Array Comparisons With Value Updates

Research Question: How to determine whether two arrays of equal size are identical if array values will be updated

Subject Area: Computer Science

Word count: 3914

Contents

1 Introduction

In a computing contest hosted on the online judging platform DMOJ, I came across a particular computational problem that I was unable to solve within the time provided.

> **The exact problem statement and constraints are as follows**(maxcruikshanks, 2023):
>
> Given an array, A of N integers, find the answers to Q queries of 2 types,
>
> 1 l_1 r_1 l_2 r_2: Output 1 if the subarray from $[l_1, r_1]$ is identical to $[l_2, r_2]$ or 0 if they are not identical. An array is identical to another if the lengths are equal and the same numbers are present in the same order.
>
> 2 i v: Update the integer at index i to v.
>
> Constraints:
>
> For all subtasks:
>
> Time Limit: 3 seconds
>
> $1 \leq N, Q \leq 5 \times 10^5$
>
> $1 \leq A_i, v \leq 10^9$
>
> $r_1 - l_1 = r_2 - l_2$
>
> $1 \leq l_1 \leq r_1 \leq N$
>
> $1 \leq l_2 \leq r_2 \leq N$
>
> Subtask 1
>
> $1 \leq N, Q \leq 5000$
>
> Subtask 2
>
> No additional constraints.

The problem asks to find a way to check whether two equal-sized arrays are identical; numbers are present in the same order. The problem first presents an array, A of N integers and asks the contestant to answer Q queries of two types. The first type query is a query that asks

the contestant to check whether two subarrays of the same length formed by given indices are identical, requesting the contestant to output a "1" (signifying the boolean value true) if the arrays are identical and a "0" (signifying the boolean value false) if the arrays are not identical. This first type of query begins with a "1" to indicate the first type of query followed by l_1 and r_1, giving the left and right index of one array and l_2 and r_2, giving the left and right index of the other array. Together, the first type of query can be represented as "1 l_1 r_1 l_2 r_2". The second type of query is a query that asks the contestant to update the value present at an index in the array with another value. This second type of query begins with a "2" to indicate the second type of query followed by i and v, giving the index of the array to update and the updated value.

The problem was divided into two subtasks where each subtasks had different constraints for the input given. Most importantly, subtask 1 had constraints on N (the length of the array) and Q (the number of queries) that were significantly more strict ($1 \leq N \leq 5000$ and $1 \leq Q \leq 5000$) than that of subtask 2 ($1 \leq N \leq 5 \times 10^5$ and $1 \leq Q \leq 5 \times 10^5$). Subtask 1's strict constraints rewarded people who found simpler solutions that were sufficient for smaller cases while subtask 2 of the problem asked for a more optimized solution given the loose constraints.

In this paper, I aim to investigate the different possible solutions possible for each subtask of this problem, as well as the differing solutions' respective performance and scalability.

2 Naive Solution in $O(QN)$ Time Complexity

For subtask 1, it can be recognized by looking at the constraints that N and Q are at most 5000. After understanding that N and Q are at most 5000, a simple brute-force algorithm can be attempted, where the subarray comparisons are computed directly by comparing the value present at each index of the two subarrays and checking for equivalence. If there is any index where the value from the subarrays do not match the subarrays are not identical and the subarrays are identical. In the worst case, array comparisons will require N operations

(if $l_1 = l_2 = 1$ and $r_1 = r_2 = N$). Therefore this results in an worst-case time complexity of $O(QN)$ to process all Q queries. Since N and Q are at most 5000, it will take around 2.5×10^7 operations to process all Q queries in the worst case, which is possible for many programming languages as most are able to perform at least 10^7 operations per second. The procedure of the subarray comparison may be expressed in pseudocode as follows.

Algorithm 1 Naive Subarray Comparison

```
1: procedure COMPARE(arr, l1, r1, l2, r2)
2:     arrlen := r1 - l1                          ▷ Equivalently, r2 - l2
3:     for i:=0 to arrlen do
4:         if arr[l1 + i - 1] ≠ arr[l2 + i - 1] then
5:             return False                       ▷ The subarrays are not equal
6:     return True                                ▷ The subarrays are equal
```

3 Optimized Solution in $O(QlogN)$ Time Complexity

For subtask 2, N and Q are at most 5×10^5, rendering the previously discussed naive $O(QN)$ solution inefficient. For this subtask, a more optimized solution needs to be developed. Specifically, the way subarrays are compared needs to be made more efficient. One method of optimization is through the use of hashing and data structures that can handle range-sum queries and value updates with a time complexity faster than linear time complexity. Two data structures that could be used is the binary indexed tree (also known as a Fenwick tree) and the segment tree as they are both able to handle the two aforementioned types of operations in logarithmic time complexity. In this paper, the binary indexed tree will be the data structure that is used due to its relative ease to understand and implement compared to the segment tree. Likewise, polynomial hashing will be the method of hashing that will be used in this paper as it is a relatively simple approach to hashing, using only addition, multiplication, and the modulo operator to generate hashes.

The optimized solution will use the algorithms and data structures, like polynomial hashing and the binary indexed tree, introduced above to speed up subarray comparisons by comparing

the hash values of subarrays instead of the elements in the subarrays. In the optimized solution, hash values of the indicated subarrays will be obtained and compared for equivalence. If the hash of the two subarrays are equal the two subarrays are identical (output "1"), otherwise the two subarrays are not identical (output "0"). By comparing hash values instead of individual elements of the subarrays, the total number of comparisons needed for each query of the first type is reduced as only one comparison is needed instead of N comparison. In the optimized solution, the binary indexed tree is used to help maintain the hash values of subarrays and keep them up-to-date with the changes in the array that happens due to queries of the second type.

3.1 Polynomial Hash

Polynomial hash is a relatively simple approach to string hashing that only uses addition, multiplication, and the modulo operator. Polynomial hashing is also sometimes known as the rolling-hash of strings, originating from its extensive use in the explanation of the *Rabin-Karp string searching algorithm*(Moore and Chumbley and Khim, n.d).

A polynomial hash, H, generated using an array a, is generally defined as

$$H(a) = (a_0 \times p^{n-1} + a_1 \times p^{n-2} + a_2 \times p^{n-3} + ... + a_{n-1} \times p^0) \ mod \ M$$

where a_i is the integer at the i^{th} index of an array (in the case where a is a string, a_i is the ASCII[1] value of the i^{th} character of the string) and p and M are arbitrary prime numbers where $p \ll M$.

The procedure of constructing a polynomial hash from an index l to an index r ($(1 \leq l \leq r \leq N)$) could be expressed in pseudocode as follows.

[1]American Standard Code for Information Interchange

Algorithm 2 Polynomial Hashing

```
1: procedure HASH(arr, p, M, l, r)
2:     hash := 0
3:     for i:=l to r do
4:         hash *= p
5:         hash += arr[i]
6:         hash %= M                          ▷ % denotes the modulus operator
7:     return hash
```

As seen in the pseudo-code, polynomial hashes of any arbitrary length can be computed using a sequence of multiplication, addition, and modular arithmetic (see Algorithm 2). This is because of how polynomial hashes factor into a series of basic arithmetic operations as shown below.

$$
\begin{aligned}
H(a) &= (a_0 \times p^{n-1} + a_1 \times p^{n-2} + a_2 \times p^{n-3} + ... + a_{n-2} \times p^1 + a_{n-1} \times p^0) \ mod \ M \\
&= (p(a_1 \times p^{n-2} + a_2 \times p^{n-3} + a_3 \times p^{n-4} + ... + a_{n-3} \times p^1 + a_{n-2} \times p^0) + a_{n-1}) \ mod \ M \\
&= (p(p(a_1 \times p^{n-3} + a_2 \times p^{n-4} + a_3 \times p^{n-5} + ... + a_{n-4} \times p^1 + a_{n-3} \times p^0) + a_{n-2}) + a_{n-1}) \ mod \ M \\
&= (p(p(p(a_1 \times p^{n-4} + a_2 \times p^{n-5} + a_3 \times p^{n-6} + ... + a_{n-5} \times p^1 + a_{n-4} \times p^0) + a_{n-3}) + a_{n-2}) \\
&\quad + a_{n-1}) \ mod \ M \qquad ... \\
&= p(p(p(...p(p(0) \ mod \ M + a_1 \ mod \ M) + ...) + a_{n-2} \ mod \ M) + a_{n-1} \ mod \ M) + a_n \ mod \ M
\end{aligned}
$$

The 0 is included at the core of the factorized form to cover the case of a polynomial hash of length 0, in which the polynomial hash would simply have a value of 0.

3.1.1 Using Prefix Sum To Compare Equal-Sized Segments of Sequences

Prefix sum or cumulative sum is a method of pre-computation to generate a sequence of numbers representing the sum of prefixes (Blelloch, n.d); running sum of all numbers up to a certain index i (inclusive). For a 1-indexed array of numbers, x consisting of $x_1, x_2, x_3, ...$, a 1-indexed sequence, $prefix$, representing its prefix sums can be expressed as follows.

$$prefix_0 = 0$$

$$prefix_1 = x_1$$

$$prefix_2 = x_1 + x_2$$

$$prefix_3 = x_1 + x_2 + x_3$$

$$prefix_4 = x_1 + x_2 + x_3 + x_4$$

$$prefix_5 = x_1 + x_2 + x_3 + x_4 + x_5$$

$$...$$

By manipulating these pre-computed prefix sums through subtraction, the sum of a range of values from any index i to any index j ($1 \leq i \leq j \leq N$) can be obtained (the sum of a certain range of indices will be referred to as a "range sum"). Let the function *range_sum(i, j)* denote a procedure that sums the values from index i to index j, inclusive. Using the previously described sequence, x, $range\_sum(2, 3) = x_2 + x_3$. Similarly, using prefix sums and subtraction:

$$\begin{aligned} range\_sum(2, 3) &= prefix_3 - prefix_1 \\ &= (x_1 + x_2 + x_3) - (x_1) \\ &= x_2 + x_3 \end{aligned}$$

In general, any sum of numbers from index i to index j ($1 \leq i \leq j \leq N$) can be obtained through the following expression where $prefix_j$ denotes the prefix sum of values up to the j^{th} index and $prefix_{i-1}$ denotes the prefix sum of values up to the $(i-1)^{th}$ index.

$$range\_sum(i, j) = prefix_j - prefix_{i-1}$$

Knowing this, prefix sums can be extended to compare subarrays with the help of polynomial hashing in a similar way. By computing prefix hashes instead of prefix sums, the hash of any range of indices can be found using a similar formula used for range sums. In this context, a prefix hash of some arbitrary length n represents the polynomial hash that is obtained if we consider only the first n elements of an array. Using the previously declared sequence of numbers x, a prefix hash of length 3 can be expressed as follows, where $hash$ denotes the resultant polynomial hash and p and M are arbitrary prime numbers where $p \ll M$.

$$hash = (x_1 \times p^2 + x_2 \times p^1 + x_3 \times p^0) mod\ M$$

For array x declared above, let $prefix\_hash$ denote its sequence of prefix hashes, where $prefix\_hash_i$ gives a prefix hash of length i of array x.

$$prefix\_hash_0 = 0$$

$$prefix\_hash_1 = x_1 \times p^0$$

$$prefix\_hash_2 = x_1 \times p^1 + x_2 \times p^0$$

$$prefix\_hash_3 = x_1 \times p^2 + x_2 \times p^1 + x_3 \times p^0$$

$$prefix\_hash_4 = x_1 \times p^3 + x_2 \times p^2 + x_3 \times p^1 + x_4 \times p^0$$

$$prefix\_hash_5 = x_1 \times p^4 + x_2 \times p^3 + x_3 \times p^2 + x_4 \times p^1 + x_5 \times p^0$$

...

Using $prefix\_hash$, the hash of a range from any index i to any index j $(1 \leq i \leq j \leq N)$ can be computed similarly to range sums (the hash of a certain range of indices will be referred to as a "range hash"). Let the function $range\_hash(i, j)$ denote the procedure for generating a polynomial hash for values from index i to index j, inclusive. For example, if i was 3 and j was 5, $range\_hash(3, 5)$ would return $x_3 \times p^2 + x_4 \times p^1 + x_5 \times p^0$. Similarly, applying the

same technique that was used to calculate range sums

$$\begin{aligned} range\_hash(3,5) &= prefix\_hash_5 - prefix\_hash_{3-1} \\ &= prefix\_hash_5 - prefix\_hash_2 \\ &= (x_1 \times p^4 + x_2 \times p^3 + x_3 \times p^2 + x_4 \times p^1 + x_5 \times p^0) \\ &- (x_1 \times p^1 + x_2 \times p^0) \end{aligned}$$

, it can be seen that the hash value computed does not match that of the proper polynomial hash from index 3 to index 5. However, in the computation using prefix hashes, it can also be observed that the difference between the index of p on the first term, x_1, between the two prefix hashes is equal to $4 - 1 = 3$. Likewise, comparing the index of p on the second term, x_2, the same difference of 3 can be observed. More generally, this difference can be expressed by $(j-1) - (i-1-1) = j - i + 1$, obtained simply by subtracting the highest powers of p of the two prefix hashes. This difference in the index of p between common terms between the two prefix hashes used to compute range hashes will always be equal to $j - i + 1$ as the index of p decreases by a one each time moving to the next term. In the previous equation used to computer range_hash(3, 5), scaling $prefix\_hash_{i-1} = prefix\_hash_2$ by p^{j-i+1} results in the correct polynomial hash as shown below.

$$\begin{aligned} range\_hash(3,5) &= prefix\_hash_5 - prefix\_hash_2 \times p^{j-i+1} \\ &= prefix\_hash_5 - prefix\_hash_2 \times p^3 \\ &= (x_1 \times p^4 + x_2 \times p^3 + x_3 \times p^2 + x_4 \times p^1 + x_5 \times p^0) \\ &- (x_1 \times p^4 + x_2 \times p^3) \\ &= x_3 \times p^2 + x_4 \times p^1 + x_5 \times p^0 \end{aligned}$$

More generally, the hash of any sequence from index i to index j inclusive ($1 \leq i \leq j \leq N$) can be expressed as follows (TimonKnigge, 2014).

$$range\_hash(i, j) = prefix\_hash_j - prefix\_hash_{i-1} \times p^{j-i+1}$$

The procedure for calculating range hashes using prefix hashes can be expressed using pseudocode as follows.

Algorithm 3 Prefix Hashes/Range Hashes

```
1:  procedure PREFIX_HASH(arr, p, M)
2:      arrLen := length of arr
3:      hash := 0
4:      prefix_hash := empty list
5:      prefix_hash → push_back(hash)
6:      for i:=0 to arrLen do
7:          hash *= p
8:          hash += arr[i]
9:          hash %= M                          ▷ % denotes the modulus operator
10:         prefix_hash → push_back(hash)
11:     return prefix_hash
12:
13: procedure RANGE_HASH(arr, p, M, l, r)
14:     prefix_hash := PREFIX_HASH(arr, p, M)
15:     return (prefix_hash[j]) − (prefix_hash[i-1] × p^(j−i+1))
```

3.2 Binary Indexed Tree

The binary indexed tree is a data structure that provides range queries and point value updates for binary operations (a mathematical operation that has both an identity element and an inverse element) efficiently in logarithmic time complexity (cp-algorithms, 2024). The binary indexed tree was first popularized in 1994 by Peter Fenwick as "a new data structure for cumulative frequency tables" (Fenwick, 1994).

As suggested by the name, binary indexed trees take advantage of the binary number system to efficiently perform its operations. Although its name is evocative of a tree structure, binary indexed trees are often more easily understood and more commonly represented by a

one-indexed array. This method of representing the binary indexed tree as an array will also be how the different operations of the binary indexed tree will be explained in this paper. In a binary indexed tree, each index is "responsible" for a certain range of values based on the index's *least significant bit* (LSB). In the context of this paper, where the binary indexed tree will be used to store a sum of value and operate using addition, an index's responsibility refers to the range of values that it sums (and later stored at that given index). It should be noted that a binary indexed tree is not only limited to addition, but any binary operations, as mentioned previously. Additionally, the LSB of any number is the value of the active bit that is the furthest right, for example, the LSB of 10 (1010_2 written in binary) would be 2 and the LSB of 8 (1000_2 written in binary) would be 8. **Figure 1** gives a visual representation of a binary indexed tree represented using a one-indexed array with the indices, the binary forms of the indices, and the range of responsibility of each index all displayed from left to right, respectively. The ranges of responsibility for each index are denoted visually by coloured bars that begin orange coloured and traverse down as a dark blue colour if needed. The total number of indices each bar will reach is equal to that of the respective starting index's LSB. For example, in the case of index 6, its range of responsibility is 2 in turn reaching itself as well as index 5. In the case of index 3, its range of responsibility is 1, making it reach only one singular index which is itself.

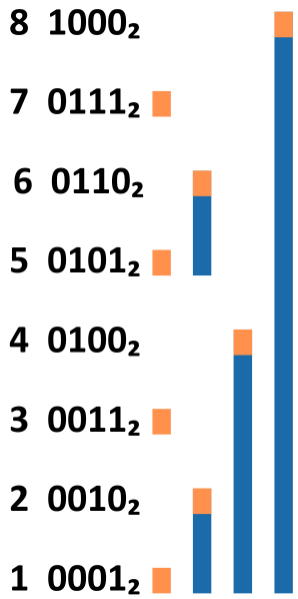

Figure 1: Ranges of responsibility for each index in a binary indexed tree with 8 values

3.2.1 Querying using binary indexed tree

In the context of binary indexed tree, the query operation can be understood as an operation used to find the cumulative value, based on the binary operation that the binary indexed tree is told to support, from a certain starting index to 1. If a binary indexed tree using addition is considered, a query starting at index i would return the sum of values up to that index from the original array used to construct the binary indexed tree (or in other words a prefix sum up to index i).

When querying a binary indexed tree, a starting index, i, must first be given as a parameter. Using this starting index i, the query operation will continuously add the value at index i from the array used to represent a binary indexed tree to some variable (to keep track of the cumulative value) and remove the LSB of i from i until i equals 0 (all of the starting index's active bits have been removed). As an example, consider a query starting at index 7 of a binary indexed tree and let *total* be the value that will store the resulting value from the query. In this example, the value at index 7 of the binary indexed tree will first be added to *total* and 1 will be subtracted from 7 (0111_2 in binary) as 1 is its LSB resulting in the next

index being considered to be index 6. The value at index 6 of the binary indexed tree will then be added to *total* and 2 will be subtracted from 6 (0110_2 in binary) as 2 is its LSB resulting in the next index being considered to be index 4. The value at index 4 of the binary indexed tree will be added and the query is complete as the LSB of 4 (0100_2 in binary) is equal to 4 resulting in the next index being considered to be index 0, which "does not exist" within a one-indexed array used to represent a binary indexed tree.

Visually, this process can be understood as a downward cascade starting from the starting index until index 0 is reached. In **Figure 2**, the cascade of values used for querying a binary indexed tree for the prefix sum starting from index 7 is indicated in yellow where the downward arrows point towards the subsequent index used in the query. From **Figure 2**, it could also be noticed that the bars coloured in yellow (the range of responsibilities of indexed using the query) are able to come together to form a larger cumulative range of responsibility that spans from index 7 to index 1.

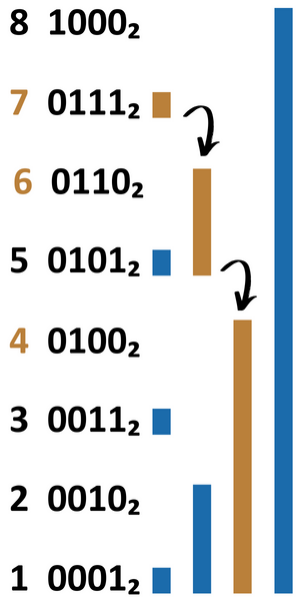

Figure 2: Querying binary indexed tree for prefix sum starting from index 7 (in yellow)

The procedure for querying a binary indexed tree (using addition) can be expressed as follows, where *tree* would store the array representation of a binary indexed tree and i is the starting index.

Algorithm 4 Querying Using binary indexed tree

```
1: procedure QUERY(tree, i)
2:     total := 0
3:     while i > 0 do
4:         total += tree[i]
5:         i -= i &(−i)                ▷ i &(−i) returns the least significant bit of i
6:     return total
```

3.2.2 Value Updates using Binary Indexed Tree

Updating a value in a binary indexed tree functions similarly to the process for querying, as both rely heavily on binary operations and the *least significant bit*. For value updates, however, the *least significant bit* is added to the current node being processed, rather than subtracted, until we reach an index that is equal to or larger than the highest index being considered. Graphically, this means that instead of cascading downwards like when querying a binary indexed tree, the indices being updated will cascade upwards. Likewise, it could also be visualized by drawing a horizontal line at the index being updated and updating every single index where its range of responsibility crosses the horizontal line (see **Figure 3**). For example, if the value stored in index 3 of the original array used to construct the binary indexed tree was updated, the value at index 3 in the binary indexed tree will first be updated then the value at index 4 (3 plus LSB of 3, which is 1) then the value at index 8 (4 plus LSB of 4, which is 4). **Figure 3** illustrates the process explained above graphically, where all of the indices coloured in yellow are indices that will be updated if the value at index 3 of the original array used to construct the binary indexed tree was updated. The upward arrows in **Figure 3** indicate the subsequent index of the binary indexed tree being updated.

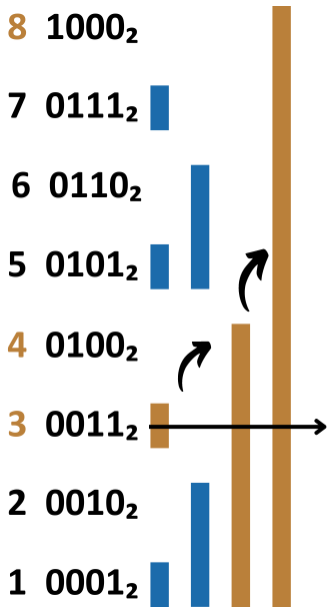

Figure 3: Point Update of binary indexed tree starting from index 3 (in yellow)

The procedure for updating a binary indexed tree can be expressed as follows, where *tree* would store the array representation of a binary indexed tree, i being the index of the original array that is being updated and *diff* being the difference between the updated value and the original value.

Algorithm 5 Point Updates Using binary indexed tree

```
1: procedure UPDATE(tree, i, diff)
2:     while i < length of tree do
3:         tree at i += diff
4:         i += i &(−i)          ▷ i &(−i) returns the least significant bit of i
5:     return                    ▷ void function, no return type
```

3.2.3 Constructing a Binary Indexed Tree

To construct a binary indexed tree, it is possible to simply progressively populate each index by updating each index in the binary indexed tree with each index in an array A using the procedure for value updates as described above. Although this algorithm for constructing a binary indexed tree would take $O(NlogN)$ time, it should be noted that there exists an algorithm that can construct the same array representation of a binary indexed tree in $O(N)$

time (Fiset, 2017).

The procedure for constructing a binary indexed tree in $O(NlogN)$ time can be expressed as follows, where *tree* denotes a 1-indexed array that will represent the binary indexed tree, *arr* being the array the BIT will be formed from, and *UPDATE(...)* referring to the point update function described above.

Algorithm 6 Construction of Binary Indexed Tree

```
1: procedure CONSTRUCT(arr)
2:     arrlen := length of arr
3:     tree := array of size (arrlen + 1)
4:     for i:=1 to arrlen + 1 do
5:         UPDATE(tree, i, arr[i])
6:     return tree
```

3.2.4 Finding Least Significant Bit of a Number

To find the value of the *least significant bit* of any integer K, the following operation can be used, where & represents the bitwise AND operator.

$$K\&(-K)$$

To understand why this operation works to find the value of the least significant bit, we can consider the value of 10^6 in binary, which is expressed by 11110100001001000000 in *little endian* (Parr, 2021). To find -10^6 in binary, we can consider the *two's complement* method of representing signed integers, which can be achieved in two steps: inverting all bits (changing every 1 to a 0 and every 0 to a 1) and adding 1 to the inverted number. After inverting all bits in 10^6, we get 00001011110110111111 (Finley, 2000). Notice how all the trailing 0's from the binary form of 10^6 are converted to trailing 1's. After adding 1 to this inverted number, all trailing 1's will be flipped to 0's and the rightmost 0 being flipped to 1 which is also the position of our rightmost 1 in the original number. When we take bitwise AND of both numbers, only the rightmost active bit will remain as no other pairs of bits between the two numbers are both active.

To clarify the assertions above, the following binary numbers express 10^6 in *little endian* and -10^6 using *two's complement* where the underlined portion describes the similarities between the two binary representations as well as the the changes that occurred after adding 1 to the inverted number:

$$-10^6 : 00001011110111\underline{000000}$$

$$10^6 : 11110100001001\underline{000000}$$

3.3 Extending Binary Indexed Trees To Maintain Polynomial Hashes

In order to properly maintain polynomial hashes using binary indexed trees, some crucial properties of polynomial hashing and prefix sums must be observed. First, the similarities between a prefix hash and a prefix sum can be observed as follows.

$$H(a) = (a_1 \times p^{n-1} + a_2 \times p^{n-2} + a_3 \times p^{n-3} + ... + a_n \times p^0) \ mod \ M$$

$$prefix(a) = a_1 + a_2 + a_3 + ... + a_n$$

Between the two, it can notice that they both involve summing a consecutive sequence of values, with the only difference being the multiplication and modulo operations that occurs when calculating prefix hashes. Based on this observation, it is reasonable to assume that it is possible to process polynomial hashes using binary indexed trees similarly to how addition is processed. More formally, for a binary indexed tree maintaining polynomial hashes, any index i in the binary indexed tree will maintain the polynomial hash for the values from index $i - LSB(i)$ to i in the original array used to construct the binary indexed tree. Though it must be noted that in this case, the factors of p for each value must be consistently maintained within the binary indexed tree. After understanding the similarities between prefix sums and prefix hashes, maintaining polynomial hashes using binary indexed trees breaks down into a problem of properly maintaining the powers of p that scale each value.

To properly maintain these powers of p, we need to observe the changes in multiplying powers of p for any values, a_i, used to form a polynomial hash. The generalized formula for polynomial hashes is as follows:

$$H(a) = (a_1 \times p^{n-1} + a_2 \times p^{n-2} + a_3 \times p^{n-3} + ... + a_n \times p^0) \ mod \ M$$

Through this generalized formula for polynomial hashes, it can be seen that for prefix hashes, the index for the power of p for any value a_i is always $n - i$ where n denotes the length of the array and i denotes the index of the element. Similarly, for hashes being maintained which are not prefix sums but rather a latter portion of a prefix hash (these hashes will be termed as "partial hash") the same principle applies. This is because the ranges of responsibility of any index in a binary indexed tree can be expressed by the range $[i - LSB(i), i]$, making any partial hash equivalent to the sum of the last $LSB(i)$ values in the polynomial representing a prefix hash of length i. This can be seen in the case of index 6 as follows, where it will store the sum of the last two components of the full prefix hash of length 6 because its LSB is equal to two. The partial hash and the prefix hash is aligned to highlight how the partial hash is a latter portion of the prefix hash.

$$\begin{aligned} \textit{Partial Hash} \ &= a_5 \times p^{6-5} + a_6 \times p^{6-6} \\ &= a_5 \times p^1 + a_6 \times p^0 \end{aligned}$$

$$\textit{Prefix Hash of length } 6 = a_1 \times p^5 + ... + a_4 \times p^2 + a_5 \times p^1 + a_6 \times p^0$$

Using this observation about the the powers of p, the construction of the binary indexed tree follows the same algorithm as described in **section 3.2.3** but instead of simply adding on a value, an additional factor of $p^{i_{current} - i_{start}}$ is applied, where $i_{current}$ describes the current index being added using the update operator and i_{start} describes the index that we started at when the update operator was first called.

Although the above maintains our binary indexed tree to store polynomial hashing, it does not yet return accurate results for querying the hash of a range. Let $prefix\_hash(r)$ denote the operation for querying a prefix hash of length r using a binary indexed tree. If the function $prefix\_hash(6)$ were to call, it should return a value equal to *BIT[6] + BIT[4]* (see **section 3.2.1** about why), if no further processing is applied, which can be expressed by

$$BIT[6] + BIT[4] = (a_5 \times p^1 + a_6 \times p^0) + (a_1 \times p^3 + a_2 \times p^2 + a_3 \times p^1 + a_4 \times p^0)$$

However, accurately, $prefix\_hash(6)$ should return a prefix hash of length 6 and should be expressed as

$$prefix\_hash(6) = a_1 \times p^5 + a_2 \times p^4 + a_3 \times p^3 + a_4 \times p^2 + a_5 \times p^1 + a_6 \times p^0$$

Rearranging the first equation, it can be seen that

$$BIT[4] + BIT[6] = (a_1 \times p^3 + a_2 \times p^2 + a_3 \times p^1 + a_4 \times p^0) + (a_5 \times p^1 + a_6 \times p^0)$$

Comparing the two equations, it can be seen that in the first equation, $BIT[4]$ is offset by a multiple of $p^2 = p^{6-4}$; if $BIT[4]$ was scaled by p^2, the two equations would be equal. In general, all indexes being considered in a query to compute a prefix hash up to index i_{start} need to be scaled by a factor of $p^{i_{start} - i_{current}}$, where $i_{current}$ denotes the index that is currently being considered in the query.

4 Conclusion

This essay explored two different ways of comparing arrays when the values of the array are being updated, one more efficient running in $O(QlogN)$ time complexity, while the other being less efficient running in $O(QN)$.

5 Citations

1. maxcruickshanks. (n.d.). Comparing arrays - DMOJ: Modern online judge. DMOJ. https://dmoj.ca/problem/comparingarrays

2. Moore, K., Chumbley, A., & Khim, J. (n.d.). Rabin-Karp algorithm. Brilliant Math & Science Wiki. https://brilliant.org/wiki/rabin-karp-algorithm/

3. TimonKnigge. (2014). How do I find hash of any sub-string in O(1) by O(N) preprocessing. Codeforces. https://codeforces.com/blog/entry/18407

4. cp-algorithms. (2024, January 24). Fenwick tree. Fenwick Tree - Algorithms for Competitive Programming. https://cp-algorithms.com/data_structures/fenwick.html

5. Fenwick, P. (1994, March 1). [PDF] a new data structure for cumulative frequency tables — semantic ... https://www.semanticscholar.org/paper/A-new-data-structure-for-cumulative-frequency-Fenwick/769fb8055fbe0997ef8d9dab6c9abf37489c6575

6. Fiset, W. (2017, June 9). Fenwick Tree Construction. YouTube. https://www.youtube.com/watch?v=BHPez138yX8

7. Parr, K. (2021, February 1). What is endianness? big-endian vs little-endian explained with examples. freeCodeCamp.org. https://www.freecodecamp.org/news/what-is-endianness-big-endian-vs-little-endian/

8. Finley, T. (2000, April). Two's Complement. Two's complement. https://www.cs.cornell.edu/ tomf/notes/cps104/twoscomp.html