1 Fast string matching

We will discuss

- "Online" methods
  - Shift-And/Shift-Or
  - Horspool
- "Index-based" methods
  - Suffix trees
  - Suffix arrays

This exposition is based on earlier versions of this lecture and the following sources, which are all recommended reading:

- Shift-And/Shift-Or, Horspool
  1. Flexible Pattern Matching in Strings, Navarro, Raffinot, 2002, pages 15ff.
  2. A nice overview of the plethora of string matching algorithms with implementations can be found under `http://www-igm.univ-mlv.fr/~lecroq/string`.
- Suffix trees, suffix arrays
  1. Dan Gusfield: Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997, pages 94ff. ISBN 0-521-58519-8

1.1 Thoughts about string matching

We will get to know several string matching algorithms for a single pattern (two of which are very practical) and learn the basics about *suffix trees* and *suffix arrays*, two central data structures in computational biology.

Let's start with the classical string matching problem:

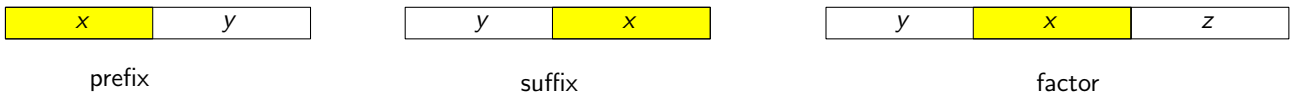
The task at hand is to find all occurrences of a given pattern $p = p_1, \ldots, p_m$ in a text $T = t_1, \ldots, t_n$ usually with $n \gg m$.

The algorithmic ideas of exact string matching are useful to know, although in computational biology algorithms for *approximate* string matching, or *indexed* methods are of more use. However, in online scenarios it is often not possible to precompute an index for finding exact matches.

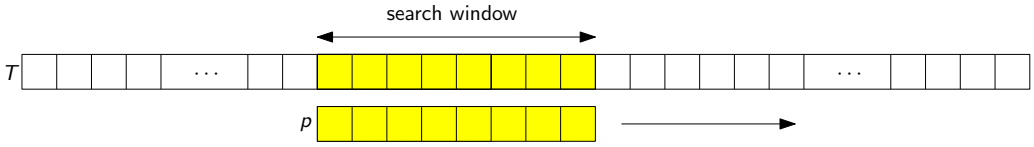
String matching is known for being amenable to approaches that range from the extremely theoretical to the extremely practical.

One example is the famous *Knuth-Morris-Pratt* algorithm which is in practice twice as slow as the brute force algorithm, and the well-known *Boyer-Moore* algorithm which, in its original version, has a worst case running time of $O(nm)$ but is quite fast in practice.

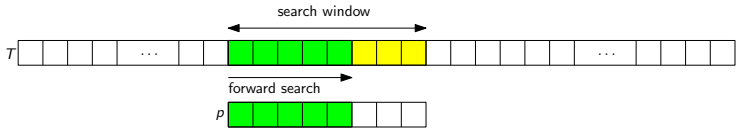
Some easy terminology: Given strings x, y, and z, we say that x is a *prefix* of xy, a *suffix* of yx, and a *factor* (:=substring) of yxz.



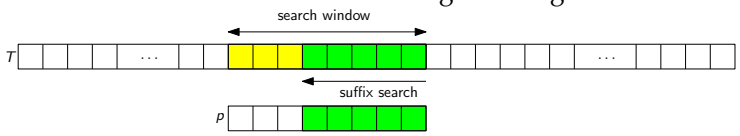
In general, string matching algorithms follow three basic approaches. In each a *search window* of the size of the pattern is slid from left to right along the text and the pattern is searched within the window. The algorithms differ in the way the window is shifted.



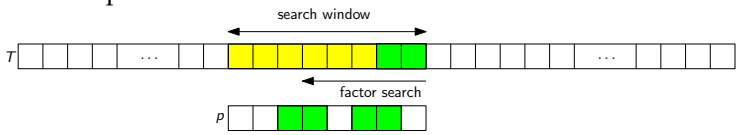
1. *Prefix searching*. For each position of the window we search the longest prefix of the window that is also a prefix of the pattern.



2. *Suffix searching*. The search is conducted backwards along the search window. On average this can avoid to read some characters of the text and leads to sublinear average case algorithms.



3. *Factor searching*. The search is done backwards in the search window, looking for the longest suffix of the window that is also a factor of the pattern.



1.2 Prefix based approaches

Suppose we have read the text up to position i and that we know the length of the longest suffix p' of the text read that corresponds to a prefix of p. If $|p'| = |p|$ we have found an occurrence, otherwise we have to shift the search window in a *safe* way.

The main algorithmic problem is to efficiently compute this length when reading the next character. There are two classical ways:

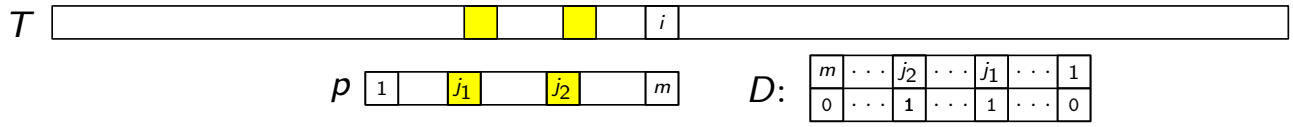
1. compute the longest suffix of the text read that is also a prefix of p and for which the next character is different. (used by the *Knuth-Morris-Pratt* algorithm). This insures that the implied shift is safe and allows the detection of a match.

2. maintain the set of *all* prefixes of p that are also suffixes of the text read and update the set when reading a character (e.g., the Shift-And algorithm).

1.3 Shift-And and Shift-Or

We keep the following *invariant*:

There is a 1 at the j-th position of D if and only if $p_1, \ldots, p_j$ is a suffix of $t_1, \ldots, t_i$.



If the size of p is less than the word length, this array will fit into a computer register.

When reading the next character t_{i+1}, we have to compute the new set D'. We use the following fact:

Observation. A position $j+1$ in the set D' will be active if and only if

1. the position j was active in D, that is, $p_1, \ldots, p_j$ was a suffix of $t_1, \ldots, t_i$, and
2. t_{i+1} matches p_{j+1}.

The algorithm builds a table B which stores a bit mask $b_m, \ldots, b_1$ for each character of Σ. The mask $B[c]$ has the j-th bit set if $p_j = c$.

Example. $p = \texttt{atat}$, $\Sigma = \{\texttt{c}, \texttt{g}, \texttt{t}, \texttt{a}\}$
$B[c] = \texttt{0000}$, $B[g] = \texttt{0000}$, $B[t] = \texttt{1010}$, $B[a] = \texttt{0101}$

Initially we set $D = \texttt{0}^m$ and for each new character t_{pos} we update D using the formula

$$D' = ((D \ll 1) \mid \texttt{0}^{m-1}\texttt{1}) \;\&\; B[t_{pos}] \quad .$$

This update maintains our invariant using the above observation.

The shift operation marks all positions as potential prefix-suffix matches that were such matches in the previous step (notice that this includes the empty string ε). In addition, to stay a match, the character t_{pos} has to match p at those positions. This is achieved by applying an &-operation with the corresponding bitmask $B[t_{pos}]$.

1.4 Shift-And Pseudocode

(1) **Input:** text T of length n and pattern p of length m
(2) **Output:** all occurences of p in T

(4) *Preprocessing:*
(5) **for** $c \in \Sigma$ **do** $B[c] = \texttt{0}^m$;
(6) **for** $j \in 1 \ldots m$ **do** $B[p_j] = B[p_j] \mid \texttt{0}^{m-j}\texttt{1}\texttt{0}^{j-1}$;
(7) *Searching:*
(8) $D = \texttt{0}^m$;
(9) **for** $pos \in 1 \ldots n$ **do**

(10) $D = ((D \ll 1) \mid \mathtt{0}^{m-1}\mathtt{1}) \;\&\; B[t_{pos}]$;
(11) **if** $D \;\&\; \mathtt{10}^{m-1} \neq \mathtt{0}^m$ **then output** "p occurs at position $pos - m + 1$";

1.5 Shift-And and Shift-Or

So what is the Shift-Or algorithm about? It is just an implementation trick to avoid a bit operation, namely the $\mid \mathtt{0}^{m-1}\mathtt{1}$ in line 10.

In the Shift-Or algorithm we complement all bit masks of B and use a complemented bit mask D. Now the $\ll$ operator will introduce a `0` to the right of D' and the new suffix stemming from the empty string is already in D'. Obviously we have to use a bit | instead of an & and report a match whenever $d_m = \mathtt{0}$.

Let's look at an example of Shift-And and Shift-Or.

1.6 Example

Find all occurrences of the pattern $p =$ `atat` in the text $T =$ `atacgatatata`.[1]

```
          Shift-And         Shift-Or
          B[a] = 0101       B[a] = 1010
          B[t] = 1010       B[t] = 0101
          B[*] = 0000       B[*] = 1111

          D    = 0000       D    = 1111

1 Reading a
           0001               1110
           0101               1010
           ----               ----
           0001               1110


2 Reading t         3 Reading a
0011     1100        0101     1010
1010     0101        0101     1010
-------------        -------------
0010     1101        0101     1010

4 Reading c          5 Reading g
1011     0100        0001     1110
0000     1111        0000     1111
-------------        -------------
0000     1111        0000     1111


6 Reading a          7 Reading t
0001     1110        0011     1100
0101     1010        1010     0101
-------------        -------------
0001     1110        0010     1101
```

[1] **Note:** The bit order is reversed in the Java applet on `http://www-igm.univ-mlv.fr/~lecroq/string`.

```
8 Reading a          9 Reading t
0101     1010        1011     0100
0101     1010        1010     0101
-------------        -------------
0101     1010        1010     0101
```

Hence, in step 9, we found the first occurrence of p at position $9 - 4 + 1 = 6$.

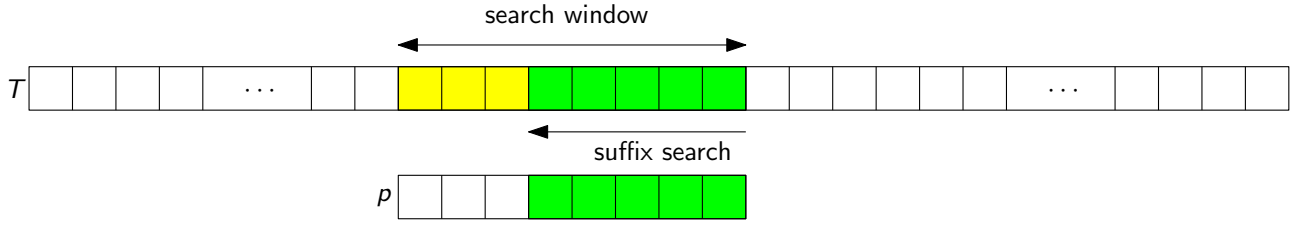
The running time of the algorithms is $O(n)$, assuming

1. a constant alphabet size
2. the operations on D can be done in constant time.

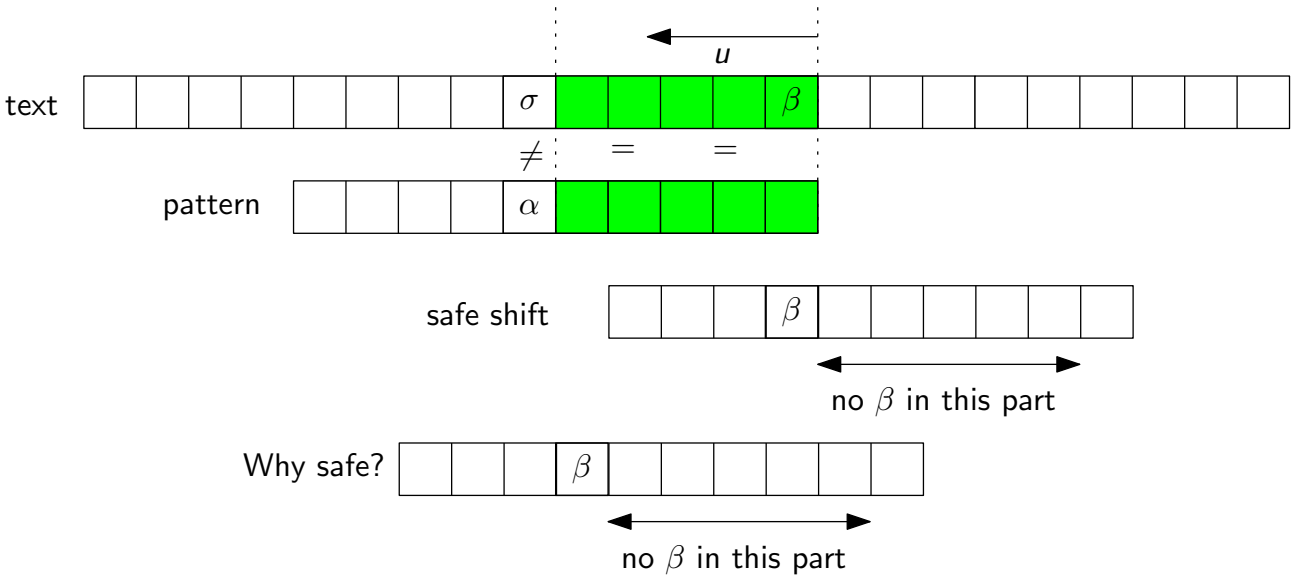
1.7 Suffix based approaches

As noted above, in the suffix based approaches we match the characters from the back of the search window. Whenever we find a mismatch we can shift the window in a safe way, that means without missing an occurrence of the pattern.

We present the idea of the *Horspool* algorithm, which is a simplification of the *Boyer-Moore* algorithm.



For each position of the search window we compare the last character β with the last character of the pattern. If they match we verify until we find the pattern or fail on the text character σ. Then we simply shift the window according to the next occurrence of β in the pattern.



Input: text T of length n and pattern p of length m
Output: all occurences of p in T

Preprocessing:
for $c \in \Sigma$ **do** $d[c] = m$;
for $j \in 1 \ldots m-1$ **do** $d[p_j] = m - j$
Searching:
$pos = 0$;
while $pos \leq n - m$ **do**
    $j = m$;
    while $j > 0 \ \wedge \ t_{pos+j} = p_j$ **do** $j{-}{-}$;
    if $j = 0$ **then output** "p occurs at position $pos + 1$"
    $pos = pos + d[t_{pos+m}]$;

We notice two things:

1. The verification could also be done forward. Many implementations use built-in memory comparison instructions of the machines (i. e. `memcmp`). (The java applet also compares forward.)
2. The main loop can be "unrolled", which means that we can first shift the search window until its last character matches the last character of the pattern and then perform the verification.

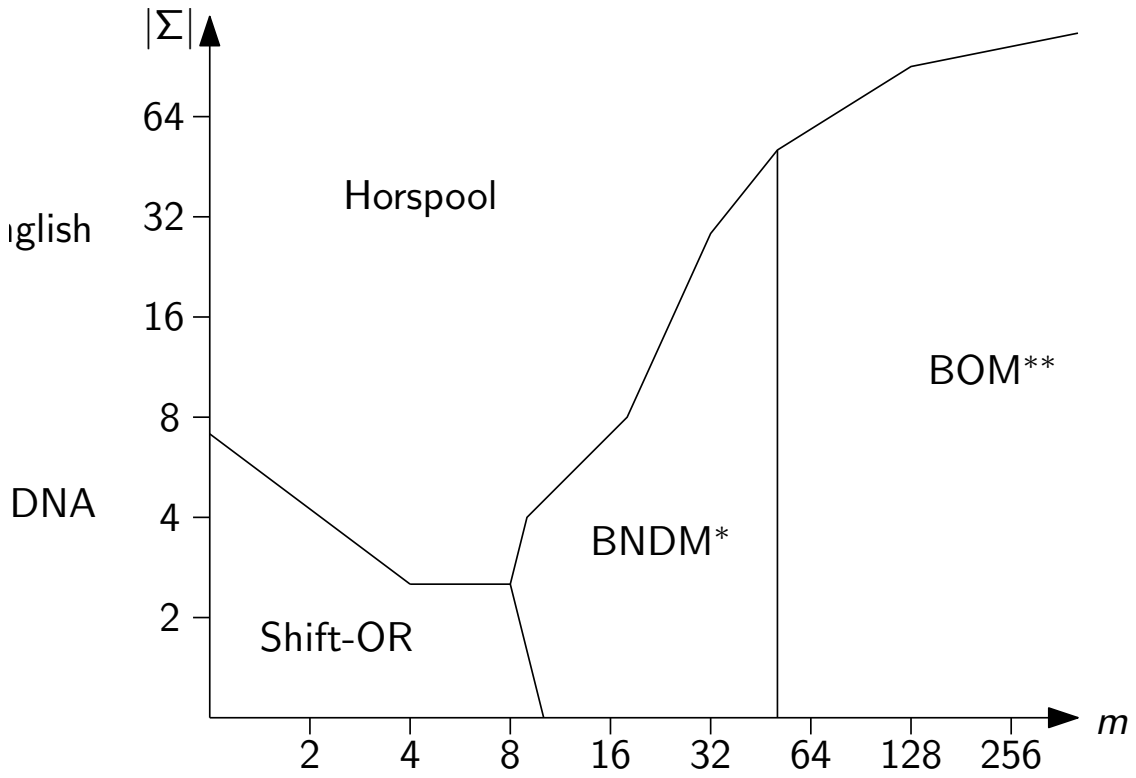
1.8 Horspool example

We search for the string announce in the text CPM_annual_conference_announce.

```
m=8,  d = a c n o u *
          7 1 2 4 3 8

1)  | CPM_annu | al_conference_announce
   u != e, d[u]=3

2) CPM | _annual_ | conference_announce
   _ != e, d[_] = 8

3) CPM_annual_ | conferen | ce_announce
   n != e, d[n] = 2


4) CPM_annual_co | nference | _announce
   e == e => verify  until it fails with e != u => d[e] = 8

5) CPM_annual_conference | _announc | e
   c != e, d[c]=1

6) CPM_annual_conference_ | announce
   e == e  =>  verify  until occurrence is found
```

1.9 Experimental Map

(Gonzalo Navarro & Mathieu Raffinot, 2002)



ondeterministic DAWG Matching algorithm, factor-based, not covered in this lecture
Oracle Matching, factor-based, not covered in this lecture

1.10 Introduction

Exact string matching is used in many algorithms in computational biology as a first step:

> Given a pattern $p = p[1 \dots m]$, find all j occurrences of p in a text $T = T[1 \dots n]$.

This can readily be done with string matching algorithms in time $O(m + n)$. If however, the text is very long, we would prefer not to scan it completely for every query, but rather spend time $O(m + j)$ per query.

To do that we have to preprocess the *text*. The preprocessing step is especially useful in scenarios where the text is relatively constant over time (e. g., a genome), and we will search for many different patterns.

In this lecture we first introduce such a preprocessing, namely the construction of a *suffix tree*. Later we get to know the related, more space-efficient, *suffix array*.

Both are central data structures in computational molecular biology.

This is just a brief introduction, and we will skip some important topics. These will, however, be covered in detail in future classes.

History.

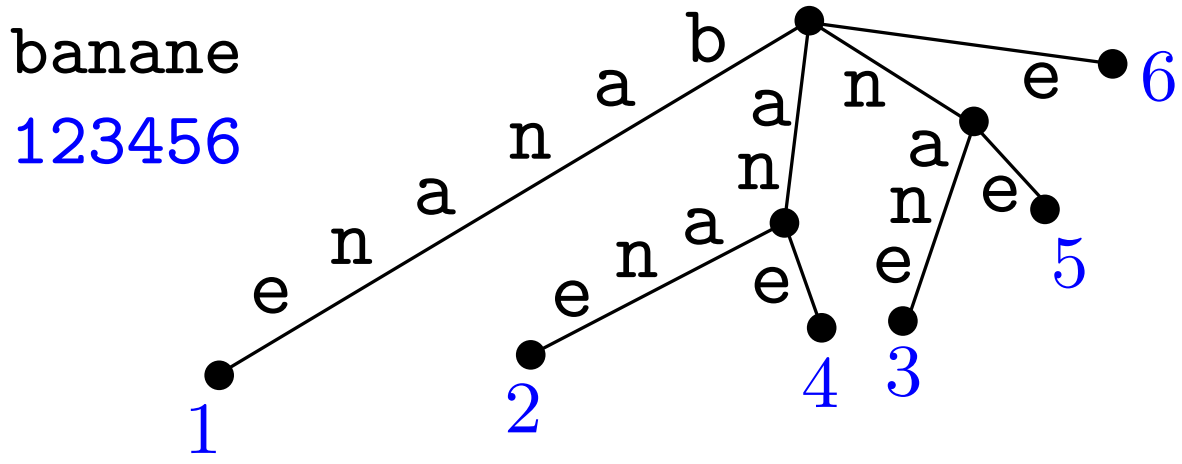
- Weiner, 1973: suffix trees introduced, linear-time construction algorithm, "algorithm of the year"
- McCreight, 1976: reduced space-complexity
- Ukkonen, 1995: new algorithm, easier to describe

In this lecture, we will only cover a naïve (quadratic-time) construction.

1.11 Suffix trees

Definition. Let $T = T[1 \,..\, n]$ be a text of length n over a fixed alphabet Σ. A *suffix tree* for T is a tree with n numbered leaves and the following properties:

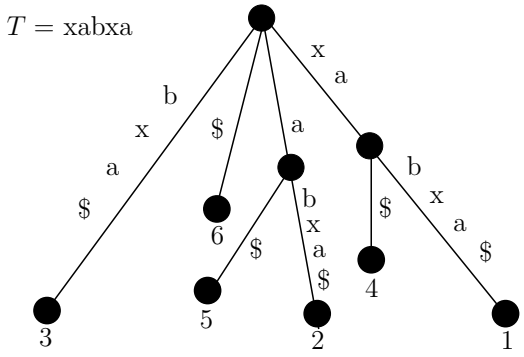
1. Every internal node other than the root has at least two children.
2. Every edge is labeled with a nonempty substring of T.
3. The edges leaving a given node have labels starting with different letters.
4. The concatenation of the labels of the path from the root to leaf i spells out the i-th suffix $T[i \ldots n]$ of T. We denote $T[i \ldots n]$ by T_i.



1.12 Marking the end of T

Note that according to the above definition there is no suffix tree if a suffix of T is a prefix of another suffix of T. For example for $T =$ `xabxa` there is no leaf for the fourth suffix T_4.

This problem is easily overcome by adding a special letter `$` to the alphabet which does not occur in T, and putting it to the end of T. This way no suffix can be a prefix of another suffix.



The above figure shows a suffix tree for the string `xabxa$`.

1.13 Storing the edge labels efficiently

What about the space consumption? The total length of all edge labels in a suffix tree can easily be $\Omega(n^2)$, e.g., for $T = abc \cdots xyz$.

Therefore, we do not store the substrings $T[i \dots j]$ of T in the edges, but only their start and end indices (i, j). Nevertheless we keep thinking of the edge labels as substrings of T.

1.14 A naïve algorithm for suffix tree construction

The *naïve algorithm* for constructing a suffix tree is as follows:

We insert the suffixes $T_1, T_2, \dots, T_n$ (in this order) and modify the tree according to the definition.

1. Say we want to insert T_i into the current tree. We read the letters of T_i and walk down the path from the root accordingly, until a mismatch occurs. At this point we have to branch off a new edge.
2. If the mismatch occurs in the midst of an edge, then we split it into two edges and branch off from the inserted vertex.
3. Otherwise we can branch off from a vertex which is already present.

We need time $O(n - i + 1)$ for the i-th suffix and hence the total running time is

$$\sum_{1}^{n} O(n - i + 1) = \sum_{1}^{n} O(i) = O(n^2) \ .$$

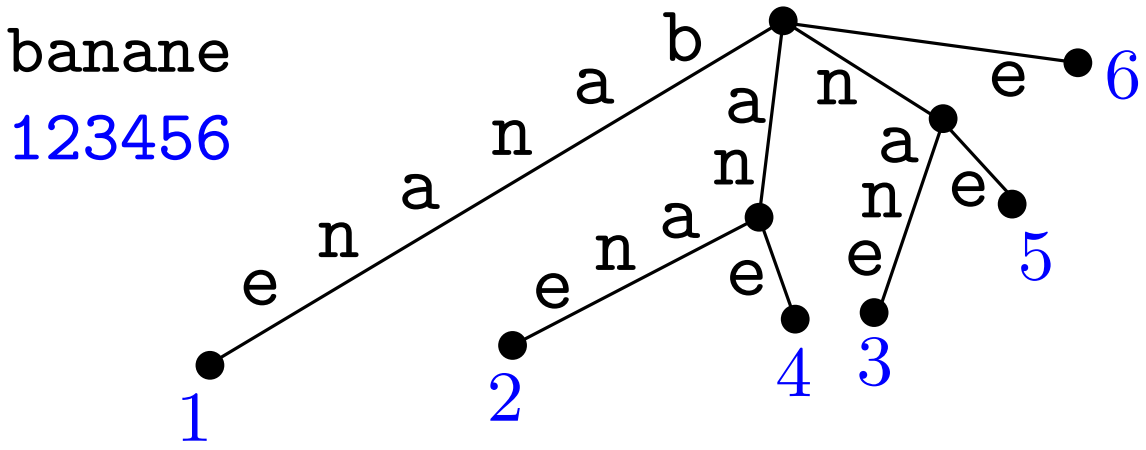
1.15 Searching with suffix trees

Given T and p. How do we find all occurrences of p in T?

Observation. Each occurrence has to be a prefix of some suffix. Each such prefix corresponds to a path starting at the root.

1. Of course, as a first step, we construct the suffix tree for T. Using the naïve method this takes quadratic time, but *linear-time* algorithms (e.g., Ukkonen's algorithm) exist.
2. Try to match p on a path, starting from the root. Three cases:
   (a) The pattern does not match $\rightarrow$ p does not occur in T
   (b) The match ends in a node u of the tree. Set $x = u$.
   (c) The match ends inside an edge (v, w) of the tree. Set $x = w$.
3. All leaves below x represent occurrences of p.

Example. (completion by blackboard)



Theorem. We can find all j occurrences of a pattern $p = p[1 \ldots m]$ in a suffix tree in time $O(m + j)$.

Proof.

- Finding x takes time $O(m)$.
- Collecting the j leaves (e.g., using depth-first search) takes time $O(j)$.[2]
- Together, this yields $O(m + j)$.

[2] Why? How many nodes does the subtree have?

1.16 Suffix arrays

Suffix arrays were introduced by Manber and Myers in 1989 (and published in 1993).

While both suffix trees and suffix arrays require $O(n)$ space, suffix arrays are more space efficient. A recent suffix tree implementation requires 15-20 Bytes per character. For suffix arrays, as few as 5 bytes are sufficient (with some tricks).

This is offset by a moderate increase in search time from $O(m+j)$ to $O(m+j+\log n)$. In practice this increase is counterbalanced by better cache behavior.

Definition. Given a text T of length n, the *suffix array* for T, called *suftab*, is an array of integers of range 1 to $n+1$ specifying the lexicographic ordering of the $n+1$ suffixes of the string $T\$$.

```
mississippi$
123456789012
```

i	T_i	$T_{\text{suftab}[i]}$	$\text{suftab}[i]$	
1	`mississippi$`	`$`	12	
2	`ississippi$`	`i$`	11	
3	`ssissippi$`	`ippi$`	8	
4	`sissippi$`	`issippi$`	5	
5	`issippi$`	`ississippi$`	2	
6	`ssippi$`	`mississippi$`	1	
7	`sippi$`	`pi$`	10	
8	`ippi$`	`ppi$`	9	
9	`ppi$`	`sippi$`	7	
10	`pi$`	`sissippi$`	4	← $\text{suftab}(10) = 4$ means: T_4 is number 10 in the sorted list.
11	`i$`	`ssippi$`	6	
12	`$`	`ssissippi$`	3	

We will assume that n fits into 4 bytes of memory. Then the basic form of a suffix array needs only $4n$ bytes. The suffix array can be computed by sorting the suffixes, as illustrated in the above example.

1.17 Why another algorithm?

The suffix array can be constructed in $O(n^2 \log n)$ time by sorting the suffix indices using a sorting algorithm[3]. But such an approach fails to take advantage of the fact that we are sorting a collection of related suffixes. We cannot get an $O(n)$ time algorithm in this way.

Alternatively, we could first build a suffix tree in linear time, then transform the suffix tree into a suffix array *in linear time**, and finally discard the suffix tree. Of course, sufficient memory has to be available to construct the suffix tree. Thus this approach fails for large texts.

In this lecture, we will *not* discuss the original construction proposed by Manber and Myers which needs (essentially) $8n$ bytes and runs in $O(n \log n)$ time.

[3] How?

1.18 Searching

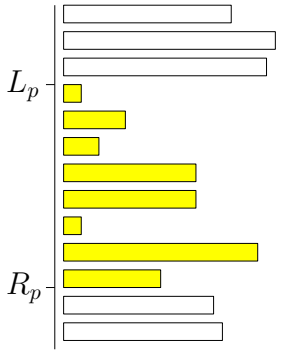
After constructing our suffix array we have the table *suftab*, which gives us the suffixes of T in sorted order. Suppose now we want to find all j instances of a pattern $p = p_1, \ldots, p_m$ of length $m < n$ in T.

Then let

$$L_p = \min\{k \mid p \leq T_{suftab[k]} \text{ or } k = n\}$$

and

$$R_p = \max\{k \mid p \geq T_{suftab[k]}[1 \ldots m] \text{ or } k = -1\} \ .$$



Since *suftab* is ordered, it follows that p matches a suffix T_i if and only if $i = suftab[k]$ for some $k \in [L_p, R_p]$. Hence a simple binary search can find L_p and R_p. Each comparison in the search needs $O(m)$ character comparisons, and we can find all j instances in the string in time $O(m \log n + j)$.

This is the simple code piece to search for L_p (the search for R_p is similar):

if $p \leq T_{suftab[1]}$ **then** $L_p = 1$;
else if $p > T_{suftab[n]}$ **then** $L_p = n + 1$;
else
    $(L, R) = (1, n)$;
    while $R - L > 1$ **do**
        $M = \lceil (L + R)/2 \rceil$;
        if $p \leq T_{suftab[M]}$ **then** $R = M$; **else** $L = M$;
    $L_p = R$;

For example if we search for $p =$ `aca` in the text $T =$ `acaaacatat$` $L_p = 4$ and $R_p = 5$. We find the value L_p and R_p respectively, by setting (L, R) to $(1, n)$ and changing the borders of this interval based on the comparison with the suffix at position $\lceil (L+R)/2 \rceil$, e.g., we find L_p with the sequence: $(1, 10) \Rightarrow (1, 6) \Rightarrow (1, 4) \Rightarrow (3, 4)$. Hence $L_p = 4$.

1	`$`
2	`aaacatat$`
3	`aacatat$`
4	`acaaacatat$`
5	`acatat$`
6	`at$`
7	`atat$`
8	`caaacatat$`
9	`catat$`
10	`t$`
11	`tat$`

The binary searches each need $O(\log n)$ steps. In each step we need to compare m characters of the text and the pattern in the $\leq$ operations. Finally we have to report the j matches. This leads to a running time of $O(m \log n + j)$.

Can we do better?

While the binary search continues, L and R are the left and right boundaries of the current search interval. At the start, L equals 1 and R equals n. Then in each iteration of the binary search a query is made at location $M = \lceil (R + L)/2 \rceil$ of $suftab$.

We keep track of the longest prefixes of $suftab(L)$ and $suftab(R)$ that match a prefix of p. Let l and r denote the prefix lengths respectively and let

$$mlr = \min(l, r) \ .$$

Then we can use the value mlr to accelerate the lexicographical comparison of p and the suffix $suftab[M]$. Since $suftab$ is ordered, it is clear that all suffixes between L and R share the same prefix. Hence we can start the first comparison at position $mlr + 1$.

In practice this trick already brings the running time to $O(m + \log n + j)$, however one can construct an example that still needs time $O(m \log n + j)$.

We call an examination of a character of P *redundant* if that character has been examined before. The goal is to limit the number of redundant character comparisons to $O(1)$ per iteration of the binary search.

The use of mlr alone does not suffice: In the case that $l \neq r$, all characters in P from $mlr + 1$ to $\max(l, r)$ will have already been examined. Thus all comparisons to these characters are redundant. We need a way to begin the comparisons at the *maximum* of l and r.

To do this we introduce the following definition.

Definition 1. $lcp(i, j)$ is the length of the longest common prefix of the suffixes specified in positions i and j of $suftab$.

For example for $S = aabaacatat$ the $lcp(1, 2)$ is the length of the longest common prefix of *aabaacata* and *aacata* which is 2.

With the help of the lcp information, we can achieve our goal of one redundant character comparison per iteration of the search. For now assume that we know $lcp(i, j), \forall i, j$.

How do we use the lcp information? In the case of $l = r$ we compare P to $suftab[M]$ as before starting from position $mlr + 1$, since in this case the minimum of l and r is also the maximum of the two and no redundant character comparisons are made.

If $l \neq r$, there are three cases. We assume w.l.o.g. $l > r$.

Case 1: $lcp(L, M) > l$.
Then the common prefix of the suffixes $S_{suftab[L]}$ and $S_{suftab[M]}$ is longer than the common prefix of P and $S_{suftab[L]}$.

Therefore, P agrees with the suffix $S_{suftab[M]}$ up through character l. Or to put it differently, characters $l + 1$ of $S_{suftab[L]}$ and $S_{suftab[M]}$ are identical and lexically less than character $l + 1$ of P.

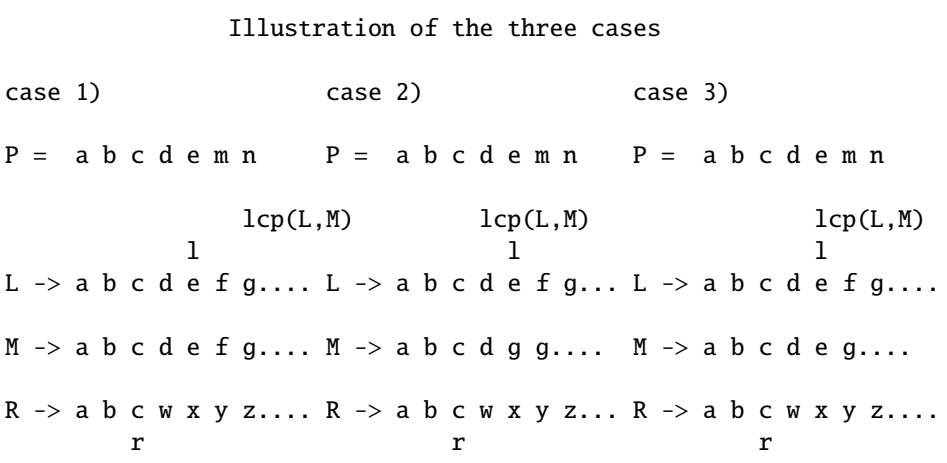
Hence any possible starting position must start to the right of M in $suftab$. So in this case *no* examination of P is needed. L is set to M and l and r remain unchanged.

Case 2: $lcp(L, M) < l$.
Then the common prefix of suffix $suftab[L]$ and $suftab[M]$ is smaller than the common prefix of $suftab[L]$ and P.

Therefore P agrees with $suftab[M]$ up through character $lcp(L,M)$. The $lcp(L,M)+1$ characters of P and and $suftab[L]$ are identical and lexically less than the character $lcp(L,M)+1$ of $suftab[M]$.

Hence any possible starting position must start left of M in $suftab$. So in this case again *no* examination of P is needed. R is set to M, r is changed to $lcp(L,M)$, and l remains unchanged.

Case 3: $lcp(L,M) = l$.
Then P agrees with $suftab[M]$ up to character l. The algorithm then lexically compares P to $suftab[M]$ starting from position $l+1$. In the usual manner the outcome of the compare determines which of L and R change along with the corresponding change of l and r.



Then the following theorem holds:

Theorem 2. *Using the lcp values, the search algorithm does at most $O(m + \log n)$ comparisons and runs in that time.*

Proof: Exercise. Use the fact that neither l nor r decrease in the binary search, and find a bound for the number of redundant comparisons per iteration of the binary search.

1.19 Computing the *lcp* values

We now know how to search fast in a suffix array under the assumption, that we know the *lcp* values for all pairs i, j.

But how do we compute the *lcp* values? And which ones? Computing them all would require too much time and, worse, quadratic space!

We will now first dicuss, which *lcp* values we really need. There exists a simple $O(n)$ algorithm to compute the *lcp* values given the suffix array *suftab* which we will not adress.

We first observe that indeed we only need the *lcp* values of L and R that we encounter in the binary search for L_P and R_P. However, the set of pairs (i, j) which can be considered is contained in a binary search tree which does not depend on P, and has linear size.

Observation 3. Only $O(n)$ many *lcp* values are needed for the *lcp* based search in a suffix array.

Example: $n = 9$

```
(1,9)
(1,5)                   (5,9)
(1,3)       (3,5)       (5,7)       (7,9)
(1,2) (2,3) (3,4) (4,5) (5,6) (6,7) (7,8) (8,9)
```

We get those values in a two step procedure:

1. Compute the *lcp* values for pairs of suffixes *adjacent* in *suftab* using an array *height* of size n.
2. For the fixed binary search tree used in the search for L_P and R_P compute the *lcp* values for its internal nodes using the array *height*. (exercise *)

(*) The value at an internal node is the minimum of its successors (why?)

Hence the essential thing to do is to compute the array *height*, i.e. the *lcp* values of adjacent suffixes in *suftab* .

1.20 Final remarks

- This was just an introduction.
- We did not cover:
  - linear-time construction of suffix trees (e. g., Ukkonen's algorithm)
  - direct $O(n \log n)$ time construction of suffix arrays (Manber & Myers)
  - the linear time construction algorithm for the height array.
  - ...