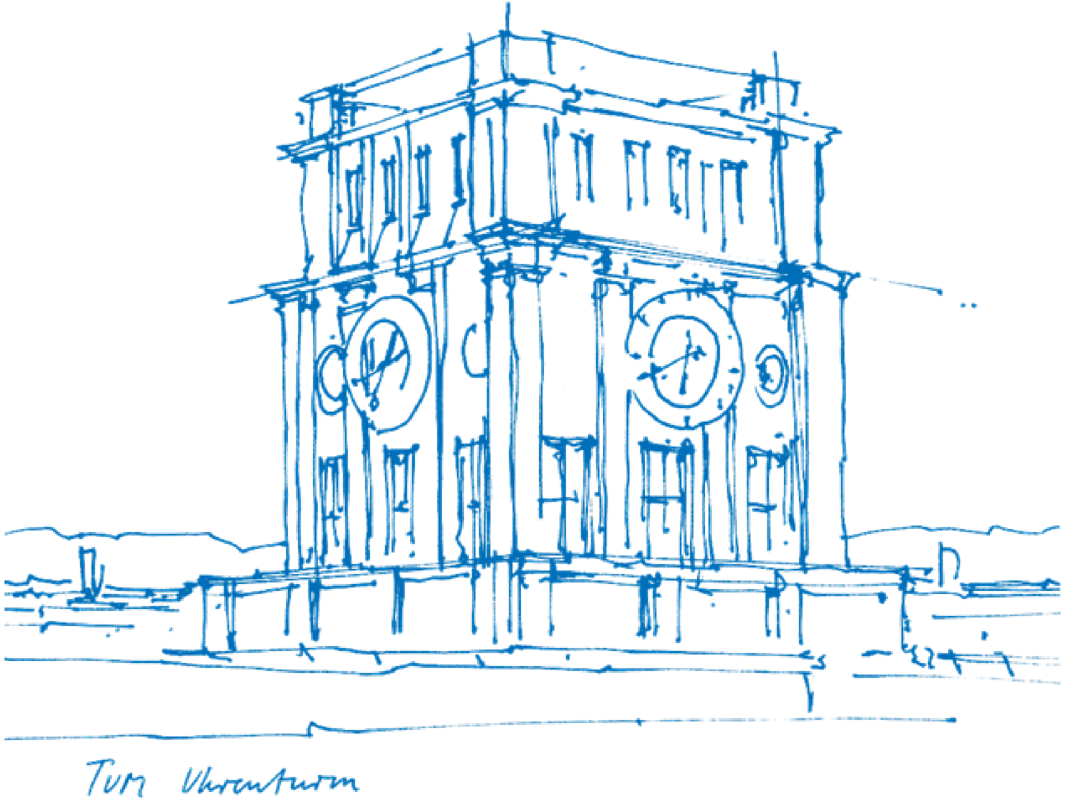
AACPP 2025

Week 10: String Algorithms

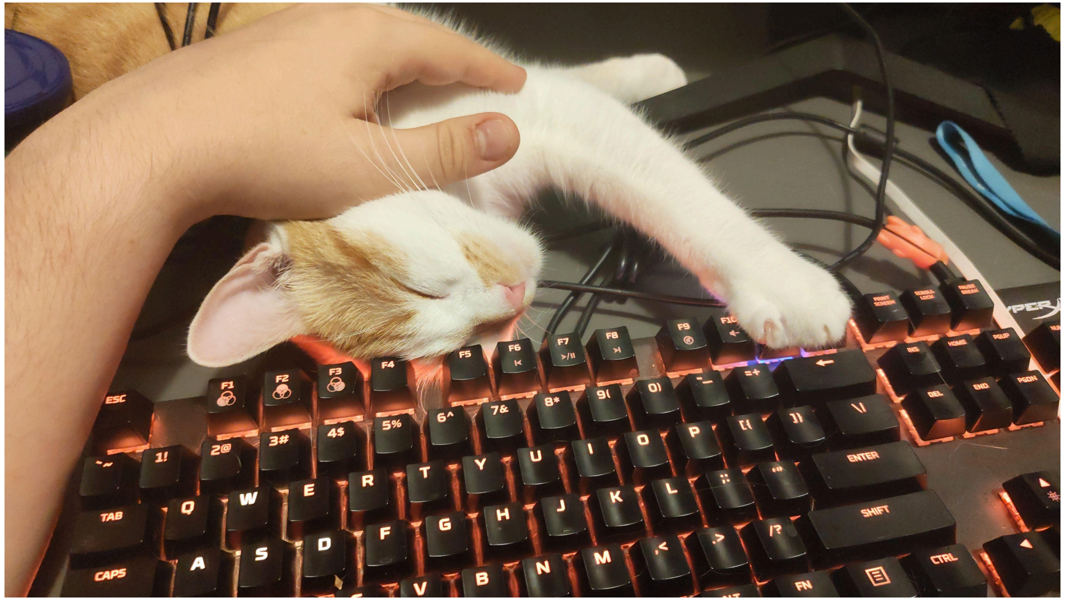
Mateusz Gienieczko, Mykola Morozov

School of Computation, Information and Technology
Technical University of Munich

2025.07.09



Seventh round – survey



Eighth round

Deadline – 15.07.2025, 10:00 AM.

Only one task this time!

TAP – Totally Aligned Buttons

Given s_1, c_1, s_2, c_2 and n find smallest $k \geq n$ such that there exist $x, y \in \mathbb{N}$:

$$xc_1 + s_1 = k = yc_2 + s_2$$

Equivalently, find minimal such x, y that:

$$xc_1 + s_1 = yc_2 + s_2 \geq n$$

We have no guarantees on the parameters (might not be coprime).

TAB – Totally Aligned Buttons

$$xc_1 + s_1 = yc_2 + s_2$$

Let $d = \gcd(c_1, c_2)$.

If $s_1 \not\equiv s_2 \texttt{ mod } d$ then there is no solution, since changing x, y will never change $xc_1 + s_1 \texttt{ mod } d$ or $yc_2 + s_2 \texttt{ mod } d$.

We want to find the number of keys such that if we take k modulo c_1 we get s_1, and modulo c_2 we get s_2.

$$\begin{cases} k \equiv s_1 \texttt{ mod } c_1 \\ k \equiv s_2 \texttt{ mod } c_2 \end{cases}$$

TAB – Totally Aligned Buttons

$$\begin{cases} k \equiv s_1 \bmod c_1 \\ k \equiv s_2 \bmod c_2 \end{cases}$$

We know how to solve these, we use EEA to get:

$$c_1 x + c_2 y = d$$

and take

$$k = \frac{s_2 x c_1 + s_1 y c_2}{d} \bmod \frac{c_1 c_2}{d}$$

TAB – Totally Aligned Buttons

This number might be too low. We need it to be at least n.

One more edge case – it also has to be at least s_1 and s_2.

We can add $\frac{c_1 c_2}{d}$ however many times are required to get to the requirement.

In total we pay $\mathcal{O}(\log c_1 + \log c_2)$ for EEA.

TAB – Totally Aligned Buttons

```
(x, y, gcd) = extended_euclid(c1, c2)
lcm = x * y / gcd
if s1 % gcd != s2 % gcd { return "IMPAWSSIBLE" }
k = (s2 * x * c1 + s1 * y * c2) / gcd % lcm

req = max(s1, s2, n)
if k >= req { return k }
else {
  m = div_ceil(req - sol_2, lcm)
  return k + lcm * m
}
```

Recall the plan

- Greedy and dynamic programming (DP)
- Trees
- Graphs
- Ways to turn graphs into trees (DFS, BFS, Dijkstra, MST)
- Ways to run DP on graphs (Toposort)
- Advanced graph algorithms (Matchings, flows)
- Binary Search Trees
- Number theory
- **String algorithms (KMP, tries, suffix tables)** ← *we are here*
- Some problems can't* even be solved efficiently (NP-completeness)

Strings

When we talk about strings we usually fix an alphabet Σ, which is a set of characters that strings contain.

Often we use a binary alphabet, $\Sigma = \{0, 1\}$ (equivalently $\{a, b\}$), or the alphabet of all lowercase English letters, $\Sigma = \{a, b, \ldots, z\}$. Usually $|\Sigma|$ is a small constant.

Strings

When we talk about strings we usually fix an alphabet Σ, which is a set of characters that strings contain.

Often we use a binary alphabet, $\Sigma = \{0, 1\}$ (equivalently $\{a, b\}$), or the alphabet of all lowercase English letters, $\Sigma = \{a, b, \ldots, z\}$. Usually $|\Sigma|$ is a small constant.

- we will usually use a for a character and w, u for words;
- length of a word is given by $|w|$;
- a^k is a repeated k times, and w^k is w repeated k times;
- Σ^n is the set of all words of length n over Σ;
- string concatenation as multiplication, e.g. aw, wu, or explicitly as $a \cdot w$, $w \cdot u$;
- $w[x..y]$ is the substring of w starting at x-th character up to y-th (inclusive).

Prefixes and suffixes

We say p is a *prefix* of w when there exists $k \le |w|$ such that $p = w[0..k]$. We write $p \sqsubseteq w$. We say p is a *proper* prefix if $k < |w|$ ($p \sqsubset w$).

Prefixes and suffixes

We say p is a *prefix* of w when there exists $k \leq |w|$ such that $p = w[0..k]$. We write $p \sqsubseteq w$. We say p is a *proper* prefix if $k < |w|$ ($p \sqsubset w$).

Similarly, s is a suffix of w if $s = w[k..]$ for some k and we write $s \sqsupseteq w$ ($s \sqsupset w$ for a proper suffix).

Prefixes and suffixes

We say p is a *prefix* of w when there exists $k \leq |w|$ such that $p = w[0..k]$. We write $p \sqsubseteq w$. We say p is a *proper* prefix if $k < |w|$ ($p \sqsubset w$).

Similarly, s is a suffix of w if $s = w[k..]$ for some k and we write $s \sqsupseteq w$ ($s \sqsupset w$ for a proper suffix).

A word u is a *prefix-suffix* or *border* of w if $u \sqsubset w \land u \sqsupset w$.

For example, in *ababa* the word *aba* is a prefix-suffix.

String matching

The fundamental problem is matching, finding a pattern u in a word w.

Usually the pattern is short and the word is long.

Naively this can be done in $\mathcal{O}(|w||u|)$ since the occurrence can start at any position and get mismatched at a very late position.

For example, imagine matching $a^k b$ in a^n.

String matching

The key idea is to not repeat matching that we know must or must not succeed.

Once we compare $a^k b$ at the first position of a^n we know a^{k-1} was matched, so we can just move by one and look for ab.

On the other hand, imagine a repetition of $a^k c$ as the text w. Then once we compare $a^k b$ at position 1, we know there's no way to match the pattern at any of the first $k+1$ positions.

KMP – Knuth-Morris-Pratt

The main idea is that when we match the first k characters of u and get a mismatch on $k+1$, then the part of the pattern that is already matched and can be used when restarting is the *longest prefix-suffix* of $u[..k]$.

Pattern: *abababbaba*

$\boldsymbol{i}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$\boldsymbol{w}$	a	b	a	b	a	b	b	b	a	b	a	b	a	b	a	b	b	a	b	a	b	a	b	b	$\$$
$\boldsymbol{u_1}$	a	b	a	b	a	b	b	a																	
$\boldsymbol{u_2}$																									
$\boldsymbol{u_3}$																									
$\boldsymbol{u_4}$																									

KMP – Knuth-Morris-Pratt

The main idea is that when we match the first k characters of u and get a mismatch on $k+1$, then the part of the pattern that is already matched and can be used when restarting is the *longest prefix-suffix* of $u[..k]$.

Pattern: *abababbaba*

$\boldsymbol{i}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$\boldsymbol{w}$	*a*	*b*	*a*	*b*	*a*	*b*	*b*	*b*	*a*	*b*	*a*	*b*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*a*	*b*	*a*	*b*	*b*	\$
$\boldsymbol{u}_1$	*a*	*b*	*a*	*b*	*a*	*b*	*b*	*a*																	
$\boldsymbol{u}_2$									*a*	*b*	*a*	*b*	*a*	*b*	*b*										
$\boldsymbol{u}_3$																									
$\boldsymbol{u}_4$																									

KMP – Knuth-Morris-Pratt

The main idea is that when we match the first k characters of u and get a mismatch on $k+1$, then the part of the pattern that is already matched and can be used when restarting is the *longest prefix-suffix* of $u[..k]$.

Pattern: *abababbaba*

$\boldsymbol{i}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$\boldsymbol{w}$	a	b	a	b	a	b	b	b	a	b	a	b	a	b	a	b	b	a	b	a	b	a	b	b	$\$$
$\boldsymbol{u}_1$	a	b	a	b	a	b	b	a																	
$\boldsymbol{u}_2$									a	b	a	b	a	b	b										
$\boldsymbol{u}_3$											a	b	a	b											
$\boldsymbol{u}_4$																									

KMP – Knuth-Morris-Pratt

The main idea is that when we match the first k characters of u and get a mismatch on $k + 1$, then the part of the pattern that is already matched and can be used when restarting is the *longest prefix-suffix* of $u[..k]$.

Pattern: *abababbaba*

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
w	*a*	*b*	*a*	*b*	*a*	*b*	*b*	*b*	*a*	*b*	*a*	*b*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*a*	*b*	*a*	*b*	*b*	$
u_1	*a*	*b*	*a*	*b*	*a*	*b*	*b*	*a*																	
u_2									*a*	*b*	*a*	*b*	*a*	*b*	*b*										
u_3											*a*	*b*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*a*					
u_4																									

KMP – Knuth-Morris-Pratt

The main idea is that when we match the first k characters of u and get a mismatch on $k + 1$, then the part of the pattern that is already matched and can be used when restarting is the *longest prefix-suffix* of $u[..k]$.

Pattern: *abababbaba*

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
w	a	b	a	b	a	b	b	b	a	b	a	b	a	b	a	b	b	a	b	a	b	a	b	b	$\$$
u_1	a	b	a	b	a	b	b	a																	
u_2									a	b	a	b	a	b	b										
u_3											a	b	a	b	a	b	b	a	b	a					
u_4																		a	b	a					

KMP – Knuth-Morris-Pratt

The main idea is that when we match the first k characters of u and get a mismatch on $k+1$, then the part of the pattern that is already matched and can be used when restarting is the *longest prefix-suffix* of $u[..k]$.

Pattern: *abababbaba*

$\boldsymbol{i}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$\boldsymbol{w}$	a	b	a	b	a	b	b	b	a	b	a	b	a	b	a	b	b	a	b	a	b	a	b	b	$\$$
$\boldsymbol{u}_1$	a	b	a	b	a	b	b	a																	
$\boldsymbol{u}_2$									a	b	a	b	a	b	b										
$\boldsymbol{u}_3$											a	b	a	b	a	b	b	a	b	a					
$\boldsymbol{u}_4$																		a	b	a	b	a	b	b	a

KMP – Knuth-Morris-Pratt

i	0	1	2	3	4	5	6	7	8	9
u	a	b	a	b	a	b	b	a	b	a
π	0	0	1	2	3	4	0	1	2	3

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
w	a	b	a	b	a	b	b	b	a	b	a	b	a	b	a	b	b	a	b	a	b	a	b	b	$\$$
u_1	a	b	a	b	a	b	b	a																	
u_2																									
u_3																									
u_4																									

KMP – Knuth-Morris-Pratt

$\boldsymbol{i}$	0	1	2	3	4	5	6	7	8	9
$\boldsymbol{u}$	a	b	a	b	a	b	b	a	b	a
$\boldsymbol{\pi}$	0	0	1	2	3	4	0	1	2	3

$\boldsymbol{i}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$\boldsymbol{w}$	a	b	a	b	a	b	b	b	a	b	a	b	a	b	a	b	b	a	b	a	b	a	b	b	$\$$
$\boldsymbol{u}_1$	a	b	a	b	a	b	b	a																	
$\boldsymbol{u}_2$									a	b	a	b	a	b	b										
$\boldsymbol{u}_3$																									
$\boldsymbol{u}_4$																									

KMP – Knuth-Morris-Pratt

i	0	1	2	3	4	5	6	7	8	9
u	a	b	a	b	a	b	b	a	b	a
π	0	0	1	2	3	4	0	1	2	3

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
w	a	b	a	b	a	b	b	b	a	b	a	b	a	b	a	b	b	a	b	a	b	a	b	b	$\$$
u_1	a	b	a	b	a	b	b	a																	
u_2									a	b	a	b	a	b	b										
u_3											a	b	a	b											
u_4																									

KMP – Knuth-Morris-Pratt

i	0	1	2	3	4	5	6	7	8	9
u	a	b	a	b	a	b	b	a	b	a
π	0	0	1	2	3	4	0	1	2	3

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
w	a	b	a	b	a	b	b	b	a	b	a	b	a	b	a	b	b	a	b	a	b	a	b	b	$
u_1	a	b	a	b	a	b	b	a																	
u_2									a	b	a	b	a	b	b										
u_3											a	b	a	b	a	b	b	a	b	a					
u_4																									

KMP – Knuth-Morris-Pratt

i	0	1	2	3	4	5	6	7	8	9
u	a	b	a	b	a	b	b	a	b	a
π	0	0	1	2	3	4	0	1	2	3

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
w	a	b	a	b	a	b	b	b	a	b	a	b	a	b	a	b	b	a	b	a	b	a	b	b	\$
u_1	a	b	a	b	a	b	b	a																	
u_2									a	b	a	b	a	b	b										
u_3											a	b	a	b	a	b	b	a	b	a					
u_4																		a	b	a					

KMP – Knuth-Morris-Pratt

i	0	1	2	3	4	5	6	7	8	9
u	a	b	a	b	a	b	b	a	b	a
π	0	0	1	2	3	4	0	1	2	3

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
w	a	b	a	b	a	b	b	b	a	b	a	b	a	b	a	b	b	a	b	a	b	a	b	b	$\$$
u_1	a	b	a	b	a	b	b	a																	
u_2									a	b	a	b	a	b	b										
u_3											a	b	a	b	a	b	b	a	b	a					
u_4																		a	b	a	b	a	b	b	a

KMP – Knuth-Morris-Pratt

Fact: *A prefix-suffix of a prefix-suffix of w is a prefix-suffix of w.*

This implies that $\pi[i+1] \leq \pi[i] + 1$.

Here's a $\mathcal{O}\left(|u|^2\right)$ algorithm to compute the π function for u:

```
pi[0] = 0
for i in 1..n
  j = pi[i - 1]
  while j > 0 && u[i] != u[j]
    j -= 1
  if u[i] == u[j]
    j += 1
  pi[i] = j
```

KMP – Knuth-Morris-Pratt

Fact: *A prefix-suffix of a prefix-suffix of w is a prefix-suffix of w.*

One more optimisation stemming from this is that we only need to look at prefix-suffix of the previous prefix-suffix to possibly extend. We get $\mathcal{O}(|u|)$:

```
pi[0] = 0
for i in 1..n
  j = pi[i - 1]
  while j > 0 && u[i] != u[j]
    j = pi[j - 1] // <--
  if u[i] == u[j]
    j += 1
  pi[i] = j
```

KMP – Knuth-Morris-Pratt

Having a precomputed π we can find all matches of u in any w in time $\mathcal{O}(|w|)$.

```
i = 0
j = 0
while i < |w|
  if w[i] == u[j]
    i += 1
    j += 1
    if j == |u|
      report_found(j - |u|)
      j = pi[j]
  else // w[i] != u[j]
    j = pi[j]
    if j == 0
      i += 1
```

The time analysis is amortised – j cannot increase more than $|w|$ times.

Tries

A *trie* (pronounced either way, you do you), also called a *prefix tree*, is a specific tree structure representing a set of strings S over a fixed alphabet Σ.

Each node represents a prefix of some string in $|S|$, and is a leaf or has Σ children. Going down in the trie, the edge that we choose appends another letter to the current string.

Nodes that represent strings in S are marked.

The depth of the tree is thus the length of the longest string in the set, while the number of nodes is limited simultaneously by:

- sum of lengths,
- Σ times the length of the longest string.

Tries

Tries can be further compressed by representing long paths with no branching with a single long edge.

When traversing we still look up by the next letter, but then retrieve a potentially longer substring to compare with our lookup.

This is then called a *radix tree*.

Tries

Tries have predictable speed and no collisions, unlike generic hash tables.

They automatically sort all keys lexicographically.

One can run prefix-based queries, like retrieving all strings with a given prefix.

Aho-Corasick

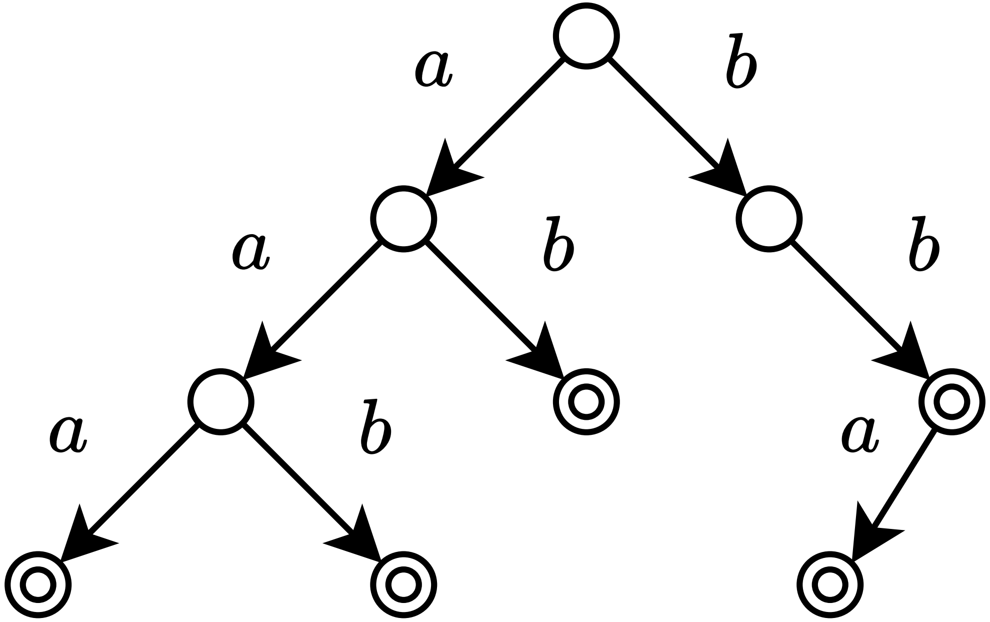
Aho-Corasick is an algorithm that preprocesses k different patterns and creates a *deterministic finite automaton* which can match all of them simultaneously.

The structure takes $\mathcal{O}(\Sigma m)$ space, where m is the sum of lengths of patterns. Matching happens in $\mathcal{O}(n)$, where n is the length of text.

Aho-Corasick – construction

We start by building a trie on all keys.

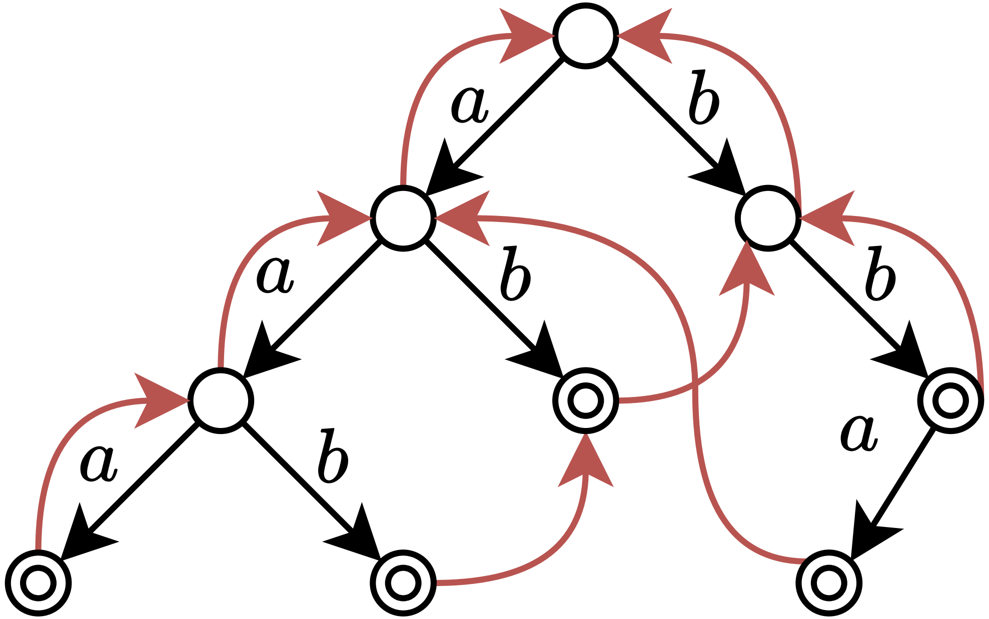
For a running example we take $\{aaa, aab, ab, bb, bba\}$ as the dictionary.



Aho-Corasick – construction

We now need to add all missing transitions.

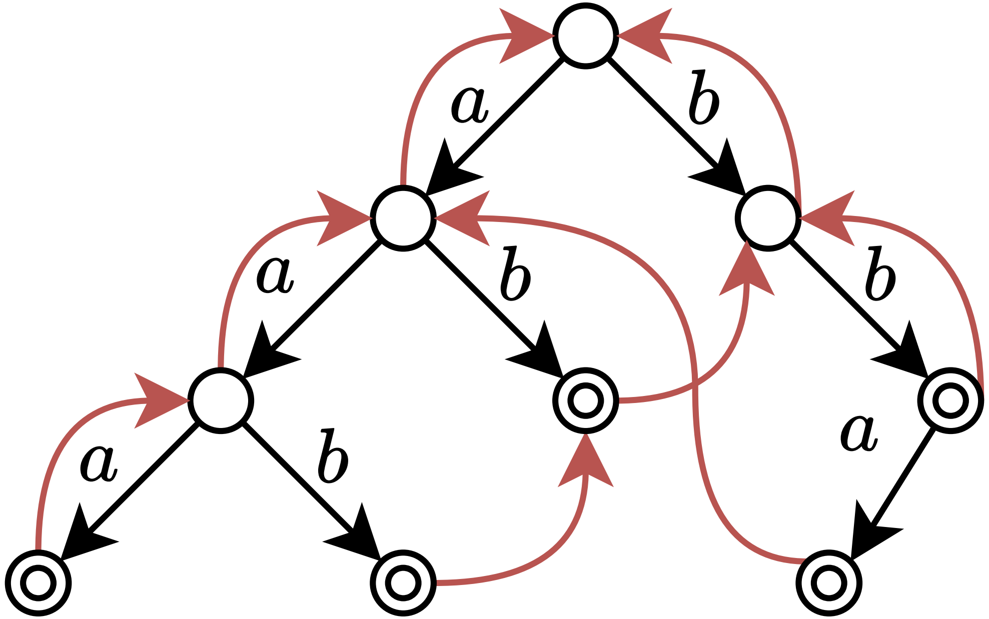
We compute *suflinks*, links to the nodes representing the longest proper suffix of the string represented by the current node.



Aho-Corasick – construction

A suflink of v with parent p where the $p \to v$ edge is labelled with x is now given by going to the parent, following its suflink, and then transitioning over x.

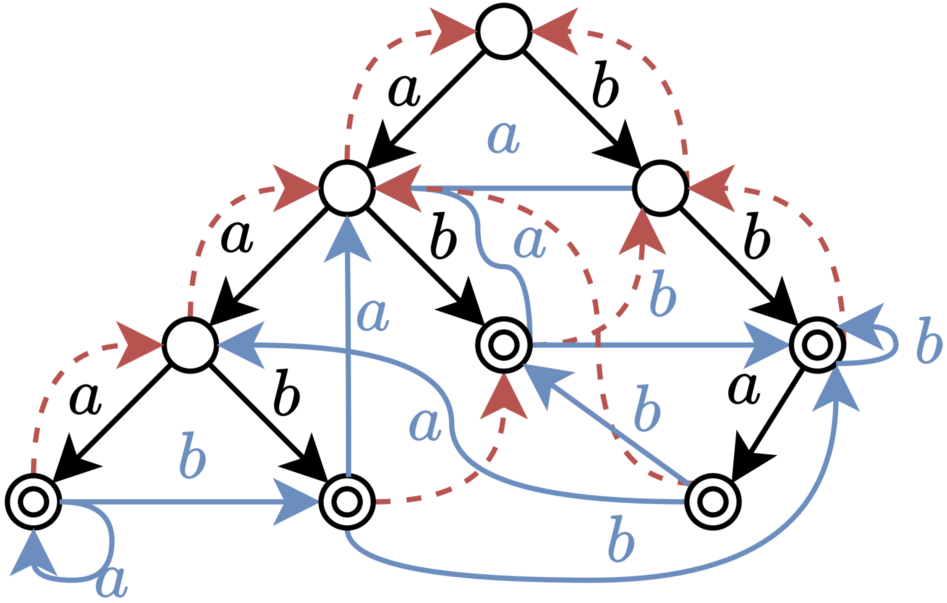
Suflinks always go "up" in the tree, so this can be computed directly with a BFS.



Aho-Corasick – construction

Finally, a missing transition over x is given: go to the suflink and picking x.

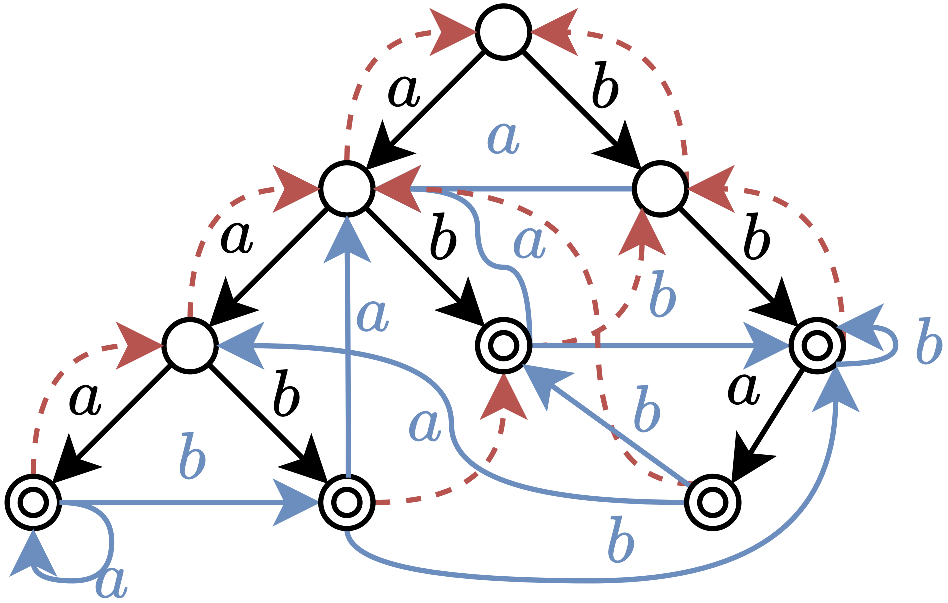
This can be computed right after the suflink in BFS order.



Aho-Corasick – construction

Matching a text is now just running the DFA: read a letter of the text, transition through the appropriate edge.

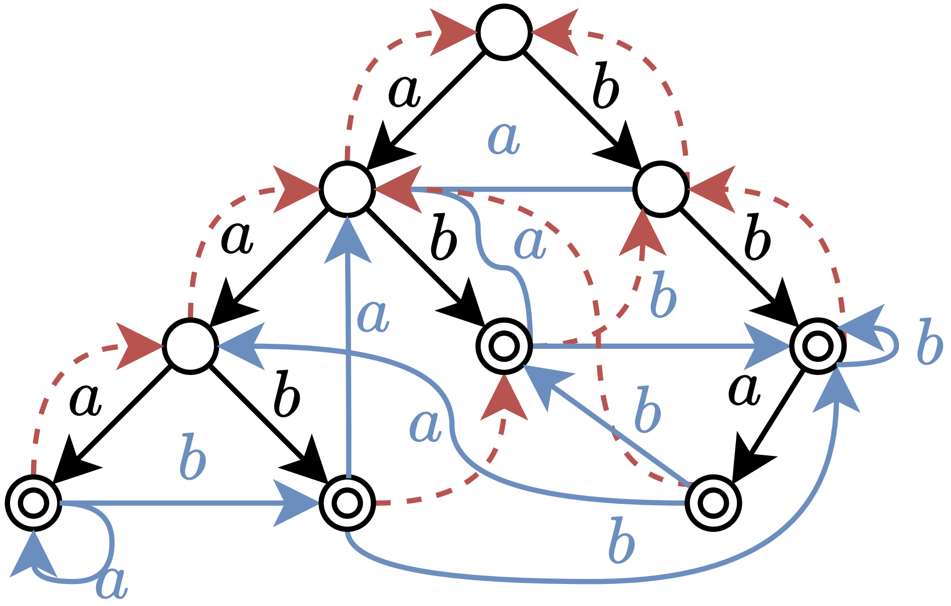
When we visit a marked node, output a match.



Aho-Corasick – construction

If we want to output *all* matches then we also need to compute the next marked node that can be reached by following suflinks (if any).

E.g. here matching *aab* also matches *ab*.



Karp-Miller-Rosenberg (Suffix Array)

A *suffix array* is a sorted array of all suffixes of a string w, $|w| = n$.

It can be constructed in linear time, but the simplest way is KMR in $\mathcal{O}(n \log n)$.

KMR constructs an array where $\mathrm{kmr}[i][k]$ is an identifier of the substring

$$w\left[i..i + 2^k\right]$$

The identifiers are different for different substrings and equal for equal ones. Moreover, order on the ids is also the lexicographical order on the substrings.

The final level of this array ($k = \log n$) is the suffix array.

Karp-Miller-Rosenberg (Suffix Array)

Construction is inductive, starting from level 0 – map each character to its ID (index in an ordered alphabet). For the next level we combine pairs of identifiers to construct a twice-longer substring and give unique IDs by sorting.

	a	b	a	a	b	b	b	a	b	a	a	b	#...
2^0	1	2	1	1	2	2	2	1	2	1	1	2	0...
2^1	2	4	1	2	5	5	4	2	4	1	2	3	0...
2^2	4	7	2	5	10	9	8	4	7	1	3	6	0...
2^3	5	9	2	6	12	11	10	4	8	1	3	7	0...
2^4	5	9	2	6	12	11	10	4	8	1	3	7	0...

Highlighted: kmr[4][3] and the IDs from previous level that contribute to it.

Suffix Array

The suffix array itself already allows us to do some cool things.

For example, it allows us to look for a pattern of length m in the string in $m \log n$ time by binary-searching the sorted suffixes.

Compare this with KMP:

- in KMP the pattern is fixed and preprocessed, then we can search *any text* for the pattern;
- with a suffix array the text is fixed, and we can search for *any pattern*.

LCP Array

A suffix array can be augmented further with a Longest Common Prefix array.

$\mathrm{lcp}[i]$ is the length of the longest common prefix of the i-th and $i - 1$-th suffix.

LCP Array

0	#	lcp
1	*aab#*	
1	*aabbbabaab#*	
1	*ab#*	
1	*abaab#*	
1	*abaabbbabaa#*	
1	*abbbabaab#*	
1	*b#*	
1	*baab#*	
1	*baabbbabaab#*	
1	*babaab#*	
1	*bbabaab#*	
1	*bbbabaab#*	

LCP Array

0	#	lcp
1	*aab#*	0
1	*aabbbabaab#*	3
1	*ab#*	1
1	*abaab#*	2
1	*abaabbbabaa#*	5
1	*abbbabaab#*	2
1	*b#*	0
1	*baab#*	1
1	*baabbbabaab#*	4
1	*babaab#*	2
1	*bbabaab#*	1
1	*bbbabaab#*	2

Using LCP we can solve various problems about substrings.

For example, finding the longest substring that occurs at least twice in linear time (exercise 😏).

Using a constant time RMQ structure we can also perform the arbitrary pattern matching in $\mathcal{O}(m + \log n)$.

Suffix Tree

A *suffix tree* of w is a compressed trie of suffixes of w. It has n leaves, where leaf number i represents the suffix $w[i..n]$. It has $2n$ nodes total and can be built in linear time.

Finding a pattern in the string can now be done in $\mathcal{O}(m)$. It can also be used for solving a plethora of other problems in linear time:

- Longest common substrings of two strings,
- aforementioned longest repeating substring,
- most frequently occurring substrings (of some length),
- shortest string not occurring in a set of strings,
- matching a pattern with k allowed mistakes (in $\mathcal{O}(nk)$).

Suffix Tree – Construction

One can construct the suffix tree from a suffix array and LCP.

The idea is that one can traverse the tree down by going through the suffix array and then find the branches using LCP.

Instead we'll present the Ukkonen's Algorithm that builds a suffix tree without this intermediate step.

Note that the suffix array can be trivially constructed from a suffix tree (leaves are already ordered because it's a trie).

Suffix Tree – Ukkonen's Algorithm

We add suffixes to the trie one by one, starting with just the root.

During construction we ignore terminals.

Naively, we would add the suffix $w[..i]$ by finding all $w[j..i]$ in the tree and making sure the extension $w[j..i]w[i+1]$ is also present in the tree.

- If $w[j..i+1]$ is already present as a node we do nothing.
- If $w[j..i]$ is a leaf then just extend the label to the leaf by $w[i+1]$.
- If $w[j..i]$ is in the middle of an edge, the edge has to be split and a new node inserted.

Suffix Tree – Ukkonen's Algorithm

We add suffixes to the trie one by one, starting with just the root.

During construction we ignore terminals.

Naively, we would add the suffix $w[..i]$ by finding all $w[j..i]$ in the tree and making sure the extension $w[j..i]w[i+1]$ is also present in the tree.

- If $w[j..i+1]$ is already present as a node we do nothing.
- If $w[j..i]$ is a leaf then just extend the label to the leaf by $w[i+1]$.
- If $w[j..i]$ is in the middle of an edge, the edge has to be split and a new node inserted.

A number of speedups is added to make this $\mathcal{O}(n)$. An extensive explanation is given in Gusfeld, Algorithms on Strings, Trees, and Sequences.

Hashing

Hashing is a *heuristic* method of solving many string problems relying on comparisons.

The idea is to compute a single integer value as a *fingerprint* of the string that allows us to quickly say if two strings are *different*, but which can have collisions causing different strings to appear equal.

We will compute the fingerprint in a way that allows for quick recovery of substring fingerprints.

Hashing

We need two parameters: a base b and a modulus m. Best characteristics are obtained when there are both prime.

We need $b \geq |\Sigma|$, and the higher the modulus the lower the chance of collisions.

Good values for base are $b = 2$ for binary alphabets, $p = 29$ for English lowercase.

Good values for modulus are $m = 10^9 + 7$, $m = 10^9 + 9$, $m = 10^{18} + 9$.

Hashing

The hash of a word w of length n for given b, m is:

$$w[0]b^0 + w[1]b^1 + \ldots + w[n-1]b^{n-1} \bmod m$$

If we compute the *prefix hash* of w we can then obtain the hash of any substring via:

$$h(w[i..j]) = (h(w[..j]) - h(w[..i])) \cdot b^{-i} \bmod m$$

We need the multiplicative inverse of b modulo m, but it is constant for fixed b and m.

Hashing

Calculating the chance for collision requires some probability theory.

The strongest result is given if m is a randomly chosen prime number, but for competitive programming it's enough to "randomly" choose your favourite one.

The chance is roughly $\frac{1}{m}$ of collision *per string pair*, which gives about 0.1% chance of collision under 10^6 compared pairs.

Tests can be antagonistic for a given modulo choice. A safe bet is using two different moduli and comparing both hashes. Strings are equal iff they're equal under both. This decreases collision chances to $\frac{1}{m_1 m_2}$ per comparison.

Hashing – practical notes

When computing hashes it’s important to use modulo everywhere.

The powers b^k have to be computed modulo m. Multiplying by the character code has to be done modulo. Addition has to be done modulo.

Hashing is a quite expensive operation for the CPU, as modulo is expensive. Hashing solutions are likely to be slower than well-implemented deterministic algorithms. This is especially visible when using more than one hash.

See you next week

PPM: 15.07.2025, 10:00 AM

Good luck!