We can further improve string binary search using precomputed information about the lcp's between the strings in $R$.
Consider again the basic situation during string binary search:
- We want to compare $P$ and $S_{mid}$.
- We have already compared $P$ against $S_{left}$ and $S_{right}$, and we know $lcp(S_{left},P)$ and $lcp(P,S_{right})$.

The values $left$ and $right$ are fully determined by $mid$ independently of $P$.
That is, $P$ only determines whether the search ends up at position $mid$ at all, but if it does, $left$ and $right$ are always the same.

Thus, we can precompute and store the values
\[
LLCP[mid] = lcp(S_{left},S_{mid})
\]
\[
RLCP[mid] = lcp(S_{mid},S_{right})
\]

Now we know all lcp values between $P$, $S_{left}$, $S_{mid}$, $S_{right}$ except $lcp(P,S_{mid})$.
The following lemma shows how to utilize $lcp(A,B)$ and $lcp(B,B')$
\[
lcp(B,B') \geq \min\{lcp(B,C),lcp(B',C)\} = lcp(B,C)
\]
Thus $lcp(B,B') \geq \max\{lcp(A,B),lcp(B,C)\}$. □

**Algorithm 1.35:** String binary search (with precomputed lcps)

**Input:** Ordered string set $R = \{S_1, S_2, ..., S_n\}$, arrays LLCP and RLCP, query string $P$.

**Output:** The number of strings in $R$ that are smaller than $P$.

1. $left \leftarrow 0$; $right \leftarrow n + 1$
2. $left \leftarrow 0$; $right \leftarrow 0$
3. while $right - left > 1$
4. $mid \leftarrow \lfloor (left + right)/2 \rfloor$
5. if $LLCP[mid] > lcp$ then $left \leftarrow mid$
6. else if $RLCP[mid] < lcp$ then $right \leftarrow mid$
7. else $lcp \leftarrow LLCP[mid]$
8. else $lcp \leftarrow RLCP[mid]$
9. $mclp \leftarrow max\{lcp(left),lcp(mid),lcp(right)\}$
10. $(x, mclp) \leftarrow LcpCompare(S_{mid}, P, mclp)$
11. if $x = "<"$ then $left \leftarrow mid$; $lcp \leftarrow mclp$
12. else $right \leftarrow mid$; $lcp \leftarrow mclp$
13. return $left$

**Theorem 1.36:** An ordered string set $R = \{S_1, S_2, ..., S_n\}$ can be preprocessed in $O(\Sigma LCP(R) + n)$ time and $O(n)$ space so that a binary search with a query string $P$ can be executed in $O(\log n)$ time.

**Proof.** The values $LLCP[mid]$ and $RLCP[mid]$ can be computed in $O(lcp(S_{mid}, R) + (S_{mid}) + 1)$ time. Thus the arrays $LLCP$ and $RLCP$ can be computed in $O(\Sigma lcp(R) + n)$ time and stored in $O(n)$ space.

The main while loop in Algorithm 1.35 is executed $O(\log n)$ 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 $lcp$ or $lcp(left)$ increases by at least $t$, since $mclp$ is max bounded. If $lcp(left)$ never decrease and never grow larger than $\lceil t \rceil$, the total number of extra symbol comparisons in $LcpCompare$ during the binary search is $O(\log n)$.

**Hashing and Fingerprints**

Hashing is a powerful technique for dealing with strings based on mapping each string to an integer using a hash function:
\[
H : \Sigma^* \rightarrow \{0,q\} \subset \mathbb{N}
\]

The most common use of hashing is with hash tables. Hash tables come in many shapes and sizes as well as with each object of type 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 a hash table size.

**Definition 1.37:** The Karp–Rabin hash function for a string $S = s_0s_1...s_{n-1}$ over an integer alphabet is
\[
H(S) = \sum_{i=0}^{n-1} s_i \alpha^{n-i-1} + s_i \alpha^{n-i-2} + ... + s_{n-2} \alpha + s_{n-1} \mod q
\]
for some fixed positive integers $q$ and $\alpha$.

**Lemma 1.38:** For any two strings $A$ and $B$,
\[
H(AB) \equiv (H(A) - \alpha^{lcp(A,B)}) \mod q
\]
\[
H(B) \equiv (H(AB) - H(A)) \mod q
\]

**Proof.** Without the modulo operation, the result would be obvious. The modulo does not interfere because of the rules of modular arithmetic:
\[
(x + y) \mod q = ((x \mod q) + (y \mod q)) \mod q
\]
\[
(xy) \mod q = ((x \mod q)(y \mod q)) \mod q
\]

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.

If $q$ and $\alpha$ are coprime, then $\alpha$ has a multiplicative inverse $\alpha^{-1} \mod q$, and we can also compute $H(AB) \equiv (H(A) - H(B)) \cdot (\alpha^{-1})^{lcp(A,B)} \mod q$. 


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 $q = 2^r$, 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 $\lceil \log_2 q \rceil / \log_2 r$ characters of the string would affect the hash value. More generally, $q$ and $r$ should be coprime, i.e., have no common divisors other than 1.
- The hash function can be randomized by choosing $q$ or $r$ randomly. For example, if $q$ is a prime and $r$ is chosen uniformly at random from $[0, q)$, the probability that two strings of length $m$ collide is at most $m/q$.
- A random choice over a set of possibilities has the additional advantage that we can change the choice if the first choice leads to too many collisions.

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:** $P = \text{aine}$ occurs at position 6 in $T = \text{karjalaainen}$.

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 $T$.

**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:**

<table>
<thead>
<tr>
<th>Brute force</th>
<th>Morris–Pratt</th>
<th>Knuth–Morris–Pratt</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ainaisesti-ainaainen</em></td>
<td><em>ainaainen</em> (6 comp.)</td>
<td><em>ainaainen</em> (6)</td>
</tr>
<tr>
<td><em>ainaainen</em> (1)</td>
<td><em>ainaainen</em> (1)</td>
<td><em>ainaainen</em> (1)</td>
</tr>
<tr>
<td><em>ainaainen</em> (3)</td>
<td><em>ainaainen</em> (1)</td>
<td><em>ainaainen</em> (1)</td>
</tr>
<tr>
<td><em>ainaainen</em> (1)</td>
<td><em>ainaainen</em> (1)</td>
<td><em>ainaainen</em> (1)</td>
</tr>
</tbody>
</table>

**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 1.39:** Compacted minimal automaton for $R = \{\text{pot}$, $\text{potato}$, $\text{pottery}$, $\text{tattoo}$, $\text{tempo}$\}.

**Sets of Strings: Summary**

Efficient algorithms and data structures for sets of strings:

- Storing and searching: trie and ternary trie and their compact versions, string binary search and string binary search tree, Karp–Rabin hashing.
- Sorting: string quicksort and mergesort, LSD and MSD radix sort.

**Lower bounds:**

- Many of the algorithms are optimal.
- General purpose algorithms are asymptotically slower.

The central role of longest common prefixes:

- LCP array $LCP_R$ and its sum $\Sigma LCP(R)$.
- Lcp-comparison technique.

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

**Algorithm 2.2:** Brute force

**Input:** text $T = T[0..n]$, pattern $P = P[0..m]$

**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 $j \leftarrow j + 1; i \leftarrow i$
(5) if $i = m$ then return $j - m$ else return $n$

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

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 $\text{fail}[i]$ is the next pattern position to compare.

The only difference between MP and KMP is how they compute the failure function $\text{fail}$.

**Algorithm 2.4:** Knuth–Morris–Pratt / Morris–Pratt

**Input:** text $T = T[0..n]$, pattern $P = P[0..m]$

**Output:** position of the first occurrence of $P$ in $T$

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

$\text{fail}[i] = -1$ means that there is no more pattern positions to compare against this text positions and we should move to the next text position.

$\text{fail}[n]$ is never needed here, but if we wanted to find all occurrences, it would tell us 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 \( P[i..i+j) = T[j..j+i) \). Now we want to find a new \( i' < i \) such that \( P[i..i'] = T[j..j+i') \). Specifically, we want the largest such \( i' \).
- This means that \( P[0..i'] = T[j..j+i'] \). In other words, \( P[0..i] \) is the length of the longest proper border of \( P[0..i) \).

\[ P[0..0) = \varepsilon \] has no proper border. We set \( \text{fail}[0] = -1 \).

**Example:** \( a \) is the longest proper border of \( \varepsilon \).

\[ P[0..0] = \varepsilon \] has no proper border. We set \( \text{fail}[0] = -1 \).

A shift-and algorithm is a minor optimization of shift-or. It is the same algorithm except that it makes a transition only when there is no other transition to take.

**Example 2.5:** Let \( P = \text{ainainen} \).

\[
\begin{array}{|c|c|}
\hline
\text{index} & \text{fail} \\
\hline
0 & -1 \\
1 & a \\
2 & a \\
3 & ain \\
4 & aina \\
5 & ainai \\
6 & ainain \\
7 & ainaining \\
8 & ainaining \\
\hline
\end{array}
\]

**Shift-And (Shift-Or)**

When the MP algorithm is at position \( j \) in the text \( T \), it computes the longest prefix of the pattern \( P[0..i) \) that are suffixes of \( T[0..j] \).

1. \( D \leftarrow D & B[T[j]] \) (the bitwise and). Remove the prefixes where \( T[j] \) does not match.

2. \( D \leftarrow D | B[T[j]] \) (the bitwise or). The advantage is that we don't need to "+1" on line 5.

**Example 2.9:** \( P = \text{assi} \), \( T = \text{apassi} \), bitvectors are columns.

<table>
<thead>
<tr>
<th>( B[c] )</th>
<th>( D ) at each step</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( {a, p, s} )</td>
</tr>
<tr>
<td>( s )</td>
<td>( {s, 0, 1} )</td>
</tr>
<tr>
<td>( i )</td>
<td>( {i, 0} )</td>
</tr>
</tbody>
</table>

**Algorithm 2.6:** Morris-Pratt failure function computation

**Input:** pattern \( P = P[0..m] \)

**Output:** array \( \text{fail}[0..n] \) for \( P \)

1. \( i \leftarrow -1; j \leftarrow 0; \text{fail}[i] \leftarrow -1 \)
2. while \( j < m \) do
3. if \( i = -1 \) or \( P[i] = P[j] \) then \( i \leftarrow i + 1; j \leftarrow j + 1; \text{fail}[j] \leftarrow i \)
4. else \( i \leftarrow \text{fail}[i] \)
5. return \( \text{fail} \)

- When the algorithm reads \( \text{fail} \) on line 4, \( \text{fail} \) has already been computed.

Let \( w \) be the wordsize of the computer, typically 64. Assume first that \( m \leq w \). Then each bitvector can be stored in a single integer.

**Algorithm 2.8:** Shift-And

**Input:** text \( T = T[0..n] \), pattern \( P = P[0..m] \)

**Output:** position of the first occurrence of \( P \) in \( T \)

1. for \( c \in \Sigma \) do \( B[c] \leftarrow 0 \)
2. for \( i \leftarrow 0 \) to \( m-1 \) do \( B[P[i]] \leftarrow B[P[i]] + 2^i \) // \( B[P[i]] = \text{fail}[i] + 1 \)
3. \( D \leftarrow 0 \)
4. for \( j \leftarrow 0 \) to \( n-1 \) do
5. if \( D = 0 \) then return \( j \) // \( D = 0 \) otherwise
6. if \( D \neq 2^m-1 \) then return \( j - m + 1 \) // \( D = 1 \) otherwise

**Shift-Or** is a minor optimization of Shift-And. It is the same algorithm except the roles of 0's and 1's in the bitvectors have been swapped. Then \& on line 5 is replaced by | (bitwise or). The advantage is that we don't need to "+1" on line 5.