A Case Study in Optimization of Brute-Force Algorithm : Finding Longest Substring With Unique Characters

Fatih Nararya Rashadyfa Ilhamsyah - 13521060
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13521060@std.stei.itb.ac.id

***Abstract*—This paper examines how a brute-force algorithm could provide deeper insights to a problem by attempting to solve the problem of finding the longest substring with unique characters**

Keywords—brute-force; algorithm; language; string; subtstring

NOMENCLATURE

S	A string of characters
S_u	A string with all unique characters
s	A substring of another string S
s_u	A substring of another string S with unique characters

I. INTRODUCTION

In the field of algorithmics, there are plenty of algorithm design strategies. Brute-force algorithm is one of them and perhaps one of the most overlooked. This is not without reason as brute-force algorithms are generally not at all smart and thus, inefficient. Nevertheless, brute-force algorithms are still an important starting point for one to understand a problem down to its quirks in order to design a faster algorithm, even if the resulting algorithm is still a brute-force algorithm at heart. This paper then attempts to show how one could utilize keen observation and deeper understanding about a problem to optimize an initial approach by brute-force algorithm. This paper also shows that by first making a brute-force algorithm to solve a particular problem and seeing how the algorithm runs during the search for a solution of a problem instance, one may come to the aforementioned understanding that could be used to design a better algorithm. To demonstrate this, the paper will take one particular problem of finding what is the longest unique substring of a given string.

As an important note, the problem used here will only give the string to be checked without defining what is the character set that is used by the string. This assures that cheap optimization of the algorithm by exiting once a unique substring consist of all characters in a character set is not an option and thus forces the algorithm to use smarter optimization.

II. THEORETICAL BASIS

A. Substring of a String With Unique Characters

It's important to distinguish and make clear the distinction between the terms used in the nomenclature of this paper by giving some examples. "kmnlopqrt" is an S (a string) and an S_u (a string with all unique characters) because it's a string that does not repeat any characters. "kkopok" is an S but not an S_u because it repeats the character 'k'. "opok" is an s (a substring, in this case a substring of "kkopok") and s_u (a substring that has unique characters) of "kopok" because it's a substring (slicing from index 2 to 5 inclusive). "kopok" is an s but not an s_u of "kkopok" because while it's a substring (slicing from index 1 to 5 inclusive) it contains a repetition of the character 'k'. The s and s_u nomenclature will be used within the context of a certain S just like the example shown here as its existence only makes sense within the context of a "parent" string.

B. Naive Algorithm

Naive algorithms are algorithms that are the most obvious solution to a given problem. Because of that, naive algorithms usually have very simple rules – which makes them easy to implement – but, as the name suggests, they typically aren't smart. Naive algorithms usually are not the fastest possible algorithm to solve a problem (although there are certain problems that only have naive algorithm solutions). A common example to delineate between naive algorithms and non-naive algorithms is the problem of string-matching. The naive algorithm approach to string-matching would compare every character in a pattern P of text T sequentially from the start. When a mismatch is found, P would be shifted by one position and checking would be repeated. This is done until the match is found or the string ends. The naive algorithm has a complexity of $O(mn)$ in the worst case where m is the length of P and n is the length of T. In contrast, smarter algorithms for string-matching such as Knuth-Morris-Pratt algorithm or Boyer-Moore algorithm could find a match within a string in $O(n)$.

A huge subset of naive algorithms is brute-force algorithms. Brute-force algorithm is a class of algorithms that first generates all possible solutions to a problem and then tests whether that candidate solution satisfies the given instance of the problem. Brute-force algorithms are guaranteed to produce a correct solution and are simple in theory, making them very easy to implement. However, the biggest downsides to this class of algorithm is its inefficiency. For a problem with $g(n)$ where n is the input size of the problem and a complexity of checking each solution as $O(f(n))$, the complexity of a brute-force algorithm for that problem would be $O(f(n) * g(n))$. As such, a problem that has an enormous amount of possible solutions in proportion to its input size (which describes a lot of problems) would quickly render a brute-force algorithm as impractical. A popular example of combinatorial explosion in the complexity of a brute-force algorithm is the Traveling Salesman Problem. The problem asks that for an n amount of cities that are all connected with each other with a certain distance, what would be the shortest route that goes through each of the cities only once? There are $n!$ possible solutions (owing to permutation) and the complexity to evaluate each solution is $O(n)$. This means that the brute-force algorithm would have a time complexity of $O(n * n!)$, which are already unbearable for small n.

A common method to make a faster algorithm out of naive algorithms is by utilizing heuristics or observation about the problem to cut unnecessary computation. The aforementioned example of Boyer-Moore algorithm and Knuth-Morris-Pratt algorithm are a prime example of this where they managed to have a better time complexity by not merely shifting by one position on a mismatch but instead have variable size shifts based on P, T, and circumstances of the mismatch. The optimized algorithm to be presented later on this paper will optimize the naive algorithm for finding longest s_u of an S using this method.

III. Algorithms

A. Naive

The initial solution that most likely came to mind when presented with the problem of finding the longest substring with all unique characters from a string would likely be through the steps below :

1. Start a pointer A and B from the start of the string and two variables : a counter starting from zero and a highest substring length also starting from zero. Pointer A would represent the start of the substring being examined.
2. Move the pointer B forward.
3. If the letter that is at pointer B already existed before, reset the information about the letter contained in the substring and move pointer A forward by one. Reset pointer B to the position of pointer A. If the substring that was examined is longer than the previous substring, update the corresponding variable. Reset the counter variable. This represents moving on to a new substring.
4. If the letter that is at pointer B has not existed before, save the letter and increment the counter.

The pseudocode of this naive algorithm is presented below :

NaiveSearch Algorithm

```
Local dictionary :
letterOccurence : A hashmap with character as a
key and boolean as a value
longest : the longest substring with unique
characters yet
s : the analyzed string
```

```
def naiveSearch(s)
        longest = 0
        for i to s.length
                j = i
                foundRepeating = false
                counter = 0
                        while not foundRepeating
                    and j < s.length do
                        if letterOccurence[s[j]] is
                    true then foundRepeating = true

                        else letterOccurence[s[j]]
                    = true
                        j = j + 1
                if j - i + 1 > longest then
                        longest = j - i + 1
        return longest
```

And below would be a concrete example of the code written in the Java programming language.

NaiveSearch Algorithm in Java

```
public int lengthOfLongestSubstring(String s) {
        int longest = 0;
        for (int i = 0; i < s.length(); i++) {
            int j = i;
            boolean foundRepeating = false;
            StringBuilder sb = new
StringBuilder();
            HashMap<Character, Boolean> table =
new HashMap<>();
            while(!foundRepeating && j <
s.length()) {
                if
(table.containsKey(s.charAt(j))) {
                    foundRepeating = true;
                } else {
                    table.put(s.charAt(j), true);
                    sb.append(s.charAt(j));

                }
```

```
                j++;
            }
            if (sb.length() > longest) {
                longest = sb.length();
            }
            sb = new StringBuilder();
        }
        return longest;
    }
```

The algorithm is faster when the given string has many occurrences of the same character repeated consecutively or when most substring with the unique characters are very short in length as there would seldom be repetition of character comparison. The best case for this algorithm would be a string that only uses one character repeated throughout the string, such as 'kkkkkkkkkk'. In that case, the time complexity of the algorithm approaches $O(n)$ as the algorithm degenerate into a simple traversal of the whole string.

And the opposite is also true. The algorithm will become slower when the given string has a lot of substring with unique characters that are long in length. The worst case would be when the whole string itself does not repeat any characters. In that case, the time complexity of the algorithm would seem to be $O(nm^2)$, n as the length of the string and m as the number of characters in the character set used by the string, as for every i-th characters of the string, there would be at most m comparison to be done. The reason behind the square for the m will be explained in the next section.

B. Optimization

Some simple optimization arising from simple observation about the problem would now be presented to the previous algorithm. The optimization made are the following :

1. Once pointer B already reaches the end of the string as a whole, searching can be stopped as no remaining substring would be longer than the current substring
2. Once a letter X at pointer B is found to already occur in the currently examined substring, instead of shifting the pointer A by just one, it could be shifted to the position after the occurrence of the letter X in the currently examined substring.

The reasoning behind the first optimization is self-explanatory and does not require any explanation. The reasoning behind the second optimization is better explained through an example. Fig. 1 is a snapshot of the algorithm running on an instantiation of the problem using the string "popoapck". The position of pointer A is highlighted in bold and pointer B is highlighted with an underline.

p	**o**	p	o	a	p	c	k

Fig. 1. Pointer B meeting 'p', a character that had occurred before in the substring

Under the naive algorithm, the pointer A would be shifted by one and the pointer B would be reset into the position of pointer A as seen in Fig. 2. The substring "opoa" would have its length become the longest length of the substring with unique characters found so far.

p	o	**p**	o	a	p	c	k

Fig. 2. Pointer A and B being reset after meeting character 'p' that had occurred before.

Once the algorithm resumes, it will be in the position at Fig. 3 after a few steps.

p	o	**p**	o	a	p	c	k

Fig. 3. Pointer B meeting the character 'p' at the same position as previously

As can be seen the algorithm meets the same 'p' that stopped it before. But notice that because the algorithm only shifted pointer A by one, the substring that was just examined, "poa" is actually just a substring of the previously examined substring of "opoa". Thus, it can be seen that the examination of the substring "poa" is of no purpose as it's not a substring that is longer than the previously examined substring "opoa".

Another example of this occurrence is better exemplified with the string "wopkkelp". The first time the pointer B would have to be reset, the two pointers would be in the position of Fig. 4.

w	o	p	k	k	e	l	p

Fig. 4. Pointer B meeting 'k', a character that had occurred before in the substring

On the second time it's reset, it will be in the position of Fig. 5.

w	**o**	p	k	k	e	l	p

Fig. 5. Pointer B meeting 'k' at the same position as previous examination

On the third time, it will be in the position of Fig. 6.

w	o	**p**	k	k	e	l	p

Fig. 6. Pointer B meeting 'k' at the same position as previous examination

The previous example had established the existence of wasteful examination without giving much clue on what exactly causes it. This example now illustrates perfectly the condition that leads to a wasteful examination. As seen above, it's obvious that examining 'opk' and 'pk' is wasteful because it's just a substring of the first examined substring and thus guaranteed to not be longer than the original.

The observation can then be formulated as the following. Suppose that at some point, the algorithm is stopped at position p, which causes it to shift the pointer A by one and move pointer B back to A, let's call the substring that were found by the examination s_{u1}. All of the substring examined subsequently whose examination are stopped at the position p

are guaranteed to be shorter than s_{u1}, examining them is a wasteful endeavor.

This is the reasoning behind the square for the m in the time complexity of the naive algorithm. In the worst case, the algorithm find the longest substring with unique characters (at most m characters long) and then incrementally cuts the substring by one until it's one character long repeating the same computation. This then yields m^2.

The aforementioned observation results in the second optimization. The second optimization skips the wasteful examination by moving pointer A forward to the first position where the algorithm would not be stopped at the same position that had stopped it before. This is better illustrated through examples. The previous examples will be used to present contrast between the naive algorithm and the optimization.

Using the first example of string "popoapck", after Fig.1 occurred, the algorithm would move instead to the position of Fig. 7.

p	o	p	**o**	a	p	c	k

Fig. 7. Pointer A and B being reset under the optimized algorithm

As illustrated in Fig. 7, the algorithm skipped the wasteful examination of the substring 'poa', instead moving forward the pointer A to the first position where it would not be stopped by the character 'p' at the same position as previously.

Using the second example of string "wopkkelp", after Fig.2 occurred, the algorithm would move instead to the position of Fig. 8.

w	o	p	k	**k**	e	l	p

Fig. 8. Pointer A and B being reset under the optimized algorithm

The algorithm here skips examination of 'opk', 'pk' and 'k' by moving pointer A forward to the first position where it would not be stopped by the character 'k' at the same position as previously. Where such a position is located is after the occurrence of the character that stopped the algorithm in the examined substring. Because the character that stopped the examination is 'k', the algorithm checks where in the currently examined substring the character 'k' appears, which is at index 3. Thus, the pointer A then gets shifted to index 4. To better illustrate this, suppose that the string examined instead is "wopkpelp". When the algorithm first gets its pointer reset after being at position as shown by Fig. 9,

w	o	p	k	**p**	e	l	p

Fig. 9. Pointer A and B being reset under the optimized algorithm

it would be reset to Fig. 10. Because the character 'p' first appears at index 2.

w	o	p	**k**	p	e	l	p

Fig. 10. Pointer A and B being reset under the optimized algorithm

For the sake of brevity, the optimized algorithm would only be presented in Java.

OptimizedNaiveSearch Algorithm in Java

```
public int lengthOfLongestSubstring(String s) {
        if (s.length() ⩽ 1) {
            return s.length();
        }
        int longest = 0;
        boolean reachedTheEnd = false;
        int i = 0;
        int j = i;
        int currentLength = 1;
        HashMap<Character, Integer> table = new
HashMap<>();
        table.put(s.charAt(i), 0);
        while (!reachedTheEnd) {
            j++;
            reachedTheEnd = j ⩾ s.length() - 1;
            boolean sameCharacterFound = false;
            Character observed = s.charAt(j);
            Integer charValue =
table.get(observed);
            if (charValue == null || charValue ==
-1) {
                table.put(observed, j);
                currentLength++;
            } else if (!reachedTheEnd) {
                int newI = table.get(observed) +
1;
                for (Character c : table.keySet())
{
                    if (table.get(c) < newI &&
!c.equals(observed)) {
                        table.put(c, -1);
                    }
                }
                i = newI;
                table.put(observed, j);
                sameCharacterFound = true;
            }
```

```
            if (currentLength > longest) {
                longest = currentLength;
            }

            if (sameCharacterFound) {
                currentLength = j - i + 1;
            }

            reachedTheEnd = j ⩾ s.length() - 1;

        }
        return longest;
    }
```

In the optimized algorithm, the hashmap is not just used to memoize what characters have occurred in the examined substring but also on what index does the character occur.

One of the qualities of this optimization is that pointer A never moves backward, only forward. This makes the optimization suitable for continuous stream of data. It is also reminiscent of how the KMP algorithm optimized the naive algorithm of string matching.

The complexity of this algorithm will be $O(n)$ in the best-case and $O(nm)$ in the worst-case. This is because in the event that there are many long substrings, the algorithm would not repeat redundant comparisons for the substring, but instead move on to a position that allows it to find a new substring. However, the algorithm does have to reset the content of the hashmap in the event of a pointer reset (see the for loop in the while loop of the code snippets given above) which explains the existence of the m term.

IV. Empirical Analysis

A. Random Alphabetical Texts

The two algorithms will be tested alongside a simple traversal of each character as control to see how effective the optimization that was done. The string used here consists of only alphabetical characters both lowercase and uppercase. Each of the times here is an average of 10 runs. The test is done on a Ryzen 7 2700X at 4.00GHz with 16GB of RAM.

TABLE I
TIME FOR SIMPLE TRAVERSAL

String Length (n)	Time for Simple Traversal (ns)	Time for Naive Algorithm (ns)	Time for Optimized Algorithm (ns)
10	$27,154$	$50,341$	$11,789$
10^2	$265,557$	$307,531$	$246,113$
10^3	$1,803,948$	$2,140,813$	$1,741,685$
10^4	$4,893,742$	$4,334,419$	$4,277,883$
10^5	$13,100,485$	$22,170,625$	$12,050,632$
10^6	$121,531,095$	$152,093,631$	$120,825,727$

How fast both of the algorithms against the simple traversal (as control) is as follows.

TABLE II
TIME OF NAIVE AND OPTIMIZED ALGORITHM AS PROPORTION TO SIMPLE TRAVERSAL

String Length (n)	Time for Naive Algorithm as a Percentage of Control (%)	Time for Optimized Algorithm as a Percentage of Control (%)
10	185.39	43.42
10^2	115.81	93.68
10^3	118.67	96.55
10^4	88.57	87.41
10^5	169.24	91.99
10^6	125.15	99.42

As seen above, the empirical result is an interesting insight into the problem itself. The algorithm that has been optimized is able to run consistently faster than simple traversal (which should have a time complexity of $O(n)$, faster than the worst-case time complexity of the optimized algorithm of $O(nm)$). In addition, the naive algorithm is not that much slower than the simple traversal in the test that was done.

Both of these "anomalies" suggests that the cases that slow down both of those algorithms do not happen as much or do not happen in such severity to make them multiple times slower than simple traversal as the worst-case time complexity analysis suggests. An average-case analysis is thus needed to properly assess these algorithms and determine how fast both of them are exactly but that would be beyond the scope of this paper. For now, it can be concluded that the worst-case time

complexity is quite far from the average-case complexity as shown by the empirical analysis above.

V. Conclusion

Brute-force algorithms have a bad reputation for being inefficient. Although they still are, it is still a valuable tool to analyze the particulars and quirks of the problem at hand by observing the algorithm in action. The particulars and quirks could then be exploited into making a better, more efficient algorithm as have been shown in this paper. Although the optimized algorithm still falls under the algorithm design strategy of brute-force, it does not mean that optimization created from observation can only be incorporated into a brute-force algorithm. Observation(s) could be a basis for an entire redesign of the approach by switching the strategies to a more sophisticated one, like backtracking, divide-and-conquer, etc..

The empirical analysis of this problem also re-emphasize the importance of average-case time complexity analysis in the analysis of algorithms as it is clear that the worst-case for this problem does not reflect how a typical string looks like.

Github Repository

https://github.com/Fatih20/LongestSubstring

Acknowledgment

The author thanks LeetCode for their third problem, “Longest Substring With Unique Characters” for sparking inspiration for the naive algorithm and its optimization that are presented in this paper.

References

[1] R. Munir, *Brute-Force Algorithm (Bagian 1)*. Bandung, West Java, 2022.
[2] R. Munir, *Brute-Force Algorithm (Bagian 2)*. Bandung, West Java, 2022.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Mei 2022

Fatih Nararya Rashadyfa Ilhamsyah, 13521060