String Comparison / Pattern Matching

The two "killer apps" of modern string processing algorithms have been computational biology and searching the Internet.

While *exact matching* algorithms are more important in text processing than biology, they are important (1) to illustrate techniques, and (2) as a component of heuristics for approximate matching.

Three primary classes of string matching problems arise, determining whether preprocessing techniques are appropriate:

String Comparison / Pattern Matching

- *Fixed texts, variable patterns* – e.g. search the human genome or the Bible.

  Suffix trees/arrays are data structures to efficiently support repeated queries on fixed strings.

- *Variable texts, fixed patterns* – e.g. search a news feed for dirty words, or the latest Genbank entries for specific motifs.

  Such applications justify preprocessing the set of patterns so as to speed search.

- *Variable texts, variable patterns* – e.g. the naive $O(nm)$ brute force search algorithm.

  Suffix trees solve this problem in linear time, but with excessive complexity and overhead.

Efficient Exact String Matching

We have seen the naive brute force algorithm searching for pattern p in text t in $O(mn)$ time, where $m = |p|$ and $n = |t|$.

The brute force algorithm can be viewed as sliding the pattern across from left to right by *one position* when we detect a mismatch between the pattern and the text.

But in certain circumstances we can slide the pattern to the right by more than one position:

```
pattern:        ABCDABCE
text:       ....ABCDABCD....
```

Since we know the last seven characters of the text must be $ABCDABC$, we can shift the pattern four positions without missing any matches.

The Knuth-Morris-Pratt algorithm

Whenever a character match fails, we can shift the pattern forward according to the *failure function* $fail(q)$, which is the length of the longest prefix of P which is a *proper* suffix of P_q

i:	1	2	3	4	5	6	7	8	9	10
P[i]:	a	b	a	b	a	b	a	b	c	a
fail[i]:	0	0	1	2	3	4	5	6	0	1

Given this prefix function we can match efficiently – on a character match we increment the pointers, on a mismatch we slide the pattern one step plus the failure function.

The Knuth-Morris-Pratt algorithm

```
a b b b a b a b a b a c a b a b a b a b c a
a b *
    *
      *
        a b a b a b a *
                      *
                        a b a b a b a b c a
```

Note that we will not match the pattern from the begining, but according to where we are in the text – we never look backwards in the text.

Computing the failure function for the pattern can be done in $O(m)$ preprocessing, and hence does not change the asymptotic complexity.

Complexity of KMP – Amortized analysis

If the pattern rejects near the beginning, we did not waste much time in the search.

If the pattern rejects near the end (e.g. EEEEEEEH in E^m, there is an opportunity to slide it back many positions.

Note that we never move backwards through the text to compare a character again – we slide the *pattern* forward accordingly.

The linearity of KMP follows from an *amortized* analysis. Each time we move forward in the text we add c steps to our account, while each mismatch costs us one step.

Complexity of KMP – Amortized analysis

Provided the account never goes negative, we only did $O(n)$ work total since there were a total of $\leq cn$ steps put in the account.

Thus if many consecutive mismatches requires the pattern to shift repeatedly, it is only because we have enough previous forward moves to compensate.

There are "improved" failure functions which do even cleverer preprocessing to reduce the number of pattern shifts. However, such improvements have little effect in practice and are tricky to program correctly.

The Boyer-Moore Algorithm

An alternate linear algorithm, *Boyer-Moore*, starts matching from right side of the pattern instead of the left side:

```
pattern:    ABCDABCD
text:       ABCDABCE....
```

In this example, the last character in the window does not occur anywhere in the pattern. Thus we can shift the pattern m positions after *one* comparison.

Thus the best case time for Boyer-Moore is *sub-linear*, i.e. $O(n/m)$. The worst-case is $O(n + rm)$, where r is the number of times the pattern occurs in the text.

The Boyer-Moore Algorithm

The algorithm precomputes a table of size $|\Sigma|$ describing how far to shift for a mismatch on each letter of the alphabet, i.e. depending upon the position of the rightmost occurrence of that letter in the pattern.

Further, since we know that the suffix of the pattern matched up to the point of mismatch, we can precompute a table recording the next place where the suffix occurs in the pattern.

We can use the *bigger* of the two shifts, since no intermediate position can define a legal match.

Boyer-Moore *may* be the fastest algorithm for alphanumeric string matching in practice when implemented correctly.

Randomized string matching

The *Rabin-Karp* algorithm computes an appropriate hash function on all m-length strings of the text, and does a brute force comparison only if the hash value is the same for the text window and the pattern.

An appropriate hash function is

$$H(S) = \sum_{i=1}^{m} d^i S_i \bmod q$$

which treats each string as an m-digit base-d number, mod q.

This hash function can be computed *incrementally* in constant time as we slide the window from left to right since

$$H(S_{j+1}) = dH(S_j) + S_{j+1} - d^m S_{j-m}$$

Further, if q is a random prime the expected number of false positives is small enough to yield a *randomized* linear algorithm.

Multiple exact patterns

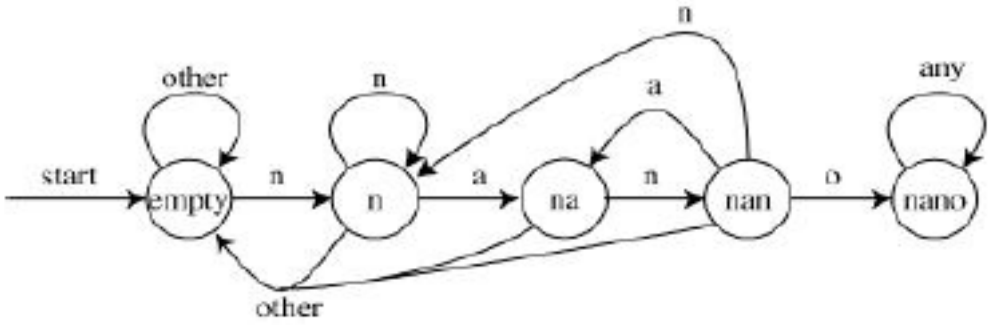
Many applications require searching a text for occurrences of any one of many patterns, e.g. searching text for dirty words or searching a genome for any one of a set of known motifs.

Pattern matching with *wild card* characters ($ACG?T$) is an important special case of multiple patterns.

Techniques from *automata theory* come into play, since any finite set of patterns can be modeled by *regular expressions*, and many interesting infinite sets (e.g. $G(AT) * C$) as well.

Multiple exact patterns

The standard UNIX tool *grep* stands for "general regular expression pattern matcher".



The Aho-Corasick algorithm builds a DFA from the set of patterns and then walks through the text in linear time, taking action when reaching any accepting state.