Exact string searching

Given a text $T$ of length $n$ and pattern $P$ of length $m$
"Quickly" find an occurrence (or all occurrences) of $P$
in $T$

A Naïve solution:
   Compare $P$ with $T[i...i+m]$ for all $i$ — $O(nm)$ time

How about $O(n+m)$ time? (Knuth Morris Pratt)
How about $O(n)$ preprocessing time and $O(m)$ search time?

TRIEs

Dictionary = {at, middle, miss, mist}
TRIEs (searching)

Consider searching a string of length $m$ to see if it is a prefix of an element in the dictionary.
Search time depends on implementation of nodes.

$\textbf{e.g.}, \text{Search}(\text{"mid"})$

---

TRIEs (node implementation)

Consider searching a string of length $m$ to see if it is a prefix of an element in the dictionary.
Consider an $n$-node trie over alphabet $\Sigma$, with $|\Sigma|=k$.

Implementation choices:

- **Array per node:** $O(nk)$ space, $O(m)$ time search
- **Tree per node:** $O(n)$ space, $O(m \log k)$ time search
- **Hash children:** $O(n)$ space, $O(m)$ time

Don’t need a separate table per node — can hash node pointer and child character.

$\text{Table.Lookup}((0x0054, t)) = 0x0932$

---

Compressed Tries

Also called PATRICIA tries/trees or radix trees.
All nodes with a single child are collapsed.
Edges now represent substrings.

Dictionary = \{at, middle, miss, mist\}

Takes less space in practice.

---

Insertion

Inserting string $S$ into a compressed trie:

- Find longest common prefix
- Split edge if necessary
- Add remaining suffix

Insert(“midday”) 

Takes $O(|S|)$ time.
Using Suffixes
Tries allow prefix searching. What about searching for any substring?
Store all the suffixes in a (compressed) trie
S = "nananabana"
⇒ Dictionary = {nanabana, nananabana, nanabanana, anabanana, nabana, abanana, banana, anana, nana, ana, na, a}

Typically use special character ($) at the end of a string to make sure every entry has a leaf.

Suffix Trees
A compressed trie on all suffixes of a string
S = "nananabana$"

Suffix Tree Representation
How do we store a suffix tree in O(n) space?
S = "nananabana$" |S| = n

Suffix Tree Construction
Simple algorithm:
T = empty
for i = 1 to n
insert(S[i..n],T)

Takes O(n^2) time in the worst case
Suffix Tree Construction

$S = \text{nana}\text{banana}$. 

1. Inserting "$\text{nana}\text{banana}$" matches "$\text{nana}$": 4 comparisons.
2. Inserting "$\text{ana}\text{banana}$" will match "$\text{ana}$": 3 comparisons.
3. Then "$\text{banana}\$" and "$\text{abana}\$"...
First Improvement
Consider insert of "xAB" (nanabanana$)
Suppose "xA" (nana) is a prefix in the tree
Then "A" (ana) is a prefix in the tree
⇒ Don't need to match A again — assume it's there.

Suffix Tree Construction
S = nanabanana$
\uparrow \uparrow \downarrow \downarrow
nanabanana$
\begin{array}{c}
\text{nana} \\
\text{banana} \\
\text{nabanana}$
\end{array}
\begin{array}{c}
\text{nana} \\
\text{banana} \\
\text{nabanana}$
\end{array}

inserting "nanabanana$" matches "nana": 4 comparisons

Suffix Tree Construction
S = nanabanana$
\uparrow \uparrow \downarrow \downarrow
nanabanana$
\begin{array}{c}
\text{ana} \\
\text{banana} \\
\text{nabanana}$
\end{array}
\begin{array}{c}
\text{ana} \\
\text{banana} \\
\text{nabanana}$
\end{array}

inserting "ananabanana$", "ana" already matched.
just follow "a..." edge and split : 0 comparisons
Suffix Tree Construction

S = nananabanana$

inserting "nabanana$", "na" already matched.
just follow "n..." edge and split : 0 comparisons

15-853  Page 21

Suffix Tree Construction

S = nananabanana$

inserting "abanana$", "a" already matched.
just follow "a..." edge and split : 0 comparisons

15-853  Page 22

Suffix Tree Construction

S = nananabanana$

15-853  Page 23

Suffix Tree Construction

S = nananabanana$

15-853  Page 24
**Suffix Tree Construction**

$S = \text{nananabanana}$

Uh oh. Yes, "nana" is already there, but it takes extra work to follow intermediate nodes.

**Second Improvement: Suffix Links**

For every internal node "xA", keep a pointer to the node for "A".

Getting from xA to A becomes cheaper — follow suffix links!
Suffix Tree Construction

$ S = \text{nananabanana}$

Follow nearest (parent of new node's) suffix link to get most of the way to “nana”

Page 29

Suffix Tree Construction

$ S = \text{nananabanana}$

After inserting “nana$”, “nana” internal node exists. Connect “anana” to “nana”

Page 30

Suffix Tree Construction

$ S = \text{nananabanana}$

Follow nearest suffix link to get from “nana” to “ana”

Page 31

Suffix Tree Construction

$ S = \text{nananabanana}$

Page 32
**Suffix Tree Construction**

Algorithm:
1. Search for $S[i..j-1]$, incrementing $j$ until no match, i.e., $S[i..j-1]$ in tree but not $S[i..j]$.
2. If search is in middle of an edge:
   - then split edge at $S[i..j-1]$ and add new child of $S[i..j-1]$ with suffix $S[j..n]$.
3. Use parent's suffix link to find $S[i+1..j-1]$ and split edge here if not already split.
4. If split edge in (2), add suffix link from $S[i..j-1]$ to $S[i+1..j-1]$.

**Almost Correct Analysis**

Each increment of $j$ takes $O(1)$ time
- Just search one more character.
Each iteration of $i$ takes $O(1)$ additional time
- Just following suffix link (or starting from root) to get from $S[i..j-1]$ to $S[i+1..j-1]$.

⇒ total time is $O(n)$ because $i$ and $j$ are each incremented $n$ times.

What's wrong with this argument?

**Following Suffix Links**

Suppose $S[i..j-1]$ node is not a new node, then suffix link from $S[i..j-1]$ to $S[i+1..j-1]$ already exists, so find $S[i+1..j-1]$ in $O(1)$ time.
Following Suffix Links
Suppose \( S[i..j-1] \) is a new node (i.e., resulting from an edge split)

1. Go to parent node \( S[i.k] \), for some \( i \leq k < j-1 \)
2. Follow suffix link to \( S[i+1..k] \)
3. Search down to \( S[i+1..j-1] \)

   **not guaranteed to be** \( O(1) \)

This node has a suffix link!
Thus, \( k \) must increase in next iteration!

Better Analysis
\[ S = \]

\[ \text{current} \] \[ \text{matched} \] \[ \text{nearest} \] \[ \text{sufffix} \] \[ \text{prefix} \] \[ \text{sufffix link} \] (\( k \) not maintained by algorithm)

\( j \): Each increment of \( j \) takes \( O(1) \) time
\( k \): If searching from \( S[i+1..k] \) to \( S[i+1..j-1] \) follows \( r \) edges, then \( k \) increases by at least \( r \)
\( i \): Each iteration of \( i \) takes \( O(1) \) additional time

\( \Rightarrow \) total time is \( O(n) \) because \( i, j, \) and \( k \) are each incremented at most \( n \) times.

Extending to multiple lists
Suppose we want to match a pattern with a dictionary of \( m \) texts with a total length of \( n \)

Concatenate all texts (interspersed with special characters) and construct a common suffix tree
(Optional) truncate tree below special characters
Time : \( O(n+m) = O(n) \)
or
Construct suffix tree on first text, then insert suffixes of second text and so on.
Time: also \( O(n) \)

Longest Common Substring
Find the longest string that is a substring of both \( S_1 \) and \( S_2 \)

Construct a common suffix tree for both texts
All leaves should be labeled with \( S_1 \) and/or \( S_2 \)
Any node with a descendent labeled \( S_1 \) and \( S_2 \) is a common substring
The "deepest" such node is the longest common substring
Takes linear time with a tree traversal
Common substrings of m texts

Given m texts of total length n
Given some value 1 \leq k \leq m, find the longest string
that is a substring of at least k texts

Construct a common suffix tree, labeling each leaf
with the text it comes from
For every internal node, find the number of
distinctly labeled descendents: O(m) time per
node.
Can report for all k with a single tree traversal
time: O(mn) — not linear