187. Repeated DNA Sequences

Sept. 10, 2019 | 10.4K views

Previous Next

Average Rating: 4.75 (16 votes)

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

Example:

```
Input: s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"

Output: ["AAAAACCCCC", "CCCCCAAAAA"]
```

Solution

Overview

Follow-up here is to solve the same problem for arbitrary sequence length L, and to check the situation when L is quite large. Hence let's use $L = 10$ notation everywhere to ease the problem generalisation.

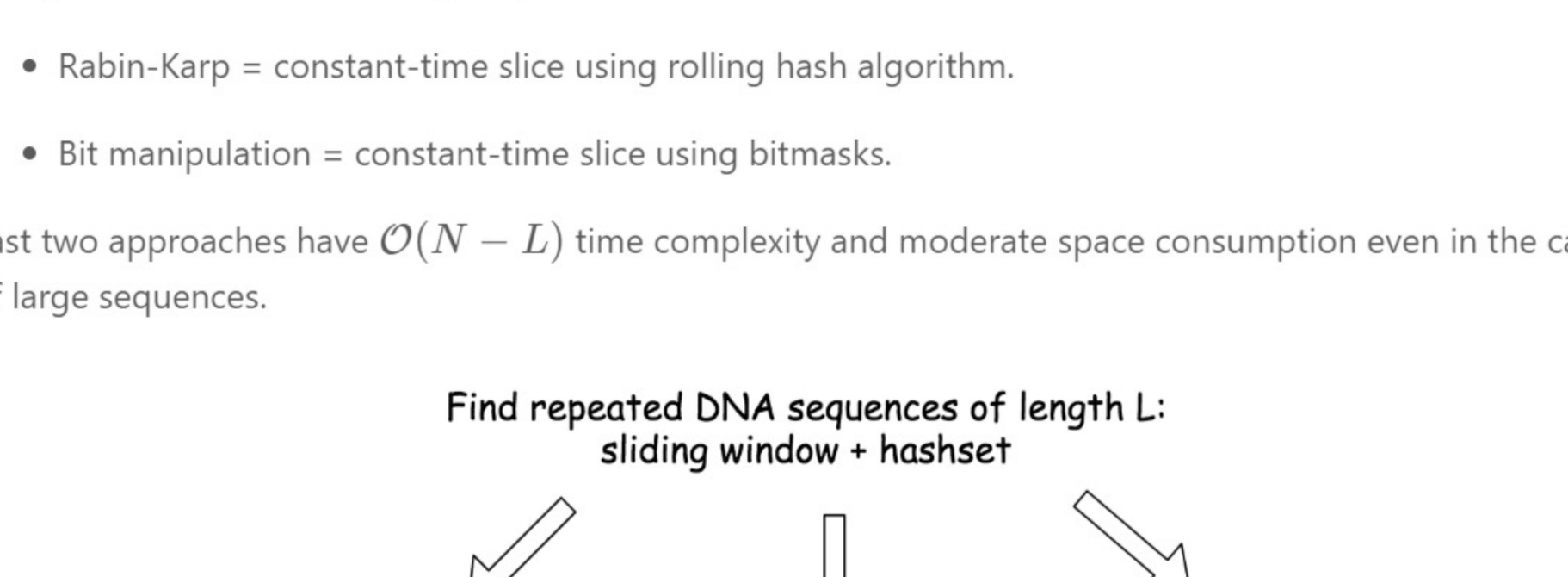
> We will discuss three different ideas how to proceed. They are all based on sliding window + hashset. The key point is how to implement a window slice.

Linear-time window slice $\mathcal{O}(L)$ is easy stupid, just take a substring. Overall that would result in $\mathcal{O}((N - L)L)$ time complexity and huge space consumption in the case of large sequences.

Constant-time slice $\mathcal{O}(1)$ is where the fun starts, because the way you choose will show your actual background. There are two ways to proceed:

- Rabin-Karp = constant-time slice using rolling hash algorithm.
- Bit manipulation = constant-time slice using bitmasks.

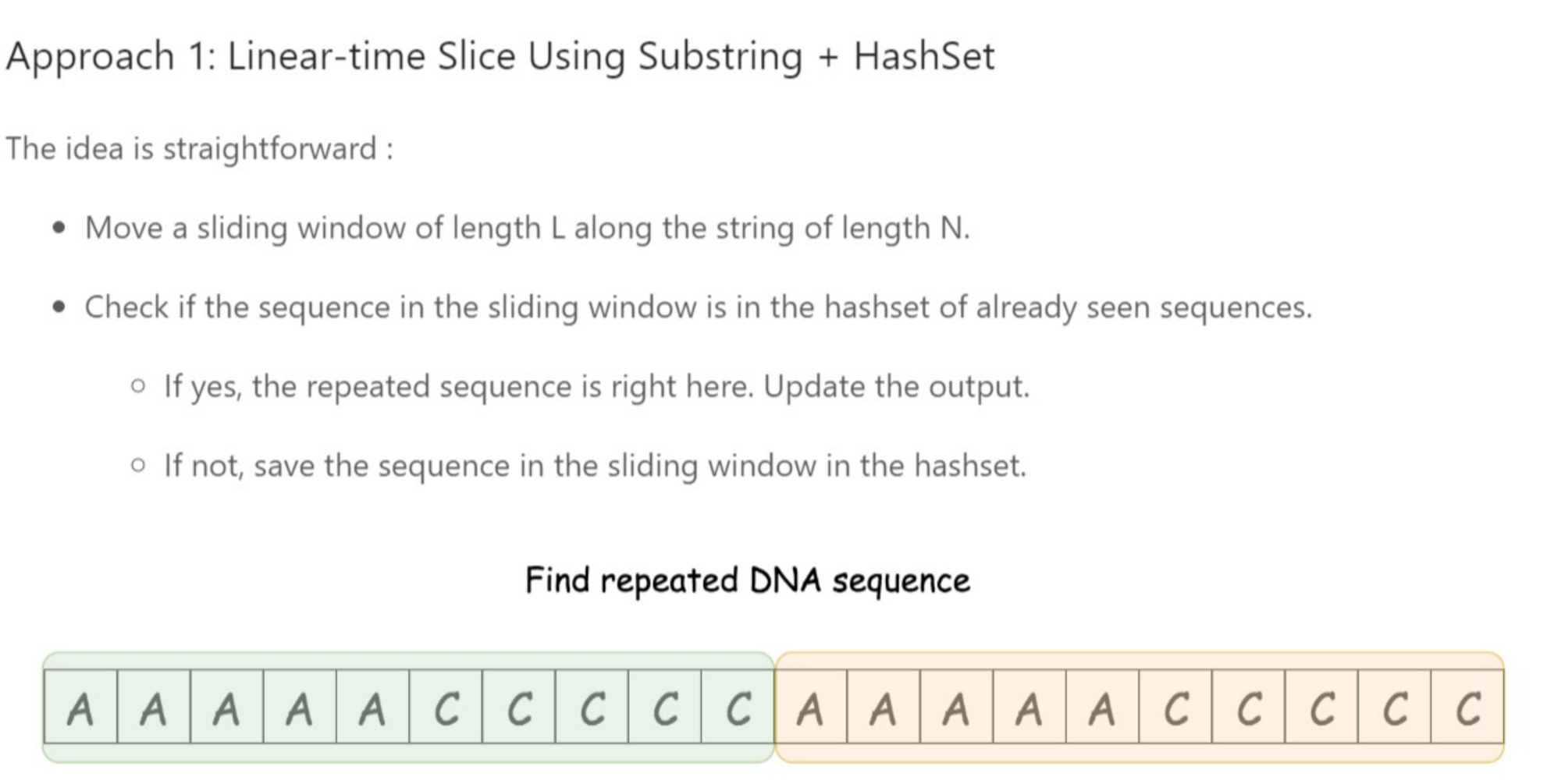
Last two approaches have $\mathcal{O}(N - L)$ time complexity and moderate space consumption even in the case of large sequences.



Approach 1: Linear-time Slice Using Substring + HashSet

The idea is straightforward :

- Move a sliding window of length L along the string of length N.
- Check if the sequence in the sliding window is in the hashset of already seen sequences.
  - If yes, the repeated sequence is right here. Update the output.
  - If not, save the sequence in the sliding window in the hashset.



Java | Python | Copy

```java
class Solution {
  public List<String> findRepeatedDnaSequences(String s) {
    int L = 10, n = s.length();
    HashSet<String> seen = new HashSet(), output = new HashSet();

    // iterate over all sequences of length L
    for (int start = 0; start < n - L + 1; ++start) {
      String tmp = s.substring(start, start + L);
      if (seen.contains(tmp)) output.add(tmp);
      seen.add(tmp);
    }
    return new ArrayList<String>(output);
  }
}
```

Complexity Analysis

- Time complexity : $\mathcal{O}((N - L)L)$, that is $\mathcal{O}(N)$ for the constant $L = 10$. In the loop executed $N - L + 1$ one builds a substring of length L. Overall that results in $\mathcal{O}((N - L)L)$ time complexity.
- Space complexity : $\mathcal{O}((N - L)L)$ to keep the hashset, that results in $\mathcal{O}(N)$ for the constant $L = 10$.

Approach 2: Rabin-Karp : Constant-time Slice Using Rolling Hash

Rabin-Karp algorithm is used to perform a multiple pattern search. It's used for plagiarism detection and in bioinformatics to look for similarities in two or more proteins.

Detailed implementation of Rabin-Karp algorithm for quite a complex case you could find in the article Longest Duplicate Substring, here we do a very basic implementation.

> The idea is to slice over the string and to compute the hash of the sequence in the sliding window, both in a constant time.

Let's use string `AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT` as an example. First, convert string to integer array:

- 'A' -> 0, 'C' -> 1, 'G' -> 2, 'T' -> 3

`AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT` -> `00000111110000011111100000222333`. Time to compute hash for the first sequence of length L: `0000011111`. The sequence could be considered as a number in a numeral system with the base 4 and hashed as

$h_0 = \sum_{i=0}^{L-1} c_i 4^{L-1-i}$

Here $c_{0..4} = 0$ and $c_{5..9} = 1$ are digits of `0000011111`.

Now let's consider the slice `AAAAACCCCC` -> `AAAACCCCCA`. For int arrays that means `0000011111` -> `0000111110`, to remove leading 0 and to add trailing 0. One integer in, and one out, let's recompute the hash:

$h_1 = (h_0 \times 4 - c_0 4^L) + c_{L+1}$.

Voila, window slice and hash recomputation are both done in a constant time.

Algorithm

- Iterate over the start position of sequence : from 1 to $N - L$.
  - If `start == 0`, compute the hash of the first sequence `s[0: L]`.
  - Otherwise, compute rolling hash from the previous hash value.
  - If hash is in the hashset, one met a repeated sequence, time to update the output.
  - Otherwise, add hash in the hashset.
- Return output list.

Implementation

Java | Python | Copy

```java
class Solution {
  public List<String> findRepeatedDnaSequences(String s) {
    int L = 10, n = s.length();
    if (n <= L) return new ArrayList();

    // rolling hash parameters: base a
    int a = 4, aL = (int)Math.pow(a, L);

    // convert string to array of integers
    Map<Character, Integer> toInt = new
           HashMap() {{put('A', 0); put('C', 1); put('G', 2); put('T', 3); }};
    int[] nums = new int[n];
    for(int i = 0; i < n; ++i) nums[i] = toInt.get(s.charAt(i));

    int h = 0;
    Set<Integer> seen = new HashSet();
    Set<String> output = new HashSet();
    // iterate over all sequences of length L
    for (int start = 0; start < n - L + 1; ++start) {
      // compute hash of the current sequence in O(1) time
      if (start != 0)
        h = h * a - nums[start - 1] * aL + nums[start + L - 1];
      // compute hash of the first sequence in O(1) time
      else
        for(int i = 0; i < L; ++i) h = h * a + nums[i];
      // update output and hashset of seen sequences
      if (seen.contains(h)) output.add(s.substring(start, start + L));
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N - L)$, that is $\mathcal{O}(N)$ for the constant $L = 10$. In the loop executed $N - L + 1$ one builds a hash in a constant time, that results in $\mathcal{O}(N - L)$ time complexity.
- Space complexity : $\mathcal{O}(N - L)$ to keep the hashset, that results in $\mathcal{O}(N)$ for the constant $L = 10$.

Approach 3: Bit Manipulation : Constant-time Slice Using Bitmask

> The idea is to slice over the string and to compute the bitmask of the sequence in the sliding window, both in a constant time.

As for Rabin-Karp, let's start from conversion of string to 2-bits integer array:

$A -> 0 = 00_2, \quad C -> 1 = 01_2, \quad G -> 2 = 10_2, \quad T -> 3 = 11_2$

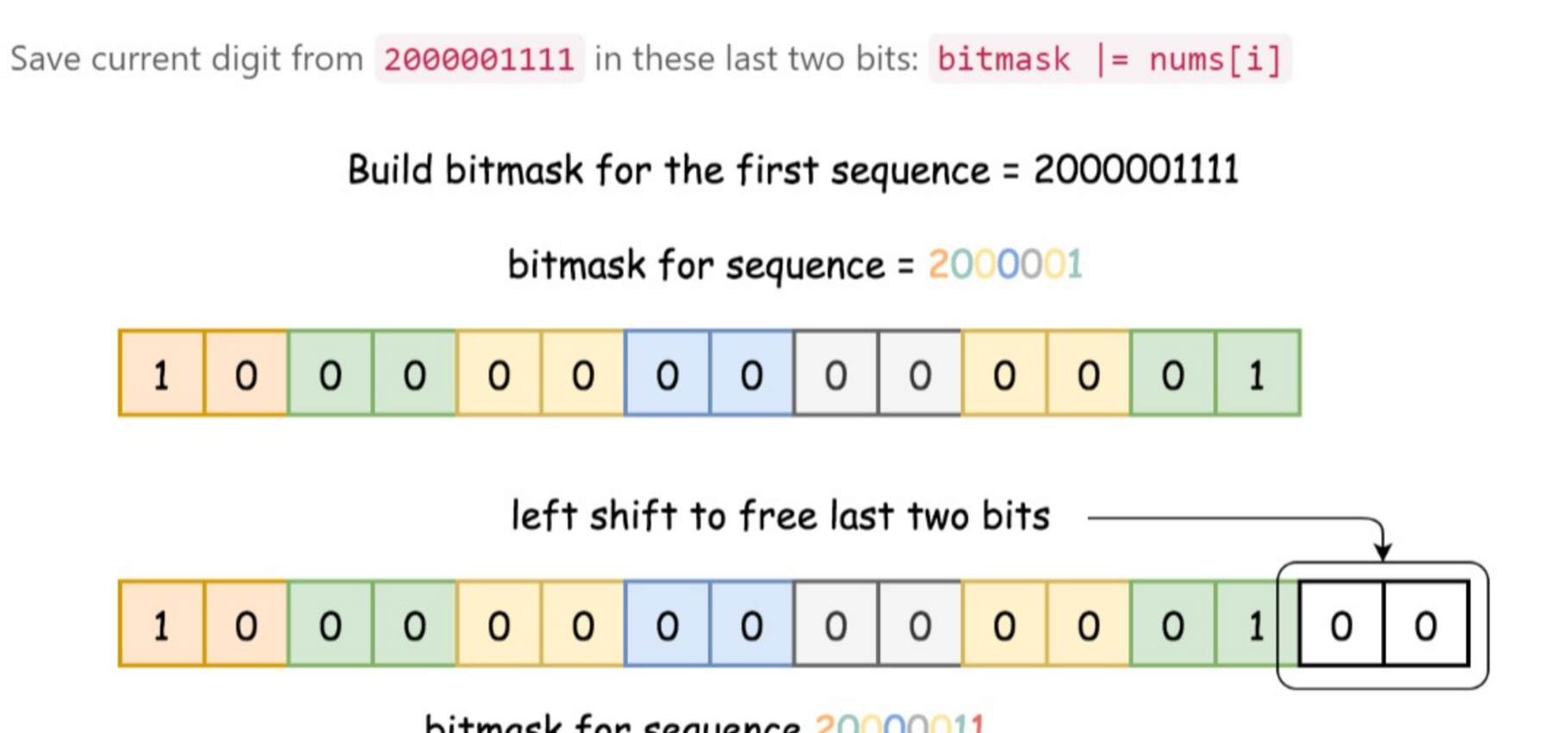
`GAAAAACCCCCAAAAACCCCCCAAAAAGGGTTT` -> `200000111110000011111100000222333`.

Time to compute bitmask for the first sequence of length L: `2000001111`. Each digit in the sequence (0, 1, 2 or 3) takes not more than 2 bits:

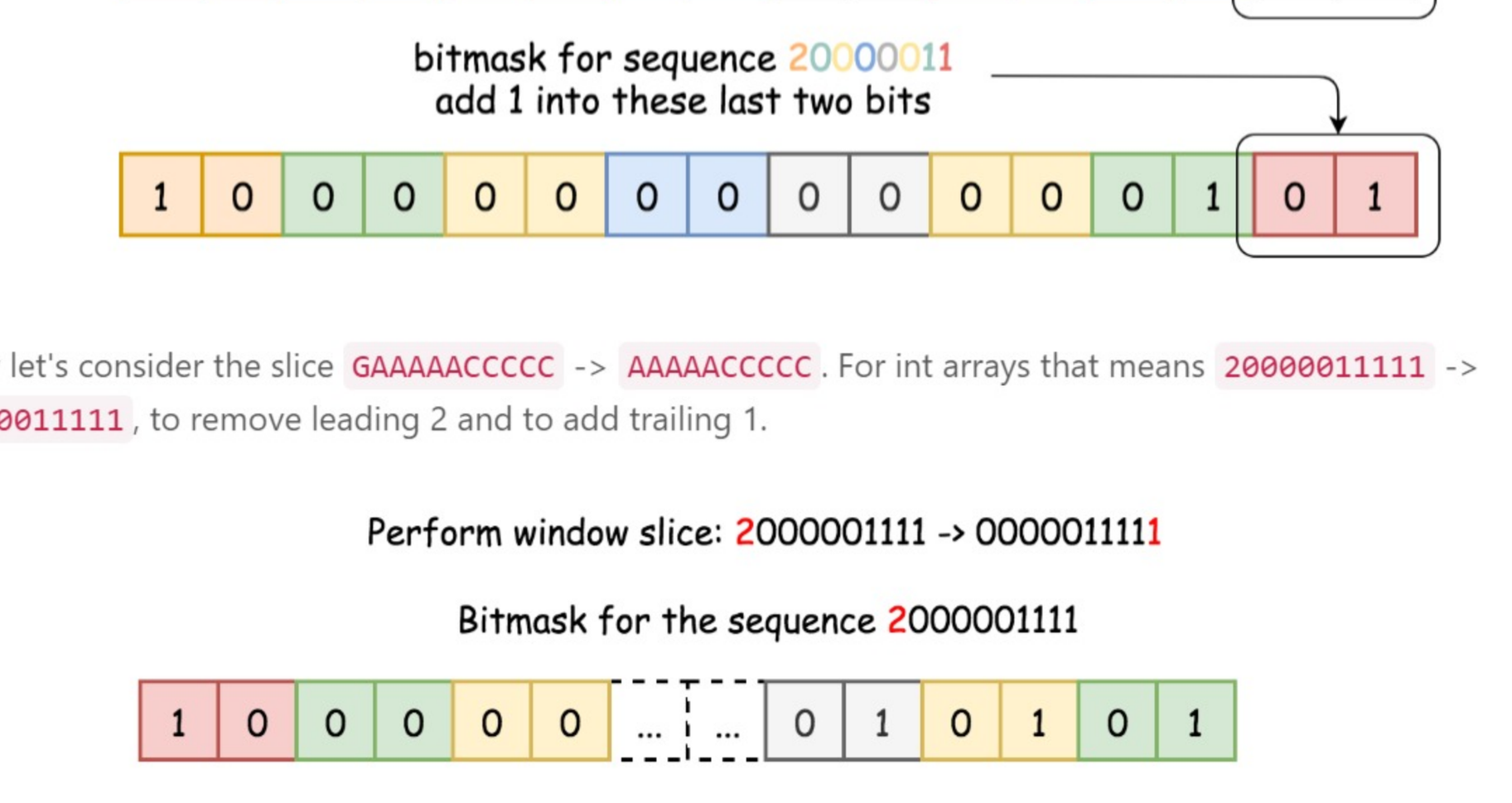
$0 = 00_2, \quad 1 = 01_2, \quad 2 = 10_2, \quad 3 = 11_2$

Hence the bitmask could be computed in the loop:

- Do left shift to free the last two bits: `bitmask <<= 2`
- Save current digit from `2000001111` in these last two bits: `bitmask |= nums[i]`



Now let's consider the slice `GAAAAACCCCC` -> `AAAAACCCCC`. For int arrays that means `20000011111` -> `0000011111`, to remove leading 2 and to add trailing 1.



To add trailing 1 is simple, the same idea as just above:

- Do left shift to free the last two bits: `bitmask <<= 2`
- Save 1 into these last two bits: `bitmask |= 1`

Now the problem is to remove two leading bits, which contain 2. In other words, the problem is to set 2L-bit and (2L + 1)-bit to zero.

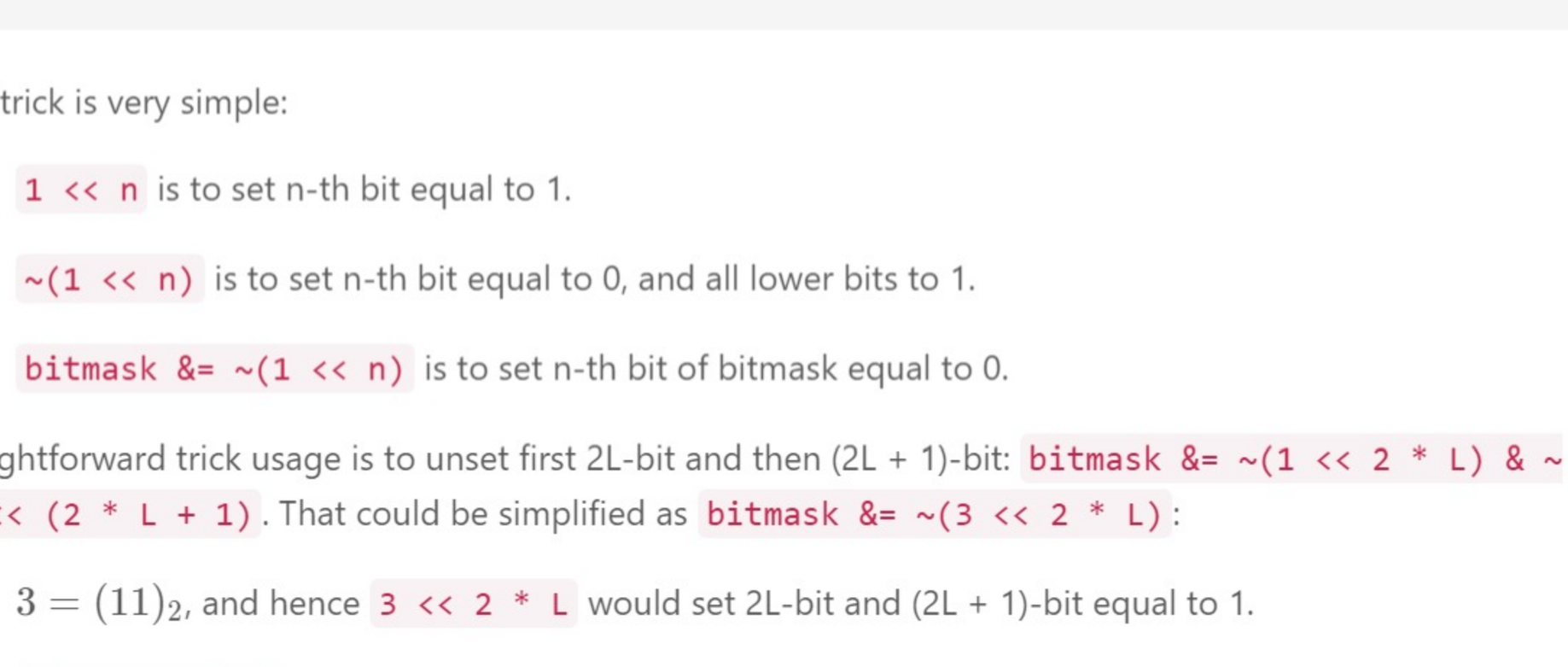
> Let's use bitwise trick to unset n-th bit: `bitmask &= ~(1 << n)`.

This trick is very simple:

- `1 << n` is to set n-th bit equal to 1.
- `~(1 << n)` is to set n-th bit equal to 0, and all lower bits to 1.
- `bitmask &= ~(1 << n)` is to set n-th bit of bitmask equal to 0.

Straightforward trick usage is to unset first 2L-bit and then (2L + 1)-bit: `bitmask &= ~(1 << 2 * L) & ~(1 << (2 * L + 1)`. That could be simplified as `bitmask &= ~(3 << 2 * L)`:

- $3 = (11)_2$, and hence `3 << 2 * L` would set 2L-bit and (2L + 1)-bit equal to 1.
- `~(3 << 2 * L)` would set 2L-bit and (2L + 1)-bit equal to 0, and all lower bits to 1.
- `bitmask &= ~(3 << 2 * L)` would set 2L-bit and (2L + 1)-bit of bitmask equal to 0.



Voila, window slice and bitmask recomputation are both done in a constant time.

Algorithm

- Iterate over the start position of sequence : from 1 to $N - L$.
  - If `start == 0`, compute the bitmask of the first sequence `s[0: L]`.
  - Otherwise, compute bitmask from the previous bitmask.
  - If bitmask is in the hashset, one met a repeated sequence, time to update the output.
  - Otherwise, add bitmask in the hashset.
- Return output list.

Implementation

Java | Python | Copy

```java
class Solution {
  public List<String> findRepeatedDnaSequences(String s) {
    int L = 10, n = s.length();
    if (n <= L) return new ArrayList();

    // rolling hash parameters: base a
    int a = 4, aL = (int)Math.pow(a, L);

    // convert string to array of integers
    Map<Character, Integer> toInt = new
           HashMap() {{put('A', 0); put('C', 1); put('G', 2); put('T', 3); }};
    int[] nums = new int[n];
    for(int i = 0; i < n; ++i) nums[i] = toInt.get(s.charAt(i));

    int bitmask = 0;
    Set<Integer> seen = new HashSet();
    Set<String> output = new HashSet();
    // iterate over all sequences of length L
    for (int start = 0; start < n - L + 1; ++start) {
      // compute bitmask of the current sequence in O(1) time
      if (start != 0) {
        // left shift to free the last 2 bit
        bitmask <<= 2;
        // add a new 2-bits number in the last two bits
        bitmask |= nums[start + L - 1];
        // unset first two bits: 2L-bit and (2L + 1)-bit
        bitmask &= ~(3 << 2 * L);
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N - L)$, that is $\mathcal{O}(N)$ for the constant $L = 10$. In the loop executed $N - L + 1$ one builds a bitmask in a constant time, that results in $\mathcal{O}(N - L)$ time complexity.
- Space complexity : $\mathcal{O}(N - L)$ to keep the hashset, that results in $\mathcal{O}(N)$ for the constant $L = 10$.

Rate this article: ★★★★★

Previous Next

Comments: 11 Sort By

Type comment here... (Markdown is supported)

Preview Post

mingrui ★112 October 29, 2019 3:40 PM
For approach 3: "hash = (hash << 2) & 0xFFFFF" is more obvious.
9 Share Reply
SHOW 2 REPLIES

fuyaoli ★94 January 6, 2020 1:27 AM
The second approach doesn't take care of the hash collision. The problem statement doesn't give us the length of the input, there is a chance that collision could happen.
2 Share Reply
SHOW 2 REPLIES

rahulkun ★446 September 27, 2019 6:56 AM
Very confusing way of writing the code.
3 Share Reply

luannv ★12 February 2, 2020 7:21 AM
For Approach 3, if we use int for bitmask then it can't be longer than 32, i.e. L is max at 16. Does it mean Rabin-Karp is the only way to go when L is super large?
1 Share Reply

infinute ★365 September 18, 2019 4:13 AM
In the last solution, is `bitmask &= ~(3 << 2 * L)` considered as an O(1) operation?
1 Share Reply
SHOW 5 REPLIES

lksbos ★0 June 23, 2020 12:24 PM
Oh my. It took me some time to figure out why approach 2 formula was different from wikipedia one. Basically we go from base^0 to base^length-1 there but here since we are doing `aL = (int)Math.pow(a, L);` instead of `aL = (int)Math.pow(a, L-1);`, to roll the hash, instead of removing first, shift right and add like this: `h = (h - nums[start - 1] * aL) * a + nums[start`
Read More
0 Share Reply

rajs92 ★1 June 9, 2020 8:59 AM
In solution 2, isn't "output.add(s.substring(start, start + L))" an O(L) operation? That should bring the total TC to O((N-L)*L)
0 Share Reply

barazesh ★21 June 3, 2020 5:08 PM
Why is the "Follow up" missing from the question itself ?
0 Share Reply

plusplusk ★1 February 29, 2020 9:55 PM
@liaison and @andvary For Approach 3, in worst case this can also be a `O(NL)` complexity, as if all the characters in the string are same, then `if (seen.contains(bitmask)) output.add(s.substring(start, start + L));` this line would be executed for every `N-L` positons and creating a string is `O(L)` time, so overall `O(NL)`. Am I doing something wrong here?
0 Share Reply

ivaylova ★4 February 11, 2020 3:29 AM
Approach 2: in the formula for h_1, why is the subscript of the last term L+1? Isn't the new character at index L, if the previous start was index 0?
0 Share Reply
SHOW 1 REPLY

< 1 2 >