CS 261: Data Structures

Week 10: Odds and ends

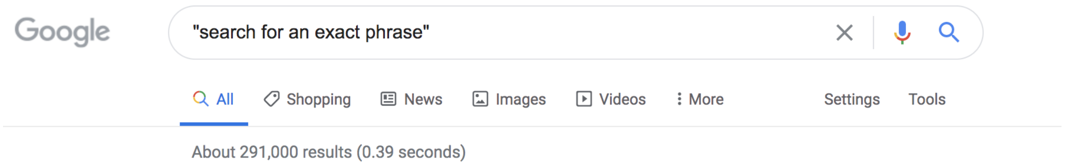
Lecture 10a: Strings

David Eppstein
University of California, Irvine

Spring Quarter, 2025

Background

If you enter a quoted string in Google, it will search for documents containing that phrase


"search for an exact phrase"

About 291,000 results (0.39 seconds)

How does it do it? Not hashing

There are too many different phrases for it to hash all of them

We need a data structure that can

- Index an enormous text document or documents
- Quickly find all copies of a string (appearing without gaps)
- Uses small space, linear in document length

Some basic definitions

Alphabet: any finite set

For text searching, might be ASCII or unicode characters

For genomics, might be the four bases adenine, thymine, cytosine, and guanine (A, T, C, G) that encode genetic information in DNA

For Google's search problem, might be the words (rather than the individual letters) appearing in its documents

Symbol: a member of the given alphabet

String: a finite sequence of symbols

Substring: a connected subsequence of the sequence (might be empty, might be the whole sequence)

End-of-string markers

It is convenient to have a special character that appears
at the end of each string (and nowhere else)

In C programming, this is the null character (byte value zero)

Some older operating systems have used control-D or control-Z

None of these are visible in lecture slides!

Standard convention in string algorithms is to use a dollar sign: $

Suffix arrays

Basic idea of suffix arrays

Each position in a string defines a *suffix*, the string that starts at that position and continues to the end of the string

Sort the suffixes alphabetically and store the sorted array of their starting positions

Then we can look up any substring using binary search in the sorted array

[Gonnet et al. 1992; Manber and Myers 1993]

Example of suffix arrays

String: Mississippi\$

Suffixes: Mississippi\$, ississippi\$, ssissippi\$, sissippi\$, issippi\$, ssippi\$, sippi\$, ippi\$, ppi\$, pi\$, i\$, \$

Sorted: \$, i\$, ippi\$, issippi\$, ississippi\$, Mississippi\$, pi\$, ppi\$, sippi\$, sissippi\$, ssippi\$, ssissippi\$

Suffix array: [11,10,7,4,1,0,9,8,6,3,5,2]

Searching a suffix array (basic version)

Just binary search! Each step is a string comparison

query time $O(\log n) \times \text{length}(q)$

Query string is a substring if its successor in the binary search begins with the query

Example: query sip in Mississippi\$

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

Can also count # occurrences of query (they will all be consecutive in the array)

Searching a suffix array (speeding with common prefixes)

Within binary search, maintain

- Strings at bottom B and top T of current search interval
- Longest common prefix length of query with these two strings (number of characters before first mismatch), LCP(B,Q) and LCP(T,Q)

At each step,

- Choose bottom/top string S with max(LCP(B,Q),LCP(T,Q))
- Look up precomputed LCP of S with middle string M
- If LCP(S,M) $>$ LCP(S,Q), comparison with M is same as with S
- If LCP(S,M) $<$ LCP(S,Q), comparison same as S vs M
- Otherwise (LCPs are equal), start comparing M to Q at that position

Each character of Q used for only a single comparison!

query time $O(\log n) + \text{length}(q)$

Augmented suffix array

With each position in the sorted array, store

- Its position in the input string (as before)
- Its LCP with bottom and top string in the search interval containing it (the information we need for the query speedup)

Total space $O(n)$

Suffix array construction

Basic idea

Divide string into triples of characters, grouped by start position mod 3

- 0 mod 3: Mis, sis, sip, pi$
- 1 mod 3: iss, iss, ipp, i$$
- 2 mod 3: ssi, ssi, ppi

Think of these as three length-$n/3$ strings!

Recurse on groups 1 mod 3 and 2 mod 3 (together!)

Merge group 0

“The skew algorithm”, Kärkkäinen et al. [2006]

From triples of characters to new characters

Use radix sort on the triples of characters appearing in groups 1 mod 3 and 2 mod 3, then relabel by position in sorted order

i$$, ipp, iss, ppi, ssi $\Rightarrow$ A, B, C, D, E

iss, iss, ipp, i$$ $\Rightarrow$ CCBA

ssi, ssi, ppi $\Rightarrow$ EED

The recursive step

At this point we have two length-$n/3$ strings representing groups 1 mod 3 and 2 mod 3:

- iss, iss, ipp, i$$ $\Rightarrow$ CCBA
- ssi, ssi, ppi $\Rightarrow$ EED

Concatenate them together and compute their suffix array recursively!

CCBA$EED $\Rightarrow$ [4,3,2,1,0,7,6,5]

This gives the combined ordering of those suffixes in the original string (plus extra junk at the end of some suffixes that doesn't affect their comparisons)

Sorting the remaining group

Each item in group 0 mod 3 consists of a single character + a suffix in group 1 mod 3

We already know the ordering of the suffixes in group 1 mod 3!

Radix sort the pairs (first character, position of suffix in group 1 mod 3)

Merging the groups

How to compare group 0 mod 3 suffix X with a string Y from the other two groups?

Y is in group 1 mod 3:

- X = (character + suffix in group 1 mod 3)
- Y = (character + suffix in group 2 mod 3)

Y is in group 2 mod 3:

- X = (character + character + suffix in group 2 mod 3)
- Y = (character + character + suffix in group 1 mod 3)

Compare characters then break ties by positions of suffixes
in recursively computed suffix array

Analysis

Time for radix sorting triples: $O(n)$

Time for recursion: $T(2n/3)$

Time to extract sorted order on group 0 mod 3: $O(n)$

Time to merge: $O(n)$

Total: $T(n) = T(2n/3) + O(n) = O(n)$

Tries

A simpler warm-up problem

Given a “dictionary”, a list of n strings over some alphabet, handle queries that either ask whether a query string q belongs to the dictionary, or that look up some extra information associated with the string (its definition)

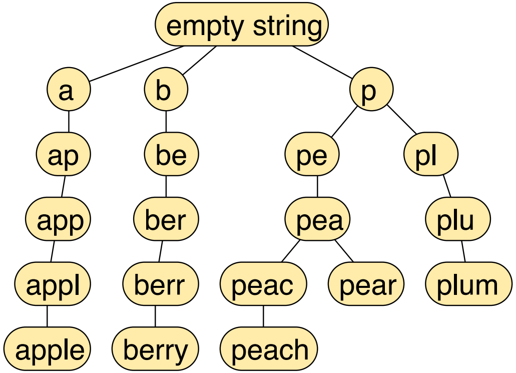
Binary search tree: query time $O(\log n) \times \text{length}(q)$

Hash table: query time $\text{length}(q)$, randomized

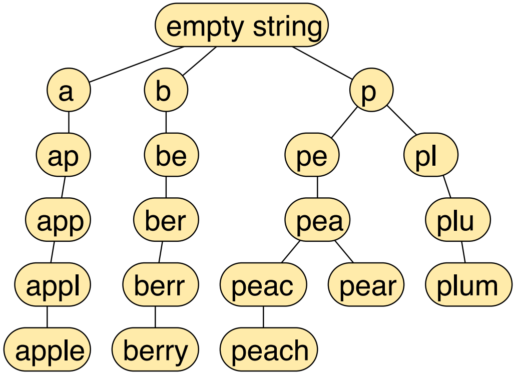
Trie (or digital search tree)

Nodes = prefixes of given strings (substrings that start at the start)

Parent of a node = prefix with one less symbol



Two complications



Leaves are words but not all words are leaves:
Is “pea” a word in this dictionary?

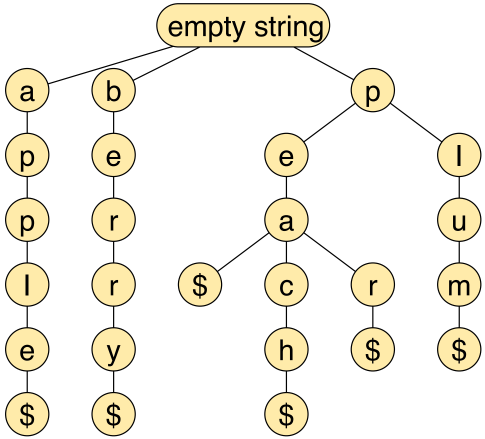
Separately storing each prefix would take too much space

Representing a trie

Add extra end-of-word symbol "$" to all words

Label vertices by single symbol added to parent's prefix (not by whole prefix)

For large alphabets, nodes can store hash table mapping alphabet symbols to corresponding children



Querying a tree

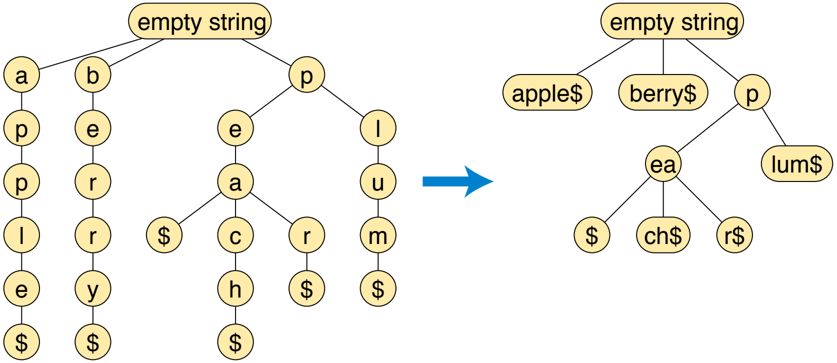
```
Start at root of tree (empty prefix)
For each symbol $c$ of query $q + \$$:
    Go to the child whose label is $c$
    If no such child, $q$ is not in the dictionary
If search succeeded, $q$ is in the dictionary, and current leaf node is its representative in the tree
```

Time = length(q)

Space = number of tree nodes
= total length of all dictionary words

Compressed tries

When nodes have exactly one child, merge them with their child



Tree has n leaves, no nodes with one child $\Rightarrow \leq 2n - 1$ nodes

Represent substrings by pointer to string they come from + position and length $\Rightarrow$ $O(1)$ words of information per node

[Morrison 1968]

Searching a compressed trie

- Current node = root of compressed trie
- Current position in substring = 0
- For each symbol c in query $q + \$$:
    - If current position $<$ length of substring in current node, check that c matches the symbol at that position in the substring, and add 1 to current position
    - Otherwise, look for the child whose substring begins with c (by scanning all children or by using a hash table at the node mapping symbols to children), set current node to that child, and set current position = 0
- If any check fails or any child is not found, q is not in the dictionary; otherwise, it is represented in the dictionary by the current node

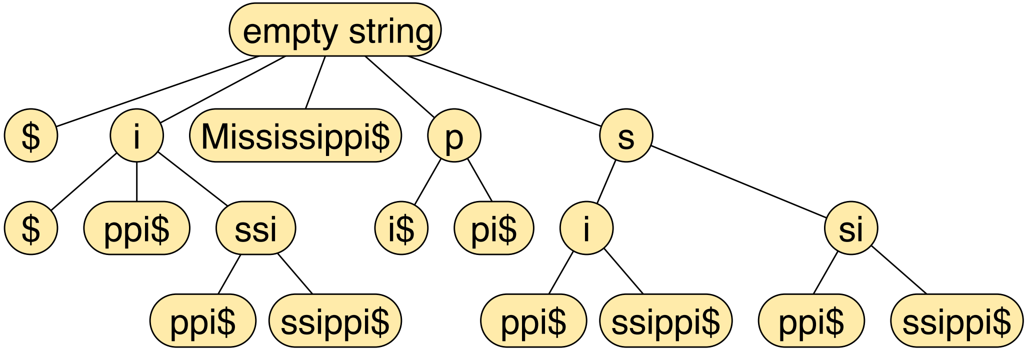
Suffix trees

Suffix trees

Suffix = substring of input string that includes the final dollar sign

Suffix tree = compressed trie of all suffixes [Weiner 1973]

E.g., for input = "Mississippi":



The suffixes in order by length are \$, i\$, pi\$, ppi\$, ippi\$, sippi\$, ssippi\$, issippi\$, sissippi\$, ssissippi\$, ississippi\$, Mississippi\$

Tree leaves represent these same suffixes in alphabetical order

Searching a suffix tree

Given a query string q, search character-by-character, just like in any other compressed trie

But do not search for the \$ after the end of q

Search succeeds $\Rightarrow$ q is a substring of the input string

The positions where q appears in the input are the starting positions of the suffixes whose leaves are descendants of the node where the search finishes

Possible solution to Google's exact phrase search

Build suffix tree where input = all known web pages concatenated together, alphabet = words in each web page

Tree size: total number of words in all known web pages

Why words rather than characters: to reduce suffix tree size

Query time $\approx$ length of query phrase

Search result $\Rightarrow$ positions where q appears in concatenation $\Rightarrow$ web pages containing the given phrase

Other uses of suffix trees

Many other algorithmic problems on strings can be solved using suffix trees, for instance:

To find longest matching substrings from starting positions x and y in given string, find lowest common ancestor of leaves for suffixes starting at x and y

To sort the suffixes of an input (the suffix array! also used in Burrows–Wheeler data compression algorithm), traverse tree using sorted order of characters at each node

Longest common prefix of two suffixes (needed for fast suffix array search precomputation) is their lowest common ancestor

To find longest common substring of two strings s and t, build a suffix tree of $s\$t\$$ and look for a node whose leaf descendants include suffixes starting in both strings and whose distance from the root (measured by string length) is maximum

How to construct the suffix tree?

Linear time for fixed alphabet size

[Weiner 1973; McCreight 1976; Chen and Seiferas 1985; Kosaraju 1994; Kosaraju and Delcher 1995, 1996]

Linear time + time to sort the alphabet

[Farach and Muthukrishnan 1996; Farach 1997]

Farach 1997: divide and conquer, very complicated!

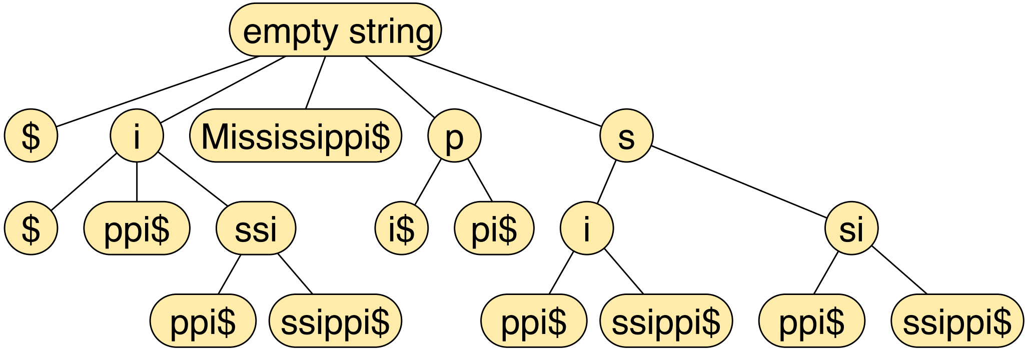
- Divide into positions that are 0 mod 2, 1 mod 2
- Recurse on 1 mod 2 positions
- Use result to build separate tree on 0 mod 2
- Merge the trees

The skew algorithm to the rescue!

Recursively build suffix tree on suffix positions that are 1 mod 3 or 2 mod 3 (size $2n/3$)

Sort all the suffixes (leaves of the full suffix tree)

Build the full suffix tree in left-to-right order (like Cartesian tree construction)



To find where to connect each leaf as we add it, we need its LCP with the previous leaf

Use the same comparison trick as in suffix arrays to turn this into one or two single characters + lowest common ancestor query in the recursive $2n/3$ tree

References, I

M. T. Chen and Joel Seiferas. Efficient and elegant subword-tree construction. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words: Papers from the workshop held in Maratea, June 18–22, 1984*, volume 12 of *NATO Advanced Science Institutes Series F: Computer and Systems Sciences*, pages 97–107. Springer, 1985.

Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19–22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi: 10.1109/SFCS.1997.646102.

Martin Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In Friedhelm Meyer auf der Heide and Burkhard Monien, editors, *Automata, Languages and Programming, 23rd International Colloquium, ICALP96, Paderborn, Germany, 8–12 July 1996, Proceedings*, volume 1099 of *Lecture Notes in Computer Science*, pages 550–561. Springer, 1996. doi: 10.1007/3-540-61440-0_158.

References, II

Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: pat trees and pat arrays. In William B. Frakes and Ricardo A. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice-Hall, 1992.

Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. doi: 10.1145/1217856.1217858.

S. Rao Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In Frank Thomson Leighton and Michael T. Goodrich, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23–25 May 1994, Montréal, Québec, Canada*, pages 310–316. ACM, 1994. doi: 10.1145/195058.195170.

S. Rao Kosaraju and Arthur L. Delcher. Large-scale assembly of DNA strings and space-efficient construction of suffix trees. In Frank Thomson Leighton and Allan Borodin, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 169–177. ACM, 1995. doi: 10.1145/225058.225108.

References, III

S. Rao Kosaraju and Arthur L. Delcher. Large-scale assembly of DNA strings and space-efficient construction of suffix trees (correction). In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22–24, 1996*, page 659. ACM, 1996. doi: 10.1145/237814.250975.

Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi: 10.1137/0222058.

Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi: 10.1145/321941.321946.

Donald R. Morrison. PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968. doi: 10.1145/321479.321481.

Peter Weiner. Linear Pattern Matching Algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15–17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi: 10.1109/SWAT.1973.13.