Solution: Repeated DNA Sequences

Let's solve the Repeated DNA Sequences problem using the Sliding Window pattern.

We'll cover the following

- Statement
- Solution
  - Naive approach
  - Optimized approach using sliding window
    - Hashing and comparison in linear time
    - Hashing and comparison in constant time
    - Step-by-step solution construction
    - Just the code
    - Solution summary
    - Time complexity
    - Space complexity

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 subsequences in it.

Constraints:

- $1 \leq$ `s.length` $\leq 10^4$
- `s[i]` is either *A*, *C*, *G*, or *T*.

Solution

So far, you've probably brainstormed some approaches on how to solve this problem. Let's explore some of these approaches and figure out which one to follow while considering time complexity and any implementation constraints.

Naive approach

A naive approach would be to iterate through the input DNA sequence and add all the unique substrings of length k to a set. If a substring is already present in a set, it is a repeated substring.

Here's how the algorithm works:

- We iterate the string using a pointer i, ranging from 0 to $(n - k + 1)$. This is the number of k-length substrings present in the sequence.
- At each iteration, we generate the current k-length substring, i.e., **s[i]...s[i + k - 1]**.
- Next, we check if this substring is already present in the set.
    - If it is, the current substring is a repeated sequence, so we add it to our output.
    - Otherwise, the current substring has not yet been repeated, so we just add it to the set.
- We repeat the above process for all k-length substrings.
- Once all k-length substrings have been evaluated, we return the output.

The time complexity of this approach is $O((n - k) \times k)$, where n is the length of the input sequence and k is the size of each contiguous subsequence we consider. This is because we iterate over $(n - k + 1)$ substrings of length k, and at each iteration, the time taken to generate a k-length substring is $O(k)$.

The space complexity of this approach is $O((n - k) \times k)$, since in the worst case, our set can contain $(n - k + 1)$ elements, and at each iteration of the traversal, we are allocating memory to generate a new k-length substring.

Optimized approach using sliding window

The problem can be optimized using a sliding window approach. We use the Rabin-Karp algorithm that utilizes a sliding window with **rolling hash** for pattern matching.

Here's the basic idea of the algorithm:

- We traverse the string by using a sliding window of length k, which slides one character forward on each iteration.
- On each iteration, we compute the hash of the current k-length substring in the window.
- We check if the hash is already present in the set.
    - If it is, the substring is repeated, so we add it to the output.
    - Otherwise, the substring has not yet been repeated, so we add the computed hash to the set.
- We repeat the above process for all k-length substrings by sliding the window one character forward on each iteration.
- After all k-length substrings have been evaluated, we return the output.

There are multiple approaches for computing hash values, and the choice of the hash function can impact the algorithm's time complexity. Let's look at some approaches below.

Hashing and comparison in linear time

Let's use a simple hashing method that sums the ASCII code of characters present in a window.

Consider the sequence $ACTCT$ with $k = 2$.

1. Initially, the sequence in the window is AC and its hash value is:

    $$H(AC) = 65 + 67 = 132$$

    Since the above hash value has not been repeated yet, we add this hash value to the set and slide the window one character forward.

2. The sequence in the window is now CT. To compute the hash value of CT, the ASCII of A will be removed from the previous hash value and the ASCII of T will then be added:

$$H(CT) = 132 - 65 + 84 = 151$$

Since the above hash value has not been repeated yet, we add this hash value to the set and slide the window one character forward.

3. The sequence in the window is now TC. To compute the hash value of TC, the ASCII of C will be removed from the previous hash value and then again added:

$$H(TC) = 151 - 67 + 67 = 151$$

Here, we have the same hash value but different sequences—CT and TC. This means that if a hash value is already present in the set, we need to compare the corresponding sequences as well to confirm if they are identical. In this case, they are not, so we add this hash value to the set and slide the window one character forward.

4. The sequence in the window is now CT. To compute the hash value of CT, the ASCII of T will be removed from the previous hash value and then again added:

$$H(CT) = 151 - 84 + 84 = 151$$

Here we have the same hash value, so we compare the two sequences. Since they are identical, the sequence has been repeated and is therefore added to the output.

Computing the hash value and then comparing the strings if the hashes are equal will take linear time, $O(k)$. In the worst case, the comparisons will occur after each slide, which will make the running time the same as that of the naive approach, which is $O((n - k + 1) \times k)$.

Hashing and comparison in constant time

We need a hash function that helps us achieve constant-time hashing.For this purpose, we use the **polynomial rolling hash** technique:

$$H = c_1a^{k-1} + c_2a^{k-2} + ... + c_ia^{k-i} + ... + c_{k-1}a^1 + c_ka^0$$

Here, a is a constant, $c_1, \ldots, c_k$ are the characters in a sequence, and k is the substring length. Since we only have 4 possible nucleotides, our a would be 4. We also assign numeric values to the nucleotides, as shown in the table below:

A	C	G	T
1	2	3	4

> **Note:** We use a value of 4 for the constant a since this ensures that each nucleotide is assigned a unique value in the polynomial hash function. This reduces the number of hash collisions and helps reduce the randomness in the behaviour of the hash function.
>
> It is not necessary to set the constant a equal to 4. In fact, this choice is somewhat arbitrary and can be any number that is greater than or equal to the number of nucleotides. The idea is to choose a value that is large enough to avoid hash collisions while still being relatively small, so as to minimize the risk of arithmetic overflow.

> **Example:** The polynomial hash value for the sequence ATG will be
>
> $$H(ATG) = (1 \times 4^2) + (4 \times 4^1) + (3 \times 4^0) = 35$$

Consider the same sequence $ACTCT$ with $k = 2$.

1. Initially, the sequence in the window is AC, and its hash value is:

   $$H(AC) = H(A) + H(C)$$

   $$= (1 \times 4^1) + (2 \times 4^0)$$

   $$= 6$$

   Since the above hash value has not been repeated yet, we add this hash value in the set and slide the window one character forward.

2. The sequence in the window is now CT. To compute the hash value of CT, we first need to remove the contribution of A from the previous hash value:

   $$H(C) = H(AC) - H(A) =$$

   $$= 6 - (1 \times 4^1)$$

   $$= 2$$

   and then add the contribution of T:

   $$H(CT) = H(C) + H(T)$$

   $$= 2 + (4 \times 4^0)$$

   $$= 6$$

   This can't be right, since both AC and CT cannot yield the same hash value. Examining our work carefully, we notice that we are not correctly accounting for place values. So, we need to shift the remaining bases to the left by one position so that the hash corresponds to the new sliding window. We do this by multiplying the current hash value by the base value $a = 4$. This means that $H(C)$ becomes 2×4^1 and $H(T)$ becomes 4×4^0. After that, we add the contribution of the incoming character T to get the new hash value for the current sliding window:

   $$H(CT) = [(H(AC) - H(A)) \times 4] + H(T)$$

   $$H(CT) = [6 - (1 \times 4^1)] \times 4 + (4 \times 4^0)$$

   $$= 12$$

   Since this hash value has not been seen yet, we add it to the set and slide the window one character forward.

3. The sequence in the window is now TC. We compute the hash value of TC in the same way as explained above:

   $$H(TC) = ([H(CT) - H(C)] \times 4) + H(C)$$

$$= ([H(CT) - (2 \times 4)] \times 4) + H(C)$$

$$= ([12 - 8] \times 4) + 2$$

$$= (4 \times 4) + 2$$

$$= 16 + 2 = 18$$

Since this hash value has not been seen yet, we add it to the set and slide the window one character forward. As you can see, even though CT and TC contain the same characters, since their relative order is different, the respective hash values of the two strings are also different.

4. The sequence in the window is now CT. We compute the hash value of CT in the same way as explained above:

$$H(CT) = [(H(TC) - H(T)) \times 4] + H(T)$$

$$= [(18 - 16) \times 4] + 4$$

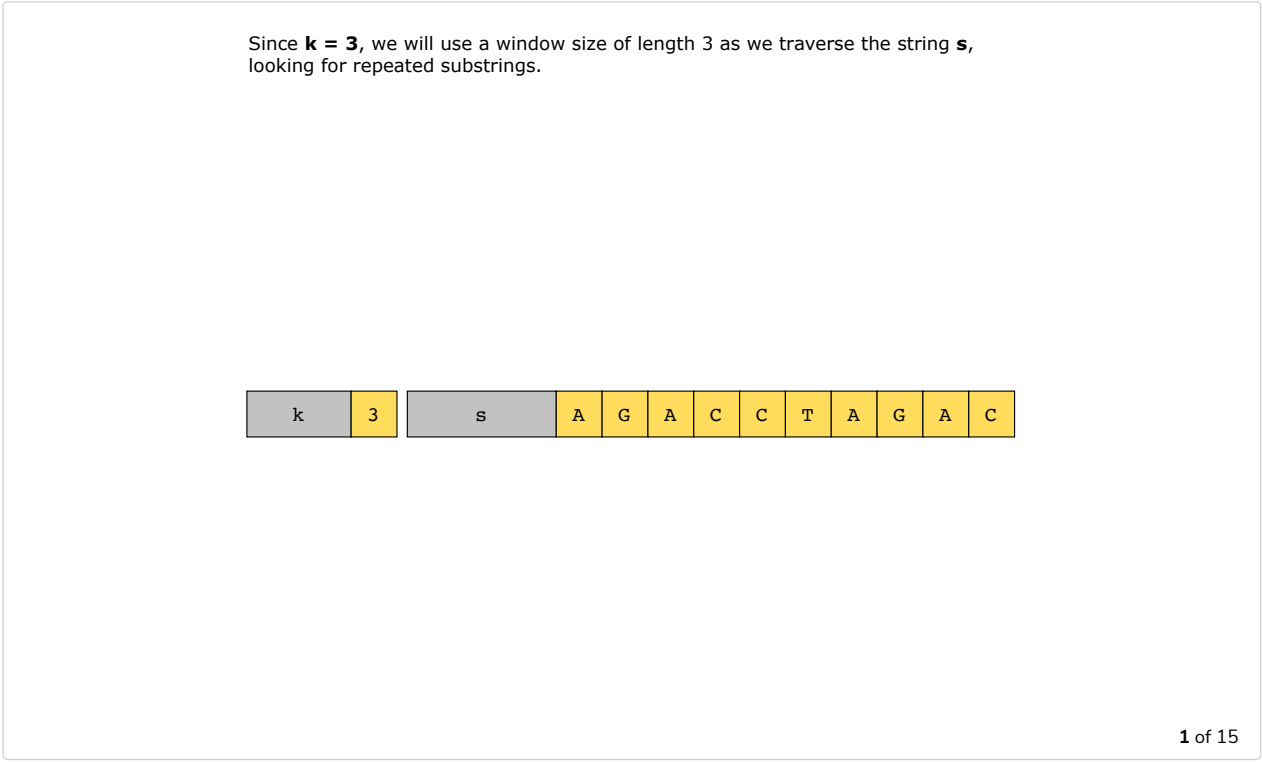
$$= [2 \times 4] + 4$$

$$= 12$$

This hash value is present in the set. Therefore, we add it to the output without checking if the sequences are identical.

From the above two approaches, it is clear that we will be using the polynomial rolling hash function in our solution to compute the hash values of the k-length substrings since this method skips the need to compare two sequences if their hash values are the same and, therefore, helps us achieve constant-time hashing and comparison.

The slides below illustrate how the algorithm runs:



> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

We will first define the following variables to set ourselves up to implement the polynomial rolling hash function:

- `mapping`: This is a hash map defining the numeric mapping of nucleotides. It will be of the form: $\{A:1, C:2, G:3, T:4\}$.
- `a`: This is the base value used in the polynomial hash function. We will use a base value of 4 since there are 4 nucleotides in the sequence.
- `numbers`: This is an array storing the integer form of the string `s` based on the mapping defined above. For example, for the sequence $ACTCT$, `numbers` will consist of $[1, 2, 4, 2, 4]$. This array will make it easier to access the numeric value of the nucleotides when calculating the hash value.

Java

```java
class RepeatedDNA {

    public static void findRepeatedSequences(String s, int k) {

        int n = s.length();

        // Mapping of characters
        Map<Character, Integer> mapping = new HashMap<>();
        mapping.put('A', 1);
        mapping.put('C', 2);
        mapping.put('G', 3);
        mapping.put('T', 4);

        // Base value
        int a = 4;

        // Numeric representation of the sequence
        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)));
        }

        System.out.println("\tConverted sequence: " + numbers.toString());
      }

    // Driver code
    public static void main(String[] args) {
```

Repeated DNA Sequences

We declare a variable, `hashValue`, to store the hash value of the current k-length sequence in the window. It is initialized to 0.

Next, we slide the window along the string, `s`, using a pointer, `i`, ranging from 0 to $(n - k + 1)$:

- When `i` is 0, the window is at its starting position, i.e., the first k-length substring. For this sequence, we calculate the hash value from scratch using the above-mentioned polynomial hash function.

```
for (int j = 0; j < k; j++) {
    hashValue += numbers.get(j) * (int) Math.pow(a, k - j - 1);
```

```
}
```

- Otherwise, the window is not at its starting position. So, we calculate the hash value of the current k-length substring by utilizing the hash value of the previous k-length substring:

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

- The above process is repeated by sliding the window one character forward.

Java

```
class RepeatedDNA {

    public static void findRepeatedSequences(String s, int k) {

        int n = s.length();

        // Mapping of characters
        Map<Character, Integer> mapping = new HashMap<>();
        mapping.put('A', 1);
        mapping.put('C', 2);
        mapping.put('G', 3);
        mapping.put('T', 4);

        // Base value
        int a = 4;

        // Numeric representation of the sequence
        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)));
        }

        // Current hash value
        int hashValue = 0;

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

Repeated DNA Sequences

We declare the following variables to keep track of the hash values and store the repeated substrings:

- `hashSet`: This is a set that stores all the unique hash values of the k-length substrings. It is initialized to empty.
- `output`: This is a set that stores the repeated substrings.

We check if the calculated hash value of the current k-length substring is present in `hashSet`:

- If it is, the substring is repeated, so it is added to `output`:

```
if (hashSet.contains(hashValue)) {
    output.add(s.substring(i, i + k));
}
```

- Otherwise, we will just add the hash value of the substring to `hashSet`:

```
hashSet.add(hashValue)
```

When the hash values of all k-length substrings have been evaluated, i.e., the sliding window can not move forward, we return `output`.

Java

```java
class RepeatedDNA {

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

        int n = s.length();

        // Mapping of characters
        Map<Character, Integer> mapping = new HashMap<>();
        mapping.put('A', 1);
        mapping.put('C', 2);
        mapping.put('G', 3);
        mapping.put('T', 4);

        // Base value
        int a = 4;

        // Numeric representation of the sequence
        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)));
        }

        // Current hash value
        int hashValue = 0;

        // 1. Hash set to store the unique hash values
        // 2. Output set to store the repeated substrings
```

Repeated DNA Sequences

The above solution works for most inputs. However, it will not work if the length of the string, `s`, is less than the window size, `k`. So, we need to handle this case by returning an empty array.

Java

```java
class RepeatedDNA {

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

        int n = s.length();

        if (n < k) {
            return new HashSet<>();
        }

        // Mapping of characters
        Map<Character, Integer> mapping = new HashMap<>();
        mapping.put('A', 1);
        mapping.put('C', 2);
        mapping.put('G', 3);
        mapping.put('T', 4);

        // Base value
        int a = 4;

        // Numeric representation of the sequence
        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)));
        }

        // Current hash value
```

Just the code

Here's the complete solution to this problem:

Java

```java
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<>();
```

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 is already present in the set, add the substring to the output array. Otherwise, add it to the output array.
6. Once all substrings have been evaluated, return the output array.

Time complexity

The time complexity of this solution is $O(n - k)$, where n is the length of the input string, and k is the length of the repeated substrings.

Explanation:

- 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 overall time complexity becomes $O(2n - k + 1)$ which simplifies to $O(n - k)$.

Space complexity

The space complexity of this solution is $O(n - k)$.

Explanation:

- 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 overall space complexity becomes $O(2n - k + 1)$, which simplifies to $O(n - k)$.

Back
Repeated DNA Seque...

Next
Find Maximum in Slidi...