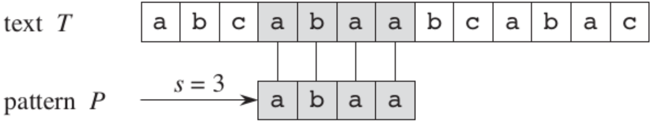
String Matching

Pedro Ribeiro

DCC/FCUP

2025/2026



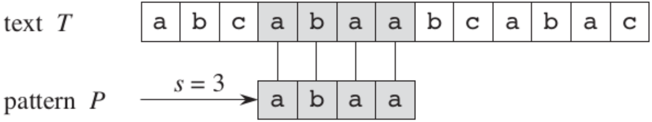
String Problems

- There is an entire area of study dealing with string related problems
- Examples of string related problems:
  - Given a text and a pattern, find all exact or approximate occurrences of the pattern in the text (classic text search)
  - Given a string, find the largest string that occurs at least k times
  - Given two strings find the edit distance between them, with various operations available, such as deletions, additions and substitutions.
  - Given two strings, find the largest common substring
  - Given a set of strings, find the "better" tree that can describe and connect them (phylogeny tree)
  - Given a set of strings, find the shortest superstring that contains all the strings (one of the core problems of DNA sequencing)
  - ...
- Here at Competitive Programming we will just give a brief glimpse on the whole field and in particular we will focus on the **string matching problem** (and some variants)

The String Matching Problem

Let's formalize the string matching problem:

- **Text**: array $T[1..n]$ of length n
- **Pattern**: array $P[1..m]$ of length $m \leq n$
- The characters of T and P are characters drawn from an alphabet Σ
  - For example, we could have $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, ..., z\}$
- A pattern P occurs with shift s in text T (or occurs beginning at position $s+1$) if $T[s+i] = P[i]$ for $1 \leq i \leq m$



String Matching Problem

Given a text T and a pattern P, find all valid shifts of P in T, or output that no occurrence can be found.

- One common variation is to find only one (ex: the first) possible shift

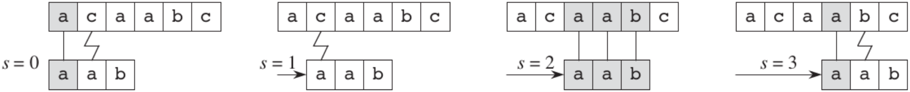
Naive String Matching

- Here is an (obvious) brute force algorithm for finding all valid shifts:

NAIVE-STRING-MATCHER(T, P)

1 $\quad n = T.length$
2 $\quad m = P.length$
3 $\quad$ **for** $s = 0$ **to** $n - m$
4 $\qquad$ **if** $P[1\,..\,m] == T[s+1\,..\,s+m]$
5 $\qquad\quad$ print "Pattern occurs with shift" s

- This algorithm tries explicitly every possible shift *s*
- Line 4 implies a loop to check if all characters match or exits if there is a mismatch



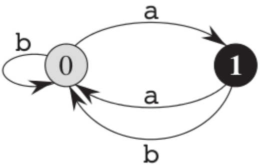
Naive String Matching

- What is the time complexity of the naive algorithm?
- $\mathcal{O}((n-m)m)$, which is $\mathcal{O}(mn)$ assuming m is "relatively small" ($m < n/2$) compared to n.
- The worst case is something like searching for *aaa...aaab* in a text consisting solely of *a*'s.
- If the text is random, this algorithm would be "not too bad" (if exiting as soon as a mismatch is found) but real text (ex: english or DNA) is really not completely random.
- This solution can also be acceptable if m is "really" small

Deterministic Finite Automaton

- How can we do better?
- Once we are at a certain shift, what information can we use about the previous shifts we tested?
- One possible (high-level) idea is to build a **deterministic finite automaton** (DFA) to represent what we know about the pattern and in what state of the search we are.

	input	
state	a	b
0	1	0
1	0	0



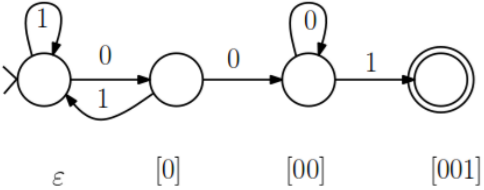
An example DFA that matches strings of $\Sigma = \{a, b\}$ finishing with an odd number of a's

Deterministic Finite Automaton

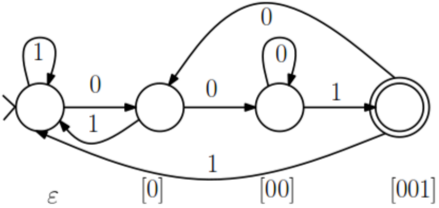
- Imagine a DFA with $m + 1$ states, arranged in a "line"
- The i-th state represents that we are now at position i of the pattern, that is, we matched the first i characters.
- Now, if we match the next character, we move to state $i + 1$ (matched $i + 1$ characters). If not, we can skip to another (previous) state.
- Which state should we go once we have a miss? If we go back to the initial state, then we are no better than the naive algorithm! We should go to the furthest state we know its possible.

Deterministic Finite Automaton

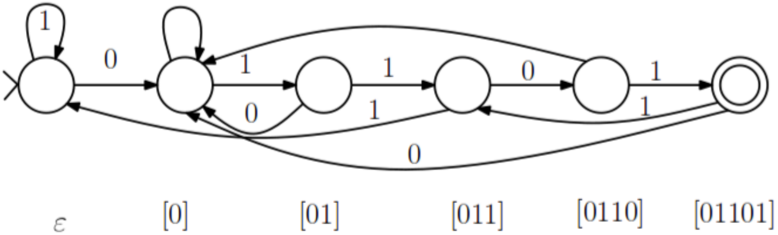
Imagine $P = 001$. We could use the following DFA:



This would only find the first occurrence of P. What to change so that it finds all occurrences?



What if the pattern is for instance $P = 01101$?



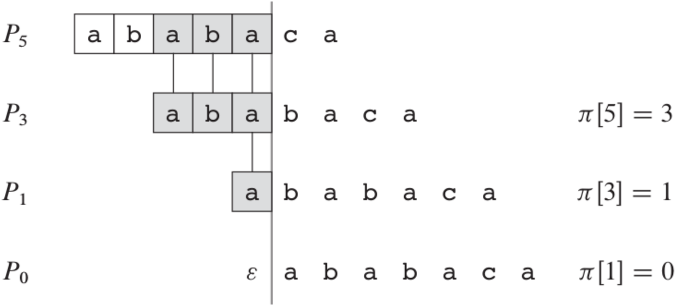
Deterministic Finite Automaton

- What is the complexity of matching after having a DFA like this?
- The matching is linear on the size of the text! $\mathcal{O}(n)$
- We must however take in account the time to build the respective DFA. If it takes $f(m)$, than the total time is $f(m) + \mathcal{O}(n)$.
- We will now show how the **Knuth-Morris-Pratt (KMP) algorithm** can build the "equivalent" of this DFA in time linear on the size of the pattern! $\mathcal{O}(m)$

Knuth-Morris-Pratt Algorithm

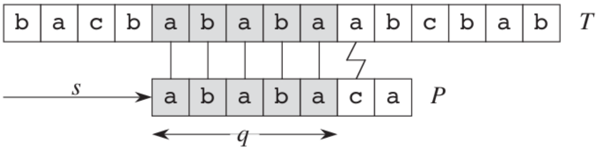
- Let $\pi[i]$ be the largest integer smaller than i such that $P[1..\pi[i]]$ (longest prefix) is a suffix of $P[1..i]$.

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

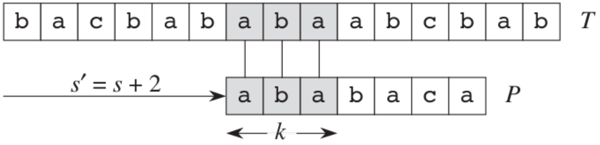


Knuth-Morris-Pratt Algorithm

- How can we use the information in $\pi[]$ to our matching?
- When he have a mismatch at position $i + 1$... we rollback to $\pi[i]$!
  - This is the next possible "largest" partial match



(a)



(b)

Knuth-Morris-Pratt Algorithm

- Let us look at the KMP main algorithm:

```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                      // number of characters matched
for i = 1 to n                             // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                           // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                          // next character matches
    if q == m                              // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                           // look for the next match
```

- What is the **temporal complexity** of this algorithm?

Knuth-Morris-Pratt Algorithm

- Let's for now ignore the time taken in computing π.
- The loop on line 5 takes time n. But what about the loop on line 6?
- The main "insight" is that we can never go back more than what we have already advanced. If we advance k characters in the text, than the call to line 7 can only make q go back k characters
- In other words, q is only increased in line 9 (at most once per each iteration of the cycle of line 5). Since when it is decreased it can never be negative (by the definition of π), this means it will have at most n decrements.
- This means that the while loop will never have more than n iterations!
- In an amortized sense (aggregate method), the time needed for the entire procedure is **linear on the size of the text**: $\mathcal{O}(n)$

- What about computing π?
- It is basically comparing the pattern against itself!

COMPUTE-PREFIX-FUNCTION(P)

```
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k+1] ≠ P[q]
        k = π[k]
    if P[k+1] == P[q]
        k = k + 1
    π[q] = k
return π
```

- What is the **temporal complexity** of this part?

Knuth-Morris-Pratt Algorithm

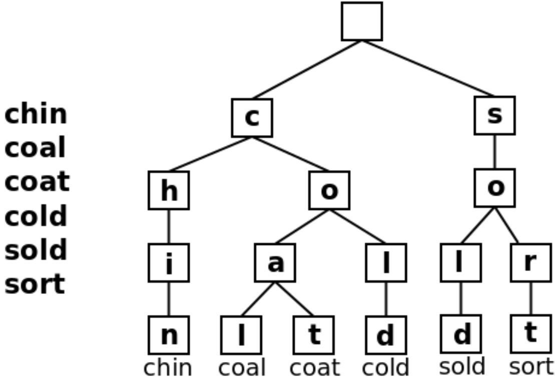
- Using a similar rationale to what we did before, the time is **linear on the size of the pattern**: $\mathcal{O}(m)$
- The entire KMP algorithm then takes $\mathcal{O}(n + m)$
  - Pre-processing: $\mathcal{O}(m)$
  - Matching: $\mathcal{O}(n)$

Trie

- From an algorithm revolving around the pattern, we will now focus on a data-structure centered on a set of words being searched
- A **trie** (also known as **prefix tree**) is a data structure representing a set of words (that can have values associated with it)
  - The root represents the empty string
  - Descendants share the same prefix

Trie

An example trie with 6 words



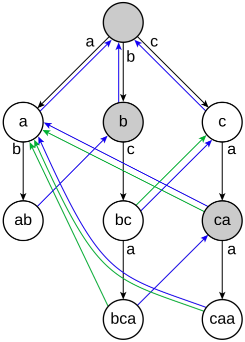
Note that the letters can be thought of as the edges and not the nodes

Trie

- We can **check** if a string of size n is stored in the trie in $\mathcal{O}(n)$ time
- We can **insert** a new word of size n in $\mathcal{O}(n)$ time
- We can **remove** a word of size n in $\mathcal{O}(n)$ time
- We exemplified with words, but tries **can store other types of data** (ex: numbers, or any data that we can separate in individual pieces)

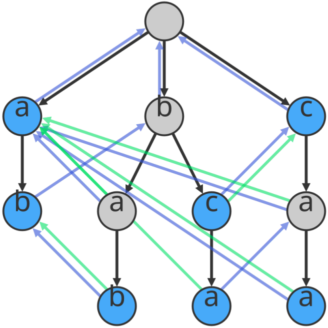
Aho–Corasick Algorithm

- What if we **combine** the ideas of a **trie** with **KMP** to search **at the same time** for several subtrings in a text?
- That's precisely the idea of the **Aho–Corasick** algorithm!
- The informal idea is simply to **build a trie with all the strings** and then create an **"automaton"** from that, using **extra links**.



Aho–Corasick Algorithm

- Let's see a real example:

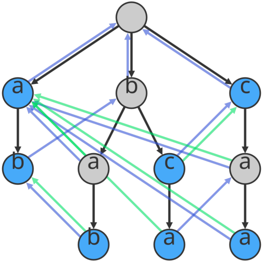


Dictionary {a, ab, bab, bc, bca, c, caa}

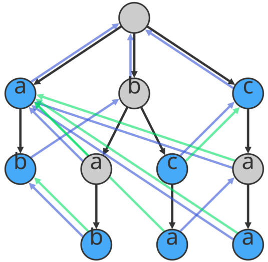
Path	In dictionary	Suffix link	Dict suffix link
()	–		
(a)	+	()	
(ab)	+	(b)	
(b)	–	()	
(ba)	–	(a)	(a)
(bab)	+	(ab)	(ab)
(bc)	+	(c)	(c)
(bca)	+	(ca)	(a)
(c)	+	()	
(ca)	–	(a)	(a)
(caa)	+	(a)	(a)

- Nodes in gray: Nodes without words ending on it
- Nodes in blue: Nodes with a word ending on it
- Edges in **black**: normal trie links
- Edges in blue: suffix links (largest proper suffix)
- Edges in green: dictionary suffix links (largest dictionary proper suffix)

Aho–Corasick Algorithm



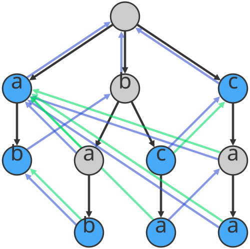
- At each step, the current node is extended by finding its **child**, and if that doesn't exist, finding its **suffix's child**, and if that doesn't work, finding its **suffix's suffix's child**, and so on, finally ending in the root node if nothing's seen before.
- When the algorithm reaches a node, it **outputs** all the dictionary entries that end at the current character position in the input text. This is done by printing every node reached by following the **dictionary suffix links**, starting from that node, and continuing until it reaches a node with no **dictionary suffix link**. In addition, the node itself is printed, if it is a dictionary entry.



- Execution on input string **abccab** yields the following steps:

Node	Remaining string	Output:end position	Transition	Output
()	abccab		start at root	
(a)	bccab	a:1	() to child (a)	Current node
(ab)	ccab	ab:2	(a) to child (ab)	Current node
(bc)	cab	bc:3, c:3	(ab) to suffix (b) to child (bc)	Current Node, Dict suffix node
(c)	ab	c:4	(bc) to suffix (c) to suffix () to child (c)	Current node
(ca)	b	a:5	(c) to child (ca)	Dict suffix node
(ab)		ab:6	(ca) to suffix (a) to child (ab)	Current node

Aho–Corasick Algorithm



- But how to build an Aho-Corasick automaton **efficiently**?
  - **Blue arcs** can be computed in **linear time** with a BFS from the root

    The target for the **blue arc** of a visited node can be found by following its parent's **blue arc** to its longest suffix node and searching for a child of the suffix node whose character matches that of the visited node.

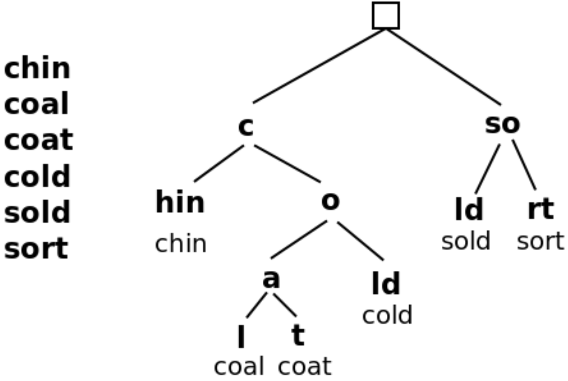
    If the character does not exist as a child, we can find the next longest suffix (following the **blue arc** again) and then search for the character.

    We can do this until we either find the character (as child of a node) or we reach the root (which will always be a suffix of every string).
  - The **green arcs** can be computed in **linear time** by repeatedly traversing **blue arcs** until a **blue node** is found (and **memoizing** this)

Compressed Tries

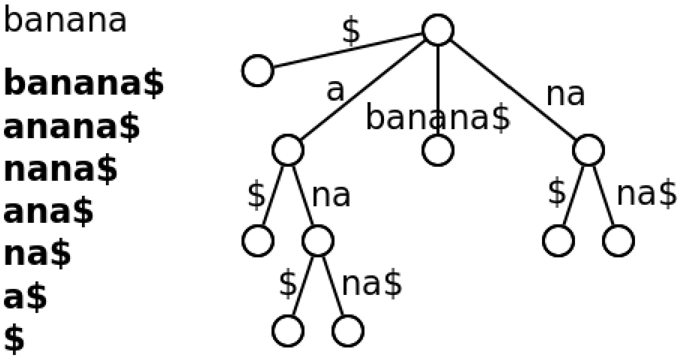
An example compressed prefix tree with 6 words

- Let's go back to **tries**!
- For **space efficiency** we can compact the tree: if a node has only one child, merge it with that child. This type of tree is called a **compressed prefix tree**, which is sometimes called **radix tree**



Suffix Tree

- A trie is not efficient in searching for **substrings** *(of our stored strings)*.
- For that we need a different data structure: a **suffix tree**. It is essentially a compressed trie of all suffixes of a given word.



$ is being used for marking the end of a word

Suffix Tree

- We can check if a string of size n is a **substring** in $\mathcal{O}(n)$ time
- A suffix tree of a word of size n can be **created** in $\mathcal{O}(n)$ time, but the algorithms have an high constant factor and are not trivial to implement (ex: Ukkonen's algorithm)
- We can put more than one word in the suffix tree: a **generalized suffix tree** is just is a suffix tree of a set of words.

Suffix Tree

There are many possible applications besides the "obvious" substring matching. Here are some examples:

- **Longest repeated substring** on a single word? Just find node with the highest depth through which two different suffixes passed by.
- **Longest common substring** of two words? Just put both on a suffix tree and find the node with the highest depth through which both strings passed by.
- **Most frequent k-gram**? (substring of size k) For all nodes with depth k, find which has more leafs descending from it.
- **Shortest unique substring**? Find the lowest depth node with only one leaf descending from it
- ...

Suffix Arrays

- The biggest problem with suffix trees is it **high memory usage**.
- A much more space efficient alternative with the same kind of applications is the **suffix array**: a sorted array of all suffixes

Consider **S="banana"**

i	1	2	3	4	5	6	7
s[i]	b	a	n	a	n	a	$

Suffixes of S

Suffix	i
banana$	1
anana$	2
nana$	3
ana$	4
na$	5
a$	6
$	7

Sorted Suffixes

Suffix	i
$	7
a$	6
ana$	4
anana$	2
banana$	1
na$	5
nana$	3

The suffix array A contains the starting positions of these sorted suffixes:

i	1	2	3	4	5	6	7
A[i]	7	6	4	2	1	5	3

- How to search if a string P is a substring of a text T?
- You can use **binary search** on the suffix array of T!
- Without auxiliary data structures each comparison takes $\mathcal{O}(|P|)$ and you need to make $\mathcal{O}(log|T|)$ comparisons, leading to an $\mathcal{O}(|P| \times log|T|)$ algorithm.

Suffix Arrays vs Suffix Trees

- Suffix arrays can be constructed by performing a depth-first traversal (DFS) of a suffix tree. The suffix array corresponds to the leaf-labels given in the order in which these are visited during the traversal, if edges are visited in the lexicographical order of their first character.
- A suffix tree can be constructed in linear time by using a combination of suffix arrays and **LCP array**
- In fact, **every suffix tree algorithm can be systematically replaced by an algorithm with suffix arrays** by using auxiliary information (such as the LCP array), having an **"almost equivalent" time complexity** (just a bit slower).

- What is the **LCP array**? LCP = Longest Common Prefix
  It stores the lengths of the longest common prefixes between pairs of consecutive suffixes in the sorted suffix array.

Consider **S="banana"**

i	1	2	3	4	5	6	7
s[i]	b	a	n	a	n	a	$

Suffix Array A:

i	1	2	3	4	5	6	7
A[i]	7	6	4	2	1	5	3

LCP Array H

i	1	2	3	4	5	6	7
A[i]	_	0	1	3	0	0	2

Example: $H[4] = 3$ because ana and anana have a common prefix of size 3

- How can we use the LCP array?
- Imagine again you want to check if a string P is a substring of T.
- You can use binary search on the suffix array of T
- Without anything else we can use binary search in $\mathcal{O}(|P| \times log|T|)$
- With LCP and derivatives you can turn this into $\mathcal{O}(|P| + log|T|)$
- Consider an LCP-LR array that tells you the longest common prefix of any given suffixes (not necessarily consecutive).
- We can use LCP-LR to only check the "new characters". How?

Suffix Arrays and Binary Search

- During the binary search we consider a range $[L, R]$ and its central point M. We then decide whether to continue with the left half $[L, M]$ or the right half $[M, R]$.
- For that decision, we compare P to the string at position M. If $P == M$, we are done. If not, we have compared the first k chars of P and then decided whether P is lexicographically smaller or larger than M. Let's assume the outcome is that P is larger than M.
- In the next step we will therefore consider $[M, R]$ and a new central point M' in the middle:

```
          M ...... M' ...... R
          |
   we know:
      lcp(P,M)==k
```

Suffix Arrays and Binary Search

```
      M ...... M' ...... R
      |
   lcp(P,M)==k
```

- The "trick" now is that LCP-LR is precomputed such that a $\mathcal{O}(1)$ lookup gives the longest common prefix of M and M', lcp(M,M').
- We know already that M itself has a prefix of k chars common with P: $lcp(P, M) = k$. Now there are 3 possibilities:
  - $\mathrm{k} < \mathrm{lcp}(\mathrm{M}, \mathrm{M}')$. This means the (k+1)-th char of M' is the same as M. Since P is lexicographically larger than M, it must be lexicogr. larger than M', too. We continue in the right half $[M', R]$
  - $\mathrm{k} > \mathrm{lcp}(\mathrm{M}, \mathrm{M}')$. the common prefix of P and M' would be $< k$, and M' would be lexicographically larger than P, so, without actually making the comparison, we continue in the left half $[M, M']$
  - $\mathrm{k} == \mathrm{lcp}(\mathrm{M}, \mathrm{M}')$. M and M' have the same first k chars as P. It suffices to compare P to M' starting from the $(k+1)$-th char.

Suffix Arrays and Binary Search

- In the end every character of P is compared to any character of T only **once**!
- We get our desired $\mathcal{O}(|P| + log|T|)$ complexity!
- But how to build the LCP-LR array?
  - Only certain ranges may appear during a binary search
  - In fact, every entry of the suffix array is the central point of exactly one possible range
  - So there are $|T|$ distinct ranges, and it suffices to compute $lcp(L, M)$ and $lcp(M, R)$ for those ranges
  - In the end we have $2 \times |T|$ values to pre-compute
  - There is a "straightforward" recursive algorithm to compute the $2 \times |T|$ values of LCP-LR in $\mathcal{O}(|T|)$ from the standard LCP array.