
  }
  return ans;
}
```

Java Code

```
public static void reverse(char[] ch, int left, int right) {
  while (left <= right) {
    char temp = ch[right];
  int beg = 0;
  for (int i = 0; i < ch.length; i++) {
    if (ch[i] == ' ') {
      reverse(ch, beg, i);
      beg = i + 1;
    }
  }
  reverse(ch, beg, ch.length - 1);
  reverse(ch, 0, ch.length - 1);
  String ans = Arrays.toString(ch);
  return ans;
}
```

Python Code

```
def reverse(s, left, right):
    while left <= right:
        s[left], s[right] = s[right], s[left]
        left += 1
        right -= 1


def reverseByWords(s):
    s = list(s)
    n = len(s)
    return s
```

Complexity Analysis:

- Time Complexity: O(n)
- Space Complexity: O(n)

Optimal Approach

The optimal approach tries to swap the words of the string from the beginning and end, using a **two-pointers-based** approach, to reverse the string in **constant space**. The algorithm is as follows:

- Convert the string into an array of strings, which will store the words.
- Initialize the 2 pointers **left** and **right** to **0** and **string.length() – 1** respectively.
- While the **left** pointer does not exceed the **right** pointer, swap the elements at the **left** and **right** pointer, move the **left** pointer forward and the **right** pointer backward by **1** place.
- Finally, return the final calculated string.

Implementation:

```
      words.push_back(str);
      str = "";
    } else {
      str += c;
    }
  }
  words.push_back(str);
  int left = 0, right = words.size() - 1;
  while (left <= right) {
    swap(words[left], words[right]);
    left++;
    right--;
  }
  string ans = "";
  for (auto x: words) {
    ans += x;
    ans += " ";
  }
  ans.pop_back();
  return ans;
}
```

Java Code

```
public static String reverseByWords(String s) {
  }
  String ans = String.join(" ", words);
  return ans;
}
```

Python Code

```
def reverseByWords(s):
    s = s.split(" ")
    left = 0
    right = len(s) - 1
    while left <= right:
        s[left], s[right] = s[right], s[left]
        left += 1
        right -= 1
    s = " ".join(s)
    return s
```

Complexity Analysis:

- Time Complexity: O(n)
- Space Complexity: O(1)

1. When a problem is asked to be solved in constant space, what should be the thought process?

A. While the idea may vary from problem to problem, **swapping** is a very common method used in problems requiring to be solved in constant space.

2. What is the time complexity of the swap function in C++?

A. The swap function in C++ works in O(1) time complexity.

Reverse Words in a String

PREVIOUS POST

CODING PROBLEMS

Subarray Sum Equals K

November 12, 2021

NEXT POST