Theorem 1.38: An ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ can be preprocessed in $\mathcal{O}(L(\mathcal{R}))$ time and $\mathcal{O}(n)$ space so that a binary search with a query string P can be executed in $\mathcal{O}(|P| + \log n)$ time.

Proof. The values $LLCP[mid]$ and $RLCP[mid]$ can be computed in $\mathcal{O}(lcp(S_{mid}, \mathcal{R} \setminus \{S_{mid}\})$ time. Thus the arrays $LLCP$ and $RLCP$ can be computed in $\mathcal{O}(lcp(\mathcal{R})) = \mathcal{O}(L(\mathcal{R}))$ time and stored in $\mathcal{O}(n)$ space.

The main while loop in Algorithm 1.37 is executed $\mathcal{O}(\log n)$ times and everything except LcpCompare on line (11) needs constant time.

If a given LcpCompare call performs $t + 1$ symbol comparisons, $mclp$ increases by t on line (11). Then on lines (12)–(13), either $llcp$ or $rlcp$ increases by at least t, since $mlcp$ was $\max\{llcp, rlcp\}$ before LcpCompare. Since $llcp$ and $rlcp$ never decrease and never grow larger than $|P|$, the total number of extra symbol comparisons in LcpCompare during the binary search is $\mathcal{O}(|P|)$. $\square$

String Binary Search Trees

Binary search can be seen as a search on an *implicit* binary search tree, where the middle element is the root, the middle elements of the first and second half are the children of the root, etc.. The string binary search technique can be extended for arbitrary *binary search trees*.

- Let S_v be the string stored at a node v in a binary search tree. Let $S_<$ and $S_>$ be the closest lexicographically smaller and larger strings stored at *ancestors* of v.

- The comparison of a query string P and the string S_v is done the same way as the comparison of P and S_{mid} in string binary search. The roles of S_{left} and S_{right} are taken by $S_<$ and $S_>$.

- If each node v stores the values $lcp(S_<, S_v)$ and $lcp(S_v, S_>)$, then a search in a balanced search tree can be executed in $\mathcal{O}(|P| + \log n)$ time. Other operations including insertions and deletions take $\mathcal{O}(|P| + \log n)$ time too.

Hashing

Hashing is a powerful technique for dealing with strings based on mapping each string to an integer using a *hash function*:

$$H : \Sigma^* \to [0..q) \subset \mathbb{N}$$

The most common use of hashing is with *hash tables*. Hash tables come in many flavors that can be used with strings as well as with any other type of object with an appropriate hash function. A drawback of using a hash table to store a set of strings is that they do not support lcp and prefix queries.

Hashing is also used in other situations, where one needs to check whether two strings S and T are the same or not:

- If $H(S) \neq H(T)$, then we must have $S \neq T$.

- If $H(S) = H(T)$, then $S = T$ and $S \neq T$ are both possible. If $S \neq T$, this is called a *collision*.

When used this way, the hash value is often called a *fingerprint*, and its range $[0..q)$ is typically large as it is not restricted by the hash table size.

Any good hash function must depend on all characters. Thus computing $H(S)$ needs $\Omega(|S|)$ time, which can defeat the advantages of hashing:

- A plain comparison of two strings is faster than computing the hashes.

- The main strength of hash tables is the support for constant time insertions and deletions, but inserting a string S into a hash table needs $\Omega(|S|)$ time when the hash computation time is included. Compare this to the $\mathcal{O}(|S|)$ time for a trie under a constant alphabet and the $\mathcal{O}(|S| + \log n)$ time for a ternary trie.

However, a hash table can still be competitive in practice. Furthermore, there are situations, where a full computation of the hash function can be avoided:

- A hash value can be computed once, stored, and used many times.

- Some hash functions can be computed more efficiently for a related set of strings. An example is the Karp–Rabin hash function.

Definition 1.39: The *Karp–Rabin hash function* for a string $S = s_0 s_1 \ldots s_{m-1}$ is

$$H(S) = (s_0 r^{m-1} + s_1 r^{m-2} + \cdots + s_{m-2} r + s_{m-1}) \bmod q$$

for some fixed positive integers r and q.

Lemma 1.40: For any two strings A and B,

$$H(AB) = (H(A) \cdot r^{|B|} + H(B)) \bmod q$$
$$H(B) = (H(AB) - H(A) \cdot r^{|B|}) \bmod q$$

Proof. Without the modulo operation, the result would be obvious. The modulo does not interfere because of the rules of *modular arithmetic*:

$$(x + y) \bmod q = ((x \bmod q) + (y \bmod q)) \bmod q$$
$$(xy) \bmod q = ((x \bmod q)(y \bmod q)) \bmod q$$

$\square$

Thus we can quickly compute $H(AB)$ from $H(A)$ and $H(B)$, and $H(B)$ from $H(AB)$ and $H(A)$. We will see applications of this later.

(The equation $(H(A) \cdot r^{|B|}) \bmod q = (H(AB) - H(B)) \bmod q$ is not as useful for computing $H(A)$ because division is harder under modular arithmetic and not even well-defined for all q and r.)

The parameters q and r have to be chosen with some care to ensure that collisions are rare for any reasonable set of strings.

- The original choice is $r = \sigma$ and q is a large *prime*.

- Another possibility is that q is a *power of two* and r is a small prime ($r = 37$ has been suggested). This is faster in practice, because the slow modulo operations can be replaced by bitwise shift operations. If $q = 2^w$, where w is the machine word size, the modulo operations can be omitted completely.

- If q and r were both powers of two, then only the last $\lceil (\log q)/\log r \rceil$ characters of the string would affect the hash value.

- The hash function can be *randomized* by choosing q or r randomly. Furthermore, we can change q or r if collisions are too frequent.

Automata

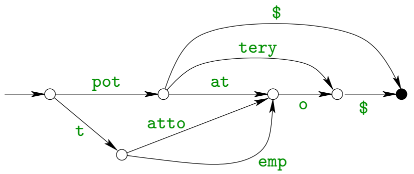
Finite automata are a well known way of representing sets of strings. In this case, the set is often called a **language**.

A trie is a special type of an automaton.

- Trie is generally not a *minimal* automaton.

- Trie techniques including path compaction and ternary branching can be applied to automata.

Example 1.41: Compacted minimal automaton for $\mathcal{R} = \{\texttt{pot\$}, \texttt{potato\$}, \texttt{pottery\$}, \texttt{tattoo\$}, \texttt{tempo\$}\}$.



Automata are much more powerful than tries in representing languages:

- Infinite languages

- Nondeterministic automata

- Even an acyclic, deterministic automaton can represent a language of exponential size.

Automata do not support all operations of tries:

- Insertions and deletions

- Satellite data, i.e., data associated to each string.

2. Exact String Matching

Let $T = T[0..n)$ be the *text* and $P = P[0..m)$ the *pattern*. We say that P *occurs* in T at position j if $T[j..j+m) = P$.

Example: $P =$ `aine` occurs at position 6 in $T =$ `karjalainen`.

In this part, we will describe algorithms that solve the following problem.

Problem 2.1: Given text $T[0..n)$ and pattern $P[0..m)$, report the first position in T where P occurs, or n if P does not occur in T.

The algorithms can be easily modified to solve the following problems too.

- Existence: Is P a factor of T?
- Counting: Count the number of occurrences of P in T.
- Listing: Report all occurrences of P in T.

The naive, brute force algorithm compares P against $T[0..m)$, then against $T[1..1+m)$, then against $T[2..2+m)$ etc. until an occurrence is found or the end of the text is reached.

Algorithm 2.2: Brute force
Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T

(1) $i \leftarrow 0; j \leftarrow 0$
(2) *while* $i < m$ *and* $j < n$ *do*
(3) *if* $P[i] = T[j]$ *then* $i \leftarrow i+1; j \leftarrow j+1$
(4) *else* $j \leftarrow j - i + 1; i \leftarrow 0$
(5) *if* $i = m$ *then* output $j - m$ *else* output n

The worst case time complexity is $\mathcal{O}(mn)$. This happens, for example, when $P = \mathtt{a}^{m-1}\mathtt{b} =$ `aaa..ab` and $T = \mathtt{a}^n =$ `aaaaaa..aa`.

Knuth–Morris–Pratt

The Brute force algorithm forgets everything when it moves to the next text position.

The Morris–Pratt (MP) algorithm remembers matches. It never goes back to a text character that already matched.

The Knuth–Morris–Pratt (KMP) algorithm remembers mismatches too.

Example 2.3:

Brute force	Morris–Pratt	Knuth–Morris–Pratt
ainaisesti-ainainen	ainaisesti-ainainen	ainaisesti-ainainen
ainai~~n~~en (6 comp.)	ainai~~n~~en (6)	ainai~~n~~en (6)
~~a~~inainen (1)	ai~~n~~ainen (1)	~~a~~inainen (1)
~~a~~inainen (1)	~~a~~inainen (1)	
ai~~n~~ainen (3)		
~~a~~inainen (1)		
~~a~~inainen (1)		

MP and KMP algorithms never go backwards in the text. When they encounter a mismatch, they find another pattern position to compare against the same text position. If the mismatch occurs at pattern position i, then $fail[i]$ is the next pattern position to compare.

The only difference between MP and KMP is how they compute the *failure function* $fail$.

Algorithm 2.4: Knuth–Morris–Pratt / Morris–Pratt
Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T

(1) compute $fail[0..m]$
(2) $i \leftarrow 0; j \leftarrow 0$
(3) *while* $i < m$ *and* $j < n$ *do*
(4) *if* $i = -1$ *or* $P[i] = T[j]$ *then* $i \leftarrow i+1; j \leftarrow j+1$
(5) *else* $i \leftarrow fail[i]$
(6) *if* $i = m$ *then* output $j - m$ *else* output n

- $fail[i] = -1$ means that there is no more pattern positions to compare against this text positions and we should move to the next text position.
- $fail[m]$ is never needed here, but if we wanted to find all occurrences, it would tell how to continue after a full match.

We will describe the MP failure function here. The KMP failure function is left for the exercises.

- When the algorithm finds a mismatch between $P[i]$ and $T[j]$, we know that $P[0..i) = T[j-i..j)$.
- Now we want to find a new $i' < i$ such that $P[0..i') = T[j-i'..j)$. Specifically, we want the largest such i'.
- This means that $P[0..i') = T[j-i'..j) = P[i-i'..i)$. In other words, $P[0..i')$ is the *longest proper border* of $P[0..i)$.

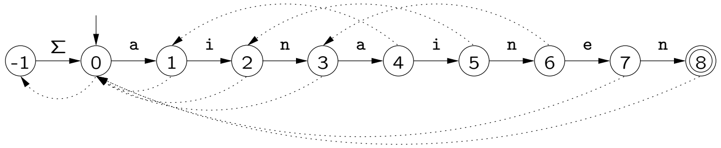
Example: `ai` is the longest proper border of `ainai`.

- Thus $fail[i]$ is the length of the longest proper border of $P[0..i)$.
- $P[0..0) = \varepsilon$ has no proper border. We set $fail[0] = -1$.

Example 2.5: Let $P =$ `ainainen`.

i	$P[0..i)$	border	$fail[i]$
0	ε	–	-1
1	`a`	ε	0
2	`ai`	ε	0
3	`ain`	ε	0
4	`aina`	`a`	1
5	`ainai`	`ai`	2
6	`ainain`	`ain`	3
7	`ainaine`	ε	0
8	`ainainen`	ε	0

The (K)MP algorithm operates like an automaton, since it never moves backwards in the text. Indeed, it can be described by an automaton that has a special *failure transition*, which is an ε-transition that can be taken only when there is no other transition to take.



An efficient algorithm for computing the failure function is very similar to the search algorithm itself!

- In the MP algorithm, when we find a match $P[i] = T[j]$, we know that $P[0..i] = T[j-i..j]$. More specifically, $P[0..i]$ is the longest prefix of P that matches a suffix of $T[0..j]$.
- Suppose $T = \#P[1..m)$, where $\#$ is a symbol that does not occur in P. Finding a match $P[i] = T[j]$, we know that $P[0..i]$ is the longest prefix of P that is a proper suffix of $P[0..j]$. Thus $fail[j+1] = i+1$.

Algorithm 2.6: Morris–Pratt failure function computation
Input: pattern $P = P[0 \dots m)$
Output: array $fail[0..m]$ for P
(1) $i \leftarrow -1; j \leftarrow 0;$ $fail[j] \leftarrow i$
(2) *while* $j < m$ *do*
(3) *if* $i = -1$ *or* $P[i] = P[j]$ *then* $i \leftarrow i+1; j \leftarrow j+1;$ $fail[j] \leftarrow i$
(4) *else* $i \leftarrow fail[i]$
(5) output $fail$

- When the algorithm reads $fail[i]$ on line 4, $fail[i]$ has already been computed.

Theorem 2.7: Algorithms MP and KMP preprocess a pattern in time $\mathcal{O}(m)$ and then search the text in time $\mathcal{O}(n)$.

Proof. We show that the text search requires $\mathcal{O}(n)$ time. Exactly the same argument shows that pattern preprocessing needs $\mathcal{O}(m)$ time.

It is sufficient to count the number of comparisons that the algorithms make. After each comparison $P[i] = T[j]$, one of the two conditional branches is executed:

then Here j is incremented. Since j never decreases, this branch can be taken at most $n+1$ times.

else Here i decreases since $fail[i] < i$. Since i only increases in the then-branch, this branch cannot be taken more often than the then-branch.

□