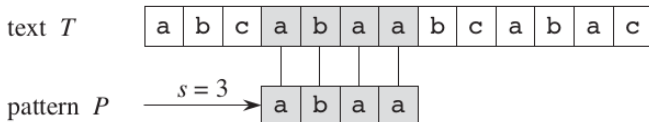


# String Matching

Pedro Ribeiro

DCC/FCUP

2022/2023



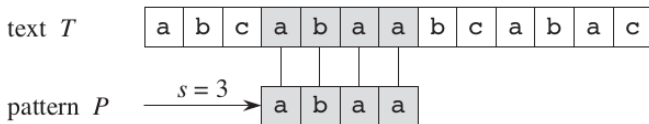
# String Problems

- 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 we will just give a brief glimpse on the whole field and in particular we will focus on the **string matching problem**

# 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  $\Sigma$ 
  - ▶ For example, we could have  $\Sigma = \{0, 1\}$  or  $\Sigma = \{a, b, \dots, z\}$
- 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

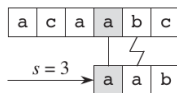
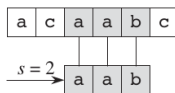
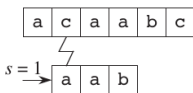
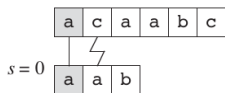
# Naive String Matching

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

NAIVE-STRING-MATCHER( $T, P$ )

```
1  $n = T.length$ 
2  $m = P.length$ 
3 for  $s = 0$  to  $n - m$ 
4     if  $P[1..m] == T[s + 1..s + m]$ 
5         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



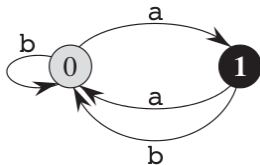
# Naive String Matching

- 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.

state	input	
	a	b
0	1	0
1	0	0



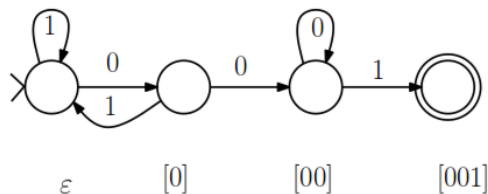
An example DFA that matches strings of  $\Sigma = \{a, b\}$  finishing with an odd number of  $a$ 's

# Deterministic Finite Automaton

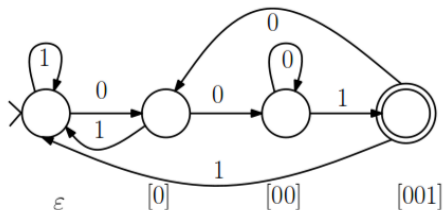
- Imagine a DFA with  $m + 1$  states, arranged in a "line"
- The  $i$ -th state represents that we are now at position  $i$  of the pattern, that is, we matched the first  $i$  characters.
- Now, if we match the next character, we move to state  $i + 1$  (matched  $i + 1$  characters). If not, we can 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:



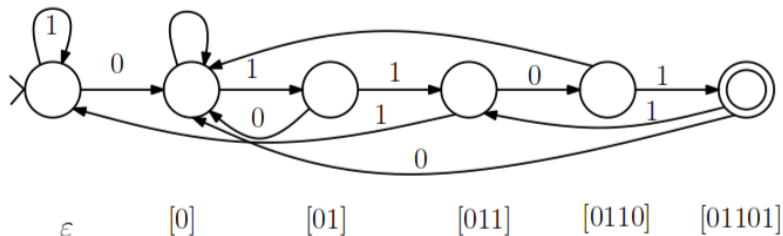
This would only find the first occurrence of  $P$ . What to change so that it finds all occurrences?





# Deterministic Finite Automaton

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



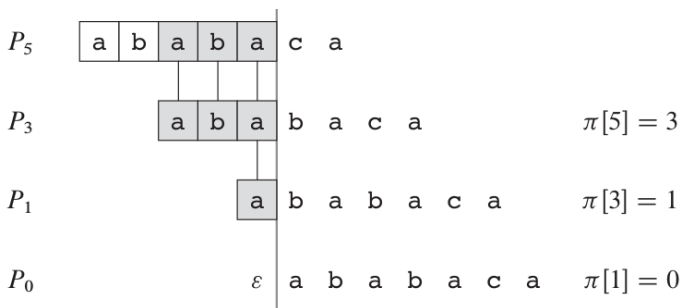
# Deterministic Finite Automaton

- 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

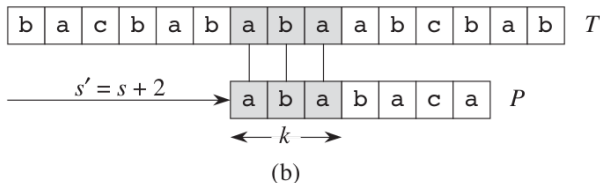
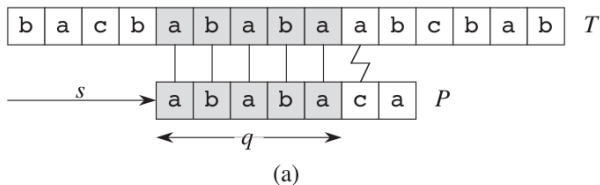
- Let  $\pi[i]$  be the largest integer smaller than  $i$  such that  $P[1..\pi[i]]$  (longest prefix) is a suffix of  $P[1..i]$ .

$i$	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 we have a mismatch at position  $i + 1 \dots$  we rollback to  $\pi[i]!$ 
  - ▶ This is the next possible "largest" partial match



# Knuth-Morris-Pratt Algorithm

- Let us look at the KMP main algorithm:

KMP-MATCHER( $T, P$ )

```
1   $n = T.length$ 
2   $m = P.length$ 
3   $\pi = \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q = 0$  // number of characters matched
5  for  $i = 1$  to  $n$  // scan the text from left to right
6      while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7           $q = \pi[q]$  // next character does not match
8      if  $P[q + 1] == T[i]$ 
9           $q = q + 1$  // next character matches
10     if  $q == m$  // is all of  $P$  matched?
11         print "Pattern occurs with shift"  $i - m$ 
12          $q = \pi[q]$  // 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  $\pi$ .
- 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, then 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  $\pi$ ), 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**:  $\mathcal{O}(n)$

# Knuth-Morris-Pratt Algorithm

- What about computing  $\pi$ ?
- It is basically comparing the pattern against itself!

COMPUTE-PREFIX-FUNCTION( $P$ )

```
1   $m = P.length$ 
2  let  $\pi[1..m]$  be a new array
3   $\pi[1] = 0$ 
4   $k = 0$ 
5  for  $q = 2$  to  $m$ 
6      while  $k > 0$  and  $P[k + 1] \neq P[q]$ 
7           $k = \pi[k]$ 
8      if  $P[k + 1] == P[q]$ 
9           $k = k + 1$ 
10      $\pi[q] = k$ 
11 return  $\pi$ 
```

- 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:  $\mathcal{O}(n)$



# Rabin-Karp Algorithm

- Let's now look at a completely different approach
- Imagine that we have an **hash function**  $h$  that maps each possible string to an integer.
- We could then proceed as follows:
  - ▶ Start by computing  $h(P)$
  - ▶ For every possible shift  $s$ , compute  $h_i = h(T[s + 1 \dots s + m])$
  - ▶ If  $h_i \neq h(P)$  then we know we do not have a match
  - ▶ If  $h_i = h(P)$  we could have a match, and we loop to see if its really a match on that position
- The efficiency of this procedure depends mainly on two things:
  - ▶ How good is the hash function (how well does it separate strings), because some invalid shifts may not be filtered out
  - ▶ How many valid occurrences exist, because for each of these shifts we will really make a loop of at most  $m$

# Rabin-Karp Algorithm

- Let's actually create a procedure using these core ideas
- We will start by defining a suitable **rolling hash function**.
- Suppose each character is assigned an integer. For ease of explanation, we will show examples only with digits (0..9) and a decimal base, but if we have  $k = |\Sigma|$  characters, we could use base  $k$ .
- A pattern of a  $k$ -sized alphabet can be seen as a number on base  $k$ . With our simple scheme for digits, the pattern "12345" could then be viewed as the number 12,345. Let's call this function *value*.

# Rabin-Karp Algorithm

- We can compute the value of the pattern in time  $\mathcal{O}(m)$ :

$$\text{value}(P) = P[m] + 10(P[m-1] + 10(P[m-2] + \dots + 10(P[2] + 10P[1])\dots))$$

Example:  $\text{value}(\text{"324"}) = 4 + 10(2 + 10 \times 3) = 324$

- Similarly, if  $T_i = T[i+1..i+m]$ , we can compute  $\text{value}(T_i)$  in  $\mathcal{O}(m)$
- After we compute  $T_0$ , do we really need  $m$  operations to compute  $T_1$ ? No! We can do it in constant time:

$$\text{value}(T_{s+1}) = 10(T_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

Example:  $\text{value}(\text{"5678"}) = 5,678$

$$\text{value}(\text{"6789"}) = 10(5,678 - 10^3 \times 5) + 9 = 6,789$$

- This means we can compute all  $T_i$ 's in time linear to the size of the text!

# Rabin-Karp Algorithm

- If we ignore the fact that our  $value()$  could get really large, we would have an  $\mathcal{O}(n)$  algorithm for doing string matching
- The problem is that we cannot assume that the  $m$  characters of  $P$  will give origin to arithmetic operations that take constant time.

- How can we solve this problem? Consider that we know that:

$$(a \times b) \bmod c = ((a \bmod c) \times (b \bmod c)) \bmod c$$

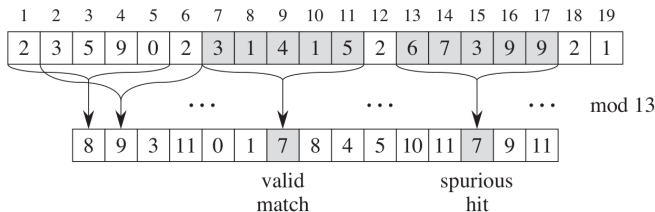
$$(a + b) \bmod c = ((a \bmod c) + (b \bmod c)) \bmod c$$

- What we can do is always apply **mod q** operation on our results! In that way the value will always stay between 0 and  $q - 1$ !

$$value(T_{s+1}) = (10(T_s - 10^{m-1} T[s + 1]) + T[s + m + 1]) \bmod q$$

# Rabin-Karp Algorithm

- The solution with **mod  $q$**  is not perfect, however...
  - ▶  $value(T_s) \bmod q = value(P) \bmod q$  does not imply  $T_s = P$
  - ▶ However,  $value(T_s) \bmod q \neq value(P) \bmod q$ , implies that  $T_s \neq P$
- If the values are equal **mod  $q$**  we still have to test to see if we have a match or not. On case it is not a match we have a **spurious hit**.
- Example: imagine we are looking for 31,415 and use  $q = 13$   
We have that  $31,415 \bmod 13 = 7$ .



# Rabin-Karp Algorithm

- Our  $value()$  function is in reality just a fast *heuristic* for ruling out invalid shifts.
- If  $q$  is high enough, we *hope* that the spurious hits will be rare

RABIN-KARP-MATCHER( $T, P, d, q$ )

```
1   $n = T.length$ 
2   $m = P.length$ 
3   $h = d^{m-1} \bmod q$ 
4   $p = 0$ 
5   $t_0 = 0$ 
6  for  $i = 1$  to  $m$            // preprocessing
7       $p = (dp + P[i]) \bmod q$ 
8       $t_0 = (dt_0 + T[i]) \bmod q$ 
9  for  $s = 0$  to  $n - m$        // matching
10     if  $p == t_s$ 
11         if  $P[1..m] == T[s + 1..s + m]$ 
12             print "Pattern occurs with shift"  $s$ 
13     if  $s < n - m$ 
14          $t_{s+1} = (d(t_s - T[s + 1])h + T[s + m + 1]) \bmod q$ 
```

# Rabin-Karp Algorithm

- How to analyze the running time?
- What would the **worst case be**? Imagine a string always with the same characters, and a pattern also with the same characters. In that case we will always have a hit and will always be making the verification.
- In many applications, however, the valid shifts are rare. In those cases this may be a good choice.
- If we have only  $c$  occurrences, than the expected time will be  $\mathcal{O}(n + cm)$ , plus the time for the spurious hits.

# Rabin-Karp Algorithm

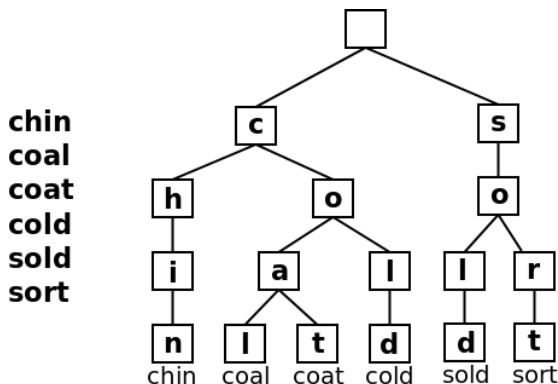
- How often do spurious hit occur? How good is our hash function?  
This is not going to be explored today, but choosing a (large) prime not close to a power of two is a good choice.
- If we are able to really spread the possible values, and the text is "random", than the number of expected spurious hits is  $\mathcal{O}(n/q)$  (the chance that an arbitrary substring has the same value of  $P$  is  $1/q$ ).
- If  $v$  is the number of valid shifts, then the running time is  $\mathcal{O}(n + m(v + n/q))$ .
- If  $v$  is  $\mathcal{O}(1)$  and  $q > m$  then the total expected running time is  $\mathcal{O}(n)$ !



- From algorithms revolving around the pattern, we will now focus on data-structures centered on the text (or 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

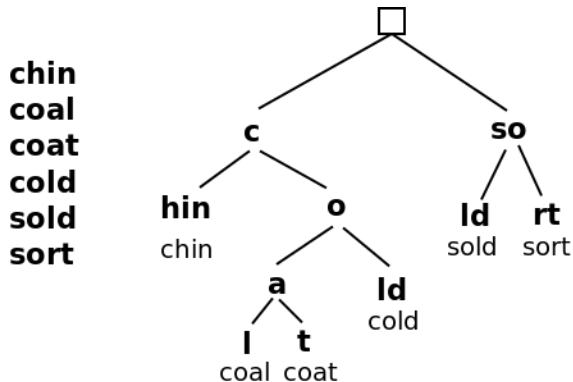


Note that the letters can be thought of as the edges and not the nodes

- 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)
- 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**

# Compressed Prefix Tree

An example compressed prefix tree with 6 words



# Suffix Tree

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

banana

**banana\$**

**anana\$**

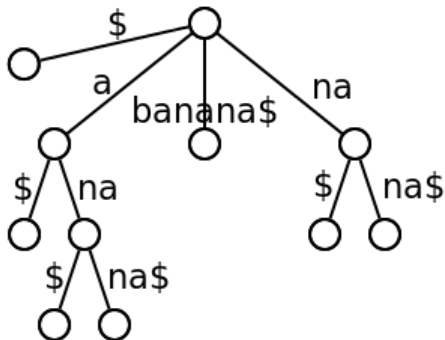
**nana\$**

**ana\$**

**na\$**

**a\$**

**\$**



*\$ is being used for marking the end of a word*

- 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 a 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 a suffix tree of a set of words.

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
- ...

- The biggest problem with suffix trees is its **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



# Suffix Arrays

## An example

Consider  $S = \text{"banana"}$

<b>i</b>	1	2	3	4	5	6	7
<b>s[i]</b>	b	a	n	a	n	a	\$

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

Suffixes of S

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

Sorted Suffixes

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

<b>i</b>	1	2	3	4	5	6	7
<b>A[i]</b>	7	6	4	2	1	5	3

# Suffix Arrays

## Substring matching

- 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 **"equivalent" time complexity** (just a bit slower).

# Suffix Arrays

## LCP Array

- 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"**

<b>i</b>	1	2	3	4	5	6	7
<b>s[i]</b>	b	a	n	a	n	a	\$

Suffix Array **A**:

<b>i</b>	1	2	3	4	5	6	7
<b>A[i]</b>	7	6	4	2	1	5	3

**LCP Array H**

<b>i</b>	1	2	3	4	5	6	7
<b>A[i]</b>	-	0	1	3	0	0	2

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

# Suffix Arrays

## LCP Array

- 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:

$\text{lcp}(P, M) == k$

# Suffix Arrays and Binary Search

$$\begin{array}{c} M \dots\dots M' \dots\dots R \\ | \\ \text{lcp}(P, M) = k \end{array}$$

- The "trick" now is that LCP-LR is precomputed such that a  $\mathcal{O}(1)$  lookup gives the longest common prefix of  $M$  and  $M'$ ,  $\text{lcp}(M, M')$ .
- We know already that  $M$  itself has a prefix of  $k$  chars common with  $P$ :  $\text{lcp}(P, M) = k$ . Now there are 3 possibilities:
  - ▶  $k < \text{lcp}(M, M')$ . 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  $[M', R]$
  - ▶  $k > \text{lcp}(M, M')$ . 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  $[M, M']$
  - ▶  $k == \text{lcp}(M, M')$ .  $M$  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.