Similarly, the $\Omega(\Sigma LCP(\mathcal{R}) + n \log n)$ lower bound does not apply to string sorting algorithms that use stronger operations than symbol comparisons. Radix sort is such an algorithm for integer alphabets.

Radix sort was developed for sorting large integers, but it treats an integer as a string of digits, so it is really a string sorting algorithm.

There are two types of radix sorting:

MSD radix sort starts sorting from the beginning of strings (most significant digit).

LSD radix sort starts sorting from the end of strings (least significant digit).

The LSD radix sort algorithm is very simple.

Algorithm 1.25: LSDRadixSort($\mathcal{R}$)
Input: (Multi)set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings of length m over alphabet $[0..\sigma)$.
Output: $\mathcal{R}$ in ascending lexicographical order.
(1) for $\ell \leftarrow m-1$ to 0 do CountingSort($\mathcal{R}$,ℓ)
(2) return $\mathcal{R}$

- CountingSort($\mathcal{R}$,ℓ) sorts the strings in $\mathcal{R}$ by the symbols at position ℓ using counting sort (with k_i replaced by $S_i[\ell]$). The time complexity is $\mathcal{O}(|\mathcal{R}| + \sigma)$.
- The stability of counting sort is essential.

Example 1.26: $\mathcal{R} = \{$cat, him, ham, bat$\}$.

cat	$\Longrightarrow$	hi $\boxed{\text{m}}$	$\Longrightarrow$	h $\boxed{\text{a}}$ m	$\Longrightarrow$	$\boxed{\text{b}}$ at
him		ha $\boxed{\text{m}}$		c $\boxed{\text{a}}$ t		$\boxed{\text{c}}$ at
ham		ca $\boxed{\text{t}}$		b $\boxed{\text{a}}$ t		$\boxed{\text{h}}$ am
bat		ba $\boxed{\text{t}}$		h $\boxed{\text{i}}$ m		$\boxed{\text{h}}$ im

It is easy to show that after i rounds, the strings are sorted by suffix of length i. Thus, they are fully sorted at the end.

The algorithm assumes that all strings have the same length m, but it can be modified to handle strings of different lengths (exercise).

Theorem 1.27: LSD radix sort sorts a set $\mathcal{R}$ of strings over the alphabet $[0..\sigma)$ in $\mathcal{O}(||\mathcal{R}|| + m\sigma)$ time, where $||\mathcal{R}||$ is the total length of the strings in $\mathcal{R}$ and m is the length of the longest string in $\mathcal{R}$.

Proof. Assume all strings have length m. The LSD radix sort performs m rounds with each round taking $\mathcal{O}(n + \sigma)$ time. The total time is $\mathcal{O}(mn + m\sigma) = \mathcal{O}(||\mathcal{R}|| + m\sigma)$.

The case of variable lengths is left as an exercise. $\square$

- The weakness of LSD radix sort is that it uses $\Omega(||\mathcal{R}||)$ time even when $\Sigma LCP(\mathcal{R})$ is much smaller than $||\mathcal{R}||$.
- It is best suited for sorting short strings and integers.

MSD radix sort resembles string quicksort but partitions the strings into σ parts instead of three parts.

Example 1.28: MSD radix sort partitioning.

al	$\boxed{\text{p}}$	habet		al	$\boxed{\text{g}}$	orithm
al	$\boxed{\text{i}}$	gnment		al	$\boxed{\text{i}}$	gnment
al	$\boxed{\text{l}}$	ocate		al	$\boxed{\text{i}}$	as
al	$\boxed{\text{g}}$	orithm	$\Longrightarrow$	al	$\boxed{\text{l}}$	ocate
al	$\boxed{\text{t}}$	ernative		al	$\boxed{\text{l}}$	
al	$\boxed{\text{i}}$	as		al	$\boxed{\text{p}}$	habet
al	$\boxed{\text{t}}$	ernate		al	$\boxed{\text{t}}$	ernative
al	$\boxed{\text{l}}$			al	$\boxed{\text{t}}$	ernate

Algorithm 1.29: MSDRadixSort($\mathcal{R}$, ℓ)
Input: (Multi)set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings over the alphabet $[0..\sigma)$ and the length ℓ of their common prefix.
Output: $\mathcal{R}$ in ascending lexicographical order.
(1) if $|\mathcal{R}| < \sigma$ then return StringQuicksort($\mathcal{R}$, ℓ)
(2) $\mathcal{R}_\perp \leftarrow \{S \in \mathcal{R} \mid |S| = \ell\}$; $\mathcal{R} \leftarrow \mathcal{R} \setminus \mathcal{R}_\perp$
(3) $(\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{\sigma-1}) \leftarrow$ CountingSort($\mathcal{R}$, ℓ)
(4) for $i \leftarrow 0$ to $\sigma - 1$ do $\mathcal{R}_i \leftarrow$ MSDRadixSort($\mathcal{R}_i$, $\ell + 1$)
(5) return $\mathcal{R}_\perp \cdot \mathcal{R}_0 \cdot \mathcal{R}_1 \cdots \mathcal{R}_{\sigma-1}$

- Here CountingSort($\mathcal{R}$,ℓ) not only sorts but also returns the partitioning based on symbols at position ℓ. The time complexity is still $\mathcal{O}(|\mathcal{R}| + \sigma)$.
- The recursive calls eventually lead to a large number of very small sets, but counting sort needs $\Omega(\sigma)$ time no matter how small the set is. To avoid the potentially high cost, the algorithm switches to string quicksort for small sets.

Theorem 1.30: MSD radix sort sorts a set $\mathcal{R}$ of n strings over the alphabet $[0..\sigma)$ in $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n \log \sigma)$ time.

Proof. Consider a call processing a subset of size $k \geq \sigma$:

- The time excluding the recursive calls but including the call to counting sort is $\mathcal{O}(k + \sigma) = \mathcal{O}(k)$. The k symbols accessed here will not be accessed again.
- At most $dp(S, \mathcal{R} \setminus \{S\}) \leq lcp(S, \mathcal{R} \setminus \{S\}) + 1$ symbols in S will be accessed by the algorithm. Thus the total time spent in this kind of calls is $\mathcal{O}(\Sigma dp(\mathcal{R})) = \mathcal{O}(\Sigma lcp(\mathcal{R}) + n) = \mathcal{O}(\Sigma LCP(\mathcal{R}) + n)$.

The calls for a subsets of size $k < \sigma$ are handled by string quicksort. Each string is involved in at most one such call. Therefore, the total time over all calls to string quicksort is $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n \log \sigma)$. $\square$

- There exists a more complicated variant of MSD radix sort with time complexity $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n + \sigma)$.
- $\Omega(\Sigma LCP(\mathcal{R}) + n)$ is a lower bound for any algorithm that must access symbols one at a time.
- In practice, MSD radix sort is very fast, but it is sensitive to implementation details.

Lcp-Comparisons

General (non-string) comparison-based sorting algorithms are not optimal for sorting strings because of an imbalance between effort and result in a string comparison: it can take a lot of time but the result is only a bit or a trit of useful information.

String quicksort solves this problem by processing the obtained information immediately after each symbol comparison.

An opposite approach is to replace a standard string comparison with an lcp-comparison, which is the operation LcpCompare(A, B, k):

- The return value is the pair (x, ℓ), where $x \in \{<, =, >\}$ indicates the order, and $\ell = lcp(A, B)$, the length of the longest common prefix of strings A and B.
- The input value k is the length of a known common prefix, i.e., a lower bound on $lcp(A, B)$. The comparison can skip the first k characters.

Extra time spent in the comparison is balanced by the extra information obtained in the form of the lcp value.

The following result shows how we can use the information from earlier comparisons to obtain a lower bound or even the exact value for an lcp.

Lemma 1.31: Let A, B and C be strings.

(a) $lcp(A, C) \geq \min\{lcp(A, B), lcp(B, C)\}$.
(b) If $A \leq B \leq C$, then $lcp(A, C) = \min\{lcp(A, B), lcp(B, C)\}$.
(c) If $lcp(A, B) \neq lcp(B, C)$, then $lcp(A, C) = \min\{lcp(A, B), lcp(B, C)\}$.

Proof. Assume $\ell = lcp(A, B) \leq lcp(B, C)$. The opposite case $lcp(A, B) \geq lcp(B, C)$ is symmetric.

(a) Now $A[0..\ell) = B[0..\ell) = C[0..\ell)$ and thus $lcp(A, C) \geq \ell$.
(b) Either $|A| = \ell$ or $A[\ell] < B[\ell] \leq C[\ell]$. In either case, $lcp(A, C) = \ell$.
(c) Now $lcp(A, B) < lcp(B, C)$. If $lcp(A, C) > \min\{lcp(A, B), lcp(B, C)\}$, then $lcp(A, B) < \min\{lcp(A, C), lcp(B, C)\}$, which violates (a).

$\square$

The above means that the three lcp values between three strings can never be three different values. At least two of them are the same and the third one is the same or bigger.

It can also be possible to determine the order of two strings without comparing them directly.

Lemma 1.32: Let A, B, B' and C be strings such that $A \le B \le C$ and $A \le B' \le C$.

(a) If $lcp(A,B) > lcp(A,B')$, then $B < B'$.

(b) If $lcp(B,C) > lcp(B',C)$, then $B > B'$.

Proof. We show (a); (b) is symmetric. Assume to the contrary that $B \ge B'$. Then by Lemma 1.31, $lcp(A,B) = \min\{lcp(A,B'), lcp(B',B)\} \le lcp(A,B')$, which is a contradiction. □

Intuitively, the above result makes sense if you think of $lcp(\cdot,\cdot)$ as a *measure of similarity* between two strings. The higher the lcp, the closer the two strings are lexicographically.

String Mergesort

String mergesort is a string sorting algorithm that uses lcp-comparisons. It has the same structure as the standard mergesort: sort the first half and the second half separately, and then merge the results.

Algorithm 1.33: StringMergesort($\mathcal{R}$)
Input: Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings.
Output: $\mathcal{R}$ sorted and augmented with $LCP_{\mathcal{R}}$ values.
(1) if $|\mathcal{R}| = 1$ then return $((S_1, 0))$
(2) $m \leftarrow \lfloor n/2 \rfloor$
(3) $\mathcal{P} \leftarrow$ StringMergesort($\{S_1, S_2, \ldots, S_m\}$)
(4) $\mathcal{Q} \leftarrow$ StringMergesort($\{S_{m+1}, S_{m+2}, \ldots, S_n\}$)
(5) return StringMerge($\mathcal{P}, \mathcal{Q}$)

The output is of the form

$$((T_1, \ell_1), (T_2, \ell_2), \ldots, (T_n, \ell_n))$$

where $\ell_i = lcp(T_i, T_{i-1})$ for $i > 1$ and $\ell_1 = 0$. In other words, $\ell_i = LCP_{\mathcal{R}}[i]$.

Thus we get not only the order of the strings but also a lot of information about their common prefixes. The procedure StringMerge uses this information effectively.

Algorithm 1.34: StringMerge($\mathcal{P}$,$\mathcal{Q}$)
Input: Sequences $\mathcal{P} = ((S_1, k_1), \ldots, (S_m, k_m))$ and $\mathcal{Q} = ((T_1, \ell_1), \ldots, (T_n, \ell_n))$
Output: Merged sequence $\mathcal{R}$

```
(1)  R ← ∅; i ← 1; j ← 1
(2)  while i ≤ m and j ≤ n do
(3)      if k_i > ℓ_j then append (S_i, k_i) to R; i ← i + 1
(4)      else if ℓ_j > k_i then append (T_j, ℓ_j) to R; j ← j + 1
(5)      else   // k_i = ℓ_j
(6)          (x, h) ← LcpCompare(S_i, T_j, k_i)
(7)          if x = "<" then
(8)              append (S_i, k_i) to R; i ← i + 1
(9)              ℓ_j ← h
(10)         else
(11)             append (T_j, ℓ_j) to R; j ← j + 1
(12)             k_i ← h
(13) while i ≤ m do append (S_i, k_i) to R; i ← i + 1
(14) while j ≤ n do append (T_j, ℓ_j) to R; j ← j + 1
(15) return R
```


Lemma 1.35: StringMerge performs the merging correctly.

Proof. We will show that the following invariant holds at the beginning of each round in the loop on lines (2)–(12):

> Let X be the last string appended to $\mathcal{R}$ (or ε if $\mathcal{R} = \emptyset$). Then $k_i = lcp(X, S_i)$ and $\ell_j = lcp(X, T_j)$.

The invariant is clearly true in the beginning. We will show that the invariant is maintained and the smaller string is chosen in each round of the loop.

- If $k_i > \ell_j$, then $lcp(X, S_i) > lcp(X, T_j)$ and thus
  - $S_i < T_j$ by Lemma 1.32.
  - $lcp(S_i, T_j) = lcp(X, T_j)$ because, by Lemma 1.31, $lcp(X, T_j) = \min\{lcp(X, S_i), lcp(S_i, T_j)\}$.

  Hence, the algorithm chooses the smaller string and maintains the invariant. The case $\ell_j > k_i$ is symmetric.

- If $k_i = \ell_j$, then clearly $lcp(S_i, T_j) \ge k_i$ and the call to LcpCompare is safe, and the smaller string is chosen. The update $\ell_j \leftarrow h$ or $k_i \leftarrow h$ maintains the invariant. □

Theorem 1.36: String mergesort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n \log n)$ time.

Proof. If the calls to LcpCompare took constant time, the time complexity would be $\mathcal{O}(n \log n)$ by the same argument as with the standard mergesort.

Whenever LcpCompare makes more than one, say $t+1$ symbol comparisons, one of the lcp values stored with the strings increases by t. Since the sum of the final lcp values is exactly $\Sigma LCP(\mathcal{R})$, the extra time spent in LcpCompare is bounded by $\mathcal{O}(\Sigma LCP(\mathcal{R}))$. □

- Other comparison based sorting algorithms, for example heapsort and insertion sort, can be adapted for strings using the lcp-comparison technique.

String Binary Search

An ordered array is a simple static data structure supporting queries in $\mathcal{O}(\log n)$ time using binary search.

Algorithm 1.37: Binary search
Input: Ordered set $R = \{k_1, k_2, \ldots, k_n\}$, query value x.
Output: The number of elements in R that are smaller than x.
(1) $left \leftarrow 0$; $right \leftarrow n + 1$ // output value is in the range $[left..right)$
(2) while $right - left > 1$ do
(3) $\quad mid \leftarrow \lfloor (left + right)/2 \rfloor$
(4) $\quad$ if $k_{mid} < x$ then $left \leftarrow mid$
(5) $\quad$ else $right \leftarrow mid$
(6) return $left$

With strings as elements, however, the query time is

- $\mathcal{O}(m \log n)$ in the worst case for a query string of length m
- $\mathcal{O}(\log n \log_\sigma n)$ on average for a random set of strings.

We can use the lcp-comparison technique to improve binary search for strings. The following is a key result.

Lemma 1.38: Let A, B, B' and C be strings such that $A \le B \le C$ and $A \le B' \le C$. Then $lcp(B, B') \ge lcp(A, C)$.

Proof. Let $B_{min} = \min\{B, B'\}$ and $B_{max} = \max\{B, B'\}$. By Lemma 1.31,

$$\begin{aligned} lcp(A,C) &= \min(lcp(A, B_{max}), lcp(B_{max}, C)) \\ &\le lcp(A, B_{max}) = \min(lcp(A, B_{min}), lcp(B_{min}, B_{max})) \\ &\le lcp(B_{min}, B_{max}) = lcp(B, B') \end{aligned}$$

□

During the binary search of P in $\{S_1, S_2, \ldots, S_n\}$, the basic situation is the following:

- We want to compare P and S_{mid}.
- We have already compared P against S_{left} and S_{right}, and we know that $S_{left} \le P, S_{mid} \le S_{right}$.
- By using lcp-comparisons, we know $lcp(S_{left}, P)$ and $lcp(P, S_{right})$.

By Lemmas 1.31 and 1.38,

$$lcp(P, S_{mid}) \ge lcp(S_{left}, S_{right}) = \min\{lcp(S_{left}, P), lcp(P, S_{right})\}$$

Thus we can skip $\min\{lcp(S_{left}, P), lcp(P, S_{right})\}$ first characters when comparing P and S_{mid}.

Algorithm 1.39: String binary search (without precomputed lcps)
Input: Ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$, query string P.
Output: The number of strings in $\mathcal{R}$ that are smaller than P.

$left \leftarrow 0$; $right \leftarrow n + 1$
$llcp \leftarrow 0$ // $llcp = lcp(S_{left}, P)$
$rlcp \leftarrow 0$ // $rlcp = lcp(P, S_{right})$
while $right - left > 1$ **do**
  $mid \leftarrow \lfloor (left + right)/2 \rfloor$
  $mlcp \leftarrow \min\{llcp, rlcp\}$
  $(x, mlcp) \leftarrow \text{LcpCompare}(S_{mid}, P, mlcp)$
  if $x = \text{“}<\text{”}$ **then** $left \leftarrow mid$; $llcp \leftarrow mclp$
  else $right \leftarrow mid$; $rlcp \leftarrow mclp$
return $left$

- The average case query time is now $\mathcal{O}(\log n)$.
- The worst case query time is still $\mathcal{O}(m \log n)$ (exercise).

We can further improve string binary search using precomputed information about the lcp's between the strings in $\mathcal{R}$.

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 this.

Lemma 1.40: Let A, B, B' and C be strings such that $A \le B \le C$ and $A \le B' \le C$.

(a) If $lcp(A, B) > lcp(A, B')$, then $B < B'$ and $lcp(B, B') = lcp(A, B')$.
(b) If $lcp(A, B) < lcp(A, B')$, then $B > B'$ and $lcp(B, B') = lcp(A, B)$.
(c) If $lcp(B, C) > lcp(B', C)$, then $B > B'$ and $lcp(B, B') = lcp(B', C)$.
(d) If $lcp(B, C) < lcp(B', C)$, then $B < B'$ and $lcp(B, B') = lcp(B, C)$.
(e) If $lcp(A, B) = lcp(A, B')$ and $lcp(B, C) = lcp(B', C)$, then $lcp(B, B') \ge \max\{lcp(A, B), lcp(B, C)\}$.

Proof. Cases (a)–(d) are symmetrical, we show (a). $B < B'$ follows from Lemma 1.32. Then by Lemma 1.31, $lcp(A, B') = \min\{lcp(A, B), lcp(B, B')\}$. Since $lcp(A, B') < lcp(A, B)$, we must have $lcp(A, B') = lcp(B, B')$.

In case (e), we use Lemma 1.31:

$$lcp(B, B') \ge \min\{lcp(A, B), lcp(A, B')\} = lcp(A, B)$$
$$lcp(B, B') \ge \min\{lcp(B, C), lcp(B', C)\} = lcp(B, C)$$

Thus $lcp(B, B') \ge \max\{lcp(A, B), lcp(B, C)\}$. □

Algorithm 1.41: String binary search (with precomputed lcps)
Input: Ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$, arrays LLCP and RLCP, query string P.
Output: The number of strings in $\mathcal{R}$ that are smaller than P.

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

Theorem 1.42: An ordered string set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ can be preprocessed in $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n)$ 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 $\mathcal{O}(lcp(S_{mid}, \mathcal{R} \setminus \{S_{mid}\}) + 1)$ time. Thus the arrays $LLCP$ and $RLCP$ can be computed in $\mathcal{O}(\Sigma lcp(\mathcal{R}) + n) = \mathcal{O}(\Sigma LCP(\mathcal{R}) + n)$ time and stored in $\mathcal{O}(n)$ space.

The main while loop in Algorithm 1.41 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 $\mathcal{O}(|P|)$. □

Other comparison-based data structures such as binary search trees can be augmented with lcp information in the same way (study groups).

Hashing and Fingerprints

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

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

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 $H(S) \ne H(T)$, then we must have $S \ne T$.
- If $H(S) = H(T)$, then $S = T$ and $S \ne T$ are both possible.
  If $S \ne T$, 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.

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 1.43: The Karp–Rabin hash function for a string $S = s_0 s_1 \ldots s_{m-1}$ over an integer alphabet 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 q and r.

Lemma 1.44: 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:

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

□

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 r are *coprime*, then r has a multiplicative inverse r^{-1} modulo q, and we can also compute $H(A) = ((H(AB) - H(B)) \cdot (r^{-1})^{|B|}) \bmod q$.

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 $q = 2^w$, 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. 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

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

A trie is a special type of an automaton.

- The root is the initial state, the leaves are accept states, ...
- Trie is generally not a *minimal* automaton.
- Trie techniques including path compaction can be applied to automata.

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 support set inclusion testing but not other trie operations:

- No insertions and deletions
- No satellite data, i.e., data associated to each string

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, 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 $LCP_{\mathcal{R}}$ and its sum $\Sigma LCP(\mathcal{R})$.
- Lcp-comparison technique.