8 Fast string matching

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

- Shift-And/Shift-Or
  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

8.1 Thoughts about string matching

We will get to know one very practical string matching algorithms for a single pattern 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:

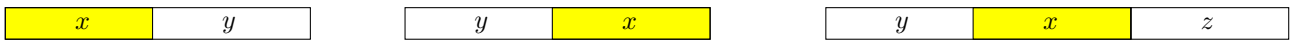
The task at hand is to find all occurrences of a given pattern $p = p_1, \ldots, p_m$ in a text $T = t_1, \ldots, t_n$ usually with $n \gg m$.

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 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 $O(mn)$ but is quite fast in practice.

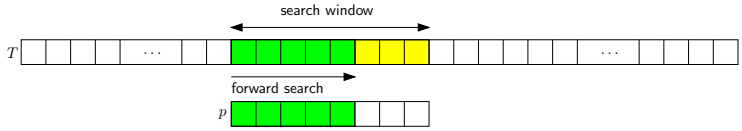
Some easy terminology: Given strings x, y, and z, we say that x is a *prefix* of xy, a *suffix* of yx, and a *factor* (:=substring) of yxz.



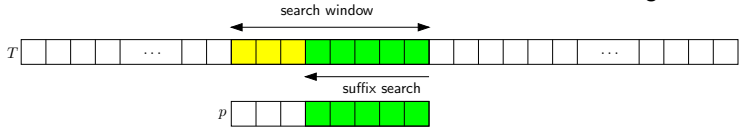
prefix suffix factor

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:

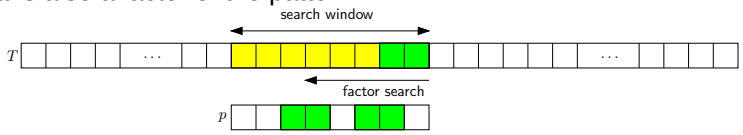
1. *Prefix searching*. For each position of the window we search the longest prefix of the window that is also a prefix of the pattern.



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.



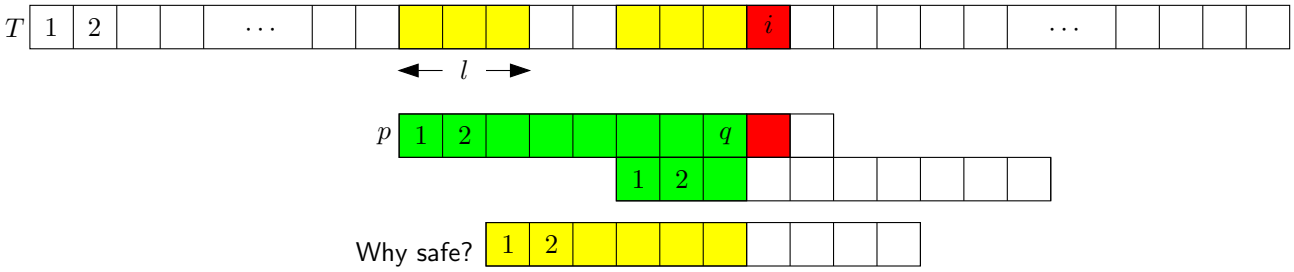
8.2 Prefix based approaches

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 (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).

Knuth-Morris-Pratt:



- Last q characters of T match with first q characters of p.
- Determine maximal prefix of $p[1 \ldots q]$ that is also suffix of $p[1 \ldots q]$ (let l be its length).
- Move the window $q - l$ positions to the right.
- Why is it *safe*?

Since the KMP algorithm is inferior in practice we concentrate on the *Shift-And* and the *Shift-Or* algorithms, which maintain the set of all prefixes of p that match a suffix of the text read.

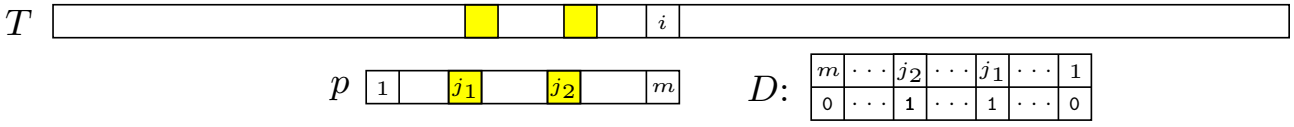
The algorithms use bit-parallelism to update this set for each new text character. The set is represented by a bit mask $D = d_m, \ldots, d_1$.

We start with the Shift-And algorithm, which is easier to explain. The Shift-Or algorithm then results as an implementation trick.

8.3 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 $p_j = c$.

```
1  Shift-And(p, T);
2  // Preprocessing
3  for c ∈ Σ do B[c] = 0^m; od
4  for j ∈ 1...m do B[p_j] = B[p_j] | 0^{m-j}10^{j-1} od
5  // Searching
6  D = 0^m;
7  for pos ∈ 1...n do
8      D = ((D ≪ 1) | 0^{m-1}1) & B[t_pos];
9      if D & 10^{m-1} ≠ 0^m
10       then report an occurrence at position pos − m + 1;
11     fi
12 od
```

Initially we set $D = \mathtt{0}^m$ and for each new character t_{pos} we update D using the formula $D' = ((D \ll \mathtt{1}) \mid \mathtt{0}^{m-1}\mathtt{1}) \,\&\, B[t_{pos}]$. 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 step i (notice that this includes the empty string ϵ). In addition, to stay a match, the character t_{i+1} has to match p at those positions. This is achieved by applying an $\&$-operation with the corresponding bitmask $B[t_{i+1}]$.

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 8.

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 $\mid$ instead of an $\&$ and report a match whenever $d_m = \mathtt{0}$.

Lets look at an example of Shift-And and Shift-Or.

8.4 Example

Find all occurrences of the pattern `P=atat` in the text `T=atacgatatata`.

```
          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        0010     1100
0101     1010        1010     0101
-------------        -------------
0001     1110        0010     1101

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 $9 - 4 + 1 = 6$.

Note: The bit order is reversed in the Java applet.

The running time of the algorithms is $O(n)$, assuming that the operations on D can be done in constant time.