

# 8 Strings

## Notation:

- $\Sigma$  is an *alphabet* – a finite set of *characters*.
- $\Sigma^*$  is the set of all *strings* (finite sequences) over  $\Sigma$ .
- We will use Greek letters for string variables, Latin letters for character and numeric variables, and **typewriter** letters for concrete characters. We will make no difference between a character and a single-character string.
- $|\alpha|$  is the *length* of the string  $\alpha$ .
- $\varepsilon$  is the *empty string* — the only string of length 0.
- $\alpha\beta$  is the *concatenation* of strings  $\alpha$  and  $\beta$ ; we have  $\alpha\varepsilon = \varepsilon\alpha = \alpha$  for all  $\alpha$ .
- $\alpha[i]$  is the  $i$ -th character of the string  $\alpha$ ; characters are indexed starting with 0.
- $\alpha[i : j]$  is the *substring*  $\alpha[i]\alpha[i+1] \dots \alpha[j-1]$ ; note that  $\alpha[j]$  is the first character *behind* the substring, so we have  $|\alpha[i : j]| = j - i$ . If  $i \geq j$ , the substring is empty. Either  $i$  or  $j$  can be omitted, the beginning or the end of  $\alpha$  is used instead.
- $\alpha[:j]$  is the *prefix* of  $\alpha$  formed by the first  $j$  characters. A word of length  $n$  has  $n+1$  prefixes, one of them being the empty string.
- $\alpha[i:]$  is the *suffix* of  $\alpha$  from character number  $i$  to the end. A word of length  $n$  has  $n+1$  suffixes, one of them being the empty string.
- $\alpha[: ] = \alpha$ .
- $\alpha \leq \beta$  denotes *lexicographic order* of strings:  $\alpha \leq \beta$  if  $\alpha$  is a prefix of  $\beta$  or if there exists  $k$  such that  $\alpha[k] < \beta[k]$  and  $\alpha[:k] = \beta[:k]$ .

## 8.1 Suffix arrays

**Definition:** The *suffix array* for a string  $\alpha$  of length  $n$  is a permutation  $S$  of the set  $\{0, \dots, n\}$  such that  $\alpha[S[i] :] < \alpha[S[i+1] :]$  for all  $0 \leq i < n$ .

**Claim:** The suffix array can be constructed in time  $\mathcal{O}(n)$ .

Once we have the suffix array for a string  $\alpha$ , we can easily locate all occurrences of a given substring  $\beta$  in  $\alpha$ . Each occurrence corresponds to a suffix of  $\alpha$  whose prefix is  $\beta$ . In the lexicographic order of all suffixes, these suffixes form a range. We can easily find the start and end of this range using binary search on the suffix array. We need  $\mathcal{O}(\log |\alpha|)$  steps, each step involves string comparison with  $\alpha$ , which takes  $\mathcal{O}(|\beta|)$  time in the worst case. This makes  $\mathcal{O}(|\beta| \log |\alpha|)$  total.

$i$	$S[i]$	$R[i]$	$L[i]$	$suffix$
0	14	3	0	$\varepsilon$
1	8	11	3	ananas
2	10	10	2	anas
3	0	7	2	annbansbananas
4	4	4	1	ansbananas
5	12	12	0	as
6	7	14	3	bananas
7	3	6	0	bansbananas
8	9	1	2	nanas
9	11	8	1	nas
10	2	2	1	nbansbananas
11	1	9	1	nnbansbananas
12	5	5	0	nsbananas
13	13	13	1	s
14	6	0	—	sbananas

Figure 8.1: Suffixes of annbansbananas and the arrays  $S$ ,  $R$ , and  $L$

**Corollary:** Using the suffix array for  $\alpha$ , we can enumerate all occurrences of a substring  $\beta$  in time  $\mathcal{O}(|\beta| \log |\alpha| + p)$ , where  $p$  is the number of occurrences reported. Only counting the occurrences costs  $\mathcal{O}(|\beta| \log |\alpha|)$  time.

**Note:** With further precomputation, time complexity of searching can be improved to  $\mathcal{O}(|\beta| + \log |\alpha|)$ .

**Definition:** The *rank array*  $R[0 \dots n]$  is the inverse permutation of  $S$ . That is,  $R[i]$  tells how many suffixes of  $\alpha$  are lexicographically smaller than  $\alpha[i : ]$ .

**Note:** The rank array can be trivially computed from the suffix array in time  $\mathcal{O}(n)$ .

**Definition:** The *LCP array*  $L[0 \dots n - 1]$  stores the length of the *longest common prefix* of each suffix and its lexicographic successor. That is,  $L[i] = \text{LCP}(\alpha[S[i] : ], \alpha[S[i + 1] : ])$ , where  $\text{LCP}(\gamma, \delta)$  is the maximum  $k$  such that  $\gamma[ : k] = \delta[ : k]$ .

**Claim:** Given the suffix array, the LCP array can be constructed in time  $\mathcal{O}(n)$ .

**Observation:** The LCP array can be easily used to find the longest common prefix of any two suffixes  $\alpha[i : ]$  and  $\alpha[j : ]$ . We use the rank array to locate them in the lexicographic order of all suffixes: they lie at positions  $i' = R[i]$  and  $j' = R[j]$  (w.l.o.g.  $i' < j'$ ). Then we compute  $k = \min(L[i'], L[i' + 1], \dots, L[j' - 1])$ . We claim that  $\text{LCP}(\alpha[i : ], \alpha[j : ])$  is exactly  $k$ .

First, each pair of adjacent suffixes in the range  $[i', j']$  has a common prefix of length at least  $k$ , so our LCP is at least  $k$ . However, it cannot be more: we have  $k = L[\ell]$  for some  $\ell \in [i', j' - 1]$ , so the  $\ell$ -th and  $(\ell + 1)$ -th suffix differ at position  $k + 1$  (or one of the suffixes ends at position  $k$ , but we can simply imagine a padding character at the end, ordered before all ordinary characters.) Since all suffixes in the range share the first  $k$  characters, their  $(k + 1)$ -th characters must be non-decreasing. This means that the  $(k + 1)$ -th character of the first and the last suffix in the range must differ, too.

This suggests building a *Range Minimum Query* (RMQ) data structure for the array  $L$ : it is a static data structure, which can answer queries for the position of the minimum element in a given range of indices. One example of a RMQ structure is the 1-dimensional range tree from section ??: it can be built in time  $\mathcal{O}(n)$  and it answers queries in time  $\mathcal{O}(\log n)$ . There exists a better structure with build time  $\mathcal{O}(n)$  and query time  $\mathcal{O}(1)$ .

**Examples:** The arrays we have defined can be used to solve the following problems in linear time:

- *Histogram of  $k$ -grams*: we want to count occurrences of every substring of length  $k$ . Occurrences of every  $k$ -gram correspond to ranges of suffixes in their lexicographic order. These ranges can be easily identified, because we have  $L[\dots] < k$  at their boundaries. We only have to be careful about suffixes shorter than  $k$ , which contain no  $k$ -gram.
- *The longest repeating substring of a string  $\alpha$* : Consider two positions  $i$  and  $j$  in  $\alpha$ . The length of the longest common substring starting at these positions is equal to the LCP of the suffixes  $\alpha[i : ]$  and  $\alpha[j : ]$ , which is a minimum over some range in  $L$ . So it is always equal to some value in  $L$ . It is therefore sufficient to consider only pairs of suffixes adjacent in the lexicographic order, that is to find the maximum value in  $L$ .
- *The longest common substring of two strings  $\alpha$  and  $\beta$* : We build a suffix array and LCP array for the string  $\alpha\#\beta$ , using a separator  $\#$  which occurs in neither  $\alpha$  nor  $\beta$ . We observe that each suffix of  $\alpha\#\beta$  corresponds to a suffix of either  $\alpha$  or  $\beta$ . Like in the previous problem, we want to find a pair of positions  $i$  and  $j$  such that the LCP of the  $i$ -th and  $j$ -th suffix is maximized. We however need one  $i$  and  $j$  to come from  $\alpha$  and the other from  $\beta$ . Therefore we find the maximum  $L[k]$  such that  $S[k]$  comes from  $\alpha$  and  $S[k + 1]$  from  $\beta$  or vice versa.

### Construction of the LCP array: Kasai's algorithm

We show an algorithm which constructs the LCP array  $L$  in linear time, given the suffix array  $S$  and the rank array  $R$ . We will use  $\alpha_i$  to denote the  $i$ -th suffix of  $\alpha$  in lexicographic order, that is  $\alpha[S[i] : ]$ .

We can easily compute all  $L[i]$  explicitly: for each  $i$ , we compare the suffixes  $\alpha_i$  and  $\alpha_{i+1}$  character-by-character from the start and stop at the first difference. This is obviously correct, but slow. We will however show that most of these comparisons are redundant.

Consider two suffixes  $\alpha_i$  and  $\alpha_{i+1}$  adjacent in lexicographic order. Suppose that their LCP  $k = L[i]$  is non-zero. Then  $\alpha_i[1:]$  and  $\alpha_{i+1}[1:]$  are also suffixes of  $\alpha$ , equal to  $\alpha_{i'}$  and  $\alpha_{j'}$  for some  $i' < j'$ . Obviously,  $\text{LCP}(\alpha_{i'}, \alpha_{j'}) = \text{LCP}(\alpha_i, \alpha_{i+1}) - 1 = k - 1$ . However, this LCP is a minimum of the range  $[i', j']$  in the array  $L$ , so we must have  $L[i'] \geq k - 1$ .

This allows us to process suffixes of  $\alpha$  from the longest to the shortest one, always obtaining the next suffix by cutting off the first character of the previous suffix. We calculate the  $L$  of the next suffix by starting with  $L$  of the previous suffix minus one and comparing characters from that position on:

### Algorithm BUILDLCP

*Input:* A string  $\alpha$  of length  $n$ , its suffix array  $S$  and rank array  $R$

1.  $k \leftarrow 0$   $\triangleleft$  The LCP computed in previous step
2. For  $p = 0, \dots, n - 1$ :  $\triangleleft$  Start of the current suffix in  $\alpha$
3.  $k \leftarrow \max(k - 1, 0)$   $\triangleleft$  The next LCP is at least previous  $- 1$
4.  $i \leftarrow R[p]$   $\triangleleft$  Index of current suffix in sorted order
5.  $q \leftarrow S[i + 1]$   $\triangleleft$  Start of the lexicographically next suffix in  $\alpha$
6. While  $(p + k < n) \wedge (q + k < n) \wedge (\alpha[p + k] = \alpha[q + k])$ :
7.  $k \leftarrow k + 1$   $\triangleleft$  Increase  $k$  while characters match
8.  $L[i] \leftarrow k$   $\triangleleft$  Record LCP in the array  $L$

*Output:* LCP array  $L$

**Lemma:** The algorithm BUILDLCP runs in time  $\mathcal{O}(n)$ .

*Proof:* All operations outside the while loop take  $\mathcal{O}(n)$  trivially. We will amortize time spent in the while loop using  $k$  as a potential. The value of  $k$  always lies in  $[0, n]$  and it starts at 0. It always changes by 1: it can be decreased only in step 3 and increased only in step 7. Since there are at most  $n$  decreases, there can be at most  $2n$  increases before  $k$  exceeds  $n$ . So the total time spent in the while loops is also  $\mathcal{O}(n)$ .  $\square$

### Construction of the suffix array by doubling

There is a simple algorithm which builds the suffix array in  $\mathcal{O}(n \log n)$  time. As before,  $\alpha$  will denote the input string and  $n$  its length. Suffixes will be represented by their starting position:  $\alpha_i$  denotes the suffix  $\alpha[i:]$ .

The algorithm works in  $\mathcal{O}(\log n)$  passes, which sort suffixes by their first  $k$  characters, where  $k = 2^0, 2^1, 2^2, \dots$ . For simplicity, we will index passes by  $k$ .

**Definition:** For any two strings  $\gamma$  and  $\delta$ , we define comparison of prefixes of length  $k$ :  $\gamma =_k \delta$  if  $\gamma[ : k] = \delta[ : k]$ ,  $\gamma \leq_k \delta$  if  $\gamma[ : k] \leq \delta[ : k]$ .

The  $k$ -th pass will produce a permutation  $S_k$  on suffix positions, which sorts suffixes by  $\leq_k$ . We can easily compute the corresponding ranking array  $R_k$ , but this time we have to be careful to assign the same rank to suffixes which are equal by  $=_k$ . Formally,  $R_k[i]$  is the number of suffixes  $\alpha_j$  such that  $\alpha_j <_k \alpha_i$ .

In the first pass, we sort suffixes by their first character. Since the alphabet can be arbitrarily large, this might require a general-purpose sorting algorithm, so we reserve  $\mathcal{O}(n \log n)$  time for this step. The same time obviously suffices for construction of the ranking array.

In the  $2k$ -th pass, we get suffixes ordered by  $\leq_k$  and we want to sort them by  $\leq_{2k}$ . For any two suffixes  $\alpha_i$  and  $\alpha_j$ , the following holds by definition of lexicographic order:

$$\alpha_i \leq_{2k} \alpha_j \iff (\alpha_i <_k \alpha_j) \vee (\alpha_i =_k \alpha_j) \wedge (\alpha_{i+k} \leq_k \alpha_{j+k}).$$

Using the ranking function  $R_k$ , we can write this as lexicographic comparison of pairs  $(R_k[i], R_k[i+k])$  and  $(R_k[j], R_k[j+k])$ . We can therefore assign one such pair to each suffix and sort suffixes by these pairs. Since any two pairs can be compared in constant time, a general-purpose sorting algorithm sorts them in  $\mathcal{O}(n \log n)$  time. Afterwards, the ranking array can be constructed in linear time by scanning the sorted order.

There remains a little problem: the suffixes  $\alpha_i$  and  $\alpha_j$  can be shorter than  $2k$  characters. In that case,  $i+k$  and/or  $j+k$  can point outside  $\alpha$ . This is easy to fix: we replace any out-of-range suffix by the empty suffix, whose rank is always zero. (Alternatively, we can imagine that  $\alpha$  is padded by  $n$  more null characters, which are smaller than all regular characters. This way, all suffixes will be well defined and  $\leq_k$  will always compare exactly  $k$  characters.)

Overall, we have  $\mathcal{O}(\log n)$  passes, each taking  $\mathcal{O}(n \log n)$  time. The whole algorithm therefore runs in  $\mathcal{O}(n \log^2 n)$  time. In each pass, we need to store only the input string  $\alpha$ , the ranking array from the previous step, the suffix array of the current step, and the encoded pairs. All this fits in  $\mathcal{O}(n)$  space.

We can improve time complexity by using Bucketsort to sort the pairs. As the pairs contain only numbers between 0 and  $n$ , we can sort in two passes with  $n$  buckets. This takes  $\mathcal{O}(n)$  time, so the whole algorithm runs in  $\mathcal{O}(n \log n)$  time. Please note that the first pass still remains  $\mathcal{O}(n \log n)$ , unless we can assume that the alphabet is small enough to index buckets. Space complexity stays linear.