Maximum Length of Repeated Subarray

Solution

Given two integer arrays `nums1` and `nums2`, return *the maximum length of a subarray that appears in **both** arrays.*

Example 1:

```
Input: nums1 = [1,2,3,2,1], nums2 = [3,2,1,4,7]
Output: 3
Explanation: The repeated subarray with maximum length is [3,2,1].
```

Example 2:

```
Input: nums1 = [0,0,0,0,0], nums2 = [0,0,0,0,0]
Output: 5
```

Constraints:

- `1 <= nums1.length, nums2.length <= 1000`
- `0 <= nums1[i], nums2[i] <= 100`

Hide Hint #1

Use dynamic programming. dp[i][j] will be the answer for inputs A[i:], B[j:].

C++

```cpp
class Solution {
public:
    int findLength(vector<int>& nums1, vector<int>& nums2) {

    }
};
```

Approach #1: Brute Force with Initial Character Map [Time Limit Exceeded]

Intuition and Algorithm

In a typical brute force, for all starting indices `i` of `A` and `j` of `B`, we will check for the longest matching subarray `A[i:i+k] == B[j:j+k]` of length `k`. This would look roughly like the following psuedocode:

```
ans = 0
for i in [0 .. A.length - 1]:
    for j in [0 .. B.length - 1]:
        k = 0
        while (A[i+k] == B[j+k]): k += 1 #and i+k < A.length etc.
        ans = max(ans, k)
```

Our insight is that in typical cases, most of the time `A[i] != B[j]`. We could instead keep a hashmap `Bstarts[A[i]] = all j such that B[j] == A[i]`, and only loop through those in our `j` loop.

Python

```
class Solution(object):
    def findLength(self, A, B):
        ans = 0
        Bstarts = collections.defaultdict(list)
        for j, y in enumerate(B):
            Bstarts[y].append(j)

        for i, x in enumerate(A):
            for j in Bstarts[x]:
                k = 0
                while i + k < len(A) and j + k < len(B) and A[i + k] == B[j + k]:
                    k += 1
                ans = max(ans, k)
        return ans
```

Java:

```
class Solution {
    public int findLength(int[] A, int[] B) {
        int ans = 0;
        Map<Integer, ArrayList<Integer>> Bstarts = new HashMap();
        for (int j = 0; j < B.length; j++) {
            Bstarts.computeIfAbsent(B[j], x -> new ArrayList()).add(j);
        }

        for (int i = 0; i < A.length; i++) if (Bstarts.containsKey(A[i])) {
            for (int j: Bstarts.get(A[i])) {
                int k = 0;
                while (i+k < A.length && j+k < B.length && A[i+k] == B[j+k]) {
                    k++;
                }
                ans = Math.max(ans, k);
```

```
                }
            }
            return ans;
        }
}
```

Complexity Analysis

- Time Complexity: $O(M * N * \min(M, N))$, where M, N are the lengths of `A, B`. The worst case is when all the elements are equal.
- Space Complexity: $O(N)$, the space used by `Bstarts`. (Of course, we could amend our algorithm to make this $O(\min(M, N))$.)

Approach #2: Binary Search with Naive Check [Time Limit Exceeded]

Intuition

If there is a length `k` subarray common to `A` and `B`, then there is a length `j <= k` subarray as well.

Let `check(length)` be the answer to the question "Is there a subarray with `length` length, common to `A` and `B`?" This is a function with range that must take the form `[True, True, ..., True, False, False, ..., False]` with at least one `True`. We can binary search on this function.

Algorithm

Focusing on the binary search, our invariant is that `check(hi)` will always be `False`. We'll start with `hi = min(len(A), len(B)) + 1`; clearly `check(hi) is False`.

Now we perform our check in the midpoint `mi` of `lo` and `hi`. When it is possible, then `lo = mi + 1`, and when it isn't, `hi = mi`. This maintains the invariant. At the end of our binary search, `hi == lo` and `lo` is the lowest value such that `check(lo) is False`, so we want `lo - 1`.

As for the check itself, we can naively check whether any `A[i:i+k] == B[j:j+k]` using set structures.

Python

```python
class Solution(object):
    def findLength(self, A, B):
        def check(length):
            seen = set(tuple(A[i:i+length])
                       for i in range(len(A) - length + 1))
            return any(tuple(B[j:j+length]) in seen
                       for j in range(len(B) - length + 1))

        lo, hi = 0, min(len(A), len(B)) + 1
        while lo < hi:
            mi = (lo + hi) // 2
            if check(mi):
                lo = mi + 1
            else:
                hi = mi
        return lo - 1
```

Java:

```java
class Solution {
    public boolean check(int length, int[] A, int[] B) {
        Set<String> seen = new HashSet();
        for (int i = 0; i + length <= A.length; ++i) {
            seen.add(Arrays.toString(Arrays.copyOfRange(A, i, i+length)));
        }
        for (int j = 0; j + length <= B.length; ++j) {
            if (seen.contains(Arrays.toString(Arrays.copyOfRange(B, j,
j+length)))) {
                return true;
            }
```

```
        }
        return false;
    }

    public int findLength(int[] A, int[] B) {
        int lo = 0, hi = Math.min(A.length, B.length) + 1;
        while (lo < hi) {
            int mi = (lo + hi) / 2;
            if (check(mi, A, B)) lo = mi + 1;
            else hi = mi;
        }
        return lo - 1;
    }
}
```

Complexity Analysis

- Time Complexity: $O((M+N) * \min(M,N) * \log(\min(M,N)))$, where M, N are the lengths of `A, B`. The log factor comes from the binary search. The complexity of our naive check of a given length is $O((M+N) * \text{length})$, as we will create the `seen` strings with complexity $O(M * \text{length})$, then search for them with complexity $O(N * \text{length})$, and our total complexity when performing our `check` is the addition of these two.
- Space Complexity: $O(M^2)$, the space used by `seen`.

Approach #3: Dynamic Programming [Accepted]

Intuition and Algorithm

Since a common subarray of `A` and `B` must start at some `A[i]` and `B[j]`, let `dp[i][j]` be the longest common prefix of `A[i:]` and `B[j:]`. Whenever `A[i] == B[j]`, we know `dp[i][j] = dp[i+1][j+1] + 1`. Also, the answer is `max(dp[i][j])` over all `i, j`.

We can perform bottom-up dynamic programming to find the answer based on this recurrence. Our loop invariant is that the answer is already calculated correctly and stored in `dp` for any larger `i, j`.

Python

```
class Solution(object):
    def findLength(self, A, B):
        memo = [[0] * (len(B) + 1) for _ in range(len(A) + 1)]
        for i in range(len(A) - 1, -1, -1):
            for j in range(len(B) - 1, -1, -1):
                if A[i] == B[j]:
                    memo[i][j] = memo[i + 1][j + 1] + 1
        return max(max(row) for row in memo)
```

Java

```
class Solution {
    public int findLength(int[] A, int[] B) {
```

```
        int ans = 0;
        int[][] memo = new int[A.length + 1][B.length + 1];
        for (int i = A.length - 1; i >= 0; --i) {
            for (int j = B.length - 1; j >= 0; --j) {
                if (A[i] == B[j]) {
                    memo[i][j] = memo[i+1][j+1] + 1;
                    if (ans < memo[i][j]) ans = memo[i][j];
                }
            }
        }
        return ans;
    }
}
```

Complexity Analysis

- Time Complexity: $O(M * N)$, where M, N are the lengths of `A, B`.
- Space Complexity: $O(M * N)$, the space used by `dp`.

Approach #4: Binary Search with Rolling Hash [Accepted]

Intuition

As in *Approach #2*, we will binary search for the answer. However, we will use a *rolling hash* (Rabin-Karp algorithm) to store hashes in our set structure.

Algorithm

For some prime p, consider the following function modulo some prime modulus $\mathcal{M}$:

$\text{hash}(S) = \sum_{0 \le i < len(S)} p^i * S[i]$

Notably, $\text{hash}(S[1:] + x) = \frac{(\text{hash}(S) - S[0])}{p} + p^{n-1}x$. This shows we can get the hash of all $A[i : i + \text{guess}]$ in linear time. We will also use the fact that $p^{-1} = p^{\mathcal{M}-2} \mod \mathcal{M}$.

For every `i >= length - 1`, we will want to record the hash of `A[i-length+1], A[i-length+2], ..., A[i]`. After, we will truncate the first element by `h = (h - A[i - (length - 1)]) * Pinv % MOD` to get ready to add the next element.

To make our algorithm air tight, we also make a naive check when our work with rolling hashes says that we have found a match.

```
class Solution(object):
    def findLength(self, A, B):
        P, MOD = 113, 10**9 + 7
        Pinv = pow(P, MOD - 2, MOD)
        def check(guess):
            def rolling(A, length):
                if length == 0:
```

```
                    yield 0, 0
                    return

                h, power = 0, 1
                for i, x in enumerate(A):
                    h = (h + x * power) % MOD
                    if i < length - 1:
                        power = (power * P) % MOD
                    else:
                        yield h, i - (length - 1)
                        h = (h - A[i - (length - 1)]) * Pinv % MOD

            hashes = collections.defaultdict(list)
            for ha, start in rolling(A, guess):
                hashes[ha].append(start)
            for ha, start in rolling(B, guess):
                iarr = hashes.get(ha, [])
                if any(A[i: i + guess] == B[start: start + guess] for i in
iarr):
                    return True
            return False

        lo, hi = 0, min(len(A), len(B)) + 1
        while lo < hi:
            mi = (lo + hi) // 2
            if check(mi):
                lo = mi + 1
            else:
                hi = mi
        return lo - 1
```

Java:

```
import java.math.BigInteger;

class Solution {
    int P = 113;
    int MOD = 1_000_000_007;
    int Pinv =
BigInteger.valueOf(P).modInverse(BigInteger.valueOf(MOD)).intValue();

    private int[] rolling(int[] source, int length) {
        int[] ans = new int[source.length - length + 1];
        long h = 0, power = 1;
        if (length == 0) return ans;
        for (int i = 0; i < source.length; ++i) {
            h = (h + source[i] * power) % MOD;
            if (i < length - 1) {
                power = (power * P) % MOD;
```

```
            } else {
                ans[i - (length - 1)] = (int) h;
                h = (h - source[i - (length - 1)]) * Pinv % MOD;
                if (h < 0) h += MOD;
            }
        }
        return ans;
    }

    private boolean check(int guess, int[] A, int[] B) {
        Map<Integer, List<Integer>> hashes = new HashMap();
        int k = 0;
        for (int x: rolling(A, guess)) {
            hashes.computeIfAbsent(x, z -> new ArrayList()).add(k++);
        }
        int j = 0;
        for (int x: rolling(B, guess)) {
            for (int i: hashes.getOrDefault(x, new ArrayList<Integer>()))
                if (Arrays.equals(Arrays.copyOfRange(A, i, i+guess),
                                  Arrays.copyOfRange(B, j, j+guess))) {
                    return true;
                }
            j++;
        }
        return false;
    }

    public int findLength(int[] A, int[] B) {
        int lo = 0, hi = Math.min(A.length, B.length) + 1;
        while (lo < hi) {
            int mi = (lo + hi) / 2;
            if (check(mi, A, B)) lo = mi + 1;
            else hi = mi;
        }
        return lo - 1;
    }
}
```

Complexity Analysis

- Time Complexity: $O((M+N) * \log(\min(M,N)))$, where M, N are the lengths of `A, B`. The log factor is contributed by the binary search, while creating the rolling hashes is $O(M+N)$. The checks for duplicate hashes are $O(1)$. If we perform a naive check to make sure our answer is correct, it adds a factor of $O(\min(M,N))$ to our cost of `check`, which keeps the complexity the same.
- Space Complexity: $O(M)$, the space used to store `hashes` and the subarrays in our final naive check.