DYNAMIC SUBSTRING COMPARISON PROBLEM: SEGMENT TREE APPROACH

ALEKSANDR MAKSIMOV

Abstract. The problem of comparing some arbitraty substrings in a given string is a well-researched theoretical Computer Science problem solvable with methods like polynomial hashing or the Knuth-Morris-Pratt algorithm. This essay focuses on dynamic variants of this problem (with point/range update queries support), considering segment trees and their modifications to keep the complexity on the standard binary search tree level (O(log n)) for both types of queries. The range update is discussed in detail.

1. Introduction

The problem of string comparison and pattern matching has been well-researched with various statements, requirements and backgrounds. One of the first steps in this area has been done by Knuth, Morris and Pratt [1], with advanced suffix structures and preprocessing algorithms to come in the next 40 years. This paper is aimed specifically to reinforce our knowledge about some simpler methods which can solve the variations of the problem efficiently, rather than to propose something specific and new.

2. Background

The only practical problem known to the author has been seen in the 2022 Summer Informatics School camp, with a slight modification of numbers used instead of latin characters. The method provided in this paper is well-extendable and generalisable to cover this case.

To our knowledge, this is the first official paper covering the particular variants of the substring comparison problem.

3. Definitions

3.1.1. *Alphabet: all lowercase latin alphabet characters.*

3.1.2. *Query: a request for the running code to output some data. The data can be output right after the query is processed or after the program terminates.*

3.1.3. *Indexing: the way the elements of the string are numerated. This paper uses 1-indexing for problem descriptions (first element in the string has number 1) and 0-indexing for implementation.*

3.1.4. $log(n)$ *denotes* $log_2(n)$ *if not stated otherwise.*

4. Problem statement

The problem discussed in this particular publication will be split into 3 parts, arranged by relative difficulty and complexity of approaches:

4.1. **Static substring comparison problem.** Given a string S of length n and k queries (l_1, r_1, l_2, r_2) $(1 \leq l_1, r_1, l_2, r_2 \leq n)$. For each query, determine whether or not the corresponding substrings of S are equal to each other. More formally, for each integer $i \mid 0 \leq i < (r_1 - l_1)$ the following is true: $S_{l_1+i} = S_{l_2+i}$. Note that the substrings can be represented by intervals $[l_1, r_1]$ and $[l_2, r_2]$.

4.2. **Dynamic point update substring comparison problem.** The "check for equality" queries have the same format as above; the new query is denoted by (p, x) $(1 \leq p \leq n)$, meaning that S_p has to be set to x, where x is an arbitrary latin alphabet lowercase character.

4.3. **Dynamic range update substring comparison problem.** The "update" query has format (l, r, x), meaning that all elements S_i where $l \leq i \leq r$ have to be set to x.

It is easy to see that correct solutions for each of the subproblems are able to solve all previous ones (not necessarily with same time complexity).

5. Polynomial hashing

Let's consider the first subproblem. The precomputation is allowed, so it is possible to implement polynomial hashing algorithm. This paper won't go into much detail about specific use-cases and implementation of polynomial hash functions, although the idea will be shown. Turns out it is possible to represent strings as numbers of base equal to the size of the alphabet (normally a prime number bigger than 26, e.g., 29, 31 or 37).

5.1. **Definition: $\mathrm{hash}(S, p) = \sum_{i=0}^{p} S_i \cdot b^{p-i-1}$, where S_i is converted to an integer by taking the position in the alphabet, and b is the chosen base $(b > 26)$.**

$\mathrm{hash}(S, p)$ computes the polynomial hash of the first p characters of S (prefix of S). It is proven that no same hash values can be obtained from another values of S and p. The reason why we define the prefix hash function here will be clear later as we generalise prefixes to substrings.

5.2. **Claim: the strings are equal if and only if their hash values are equal.**

The practical problem is that for long prefixes the hash values become very large and are hard to compare, and at a certain point the numbers become longer than the substrings being analysed. This affects time complexity. Luckily, there has been found a solution to this - using the $\mathrm{hash}(S, p) \bmod q$, where mod denotes the remainder operation. This creates a risk of a hash-collision (two different substrings have the same hash because of the birthday paradox) but makes hashes practical. From this point, the $\mathrm{hash}(S, p)$ notation includes the remainder operation by a certain big prime number q.

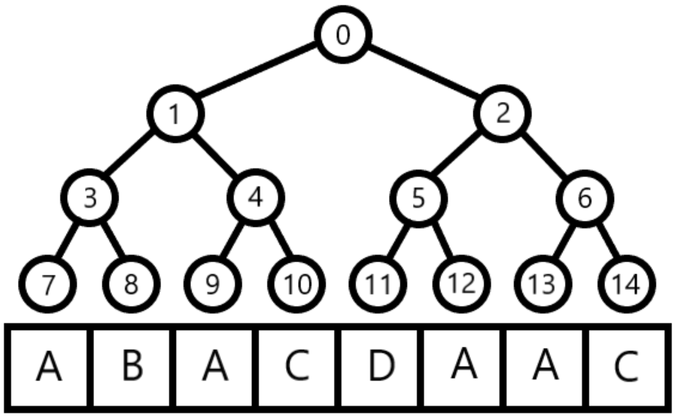
In order to find the hash of a certain substring $[l, r)$, we need two values: $h_l = \mathrm{hash}(S, l)$ and $h_r = \mathrm{hash}(S, r)$. Then all is needed is multiplying h_l by b^{r-l} and subtracting that from h_r (similarly to prefix sums). All the operations should be done mod q. The proof why this is true is left as an exercise for a reader.

Now it's easy to solve the first subproblem: precompute all hashes on prefixes (this can be done in $O(n)$ time, as $\mathrm{hash}(S, p) = \left(b \cdot \mathrm{hash}(S, p-1) + S_p\right) \bmod q)$) and then answer each query by comparing the hash values and string lengths in $O(1)$ per query (if all powers of b are precomputed in $O(n)$. With the alphabet as small as 26 characters and the length of the string not exceeding 10^5, it's very hard to create a test which will have some substrings with same length and hashes but different values. Even if such test is created, creating another hash function with a different prime base and modulo value will fix the problem of so-called AHT (anti-hash tests). The hashes and the creation of AHT has been covered by Pachocki and Radoszewski [2].

6. Segment tree for point updates

Prefix polynomial hashing has a flaw of necessity to recompute all subsequent elements after a certain element is updated. This can be fixed by building a segment tree on the string and updating the hash in the preceeding nodes after a point update.

A segment tree is a binary tree with nodes, each of which represents a subsegment of the array. The root of the tree has the hash value of the whole array; its two children have the values from the left and right halves. In the segment tree construction process, the current subsegment is split up in 2 halves and each half is processed recursively until the current subsegment has only one element, which becomes a leaf of the tree. The parent of two vertices has to get its own value in the construction process, making it necessary to define a merge(l, r) function which joins the value of two child vertices. In case of a sum segment tree, the merge function just adds the values of the vertices. Hash values have to be merged in a different way.



. Figure 1: An example segment tree structure for a string

Please note that all examples below are based on Figure 1 if not stated otherwise.

Although this paper doesn't cover the principles of the segment tree (which has been done by A. Laaksonen in his book [3]), we will cover the basics and specific details of the implementation. Each node has to store the hash value on the corresponding subsegment. Node 3 (Figure 1) stores the hash of “ab”. From an earlier claim, the hashes of the same strings are equal, meaning that node 4 and node 6 have the same hash value as the substrings they are covering are equal. For the sake of concise and easy implementation, each node is mapped with a corresponding semi-interval $[l; r)$ (thus challenging the name of the structure as it had to be named as a semi-interval tree). The root 0 covers $[0; 8)$, its children 1 and 2 cover $[0; 4)$ and $[4; 8)$. Now it is easy to see why such method of division has been chosen: the implementation avoids adding and subtracting 1 from the interval coverage as the end of the first interval is the start of the second. Note that 0-indexing is used.

Now to the actual querying part: how to get the hash of a specific subsegment? This can be done by merging the hash values of all subsegments that are part of the query.

Suppose it is necessary to compute the hash of “bac” ($[1; 4)$). Nodes 8 and 4 are parts of the answer; it is easy to see that the same subsegment can’t be obtained by merging less than those 2 nodes. The hash value in node 8 is “b” $\cdot b^0$, and in node 4 it is “a” $\cdot b^1 +$ “c” $\cdot b^0$. The whole subsegment hash is, by definition of $\text{hash}(S, p)$, “b” $\cdot b^2 +$ “a” $\cdot b^1 +$ “c” $\cdot b^0$. It turns out the merged hash can be obtained by adding the second hash and the first hash multiplied by b^l, where l is the length of the second subsegment. This “shift” produces the actual result.

This observation makes it possible to handle the point update problem: when an element is updated, its hash value is updates on a bottom-level node. Then all the ancestor nodes are updated recursively according to the change in their children nodes. There are $\log(n)$ ancestor nodes to update (because any binary search tree has $\log(n)$ levels) making it $O(\log n)$ per update query. The hash calculation query is $O(\log(n))$ as well because there are only $O(\log(n))$ segments which form the segment from the query. Any comparison query requires 2 hash computations and one $O(1)$ integer comparison, resulting in O(log(n)) per comparison query. The precalculation takes $O(n)$ time as there’s only $O(2 \cdot n) = O(n)$ nodes to compute the values for, and any $\text{merge}(l, r)$ takes $O(1)$ time.

7. Segment tree modification for range updates

The current variant of our segment tree can only handle point updates. The range updates can only be done by updating the elements one by one, which takes $O(n \log(n))$ per range update query. There has been found a modification [4] for the segment trees which makes it possible to handle range updates by storing some additional information of what has to be stored in the nodes and using it when necessary. When the range is updated, all segments that are fully in this range are assigned a value for that update query. This value is later used if the segment is a part of the hash query. A push(l, r) function is defined to update the children’s assigned values if the children’s hash values have to be updated separately. This allows to keep the complexity at $O(\log(n))$ per update and hash queries, because some unused assigned values get discarded in the process if not used. See implementations (Appendix 1) for more details.

The main problem of this modification comes from the necessity to recompute the hash of a node that has been updated such that all leaves in its subtree contain the same letter. In other words, we have to compute the hash of a string of the same letters, knowing only the letter and the length of the string. My solution can compute this in $O(\log(k))$ time, where k is the length of the string. This complexity is later optimised to just $O(1)$ using precomputation.

7.1. Subproblem: computing the hash of uniform strings.

The hash of the string with length n will have a form of

$$(\text{char} \cdot b^{n-1} + \text{char} \cdot b^{n-2} + \cdots + \text{char} \cdot b^1 + \text{char} \cdot b^0) \bmod q$$

(where char is an arbitrary character that the string consists of). It is easy to see that the formula in the brackets is a rearranged geometric progression with common difference b and first term char. The sum of the terms is

$$\left(\frac{\text{char} \cdot (b^n - 1)}{b - 1}\right) \bmod q$$

The division is what makes the computation untrivial because it is not possible to perform division without specific methods while working in the ring modulo q. Fortunately, as $\gcd(b, q) = 1$ as q is prime, it is possible to use Fermat’s Little Theorem [5]. In order to divide by $b - 1$, it is needed to multiply by the reciprocal element in the ring modulo q. By the Fermat’s Little Theorem,

$$(b-1)^{-1} \bmod q = (b-1)^{q-2} \bmod q$$

This makes it possible to perform the calculation. The complexity of 2 exponent operations performed (one in the reciprocal and one in the GP sum formula) add up to $O(\log(k))$ in the update query complexity. But this is easy to optimise by precomputing all possible base powers up to n where n is the length of the string and accessing them where needed. This takes $O(n)$ time. The reciprocal element can't be found by just accessing the precomputed powers (because $q-2$ is too large to be found in the precomputed vector), but it can be computed once in $O(\log(n))$. The final precomputation, along with the segment tree construction process, remains at $O(n)$.

8. Summary on known solution complexities

Method	**Precalculation**	**Hash computation**	**Point update**	**Range update**
Naïve approach	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Polynomial hashing	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Point update segment tree	$O(n)$	$O(\log(n))$	$O(\log(n))$	$O(n\log(n))$
Range update segment tree	$O(n)$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

9. Benchmarking and measuring practical performance

9.1. Test data generation.

The benchmarks have been done on 2 types of testsets:

- QP testset: 10^6 queries, all input string lengths from 10^4 to 10^6 with a step of 10^4.
- LN testset: input string length is 10^6, all numbers of queries from 10^4 to 10^6 with a step of 10^4.

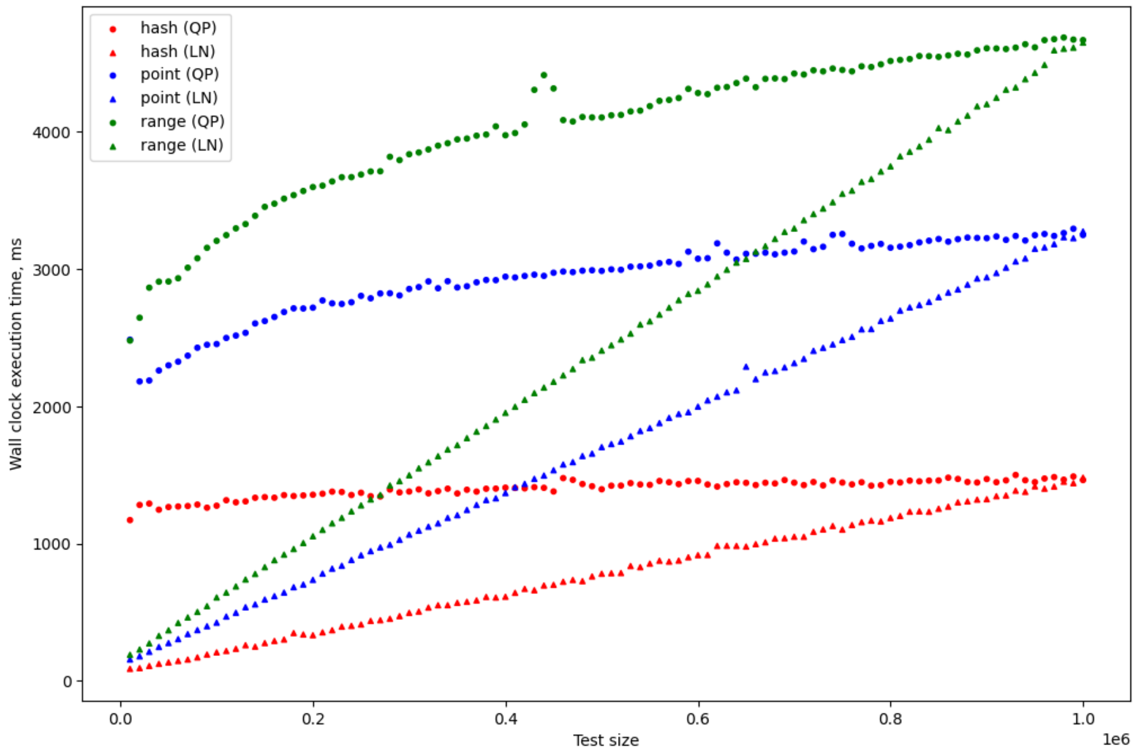
Each problem has been benchmarked separately to measure different methods' performance. Each implementation was first validated using one of the tests from the testset to ensure that the output is the same and there is no bugs or accidental errors in the tests.

The tests themselves only contain hash computing, without actually comparing the substrings - this way it is easier to create "fair" big tests. This doesn't have any effect on complexity as each of the methods just compares two substring hashes after computing them.

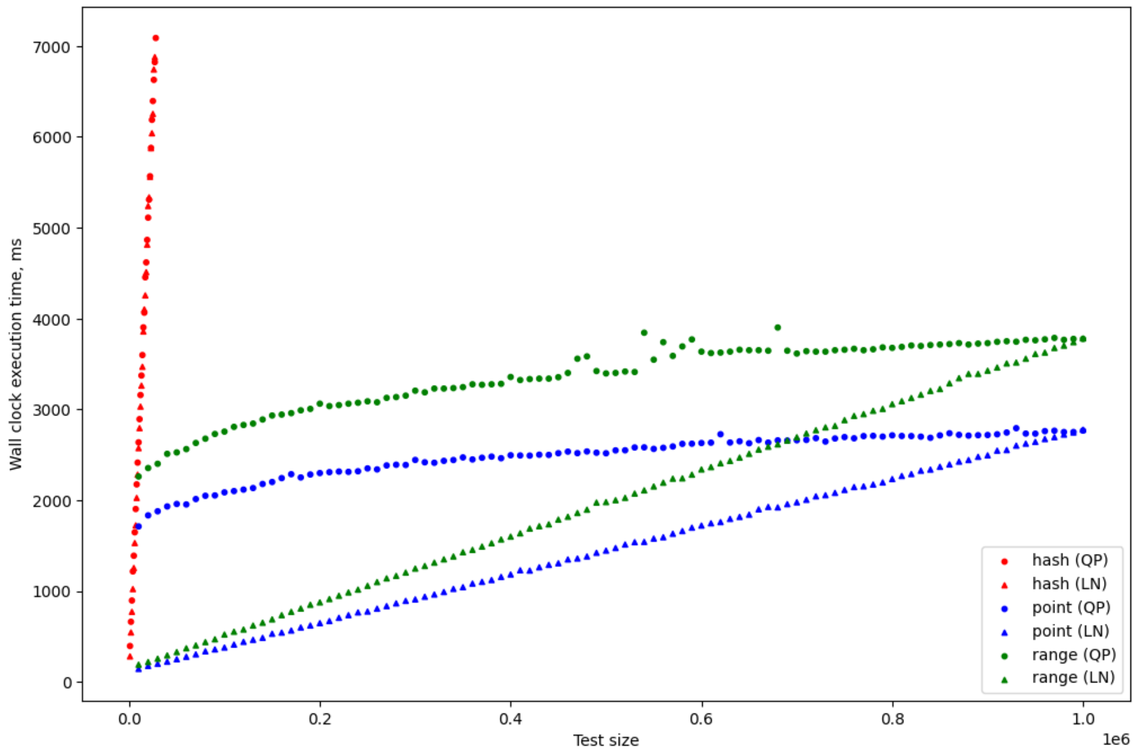
9.2. Specific implementation details.

All implementations use a single-hash technique. This means they're not completely collision-proof [2], but the tests are generated randomly and are unlikely to cause collisions. This has been done to keep the implementations readable and simple enough, but all of them are easily generalisable to use multiple hashes.

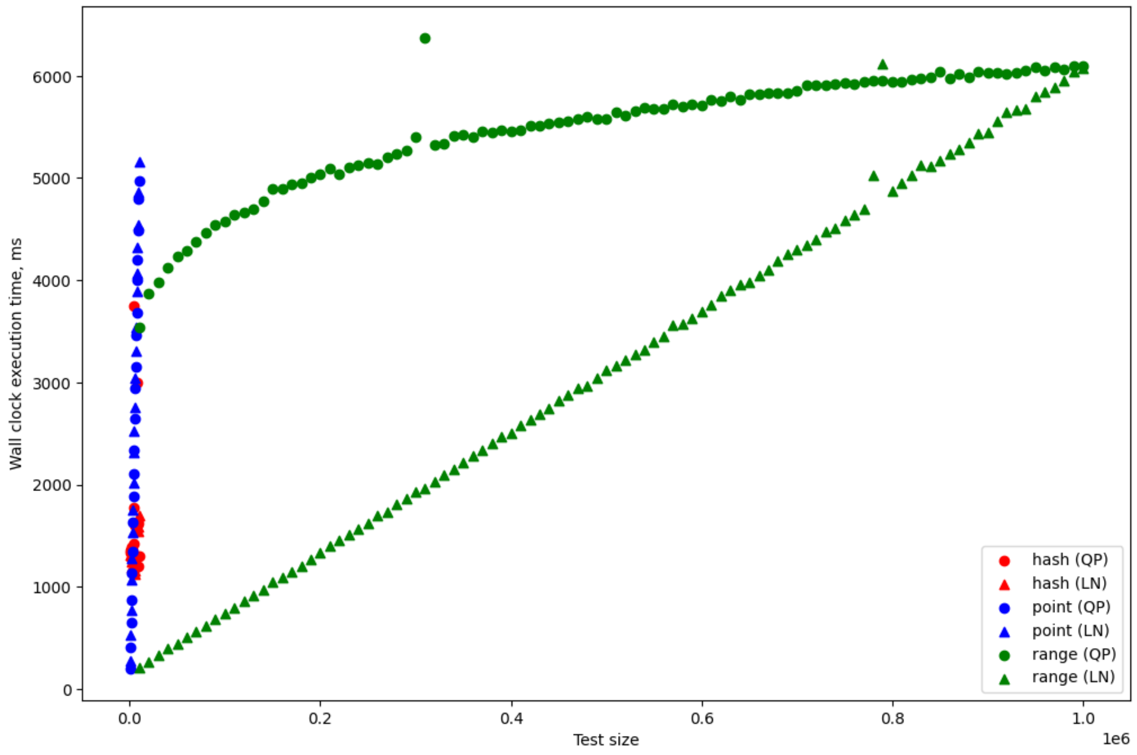
9.3. **Benchmarking results.**



. Figure 2: Hash query benchmark



. Figure 3: Hash query/point update benchmark



. Figure 4: Hash query/range update benchmark

9.4. **Evaluation.**

9.4.1. *Static problem analysis.*

The static problem is best solved by the canonical polynomial hashing method. Its QP graph is almost flat because the precomputation of hashes on prefixes is done only once, and the fixed number of queries is processed in the same amount of time. The LN graph is a straight line for all methods because it only depends on the original input size and scales proportionally to the number of queries.

The QP graphs for the segment tree implementations are similar to logarithmic graphs. This can be explained by the fact that the time of the query execution is dependent on the **height of the segment tree**, and, consequently, the length of the original input. The difference between the point update and range update segment trees graphs is a consequence of the base precomputation for different lengths of the input, but it is insignificant compared to processing queries.

The LN graphs have a different incline because of the constant push(l, r) overhead on each of the queries. The hash computation for the polynomial hash is $O(1)$, so the incline is very small compared to the difference in the input sizes (but it is still noticeable, partially because modulo (%) operation is computationally time-consuming). As the range update segment tree executes a push(l, r) on each node processing, it has a constant overhead which is only dependent on the number of the nodes processed per query (which is always $O(\log(n))$, where n is the constant length of the string).

9.4.2. *Point update problem analysis.*

The hash implementation is inefficient on this problem as each update takes $O(n)$ time, where n is the length of the input. Both graphs (QP and LN) are almost identical because the update queries are dependent on the length of the string, and even when the number of queries is fixed, the string can be large enough to have a lot of update queries which affect the execution time. Even for tests of size up to 10^5 the hash implementation can't manage to run for under 7 s (7000 ms) per test, which makes unnecessary to continue the measurements.

The segment tree implementations perform as expected with the same curve shapes (as the update and hash calculation queries have identical theoretical complexity and don't differ much in practice, so they can be considered similar in terms of the effect on the graph). The constant overhead is still noticeable (because the pushes are not used to transport information as the subtrees that the values are assigned to are always leaves of the tree, but the pushes still have to be executed to keep the invariants in the vertices).

9.4.3. *Range update problem analysis.*

Both hashing and point update algorithms perform very poorly with an almost vertical line as the graph. Their execution had to be stopped even on small tests because the time exceeded 2 minutes on a single testset. However, the range update segment tree with lazy propagation has shown a good result with only 6 seconds on maximal tests. The shape of both graphs formed by points is explained above.

9.4.4. *Practical measures summary.*

Although it may look like the no-push (point update) segment tree can be completely replaced by the segment tree with pushes because they have the same theoretical time complexities, the latter performs worse in practice because of constant query overheads. Each implementation (hashing, segtree and push segtree) performs best on the corresponding subproblem.

10. Conclusions

The segment tree hash calculation algorithms provided in this paper have been able to solve more advanced pattern-finding and substring comparison problems more effectively than the standard prefix hashing method. The range update string comparison problem has been effectively solved using the modified lazy propagation segment tree, using mathematical properties of hashing and modulo rings. This has allowed to solve the dynamic substring comparison problem by executing the hash calculation queries on the corresponding subsegments and comparing the results. The complete working implementations for all three methods can be found on GitHub[1].

[1]The code and accompanying generation scripts are on https://github.com/maleksware/hash-segtree.

References

[1] D. Knuth, J. Morris, and V. Pratt, "Fast pattern-matching in strings," 1977.

[2] J. Pachocki, and J. Radoszewski, "Where to use and how not to use polynomial string hashing," 2013.

[3] A. Laaksonen, *Guide to Competitive Programming*, Springer, 2017.

[4] J. Kogler, "Segment tree." https://cp-algorithms.com/data_structures/segment_tree.html

[5] R. E. Bishop, "On fermat's little theorem." [Online]. Available: https://math.uchicago.edu/~may/VIGRE/VIGRE2008/REUPapers/Bishop.pdf

CodeCadets, Canberra Grammar School
Email address: maleksware@gmail.com
URL: https://maleksware.github.io