The Indexing Problem:

Preprocess: Text T.

Enable queries of the form:

Query: Given pattern P, find all occurrences of P in T in time $O(|P| + tocc)$.

where $tocc$ is the number of occurrences.

Possible solutions: Using tries.

Definition:

Let $S_1 = s_{11}\ s_{12} \cdots s_{1n_1}\ \$$

$S_2 = s_{21}\ s_{22} \cdots s_{2n_2}\ \$$

$\vdots$

$S_k = s_{k1}\ s_{k2} \cdots s_{kn_k}\ \$$

where $\$ \notin \Sigma$ is an eol symbol.

A trie is a labeled tree where:

1. Its root is the null string Λ.
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 $

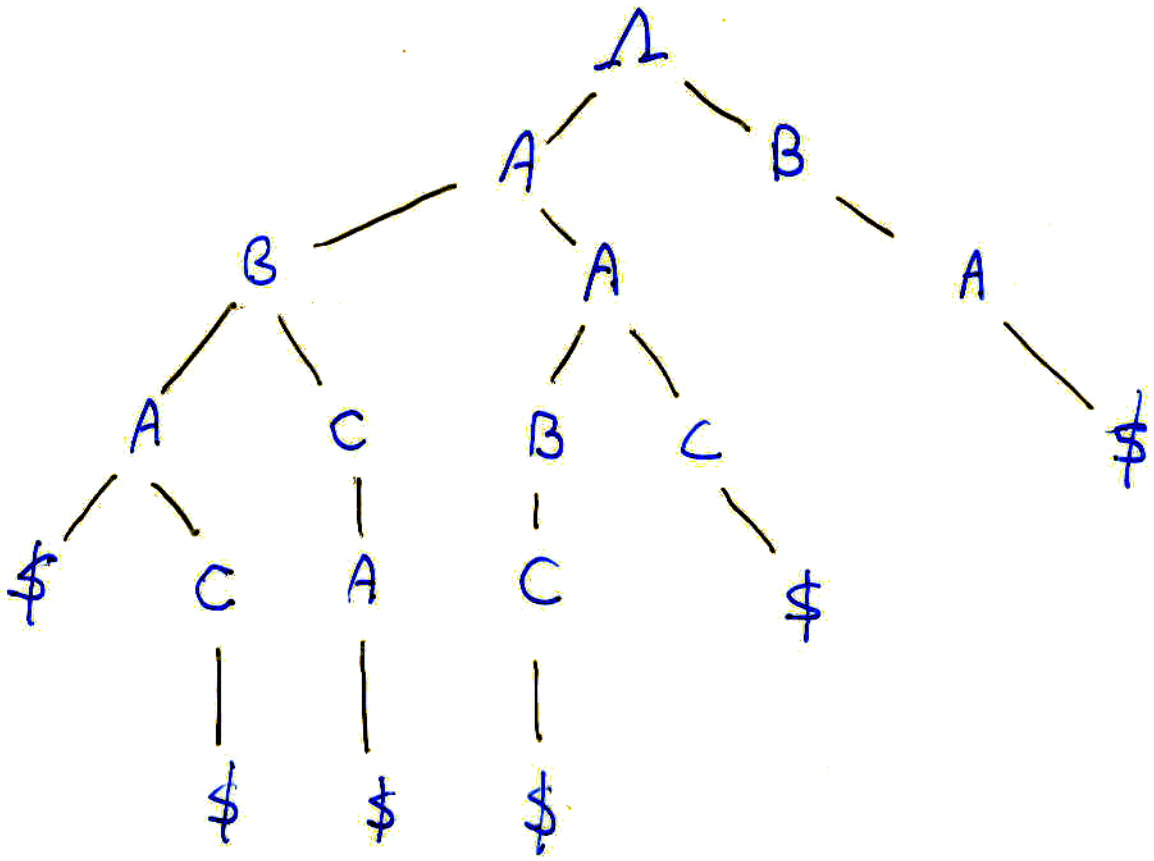
A A B C $

A A C $

A B A C $

B A $

The trie



USES:

Fast retrieval.

Given a string $p_1 \cdots p_m$ 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. (table at every node).

$O(m \log \min(|\Sigma|, k))$ otherwise. (binary search at every node).

SUFFIX TREES

Let $T = t_1 \dots t_n \$$ be a string.

Consider all $n+1$ suffixes of T.

$$\$$$

$$t_n \$$$

$$t_{n-1} t_n \$$$

$$t_{n-2} t_{n-1} t_n \$$$

$$\vdots$$

$$t_1 t_2 \dots t_n \$$$

Construct a trie of the suffixes.

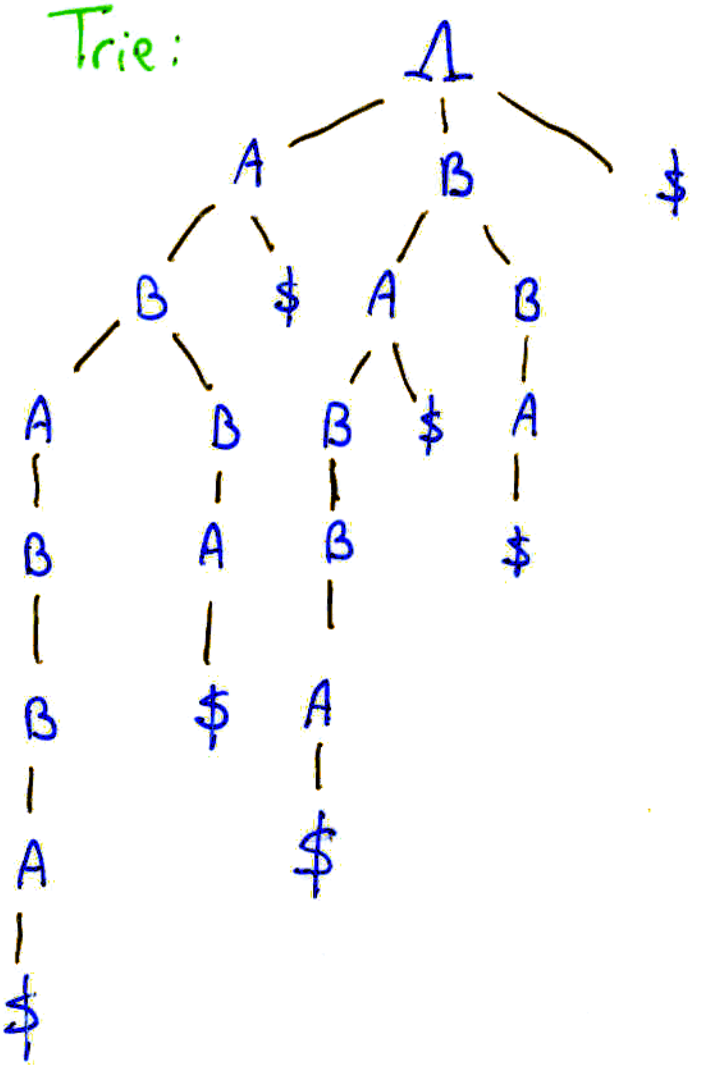
EXAMPLE:

T = ABABBA$

Suffixes:

$
A$
BA$
BBA$
ABBA$
BABBA$
ABABBA$

Trie:



Queries:

	tocc
A	3
BA	2
AB	2
AA	0

Strategy for Indexing Problem:

PREPROCESSING: Construct trie of suffixes of T.

Query: Given $P = p_1 \dots p_m$.

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

If stuck: no occurrence.

If path from root to node ω is equal to P: all leaves in subtree rooted at ω are occurrences.

Time: $O(m + tocc)$

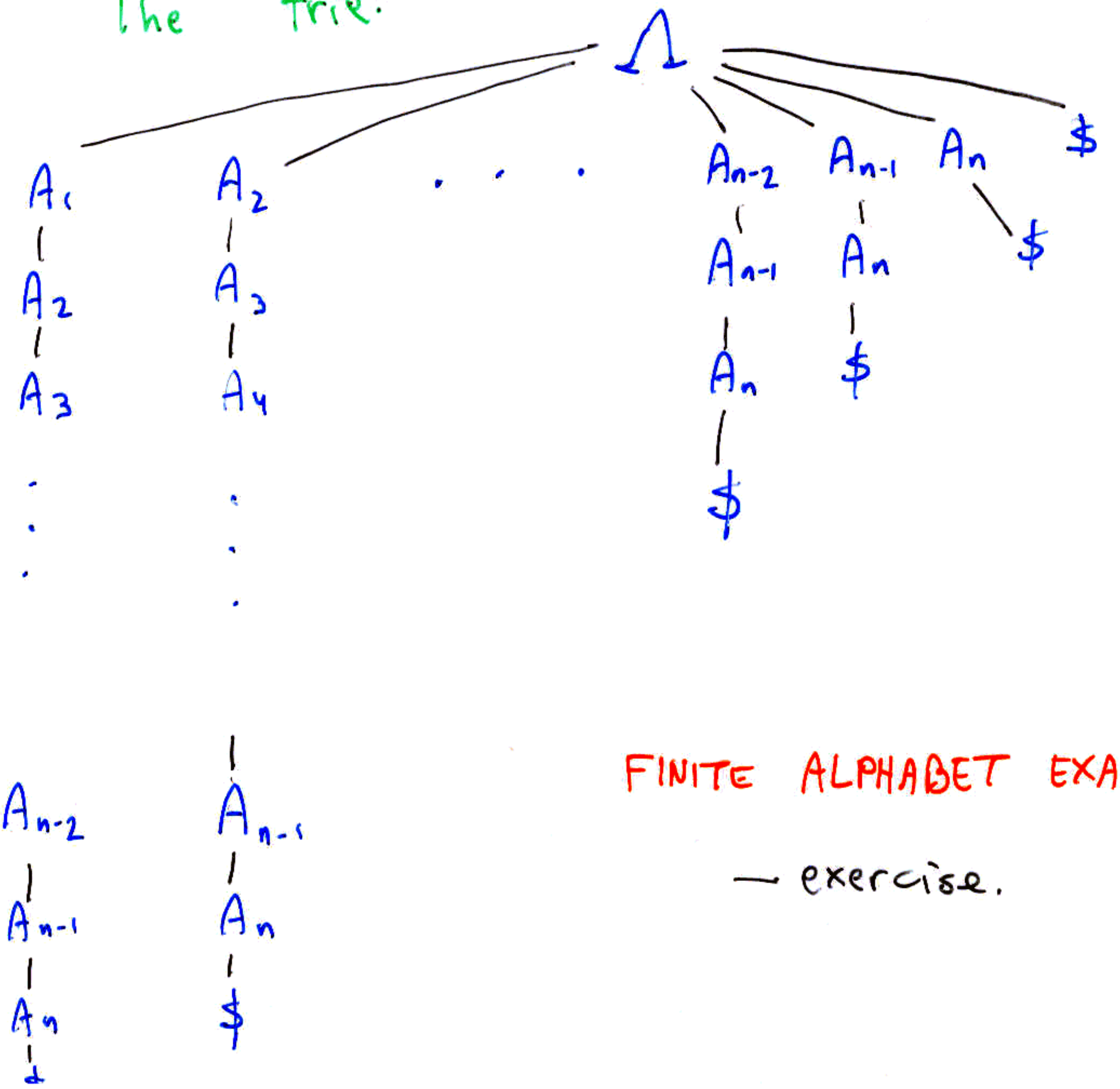
PROBLEM: Size of trie: $O(n^2)$

· This is awful for large texts!!!

EXAMPLE: $A_1 A_2 \dots A_n \$$

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

The trie:



FINITE ALPHABET EXAMPLE?

— exercise.

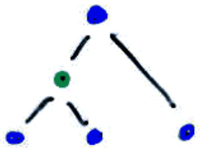
Theorem:

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  to ).

∎

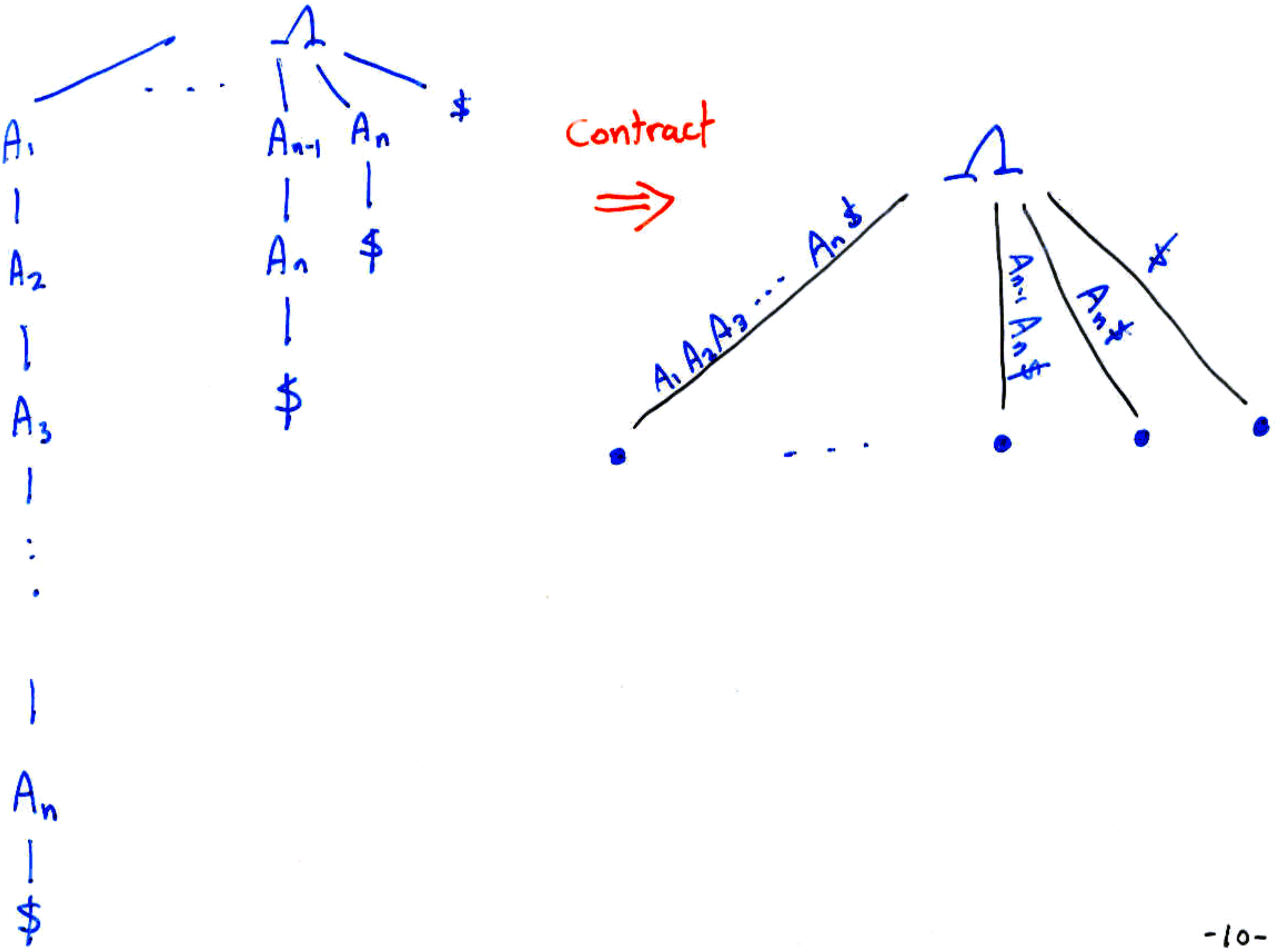
In our trie of suffixes:

n+1 leaves.

So if we contract chains of the form ●—●—●—●— … —●

we will get a tree of size $O(n)$

EXAMPLE:



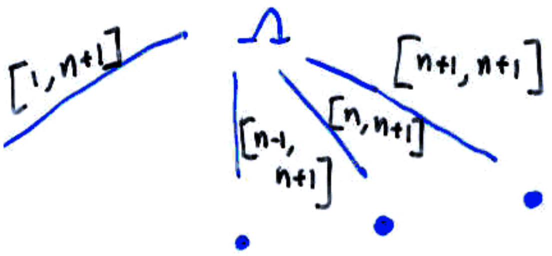
VERY FUNNY ... we still need to write on the edges the text, for comparison purposes, and that is still $O(n^2)$ even though the tree has $O(n)$ nodes.

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: $1 \quad 2 \quad \dots \quad n \quad n+1$
Text: $A_1 \, A_2 \, \dots \, A_n \, \$$



There exist many different algorithms for constructing suffix tree (compacted trie of suffixes) of text of length n in time $O(n)$.

We will see Weiner's Algorithm (1973).

At this point

Indexing Problem is solved.

ANOTHER IMPORTANT TOOL...

LOWEST COMMON ANCESTOR (LCA)

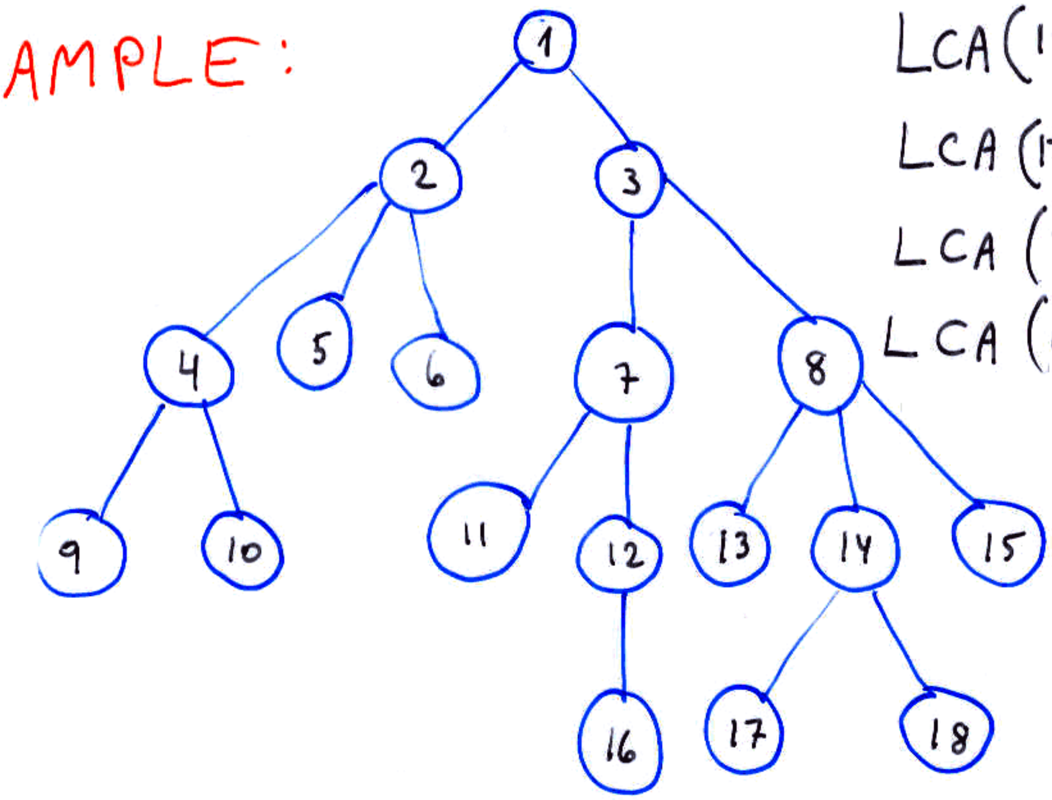
Preprocess: Tree $T=(V,E)$

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:



LCA(18, 13) = 8

LCA(17, 11) = 3

LCA(9, 10) = 4

LCA(6, 7) = 1

Harel & Tarjan (1983):

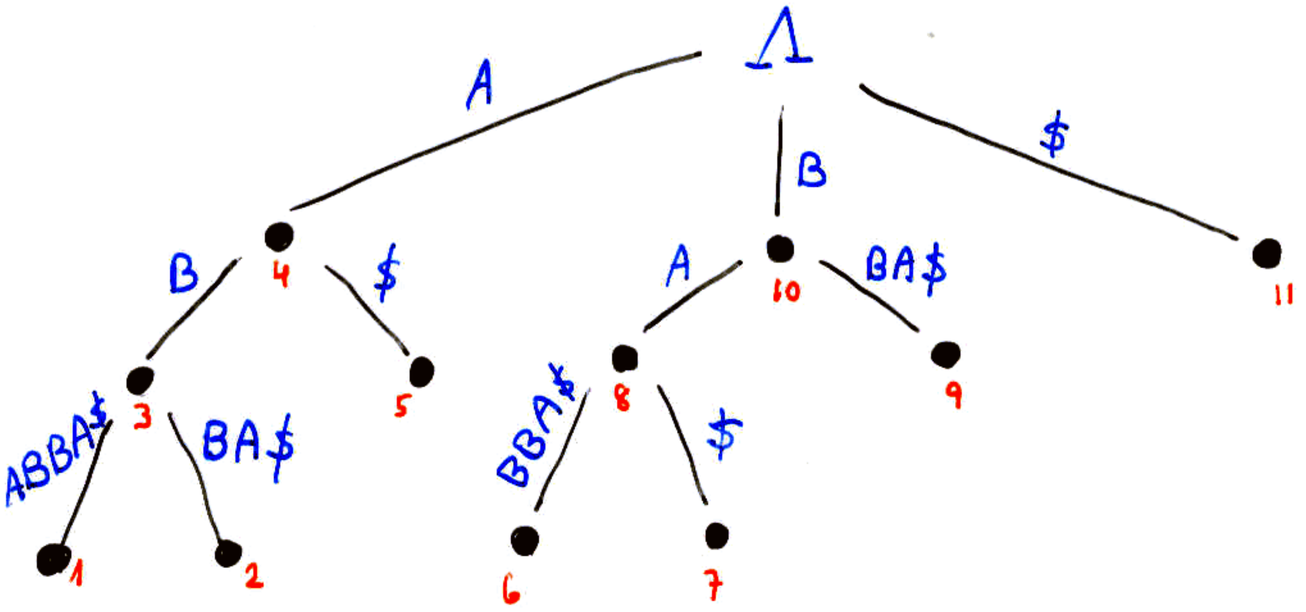
It is possible to preprocess an n-node tree in time $O(n)$ and answer subsequent LCA queries in time $O(1)$.

Landau (1984) made the following key observation:

In Suffix tree,

$LCA(a,b)$ = Longest Common Prefix of substrings ending in a, b resp.

EXAMPLE: $T = ABABBA\$$



$LCA(1,2) = 3$

longest common prefix of

ABABBA$ and ABBA$ is AB

$LCA(6,9) = 10$

longest common prefix of

BABBA$ and BBA$ is B

$LCA(5,9) = \Lambda$

longest common prefix of

A$ and BBA$ is Λ

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 $T\$_1 P\$_2$.
   $\$_1, \$_2 \notin \Sigma$.
2. Preprocess suffix tree for LCA.
3. For each node v in suffix tree write $\ell(v)$, the length of substring from root to v.

Text Scanning:

```
for i=1 to n-m+1 do
    $x \leftarrow \ell(LCA(T_i, P))$
        where $T_i$ is the suffix
            $t_i t_{i+1} \ldots t_n \$_1 P \$_2$
        and $P$ is the suffix $P \$_2$.
    If $x \geq m$ then there is a
                match at $i$
        else, no match.
end Algorithm
```

Time:

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

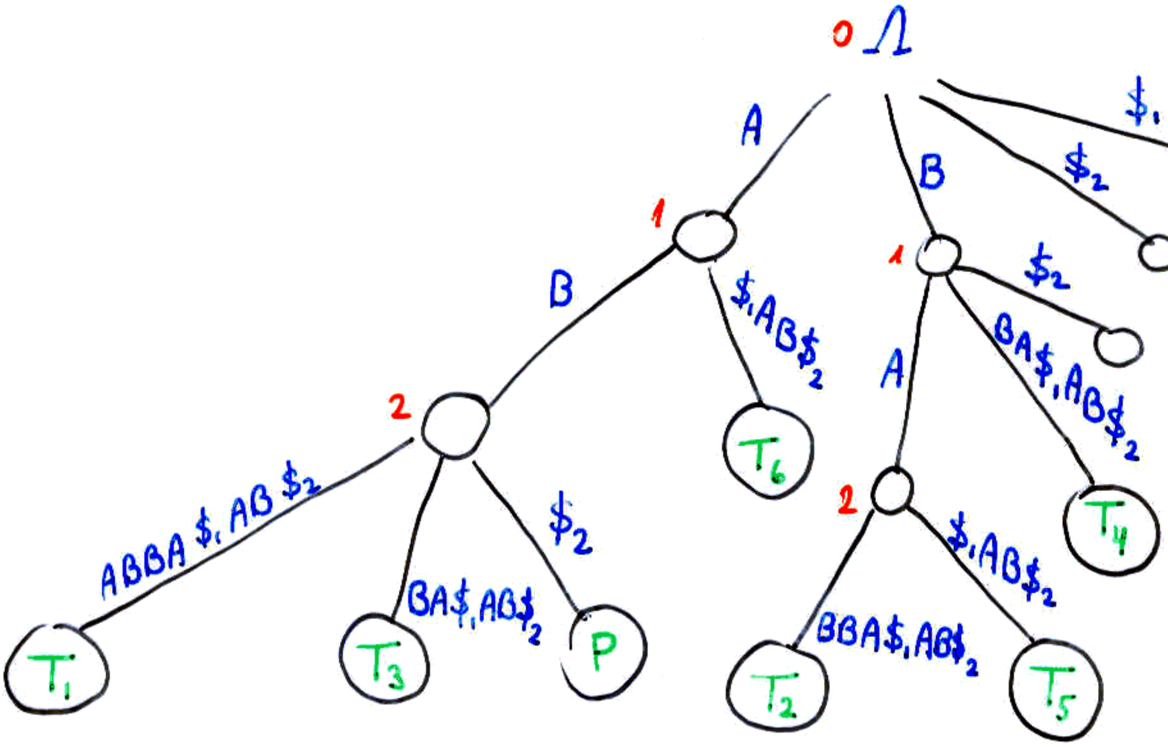
2. $O(n)$

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

Scanning: $O(n)$.

EXAMPLE: $T = \overset{1}{A}\overset{2}{B}\overset{3}{A}\overset{4}{B}\overset{5}{B}\overset{6}{A}\$_1$

$P = AB\$_2$



In red: node's length.

$$\ell(LCA(T_1, P)) = 2$$

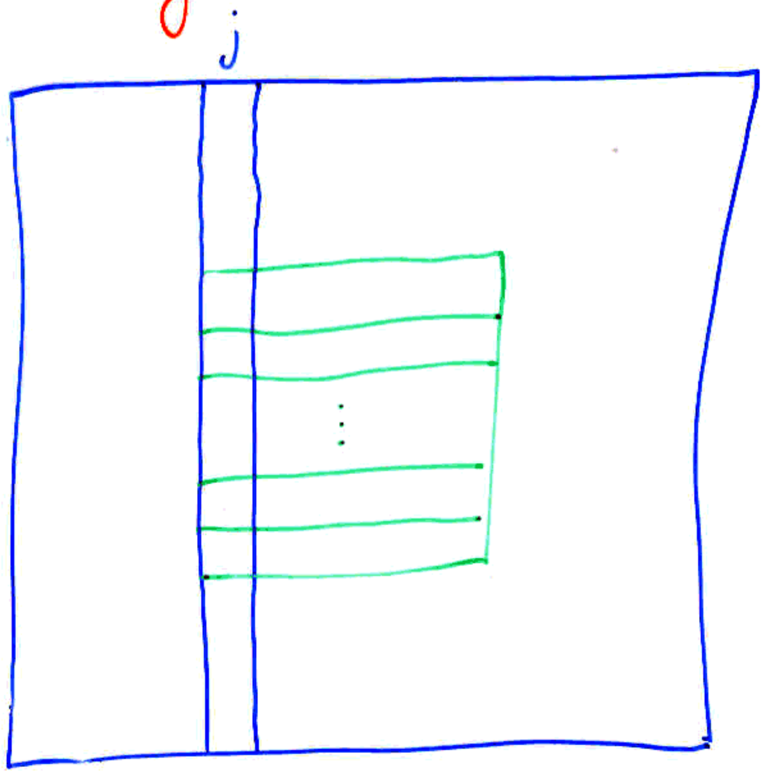
$$\ell(LCA(T_2, P)) = 0$$

$$\ell(LCA(T_3, P)) = 2$$

$$\ell(LCA(T_4, P)) = 0$$

$$\ell(LCA(T_5, P)) = 0$$

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} $\cdots$ t_{ij+m-1}
with p_{k1} p_{k2} $\cdots$ p_{km}.

If comparison time $O(f(m))$ then algorithm time $O(n^2 f(m))$.

Use suffix trees and LCA to make such comparisons in time $O(1)$

Let $P_1, P_2, \dots, P_m$ be the rows of P,

$T_1, T_2, \dots, T_n$ be the rows of T,

$\$_1, \dots, \$_{m+1} \notin \Sigma$.

Preprocessing:

1. Construct suffix tree of

$$T_1 T_2 \dots T_n \$_1 P_1 \$_2 P_2 \dots P_m \$_{m+1}$$

2. Preprocess for LCA

3. Write $\ell(v)$ - length of substring from root to v - for each node v.

```
Subroutine COMPARE ( $t_{ij}$ , $p_k$ )

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

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)

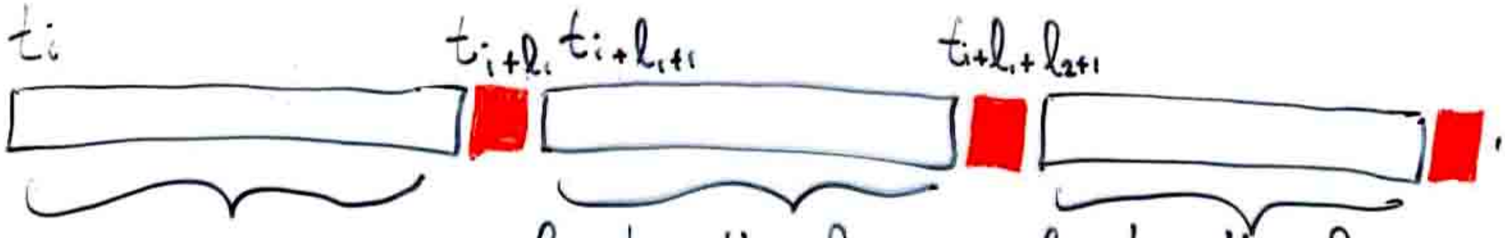
Input: T, P, k

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 = length of longest common prefix of T_i & P

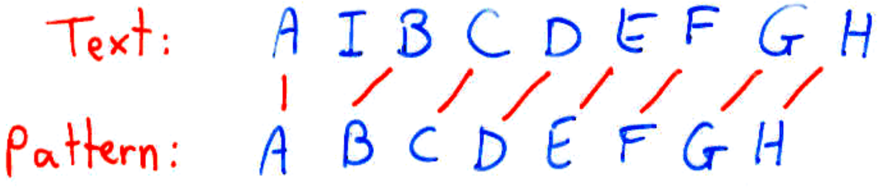
l_2 = length of longest common prefix of T_{i+l_1+1} & P_{l_1+2} ...

l_3 = length of longest common prefix of $T_{i+l_1+l_2+2}$ and $P_{l_1+l_2+3}$ ...

EDIT DISTANCE

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

One deletion error rather than 7 mismatches.



One insertion error rather than 7 mismatches.

PATTERN MATCHING WITH ERRORS

INPUT: Text $T = t_1 \cdots t_n$

Pattern $P = p_1 \cdots p_m$.

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

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 $\mathrm{edit}(i,j)$ be the minimum edit distance of $p_1 \cdots p_i$ ending at t_j.

Cases:

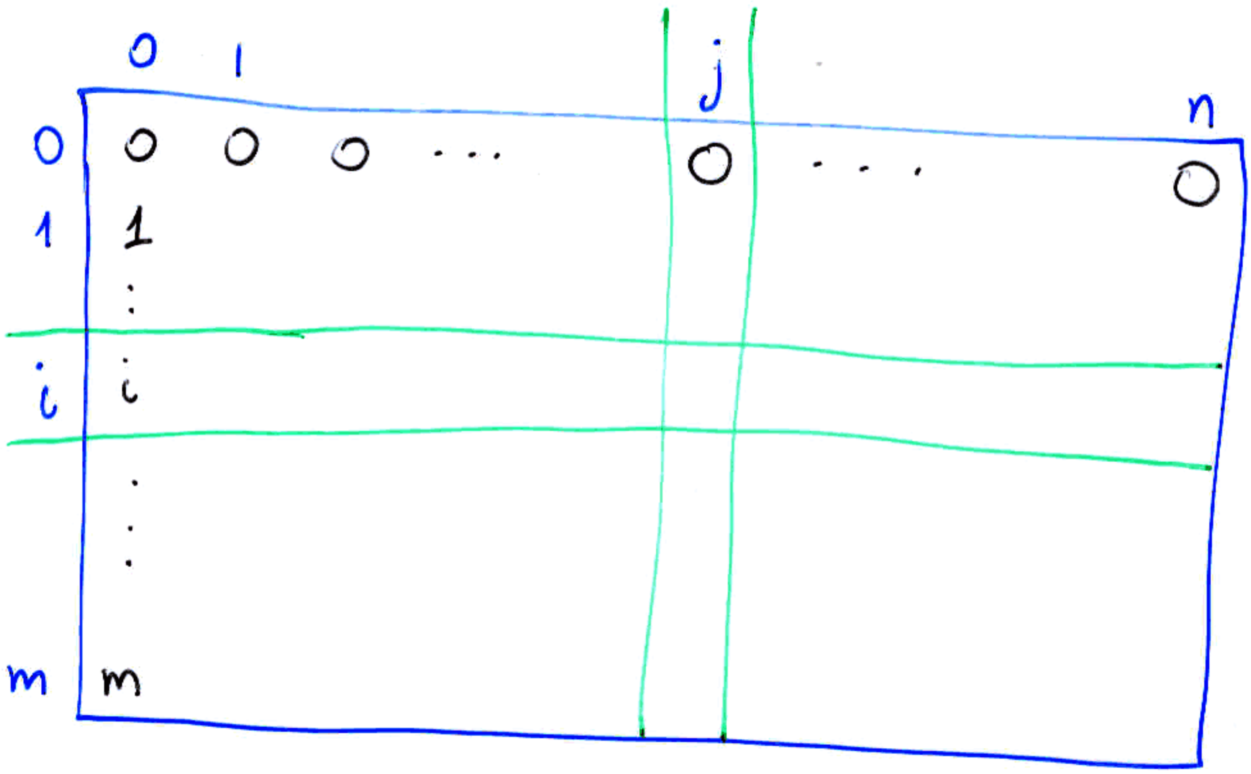
If $t_{j+1} = p_{i+1}$ then

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

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

$$\mathrm{edit}(i+1,j+1) = \min\left(\begin{array}{ll} \mathrm{edit}(i,j)+1 & \text{mismatch} \\ \mathrm{edit}(i,j+1)+1 & \text{deletion} \\ \mathrm{edit}(i+1,j)+1 & \text{insertion} \end{array}\right)$$

ALGORITHM

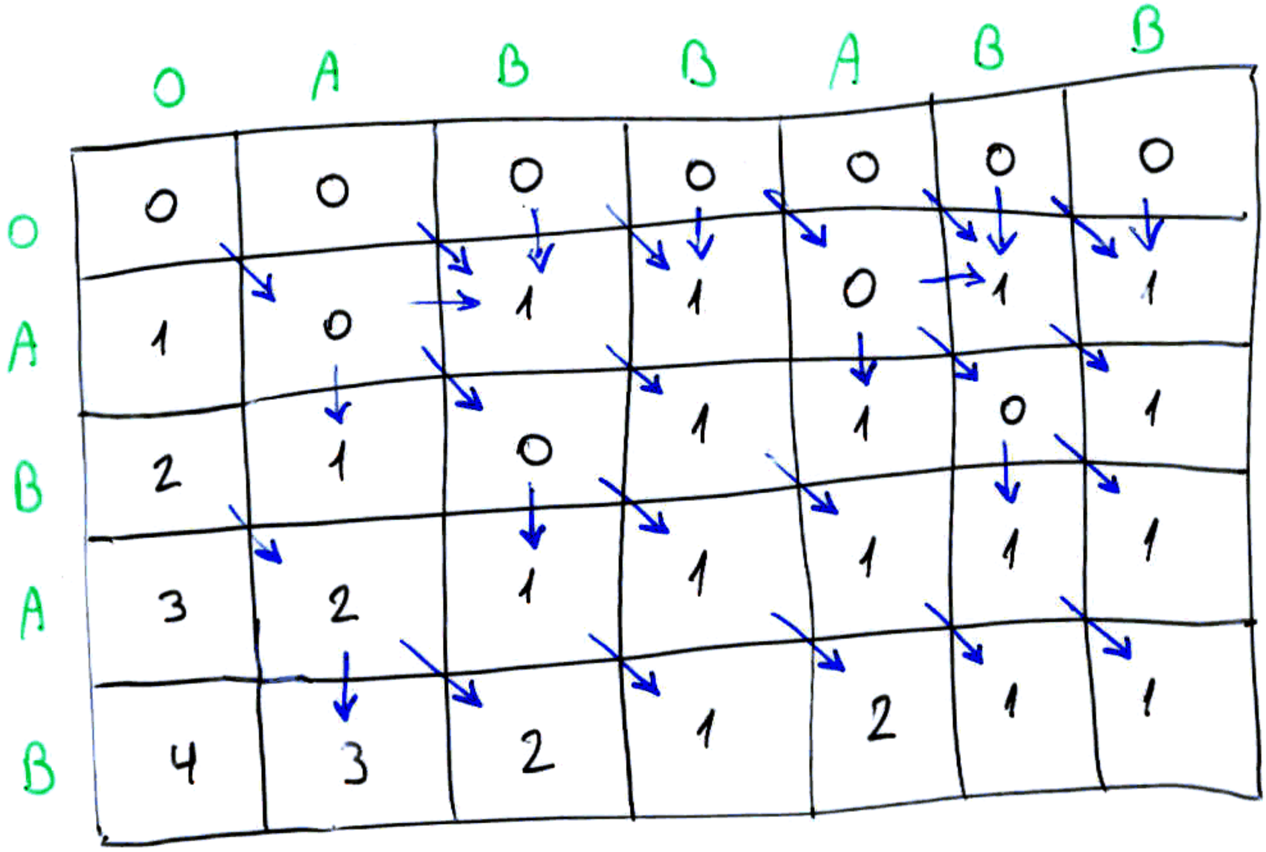


```
If $p_i = t_j$ then $E[i,j] \leftarrow E[i-1,j-1]$
        else $E[i,j] \leftarrow 1 +$
             $\min(E[i-1,j-1], E[i-1,j], E[i,j-1])$
```

Time: $O(nm)$.

EXAMPLE: Text: A B B A B B

Pattern: A B A B.

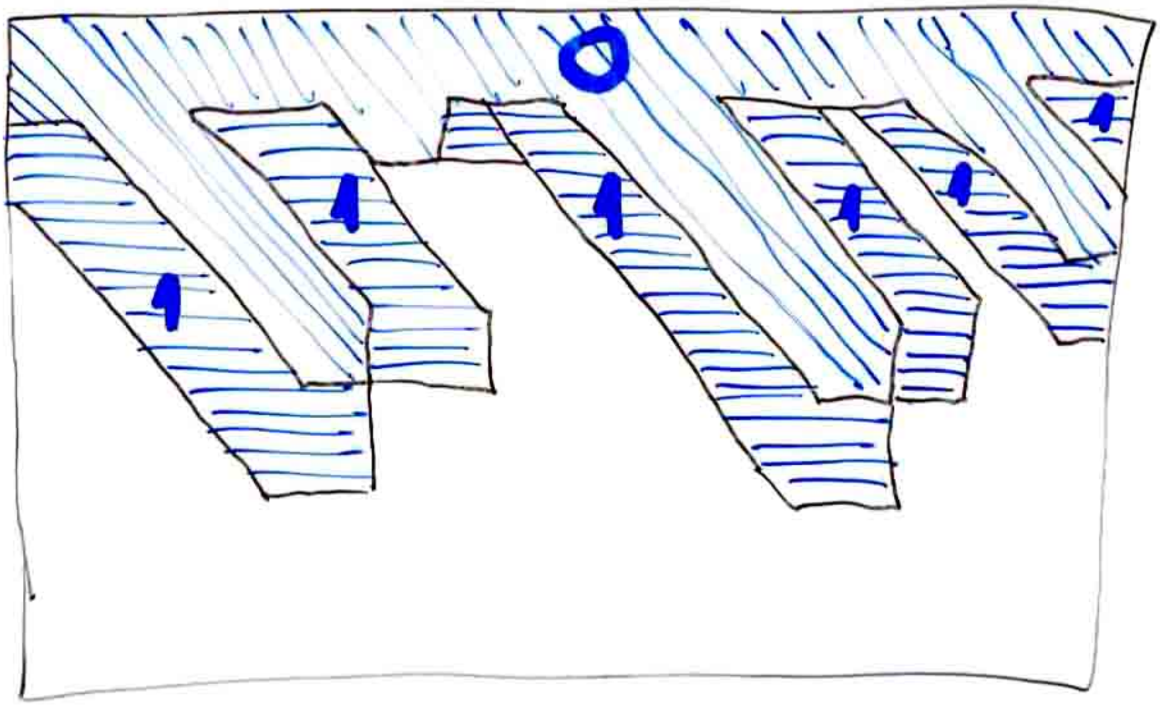


	0	A	B	B	A	B	B
0	0	0	0	0	0	0	0
A	1	0	1	1	0	1	1
B	2	1	0	1	1	0	1
A	3	2	1	1	1	1	1
B	4	3	2	1	2	1	1

Can we solve Galil's Open Problem for Edit Distance?

Idea: (Landau & Vishkin 1986)

Consider dynamic programming matrix D:

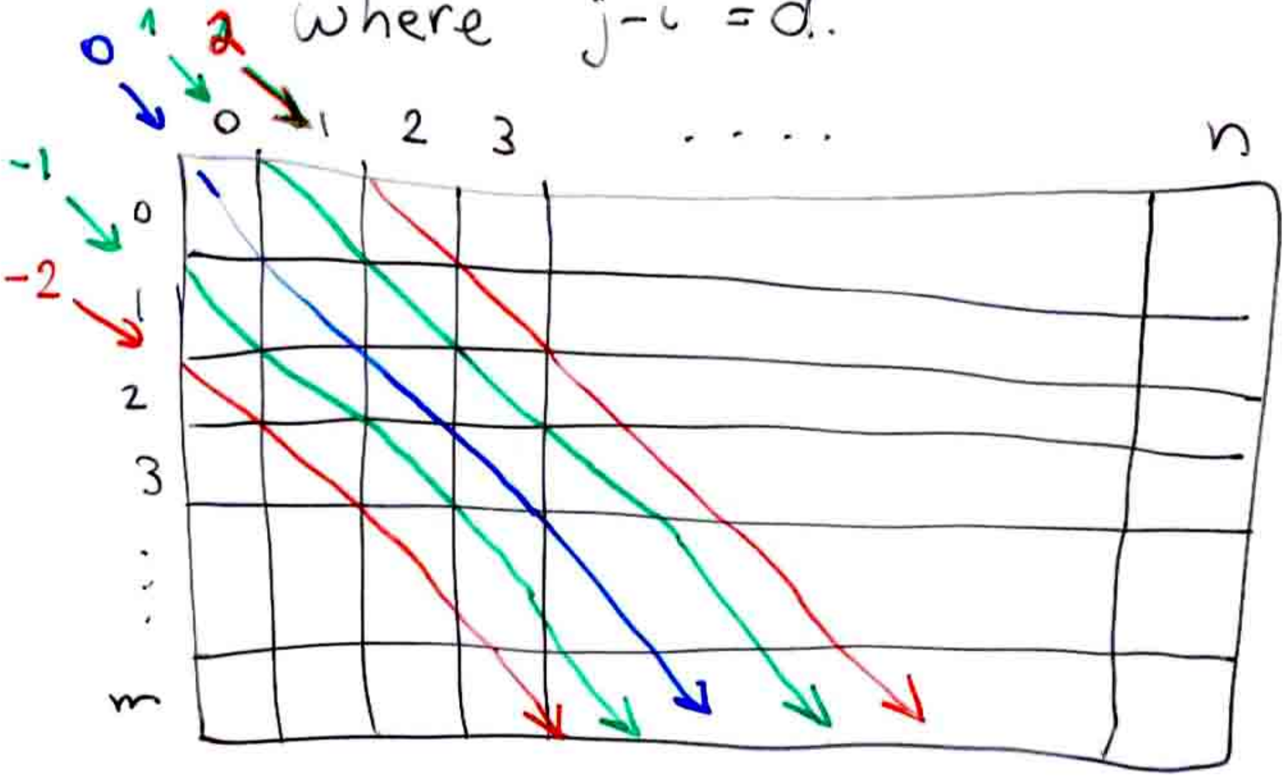


But now we can advance on diagonals as long as pattern = text.

Advance until number > k (no match) or till last row (match).

IMPLEMENTATION:

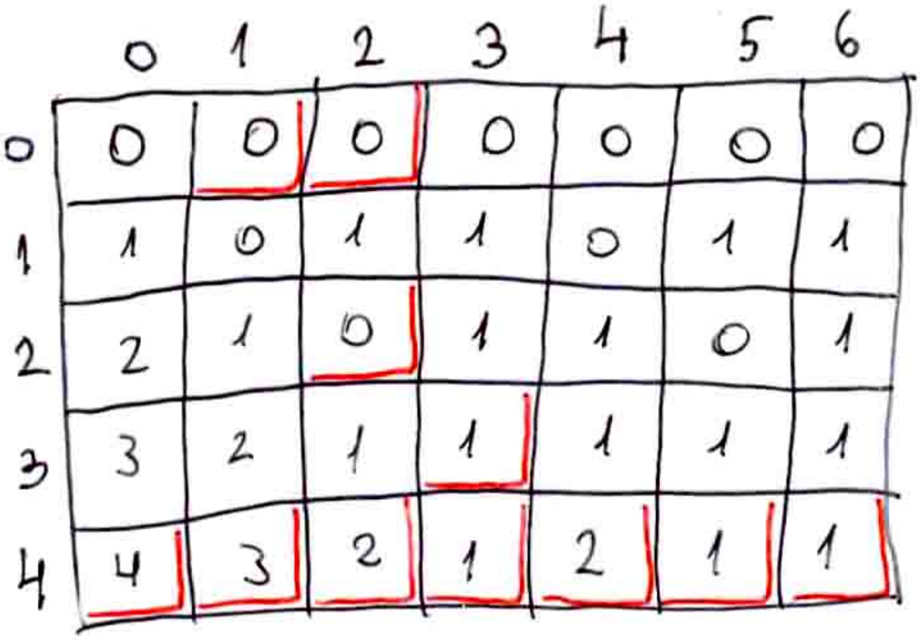
Define: diagonal d as all $D[i,j]$ where $j-i = d$.



$L_{d,e}$ = highest row in diagonal d where the number (of errors) is e.

EXAMPLE:

D



	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	1	0	1	1	0	1	1
2	2	1	0	1	1	0	1
3	3	2	1	1	1	1	1
4	4	3	2	1	2	1	1

L

← # of errors

	0	1	2	3	4
-4	/	/	/	/	4
-3	/	/	/	4	
-2	/	/	4		
-1	/	4			
0	2	3	4		
1	0	4			
2	0	4			

↑

diagonal

Algorithm (Dynamic Programming)

Given k errors.

Initialize:

	-1	0	1	2		k-1	k
-k						k-1	
-(k-1)					⋰		
⋮							
-3				2			
-2			1				
-1		0					
0	-1						
1	-1						
⋮	⋮						
(n-m)	-1						

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-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:

$$O(nk).$$