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

Theorem 1.13: Let $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ be a set of n strings. Sorting $\mathcal{R}$ into the lexicographical order by any algorithm based on symbol comparisons requires $\Omega(\Sigma LCP(\mathcal{R}) + n \log n)$ 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 $\Sigma LCP(\mathcal{R})$ symbol comparisons. No sorting algorithm could possibly do its job with less symbol comparisons. This gives a lower bound $\Omega(\Sigma LCP(\mathcal{R}))$.

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. □

- Note that the expected value of $\Sigma LCP(\mathcal{R})$ for a random set of n strings is $\mathcal{O}(n \log_\sigma n)$. 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 $\mathcal{O}(n \log n)$ 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: (Multi)set R in arbitrary order.
Output: R in ascending order.

```
if |R| ≤ 1 then return R
select a pivot x ∈ R
R< ← {s ∈ R | s < x}
R= ← {s ∈ R | s = x}
R> ← {s ∈ R | s > x}
R< ← TernaryQuicksort(R<)
R> ← TernaryQuicksort(R>)
return R< · R= · R>
```


In the normal, binary quicksort, we would have two subsets $R_\leq$ and $R_\geq$, both of which may contain elements that are equal to the pivot.

- Binary quicksort is slightly faster in practice for sorting sets.
- Ternary quicksort can be faster for sorting multisets with many duplicate keys. Sorting a multiset of size n with σ distinct elements takes $\mathcal{O}(n \log \sigma)$ comparisons (exercise).

The time complexity of both the binary and the ternary quicksort depends on the selection of the pivot (exercise).

In the following, we assume an optimal pivot selection giving $\mathcal{O}(n \log n)$ worst case time complexity.

String quicksort is similar to ternary quicksort, but it partitions using a single character position. String quicksort is also known as multikey quicksort.

Algorithm 1.15: StringQuicksort($\mathcal{R}, \ell$)

Input: (Multi)set $\mathcal{R}$ of strings and the length ℓ of their common prefix.
Output: $\mathcal{R}$ in ascending lexicographical order.

```
(1)  if |R| ≤ 1 then return R
(2)  R⊥ ← {S ∈ R | |S| = ℓ}; R ← R \ R⊥
(3)  select pivot X ∈ R
(4)  R< ← {S ∈ R | S[ℓ] < X[ℓ]}
(5)  R= ← {S ∈ R | S[ℓ] = X[ℓ]}
(6)  R> ← {S ∈ R | S[ℓ] > X[ℓ]}
(7)  R< ← StringQuicksort(R<, ℓ)
(8)  R= ← StringQuicksort(R=, ℓ + 1)
(9)  R> ← StringQuicksort(R>, ℓ)
(10) return R⊥ · R< · R= · R>
```

In the initial call, $\ell = 0$.

Example 1.16: A possible partitioning, when $\ell = 2$.

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

Theorem 1.17: String quicksort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n \log n)$ time.

- Thus string quicksort is an optimal symbol comparison based algorithm.
- String quicksort is also fast in practice.

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:

$S[\ell] = X[\ell]$: Charge the comparison on the symbol $S[\ell]$.

- Now the string S is placed in the set $\mathcal{R}_=$. The recursive call on $\mathcal{R}_=$ increases the common prefix length to $\ell + 1$. Thus $S[\ell]$ cannot be involved in any future comparison and the total charge on $S[\ell]$ is 1.
- Only $lcp(S, \mathcal{R} \setminus \{S\})$ symbols in S can be involved in these comparisons. Thus the total number of symbol comparisons resulting equality is at most $\Sigma lcp(\mathcal{R}) = \Theta(\Sigma LCP(\mathcal{R}))$. (Exercise: Show that the number is exactly $\Sigma LCP(\mathcal{R})$.)

$S[\ell] \neq X[\ell]$: Charge the comparison on the string S.

- Now the string S is placed in the set $\mathcal{R}_<$ or $\mathcal{R}_>$. The size of either set is at most $|\mathcal{R}|/2$ 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 $\mathcal{O}(n \log n)$. □

Radix Sort

The $\Omega(n \log n)$ sorting lower bound does not apply to algorithms that use stronger operations than comparisons. A basic example is counting sort for sorting integers.

Algorithm 1.18: CountingSort(R)

Input: (Multi)set $R = \{k_1, k_2, \ldots k_n\}$ of integers from the range $[0..\sigma)$.
Output: R in nondecreasing order in array $J[0..n)$.

```
for i ← 0 to σ − 1 do C[i] ← 0
for i ← 1 to n do C[k_i] ← C[k_i] + 1
sum ← 0
for i ← 0 to σ − 1 do          // cumulative sums
    tmp ← C[i]; C[i] ← sum; sum ← sum + tmp
for i ← 1 to n do          // distribute
    J[C[k_i]] ← k_i; C[k_i] ← C[k_i] + 1
return J
```

- The time complexity is $\mathcal{O}(n + \sigma)$.
- Counting sort is a stable sorting algorithm, i.e., the relative order of equal elements stays the same.

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 1.19: 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.

```
for ℓ ← m − 1 to 0 do CountingSort(R,ℓ)
return 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 1.20: $\mathcal{R} = \{\texttt{cat}, \texttt{him}, \texttt{ham}, \texttt{bat}\}$.

cat		hi m		h a m		b at
him	$\Longrightarrow$	ha m	$\Longrightarrow$	c a t	$\Longrightarrow$	c at
ham		ca t		b a t		h am
bat		ba t		h i m		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 1.21: 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. □

- 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 1.22: MSD radix sort partitioning.

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

Algorithm 1.23: 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.

```
if |R| < σ then return StringQuicksort(R, ℓ)
R⊥ ← {S ∈ R | |S| = ℓ}; R ← R \ R⊥
(R_0, R_1, ..., R_{σ−1}) ← CountingSort(R, ℓ)
for i ← 0 to σ − 1 do R_i ← MSDRadixSort(R_i, ℓ + 1)
return R⊥ · R_0 · R_1 ··· R_{σ−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 1.24: 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 $\mathcal{O}(\Sigma LCP(\mathcal{R}) + n \log \sigma)$. □

- 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.

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 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 1.25: 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 $lcp(A, C) = \min\{lcp(A, B), lcp(B, C)\}$.

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, $lcp(A, C) = \ell$. □

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

Lemma 1.26: 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 1.25, $lcp(A, B) = \min\{lcp(A, B'), lcp(B', B)\} \leq lcp(A, B')$, which is a contradiction. □

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($\mathcal{R}$)
Input: Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings.
Output: $\mathcal{R}$ sorted and augmented with $LCP_{\mathcal{R}}$ values.

if $|\mathcal{R}| = 1$ then return $((S_1, 0))$
$m \leftarrow \lfloor n/2 \rfloor$
$\mathcal{P} \leftarrow$ StringMergesort($\{S_1, S_2, \ldots, S_m\}$)
$\mathcal{Q} \leftarrow$ StringMergesort($\{S_{m+1}, S_{m+2}, \ldots, S_n\}$)
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 1.28: StringMerge($\mathcal{P}$,$\mathcal{Q}$)
Input: Sequences $\mathcal{P} = \big((S_1, k_1), \ldots, (S_m, k_m)\big)$ and $\mathcal{Q} = \big((T_1, \ell_1), \ldots, (T_n, \ell_n)\big)$
Output: Merged sequence $\mathcal{R}$

(1) $\mathcal{R} \leftarrow \emptyset$; $i \leftarrow 1$; $j \leftarrow 1$
(2) while $i \leq m$ and $j \leq n$ do
(3) if $k_i > \ell_j$ then append (S_i, k_i) to $\mathcal{R}$; $i \leftarrow i + 1$
(4) else if $\ell_j > k_i$ then append (T_j, ℓ_j) to $\mathcal{R}$; $j \leftarrow j + 1$
(5) else // $k_i = \ell_j$
(6) $(x, h) \leftarrow$ LcpCompare(S_i, T_j, k_i)
(7) if $x =$ "<" then
(8) append (S_i, k_i) to $\mathcal{R}$; $i \leftarrow i + 1$
(9) $\ell_j \leftarrow h$
(10) else
(11) append (T_j, ℓ_j) to $\mathcal{R}$; $j \leftarrow j + 1$
(12) $k_i \leftarrow h$
(13) while $i \leq m$ do append (S_i, k_i) to $\mathcal{R}$; $i \leftarrow i + 1$
(14) while $j \leq n$ do append (T_j, ℓ_j) to $\mathcal{R}$; $j \leftarrow j + 1$
(15) return $\mathcal{R}$

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):

> 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 1.26.
  - $lcp(S_i, T_j) = lcp(X, T_j)$ because, by Lemma 1.25, $lcp(X, T_j) = \min\{lcp(X, S_i), lcp(S_i, T_j)\}$.

  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 $lcp(S_i, T_j) \geq k_i$ 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. □

Theorem 1.30: 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 $1 + t$ 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 1.31: 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.

$left \leftarrow 0$; $right \leftarrow n + 1$ // output value is in the range $[left..right)$
while $right - left > 1$ do
$mid \leftarrow \lfloor (left + right)/2 \rfloor$
if $k_{mid} < x$ then $left \leftarrow mid$
else $right \leftarrow mid$
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
- $\mathcal{O}(m + \log n \log_\sigma n)$ 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 1.32: Let A, B, B' and C be strings such that $A \leq B \leq C$ and $A \leq B' \leq C$. Then $lcp(B, B') \geq lcp(A, C)$.

Proof. Let $B_{min} = \min\{B, B'\}$ and $B_{max} = \max\{B, B'\}$. By Lemma 1.25,

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

□

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

By Lemmas 1.25 and 1.32,

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

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

Algorithm 1.33: 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.

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

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

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 1.34: 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 1.26. 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:

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

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

Algorithm 1.35: 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 ← 0; right ← n + 1
(2)  llcp ← 0; rlcp ← 0
(3)  while right − left > 1 do
(4)      mid ← ⌊(left + right)/2⌋
(5)      if LLCP[mid] > llcp then left ← mid
(6)      else if LLCP[mid] < llcp then right ← mid; rlcp ← LLCP[mid]
(7)      else if RLCP[mid] > rlcp then right ← mid
(8)      else if RLCP[mid] < rlcp then left ← mid; llcp ← RLCP[mid]
(9)      else
(10)         mlcp ← max{llcp, rlcp}
(11)         (x, mlcp) ← LcpCompare(S_mid, P, mlcp)
(12)         if x = "<" then left ← mid; llcp ← mclp
(13)         else right ← mid; rlcp ← mclp
(14) return left
```


Theorem 1.36: 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 1.35 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 $\mathcal{O}(|P|)$. □