Algorithmic Problem Solving

Le 8 Strings II

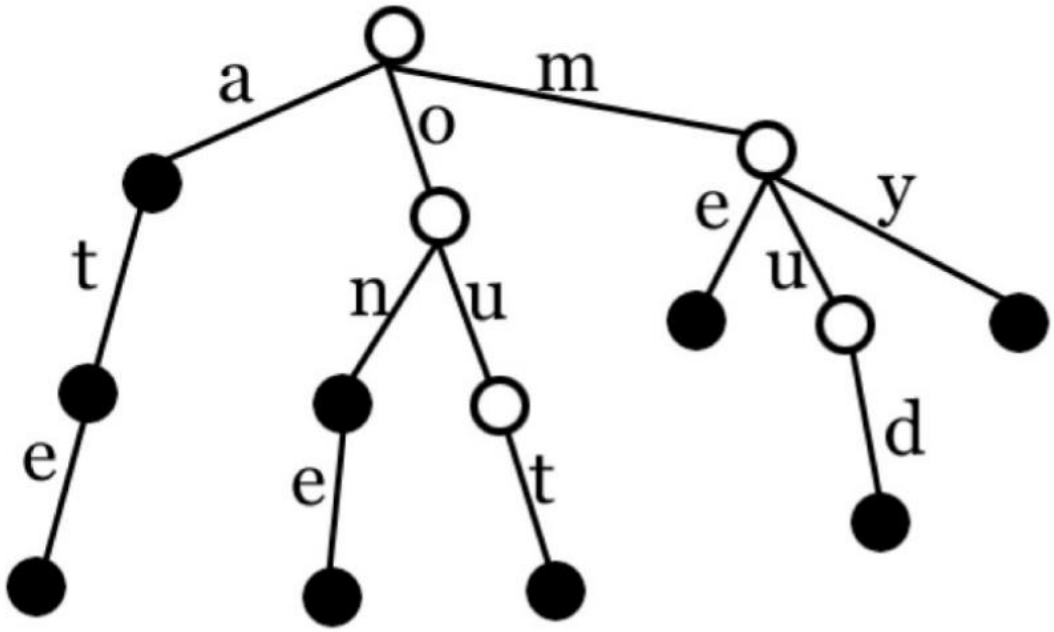
Herman Appelgren
Dept of Computer and Information Science
Linköping University

Outline

- ~~Exercise 8: Strings I~~
  - ~~A: Exercise Evil Straw Warts Live~~
  - ~~B: Dictionary Attack~~
  - ~~B: Dominant Strings~~
  - ~~C: Intellectual Property~~
- Suffix Trie & Suffix Tree
- Suffix Array (Lab 3.2)
  - Construction (Lab 3.2)
  - Longest Common Prefix extension (Lab 3.3)

Repetition from last lecture

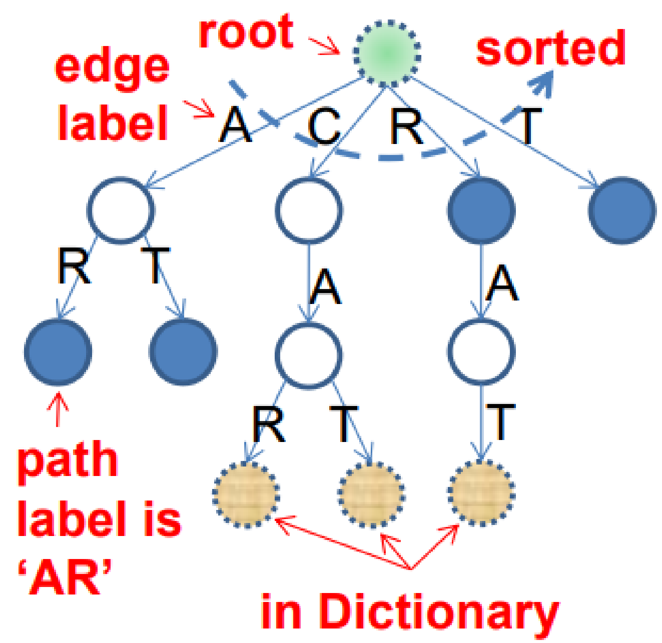
- **Suffixes** are substrings at the end of the string.
  - Example: "banana" (not proper), "anana", "nana", "ana", "na", "a", "".
- **Prefixes** are substring at the start of the string.
  - Example: "banana" (not proper), "banan", "bana", "ban", "ba", "b", "".
- A Trie is a rooted tree structure used for storing a set of strings and optimize prefix searches.



Suffix Trie

A ***suffix trie*** is a trie built for all suffixes of a set of strings

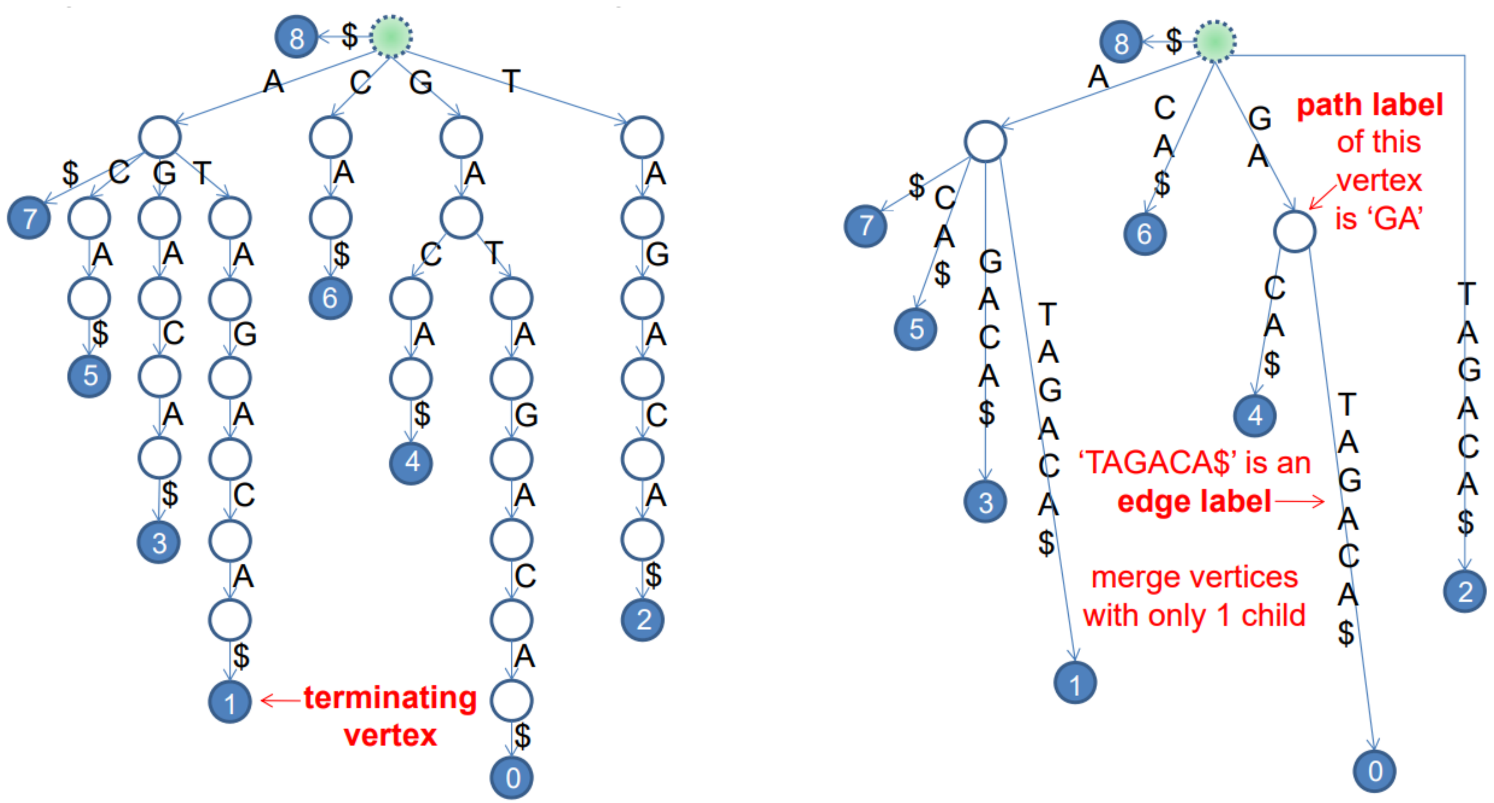
- **Check whether S is a substring of T.**
  - Follow the path for S from the root.
  - If you exhaust S, then S is in T.
- **Check whether S is a suffix of T.**
  - Follow the path for S from the root.
  - If you end at a leaf at the end of S, then S is a suffix of T.
- **Count # of occurrences of S in T.**
  - Follow the path for S from the root.
  - The result is the sum of the # of suffixes represented by the leaves under the node you end up in.



(example by Steven Halim)

Suffix Trie vs Suffix Tree

- Suffix Tree improves Suffix Trie by compressing internal nodes.
- Suffix Tries are easy to construct, Suffix Trees not so much…



(example by Steven Halim)

Suffix Tree

- String Multimatching in O(|P| + n_matches)
  - The matches are contained in the leaves in the subtree rooted at the node representing the pattern.
- Longest Repeated Substring in O(|T|)
  - Find the deepest internal node.
- Longest Common Substring in O(|T|)
  - Find the deepest internal node with leaves from both strings.

Suffix Array (Lab 3.2)

- The Suffix Array is a sorted array of all suffixes of a string.
  - Solves most problems Suffix Trees does with comparable time complexity.
  - Much easier to implement (but still far from trivial).
  - One of the most important data structures in this course!

i	Suffix
0	abcabcaaa
1	bcabcaaa
2	cabcaaa
3	abcaaa
4	bcaaa
5	caaa
6	aaa
7	aa
8	a

Sort =>

i	SA[i]	Suffix
0	8	a
1	7	aa
2	6	aaa
3	3	abcaaa
4	0	abcabcaaa
5	4	bcaaa
6	1	bcabcaaa
7	5	caaa
8	2	cabcaaa

- SA[i] = starting index of suffix i in sorted order.
- Don't store the suffixes explicitly!
  - Copying would take O(n^2) time and memory.
  - Store single copy of S, extract suffixes using SA if required.

Suffix Array – String Matching

- String Matching: Find all occurences of P in T.
  - Create Suffix Array of T.
  - Use binary search to find first and last suffix that starts with P.
  - 2 binary searches, each comparison takes at most O(|P|) => O(|P|log|T|).
  - Example: T = "abcabcaaa" and P = "ab" or P = "ac".

i	SA[i]	Suffix
0	8	a
1	7	aa
2	6	aaa
3	3	**abcaaa**
4	0	**abcabcaaa**
5	4	bcaaa
6	1	bcabcaaa
7	5	caaa
8	2	cabcaaa

i	SA[i]	Suffix
0	8	a
1	7	aa
2	6	aaa
3	3	abcaaa
4	0	abcabcaaa
5	4	bcaaa
6	1	bcabcaaa
7	5	caaa
8	2	cabcaaa

Suffix Array – Construction

- Naive implementation
  - Standard sort by comparing suffixes.
  - Time complexity in O(N^2logN), since comparisons take O(N).
  - Not feasible even for moderately large strings!

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

char T[MAX_N]; int SA[MAX_N];

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }

int main() {
   int n = (int)strlen(gets(T));
   for (int i = 0; i < n; i++) SA[i] = i;
   sort(SA, SA + n, cmp);
}
```

This is O(N)

What is the time complexity?
e do better?

Overall $O(N^2$ log

(example by Steven Halim)

Suffix Array – Construction

- Idea: Sort multiple times, comparing only parts of the string.
  - In iteration k, only compare the first 2^k characters.
  - Reuse results to avoid increasing work between iterations.
- Sort suffixes lexicographically based on C1 and C2.
- Iteration 1: Set C1[i] and C2[i] to ASCII values of first two chars.

i	SA[i]	Suffix	C1[i]	C2[i]
0	0	abcabcaaa	97	98
1	1	bcabcaaa	98	99
2	2	cabcaaa	99	97
3	3	abcaaa	97	98
4	4	bcaaa	98	99
5	5	caaa	99	97
6	6	aaa	97	97
7	7	aa	97	97
8	8	a	97	0

=>

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	97	0
1	6	aaa	97	97
2	7	aa	97	97
3	0	abcabcaaa	97	98
4	3	abcaaa	97	98
5	1	bcabcaaa	98	99
6	4	bcaaa	98	99
7	2	cabcaaa	99	97
8	5	caaa	99	97

Suffix Array – Construction

- Iteration 2: Sort by the first four characters.
  - We already know the relative order of the second pair of characters from each suffix, since they are the first two characters of another suffix!
- C1[i] = rank of suffix starting at SA[i].
- C2[i] = rank of suffix starting at SA[i] + 2, or 0 if SA[i] + 2 >= n.

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	97	0
1	6	aaa	97	97
2	7	aa	97	97
3	0	abcabcaaa	97	98
4	3	abcaaa	97	98
5	1	bcabcaaa	98	99
6	4	bcaaa	98	99
7	2	cabcaaa	99	97
8	5	caaa	99	97

=>

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	1	0
1	6	aaa	2	1
2	7	aa	2	0
3	0	abcabcaaa	3	5
4	3	abcaaa	3	5
5	1	bcabcaaa	4	3
6	4	bcaaa	4	2
7	2	cabcaaa	5	4
8	5	caaa	5	2

Suffix Array – Construction

- Sort based on new C1 and C2.

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	1	0
1	6	aaa	2	1
2	7	aa	2	0
3	0	abcabcaaa	3	5
4	3	abcaaa	3	5
5	1	bcabcaaa	4	3
6	4	bcaaa	4	2
7	2	cabcaaa	5	4
8	5	caaa	5	2

=>

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	1	0
1	7	aa	2	0
2	6	aaa	2	1
3	0	abcabcaaa	3	5
4	3	abcaaa	3	5
5	4	bcaaa	4	2
6	1	bcabcaaa	4	3
7	5	caaa	5	2
8	2	cabcaaa	5	4

Suffix Array – Construction

- Iteration 3: Sort by the first eight character.
- Update C1 and C2 in the same way, but now with step length 4.

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	1	0
1	7	aa	2	0
2	6	aaa	2	1
3	0	abcabcaaa	3	5
4	3	abcaaa	3	5
5	4	bcaaa	4	2
6	1	bcabcaaa	4	3
7	5	caaa	5	2
8	2	cbacaaa	5	4

=>

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	1	0
1	7	aa	2	0
2	6	aaa	3	0
3	0	abcabcaaa	4	5
4	3	abcaaa	4	2
5	4	bcaaa	5	1
6	1	bcabcaaa	6	7
7	5	caaa	7	0
8	2	cbacaaa	8	3

Suffix Array – Construction

- Sort based on new C1 and C2.

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	1	0
1	7	aa	2	0
2	6	aaa	3	0
3	0	abcabcaaa	4	5
4	3	abcaaa	4	2
5	4	bcaaa	5	1
6	1	bcabcaaa	6	7
7	5	caaa	7	0
8	2	cbacaaa	8	3

=>

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	1	0
1	7	aa	2	0
2	6	aaa	3	0
3	3	abcaaa	4	2
4	0	abcabcaaa	4	5
5	4	bcaaa	5	1
6	1	bcabcaaa	6	7
7	5	caaa	7	0
8	2	cbacaaa	8	3

Suffix Array – Construction

- Iteration 4: Sort by first sixteen characters (i.e. entire string).
- Update C1 and C2 with step length 8
- All strings have unique C1, so we terminate without sorting.

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	1	0
1	7	aa	2	0
2	6	aaa	3	0
3	3	abcaaa	4	2
4	0	abcabcaaa	4	5
5	4	bcaaa	5	1
6	1	bcabcaaa	6	7
7	5	caaa	7	0
8	2	cbacaaa	8	3

=>

i	SA[i]	Suffix	C1[i]	C2[i]
0	8	a	1	0
1	7	aa	2	0
2	6	aaa	3	0
3	3	abcaaa	4	0
4	0	abcabcaaa	5	1
5	4	bcaaa	6	0
6	1	bcabcaaa	7	0
7	5	caaa	8	0
8	2	cbacaaa	9	0

Suffix Array – Construction

- Time complexity
  - At most O(logn) iterations needed.
  - With comparison based sort, total time complexity is $O(n(\log n)^2)$.
- Improvement: Both C1 and C2 are integers, so we can use integer-specific sorting algorithms!

Integer Sorting

- Counting Sort of single integers
  - Let count[i] = number of elements with value i.
  - Then, elements with value i should be placed on indicies [sum(count[j < i]), sum(count[j <= i])) in sorted order.

```
vector<int> CountingSort(const vector<int>& v) {
  int m = *max_element(v.begin(), v.end());
  vector<int> count(m + 1, 0);
  for (int val : v) ++count[val];
  vector<int> idx(m + 1, 0);
  for (int i = 1; i < count.size(); ++i) {
    idx[i] = idx[i - 1] + count[i - 1];
  }
  vector<int> res(v.size());
  for (int val : v) {
    res[idx[val]] = val;
    ++idx[val];
  }
  return res;
}
```

Integer Sorting

- Radix Sort of integer tuples.
  - Counting Sort is stable.
  - To sort the tuples lexicographically, first sort the last integer using Counting Sort, then the second to last, …

$C_1[i]$	$C_2[i]$
1	0
2	1
2	0
3	5
3	5
4	4
4	2
5	4
5	2

=>

$C_1[i]$	$C_2[i]$
1	0
2	0
2	1
4	2
5	2
4	4
5	4
3	5
3	5

=>

$C_1[i]$	$C_2[i]$
1	0
2	0
2	1
3	5
3	5
4	2
4	4
5	2
5	4

Suffix Array – Construction

- Suffix Array in O(n^2logn)
  - Naïve construction by direct comparison of suffixes.
  - Useful baseline, but no course credits.
- Suffix Array in O(n(logn)^2)
  - Comparison-based sorting algorithm operating on bigrams.
  - Sufficient for course credits, but might need optimization to pass time limits, especially for exercises/sessions.
- Suffix Array in O(nlogn)
  - Radix sort of bigrams.
  - Solves all relevant course problems handily.
- Suffix Array in O(n)
  - Interesting if you want a challenge. Ask Leif for directions.
- Suffix Tree in O(n)
  - Very challenging, but doable. See e.g. Ukkonen's algorithm.

Suffix Array – Implementation

- There are multiple arrays of indices with different meaning. Make sure you understand their purpose!
- Troubleshooting advice
  - Most bugs arise when comparing suffixes towards the end of the text.
  - To troubleshoot, print the array on a nice format, e.g. the one used in the previous slides. Makes it easy to manually check correctness.
  - Good testcases are strings with repeating subpatterns, but natural sentences are often sufficient.
- Some implementations append a terminating character
  - Removes some special cases, since all "real" suffixes then has a preceding entry in the table
  - Might make other parts of the code less intuitive.

Suffix Array – Longest Common Prefix (Lab 3.3)

- To solve many Suffix Array problems, we need the Longest Common Prefix extension.
  - LCP[i] is the length of longest prefix shared between SA[i] and SA[i – 1].

i	SA[i]	LCP[i]	Suffix
0	6	0	aab
1	7	1	ab
2	3	2	abcaab
3	0	4	abcabcaab
4	8	0	b
5	4	1	bcaab
6	1	3	bcabcaab
7	5	0	caab
8	2	2	cabcaab

Suffix Array – Longest Repeated Substring

- Find the longest substring that occurs at least twice in the text.
  - Recall: Every substring is a prefix of a suffix.
  - The longest repeated substring is simply the largest LCP entry.

i	SA[i]	LCP[i]	Suffix
0	6	0	aab
1	7	1	ab
2	3	2	abcaab
3	0	4	abcabcaab
4	8	0	b
5	4	1	bcaab
6	1	3	bcabcaab
7	5	0	caab
8	2	2	cabcaab

Suffix Array – Longest Common Substring

- Find the longest substring contained in both T1 and T2.
  - Concatenate the strings, separated by a character not contained in T1 or T2.
  - Find the largest LCP value corresponding to a prefix from different texts.
  - Example: “abcabca” and “aabcb”

owner	LCP	Suffix
T1	0	#aabcb
T1	0	a#aabcb
T2	1	aabcb
T1	1	abca#aabcb
T1	4	abcabca#aabcb
T2	3	abcb
T2	0	b
T1	1	bca#aabcb
T1	3	bcabca#aabcb
T2	2	bcb
T1	0	ca#aabcb
T1	2	cabca#aabcb
T2	1	cb

Suffix Array – Longest Common Substring

- Generalizes to more than two texts.
  - Concatenate the texts.
  - Don't count the separators towards LCP, or use unique separators!
  - If m is the smallest LCP value in a range corresponding to suffixes from all texts, then there is a common substring of length m.
  - Find the largest such m.
  - Example: "ab", "abc", "a", "aaab"

owner	LCP	Suffix
T2	0	#a#aaab
T3	0	#aaab
T1	0	#abc#a#aaab
T3	0	a#aaab
T4	1	aaab
T4	2	aab
T4	1	ab
T1	2	ab#abc#a#aaab
T2	2	abc#a#aaab
T4	0	b
T1	1	b#abc#a#aaab
T2	1	bc#a#aaab
T2	0	c#a#aaab

Suffix Array – Computing LCP

- The naïve algorithm (comparing each suffix pair) is quadratic, which is unacceptably slow!

- It is actually easier to compute LCP in unsorted order.
  - Note that in this order, LCP never decreases by more than one.
  - Thus, when computing LCP[i], we can start at LCP[i - 1] – 1.
  - Since LCP decrements at most once per suffix and cannot exceed n, this reduces the time complexity to O(n).

i	SA[i]	LCP[i]	Suffix
3	0	4	abcabcaab
6	1	3	bcabcaab
8	2	2	cabcaab
2	3	1	abcaab
5	4	0	bcaab
7	5	0	caab
0	6	0	aab
1	7	1	ab
4	8	0	b

Suffix Array – Computing LCP

The first suffix has index 3 in sorted order.
Find index 2 (O(1) if precomputed in O(n))
First 4 characters match.

i	SA[i]	LCP[i]	Suffix
3	0	4	abcabcaab
6	1		bcabcaab
8	2		cabcaab
2	3		abcaab
5	4		bcaab
7	5		caab
0	6		aab
1	7		ab
4	8		b

The second suffix has index 6
Find index 5
No need to recheck first 3 characters

i	SA[i]	LCP[i]	Suffix
3	0	4	abcabcaab
6	1	3	bcabcaab
8	2		cabcaab
2	3		abcaab
5	4		bcaab
7	5		caab
0	6		aab
1	7		ab
4	8		b

Suffix Array – Computing LCP

The third suffix has index 8 in sorted order
Find index 7
No need to recheck first 2 characters

i	SA[i]	LCP[i]	Suffix
3	0	4	abcabcaab
6	1	3	bcabcaab
8	2	2	cabcaab
2	3		abcaab
5	4		bcaab
7	5		caab
0	6		aab
1	7		ab
4	8		b

The fourth suffix has index 2
Find index 1
No need to recheck first character

i	SA[i]	LCP[i]	Suffix
3	0	4	abcabcaab
6	1	3	bcabcaab
8	2	2	cabcaab
2	3	2	abcaab
5	4		bcaab
7	5		caab
0	6		aab
1	7		ab
4	8		b

Suffix Array – Computing LCP

The fifth suffix has index 5 in sorted order
Find index 4
No need to check first character

i	SA[i]	LCP[i]	Suffix
3	0	4	abcabcaab
6	1	3	bcabcaab
8	2	2	cabcaab
2	3	2	abcaab
5	4	1	bcaab
7	5		caab
0	6		aab
1	7		ab
4	8		b

i	SA[i]	LCP[i]	Suffix
3	0	4	abcabcaab
6	1	3	bcabcaab
8	2	2	cabcaab
2	3	2	abcaab
5	4	1	bcaab
7	5	0	caab
0	6		aab
1	7		ab
4	8		b

Suffix Array – Computing LCP

The 7th suffix has index 0 in sorted order
Has no preceeding suffix, so LCP = 0

i	SA[i]	LCP[i]	Suffix
3	0	4	abcabcaab
6	1	3	bcabcaab
8	2	2	cabcaab
2	3	2	abcaab
5	4	1	bcaab
7	5	0	caab
0	6	0	aab
1	7		ab
4	8		b

i	SA[i]	LCP[i]	Suffix
3	0	4	abcabcaab
6	1	3	bcabcaab
8	2	2	cabcaab
2	3	2	abcaab
5	4	1	bcaab
7	5	0	caab
0	6	0	aab
1	7	1	ab
4	8		b

Suffix Array – Computing LCP

Change to sorted suffix order

i	SA[i]	LCP[i]	Suffix
3	0	4	abcabcaab
6	1	3	bcabcaab
8	2	2	cabcaab
2	3	2	abcaab
5	4	1	bcaab
7	5	0	caab
0	6	0	aab
1	7	1	ab
4	8	0	b

i	SA[i]	LCP[i]	Suffix
0	6	0	aab
1	7	1	ab
2	3	2	abcaab
3	0	4	abcabcaab
4	8	0	b
5	4	1	bcaab
6	1	3	bcabcaab
7	5	0	caab
8	2	2	cabcaab

Outline

- Exercise 8: Strings I
  - A: Exercise Evil Straw Warts Live
  - B: Dictionary Attack
  - B: Dominant Strings
  - C: Intellectual Property
- Suffix Trie & Suffix Tree
- Suffix Array (Lab 3.2)
  - Construction (Lab 3.2)
  - Longest Common Prefix extension (Lab 3.3)