Sliding Window Algorithm

Last modified: October 19, 2020

by Said Sryheni (https://www.baeldung.com/cs/author/saeedsryhini)

Networking (https://www.baeldung.com/cs/category/networking)

If you have a few years of experience in Computer Science or research, and you're interested in sharing that experience with the community, have a look at our **Contribution Guidelines** (/cs/contribution-guidelines).

1. Overview

When dealing with problems that require checking the answer of some ranges inside a given array, the sliding window algorithm can be a very powerful technique.

In this tutorial, we'll explain the sliding window technique with both its variants, the fixed and flexible window sizes. Also, we'll provide an example of both variants for better understanding.

2. Theoretical Idea

The main idea behind the sliding window technique is to convert two nested loops into a single loop. Usually, the technique helps us to reduce the time complexity from $O(n^2)$ to $O(n)$.

The condition to use the sliding window technique is that the problem asks to find the maximum (or minimum) value for a function that calculates the answer repeatedly for a set of ranges from the array. Namely, if these ranges can be sorted based on their start, and their end becomes sorted as well, then we can use the sliding window technique.

In other words, the following must hold:

If $L_i \geq L_j$ then $R_i \geq R_j$, where L_i and L_j are the left side of some ranges, and R_i and R_j are the left ends of the same ranges.

Basically, the technique lets us iterate over the array holding two pointers L and R. These pointers indicate the left and right ends of the current range. In each step, we either move L, R, or both of them to the next range.

In order to do this, we must be able to add elements to our current range when we move R forward. Also, we must be able to delete elements from our current range when moving L forward. Each time we reach a range, we calculate its answer from the elements we have inside the current range.

In case the length of the ranges is fixed, we call this the fixed-size sliding window technique. However, if the lengths of the ranges are changed, we call this the flexible window size technique. We'll provide examples of both of these options.

3. Fixed-Size Sliding Window

Let's look at an example to better understand this idea.

3.1. The Problem

Suppose the problem gives us an array of length n and a number k. **The problem asks us to find the maximum sum of k consecutive elements inside the array**.

In other words, first, we need to calculate the sum of all ranges of length k inside the array. After that, we must return the maximum sum among all the calculated sums.

3.2. Naive Approach

Let's take a look at the naive approach to solving this problem:

Algorithm 1: Naive Approach for maximum sum over ranges

```
Data: A: The array to calculate the answer for
      n: Length of the array
      k: Size of the ranges
Result: Returns the maximum sum among all ranges of length k
answer ← 0;
for L ← 1 to n − k + 1 do
    sum ← 0;
    for i ← L to L + k − 1 do
        sum ← sum + A[i];
    end
    answer ← maximum(answer, sum);
end
return answer;
```

First, we iterate over all the possible beginnings of the ranges. For each range, we iterate over its elements from L to $L + k - 1$ and calculate their sum. After each step, we update the best answer so far. Finally, the answer becomes the maximum between the old answer and the currently calculated sum.

In the end, we return the best answer we managed to find among all ranges.

The time complexity is $O(n^2)$ in the worst case, where n is the length of the array.

3.3. Sliding Window Algorithm

Let's try to improve on our naive approach to achieve a better complexity.

First, let's find the relation between every two consecutive ranges. The first range is obviously $[1, k]$. However, the second range will be $[2, k + 1]$.

We perform two operations to move from the first range to the second one: **The first operation is adding the element with index $k + 1$ to the answer**. **The second operation is removing the element with index 1 from the answer**.

Every time, after we calculate the answer to the corresponding range, we just maximize our calculated total answer.

Let's take a look at the solution to the described problem:

Algorithm 2: Sliding window technique for maximum sum over ranges

```
Data: A: The array to calculate the answer for
      n: Length of the array
      k: Size of the ranges
Result: Returns the maximum sum among all ranges of length k
sum ← 0;
for i ← 1 to k do
    sum ← sum + A[i];
end
answer ← sum;
for i ← k + 1 to n do
    sum ← sum + A[i] − A[i − k];
    answer ← maximum(answer, sum);
end
return answer;
```


Firstly, we calculate the sum for the first range which is $[1, k]$. Secondly, we store its sum as the answer so far.

After that, we iterate over the possible ends of the ranges that are inside the range $[k + 1, n]$. In each step, we update the sum of the current range. Hence, we add the value of the element at index i and delete the value of the element at index $i - k$.

Every time, we update the best answer we found so far to become the maximum between the original answer and the newly calculated sum. In the end, we return the best answer we found among all the ranges we tested.

The time complexity of the described approach is $O(n)$, where n is the length of the array.

4. Flexible-Size Sliding Window

We refer to the flexible-size sliding window technique as the two-pointers technique. We'll take an example of this technique to better explain it too.

4.1. Problem

Suppose we have n books aligned in a row. For each book, we know the number of minutes needed to read it. However, we only have k free minutes to read.

Also, we should read some consecutive books from the row. In other words, we can choose a range from the books in the row and read them. Of course, the condition is that the sum of time needed to read the books mustn't exceed k.

Therefore, the problem asks us to find the maximum number of books we can read. Namely, **we need to find a range from the array whose sum is at most** k **such that this range's length is the maximum possible**.

4.2. Naive Approach

Take a look at the naive approach for solving the problem:

Algorithm 3: Naive approach to maximum length range

```
Data: A: The array of time needed to read each book
      n: Length of the array
      k: Maximum number of minutes to read
Result: Returns the maximum length of a range whose sum is at
        most k
answer ← 0;
for L ← 1 to n do
    sum ← 0;
    i ← L;
    length ← 0;
    while i ≤ n AND sum + A[i] ≤ k do
        sum ← sum + A[i];
        length ← length + 1;
        i ← i + 1;
    end
    answer ← maximum(answer, length);
end
return answer;
```

First, we initialize the best answer so far with zero. Next, we iterate over all the possible beginnings of the range. For each beginning, we iterate forward as long as we can read more books. Once we can't read any more books, we update the best answer so far as the maximum between the old one and the length of the range we found.

In the end, we return the best answer we managed to find.

The complexity of this approach is $O(n^2)$, where n is the length of the array of books.

4.3. Sliding Window Algorithm

We'll try to improve the naive approach, in order to get a linear complexity.

First, let's assume we managed to find the answer for the range that starts at the beginning of the array. The next range starts from the second index inside the array. However, the end of the second range is surely after the end of the first range.

The reason for this is that the second range doesn't use the first element. Therefore, the second range can further extend its end since it has more free time now to use.

Therefore, when moving from one range to the other, we first delete the old beginning from the current answer. Also, we try to extend the end of the current range as far as we can.

Hence, by the end, we'll iterate over all possible ranges and store the best answer we found.

The following algorithm corresponds to the explained idea:

Algorithm 4: Sliding window approach to maximum length range

```
Data: A: The array of time needed to read each book
      n: Length of the array
      k: Maximum number of minutes to be used
Result: Returns the maximum length of a range whose sum is at
        most k
answer ← 0;
sum ← 0;
R ← 1;
for L ← 1 to n do
    if L > 1 then
        sum ← sum − A[L − 1];
    end
    while R ≤ n AND sum + A[R] ≤ k do
        sum ← sum + A[R];
        R ← R + 1;
    end
    answer ← maximum(answer, R − L);
end
return answer;
```

Just as with the naive approach, we iterate over all the possible beginnings of the range. For each beginning, we'll first subtract the value of the index $L - 1$ from the current sum.

After that, we'll try to move R as far as possible. Therefore, we continue to move R as long as the sum is still at most k. Finally, we update the best answer so far. Since the length of the current range is $R - L$, we maximize the best answer with this value.

Although the algorithm may seem to have a $O(n^2)$ complexity, let's examine the algorithm carefully. The variable R always keeps its value. Therefore, it only moves forward until it reaches the value of n. Therefore, the number of times we execute the *while* loop in total is at most n times.

Hence, the complexity of the described approach is $O(n)$, where n is the length of the array.

5. Differences

The main difference comes from the fact that in some problems we are asked to check a certain property among all range of the same size. On the other hand, on some other problems, we are asked to check a certain property among all ranges who satisfy a certain condition. In these cases, this condition could make the ranges vary in their length.

In case these ranges had an already known size (like our consecutive elements problem), we'll certainly go with the fixed-size sliding window technique. However, if the sizes of the ranges were different (like our book-length problem), we'll certainly go with the flexible-size sliding window technique.

Also, always keep in mind the following condition to use the sliding window technique that we covered in the beginning: We must guarantee that moving the L pointer forward will certainly make us either keep R in its place or move it forward as well.

6. Conclusion

In this tutorial, we explained the sliding window approach. We provided the theoretical idea for the technique. Also, we described two examples of the fixed-size and flexible-size sliding window technique. Finally, we explained when to use each technique.