Kevin Gao

Notes on String Algorithms

With Applications in Computational Biology

TERRA-INCOGNITA.DEV

Contents

*These notes are primarily based on the book **Algorithms on Strings, Trees, and Sequences** by **Dan Gusfield** [1].*

Thanks Professor Gusfield for his great book.

*The part on indexing and compression is based on the book **Genome-Scale Algorithm Design** [2] and Ben Langmead's lectures.*

Part I

Comparison-Based Exact Matching

1

Naive Exact Matching

(Gusfield Section 1.1)

Terminology Confusion. Before starting the discussion of string matching algorithms, we should note the difference between a ***substring*** and a ***subsequence***. Given a string S, characters in a substring of S must occur contiguously in S; whereas characters in a subsequence may be interspersed gaps (or indels, as we call them in biology) and/or characters not in the original string.

Exact String Mathching Problem

Given a text string T and pattern P, the goal of the exact string matching problem is to find all occurrences of P in T.

The "Naïve" Algorithm

```
NAIVE-MATCH(P, T)
 1  matches = [ ]
 2  for i = 1 to |T| − |P| + 1
 3       match = TRUE
 4       for j = 1 to |P|
 5            if T[i + j] ≠ P[j]
 6                 match = FALSE
 7                 break
 8            if match
 9                 matches.APPEND(i)
10       return matches
```

Figure 1.1: Naive exact matching with $P = abxyabxa$ and $T = xabxyabxyabxz$.

T : $xabxyabxyabxz$
P : $abxyabxa$
$abxyabxa$
$abxyabxa$
$abxyabxa$
$abxyabxa$
$abxyabxa$

The naïve exact matching algorithm aligns the left end of P with the left end of T and compares the characters of P and T left to right until a mismatch is found, or until it reaches the end of P, in which case we report the position of P. Then, P is shifted to the right by one place. We repeat this procedure until the right end of P passes the right end of T.

Runtime of the Naive Algorithm

Let $n = |P|$ and $m = |T|$. The worst-case comparisons made by the naive algorithm is $\Theta(nm)$. We have the lower bound $\Omega(nm)$ when P and T contains the same repeated characters (e.g. $P = aaa$, $T = aaaaaaaaa$), in which case the algorithm makes $n(m - n + 1) \in \Omega(nm)$ comparisons.

2

Z Algorithm

(Gusfield Chapter 1)

Speeding Up the Naive Algorithm

The naive exact matching algorithm is stupid. It always shifts P by only one even if it knows for sure that the next shift will not yield a match. This gives us some ideas on how to improve the algorithm. If we can shift P by more than one character, but never shift so far as to miss the next occurrence of P in T, we can improve the runtime of the naive algorithm.

Doing this, however, requires us to have some prior knowledge of the pattern P or the text T.

Foundamental Preprocessing

A fundemental preprocessing is a generalized way to process the pattern P to gain knowledge of the pattern, independent of any particular algorithm.

Definition 2.1. *Given a string S and a position $i > 1$, let $Z_i(S)$ be the length of the longest substring of S that starts at i and matches a prefix of S.*

In other words, $Z_i(S)$ is the length of the longest prefix of $S[i \dots |S|]$ that matches a prefix of S.

Definition 2.2 (Z-box)**.** *For any position $i > 1$ where Z_i is greater than 0, the **Z-box** at i is the interval starting at i and ending at $i + Z_i - 1$.*

Definition 2.3. *For every $i > 1$, r_i is the right endpoint of the Z-box that begins at or befor position i (i.e. the closest Z-box to the left). More formally, r_i is the largest value of $j + Z_j - 1$ over all $1 < j \leq i$ such that $Z_j > 0$.*

Figure 2.1: Relations between i, l_i, r_i and the Z-box at l_i.

The linear-time computation of the Z_i values for all $i \in \{1, \ldots, |S|\}$ from the string S is called the ***fundamental preprocessing*** task.

The Z Algorithm

The ***Z algorithm*** is an algorithm for computing the fundamental preprocessing. The Z algorithm is similar to a dynamic programming algorithm in the sense that it uses memoized information to speed up the computation and reduce the number of comparison needed. Namely, assume there exists $j < i$ such that $j + Z[j] - 1 > i$, then we can use $Z[i - j + 1]$ to infer $Z[i]$.

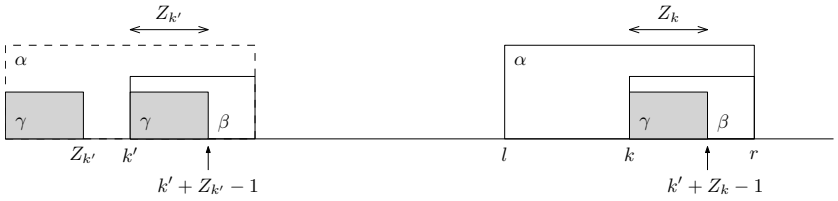


Figure 2.2: Case 2a: The longest string starting at k' that matches a prefix of S is shorter than $|\beta| = r - k + 1$. In this case, $Z_k = Z_{k'}$.

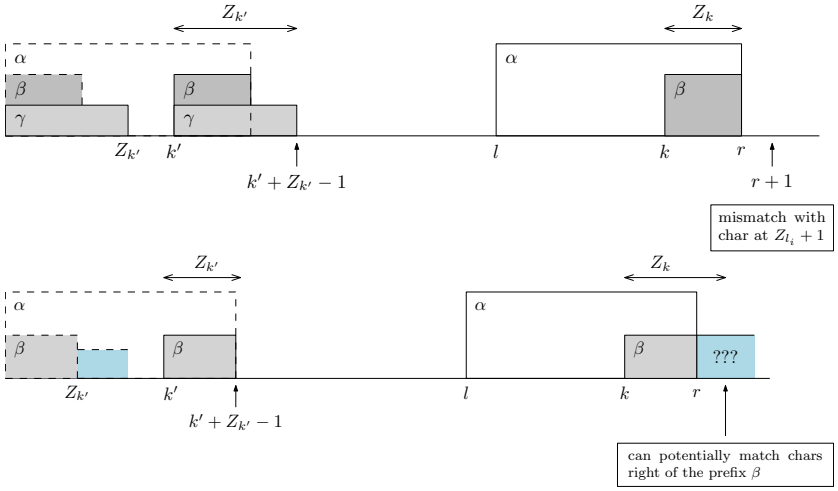


Figure 2.3: Case 2b: The longest string starting at k' that matches a prefix of S is at least $|\beta| = r - k + 1$. In this case, we continue to compare characters right of r with characters right of the $Z_{k'}$th character until mismatch.

Given Z_i for all $1 < i \leq k - 1$ and the current l and r, we can compute Z_k using the following procedure:

1. Case 1: If $k > r$, k is not in an existing Z-box. Compute Z_k by explicitly comparing each character starting at k with prefix of S.
2. Case 2: If $k \leq r$, k is in an existing Z-box, denoted α, that matches a prefix of S. Hence, character $S[k]$ also appears in the prefix at position $k' = k - l + 1$.

We also consider the substring $S[k \dots r]$, denoted β. By the same reasoning, β matches the substring $S[k' \dots Z_l]$.

(a) $Z_{k'} < |\beta|$. This implies that the longest string starting at k' that matches a prefix of S, γ, is shorter $|\beta|$. Then, $Z_k = Z_{k'}$, and l, r remain unchanged. Note that $|\gamma|$ can be 0.

(b) $Z_{k'} \geq |\beta|$. This means $S[k \dots r]$ is at least a prefix of S. However, Z_k might be larger than $|\beta|$. We cannot determine this solely based on the existing information as characters beyond the Z_lth character are not included in α. So, we compare the characters starting at position $r+1$ of S to the characters starting at position $|\beta|+1$ of S, until a mismatch occurs. Say the mismatch occur at position q. Then, $Z_k = q - k$, $r = q - 1$, and $l = k$. (If $Z_{k'} > |\beta|$, a mismatch occurs immediate after the Z_lth character because otherwise, the $(r+1)$th character matches the Z_lth character, and we would have a longer α and a larger Z_l)

COMPUTE-Z(S)

```
1  n = |S|
2  l, r, k = 1
3  Z = empty array of length n
4  for k = 2 to n
5      if k > r                                          // Case 1
6          // compute Z[k] from scratch
7          l, r = k
8          while r ≤ n and S[r − l + 1] == S[r]
9              r = r + 1
10         Z[k] = r − l
11         r = r − 1
12     else                                              // Case 2
13         k' = k − l + 1
14         if Z[k'] < r − k + 1                          // Case 2a
15             Z[k] = Z[k']
16         else                                          // Case 2b
17             q = r + 1
18             while q ≤ n and S[q − l + 1] == S[q]
19                 q = q + 1
20             Z[k] = q − k
21             l, r = k, q − 1
22 return Z
```

For a step-by-step animation of the Z algorithm, see
https://personal.utdallas.edu/~besp/demo/John2010/z-algorithm.htm

Running Time of the Z Algorithm

Theorem 2.4. *All the Z_i values can be computed by the Z algorithm with at most $2|S| \in O(|S|)$ comparisons.*

Proof. The time is proportional to the number of iterations, $|S|$, plus the number of character comparisons.

Each iteration that does any character comparison ends as soon as there is a mismatch, so there are at most $|S|$ **mismatches**. In every iteration where there is a match, r moves to the right by an amount at least as large as the number of matches. This implies that there are at most $|S|$ **matches**. Note that once a character results in a match, it is not compared again.

Every character comparison is either a match or mismatch, so the total number of comparisons is at most $2|S|$. □

Correctness of the Z Algorithm

Theorem 2.5. *At the kth iteration of the algorithm,* Compute-Z *correctly calculates Z_k, and l, r are updated correctly.*

Proof. By a case analysis.

Case 1: Trivial since it's just explicit comparison.

Case 2a: We claim that the substring at position k can match a prefix of S only for length $Z_{k'} < |\beta|$. Prove the claim by contradiction (suppose not, we would have a longer prefix matching the substring at k', contradicting the maximality of $Z_{k'}$).

Case 2b: β must be a prefix of S as established above. Characters beyond r are explicitly compared. □

Corollary 2.6. Compute-Z *correctly calculates Z_k for all $k \in \{2, \ldots, |S|\}$.*

Proof. By induction on k. □

Z Algorithm for Exact Matching

The Z algorithm lends itself to a linear-time algorithm for exact matching. Given a pattern P and text T, we construct a new string $P\$T$ where \$ is a ***separator*** (a.k.a. ***delimiter*** or ***sentinel***) such that $\$ \notin \Sigma$. We construct the Z-array for this new string. It takes $O(|P| + |T|)$

time to construct the Z-array. After this, we can simply make a pass through the Z-array and read the result from it. This takes $O(|T|)$ time.

Z-EXACT-MATCHING(P, T)

```
1  query = P + "$" + T
2  Z = COMPUTE-Z(query)
3  for i = |P| + 1 to |P| + |T| + 1
4       if Z[i] == |P|
5            pattern found at index i − |P|
```

3
Boyer-Moore

(Gusfield Chapter 2)

KEY IDEAS of the Boyer-Moore algorithm: right-to-left scan, bad character rule, good suffix rule.

Bad Character Rule

The intuition behind the bad character rule is as follows. Suppose we have the pattern P and text T, and the rightmost character of P is aligned to the character x in T, where $x \neq y$. If we know the position of the rightmost x in P, we can safely shift P by that amount to the right so that the x in P aligns with the x in T. Furthermore, if we know that x is not in P, we can shift P completely past the x in T. This intuition is formalized below, as shown in Gusfield's book.

Definition 3.1. *For each character x in the alphabet, let $R(x)$ be the position of the rightmost occurrence of character x in P. If x does not occur in P, $R(x) = 0$.*

Rule 3.2 (Bad Character Rule)**.** Suppose for a particular alignment of P against T, the rightmost $n - i$ characters of P match their counterparts in T, but the next character to the left $P(i)$ mismatches with its counterpart in T, say at position k in T. The ***bad character rule*** says that P should be shifted right by $\max\{1, i - R(T(k))\}$ places. That is, if the rightmost occurrence of $T(k)$ in P is in position $j < i$, then shift P so that the jth character of P is matched to the kth character of T (or completely shifting P past kth position if the kth character of T does not occurr in P). Otherwise, if $j > i$, shift P by one position.

Extended Bad Character Rule

Note that the bad character rule allows us to shift more than one character only when the mismatched character show up at a position left of the mismatched point. However, it is not as helpful if the character only occurs in P on the right side of the mismatched point. The ***extended bad character rule*** addresses this issue.

Rule 3.3 (Extended Bad Character Rule)**.** When a mismatch occurs at position i of P and the mismatched character in T is x, then shift P to the right so that the closest x to the left position i in P is matched with the x in T.

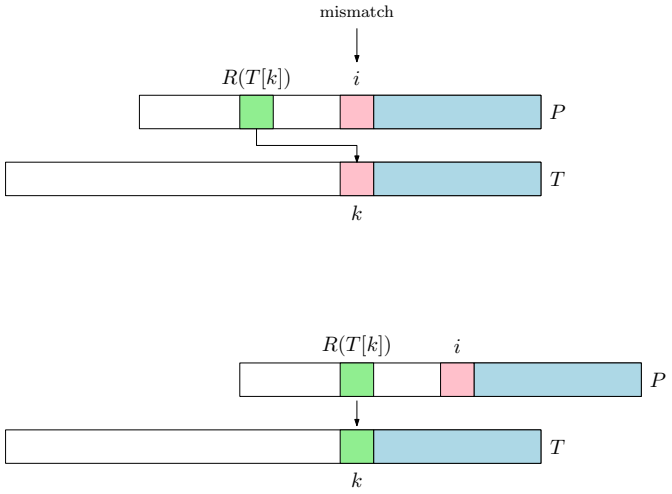


Figure 3.1: The bad character rule says to shift as much as possible so that a mismatch becomes a match.

Preprocessing for Bad Character Rule

The bad character rule is quite straightforward to implement. In the preprocessing for the bad character rule, we find, for each position i in the pattern P and for each characther $x \in \Sigma$, the position of the closest occurrence of x in P to the left of i. This uses an n matrix. However, the time complexity for building this lookup table and the space required for it could be massive, depending on the length of the pattern and the size of the alphabet. It is also not hard to see that this approach can be quite space inefficient and we can end up store duplicate information over and over.

Alternatively, we can scan P from right to left, and for each character $x \in \Sigma$, we keep a list of positions where x occurs in P (think hash table with chaining but just using another array instead of linked list). Since we scan P from right to left, the indices will be stored in

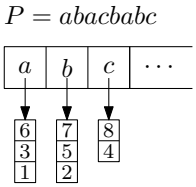


Figure 3.2: Table for storing information obtained from the bad character rule preprocessing.

decreasing order for each character. It takes $O(n)$ space. During the actual execution of the Boyer-Moore algorithm, we can query this table whenever we need to find the rightmost occurrence of character x left of index i. We can use *binary search* for this.

If the alphabet Σ is sparse, we can condense it by simply creating a mapping f from $\Sigma \to \mathbb{N}$. The two approaches can be implemented as follows, respectively.

```
CREATE-BAD-CHAR-TABLE(P, f)
1  table = [ ]
2  row = [0, ..., 0]
3  for i = 1 to |P|
4      c = P[i]
5      table.APPEND(row)
6      row[f(c)] = i + 1
7  return table
```

```
CREATE-BAD-CHAR-TABLE(P, f)
1  table = [ ]
2  for i = |P| downto 1
3      c = P[i]
4      table[f(c)].APPEND(i)
5  return table
```

Good Suffix Rule

Rule 3.4 (Good Suffix Rule)**.** Given an alignment of P against T, suppose a substring t of T matches a **suffix** of P, but a mismatch occurs at the next comparison to the left. Then, find, if exsits, the right-most copy t' of t in P such that t' is not a suffix of P. Shift P to the right so that t' in P is matched with t in T.

Essentially, the ***good suffix rule*** says we can shift as much as possible such that an *existing match does not become a mismatch*.

This rule is the weaker version of the good suffix rule used by Boyer and Moore's original publication. We now present a stronger version of the good suffix rule that allows us to prove properties of the Boyer-Moore algorithm more easily.

Strong Good Suffix Rule

Rule 3.5 (Strong Good Suffix Rule)**.** Given an alignment of P against T, suppose a substring t of T matches a **suffix** of P, but a mismatch occurs at the next comparison to the left. Then, find, if exsits, the right-most copy t' of t in P such that t' is not a suffix of P. Additionally, **we requires that the character left of t' in P differs from the character to the left of t in P.** Shift P to the right so that t' in P is matched with t in T.

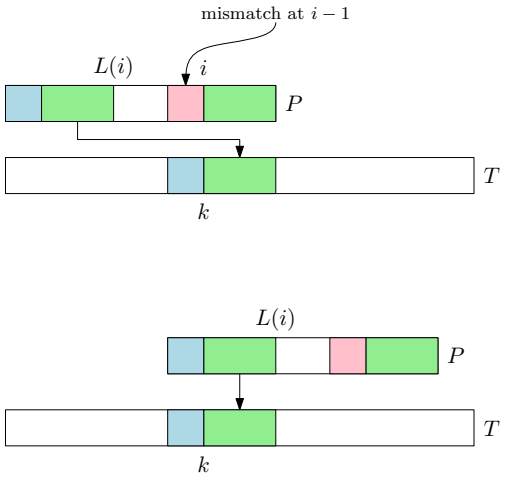


Figure 3.3: The good suffix rule says we can shift as much as possible so long as a match does not become a mismatch. The additional requirement in the strong good suffix rule ensures that we don't encounter the same mismatch at $P[i-1]$.

Note the additional requirement stated in the second-to-last sentence. The addition of this requirement ensures that we don't get the same mismatch. A copy of t is not worth looking at if it results in the same mismatch at the next character to the left. We call this rule the ***strong good suffix rule***. It should be obvious that it is safe to shift using the strong good suffix rule. That is, we won't miss any matches if we shift P using the strong good suffix rule. We formalize this idea of correctness in the following theorem.

Theorem 3.6. *The strong good suffix rule does not shift P past an occurrence in T.*

Proof. By contradiction. See Gusfield Theorem 2.2.1. □

Preprocessing for Good Suffix Rule

Definition 3.7. *For each i, let $L(i)$ be the **largest position** less than n such that string $P[i \ldots n]$ matches a suffix of $P[1 \ldots L(i)]$. If no such position exists, $L(i) = 0$.*

For each i, let $L'(i)$ be the largest position less than n such that string $P[i \ldots n]$ matches a suffix of $P[1 \ldots L'(i)]$ and such that the character preceding that suffix is not equal to $P[i-1]$. If no such position exists, $L'(i) = 0$.

By definition, $L'(i)$ gives the right endpoint of the rightmost copy of $P[i \ldots n]$ that is not a suffix of P, and whose preceding character does not equal to $P[i-1]$.

Definition 3.8. *For string P, $N_j(P)$ is the length of the **longest suffix** of the substring $P[1 \ldots j]$ that is also a **suffix** of the whole string P.*

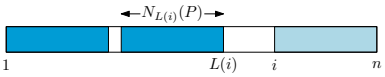


Figure 3.4: $L(i)$ gives the right-most copy of t that is not a suffix of the whole string.

Theorem 3.9. *$L(i)$ is the largest index j less than n such that $N_j(P) \geq |P[i \ldots n]|$.*

Proof. By definition of L, $L(i)$ is the largest index less than n such that $P[i \ldots n]$ matches a suffix of $P[1 \ldots L(i)]$.

Let $j = L(i)$ in the definition of N_j. Then, $N_j(P)$ is the length of the longest suffix of the substring $P[1 \ldots L(i)]$ that is also a suffix of the whole string. A suffix of the whole string must also be a suffix of $P[i \ldots n]$. Then, clearly, $N_j(P) \geq |P[i \ldots n]|$ because the substring $P[L(i) - N_j(P) + 1 \ldots L(i)]$ can possibly match more characters left of $P[i]$. □

Corollary 3.10. *$L'(i)$ is the largest index j less than n such that $N_i(P) = |P[i \ldots n]| = n - i + 1$.*

Proof. Immediate from Theorem 3.9. The definition requires that the characters left of $P[i]$ must not be contained in the suffix ending at $L'(i)$. □

One crucial observation is that N_j is, in fact, the exact reverse of Z_i when we talked about the Z-algorithm. Hence, $N_j(P)$ can be computed in linear time for all j in linear time by calling COMPUTE-Z on P^R (the reversal of P).

COMPUTE-L'(P)
1 $N =$ COMPUTE-Z(P^R)
2 **for** $i = 1$ **to** $|P|$
3 $\quad L'[i] = 0$
4 **for** $j = 1$ **to** $|P| - 1$
5 $\quad i = n - N[j] + 1$
6 $\quad L'[i] = j$
7 **return** L'

Definition 3.11. *For string P, $l'(i)$ is the length of the **longest suffix** of $P[i \ldots n]$ that is also a **prefix** of the whole string P. If none exists, $l'(i) = 0$.*

Figure 3.5: $l'(i)$ gives the index of the end of the longest prefix that matches a suffix of $P[i \ldots n]$.

Theorem 3.12. *$l'(i)$ is the largest index $j \leq |P[i \ldots n]| = n - i + 1$ such that $N_j(P) = j$.*

Proof. By definition of l', the prefix $P[1 \ldots l'(i)]$ is the longest prefix that matches a suffix of $P[i \ldots n]$. So $l'(i) \leq |P[i \ldots n]|$. A suffix of $P[i \ldots n]$ is also a suffix of P, so $P[1 \ldots l'(i)]$ also matches a suffix of P.

Let $j = l'(i)$ in the definition of N_j. Then, by definition of N_j, $P[l'(i) - N_{l'(i)}(P) + 1 \ldots l'(i)]$ is the longest suffix of $P[1 \ldots l'(i)]$ that matches a

suffix of P. The longest suffix of $P[1 \dots l'(i)]$ is $P[1 \dots l'(i)]$ itself, and indeed $P[1 \dots l'(i)]$ matches a suffix of P. Hence, $l'(i) = N_{l'(i)}(P)$. The maximality follows from the definition of $l'(i)$. □

COMPUTE-$\ell'(P)$

```
1  N = COMPUTE-Z(P^R)
2  // initialize l' array
3  for i = 1 to |P|
4      l'[i] = 0
5  // set entires of l' based on Theorem 3.12
6  for j = 1 to |P|
7      if N[j] == j
8          l'[|P| - j + 1] = j
9  // fill in l' for i's left of the longest suffix that matches a prefix
10 for i = |P| - 1 downto 1
11     if l'[i] == 0
12         l'[i] = l'[i + 1]
13 return l'
```

It may not be clear at first what the last for-loop on line 10-12 does.

It copies the values of l' from the right to fill in the blanks. Consider the example shown in Figure 3.6. $P[1 \dots l'(i')]$ is the longest prefix that matches a suffix of P. For any $i < i'$, $P[i \dots n]$ will not match a prefix of P. However, for those values of $i < i'$, $P[i' \dots n]$ is **still a proper suffix** of $P[i \dots n]$, so they "inherit" the l' values from $l'(i')$ – the left most position such that the entirety of $P[i' \dots n]$ matches a prefix of P.

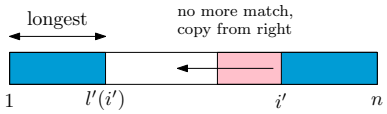


Figure 3.6: "Smear" to the left. This handles the cases when $P[i \dots n]$ itself does not match a prefix of P, but some **proper suffix** $P[i' \dots n]$ does.

Putting Everything Together

The Shifting Rules

We summarize the shifting rules here. Remember that we are scanning from right to left.

1. Bad character rule: Mismatch at i with $T[k] \neq P[i]$; Shift P to the right by $i - R(T[k])$.
2. Good suffix rule: Mismatch at i with $T[k] \neq P[i]$ but the suffix $P[i+1 \dots n]$ matches some substring of T
   (a) If $L'(i) > 0$, then shift P to the right by $n - L'(i)$
   (b) If $L'(i) = 0$, then shift P to the right by $n - l'(i)$

3. No mismatch and P is entirely matched with some substring of T, apply good suffix rule and shift P to the right by $n - l'(2)$.

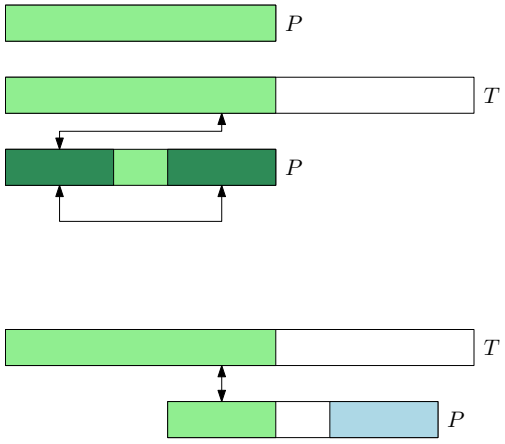


Figure 3.7: The special good suffix rule when P matches with T. We shift P to the right so that a prefix of P remains matched with T, if exists.

Implementation

For simplicity sake, we use I, the identity function when creating the bad character table. Again, in practice, if we are dealing with a sparse alphabet, we may want to change it to something else.

```
BOYER-MOORE(P, T)
 1  L' = COMPUTE-L'(P)
 2  l' = COMPUTE-ℓ'(P)
 3  R = CREATE-BAD-CHARACTER-TABLE(P, I)
 4  k = |P|
 5  while k ≤ |T|
 6      i = n, h = k
 7      while i > 0 and P[i] == P[h]
 8          i = i − 1
 9          h = h − 1
10      if i = 0
11          P found at position h in T
12          k = k + |P| − l'[2]
13      else
14          bc = i − R[T[h]]
15          gs = |P| − L'[i]
16          if gs == 0
17              gs = |P| − l'[i]
18          k = k + max{bc, gs}
```

4
Knuth-Morris-Pratt Algorithm

Part II

Suffix Trees

5
Suffix Trees

Suffix Tries

Let us recall the definition of a suffix.

Definition 5.1 (Suffix)**.** *For any string S, $S[i \dots j]$ is the **substring** starting at position i and ending at position j; $S[1 \dots i]$ is the prefix of S ending at i; and $S[j \dots |S|]$ is the **suffix** of S starting at position j. A proper substring, prefix, or suffix is a substring, prefix, or suffix that is neither the entire string S nor the empty string.*

Then, we define a trie and a suffix trie as follows.

Definition 5.2 (Suffix Trie)**.** *A **trie** is the smallest tree such that each edge is labeled with a character from the alphabet Σ, each node has at most one outgoing edge labeled with c for each $c \in \Sigma$, and each node has a key that is the concatenation of the edge labels along the path from the root to that node. A **suffix trie** is a trie where each root-to-leaf path represents a suffix.*

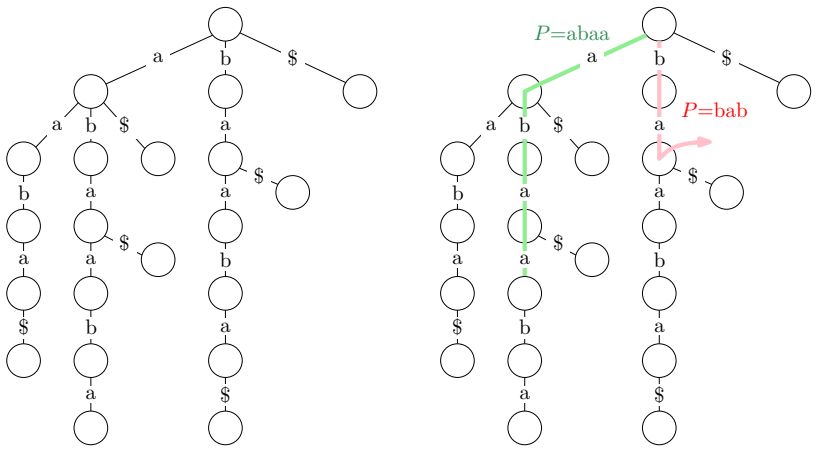


Figure 5.1: Suffix trie for $T = abaaba$. On the right: the search path for $P = abaa$ and $P = bab$. When searching for a pattern that is not in T, we "fall off" the trie.

In a regular tree (e.g. binary search tree), the key is stored at each node. In a trie, the keys are *implicitly* represented by the edge labels along the path. Figure 5.1 shows a suffix trie constructed for $T = abaaba$.

It is important to add the terminator character \$ at the end of the string. If we remove the terminator \$, it is not hard to see the result trie may no longer be a valid suffix trie. We assume that \$ is *lexicographically smaller than all characters* in Σ.

Search in Suffix Trie

SEARCH FOR PATTERN: It is easy to search for a pattern P given a suffix trie. We can **start from the root and follow the edges labeled with the characters** in P until we either finish reading the pattern and find a match, or "fall off" the trie, in which case we can return that a match is not found.

SEARCH-TRIE(P, T)

```
1  cur = T.root
2  for c in P
3      if c ∉ cur.edges
4          return FALSE
5      else cur = cur.edges[c]
6  return cur ≠ NULL
```

Assume that at each node, we maintain a hash table for each outgoing edges. Then, the algorithms runs in expected time $\Theta(|P|)$.

SEARCH FOR SUFFIX: Similarly, if we want to see if a given pattern P is a suffix of T, we can run the same algorithm and checks if the node at the end of the path has an outgoing edge labeled \$.

SEARCH FOR NUMBER OF OCCURRENCES: If we are interested in the number a pattern P occurs as a substring in T, we can run SEARCH-TRIE. Once we arrive at the end of our search path, we run a **depth-first search** from the node at the end of the search path and count the number of leaf nodes reachable from that node. Since a trie is a tree, DFS runs in $O(|V|)$ time. In this case, it takes $O(|P| + |T|)$ time to find the number of occurrences of a given pattern.

SEARCH FOR LONGEST REPEATED SUBSTRING: Find the deepest (internal) node with more than one children.

Space Complexity of Suffix Trie

Construct a Suffix Trie

Suffix Tree

From Suffix Trie to Suffix Tree

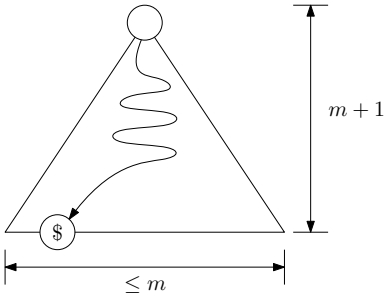


Figure 5.2: Max width and height of a suffix trie. The path from the root to the deepest leaf represents the longest suffix (the whole string plus the terminator).

6

Constructing Suffix Trees

Naive Construction

The simplest way to construct a suffix tree is to first construct a suffix trie and convert it to a suffix tree by repeatedly coalescing the paths. This takes $O(n^2)$ time and space. It takes $O(n^2)$ space because we need to store the intermediate suffix trie.

Ukkonen's Linear-Time Construction

Types of Extensions

Suppose that we have $S[j \dots i] = \beta$ be a suffix of $S[1 \dots i]$. In some iteration j, the algorithm finds the end of β and extends the path by adding $S[i+1]$ to the path. This ensures that the suffix $S[j \dots i+1]$ is included in the new tree. Observe that there are three types of insertions:

Type 1. In the current tree, path β ***leads to a leaf***. In this case, $S[i+1]$ is ***added to the end of the label*** on that edge.

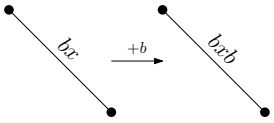


Figure 6.1: Type 1 insertion for suffix bx of $S = axabx$.

Type 2. There is ***no path*** from the end of β that starts with character $S[i+1]$, but at least one labeled path continues from the end of β. In this case, a ***new leaf edge*** starting from the end of β is created and labeled $S[i+1]$, which leads to a ***new leaf node*** with number j.

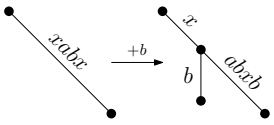


Figure 6.2: Type 2 insertion for suffix x of $S = axabx$.

Type 3. Some ***path*** from the end of β starts with character $S[i+1]$. In this case, $\beta \cdot S[i+1]$ is ***already in the implicit suffix tree***. So we ***do nothing***.

Suffix Links

Definition 6.1 (Suffix Link)**.** Let $x\alpha$ be an arbitrary string, where $x \in \Sigma$ is a *single character* and $\alpha \in \Sigma^*$ is a (possibly empty) *substring*. For an internal node v with root-to-node path label $x\alpha$, if there is another node $s(v)$ with root-to-node path label α, then we create a pointer from v to $s(v)$, called a ***suffix link***.

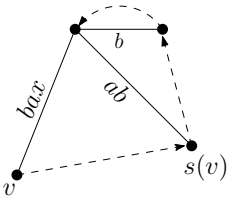


Figure 6.3: A suffix link from v (representing xab) to $s(v)$ (representing ab). The other two suffix links are also shown. Note that if a node's path label has no proper suffix, we create a link to the root.

The reason we have suffix link is because we want to *access the insertion point (end of a suffix) efficiently*. Suppose we insert a new character to the sequence $x\alpha$. Once we inserted the new character to the suffix $x\alpha$, we also need to insert the character to the end of α. Without suffix link, we would have to traverse back to the root and search for the insertion point all over again. The use of suffix link is especially useful for jumping from one suffix to the next during extension.

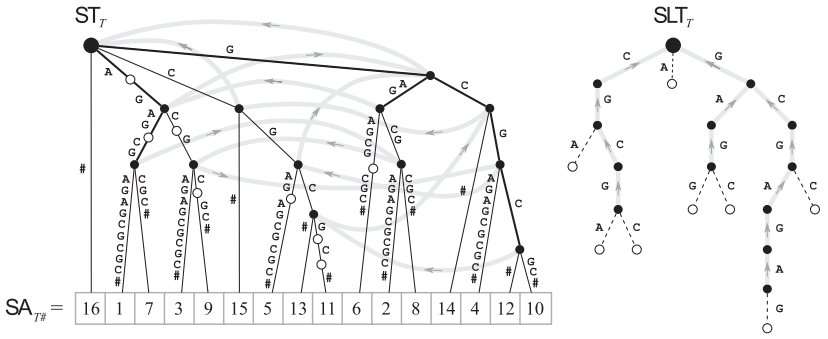


Figure 6.4: A suffix tree and its corresponding suffix link tree for $T =$ AGAGCGAGAGCGCGC.

Moreover, the suffix links induce a subtree called the ***suffix link tree***. More formally, given a text T represented by a suffix tree (V, E) with suffix links, let $\ell(v)$ denote the label on the path from the root to v. Then, $L = \{(v, a) \mid v \in V, s(v) \in V, \ell(v) = x\ell(s(v)), x \in \Sigma\}$ be the set of all suffix links, and we define the tree $\text{SLT}(T) = (V, L)$ labeled by ℓ as the ***suffix link tree***.

Count and Skip

After reaching $s(v)$ via a suffix link, we still need to travel down the path labeled γ in order to add a new character at the end of γ, either by branching off or creating a new leaf node. However, this walk along the γ path takes $O(|\gamma|)$ if implemented directly. An alternative to this would be to ***store the number of characters of on each edge*** and skip nodes whenever we can.

Let g denote the length of γ, the next suffix to which we want to

append the new character. We start the search for the insertion point starting from $s(v)$.

Recall that no two edges coming out of $s(v)$ can have labels starting with the same character. We can then use one comparison to determine the edge that we need to follow. In particular, let $h = 1$ initially and before each iteration, compare the hth character of γ with the first character of every edge coming out of the current node. Let g' be the number of characters on the edge that we have just identified. Then, consider the following two cases:

1. $g \geq g'$: Skip to the node at the end of the edge. Set $g = g - g'$, $h = h + g'$, and repeat.
2. $g < g'$: Skip to the gth character on the edge and stop.

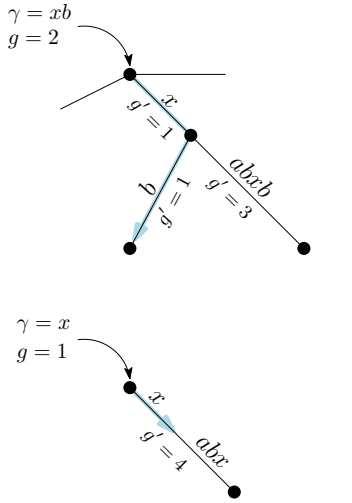


Figure 6.5:
Top: Case 1 where $g \geq g'$, skip to the next node;
Bottom: Case 2 where $g < g'$, go the the g-th character on the current edge.

Edge Label Compression

Another issue with our high-level "algorithm" for Ukkonen's algorithm is that every edge is explicitly labeled with the suffix they represent. A label of an edge can be as large as $\Theta(m)$, and there can be at most $\Theta(m)$ edges, making the total space required for a suffix tree $\Theta(m^2)$ in this case. This makes it impossible to build such a tree in $O(m)$ time. Fortunately, this issue can be solved using a simple trick: instead of storing the strings explicitly, we can store a pair of indicies representing the ***starting and ending position of the substring*** represented by each edge. That way, each edge can be maintained using only $\Theta(\log m)$ space.

In the ***word RAM model***[1], we assume that every word of size $\log m$ bits can be read and written efficiently in constant time. Hence, it is more plausible to build a suffix tree with compressed edge labels in linear time.

[1] In complexity theory, we have focused on Turing machine as our preferred model of computation. However, in analysis of algorithms, we usually use the word RAM model (often without explicitly stating it) as it is a more realistic model of how modern computers work.

Key Observations

No More Insertions After Type 3

In any given phase $i + 1$, if there is a Type 3 extension j (the new suffix $S[j \dots i + 1]$ is already in the tree), then any further extensions in the current phase will also be of Type 3. When there is a Type 3 extension, the path labeled $S[j \dots i]$ in the current tree must have already contained $S[i + 1]$. Then clearly, so does $S[j' \dots i]$ for all $j < j' \leq i + 1$ because they are all suffixes of $S[j \dots i]$.

Once a Leaf, Always a Leaf

At some point in the algorithm, if a leaf j is created for the suffix starting at position j, then the leaf will remain a leaf throughout the algorithm. This is because the algorithm never extends a leaf. If a path leads to a leaf, it will be a Type 1 insertion, in which case we extend the edge label without explicitly adding new nodes.

Part III

Dynamic Programming

7
Edit Distance

Part IV

Compression and Indexing

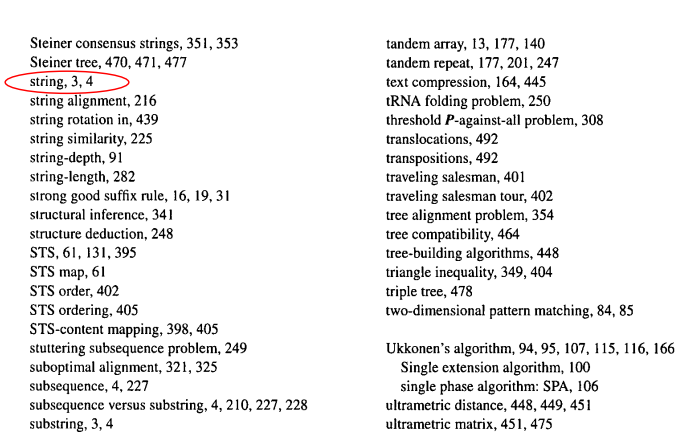
8

Entropy and Information

9
k-mer Index and Suffix Array

Idea Behind Indexing

Suppose you have a textbook and you want to look for a certain term, say, the word "*string*". It would be really time-consuming, and in some cases, impossible to look for the term by going through the book page by page. That's why most books usually (and hopefully) have an index at the end, which list the pages where each term occurs.



534 Index

Steiner consensus strings, 351, 353
Steiner tree, 470, 471, 477
string, 3, 4
string alignment, 216
string rotation in, 439
string similarity, 225
string-depth, 91
string-length, 282
strong good suffix rule, 16, 19, 31
structural inference, 341
structure deduction, 248
STS, 61, 131, 395
STS map, 61
STS order, 402
STS ordering, 405
STS-content mapping, 398, 405
stuttering subsequence problem, 249
suboptimal alignment, 321, 325
subsequence, 4, 227
subsequence versus substring, 4, 210, 227, 228
substring, 3, 4
tandem array, 13, 177, 140
tandem repeat, 177, 201, 247
text compression, 164, 445
tRNA folding problem, 250
threshold *P*-against-all problem, 308
translocations, 492
transpositions, 492
traveling salesman, 401
traveling salesman tour, 402
tree alignment problem, 354
tree compatibility, 464
tree-building algorithms, 448
triangle inequality, 349, 404
triple tree, 478
two-dimensional pattern matching, 84, 85

Ukkonen's algorithm, 94, 95, 107, 115, 116, 166
  Single extension algorithm, 100
  single phase algorithm: SPA, 106
ultrametric distance, 448, 449, 451
ultrametric matrix, 451, 475

Figure 9.1: The index of Gusfield's textbook, with the term "string" circled.

Index can help us speed up query speed and also give us a compact representation of the data.

Two big ideas behind indexing are ***grouping*** and ***ordering***. The first index that we will look at, *k*-mer index, uses grouping.

k-mer Index

Given a text T, we call all its substrings of length k, the ***k-mers*** of T. For example, consider the string `CGTGCGTGCTT`, it has the following 5-mers: `CGTGC`, `GCGTG`, `GTGCC`, `GTGCT`, `TGCCT`, and `TGCTT`. The substrings *can have overlaps*.

k-mer index groups the indices of T by the k-mer that starts at the index. So, for the string `CGTGCGTGCTT`, we have the k-mer index for $k = 5$:

k-mer	indices
`CGTGC`	0, 4
`GCGTG`	3
`GTGCC`	1
`GTGCT`	5
`TGCCT`	2
`TGCTT`	6

It is easy to construct a k-mer index. We can scan the string from left to right and record the position of each k-mer. It takes $|T| - k + 1$ time.

Querying k-mer Index

To query a k-mer index efficiently, we should first *sort the index lexicographically* by the k-mers. It takes $k(|T| - k)\log(|T| - k) \in O(|T|^2 \log |T|)$ steps. To compare two k-mers, we need k steps (unlike comparing two numbers or characters, which takes constant time), and sorting the list of $|T| - k$ k-mers takes $\Theta((|T| - k)\log(|T| - k))$. There is a bit of tradeoff here. If we choose a small k, we spend less time comparing two substrings during sorting, but we can possibly end with many k-mers in the list; in we choose a large k, it takes more time to compare two substrings, but we will have fewer k-mers to sort.

Once we sort the k-mer index, we can use *binary search* for patterns P such that $|P| \leq k$. It takes $O(|P| \log |T|)$ to query a sorted k-mer index. However, if $|P| > k$, k-mer index can be inefficient since we need to *manually extend the match* once the first k characters of P matches a substring in T.

10

Constructing Suffix Array

Naive Construction From Suffix Tree

Recall that a suffix array contains the index of all suffixes of a string, sorted in lexicographic order. Hence, given a suffix tree, a suffix array can be trivially constructed through a lexicographic depth first search (that is, at each internal node, decide which path to recurse on based on the alphabetical order of the first character of each path label).

As we have seen, a suffix tree can be constructed in $O(n)$ time. It follows that a suffix array can also be constructed in $O(n)$ time using this method. However, a major issue with this approach is that we must build an intermediate suffix tree, which may require significantly more memory, and this defeats the purpose of having a suffix array in the first place, which is to have a more compact representation of the suffixes of a string.

A Divide-and-Conquer Approach

The next natural approach one might consider when presented with a problem like constructing a suffix array is divide and conquer. We are all familiar with merge sort, which runs in $O(n \log n)$ time. Constructing a suffix array similarly involves sorting the suffixes.

Let T be the text for which we want to construct a suffix array. Consider the following divide-and-conquer approach:

1. Divide the suffix positions into $A \subset [0 \dots n]$ and $\overline{A} = [0 \dots n] \setminus A$
2. Construct a suffix array for T_A (suffixes that start at positions in A) recursively

3. Construct a suffix array for $T_{\overline{A}}$ based on the suffix array for T_A

4. Merge the two suffix arrays

The most straightforward way to divide is to divide the positions by parity (even/odd). This gives an algorithm whose runtime is given by the recurrence $T(n) = T(\lceil n/2 \rceil) + T_{merge}(n)$.

WHAT GOES WRONG? Everything seems good so far, but there are two important problems: (1) how do we construct the second suffix array non-recursively, and (2) how to merge in linear time?

For a long time, finding a way to merge two suffix arrays in linear time remained an open question. The most obvious way to merge takes $O(n^2)$ time. Researchers came up with clever tricks but still only got an $O(n \log n)$ time bound. Because of that, until the early 2000s, the best known algorithm for constructing a suffix array only ran in $O(n \log n)$ time.

In 2003, Karkkainen and Sanders published their seminal paper (along with some other researchers who independently published similar results around the same time), which proposed one of the first linear time algorithms for constructing a suffix array. It uses the same divide-and-conquer framework, with a little twist.

Karkkainen-Sanders Algorithm

Karkkainen and Sanders' algorithm uses the same divide-and-conquer approach, but instead of dividing the positions into even and odd positions like previous researchers have done, they divided the positions i's into those with $i \bmod 3 \neq 0$ and $i \bmod 3 = 0$. This, along with a neat trick during merging, is enough to give us an $O(n)$ time algorithm for construct a suffix array.

Let's first recall the general framework for constructing suffix array using divide-and-conquer

KARKKAINEN-SANDERS

1 construct suffix array for suffixes starting at positions $i \bmod 3 \neq 0$ recursively // $T(2/3n)$
2 construct suffix array for suffixes starting at positions $i \bmod 3 = 0$ using results from step 1 // $O(n)$
3 merge the two suffix arrays // $O(n)$

We will see how to perform each step within the given time. We call the suffixes starting at positions $i \bmod 3 \neq 0$ the ***sample suffixes***, and the suffixes starting at positions $i \bmod 3 = 0$ the ***non-sample suffixes***.

Sorting The Sample Suffixes, Recursively

Given a string T of length n, we define $T[j]$ for all $j > n$ to be equal to `$`, so $T[j] =$ `$` for all positions beyond n. This is just to avoid having to deal with the edge cases.

Let t_0 be the set of **triples** (not suffixes) starting at position $i \bmod 3 = 0$, so $t_0 = \{T[i \ldots i+2] \mid i \bmod 3 = 0, i \leq n\}$. Similarly, let t_1 and t_2 be the sets of triples starting at position $i \bmod 3 = 1$ and 2, respectively. For example, suppose $T =$ `dadbcddadbcd$` with the delimiter $ in the end, we will have

$$t_1 = \{\texttt{dad}, \texttt{bcd}, \texttt{dad}, \texttt{bcd}, \texttt{\$\$\$}\}$$

and

$$t_2 = \{\texttt{adb}, \texttt{cdd}, \texttt{adb}, \texttt{cd\$}\}$$

To sort the suffixes starting at positions $i \bmod 3 \neq 0$, we first sort the triples in $t_1 \cup t_2$. This can be done in $\Theta(n)$ time using **radix sort**. For $x \in t_1 \cup t_2$, we define the **rank** $rank(x)$ to be the order of the triple x in the sorted list of $t_1 \cup t_2$. If two triples have the same order in the sorted list, they will have the same rank. Further, for a set of triples X, we define $Rank(X)$ to be the list of ranks for each triple in the sorted order. That is, the ith element of $Rank(X)$ will be $rank(X[i])$. Using the same example as above where $T =$ `dadbcddadbcd$`, we have

pos	=	13	2	8	4	10	11	5	1	7
$Sorted(t_{1,2})$	=	`$$$`	`adb`	`adb`	`bcd`	`bcd`	`cd$`	`cdd`	`dad`	`dad`
$Rank(t_{1,2})$	=	1	2	2	3	3	4	5	6	6

We also record pos, the starting position of each of the triple in the original string.

Now, let us go back to the original set of triples, t_1 and t_2. We create a new string t' equals to $t_1 \cdot t_2$ (t_1 concatenated with t_2) with each triple **mapped to its rank**.

pos	=	1	4	7	10	13	2	5	8	11
$t_1 \cdot t_2$	=	`dad`	`bcd`	`dad`	`bcd`	`$$$`	`adb`	`cdd`	`adb`	`cd$`
t'	=	6	3	6	3	1	2	5	2	4

Next, we **recursively find the suffix array** for t'. We claim that the suffix array for t' specifies the suffix array for S restricted to the suffixes starting at positions $i \bmod 3 \neq 0$.

Wait! But Why?

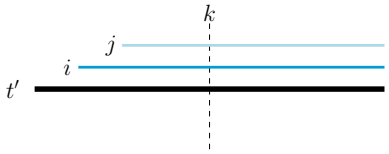
Claim. Suffix array for t' specifies the suffix array for S restricted to the suffixes starting at positions $i \bmod 3 \neq 0$

To see why this is true, we first prove this lemma.

Lemma 10.1. *Let t'_i and t'_j be two suffixes of t' starting at position i and j, respectively. If $s'_i \prec_{lex} s'_j$ (if s'_i is lexicographically less than s'_j), then the suffix of the original string T starting at position $pos[i]$ is also lexicographically smaller than the suffix of T starting at position $pos[j]$.*

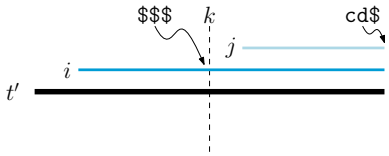
Proof. Recall that t' can be divided into two parts. The first half contains the ranks of triples whose first character starts at position $i \bmod 3 = 1$. The second half contains the ranks of triples whose first character starts at position $i \bmod 3 = 2$. To prove the lemma, we consider the following cases regarding the positions of i and j in t'.

Case 1: Both i and j are in the first half. Then, $pos[i] = pos[j] = 1 \bmod 3$. We first observe that the comparison of the two suffixes starting at i and j **will not go beyond the boundary** between the first and the second half.



More formally, let k be the position such that $pos[k] \bmod 3 = 1$ but $pos[k+1] \bmod 3 = 2$. Then, during the comparison of the two suffixes of t' starting at i and j, at most k characters are compared. This is because the triple at position $pos[k]$ will contain the unique **null terminator** that is lexicographically smaller than any character in the alphabet, thus giving the triple at $pos[k]$ a **unique rank**.

Moreover, each symbol in t' represents the **rank of a triple** starting at that position in T. By assumption, the suffix of t' starting at i is lexicographically smaller than the suffix of t' starting at j. Then, there must exists some c such that $t'[i+c] < t'[j+c]$. This implies that the triple starting at position $pos[j+c]$ that is lexicographically larger than the triple starting at position $pos[i+c]$ because t' represents the ranks of the triples. The presence of this **lexicographically larger triple** makes the suffix of the original string at $pos[i]$ lexicographically smaller than the suffix at $pos[j]$.



Case 2: Both i and j are in the second half. This case follows from a similar argument as Case 1.

Case 3: i is in the first half and j is in the second half. As in the first

two cases, the comparison will **never cross the boundary**. This, again, is due to the distinct null terminator symbol. In particular, the triple starting at $pos[k]$ (recall that k is the boundary between the two parts) contains at least one more/fewer null terminator symbol compared to the triple at $pos[|t'|]$. Hence, the triple at $pos[k]$ will have a **unique rank**, which helps us **break the tie** after the comparison at position k. We can always determine the lexicographic order of the two suffixes without crossing the boundary.

Now, going back to t', if we have the suffix of t' starting at i being lexicographically smaller than the suffix of t' starting at j, we know that there must be some c such that $t'[i+c] < t'[j+c]$, which implies the rank of some triple at $pos[i+c]$ is lexicographically smaller than that at $pos[j+c]$. Since we never cross the boundary when comparing the suffixes of t' starting at i and j, we are always comparing the rank of a contiguous and non-overlapping substring of T starting at $pos[i]$ with the rank of some other contiguous and non-overlapping substring of T at $pos[j]$ without ever going backward in the comparison (because we don't cross the boundary). Then, it follows that the lexicographic ordering of the ranks in t' implies the ordering in T.

Case 4: i is in the second half and j is in the first half. This follows from a similar argument as Case 3.

In all cases, the implication holds, so the lemma holds. □

One important takeaway from the proof of this lemma is that the null terminator symbol `$` is a tie-breaker, giving us unique ranks for triples at the end of the first and second half so that we never cross the boundary between the two halves. This unique rank, in turn, allows us to use the string of ranks to implicitly sort the suffixes in the original string T.

Using our previous example with

i		1	2	3	4	5	6	7	8	9
pos	=	1	4	7	10	13	2	5	8	11
$t_1 \cdot t_2$	=	`dad`	`bcd`	`dad`	`bcd`	`$$$`	`adb`	`cdd`	`adb`	`cd$`
t'	=	6	3	6	3	1	2	5	2	4

we have

SA for t'	=	5	8	6	4	2	9	7	3	1
SA_{12} for T	=	13	8	2	10	4	11	5	7	1

The ith entry in the SA for T is $SA_{12}(T)[i] = pos[SA(T')[i]]$. Here, $SA_{12}(T)$ refers to the suffix array for the original string T but only considering the suffixes at positions 1 or 2 mod 3.

Sorting the Non-Sample Suffixes

There is an easy way to sort the non-sample suffixes. Those are the suffixes that start at positions $i \bmod 3 = 0$. Again, we begin by considering the triples starting at these positions. Each of such positions is followed by two positions with $i \bmod 3 \neq 0$. Ordering of the triples starting at one and two positions after those with $i \bmod 3 = 0$ have already been determined recursively as discussed in the previous subsection. We can then use the information we know about the sample suffixes to sort the non-sample suffixes in linear time, non-recursively.

To this end, we construct a list t'' that contains all characters at positions $i \bmod 3 = 0$ with each character followed by the rank of the suffixes starting at position immediately after i (which can be determined from $SA[t']$ that we have constructed in the previous step).

Slightly more formally, the ith element of t'' will be

$$t''[i] = T[3i] \cdot SA_{12}(T)[3i+1]$$

In our example, $T =$ `dadbcddadbcd$` and

i		1	2	3	4	5	6	7	8	9
SA for t'	=	5	8	6	4	2	9	7	3	1
SA_{12} for T	=	13	8	2	10	4	11	5	7	1

so

triple	=	`dbc`	`dda`	`dbc`	`d$$`
t''	=	`d5`	`d8`	`d4`	`d1`
pos	=	3	6	9	12

We sort t'' using radix sort in $\Theta(n)$ time. For our example, this gives us

t''	=	`d1`	`d4`	`d5`	`d8`
pos	=	12	9	3	6

The corresponding positions in the sorted t'' is the suffix array for the suffixes starting at $i \bmod 3 = 0$, so we have $SA_3(T)$ as well.

The correctness of this step is trivial from the correctness of radix sort and the fact that the entries in SA_{12} are unique (so there won't be tie).

Merging the Two Suffix Arrays

The final punchline. We will merge $SA_{12}(T)$ and $SA_3(T)$ into one suffix array in linear time.

Recall that in the $O(n^2)$ algorithm for constructing a suffix array, the merging is done in $O(n^2)$ time using the naive method. The naive

method keeps two pointers to each of the restricted suffix arrays SA_{12} and SA_3. It then compare the suffixes explicitly in worst-case $O(n)$ time. We do this for all the $O(n)$ pairs of positions, giving us an $O(n^2)$ time algorithm.

```
1  i, j = 1, 1
2  while i ≤ |SA12(T)| and j ≤ |SA3(T)|
3        compare suffixes T[SA12(T)[i]...] and T[SA12(T)[j]...]
4        update i, j accordingly
```

However, with the suffixes arrays SA_{12} and SA_3, we can actually do the comparison in constant time. For each arbitrary pair of positions i, j, we only need at most 3 explicit character comparisons before we reach a position i', j' such that $i' = j' \mod 3$, at which point the lexicographic order of the two suffixes can be determined using an $O(1)$ **lookup** in the appropriate restricted suffix array.

For a more detailed procedure for merging, consider the following cases:

Case 1: Compare two suffixes starting at i and j where $i \bmod 3 = 2$ and $j \bmod 3 = 0$. If the encounter a character such that $T[i] \neq T[j]$, then we are done. Otherwise, continue comparing $T[i]$ with $T[j]$ and updating i and j. After at most 2 comparisons, $i \bmod 3 = 1$ and $j \bmod 3 = 2$. We can determine the ordering of the two suffixes by comparing the locations of i and j in SA_{12} in $O(1)$ time.

Case 2: Compare two suffixes starting at i and j where $i \bmod 3 = 1$ and $j \bmod 3 = 0$. If we encounter a character such that $T[i] \neq T[j]$, then we are done. Otherwise, the problem reduces to Case 1, and we can determine the lexicographic ordering of the suffixes starting at i and j with at most 3 explicit comparisons.

Case 3: $i = j \mod 3$. This case is trivial through a constant-time lookup in SA_{12} if $i \bmod 3 = j \bmod 3 \neq 0$ or in SA_3 if $i \bmod 3 = j \bmod 3 = 0$.

In all three cases, we can determine the lexicographic ordering of the two suffixes within $O(1)$ comparisons. We repeat this for all $|T|$ positions, giving us an $O(n)$ time algorithm for merging.

Wrapping It Up

And here we have it, the linear-time algorithm for constructing a suffix array. At first glance, it appears to be quite a sophisticated algorithm, but the ideas behind it are actually quite fundamental. It

based on the same divide-and-conquer approach that previous $O(n^2)$ and $O(n \log n)$ time algorithms have used, but with a few ingenious improvements that allow us to do the merging in $O(n)$ time. Note that the linear-time merging is not possible if we divide the suffixes up into positions 0 or 1 mod 2 since we are not guaranteed to be at a position $i = j \mod 2$ after just a constant number of comparisons.

As we mentioned at the beginning, this algorithm is due to Karkkainen and Sanders. It is often referred to as the ***Karkkainen-Sanders (KS) algorithm*** or the ***DC3 algorithm*** since it is a divide-and-conquer algorithm that divides the positions based on their values modulo 3.

To wrap this section up, let us prove that the KS algorithm indeed runs in linear time.

Theorem 10.2. *The suffix array for text T of length n can be computed in time $O(n)$.*

Proof. We use the Karkkainen-Sanders' algorithm. The correctness of the algorithm is argued as we introduce the algorithm. Now, we consider the runtime of the algorithm.

Sorting of the triples takes $O(n)$ time using radix sort, and so does the computation of the SA for the non-sample suffixes at position $i \bmod 3 = 0$. At each level of the recursion, the suffix array that we recursively construct is of size $\lceil 2/3n \rceil$. Finally, merging takes $O(n)$ time. Hence, the overall runtime is given by the recurrence

$$\begin{aligned} T(n) &= T(\lceil 2/3n \rceil) + 3O(n) \\ &= T(\lceil 2/3n \rceil) + O(n) \\ &\leq n \sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^i \\ &\in O(n) \end{aligned}$$

The same recurrence can also be solved using the Master's theorem. □

11

Burrows-Wheeler Transform

12

FM Index

13
Wavelet Tree

Bibliography

[1] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi: 10.1017/CBO9780511574931.

[2] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015. doi: 10.1017/CBO9781139940023.

Index