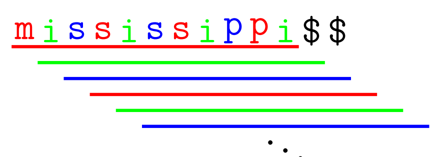
17.2 A linear-time construction algorithm

We can take this "coding" idea one level further to obtain a $O(|T|)$-time algorithm to create the suffix array. There are a few linear-time suffix array construction algorithms. The one we will see is due to Kärkkäinen and Sanders [2003].

For simplicity, we make the the text length a multiple of 3 after padding with a special character. Assume string indices start at 0. We divide the suffixes conceptually into 3 groups:

- Group 0: Suffixes starting at positions $i = 0, 3, 6, 9, \cdots = (i \mod 3 = 0)$
- Group 1: Suffixes starting at positions $i = 1, 4, 7, 10, \cdots = (i \mod 3 = 1)$
- Group 2: Suffixes starting at positions $i = 2, 5, 8, 11, \cdots = (i \mod 3 = 2)$

This gives us the following groupings for "`mississippi$`", for example:



(17.4)

The basic outline of the algorithm is to recursively handle suffixes from the $i \mod 3 = 1$ and $i \mod 3 = 2$ groups and then merge the $i \mod 3 = 0$ group after each recursion. We now describe the steps taken by the algorithm, which is called the "Skew Algorithm".

Step 1: create T'. We first create a new string T' that is the concatenation of $T[1 \ldots]$ (that is T with its first character removed) and $T[2 \ldots]$ (that is T with its first and second character removed). Suppose $T =$ `mississippi`, then we have:

$T' =$

iss	iss	ipp	i$$	ssi	ssi	ppi

(17.5)

This puts the group-1 suffixes starting in the first part of T' and the group-2 suffixes starting in the second part of T. This at most doubles the size of T and takes $O(|T|)$ time. We conceptually divide T' into blocks of length 3, as shown above.

Step 2: encode T'. We then encode each block of 3 using a new alphabet where if C_i and C_j are the codes for 3-blocks i and j then $C_i < C_j$ if and only if block i is lexicographically before block j (and $C_i = C_j$ if blocks i and j are the same 3 letters). We can do this by sorting each of the 3-blocks using a radix sort (takes $O(|T|)$ time) and assigning the new code corresponding to the sorted order. This gives us a new coded string t:

$T' =$	iss	iss	ipp	i$$	ssi	ssi	ppi
$t =$	C	C	B	A	E	E	D

(17.6)

Key Point #1: The lexicographical order of the suffixes of the coded string t is the same as the order of the group 1 and 2 suffixes of T. Why? Every suffix of t corresponds to some suffix of T (perhaps with some extra letters at the end of it — in this case the extra characters are "EED"). Because the tokens are sorted in the same order as the triples, the sort order of the suffix of t matches that of T. Therefore, we can recursively compute the suffix array for t to get the ordering of the group 1 and group 2 suffixes.

Step 3: recursively compute the suffix array for t. In the example for mississippi$, we obtain the following suffix array A from the recursive call:

```
3 AEED
2 BAEED
1 CBAEED
0 CCBAEED
6 D
5 ED
4 EED
```

(17.7)

and $A = [3, 2, 1, 0, 6, 5, 4]$. Expanding the coding back, we would obtain a *partial* suffix array for T that only includes the suffixes in group 1 and group 2.

Step 4: create the inverse suffix array. For the next steps, we need to know the position of suffix i in the suffix array. This is easy to compute from A: We create a new array S where S_i is to the position of i in the suffix array. If A was the full suffix array of T, we could do this with a single scan down A by setting $S_{A[i]} = i$. Because A is actually the partial suffix array of T', we have to do a little extra arithmetic to translate suffix numbers from T' to T and accounting for the missing suffixes. This can still be done in one pass down A. See Exercise **17.2**.

Step 5: sort the group-0 suffixes. Group-0 suffixes are related to group-1 suffixes. Specifically, we can encode a group-0 suffix as the combination of a letter followed by a group-1 suffix. If $i = 0 \mod 3$ then suffix T_i can be represented by

$$(T[i], T[i+1, \dots]). \tag{17.8}$$

Here, $T[i+1 \dots]$ is a group-1 suffix. This is a really clever insight that we will use in later steps. We therefore can encode group-0 suffixes using

$$(T[i], S_{i+1}), \tag{17.9}$$

where S_{i+1} is entry in the inverse suffix array S that we computed in the previous step corresponding to suffix $i + 1$, which is a group-1 suffix.

Now we can sort the group-0 suffixes using this encoding, again using a radix sort since they have only two digits. This gives us a sorted list L of the group-0 suffixes. This all takes $O(|T|)$ time.

Step 6: merge the group-0 suffixes back in. We have to add in the group 0 suffixes into our partial suffix array that contains group-1 and group-2 suffixes. The way to do this is to run a list merge algorithm. You're likely familiar with the list merging done in (say) merge sort. We use that here with our two lists: the list A of Group-1 and 2 suffixes and the list L of 0-suffixes, which by the previous steps are each sorted lists. Such a list merge takes $O(|T|)$ if we can compare the items in $O(1)$ time.

The challenge is how to compare an item from the group-0 list L with an item from the group-$\{1, 2\}$ list A. To do this, we use the clever idea about the relationship between the suffixes again.

To compare a group-0 suffix j with a group-1 suffix i, we can test whether

$$\underbrace{(T[i], S_{i+1})}_{\text{group 1 suffix}} < \underbrace{(T[j], S_{j+1})}_{\text{group 0 suffix}}? \tag{17.10}$$

Equation (17.10) is true if and only if the group-1 suffix is lexicographically before the group-0 suffix. To compare a group-0 suffix j with a group-2 suffix i, we can test whether:

$$\underbrace{(T[i], T[i+1], S_{i+2})}_{\text{group 2 suffix}} < \underbrace{(T[j], T[j+1], S_{j+2})}_{\text{group 0 suffix}}? \tag{17.11}$$

The reason for the particular encodings as 2- and 3-tuples is that in each case $S_{i+1}, S_{i+2}, S_{j+1}, S_{j+2}$ are either group-1 or group-2. Suppose $i = 1 \pmod 3$. Then the test we have to do is:

$$(T[i], \underbrace{S_{i+1}}_{i+1 \equiv 2 \bmod 3}) < (T[j], \underbrace{S_{j+1}}_{j+1 \equiv 1 \bmod 3}). \tag{17.12}$$

On the other hand if $i = 2 \pmod 3$, then the test we have to do is:

$$(T[i], T[i+1], \underbrace{S_{i+2}}_{i+2 \equiv 1 \bmod 3}) < (T[j], T[j+1], \underbrace{S_{j+2}}_{j+2 \equiv 2 \bmod 3}). \tag{17.13}$$

Since S_k gives the relative position of suffix k among the group-$\{1, 2\}$ suffixes, we can do the above tests by comparing these tuples directly. In either case we are comparing tuples of at most 3 items, each of these comparisons takes $O(1)$ time, and our list merge to merge A and L takes the total lengths of the lists we are merging $O(|T|)$ since we do constant work for each comparison. We now have a complete suffix array containing all the suffixes.

17.2.1 Running time

Theorem 17.1 (Skew algorithm running time). *The Skew algorithm described above takes $O(|T|)$ to create the suffix array for a string T.*

Proof: For a string of length n, the recurrence for the algorithm is:

$$T(n) = O(n) + T(2n/3), \tag{17.14}$$

where the first term is the time to sort and merge and the second term comes from the fact that the array in the recursive call is 2/3rds the size of the starting array.

So, we have $T(n) \le cn + T(2n/3)$ for some c. Suppose we "guess" that $T(n) \le 3cn$. Certainly, this is true when n is 1 for large enough c, so that takes care of the base case. We prove the general statement by induction, assuming it is true for all $i < n$. Then we have:

$$T(n) \le cn + 3c(2n/3) \quad \text{by the I.H.} \tag{17.15}$$
$$= cn + 2cn \tag{17.16}$$
$$= 3cn. \tag{17.17}$$

□

17.3 Summary and notes

We've seen a succession of more and more efficient algorithms for suffix array construction, ending up with a linear-time algorithm. The non-naïeve algorithms use an encoding of the string that preserves some sorting information plus a linear-time sort algorithm, which is possible since our encodings are all a constant number of digits. The simpler algorithm of Section 17.1 is probably fine for all but the longest strings, since the extra $O(\log n)$ factor is likely not too bad. Puglisi et al. [2007] give a survey and synthesis of various suffix array construction algorithms.

Kasai's algorithm [Kasai et al., 2001] can be used to construct the LCP array for an already constructed suffix array in linear time.

Presentation Notes

Our presentation of the Skew algorithm follows its original description [Kärkkäinen and Sanders, 2003].

17.4 Exercises

17.1 Let Σ be a constant-sized alphabet. Describe how to sort m length-k strings over Σ in $O(m)$ time, assuming k is a constant. Your answer should not be more than 2 sentences.

17.2 In Step 4 of the linear-time suffix array construction algorithm due to Kärkkäinen and Sanders (Section 17.2), the algorithm must be able to access the inverse suffix array. In particular, the algorithm requires a function $f(i_T) \rightarrow j_A$ that returns the location j_A in the partial suffix array A computed by the algorithm in this recursion of the suffix corresponding to the suffix starting at index i_T of *the string* T *that is input to this recursive call*. In order to compute f, you may want to precompute some values.

Give a careful pseudocode implementation of the function f and any pre-computation function pre. You may assume pre and f have access to any of the data that is available at step 4 of the algorithm. f should run in constant time and pre should run in at most $O(|A|)$ time.