**Sliding Window: Introduction**

Let’s go over the Sliding Window pattern, its real-world applications, and some problems we can solve with it.

**We'll cover the following**

* [Overview](https://www.educative.io/courses/grokking-coding-interview-patterns-java/sliding-window-introduction#Overview)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/sliding-window-introduction#Examples)
* [Does my problem match this pattern?](https://www.educative.io/courses/grokking-coding-interview-patterns-java/sliding-window-introduction#Does-my-problem-match-this-pattern)
* [Real-world problems](https://www.educative.io/courses/grokking-coding-interview-patterns-java/sliding-window-introduction#Real-world-problems)
* [Strategy time!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/sliding-window-introduction#Strategy-time)

**Overview**

The **sliding window** pattern is used to process sequential data by maintaining a moving subset of elements, called a window. The pattern is aimed at reducing the use of nested loops in an algorithm. It may be viewed as a variation of the two pointers pattern, with the pointers being used to set the window bounds.

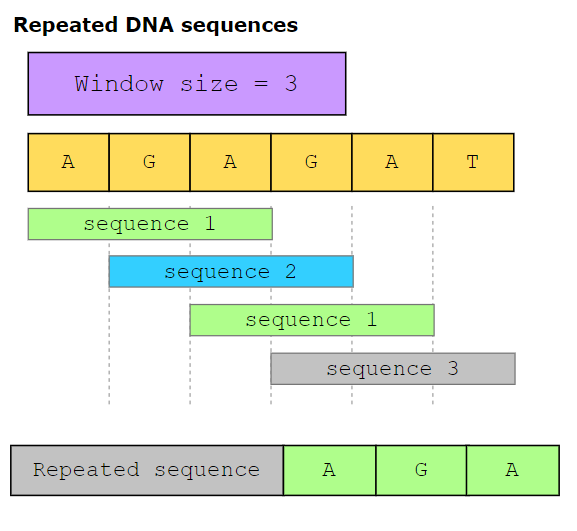
A window is a sublist formed over a part of an iterable data structure. It can be used to slide over the data in chunks corresponding to the window size. The sliding window pattern allows us to process the data in segments instead of the entire list. The segment or window size can be set according to the problem’s requirements. For example, if we have to find three consecutive integers with the largest sum in an array, we can set the window size to 33. This will allow us to process the data three elements at a time.

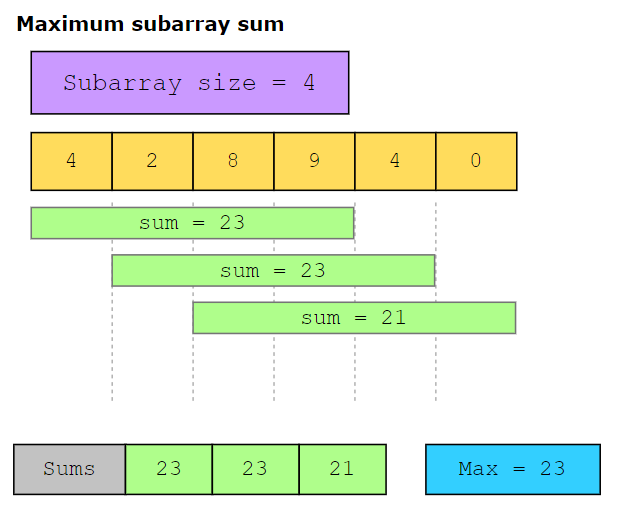
Why is this method more efficient? It isn’t if, for each window, we iterate over all the elements of the window because that gives us the same *O*(*kn*) time complexity.

Instead, what if we focused on the element entering the window and the one leaving it? For example, after calculating the sum of the first three elements, we move the window one step forward, subtract the element that is no longer in the window from the sum, and add the new element that has entered it. Next we check if the new sum is greater than the first. If it is, we update the max sum found so far. Now, each time we move the window forward, we perform at most four operations, reducing the time complexity to *O*(4*n*), that is, *O*(*n*).

## Examples

The following examples illustrate some problems that can be solved with this approach:





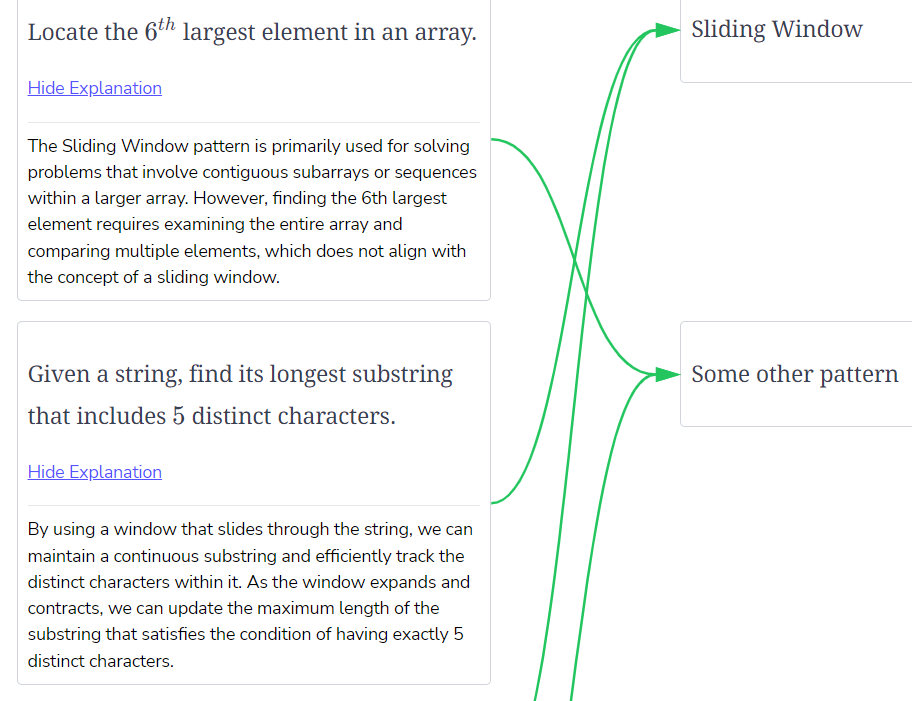
**Does my problem match this pattern?**

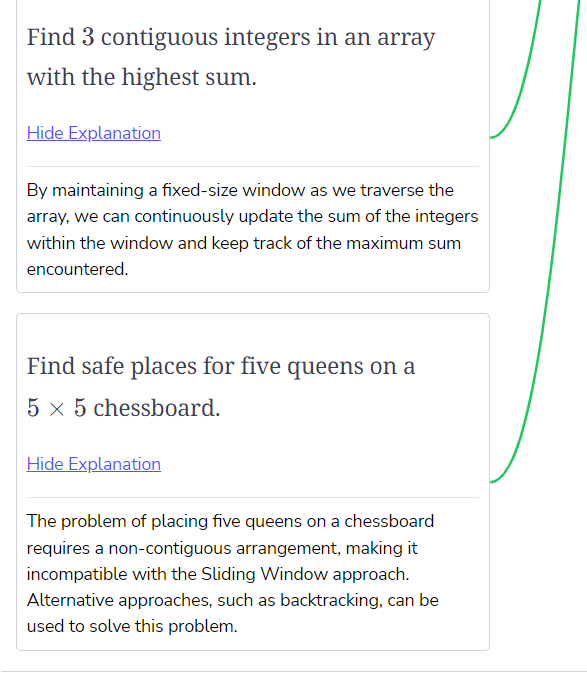
* Yes, if both these conditions are fulfilled:
  + The problem requires repeated computations on a contiguous set of data elements (a subarray or a substring), such that the window moves across the input array from one end to the other. The size of the window may be fixed or variable, depending on the requirements of the problem. The repeated computations may be a direct part of the final solution, or they may be intermediate steps building up towards the final solution.
  + The computations performed every time the window moves take *O*(1) time or are a slow-growing function, such as log, of a small variable, say *k*, where *k*≪*n*.
* No, if either of these conditions are fulfilled:
  + The input data structure does not support random access.
  + You have to process the entire data without segmentation.

**Real-world problems**

Many problems in the real world use the sliding window pattern. Let’s look at some examples.

* **Telecommunications:** Find the maximum number of users connected to a cellular network’s base station in every *k*-millisecond sliding window.
* **E-commerce:** Given a dataset of product IDs in the order they were viewed by the user and a list of products that are likely to be similar, find how many times these products occur together in the dataset.
* **Video streaming:** Given a stream of numbers representing the number of buffering events in a given user session, calculate the median number of buffering events in each one-minute interval.
* **Social media content mining:** Given the lists of topics that two users have posted about, find the shortest sequence of posts by one user that includes all the topics that the other user has posted about.





**Repeated DNA Sequences**

Try to solve the Repeated DNA Sequences problem.

**We'll cover the following**

* [Statement](https://www.educative.io/courses/grokking-coding-interview-patterns-java/repeated-dna-sequences#Statement)
* [Examples](https://www.educative.io/courses/grokking-coding-interview-patterns-java/repeated-dna-sequences#Examples)
* [Understand the problem](https://www.educative.io/courses/grokking-coding-interview-patterns-java/repeated-dna-sequences#Understand-the-problem)
* [Figure it out!](https://www.educative.io/courses/grokking-coding-interview-patterns-java/repeated-dna-sequences#Figure-it-out)
* [Try it yourself](https://www.educative.io/courses/grokking-coding-interview-patterns-java/repeated-dna-sequences#Try-it-yourself)

**Statement**

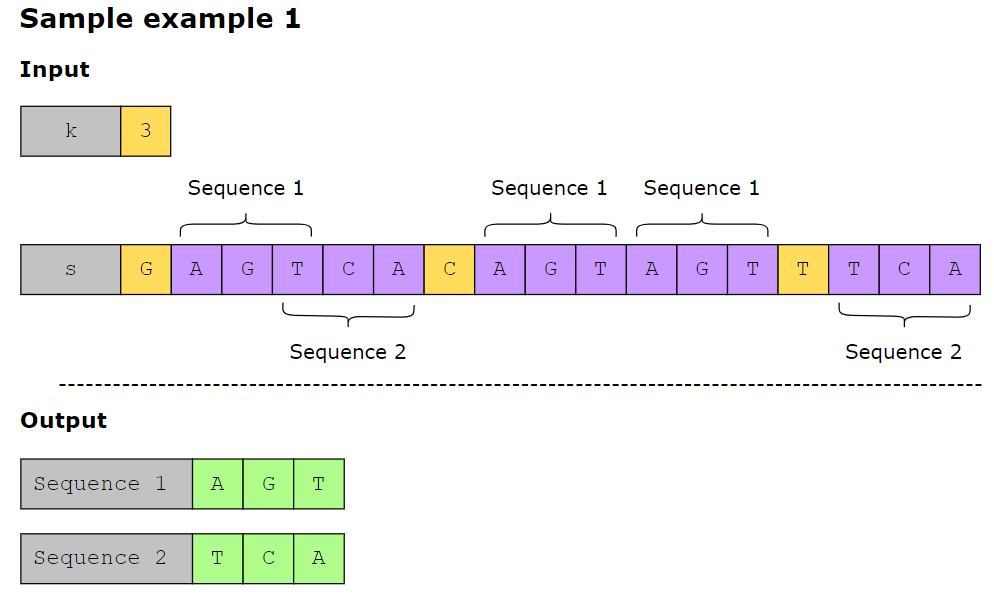
Given a string, s, that represents a DNA subsequence, and a number *k*, return all the contiguous subsequences (substrings) of length *k* that occur more than once in the string. The order of the returned subsequences does not matter. If no repeated substring is found, the function should return an empty set.

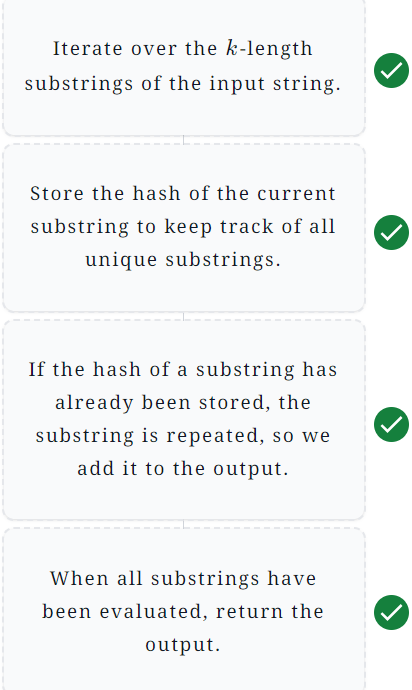
The DNA sequence is composed of a series of nucleotides abbreviated as *A*, *C*, *G*, and *T*. For example, *ACGAATTCCG* is a DNA sequence. When studying DNA, it is useful to identify repeated sequences in it.

**Constraints:**

* 1 ≤≤ s.length ≤≤ 10^4
* s[i] is either *A*, *C*, *G*, or *T*.
* 1≤*k*≤10

**Examples**[#](https://www.educative.io/courses/grokking-coding-interview-patterns-java/repeated-dna-sequences#Examples)





# Solution: Repeated DNA Sequences

#### Solution summary

To recap, the solution to this problem can be divided into the following six main parts:

1. Iterate over all *k*-length substrings.
2. Compute the hash value for the contents of the window.
3. Add this hash value to the set that keeps track of the hashes of all substrings of the given length.
4. Move the window one step forward and compute the hash of the new window using the rolling hash method.
5. If the hash value of a window has already been seen, the sequence in this window is repeated, so we add it to the output array.
6. Once all substrings have been evaluated, return the output array.

#### Time complexity

**Average case**

The average case time complexity of this solution is *O*(*n*), where *n* is the length of the input string. It is calculated as follows:

* Time taken to populate the numbers array: *O*(*n*).
* Time taken to traverse all the *k*-length substrings:*O*(*n*−*k*+1).
* Time taken to calculate the hash value of a *k*-length substring: *O*(1).

So, the dominating time complexity becomes *O*(*n*).

**Worst case**

To understand the worst case time complexity of this solution, consider the input string “AAAAAAAA” with *k*=2. This combination of inputs ensures that a repeated sequence “AA” is detected and added to the output each time the window slides forward. Therefore, we must generate a *k*-length substring on each (*n*−*k*+1) iteration of the loop. The time to generate a *k*-length substring is *O*(*k*). Therefore, the overall time complexity becomes *O*((*n*−*k*)×*k*).

#### Space complexity

The space complexity of this solution is *O*(*n*). It is calculated as follows:

* Space occupied by the mapping hash map: *O*(1).
* Space occupied by the numbers array: *O*(*n*).
* Space occupied by the hashSet set: *O*(*n*−*k*+1).

So, the dominating space complexity becomes *O*(*n*).

class RepeatedDNA {

public static Set<String> findRepeatedSequences(String s, int k) {

int n = s.length();

if (n < k) {

return new HashSet<>();

}

Map<Character, Integer> mapping = new HashMap<>();

mapping.put('A', 1);

mapping.put('C', 2);

mapping.put('G', 3);

mapping.put('T', 4);

int a = 4;

List<Integer> numbers = new ArrayList<>(Arrays.asList(new Integer[n]));

Arrays.fill(numbers.toArray(), 0);

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

numbers.set(i, mapping.get(s.charAt(i)));

}

int hashValue = 0;

Set<Integer> hashSet = new HashSet<>();

Set<String> output = new HashSet<>();

for (int i = 0; i < n - k + 1; i++) {

if (i == 0) {

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

hashValue += numbers.get(j) \* (int) Math.pow(a, k - j - 1);

}

} else {

int previousHashValue = hashValue;

hashValue = ((previousHashValue - numbers.get(i - 1) \* (int) Math.pow(a, k - 1)) \* a) + numbers.get(i + k - 1);

}

if (hashSet.contains(hashValue)) {

output.add(s.substring(i, i + k));

}

hashSet.add(hashValue);

}

return output;

}

// Driver code

public static void main(String[] args) {

List<String> inputsString = Arrays.asList("ACGT", "AGACCTAGAC", "AAAAACCCCCAAAAACCCCCC",

"GGGGGGGGGGGGGGGGGGGGGGGGG", "TTTTTCCCCCCCTTTTTTCCCCCCCTTTTTTT", "TTTTTGGGTTTTCCA",

"AAAAAACCCCCCCAAAAAAAACCCCCCCTG", "ATATATATATATATAT");

List<Integer> inputsK = Arrays.asList(3, 3, 8, 12, 10, 14, 10, 6);

for (int i = 0; i < inputsK.size(); i++) {

System.out.println((i + 1) + ".\tInput sequence: " + inputsString.get(i) +

"\n\tk: " + inputsK.get(i) +

"\n\n\tRepeated sequences: " + Print.printSetString(findRepeatedSequences(inputsString.get(i), inputsK.get(i))));

System.out.println(Print.repeat("-", 100));

}

}

}