Similarly, the $\Omega(\Sigma LCP(R) + n \log n)$ 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($R$)

**Input:** (Multi)set $R = \{S_1, S_2, \ldots, S_n\}$ of strings of length $m$ over alphabet $[0..\sigma)$

**Output:** $R$ in ascending lexicographical order.

1. for $\ell \leftarrow m - 1$ to 0 do CountingSort($R, \ell$)
2. return $R$

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

- The stability of counting sort is essential.

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

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.

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

**Example 1.28:** MSD radix sort partitioning.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>g</td>
<td>m</td>
<td>e</td>
</tr>
<tr>
<td>o</td>
<td>c</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>n</td>
<td>e</td>
<td>r</td>
<td>v</td>
</tr>
<tr>
<td>e</td>
<td>b</td>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>e</td>
<td>r</td>
<td>n</td>
</tr>
</tbody>
</table>

The LSD radix sort algorithm is very simple.

**Algorithm 1.29:** MSDRadixSort($R, \ell$)

**Input:** (Multi)set $R = \{S_1, S_2, \ldots, S_n\}$ of strings over the alphabet $[0..\sigma)$ and the length $\ell$ of their common prefix.

**Output:** $R$ in ascending lexicographical order.

1. if $|R| < \sigma$ then return StringQuickSort($R, \ell$)
2. $R_\ell \leftarrow \{S \in R \mid |S| = \ell\}$; $R \leftarrow R \setminus R_\ell$
3. $R_0, R_1, \ldots, R_{\sigma-1} \leftarrow$ CountingSort($R, \ell$)
4. for $i \leftarrow 0$ to $\sigma - 1$ do $R_i \leftarrow$ MSDRadixSort($R_i, \ell + 1$)
5. return $R_0, R_1, \ldots, R_{\sigma-1}$

- Here CountingSort($R, \ell$) not only sorts but also returns the partitioning based on symbols at position $\ell$. The time complexity is still $O(|R| + \sigma)$.

- The recursive calls eventually lead to a large number of very small sets, but counting sort needs $O(\sigma)$ time no matter how small the set is. To avoid the potentially high cost, the algorithm switches to string quicksort for small sets.

**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 use a standard comparison with an lcp-comparison, which is the operation LcpCompare($A, B, k$):

- The return value is the pair $(x, t)$, where $x \in \{<, =, >\}$ indicates the order, and $t = lcp(A, B)$, 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 $A$ and $B$ be strings.

- $lcp(A, C) \geq \min\{lcp(A, B), lcp(B, C)\}$
- If $A \leq B \leq C$, then $lcp(A, C) = \min\{lcp(A, B), lcp(B, C)\}$
- If $lcp(A, B) \neq lcp(B, C)$, then $lcp(A, C) = \min\{lcp(A, B), lcp(B, C)\}$

**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 $O(1 + \sigma) = O(k)$. The $k$ symbols accessed here will not be accessed again.
- At most $dp(S \setminus \{S_i\}) = lcp(S, R \setminus \{S_i\}) + 1$ symbols in $S$ will be accessed by the algorithm. Thus the total time spent in this kind of calls is $O(\Sigma dp(R)) = O(\Sigma dp(R) + n) = O(\Sigma LCP(R) + n)$.

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 $O(\Sigma LCP(R) + n \log n)$.

- There exists a more complicated variant of MSD radix sort with time complexity $O(\Sigma LCP(R) + n + \epsilon)$.

- $O(\Sigma LCP(R) + n)$ 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.

Theorem 1.30: MSD radix sort sorts a set $R$ of $n$ strings over the alphabet $[0..\sigma)$ in $O(\Sigma LCP(R) + n \log \sigma)$ 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 $O(1 + \sigma) = O(k)$. The $k$ symbols accessed here will not be accessed again.
- At most $dp(S \setminus \{S_i\}) = lcp(S, R \setminus \{S_i\}) + 1$ symbols in $S$ will be accessed by the algorithm. Thus the total time spent in this kind of calls is $O(\Sigma dp(R)) = O(\Sigma dp(R) + n) = O(\Sigma LCP(R) + n)$.

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 $O(\Sigma LCP(R) + n \log n)$.

**Theorem 1.30:** MSD radix sort sorts a set $R$ of $n$ strings over the alphabet $[0..\sigma)$ in $O(\Sigma LCP(R) + n \log \sigma)$ 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 $O(1 + \sigma) = O(k)$. The $k$ symbols accessed here will not be accessed again.
- At most $dp(S \setminus \{S_i\}) = lcp(S, R \setminus \{S_i\}) + 1$ symbols in $S$ will be accessed by the algorithm. Thus the total time spent in this kind of calls is $O(\Sigma dp(R)) = O(\Sigma dp(R) + n) = O(\Sigma LCP(R) + n)$.
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 $A \leq B \leq C$ and $A \leq B' \leq C$.
(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 $B \geq B'$. Then by Lemma 1.31, $lcp(A, B) = \min(lcp(A, B'), lcp(B, C)) \leq lcp(A, B')$, 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.

Algorithm 1.34: StringMerge($P, Q$)
Input: Sequences $P = (S_1, k_1), \ldots, (S_m, k_m)$ and $Q = (T_1, \ell_1), \ldots, (T_n, \ell_n)$
Output: Merged sequence $R$
(1) $R \leftarrow \emptyset$; $i \leftarrow 1$; $j \leftarrow 1$
(2) while $i \leq m$ and $j \leq n$ do
(3) if $k_i > \ell_j$ then append $(S_i, k_i)$ to $R$; $i \leftarrow i + 1$
(4) else if $\ell_j > k_i$ then append $(T_j, \ell_j)$ to $R$; $j \leftarrow j + 1$
(5) else $f / k_i = \ell_j$
(6) $(x, h) \leftarrow LcpCompare(S_i, T_j, k_i, \ell_j)$
(7) if $x = x'$ then
(8) append $(S_i, k_i)$ to $R$; $i \leftarrow i + 1$
(9) $\ell_j \leftarrow h$
(10) else
(11) append $(T_j, \ell_j)$ to $R$; $j \leftarrow j + 1$
(12) $k_i \leftarrow h$
(13) while $i \leq m$ do append $(S_i, k_i)$ to $R$; $i \leftarrow i + 1$
(14) while $j \leq n$ do append $(T_j, \ell_j)$ to $R$; $j \leftarrow j + 1$
(15) return $R$

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

Proof. If the calls to LcpCompare took constant time, the time complexity would be $O(n \log n)$ by the same argument as with the standard mergesort. However, LcpCompare makes more than one call, 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 $\Sigma LCP(R)$, the extra time spent in LcpCompare is bounded by $O(\Sigma LCP(R))$.

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

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 $A \leq B \leq C$ and $A \leq B' \leq C$. Then $lcp(B, C) \geq lcp(B', C)$.

Proof. Let $B_{\min} = \min(B, B')$ and $B_{\max} = \max(B, B')$. By Lemma 1.31, $lcp(A, C) = \min(lcp(A, B_{\min}), lcp(B_{\max}, C)) \leq lcp(A, B_{\max}) = \min(lcp(A, B_{\min}), lcp(B_{\min}, B_{\max})) \leq lcp(B_{\min}, B_{\max}) = lcp(B, B')$ □

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

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 $\ell_1 = 0$. Other words, $\ell_i = LCP_R[i]$. 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.

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 $R$ (or $\epsilon$ if $R = \emptyset$). Then $k_i = lcp(S_i, X)$ 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 $lcp(S_i, X) > lcp(X, T_j)$ and thus $S_i < T_j$ by Lemma 1.32.
- $lcp(S_i, T_j) = lcp(X, T_j)$ because, by Lemma 1.31, $lcp(X, T_j) = \min(lcp(S_i, X), lcp(S_i, T_j))$.

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 $lcp(S_i, T_j) \geq k_i$ 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. □

String Binary Search
An ordered array is a simple static data structure supporting queries in $O(\log n)$ 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) $mid \leftarrow \lfloor(right + left) / 2\rfloor$
(4) if $k_{mid} > x$ then $left \leftarrow mid$
(5) else $right \leftarrow mid$
(6) return left

With strings as elements, however, the query time is

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

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 $S_{left} \leq P \leq S_{right}$.
- By using lcp-comparisons, we know $lcp(S_{left}, P)$ and $lcp(P, S_{right})$

By Lemmas 1.31 and 1.38, $lcp(P, S_{mid}) \geq \min(lcp(S_{left}, P), lcp(P, S_{right}))$. 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 \( R = \{S_1, S_2, \ldots, S_n\} \), query string \( P \).
Output: The number of strings in \( R \) that are smaller than \( P \):
1. \( \text{left} \leftarrow 0 \); \( \text{right} \leftarrow n + 1 \)
2. \( \text{leftp} \leftarrow 0 \) /\( \text{leftp} = \text{lcp}(S_{\text{left}}, P) \)
3. \( \text{rightp} \leftarrow 0 \) /\( \text{rightp} = \text{lcp}(P, S_{\text{right}}) \)
4. while \( \text{right} - \text{left} > 1 \) do
5. \( \text{mid} \leftarrow \lfloor(\text{left} + \text{right})/2 \rfloor \)
6. \( \text{leftcp} \leftarrow \text{min}(\text{leftp}, \text{rightp}) \)
7. \( (x, \text{leftcp}) \leftarrow \text{LcpCompare}(S_{\text{left}}, P, \text{leftp}) \)
8. if \( x = "<" \) then \( \text{left} \leftarrow \text{mid} \); \( \text{leftp} \leftarrow \text{leftcp} \)
9. else \( \text{right} \leftarrow \text{mid} \); \( \text{rightp} \leftarrow \text{rightp} \)
10. return \( \text{left} \)

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

Algorithm 1.41: String binary search (with precomputed lcps)
Input: Ordered string set \( R = \{S_1, S_2, \ldots, S_n\} \), arrays \( \text{LLCP} \) and \( \text{RLCP} \), query string \( P \). Output: The number of strings in \( R \) that are smaller than \( P \):
1. \( \text{left} \leftarrow 0 \); \( \text{right} \leftarrow n + 1 \)
2. \( \text{leftp} \leftarrow 0 \) \( \text{rightp} \leftarrow 0 \)
3. while \( \text{right} - \text{left} > 1 \) do
4. \( \text{mid} \leftarrow \lfloor(\text{left} + \text{right})/2 \rfloor \)
5. \( \text{leftcp} \leftarrow \text{min}(\text{left}, \text{right}) \)
6. if \( x = "<" \) then \( \text{left} \leftarrow \text{mid} \); \( \text{leftp} \leftarrow \text{leftcp} \)
7. else \( \text{right} \leftarrow \text{mid} \); \( \text{rightp} \leftarrow \text{rightp} \)
8. return \( \text{left} \)

Theorem 1.42: An ordered string set \( R = \{S_1, S_2, \ldots, S_n\} \) can be preprocessed in \( O(\text{LLCP}(R) + n) \) time and \( O(n) \) space so that a binary search with a query string \( P \) can be executed in \( O(P) + \log n) \) time.

Proof. The values \( \text{LLCP}[\text{mid}] \) and \( \text{RLCP}[\text{mid}] \) can be computed in \( O(\text{LLCP}(R) + n) \) time and stored in \( O(n) \) space.

Other comparison-based data structures such as binary search trees can be augmented with lcp comparison 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:

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

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 a hash table size.

Definition 1.43: The Karp–Rabin hash function for a string \( S = s_0 s_1 \ldots s_m \) over an integer alphabet is

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

for some fixed positive integers \( q \) and \( r \).

Lemma 1.44: For any two strings \( A \) and \( B \),

\[ H(AB) = (H(A) \cdot r^{|B|} + H(B)) \mod q \]

\[ H(B) = (H(AB) - H(A) \cdot r^{|B|}) \mod q \]

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 \( r \) are coprime, then \( r \) has a multiplicative inverse \( r^{-1} \mod q \), and we can also compute \( H(A) = ((H(AB) - H(B)) \cdot (r^{-1})^{|B|}) \mod q \).
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 $q = 2^w$, 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 $\lfloor \log q \rfloor / \log r$ 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

**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** $LCP_k$ and its sum $\Sigma LCP(R)$.
- Lcp-comparison technique.