CSE 494/598: Algorithms in Computational Biology

Fall 2022

Instructor: Heewook Lee **Date:** 09/12/2022

String Algorithms: Exact Pattern Matching
Note: This note may contain errors.

4 Exact Pattern Matching

In the following few lectures, we will explore algorithms for performing exact pattern matching

4.1 Motivation

Exact pattern matching is the most basic string operation and useful.

1. Application is endless. (Finding a word within a document)
2. Similarly, its application in biology is also very strong. Finding restriction enzyme sites (proteins that cut the DNA at specific location in the genome) within a genome. Restriction enzyme sites are short k-mers. Finding all of these sites is an instance of exact matching.
3. We have already seen an application of exact string matching in ori-finding problem.

4.2 Problem Formulation

ExactPatternMatching: Given a pattern (query) P, does it exist in the text (reference) T or find all occurrences of the pattern in the text?

Example:

```
Text position  1    i          n = 17
Text           ATTCACTATTCGGCTAT
Pattern        GCAT
Pattern Pos    1..m = 4
```

4.3 naïve Approach

We already came up with a naïve approach where the algorithm scans the text from left to right and perform string comparisons.

For each pattern, the algorithm performs up to m comparisons (pattern length) while scanning the text at $(n - m + 1)$ positions, resulting in m(n-m+1) operations, giving us O(nm).

Q: Does the naïve algorithm need to perform m(n-m+1) operations?
A: It depends on whether we stop as soon as we find a match or we continue until we have found all the matches of the pattern within the text. In the former case, we maybe able to perform less number of steps.

Q: Assume we just need to find the first match, does it ever need to perform m(n-m+1) operations?
A: Yes, the worst case, where the pattern is located at the tail of the string.

Q: Why is the algorithm not so applicable?
A: Imagine we are trying to match a sequence (say approximately 1 kb sequence) in human genome (3 billion bp). The total run-time would be on the order of 3×10^{12} operations. On a 3GHz processor (and assuming each comparison takes exactly one CPU cycle), the calculation would take 1,000 seconds (16 mins). Does this sound not too bad?? Modern sequencing experiments generate tens to hundreds of millions of sequences. This is probably too slow.

Key Question: How can we improve on this algorithm? The key idea is to use information we've already figured out when making earlier comparisons. We will first look at Z algorithm.

4.4 Z Algorithm

Let's switch gear and first talk about a simple function Z that captures the internal structure of the text. For every position, i in $(0, n]$ in text T, we define Z array $Z[i]$ to be the length of the longest prefix of $T[i...n]$ that matches exactly a prefix of $T[0...n]$. An example is shown here:

```
Text:  A T T C A C T A T T C G G C T A T
Z[i]:  0 0 0 0 1 0 0 4 0 0 0 0 0 0 0 2 0
```

Q: Assume we have an efficient way to generate Z-array for a given text, can Z values in aligning a pattern to a text? How would we use the Z-array to find occurrences of a pattern in text?
Possible Answer: Create a string S by concatenating P and T, $S = PT$. Find all $Z[i]$ values in S and any $Z[i] >= |P|$ identifies an occurrence of P in T.
Better approach: We can construct the following string $S = P\$T$, where P and T are the pattern and the text, respectively, and \$ is a character that does not match any other character in either pattern or text.

Q: How do Z values computed for the string S defined above help us find matches between P and T? It suffices to look at the Z values corresponding to characters in T. Any Z-value equal to the length of P indicates the pattern matches.
A: Find all $Z[i]$ values in S, and any $Z[i] = |P|$ identifies an occurrence of P in T.

Q: Can we use the reverse text $S' = TP$ or $T\$P$. NO! It's not symmetric.

Q: What is the runtime and memory usage of this algorithm?
A: First the memory: Need to store the $Z[i]$'s in addition to the pattern and text, for a total of $2(n+m)+1$ memory locations. More memory but it's still linear in the size of the inputs. Once $Z[i]$'s are computed, we simply need to examine each position in the Z array to find all matches between P and T, or to decide that a match doesn't exist, resulting in $O(n)$ if $Z[i]$'s can be computed efficiently.

4.4.1 Can Z-values also be computed in linear time and space?

Let's first discuss how a naïve algorithm computes.
For each position from 2 to $|S|$, compare and write down $Z[i]$, resulting in quadratic time.
Q: can we somehow reuse the work done earlier?
A: The intuition is that when computing $Z[i]$ we can use the $Z[j]$ values for $j < i$.
Here is the algorithm. The algorithm computes $Z[i], i = 2$ to $i = |S|$

First, let's define a few things we need to further explain. Suppose we have $Z[i] > 0$, meaning we have a prefix of $S[i..|S|]$ that matches a prefix of S and this is a the longest prefix at position i. Any prefix found at any position i is a Z-box. At position i, among all Z-boxes, let r be the rightmost position among all Z-boxes found so far and l be the leftmost position in Z-box whose right-end is r (Fig 1). Since the each prefix represented by a box in the figure is the longest prefix at each position, $x \neq y$.

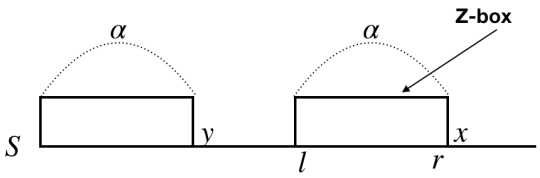


Figure 1: Z-box: r is the rightmost position in any Z-box found so far. l is the leftmost position in Z-box whose right end is r. x and y are the characters after the last position of the Z-box bounded by l and r and the matching prefix of S respectively.

Now, assume that we have already computed $Z[2] \ldots Z[k-1]$, and we want to compute $Z[k]$. We will break down into 2 main cases based on relative location of k and r.

Case 1: $(k > r)$. In this case, there isn't any information for us to utilize to deduce $Z[k]$. Therefore, we need to compute Z[k] explicitly

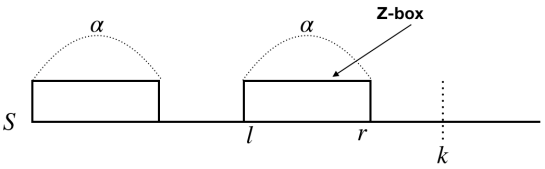


Figure 2: Case 1 $(k > r)$

Case 2: $(k <= r)$

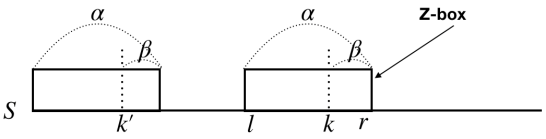
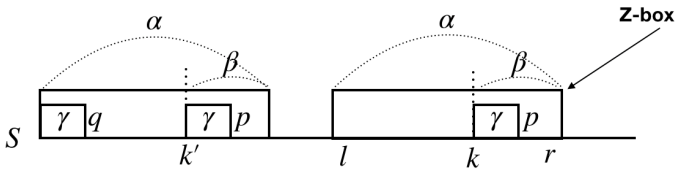


Figure 3: Case 2 $(k \leq r)$

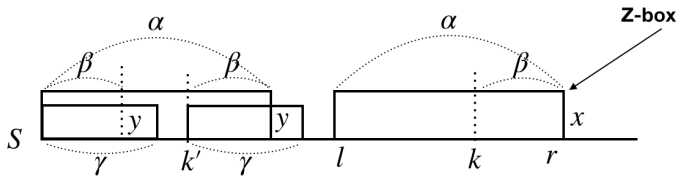
We will break this case into 3 sub-cases.

Case 2a: ($Z[k'] < |\beta|$), meaning $|\gamma| < |\beta|$



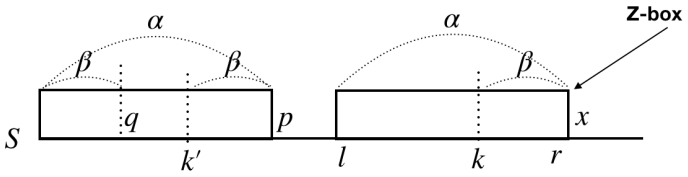
Since $p \neq q$, we can conclude that $Z[k] = Z[k']$ without any character comparisons.

Case 2b: ($Z[k'] > |\beta|$)



Since $x \neq y$ (if they were then the z-box shown in the figure should be longer), we can conclude that $Z[k] = |b|$ without any character comparisons.

Case 2c: ($Z[k'] = |\beta|$)



We know $x \neq p$ but we can't conclude whether x and q are same or not. Therefore, we need to perform character comparisons starting with x and q when computing for $Z[k]$. Note that we do not need to compare any characters that are within β.

4.4.2 Time Analysis: Z algorithm does at most $O(|S|)$ character comparisons.

Character comparison can result either in match/mismatch. First let's focus on r. It is easy to see that r is never decreasing as the algorithm runs.

- case 1: $k > r$, r potentially increases.
- case 2a/2b: r never gets extended as $Z[k] <= |b|$
- case 2c: r potentially increases if any character comparison results in a match.

It is clear that r gets extended whenever there is a match, therefore r is bounded by $|S|$.

What about mismatches? In any iteration where character comparisons are carried out, when does the iteration end? It ends as soon as a mismatch occurs. Therefore, the total number of mismatches is also bounded by $|S|$.

Then, we can conclude that the total number of character comparisons is done at $O(|S|)$.

4.5 KMP - Knuth, Morris, Pratt Algorithm

Let's look at another pattern matching algorithm that runs in linear time and space. To better understand the key idea used in KMP algorithm, let's look at the naive algorithm once again and describe how it runs.

The matching process compares the pattern to the text until either match or mismatch is found. In case of mismatch, the algorithm shifts the pattern position by one and restarts all the comparisons. Can we reuse some of the matching information we learned while matching the pattern to the text?

Let's look at an example where a pattern is made up of different characters at each position.

```
T Pos   1                n
Text    XYABCXABCXADCDAFEA
Pattern ABCD
```

On third iteration ($i = 3$), we have the following alignment (mismatch is shown in red, also marked as 'x' denoting the adjacent position of the last matched position)

```
T Pos    1 i  x           n
Text     XYABCXABCXADCDAFEA
Pattern    ABCD
```

Q: Can we shift the pattern by more than 1 for next iteration?
A: Notice that the pattern is made up of different characters. We can conclude that we can shift the pattern to position $i = 6$ skipping positions 4 and 5.

Let's consider another example where we have a different pattern but same text. At iteration $i = 3$ we have the following alignment:

```
T Pos    1 i      x       n
Text     XYABCXABCXADCDAFEA
Pattern    ABCXABCDE
```

Q: How far can we shift?
A: Notice that the suffix of $P[1..(x-1)]$ is also the prefix of P. We can simply shift the pattern over until its prefix matches a previously matched section of the text (suffix of $P[1..(x-1)]$).

The intuition is that once we run into a mismatch or run out of characters to match (a complete match), we can shift as much as the length of the **longest** suffix of P[1..(x-1)] matches exactly a prefix of P.

Here is another example where the pattern results is a complete match:

```
T Pos    1       x       n
Text     ACABACACDBACACD
Pattern ACABACA
```

You can see that P matches (ran out of character to match, x denotes the position after last match) exactly at position $i = 1$. We need to shift to find more occurrences of P in T if any.
Q: How far can we shift?
A: You can see that 'ACA' is the longest suffix of P[1..(x-1)] (which is P in this case) that matches exactly a prefix of P. Therefore, we can shift the pattern to position 5 and start comparing from $x = 8$.

In order to use the intuition obtained from our examples, we first define *lps* (longest prefix suffix) array for pattern P. Let $lps[i]$ be the length of the longest non-trivial suffix of P[1..i] that matches exactly a prefix of the pattern (non-trivial meaning the suffix and the prefix are same as P[1..i] itself).
Q: How do we compute *lps* array efficiently?

Let us assume that we are trying to compute the value $lps[i+1]$ and that we already know the values $lps[j]$ for all $j <= i$. Also, let c be the character at position $i+1$ in the pattern ($c = P[i+1]$), and x be the character at position $lps[i]+1 (x = P[sp[i]+1])$ (i.e. the character that follows the prefix of P that matched the suffix ending at i).

If the two characters match (x == c), we simply extend the matching prefix/suffix pair by one character to obtain $lps[i+1] = lps[i] + 1$.
Q: What happens, however, if $x \neq c$?
A: In this case, $lps[i]$ provides us no additional information, but we might be able to use $lps[lps[i]]$, the suffix of $P[1..lps[i]]$ that matches a prefix of P. Let x' be $P[lps[lps[i]]]$, the character that follows the character at position $lps[lps[i]]$. Here we will compare c with x' and if they are match, lps[i+1]= lps[lps[i]] + 1. If

they don't, we simply continue trying further lps[lps[..[lps[i]]..]] values until we either find a match, or we "bottom out" indicating no such prefix exists.

In the "bottome out" case, we set $lps[i+1] = 1$ if $c = P[1]$, or $lps[i+1] = 0$, otherwise.