Suffix Trees: A Natural History

Martin Farach-Colton

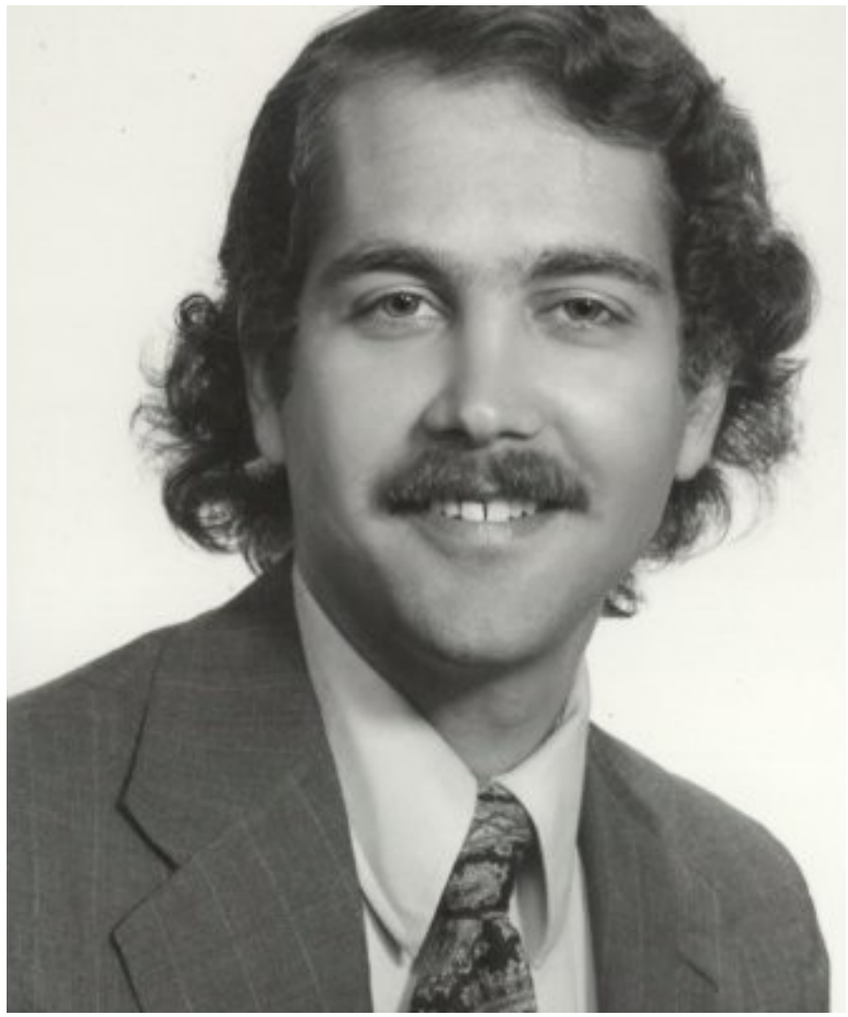
Rutgers University

What’s in this talk?

The history and gossip of suffix trees

Peter Weiner helped start the Yale CS Department

- Like all administrators, he was short on research time
- So he took a sabbatical to solve one big problem



What did big problems look like in the early 70s?

- What is O(n) vs what is O(n log n)?
- Planarity testing, strongly connected components, etc.
- Is anything Ω(n log n)???

Repeated strings

Given a string S of length n, what is longest substring that occurs twice?

- Is this nlogn-time or linear?

Karp-Miller-Rosenberg solves this in O(n log n)

Knuth conjectured that the bound is Ω(n log n)

- Knuth is right in many models
- Lower bound based on the element distinctness problem

So even for first major open problem in stringology, we need to focus on the alphabet!

Karp Miller Rosenberg Main Idea

Find a fingerprint for substrings

- Two substrings have the same fingerprint iff they they are equal

Not all fingerprints

- Substrings starting at any position
- Of any length a power of 2
- So now we can compare any two substrings in log time

So we need to compute O(*n* log *n*) fingerprints

- KMR show how to compute them in O(n log n) time

Karp-Miller-Rosenberg: Building Blocks

Replace-by-Rank (RbR)

- Given a set S, we define

$$\mathbf{R} : \Sigma^n \rightarrow [n]^n$$

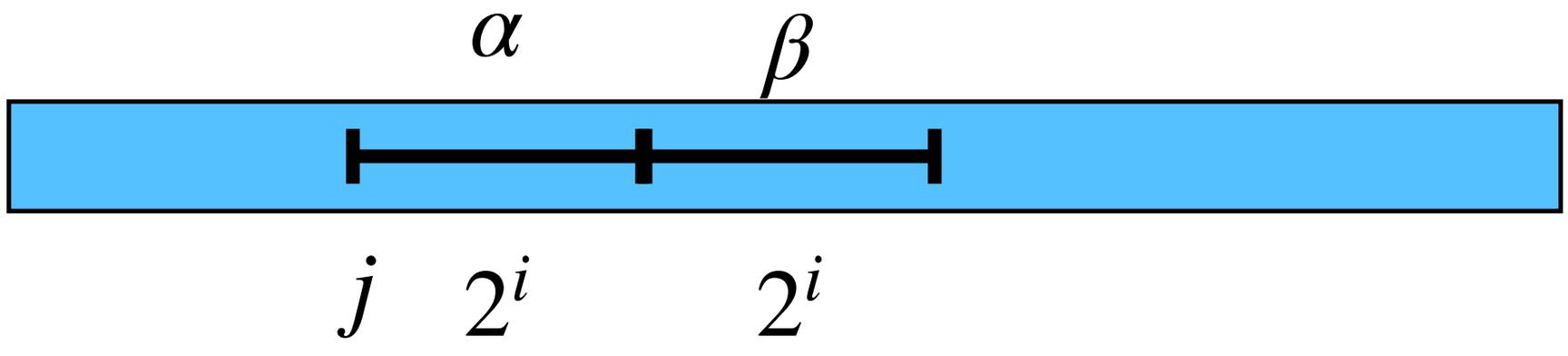
- Where R(S) replaces every character in S by its rank
  - R('aabbadaccaa') = '00110302200'
- Notice: First call to RbR has runtime that depends on the sortability of S

Bit Concat (a.k.a *chunking*):

$$\forall x, y \in \Sigma, \langle x, y \rangle = x|\Sigma| + y$$

e.g. $\langle 3,2 \rangle = 3 \times 4 + 2 = 14$

Karp-Miller-Rosenberg: Fingerprints



At each j, for length 2^{i+1} we first compute:

$$\langle \alpha, \beta \rangle$$

Then we Replace by rank the new fingerprints

- So that the number of bits doesn't blow up

Bonus Points

Reducing the number of bits:

- In Karp-Miller-Rosenberg, it's Replace by Rank
- In Karp-Rabin, it's modding by a random prime

Karp-Miller-Rosenberg: Fingerprints, Example

$S_0 = \text{mississippippiss}$

$S_0' = \mathbf{R}(S_0) = 1 \cdot 0 \cdot 3 \cdot 3 \cdot 0 \cdot 3 \cdot 3 \cdot 0 \cdot 2 \cdot 2 \cdot 0 \cdot 2 \cdot 2 \cdot 0 \cdot 3 \cdot 3$

$S_1 = \langle 1,0\rangle \cdot \langle 0,3\rangle \cdot \langle 3,3\rangle \cdot \langle 3,0\rangle \cdot \langle 0,3\rangle \cdot \langle 3,3\rangle \cdot \langle 3,0\rangle \cdot \langle 0,2\rangle \cdot \langle 2,2\rangle \cdot \langle 2,0\rangle \cdot \langle 0,2\rangle \cdot \langle 2,2\rangle \cdot \langle 2,0\rangle \cdot \langle 0,3\rangle \cdot \langle 3,3\rangle \cdot \langle 3,-\rangle$

$S_2 = \mathbf{R}(S_1') = 2 \cdot 1 \cdot 6 \cdot 5 \cdot 1 \cdot 6 \cdot 5 \cdot 0 \cdot 4 \cdot 3 \cdot 0 \cdot 4 \cdot 3 \cdot 1 \cdot 6 \cdot 7$

$S_2' = \langle 2,6\rangle \cdot \langle 1,5\rangle \cdot \langle 6,1\rangle \cdot \langle 5,6\rangle \cdot \langle 1,5\rangle \cdot \langle 6,0\rangle \cdot \langle 5,4\rangle \cdot \langle 0,3\rangle \cdot \langle 4,0\rangle \cdot \langle 3,4\rangle \cdot \langle 0,3\rangle \cdot \langle 4,1\rangle \cdot \langle 3,6\rangle \cdot \langle 1,7\rangle \cdot \langle 6,-\rangle \cdot \langle 7,-\rangle$

$S_3 = \mathbf{R}(S_2') = 3 \cdot 1 \cdot 11 \cdot 9 \cdot 1 \cdot 10 \cdot 8 \cdot 0 \cdot 6 \cdot 4 \cdot 0 \cdot 7 \cdot 5 \cdot 2 \cdot 12 \cdot 13$

$S_3' = \langle 3,1\rangle \cdot \langle 1,10\rangle \cdot \langle 11,8\rangle \cdot \langle 9,0\rangle \cdot \langle 1,6\rangle \cdot \langle 10,4\rangle \cdot \langle 8,0\rangle \cdot \langle 0,7\rangle \cdot \langle 6,5\rangle \cdot \langle 4,2\rangle \cdot \langle 0,12\rangle \cdot \langle 7,13\rangle \cdot \langle 5,-\rangle \cdot \langle 2,-\rangle \cdot \langle 12,-\rangle \cdot \langle 13,-\rangle$

$S_4 = \mathbf{R}(S_3') = 5 \cdot 3 \cdot 13 \cdot 11 \cdot 2 \cdot 12 \cdot 10 \cdot 0 \cdot 8 \cdot 6 \cdot 1 \cdot 9 \cdot 7 \cdot 4 \cdot 14 \cdot 15$

$\vdots$

$S_{\log n} = \mathbf{R}(S'_{\log n - 1}) = \cdots$

Total time: O(Sort(Σ) + n log n) = O(n log n)

Fun facts:

- Replace-by-rank preserves lexicographic order
- Sort by longest fingerprints to get the suffix sorting

Suffix sorting: the sorted order of all suffixes of a string

- This will come back later

KMR back in the day

KMR used to be described with a big table

- It's still O(n log n)

What does the big table do for you?

- It lets you compute KMR in parallel

Remember PRAMs?

- They were a wonderful computational model that theoreticians abandoned because they got teased by bullies

So now, the challenge

KMR solves the problem in O(n log n)

Knuth conjectures that it's Ω(n log n)

- Even for binary alphabets

So it's time for Weiner to do his thing

- One year later: O(n) time for finding longest repeated string in a binary string.
- Knuth declares it the "Algorithm of the Year"
- It's not trivial to see it now, but his paper invented the suffix tree

Peter Weiner disappears

Right after he proves his big result:

- Weiner leaves academia
- Starts Interactive Systems Corporation
- Which owned Unix for a while
- So all hackers hated it/him

Peter Weiner Reappears!

I cite Weiner for years, of course and then:

- July 19, 2012, he friends me (or whatever) on LinkedIn

LinkedIn

Peter G Weiner has indicated you are a Colleague at Yale University

Hi Martin,

I haven't worked in algorithms in almost 40 years, and I just came across your 1997 paper on Optimal Suffix Tree Construction with Large Alphabets. I'm going to try to understand the details, but from a higher-level perspective it looks quite interesting.

- Peter G Weiner

Accept View invitation from Peter G Weiner

We talked that day. We emailed that day:

Peter Weiner
Strings verses Trees
To: Martín L. Farach-Colton

July 19, 2012 at 8:08 PM

PW

Hi Martin,

It was a very real pleasure talking to you earlier, and I do hope we can have further contact down the road.

I also look forward to being introduced to the people you mentioned who are doing current work in Bioinformatics. One thing that I think you mentioned -- if I understood you correctly -- is that people are (on the surface) avoiding Trees by using a combination of Strings and Compression algorithms. Did I get that right?

The first presentation of my work was to a graduate school seminar at MIT attended by Vaughn Pratt. Shortly after he wrote up some notes that looked at things from a String point of view. Unfortunately he never published this work -- it's quite interesting. I'll attach a version that is missing some sections at the end. If you have the time, I would be most interested in how you see this work fitting into what has been done in the last almost 40 years. It is possible he anticipated work that was published later, or perhaps there are ideas in his paper that are still worth disseminating?

Best,
Peter

PS Vaughan also looked at the large alphabets.

Improvements and Applications for the Weiner Repetition Finder

Vaughan R. Pratt

May 1973

(Revised October 1973)

(Rerevised March 1975)

Contents

Vaughan Pratt's notes:

1. Introduction

Our objectives are:

(1) to establish the properties of strings responsible for the correctness of Weiner's [1973] string processing algorithm (whose correctness hitherto was established only by appealing to the properties of the data structures used in the algorithm); these properties appear below as Theorems 1, 2 and 3;

(ii) to simplify and clarify the algorithm, both structurally and with respect to the number of cases;

(iii) to show that the running time of Weiner's algorithm can be made independent of the alphabet size on a RAM with unit cost operations and storage preset to zero;

(iv) to describe further applications for the algorithm.

Let Σ^* be the set of all strings over some finite alphabet Σ. A subword of a string $A = a_1a_2...a_n$ is a string that occurs in A as a contiguous substring. Fulfilling objective (iv), we shall give algorithms for each of the following problems; their running time is, in each case, proportional to the total length of the input.

(a) Find the frequency of occurrence of all subwords of a string. (Since there may be up to $\binom{n+1}{2} + 1$ subwords of a string, we must be careful how we represent this information.)

Good news: the whole alphabet thing was thought about from the beginning

Bad news: Had my paper “Optimal Suffix Tree Construction with Large Alphabet” been scooped? By 24 years???

Vaughan Pratt's notes:


```
neighbors(wa) ← neighbors(x);
for  b  in neighbors(x)  do
     wa.b ← x.b
```

The lexical complexity is of value mainly when interpreting each lexical item can be done in a fixed amount of time. By using various $n \times |\Sigma|$ arrays for the objects w.a, w:a and *a:w, such a bound can be achieved provided we may assume all array entries to be initially undefined. Unfortunately this can be very wasteful of space; in fact, to achieve a time independent of $|\Sigma|$, the factor of $|\Sigma|$ has simply crossed over to the space bound! If we had a machine M that did not charge us for undefined storage locations, this objection would vanish. In practice, as far as we know a random access machine can simulate M at no extra cost in space (to within a constant factor) with at best a time overhead of a factor of log(S) where S is the total space, defined and undefined, required for the arrays. If we assume $|\Sigma| \leq n$ (which is true if Σ is restricted to the letters in A) then this amounts to an overhead of a factor of log n in time in order to keep the space requirements independent of $|\Sigma|$ on a random access machine. Any improvement to the techniques for simulating M will lead to a corresponding improvement in Weiner's algorithm.

Best news: No, I was not scooped

Suffix trees:

- Right to left
- Left to right
- Real time
- Simpler
- etc.

If you want to know the history of those results, ask the authors of those papers…

Building a Suffix Tree: The Large Alphabet Edition

Outline the algorithm

Step 1: Recursively sort odd suffixes.

- How? And how is it recursive? A recursive step must sort every suffix! We'll get to that.

Step 2: Sort even suffixes.

- Yikes. We can't afford to do this recursively.
- If we do, then we get $T(n) \geq 2T(n/2) + \Omega(n)$
- And we get an $\Omega(n \log n)$ algorithm.
- So we can only afford linear time for this step.

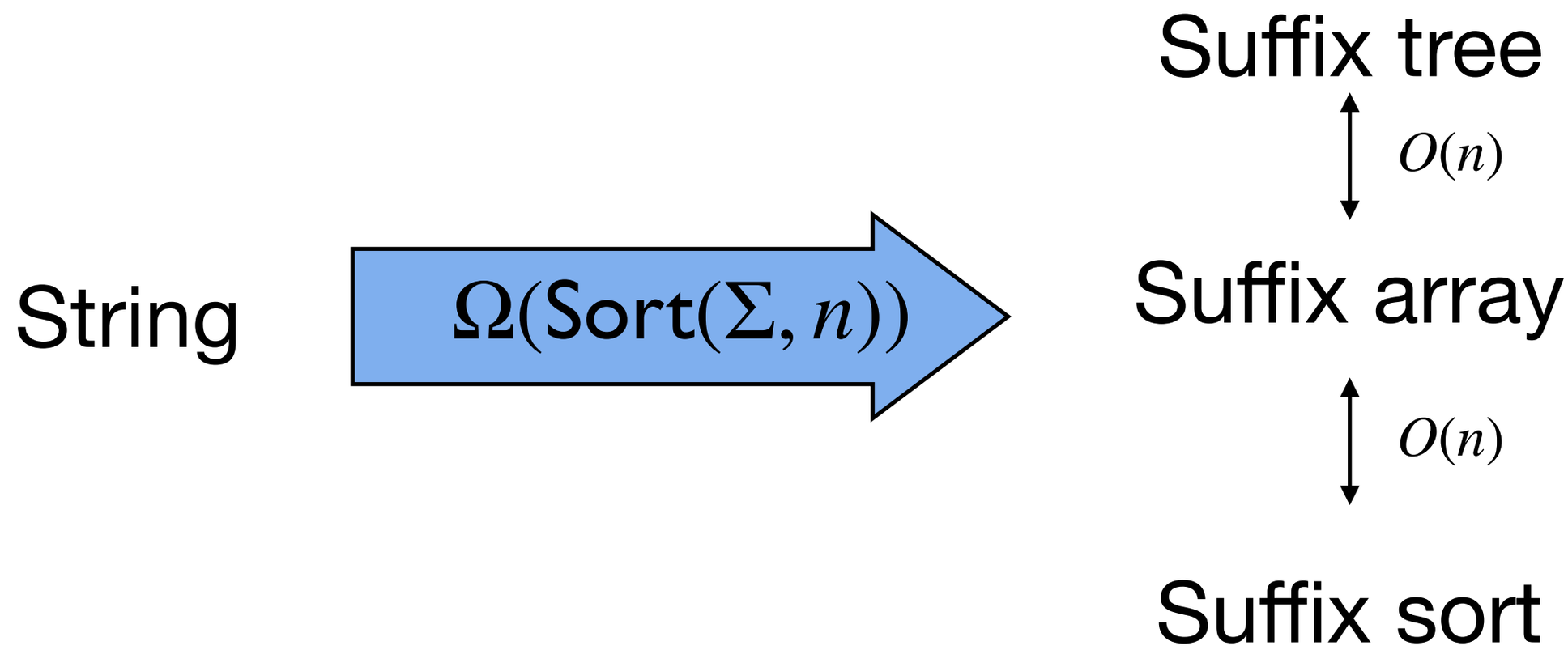
Step 3: Merge!

- How?
- Can only afford linear time for this step.

Before we move on:

A word about Suffix Arrays

Suffix Trees and Succinct Suffix Trees



Suffix Trees and Succinct Suffix Trees

String

$\Omega(\text{Sort}(\Sigma, n))$

Suffix tree

$O(n)$

Suffix array

$O(n)$

Suffix sort

More or less succinct versions of same data structure

Suffix Trees and Succinct Suffix Trees

String

$\Omega(\text{Sort}(\Sigma, n))$

Suffix tree

$O(n)$

Suffix array

$O(n)$

Suffix sort

And fastest suffix array algo is simplification of suffix tree algo

More or less succinct versions of same data structure

Suffix Trees and Succinct Suffix Trees

String

$\Omega(\text{Sort}(\Sigma, n))$

Suffix tree

$O(n)$

Suffix array

$O(n)$

Suffix sort

And fastest suffix array algo is simplification of suffix tree algo

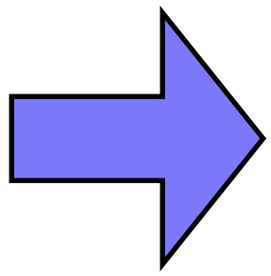
More or less succinct versions of same data structure

Example: Mississippi$

Mississippi$

1 2 3 4 5 6 7 8 9 10 11 12

Mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$
$



ippi$
issippi$
ississippi$
i$
Mississippi$
pi$
ppi$
sippi$
sissippi$
ssippi$
ssissippi$
$

8	5	2	11	1	9	10	6	3	7	4	12

How fast can we sort?

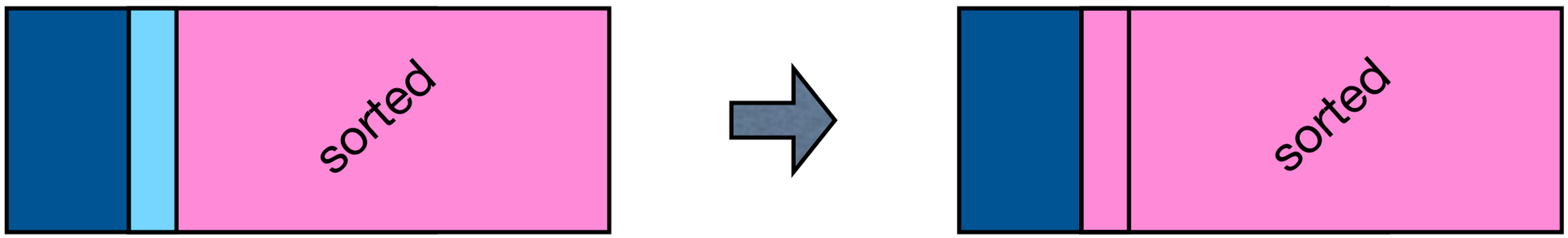
Sorting suffixes is no faster than sorting characters.

This talk: Matching this lower bound

Radix Sort Review

Recall that Radix Sort proceeds in steps:

- Lexicographically sort the last *i* characters of each string.
- Stably sort by preceding character. Now strings are lexicographically sorted by last *i*+1 characters.



It's not just for strings:

- Radix sort means you can sort *n* numbers in range $[n^{O(1)}]$ in O(*n*) time

Main idea: Combine Merge Sort with Radix Sort.

Tools:

- Replace by Rank ✔
- Radix Step ✔
- Chunking ✔

So it's related to KMR

- But we need to figure out how to avoid computing so many fingerprints

Exploring Radix Step

What happens if we sort only some suffixes?

- Say, suffixes 4, 8, and 23?
- Now we do one radix step

What happens we you add one character to the front of a suffix?

- It becomes the previous suffix
- S[3] · suffix(4) = suffix(3)
- S[7] · suffix(8) = suffix(7)
- S[22] · suffix(23) = suffix(22)

- So one radix step of sorted order of suffixes 4, 8 and 23 gives sorted order of suffixes 3, 7 and 22.

Example: sorting odd suffixes of 214414413315

214414413315
14414413315
4414413315
414413315
14413315
4413315
413315
13315
3315
315
15
5

214414413315

4414413315

14413315

413315

3315

15

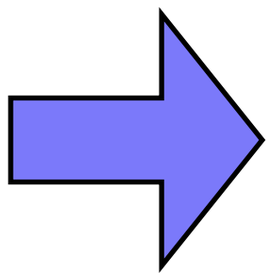
14413315
15
214414413315
3315
413315
4414413315

5	11	1	9	7	3

Odd Suffixes

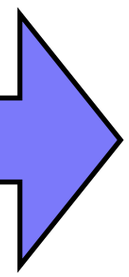
Example: 214414413315

14413315
15
214414413315
3315
413315
4414413315



4 14413315
3 15

1 3315
4 413315
1 4414413315
5



1 3315
1 4414413315
3 15
4 14413315
4 413315
5

8	2	10	6	4	12

Even Suffixes

Where are we on the algorithm?

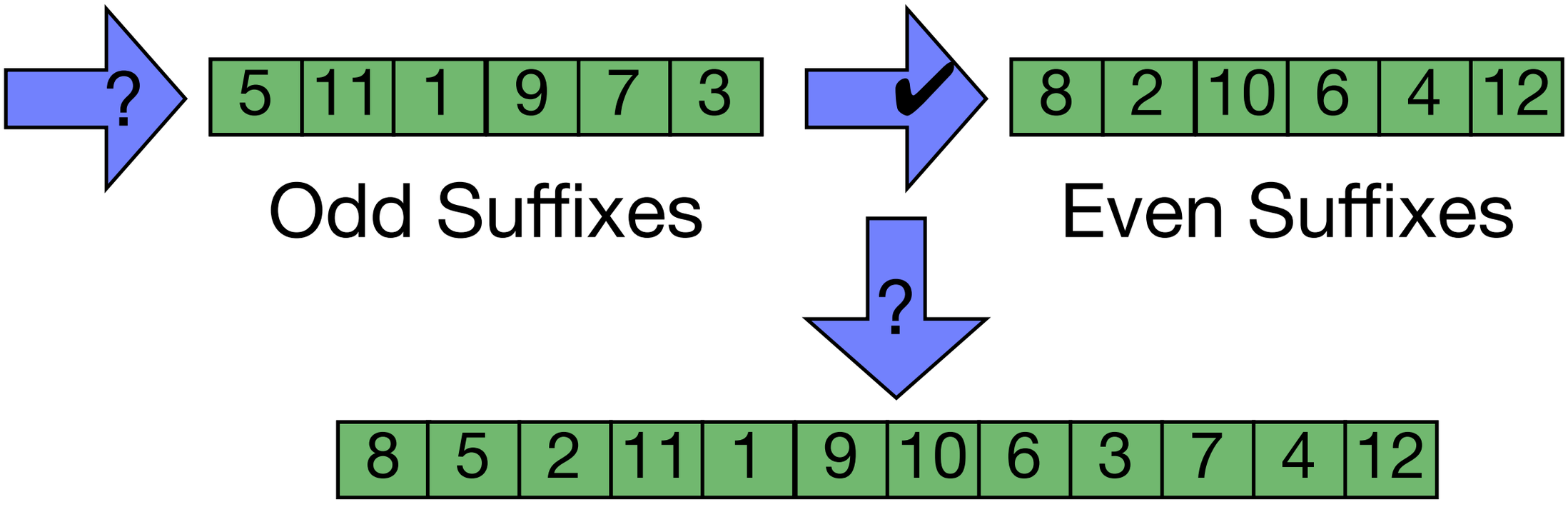
Step 1: Recursively sort odd suffixes.

- How?

Step 2: Sort even suffixes in linear time.

- By Radix Step!

Step 3: Merge!



Observation:

- The order of the odd suffixes of

$$S = (s_1, s_2 \ldots s_n)$$

- is computable from the order of all suffixes of

$$S' = (\langle s_1, s_2 \rangle, \langle s_3, s_4 \rangle, \ldots \langle s_{n-1}, s_n \rangle)$$

- Since chunking preserves lexicographic ordering.

Example: 214414413315

17 36 12 33 27 13
36 12 33 27 13
12 33 27 13
33 27 13
27 13
13

12 33 27 13
13
17 36 12 33 27 13
27 13
33 27 13
36 12 33 27 13

3	6	1	5	4	2

2*x*-1

⟨21⟩⟨44⟩⟨14⟩⟨41⟩⟨33⟩⟨15⟩
⟨44⟩⟨14⟩⟨41⟩⟨33⟩⟨15⟩
⟨14⟩⟨41⟩⟨33⟩⟨15⟩
⟨41⟩⟨33⟩⟨15⟩
⟨33⟩⟨15⟩
⟨15⟩

3 bits per character
treat as base 8

14413315
15
214414413315
3315
413315
4414413315

5	11	1	9	7	3

Chunking + Recursion: II

Chunking+Range Reduction = Recursion

- Input is in $[n]^n$.
- Chunked Input is in $[n^2]^{n/2}$.
- Replace-by-Rank Chunking is in $[n/2]^{n/2}$.
- So now problem instance is half the size and we can recurse.

Example: 214414413315

3 6 1 5 4 2
6 1 5 4 2
1 5 4 2
5 4 2
4 2
2

<14>	1
<15>	2
<21>	3
<33>	4
<41>	5
<44>	6

⟨21⟩⟨44⟩⟨14⟩⟨41⟩⟨33⟩⟨15⟩
⟨44⟩⟨14⟩⟨41⟩⟨33⟩⟨15⟩
⟨14⟩⟨41⟩⟨33⟩⟨15⟩
⟨41⟩⟨33⟩⟨15⟩
⟨33⟩⟨15⟩
⟨15⟩

3 bits per character
treat as base 8

1 5 4 2
2
3 6 1 5 4 2
4 2
5 4 2
6 1 5 4 2

14413315
15
214414413315
3315
413315
4414413315

3	6	1	5	4	2

2*x*-1

5	11	1	9	7	3

Step 1: Chunk + Range Reduction. T(n/2)

- Recurse on new string.
- Get sorted order of odd suffixes.

Step 2: Radix Step. (Not 2nd Recursion!). O(n)

- Get sorted order of even suffixes.

Step 3: Merge!

- We still don’t know how to do this.

The Trouble with Merging

Know how the odd suffixes compare.

Know how the even suffixes compare.

No idea how odd & even compare!

- And comparing them character by character takes O(n)
- For a total of $O(n^2)$

The difference between 3 and 2

It's possible to merge the lists.

- By F '97 "unintuitive" algorithm.

But Kärkkäinen & Sanders showed the elegant way to merge.

- They complicate the recursion
- It's not too bad
- And it make merging easy.

I modified their algorithm to make merging even easier.

Given a string

$$S = (s_1, s_2, \ldots, s_n)$$

Let

$$S_1 = (\langle s_1, s_2, s_3\rangle, \langle s_4, s_5, s_6\rangle \ldots, \langle s_{n-2}, s_{n-1}, s_n\rangle)$$

Let

$$S_2 = (\langle s_2, s_3, s_4\rangle, \langle s_5, s_6, s_7\rangle \ldots, \langle s_{n-1}, s_n, \$\rangle)$$

Let O_{12} be order of suffixes ≡ 1 or 2 (mod 3).

- You get this recursively from sorting the suffixes of S_1S_2

Example: 214414413315

⟨214⟩⟨414⟩⟨413⟩⟨315⟩
⟨144⟩⟨144⟩⟨133⟩⟨155⟩

1 2 3 4 5 6 7 8
⟨214⟩⟨414⟩⟨413⟩⟨315⟩⟨144⟩⟨144⟩⟨133⟩⟨155⟩
1 4 7 10 2 5 8 11

47652213

47652213
7652213
652213
52213
2213
213
13
3

13
213
2213
3
47652213
52213
652213
7652213

3(*x*-4)-1 *x*>4
3*x*-2 o/w

7	6	5	8	1	4	3	2

O_{12}

8	5	2	11	1	10	7	4

8	5	2	11	1	9	10	6	3	7	4	12

Radix Step x 2

We have O_{12} from the recursion.

One Radix Step gives us O_{01}

- Radix stepping a 1 suffix gives a 0 suffix.
- Radix stepping a 2 suffix gives a 1 suffix.

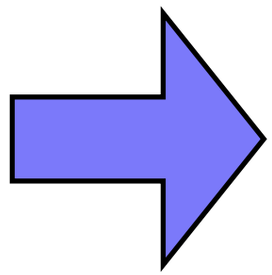
Another Radix Step gives us O_{02}

Each suffix pair is now comparable.

Each suffix appears in two lists.

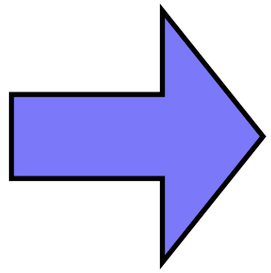
Example: 214414413315

13315
14413315
14414413315
15
214414413315
315
413315
414413315

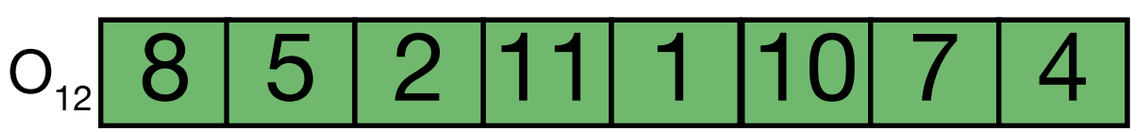


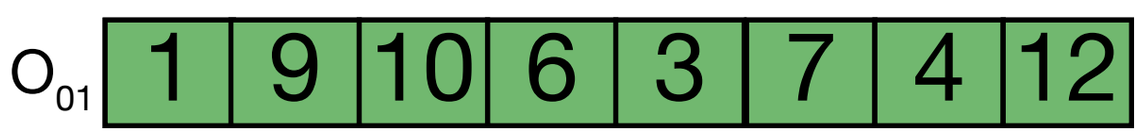
413315
414413315
214414413315
315

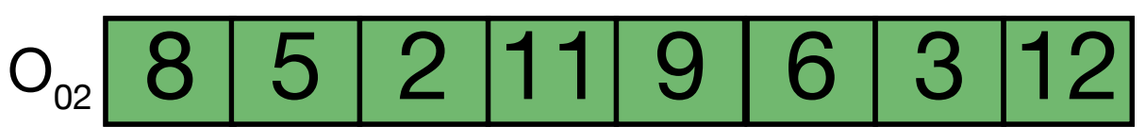
3315
4413315
4414413315
5



214414413315
315
3315
413315
414413315
4413315
4414413315
5







Merging... at last!

An example is worth a thousand words...

Example: 214414413315

O_{12}	8	5	2	11	1	10	7	4
O_{01}	1	9	10	6	3	7	4	12
O_{02}	8	5	2	11	9	6	3	12

Example: 214414413315

O_{12}

8	5	2	11	1	10	7	4

O_{01}

1	9	10	6	3	7	4	12

O_{02}

8	5	2	11	9	6	3	12

8											

Example: 214414413315

O_{12}

5	2	11	1	10	7	4

O_{01}

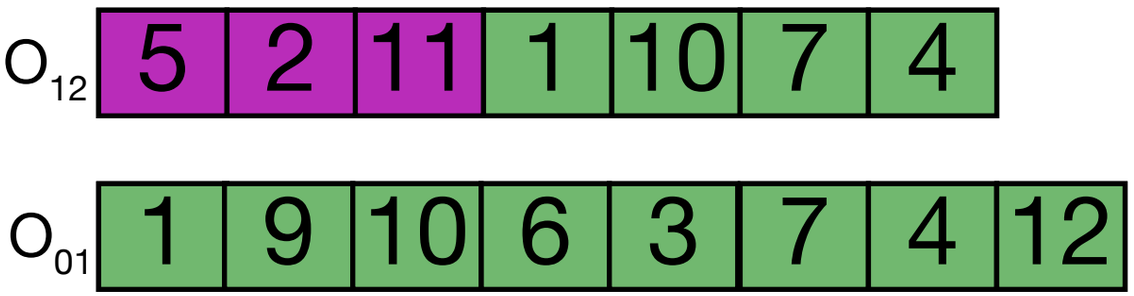
1	9	10	6	3	7	4	12

O_{02}

5	2	11	9	6	3	12

8											

Example: 214414413315



Example: 214414413315

O_{12}

1	10	7	4

O_{01}

1	9	10	6	3	7	4	12

O_{02}

9	6	3	12

8	5	2	11								

Example: 214414413315

O_{12}

1	10	7	4

O_{01}

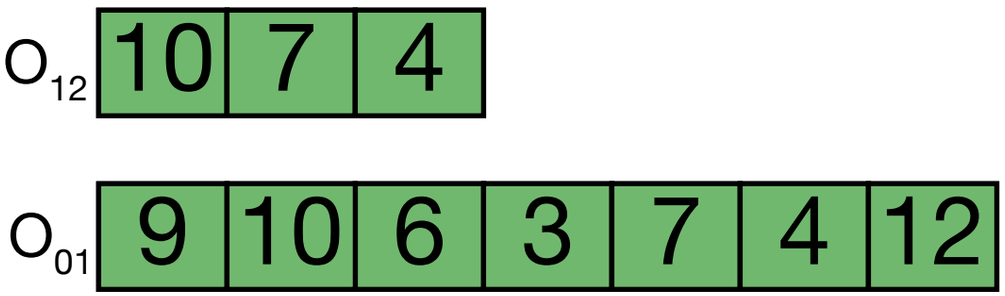
1	9	10	6	3	7	4	12

O_{02}

9	6	3	12

8	5	2	11	1							

Example: 214414413315



etc.

Total time

T(n) to sort suffix of strings in $[n]^n$

T(n) = recursion + 2*radix + merging

T(n) = O(n)+T(2n/3) + O(n) + O(n)

T(n) = O(n)

Total time

The initial Replace by Rank step to get a general string in Σ^n into the integer alphabet -- $[n]^n$ --is the bottleneck.

- So this algorithm is optimal for any alphabet.
- Or is it? More in a minute.

So why did we want to sort suffixes?

Combine two arrays:

- Suffix sorting array
- Array of longest common prefixes of adjacent suffixes

Mississippi$

1 2 3 4 5 6 7 8 9 10 11 12

1	4	1	0	0	1	0	3	1	2	0

8	5	2	11	1	9	10	6	3	7	4	12

This is called a Suffix Array

- Manber & Myers ’90
- It is the most popular succinct version of a suffix tree

Time to go from sorted suffixes to suffix tree

Computing LCPs: O(n)

- Kasai, Lee, Arimura, Arikawa & Park CPM01

Suffix Array to Standard Suffix Tree: O(n)

- Via Cartesian Tree construction
- Vuillemin ‘80

Suffix Tree Optimality

If you are using Suffix Tree as a trie, then each node must be sorted, and the construction is optimal.

If you are using Suffix Tree + LCAs, then the order of children is irrelevant.

- The children of each node can be in any order, and it need not even be consistent between nodes.

Suffix Tree Optimality

Alphabets matter:

- For small integers, construction is already O(n), so this is optimal, even for Scrambled Suffix Trees.
- In algebraic decision tree model, suffix trees have a lower bound from element uniqueness (depends on degree of root) so we have optimal algorithm.
- For large integers (word model of computation), lower bound is linear, upper bound is super-linear.

Close the gap in the time for building a large-alphabet suffix tree, when child order is irrelevant.

Related to Deterministic Hashing Open Problem:

- Given n large integers, can you map them to small integers (poly n) in linear time in the word model?

Today's construction is optimal for sorted suffix trees

What about unsorted suffix trees?

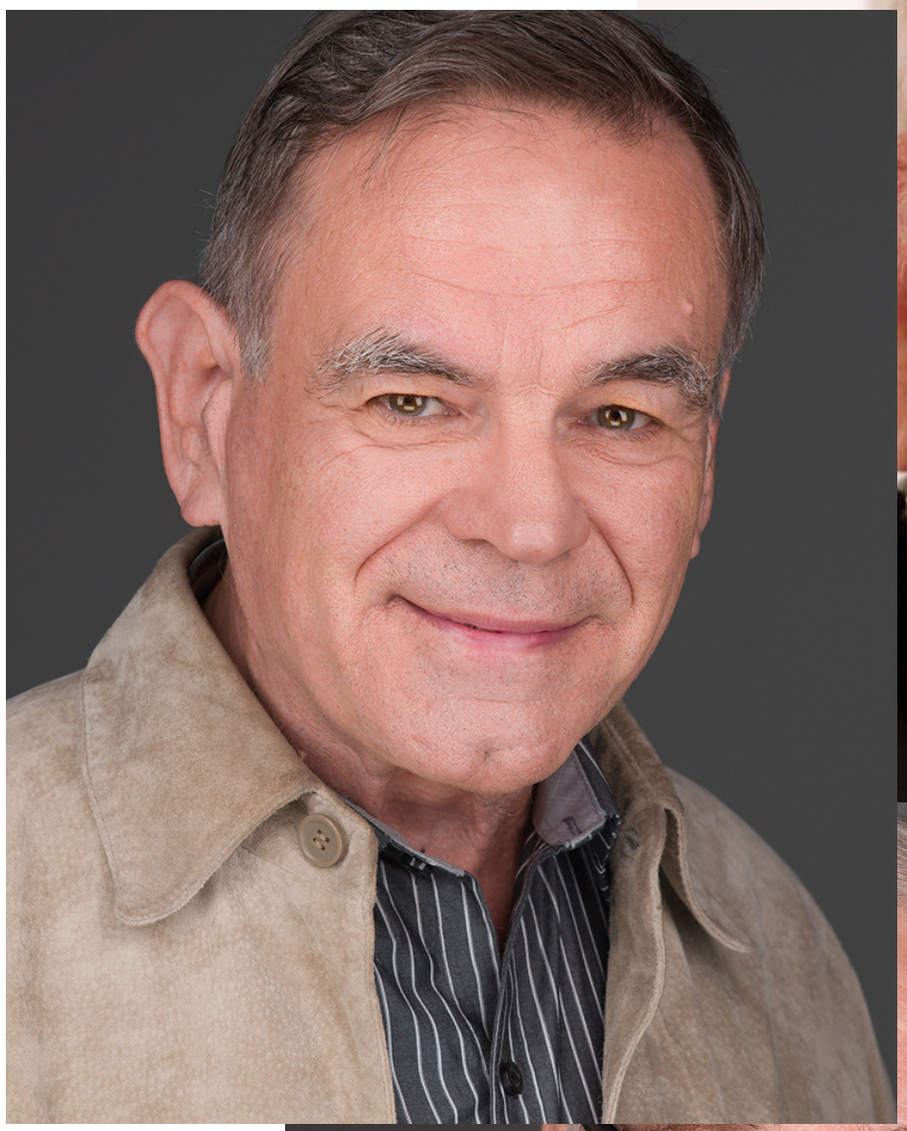
One last thing…

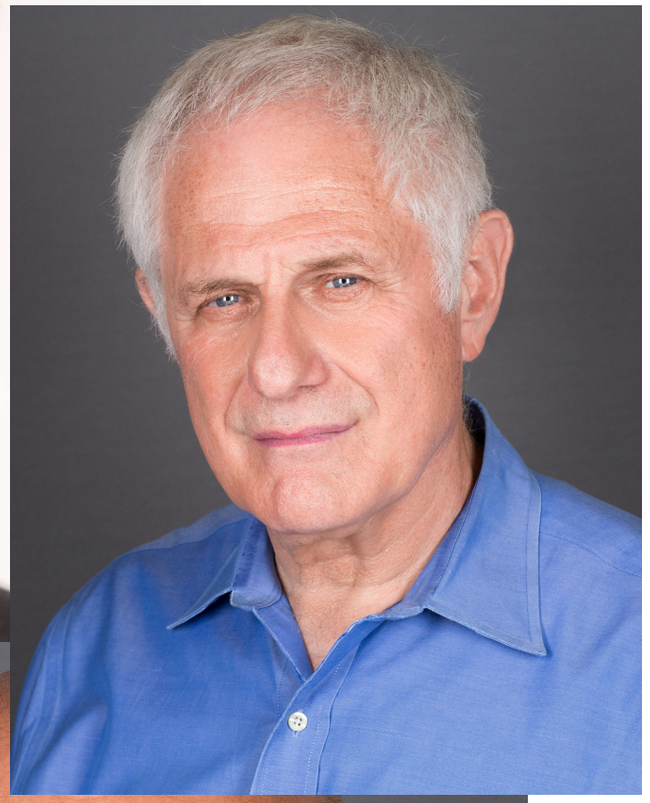
What's Peter Weiner up to now?

He's retired, and has a new career as a head-shot photographer

o to now?

head-





to now?