The Indexing Problem:

Preprocess: Text $T$.

Enable queries of the form:

Query: Given pattern $P$,

find all occurrences of $P$ in $T$ in time $O(|P| + \text{tocc})$,

where $\text{tocc}$ is the number of occurrences.

Possible solutions: Using tries.
Definition:

Let \( S_1 = A_{11} A_{12} \ldots A_{1n_1} \$ \)
\( S_2 = A_{21} A_{22} \ldots A_{2n_2} \$ \)
\[ \vdots \]
\( S_k = A_{k1} A_{k2} \ldots A_{kn_k} \$ \)

where \( \$ \in \Sigma \) is an eol symbol.

A trie is a labeled tree where:

1. Its root is the null string \( \Lambda \).
2. Has \( k \) leaves, each labeled by \( \$ \).
3. The \( k \) paths from the root to the leaves are labeled by \( S_1, \ldots, S_k \), respectively.
EXAMPLE:

The Strings:

A B A$

A B C A$

A A B C$

A A C$

A B A C$

B A$

The Trie

A

B

A

C

B

C

$
USES:

Fast retrieval.

Given a string $p_1 \ldots p_m$ we can find, in time $O(m)$ whether it is in the set $\{S_1, \ldots, S_k\}$.

In Previous Example: $ABC$ not in but $ABAC$ is in.

Issue: Alphabet size.

Time is $O(m)$ for fixed alphabet.

$O(m \log \min(|\Sigma|, k))$ otherwise.

(binary search at every node)
Let $T = t_1 \ldots t_n \$ \$ be a string. Consider all $n+1$ suffixes of $\$:

$$
\$ \\
t_n \$
\t_{n-1} t_n \\
t_{n-2} t_{n-1} t_n \\
\ldots
\t_1 t_2 \ldots \ t_n \$

Construct a trie of the suffixes.
EXAMPLE:

\[ T = ABABBA$ \]

Suffixes:
- $ A$
- $ BA$
- $ BBA$
- $ ABBA$
- $ BABBA$
- $ ABABBA$
- $ ABAABA$
- $ ABABBA$
- $ ABAABBA$

Trie:

Queries:
- $A$ 3
- $BA$ 2
- $AB$ 2
- $AA$ 0
Strategy for Indexing Problem:

**PREPROCESSING:** Construct trie of suffixes of $T$.

**Query:** Given $P = p_1 \ldots p_m$.

Run down trie from root, according to elements of $P$.

If stuck: no occurrence.
If path from root to node $w$ is equal to $P$:
all leaves in subtree rooted at $w$ are occurrences.

**Time:** $O(m + tocc)$
PROBLEM: Size of trie: $O(m^2)$
This is awful for large texts!!

EXAMPLE: $A_1, A_2, \ldots, A_n$

where $A_i \neq A_j$
for $i \neq j$

The trie:

FINITE ALPHABET EXAMPLE?

exercise.
Theorem:

Let $T$ be a tree with $n$ leaves where every node is either a leaf or has at least two children. Then $T$ has at most $2n$ nodes.

Proof:

Easy by induction, for binary trees. For higher degree trees situation only improves (i.e. even less nodes). (e.g. change $\star$ to $\bigtriangleup$.)
In our trie of suffixes: \[ n+1 \text{ leaves.} \]

So if we contract chains of the form \[ \cdots \]
we will get a tree of size \( O(n) \)

**EXAMPLE:**

![Diagram of a trie with contracted chains](image)
VERY FUNNY... we still need to write on the edges the text, for comparison purposes, and that is still $O(m^2)$ even though the tree has $O(n)$ nodes.

NOT SO. $O(n)$ words are sufficient!

write on every edge a left pointer and a right pointer to its substring's location in the text.

EXAMPLE:

Location: 1 2 ... n n+1
Text: A A$_2$ ... A$_n$ $\$
There exist many different algorithms for constructing suffix tree (compacted trie of suffixes) of text of length \( n \) in time \( O(m) \).

We will see Weiner's Algorithm (1973).

At this point

Indexing Problem is solved.
ANOTHER IMPORTANT TOOL...

LOWEST COMMON ANCESTOR (LCA)

Preprocess: Tree $T = (V, E)$

To enable following queries.

Query: Given nodes $a, b \in V$

Find $x \in V$ such that $x$ is the lowest common ancestor of $a$ and $b$.

EXAMPLE:

![Tree Diagram]

LCA$(18, 13) = 8$
LCA$(17, 11) = 3$
LCA$(9, 10) = 4$
LCA$(6, 7) = 1$
Harel & Tarjan (1983):

It is possible to preprocess an n-node tree in time $O(m)$ and answer subsequent LCA queries in time $O(1)$.

Landau (1984) made the following key observation:

In Suffix tree,

$LCA(a,b) =$ Longest Common Prefix of substrings ending in $a,b$ resp.
EXAMPLE: \( T = ABABBA$ \)

\[ \text{LCA}(1, 2) = 3 \]

longest common prefix of

\( ABABBA$ \) and \( ABBA$ \) is \( AB \)

\[ \text{LCA}(6, 9) = 10 \]

longest common prefix of

\( BABBA$ \) and \( BBA$ \) is \( B \)

\[ \text{LCA}(5, 9) = \Lambda \]

longest common prefix of

\( A$ \) and \( BBA$ \) is \( \Lambda \)
PUT TOGETHER SUFFIX TREES & LCA.

Get alternate string matching algorithm

INPUT: \( T, P \)

OUTPUT: All locations \( i \) in \( T \) where an occurrence of \( P \) starts.

Algorithm

Preprocessing:

1. Construct suffix tree for \( T \$_1, P \$_2 \), \( \$_1, \$_2 \in \Sigma \).

2. Preprocess suffix tree for LCA.

3. For each node \( v \) in suffix tree write \( l(v) \), the length of substring from root to \( v \).
Text Scanning:

for $i = 1$ to $n-m+1$ do

$x \leftarrow l(LCA(T_i, P))$

where $T_i$ is the suffix $t_i t_{i+1} \ldots t_{n-1} P S_2$

and $P$ is the suffix $P S_2$.

If $x \geq m$ then there is a match at $i$.

else, no match.

end Algorithm

Time:

Preprocessing: 1. $O((n+m) \log \min(|\Sigma|, m))$

2. $O(n)$

3. $O(n)$ (via DFS, for example)

Scanning: $O(n)$.
Example: \( T = \text{ABABBA}$, \\
\( P = \text{AB}$

In red: node's length.

\[
\ell(\text{LCA}(T_1, P)) = 2 \\
\ell(\text{LCA}(T_2, P)) = 0 \\
\ell(\text{LCA}(T_3, P)) = 2 \\
\ell(\text{LCA}(T_4, P)) = 0 \\
\ell(\text{LCA}(T_5, P)) = 0
\]
Alternate Algorithm to Bird-Baker

Do KMP algorithm going down column $j$ with following change:

- whenever KMP compares $t_{ij}$ with $P_k$,
  - compare $t_{ij}$, $t_{ij+1}$, ..., $t_{ij+m-1}$
  - with $P_{k1}$, $P_{k2}$, ..., $P_{km}$.

If comparison time $O(f(m))$ then algorithm time $O(m^2 f(m))$. 
Use suffix trees and LCA to make such comparisons in time $O(1)$

Let $P_1, P_2, \ldots, P_m$ be the rows of $P$,
$T_1, T_2, \ldots, T_n$ be the rows of $T$,
$s_1, \ldots, s_{m+1} \in \Sigma$.

Preprocessing:
1. Construct suffix tree of $T_1, T_2, \ldots, T_n, s_1, P_1, s_2, P_2, \ldots, P_m, s_{m+1}$
2. Preprocess for LCA
3. Write $l(v)$ — length of substring from root to $v$ — for each node $v$. 


Subroutine \( \text{COMPARE} \left( t_{ij}, P_k \right) \)

\[
\text{If } \ell(LCA(T_{ij}, P_k)) \geq m \\
\text{then return equal} \\
\text{else return not equal}
\]

\[\text{end}\]

Where \( T_{ij} \) is the suffix starting at \( t_{ij} \) and \( P_k \) is the suffix starting at \( P_k \).
OPEN PROBLEM Posed by Galil (1985)

Input: $T, P, k$

Output: All locations $i$ in $T$ where $P$ appears with at most $k$ mismatches.

Can this problem be solved in time $O(nk)$?

Solution: (Landau 1986) Of course!

Suffix trees & LCA.

For each location $i$ in $T$:

\[ l_1 = \text{length of longest common prefix of } T_i \text{ and } P \]
\[ l_2 = \text{length of longest common prefix of } T_{i+l_1} \text{ and } P_{i+l_1} \]
\[ l_3 = \text{length of longest common prefix of } T_{i+l_1+l_2} \text{ and } P_{i+l_1+l_2+3} \]

\[ \ldots \]
EDIT DISTANCE

In addition to mismatches, Levenshtein (1966) identified 2 more edit operations - insertions and deletions.

Text: A C D E F G H I
Pattern: A B C D E F G H

One deletion error rather than 7 mismatches.

Text: A I B C D E F G H
Pattern: A B C D E F G H

One insertion error rather than 7 mismatches.
PATTERN MATCHING WITH ERRORS

INPUT: Text $T = t_1 \ldots t_n$
Pattern $P = p_1 \ldots p_m$.

OUTPUT: For every location $i$, the minimum number of edit operations required to make $P$ match a suffix of $t_1 t_2 \ldots t_i$.

NOTE: We usually considered errors starting at $i$, not ending. However, it is the same, just reverse text and pattern.

IDEA FOR ALGORITHM:
Dynamic Programming.
For each $(i, j)$ let $\text{edit}(i, j)$ be the minimum edit distance of $p_i \ldots p_j$ ending at $t_j$.

Cases:

<table>
<thead>
<tr>
<th></th>
<th>$t_j$</th>
<th>$t_{j+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>$p_{i+1}$</td>
<td></td>
</tr>
</tbody>
</table>

If $t_{j+1} = p_{i+1}$, then

$$\text{edit}(i+1, j+1) = \text{edit}(i, j)$$

If $t_{j+1} \neq p_{i+1}$, then

$$\text{edit}(i+1, j+1) = \min(\text{edit}(i, j) + 1, \text{edit}(i, j+1) + 1, \text{edit}(i+1, j) + 1)$$

- **Mismatch**
- **Deletion**
- **Insertion**
**Algorithm**

If $p_i = t_j$ then $E[i,j] \leftarrow E[i-1,j-1]$
else $E[i,j] \leftarrow 1 + \min(E[i+1,j], E[i,j-1], E[i,j-1])$

**Time:** $O(n^2)$. 
**EXAMPLE:**  
Text:  **ABBABB**  
**Pattern:**  **ABAB**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Can we solve Galil's Open Problem for Edit Distance?

Idea: (Landau & Vishkin 1986)

Consider dynamic programming matrix $D$:

![Dynamic Programming Matrix](image)

But now we can advance on diagonals as long as pattern = text.

Advance until number $\geq k$ (no match) or till last row (match).
IMPLEMENTATION:

Define: diagonal $d$ as all $D[i, j]$ where $j - i = d$.

Let $d_{r, e} = \text{highest row in diagonal } d \text{ where the number (of errors) is } e$. 
### Example:

![Diagram of a grid with marked cells and a legend indicating the number of errors and the diagonal.](image-url)
Algorithm (Dynamic Programming)

Given $k$ errors.

Initialize:

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>k-1</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-k$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-(k-1)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$-3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-2$</td>
<td></td>
<td>$1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-1$</td>
<td></td>
<td>$0$</td>
<td>$1$</td>
<td>$2$</td>
<td>$k-1$</td>
<td>$k$</td>
</tr>
<tr>
<td></td>
<td>$0$</td>
<td>$-1$</td>
<td></td>
<td></td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td></td>
<td></td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$(n-m)$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
</tbody>
</table>

Fill columns left to right, top to bottom as follows:
compute \( L_{d,e} \):

start with \( r \leftarrow \max( \) \( L_{d,e-1} + 1 \) at least one more than row of \( e-1 \) errors at \( d\text{-th diagonal} \)

\( L_{d+1,e-1} + 1 \) at least one more than highest row of \( e-1 \) errors at the diagonal above \( (d+1) \)

\( L_{d-1,e-1} \) at least same row as the highest row of \( e-1 \) errors at the diagonal below \( (d-1) \) \)
But now, extend as long as pattern equals text, i.e.

\[ P_{r+1} = t_{r+d+1} \]
\[ P_{r+2} = t_{r+d+2} \]

\[ \vdots \]

End: Any row of \( L \) where \( m \) is reached is an occurrence.

Time: Above extension can be done in constant time by suffix trees and LCA.

Total Time: Fill table of size \( 2n \times k \). Constant time per field:

\( O(nk) \)