The preceding lower bound does not hold for algorithms specialized for sorting strings.

**Theorem 1.13:** Let \( R = \{S_1, S_2, \ldots, S_n\} \) be a set of \( n \) strings. Sorting \( R \) into the lexicographical order by any algorithm based on symbol comparisons requires \( O(|LCP(R)| + n \log n) \) symbol comparisons.

**Proof.** If we are given the strings in the correct order and the job is to verify that this is indeed so, we need at least \( |LCP(R)| \) symbol comparisons. No sorting algorithm could possibly do its job with less symbol comparisons. This gives a lower bound \( \Omega(|LCP(R)|) \).

On the other hand, the general sorting lower bound \( \Omega(n \log n) \) must hold here too.

The result follows from combining the two lower bounds. \( \square \)

Note that the expected value of \( \Sigma LCP(R) \) for a random set of \( n \) strings is \( O(n \log n) \). The lower bound then becomes \( \Omega(n \log n) \).

We will next see that there are algorithms that match this lower bound. Such algorithms can sort a random set of strings in \( O(n \log n) \) time.

**String Quicksort (Multikey Quicksort)**
Quicksort is one of the fastest general purpose sorting algorithms in practice. Here is a variant of quicksort that partitions the input into three parts instead of the usual two parts.

**Algorithm 1.14:** TernaryQuicksort\( R \)

Input: \((\text{Multi})\{\bar{s} \mid \bar{s} \in R\} \) in arbitrary order.

Output: \( R \) in ascending order.

1. \( \text{if } |R| \leq 1 \) then return \( R \)
2. select a pivot \( \ell \in R \)
3. \( R_\ell \leftarrow \{s \in R \mid s < \ell \} \)
4. \( R_\ell \leftarrow \{s \in R \mid s = \ell \} \)
5. \( R_\ell \leftarrow \{s \in R \mid s > \ell \} \)
6. \( R_\ell \leftarrow \text{TernaryQuicksort}(R_\ell) \)
7. \( R_\ell \leftarrow \text{TernaryQuicksort}(R_\ell) \)
8. return \( R_\ell \cdot R_\ell \cdot R_\ell \)

In the initial call, \( \ell = 0 \).

**Proof of Theorem 1.17.** The time complexity is dominated by the symbol comparisons on lines (4)–(6). We charge the cost of each comparison either on a single symbol or on a string depending on the result of the comparison:

\[ S[i] = X[i] \]: Charge the comparison on the symbol \( S[i] \).

- Now the string \( S \) is placed in the set \( R_m \). The recursive call on \( R_m \) increases the common prefix length to \( i + 1 \). Thus \( S[i] \) cannot be involved in any future comparison and the total charge on \( S[i] \) is 1.
- Only \( \log_2(\log |S|) \) symbols in \( S \) can be involved in these comparisons. Thus the total number of symbol comparisons resulting equality is at most \( \Sigma LCP(R) = \Theta(\Sigma LCP(R)) \).

(Exercise: Show that the number is exactly \( \Sigma LCP(R) \).)

\[ S[i] \neq X[i] \]: Charge the comparison on the string \( S \).

- Now the string \( S \) is placed in the set \( R_u \) or \( R_d \). The size of either set is at most \( |R|/2 \) assuming an optimal choice of the pivot \( X \).
- Every comparison charged on \( S \) halves the size of the set containing \( S \), and hence the total charge accumulated by \( S \) is at most \( \log n \).
- Thus the total number of symbol comparisons resulting inequality is at most \( O(n \log n) \). \( \square \)

**Radix Sort**
The \( O(n \log n) \) sorting lower bound does not apply to algorithms that use stronger operations than comparison. A basic example is counting sort for sorting integers.

**Algorithm 1.18:** CountingSort\( R \)

Input: \((\text{Multi})\{\bar{s} \mid \bar{s} \in R\} \) of integers from the range \([0..\sigma]\).

Output: \( R \) in nondecreasing order in array \( J[0..n] \).

1. \( i \leftarrow 0 \) to \( n \) do \( C[k] \leftarrow 0 \)
2. \( s \leftarrow 0 \) to \( n \) do \( C[s] \leftarrow C[s] + 1 \)
3. \( \text{cumulative sums} \)
4. \( \text{distribute} \)
5. \( \text{return} \)

The time complexity is \( O(n + \sigma) \).

Counting sort is a stable sorting algorithm, i.e., the relative order of equal elements stays the same.

Similarly, the \( O(\Sigma LCP(R) + n \log n) \) 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 1.19: 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 \( \ell = m - 1 \) 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 1.20:** \( R = \{\text{cat}, \text{him}, \text{bat}\} \).

<table>
<thead>
<tr>
<th>( S )</th>
<th>( \ell )</th>
<th>( k_i )</th>
<th>( \ell )</th>
<th>( k_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>0</td>
<td>h</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>him</td>
<td>0</td>
<td>a</td>
<td>a</td>
<td>h</td>
</tr>
<tr>
<td>bat</td>
<td>0</td>
<td>h</td>
<td>t</td>
<td>a</td>
</tr>
</tbody>
</table>

It is easy to show that after \( \ell \) rounds, the strings are sorted by suffix of length \( \ell \). 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 1.22:** MSD radix sort partitioning.

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{hab}, \text{eth} )</th>
<th>( \text{orth}, \text{ma} )</th>
<th>( \text{gin} )</th>
<th>( \text{ment} )</th>
<th>( \text{ocate} )</th>
<th>( \text{ernative} )</th>
<th>( \text{ernative} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Theorem 1.24:** 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 \( O(k + \sigma) = O(k) \). The \( k \) symbols accessed here will not be accessed again.
- At most \( dp(S, R \setminus \{S\}) \leq lcp(S, R \setminus \{S\}) + 1 \) symbols in \( S \) will be accessed by the algorithm. Thus the total time spent in this kind of calls is \( O(\Sigma dp(S)) = O(\Sigma lcp(R) + n) = O(\Sigma LCP(R) + n) \).

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

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

**Algorithm 1.23: 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. \( R \leftarrow \{S \in R \mid |S| = \ell\} \); \( R \leftarrow R \setminus R_\ell \)
3. \( R_\ell \leftarrow \{R_1, R_2, \ldots, R_{|R| - 1}\} \leftarrow \text{CountingSort}(R, \ell) \)
4. For \( i = 0 \) to \( n - 1 \) do \( R_i \leftarrow \text{MSDRadixSort}(R_i, \ell + 1) \)
5. Return \( R_\ell \cdot R_1 \cdot R_2 \cdots R_{|R| - 1} \)

- 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 \( \Omega(\sigma) \) time no matter how small the set is. To avoid the potentially high cost, the algorithm switches to string quicksort for small sets.

**String Mergesort**

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.

String mergesort takes the opposite approach. It replaces a standard string comparison with an lcp-comparison, which is the operation \( \text{LcpCompare}(A, B, k) \):

- The return value is the pair \((x, \ell)\), 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.

**Lemma 1.25:** Let \( A, B \) and \( C \) be strings.

- \( \text{lcp}(A, C) \geq \min(\text{lcp}(A, B), \text{lcp}(B, C)) \).
- \( \text{lcp}(A, C) = \min(\text{lcp}(A, B), \text{lcp}(B, C)) \) if \( A \leq B \) and \( B \leq C \).

**Proof.** Assume \( \ell = \text{lcp}(A, B) < \text{lcp}(B, C) \). The opposite case \( \text{lcp}(A, B) \geq \text{lcp}(B, C) \) is symmetric.

- Now \( |A[0: \ell]| = B[0: \ell] = C[0: \ell] \) and thus \( \text{lcp}(A, C) \geq \ell \).
- Either \( |A| = \ell \) or \( A[0: \ell] \leq B[0: \ell] \leq C[0: \ell] \). In either case, \( \text{lcp}(A, C) = \ell \).
String mergesort has the same structure as the standard mergesort: sort the first half and the second half separately, and then merge the results.

**Algorithm 1.27:** 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. \(m \leftarrow \lceil n/2 \rceil\)
3. \(P \leftarrow \text{StringMergesort}\left(\{S_1, S_2, \ldots, S_m\}\right)\)
4. \(Q \leftarrow \text{StringMergesort}\left(\{S_{m+1}, S_{m+2}, \ldots, S_n\}\right)\)
5. return StringMerge\((P, Q)\)

The output is of the form

\((T_1, t_1), (T_2, t_2), \ldots, (T_n, t_n)\)

where \(t_i = \text{lcp}(T_i, T_{i+1})\) for \(i > 1\) and \(t_1 = 0\). In other words, \(t_i = \text{LCP}(P_i)\).

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 1.29:** 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):

\[
\text{Let } X \text{ be the last string appended to } R \text{ (or } c \text{ if } R = \emptyset). \text{ Then } k_i = \text{lcp}(X, S_i) \text{ and } t_j = \text{lcp}(X, T_j). 
\]

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 \(k_i > t_j\), then \(\text{lcp}(X, S_i) > \text{lcp}(X, T_j)\) and thus
  - \(S_i < T_j\) by Lemma 1.26.
  - \(\text{lcp}(S_i, T_j) = \text{lcp}(X, T_j)\) because, by Lemma 1.25, \(\text{lcp}(X, S_i), \text{lcp}(X, T_j))\).

Hence, the algorithm chooses the smaller string and maintains the invariant. The case \(t_j > k_i\) is symmetric.

- If \(k_i = t_j\), then clearly \(\text{lcp}(S_i, T_j) = k_i\) and the call to LcpCompare is safe, and the smaller string is chosen. The update \(t_j \leftarrow h \text{ or } k_i \leftarrow h\) maintains the invariant. \(\square\)

**String Binary Search**

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

**Algorithm 1.31:** Binary search

**Input:** Ordered set \(R = \{k_1, k_2, \ldots, k_m\}\), query value \(x\).

**Output:** The number of elements in \(R\) that are smaller than \(x\).

1. left ← 0; right ← \(n-1\) // output value is in the range [left..right]
2. while right < left do
3. mid ← \([(\text{left} + \text{right})/2]\)
4. if \(k_{\text{mid}} < x\) then \(\text{left} \leftarrow \text{mid}\)
5. else \(\text{right} \leftarrow \text{mid}\)
6. return \(\text{left}\)

With strings as elements, however, the query time is

- \(O(m \log n)\) in the worst case for a query string of length \(m\)
- \(O(m + \log n \log n)\) 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 \(S_{\text{mid}}\).
- We have already compared \(P\) against \(S_{\text{left}}\) and \(S_{\text{right}}\), and we know that \(S_{\text{left}} < P \leq S_{\text{mid}} \leq S_{\text{right}}\).
- By using lcp-comparisons, we know \(\text{lcp}(S_{\text{left}}, P)\) and \(\text{lcp}(P, S_{\text{right}})\).

By Lemmas 1.25 and 1.32,

\[\text{lcp}(P, S_{\text{mid}}) \geq \max(\text{lcp}(S_{\text{left}}, P), \text{lcp}(P, S_{\text{right}}))\]

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

**Algorithm 1.28:** StringMerge\((P, Q)\)

**Input:** Sequences \(P = (S_1, k_1), \ldots, (S_n, k_n)\) and \(Q = (T_1, t_1), \ldots, (T_n, t_n)\)

**Output:** Merged sequence \(R\)

1. \(R \leftarrow \emptyset; \) i ← 1; j ← 1
2. while \(i \leq m \text{ and } j \leq n\) do
3. if \(k_i > t_j\) then append \((S_i, k_i)\) to \(R\); i ← i + 1
4. else if \(t_j > k_i\) then append \((T_j, t_j)\) to \(R\); j ← j + 1
5. else \(\text{if } k_i = t_j\)
6. \(x = \text{lcpCompare}(S_i, T_j, k_i, t_j)\)
7. if \(x = "<"\) then
8. append \((S_i, k_i)\) to \(R\); i ← i + 1
9. else
10. append \((T_j, t_j)\) to \(R\); j ← j + 1
11. while \(i \leq m\) do append \((S_i, k_i)\) to \(R\); i ← i + 1
12. while \(j \leq n\) do append \((T_j, t_j)\) to \(R\); j ← j + 1
13. return \(R\)

**Theorem 1.30:** String mergesort sorts a set \(R\) of \(n\) strings in \(O(\Sigma \text{LCP}(R) + n \log n)\) 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.

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

- Other comparison based sorting algorithms, for example heapsort and insertion sort, 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 1.32:** Let \(A, B, B'\) and \(C\) be strings such that \(A \leq B \leq C\) and \(A \leq B' \leq C\). Then \(\text{lcp}(B, B') \geq \text{lcp}(A, C)\).

**Proof.** Let \(B_{\text{min}} = \min(B, B')\) and \(B_{\text{max}} = \max(B, B')\). By Lemma 1.25, \(\text{lcp}(A, C) = \min(\text{lcp}(A, B_{\text{max}}), \text{lcp}(B_{\text{min}}, C))\)

\[\leq \min(\text{lcp}(A, B_{\text{max}}), \text{lcp}(B_{\text{min}}, B_{\text{max}}))\]
\[\leq \text{lcp}(B_{\text{min}}, B_{\text{max}}) = \text{lcp}(B, B')\]

**Algorithm 1.33:** 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. left ← 0; right ← \(n+1\)
2. lcp ← 0; rlc ← 0
3. while right > left do
4. mid ← \([(\text{left} + \text{right})/2]\)
5. lcp ← \(\min(\text{lcp}(P, \text{lcp}), \text{rLcp})\)
6. \(x, \text{mlcp}) = \text{LcpCompare}(S_{\text{mid}}, P, \text{mlcp})\)
7. if \(x = "<"\) then \(\text{left} \leftarrow \text{mid}; \text{lcp} \leftarrow \text{mlcp}\)
8. else \(\text{right} \leftarrow \text{mid}; \text{rLcp} \leftarrow \text{mlcp}\)
9. return \(\text{left}\)

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

$$\begin{align*}
LCP[mid] &= lcp(S_{left}, S_{mid}) \\
RCP[mid] &= lcp(S_{mid}, S_{right})
\end{align*}$$

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

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

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

1. $left ← 0$; $right ← n + 1$
2. $llcp ← 0$; $rlcp ← 0$
3. while $right − left > 1$ do
4.   $mid ← [(left + right)/2]$
5.   if $LCP[mid] < llcp$ then $left ← mid$
6.   else if $RLCP[mid] < rlcp$ then $left ← mid$
7.   else $RLCP[mid] > rlcp$ then $left ← mid$
8.   else $LCP[mid] > llcp$ then $right ← mid$
9.   else $llcp ← \max(llcp, rlcp)$
10.  $(x, mleq) ← LcpCompare(S_{mid}, P, mleq)$
11.   if $x = "<"$ then $left ← mid$; $llcp ← mleq$
12.   else $right ← mid$; $rlcp ← mleq$
13.  return $left$

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 1.34**: 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 $B < B'$ 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 $lcp(B, B') = \max\{lcp(A, B), lcp(B, C)\}$.

**Proof**: Cases (a)–(d) are symmetrical, we show (a). $B < B'$ follows from Lemma 1.26. Then by Lemma 1.25, $lcp(A, B') = \min\{lcp(A, B), lcp(B, B')\}$. Since $lcp(A, B') < lcp(A, B)$, we must have $lcp(A, B') = lcp(B, B')$.

In case (e), we use Lemma 1.25:

$$\begin{align*}
lcp(B, B') &\geq \min\{lcp(A, B), lcp(A, B')\} = lcp(A, B) \\
lcp(B, B') &\geq \min\{lcp(B, C), lcp(B', C)\} = lcp(B, C)
\end{align*}$$

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

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

**Proof**: The values $LCP[mid]$ and $RLCP[mid]$ can be computed in $O(\log n)$ time.

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, $mleq$ increases by $t$ on line (11). Then on lines (12)–(13), either $llcp$ or $rlcp$ increases by at least $t$, since $mleq$ 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 $O(|P|)$.