CS 194-1 Algorithms for Computational Biology **Spring 2012**

Lecture 2: September 2

Lecturer: Nir Yosef *Scribe: Rohin Shah*

Warning: This note has not been proofread by the instructor.

2.1 Exact string matching

In biological applications, we often wish to compare the sequences of nucleic acids or proteins. For example, we might be interested in finding domains (functionally important parts of proteins) that are shared between different protein families, or we may want to map a newly sequenced read to find out corresponding positions in a reference genome, and if this read only maps to a single region of said genome.

A first step towards solving this problem is to interpret biological sequences as strings. To encode DNA sequences, for example, the four nucleotides adenine, cytosine, guanine and thymine are commonly abbreviated with the letters A, C, G and T, respectively, and a similar one-letter code exists for the amino acids that describe protein sequences.

Formally, given a pattern P and a text T, which are both strings over a finite alphabet, we wish to find all occurrences of P in T. The length of P is denoted $|P| = m$ and the length of T is $|T| = n$. We further assume that the basic operations at our disposal are character comparisons; that is, for two characters c_1 and c_2, we assume we can test the following relations in constant time: $c_1 = c_2$, $c_1 \neq c_2$ and $c_1 < c_2$. In the last case, we assume some lexicographic ordering exists between the characters.

2.2 Naïve method

In the simplest algorithm, we loop over both P and T, and compare characters until we either find a mismatch or we have found all characters of P in a contiguous segment of T. In the latter case, we know we have found an occurrence of P in T. Thus, the naïve algorithm has the form:

```
for i : 1 → n − m do
  for j : 1 → m do
    found ← true
    if P[j] ≠ T[i + j] then
      found ← false
      break
    end if
  end for
  if found is true then
    print i
  end if
end for
```

The problem with this algorithm is that it is not very efficient. It is quite easy to see that due to the double for-loop, the algorithm runs in $O(nm)$.

In order to optimize this algorithm, we want to avoid repeating character comparisons that have already been done. There are two main ideas that we use in order to accomplish this:

1. Trick #1: Skipping some values of i

   For example, when we search for `abxaby` in `xabxabxya`, when $i = 2$ we do a direct comparison between `abxaby` and `abxabx`, which results in a mismatch only on character 7. A smart algorithm would notice during this comparison that the next `a` is at position 5, and so it would skip $i = 3$ and $i = 4$ and go directly to $i = 5$.

2. Trick #2: Skipping some values of j

   In the same example, when we do a direct comparison of `abxaby` and `abxabx`, a smart algorithm would notice during this comparison that the `abx` beginning at position 5 also matches the `abx` at the beginning of the pattern, and so when it starts checking $i = 5$, it would already know that the first 3 characters match, and so it would start the direct character comparison at $j = 8$ (skipping $j = 5, 6, 7$).

2.3 The Z-Algorithm

The first algorithm with runtime $O(n+m)$ was proposed by Knuth, Morris and Pratt in 1977, and a similar, slightly more efficient algorithm was developed by Boyer and Moore, also in 1977. The algorithm presented here is a simpler one that was discovered by Dan Gusfield and can be used to explain these two algorithms.

In what follows, we use S to denote a string over some finite alphabet Σ.

2.3.1 The Z-Function

Definition 2.1 (The Z-Function)**.** *For $i = 2, \ldots, |S|$, let $Z_i(S)$ denote the length of the longest substring of S that starts at position i and matches a prefix of S. (A prefix of S is a substring of S that starts at the beginning of S).*

As an example, consider the string $S =$ `AAACAAAG`. A way to proceed would be to write S aligned with itself, but shifted by $i - 1$. To compute $Z_3(S)$ we would then write:

```
AAACAAAG
__AAACAAAG
```

We start then at the bottom string, and immediately see that only the first `A` overlaps, and therefore $Z_3(S)$ must be 1.

We then ask if it is possible to find $Z_i(S)$ for all $i = 2, \ldots, |S|$ in $O(|S|)$ time. It turns out that a linear-time algorithm for computing all Z-scores exists. But first, let us see why this is useful.

2.3.2 Reducing Exact Matching to Computing Z-scores

Suppose we have an algorithm for computing the Z-scores of a string. Consider a pattern P and a text T. Intuitively, the Z-scores allow us to compare all the substrings of a string against prefixes of the string. Since we want to compare the text T against the pattern P, we should compute the Z-scores of $S = PT$ (the concatenation of P and T). However, here the Z-scores could compare against both P and T. In order to solve this, we compute $Z_i(S)$ for $S = P\$T$, where \$ is a symbol in neither P nor T. The inclusion of \$ forces all of the substrings to match only up to P (since

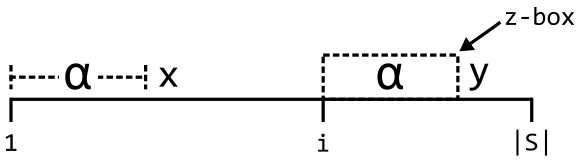


Figure 2.1: Illustration of a Z-box of S at position i, with α a prefix of S and $x, y \in \Sigma$ such that $x \neq y$. The initial occurrence of the prefix α is illustrated without a bounding box to emphasize that it is not a Z-box.

they can never match the \$). So, any Z-scores of size $|P|$ must be matches to the pattern P in the text T.

More formally, by the inclusion of \$, we see that $Z_i(S) \leq |P|$ for all i, since any $Z_i(S) > |P|$ would imply the presence of a \$ in T. Furthermore, $Z_i(S) = |P|$ for $i > |P| + 1$ if and only if P occurs in T at position $i - |P| - 1$.

2.3.3 The Linear Time Algorithm

A few more definitions are useful to describe the linear time algorithm for computing all Z-scores.

Definition 2.2. *For all positions i where $Z_i(S) > 0$, a Z-box is defined as the interval starting at i and ending at position $i + Z_i(S) - 1$ (inclusive). See Figure 2.1 for illustration.*

Thus, each substring of S that is also a prefix of S defines a Z-box.

Definition 2.3. *A rightmost Z-box is one for which the right endpoint is maximal among all Z-boxes. Remark: there may be more than one rightmost Z-box.*

Definition 2.4. *The Z-Algorithm computes $Z_i(S)$ for all $2 \leq i \leq |S|$ in $O(|S|)$ time.*

The Z-Algorithm computes $Z_i(S)$ in order from 2 to $|S|$. When computing $Z_i(S)$, it uses previously computed Z-values in order to avoid repeated work. It also keeps track of a rightmost Z-box using two indices l and r. The Z-box consists of the interval of S from l to r (inclusive). If there is no Z-box so far, then $l = r = 0$.

The Z-Algorithm proceeds as follows:

1. Compute $Z_2(S)$ by direct character comparison. If $Z_2(S) > 0$, let $l = 2$, $r = 2 + Z_2(S) - 1$, else let $l = r = 0$.
2. When computing $Z_k(S)$, we know $Z_2(S), Z_3(S), \ldots, Z_{k-1}(S)$ have been computed. l and r are the left and right endpoints, respectively, of some rightmost Z-box among $Z_2(S), Z_3(S), \ldots, Z_{k-1}(S)$. Note that because any Z-box must have started at most at position $k - 1$, it must hold that $k > l$. However, we do not know if k exceeds r, and so we consider two cases:
   (a) If $k > r$, computer $Z_k(S)$ by direct character comparisons. Then, if $Z_k(S) > 0$, update l and r by setting $l = k$ and $r = k + Z_k(S) - 1$.
   (b) $l < k \leq r$. Let β denote the substring starting at position k and ending at r, and let $j = k - l + 1$, as seen in Figure 2.2.
   Given this configuration, we compare $Z_j(S)$ (which is known because $j < k$) to $|\beta|$, the result of which may be any of the following:
      i. $Z_j(S) < |\beta|$: In this case, there exist $a, b \in \Sigma$, where $a \neq b$, and a string γ (which could be empty), such that the situation depicted in Figure 2.3 holds. So, $Z_k(S) = |\gamma| = Z_j(S)$.

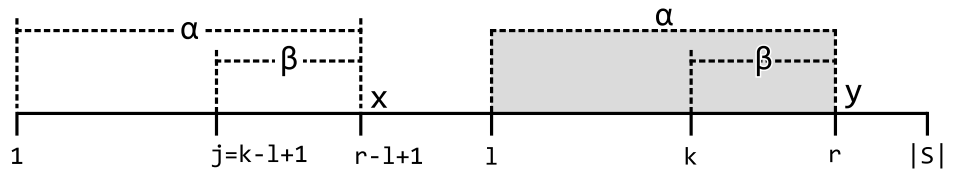


Figure 2.2: k falls inside the Z-box bounded by l and r, and β is the substring that starts at position k and ends at r. Because β is inside a Z-box, it must also be present inside α at the beginning of S, and j denotes where β starts inside of α. Note also that the character just outside the Z-box does *not* match the corresponding character (otherwise the Z-box would also include that character).

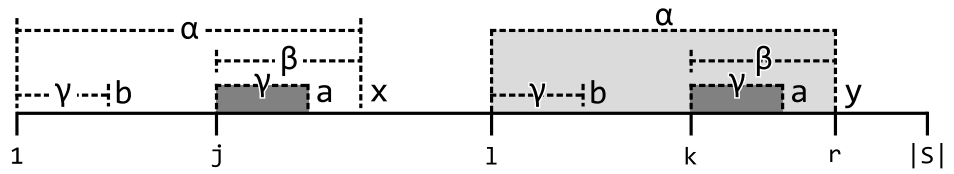


Figure 2.3: The case of $Z_j(S) < |\beta|$. Note that $Z_k(S) = |\gamma| = Z_j(S)$

ii. $Z_j(S) > |\beta|$: As illustrated in 2.4, the character at position $|\beta| + 1$ of S must be x, which we know is not equal to y. Thus, $Z_k(S) = |\beta|$. We can set $l = k$, but it's not necessary for correctness of the algorithm.

iii. $Z_j(S) = |\beta|$: As illustrated in Figure 2.5, let v denote the character at position $|\beta| + 1$ of S. Then, $Z_j(S) = |\beta|$ implies $v \neq x$, while v may or may not be equal to y. Thus, $Z_k(S)$ must be computed using direct character comparisons, but the computation can start at position $r + 1$. Set $l = k$ and $r = k + Z_k - 1$; note that this is necessary only if $Z_k > |\beta|$.

Notice that all the cases where $k \leq r$ are using trick #1 and trick #2 described previously to save some character comparisons.

2.3.4 Extended Example

Consider $P =$ `axyaxz`, $T =$ `xaxyaxyaxz`.

Then, we need to compute Z-scores of $S = P\$T =$ `axyaxz$xaxyaxyaxz`.

Direct comparison gives $Z_2 = 0$.

1. $k = 3$: We have $l = 0$, $r = 0$
   Case 1: $k > r$. Direct comparison gives $Z_k = 0$.

2. $k = 4$: We have $l = 0$, $r = 0$
   Case 1: $k > r$. Direct comparison gives $Z_k = 2$ with a Z-box of `ax`.

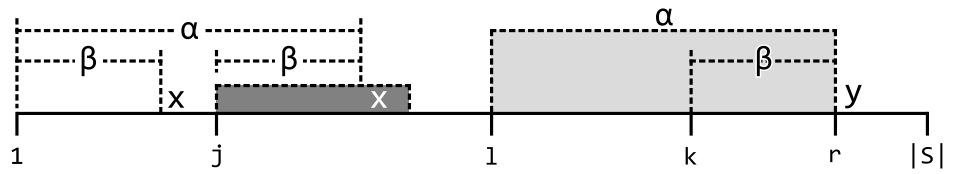


Figure 2.4: The case of $Z_j(S) > |\beta|$. Since $x \neq y$, $Z_k(S) = |\beta|$.

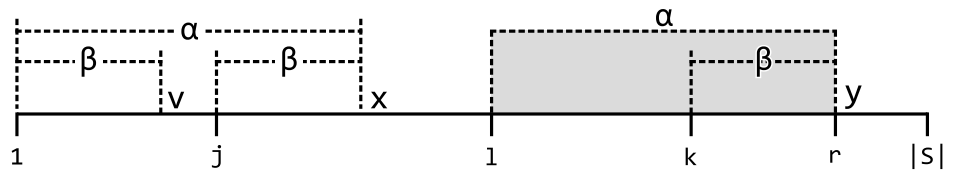


Figure 2.5: The case of $Z_j(S) = |\beta|$. $Z_k(S)$ can be computed by direct character comparisons starting at position $r + 1$.

3. $k = 5$: We have $l = 4$, $r = 5$
   Case 2: $k \leq r$, $j = k - l + 1 = 2$, $\beta =$ `x`, $Z_j = 0$
   Case 2a: $Z_j < |\beta|$, so $Z_k = Z_j = 0$

```
|---A---|
|       |   |---A---|
|   |-B-|   |       |
|   |   |   |   |-B-|
  a   x   y   a   x   z   $   x   a   x   y   a   x   y   a   x   z
  1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17
      j       l   k,r
```

4. $k = 6$: We have $l = 4$, $r = 5$
   Case 1: $k > r$. Direct comparison gives $Z_k = 0$.

5. $k = 7$: We have $l = 4$, $r = 5$
   Case 1: $k > r$. Direct comparison gives $Z_k = 0$.

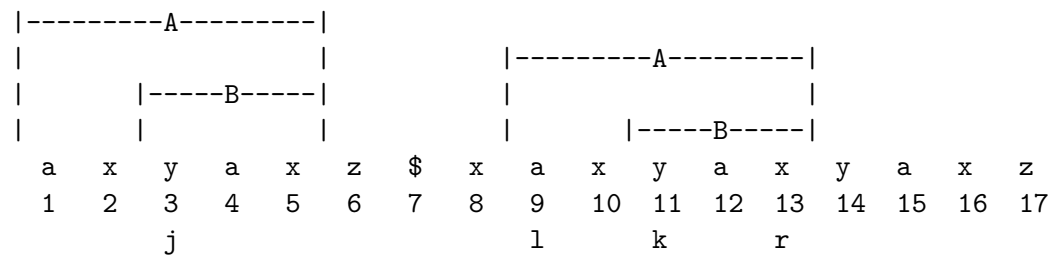
6. $k = 8$: We have $l = 4$, $r = 5$
   Case 1: $k > r$. Direct comparison gives $Z_k = 0$.

7. $k = 9$: We have $l = 4$, $r = 5$
   Case 1: $k > r$. Direct comparison gives $Z_k = 5$ with a Z-box of `axyax`.

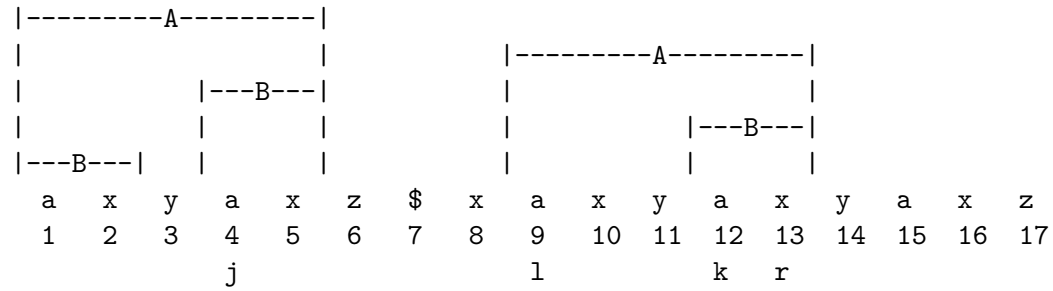
8. $k = 10$: We have $l = 9$, $r = 13$
   Case 2: $k \leq r$, $j = k - l + 1 = 2$, $\beta =$ `xyax`, $Z_j = 0$
   Case 2a: $Z_j < |\beta|$, so $Z_k = Z_j = 0$

```
|---------A---------|
|                   |           |---------A---------|
|   |-------B-------|           |                   |
|   |               |           |   |-------B-------|
  a   x   y   a   x   z   $   x   a   x   y   a   x   y   a   x   z
  1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17
      j                           l   k           r
```

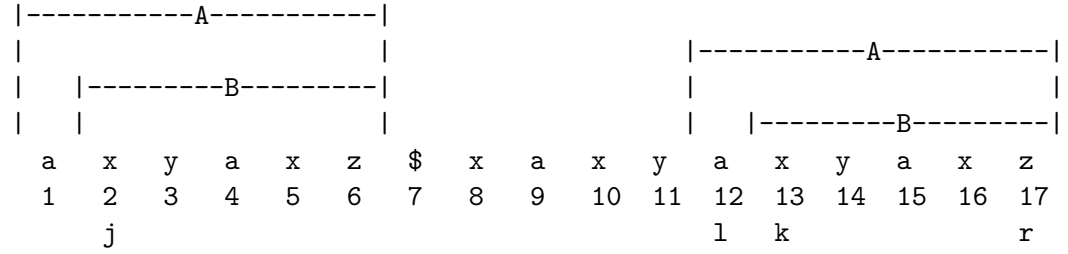
9. $k = 11$: We have $l = 9$, $r = 13$
   Case 2: $k \leq r$, $j = k - l + 1 = 3$, $\beta = \texttt{yax}$, $Z_j = 0$
   Case 2a: $Z_j < |\beta|$, so $Z_k = Z_j = 0$



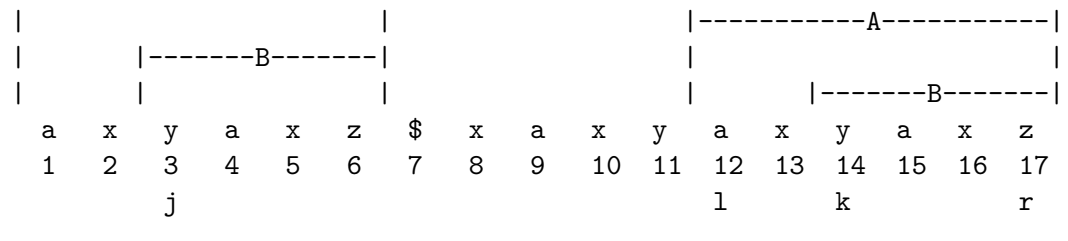
10. $k = 12$: We have $l = 9$, $r = 13$
    Case 2: $k \leq r$, $j = k - l + 1 = 4$, $\beta = \texttt{ax}$, $Z_j = 2$
    Case 2c: $Z_j = |\beta|$, so Z_k is at least $|\beta|$. Using direct comparison, $Z_k = 6$.



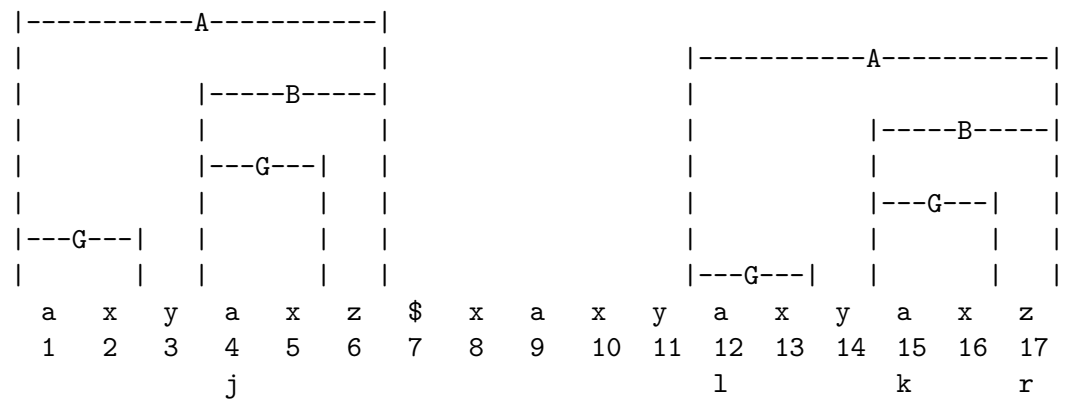
11. $k = 13$: We have $l = 12$, $r = 17$
    Case 2: $k \leq r$, $j = k - l + 1 = 2$, $\beta = \texttt{xyaxz}$, $Z_j = 0$
    Case 2a: $Z_j < |\beta|$, so $Z_k = Z_j = 0$



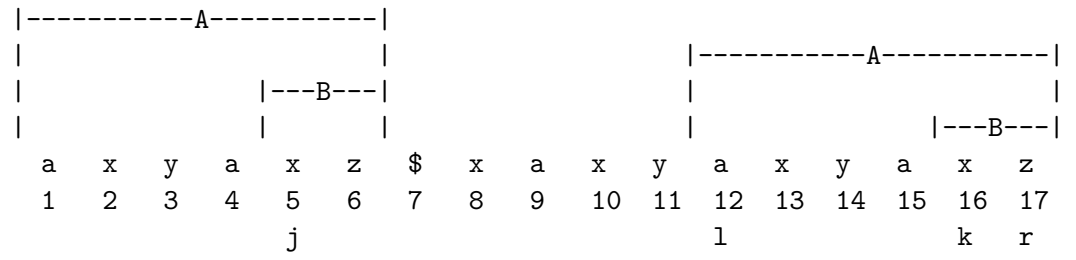
12. $k = 14$: We have $l = 12$, $r = 17$
    Case 2: $k \leq r$, $j = k - l + 1 = 3$, $\beta = \texttt{yaxz}$, $Z_j = 0$
    Case 2a: $Z_j < |\beta|$, so $Z_k = Z_j = 0$



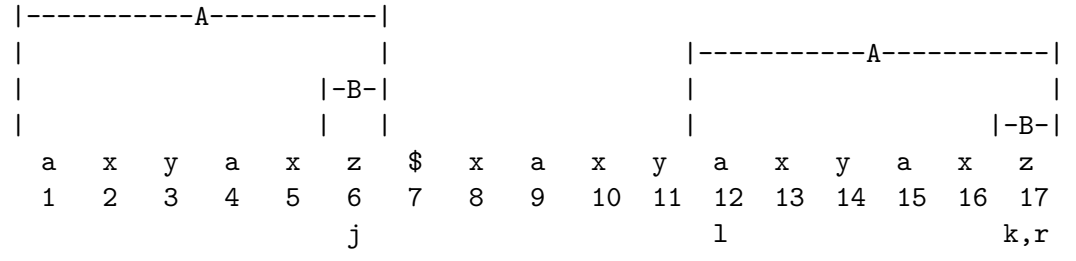
13. $k = 15$: We have $l = 12$, $r = 17$
    Case 2: $k \leq r$, $j = k - l + 1 = 4$, $\beta =$ `axz`, $Z_j = 2$
    Case 2a: $Z_j < |\beta|$, so $Z_k = Z_j = 2$



14. $k = 16$: We have $l = 12$, $r = 17$
    Case 2: $k \leq r$, $j = k - l + 1 = 5$, $\beta =$ `xz`, $Z_j = 0$
    Case 2a: $Z_j < |\beta|$, so $Z_k = Z_j = 0$



15. $k = 17$: We have $l = 12$, $r = 17$
    Case 2: $k \leq r$, $j = k - l + 1 = 6$, $\beta =$ `z`, $Z_j = 0$
    Case 2a: $Z_j < |\beta|$, so $Z_k = Z_j = 0$



2.3.5 Time complexity

To analyze the time complexity, note that # character comparisons = # mismatches+# matches.

- As each mismatch terminates an iteration, and there are $|S| - 1$ iterations (one for each $2 \leq k \leq |S|$), we immediately see that # mismatches $\leq |S|$.
- To count the number of matches, consider that
  1. all character comparisons occur to the right of r
  2. throughout the algorithm $r \leq |S|$
  3. each match increases r by at least $|\sigma|$ (see Figure 2.6).

Therefore, # matches $\leq |S|$.

As both the number of matches and the number of mismatches are bounded by $|S|$, there are $O(|S|)$ character comparisons during the second step of the algorithm. Additionally, the first step requires computing $Z_2(S)$ by direct character comparision, which requires $O(|S|)$ character comparisons in the worst case. Thus, the algorithm indeed has time complexity $O(|S|)$.

2.3.6 Space complexity

Because we analyze a string of form $S = P\$T$, all prefixes are shorter than $|P| = m$. Therefore, we only need to store $Z_2, \ldots, Z_m$ and thus space complexity of the algorithm is in $O(m)$.

In the general case of any string S, there is no limit on how large the matched prefixes can be, and so we would need $O(|S|)$ space.

2.3.7 Limitations

There are several limitations to this algorithm. First, if we have a set of Patterns $P_1, P_2 \ldots, P_d$, the complexity of the algorithm is $O(dn + \sum_{i=1}^{d} |P_i|)$, and thus increases rather fast with d. Second, this algorithm is not applicable to problems more complex than exact matching, such as finding strings with a given number of mismatches.

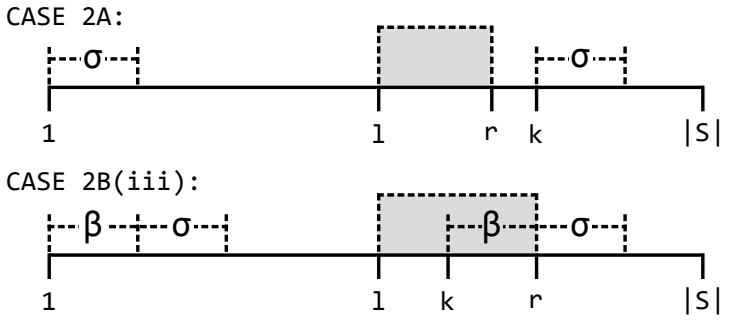


Figure 2.6: Whenever character comparisons are made, only σ is compared, and the starting point for the next set of character comparisions will always start past the end of σ.