The standard ordering for strings is the *lexicographical order*. It is *induced* by an order over the alphabet. We will use the same symbols ($\leq$, $<$, $\geq$, $\not\leq$, etc.) for both the alphabet order and the induced lexicographical order. We define the lexicographical order using the closely related concept of the *longest common prefix*.

Definition 1.5: The length of the *longest common prefix* of two strings $A[0..m)$ and $B[0..n)$, denoted by $lcp(A,B)$, is the largest integer $\ell \leq \min\{m,n\}$ such that $A[0..\ell) = B[0..\ell)$.

Definition 1.6: Let A and B be two strings over an alphabet with a total order $\leq$, and let $\ell = lcp(A,B)$. Then A is *lexicographically* smaller than or equal to B, denoted by $A \leq B$, if and only if

1. either $|A| = \ell$
2. or $|A| > \ell$, $|B| > \ell$ and $A[\ell] < B[\ell]$.

The preorder of the nodes in a trie is the same as the lexicographical order of the strings they represent assuming the children of a node are ordered by the edge labels.

The concept of longest common prefixes can be generalized for sets:

Definition 1.7: For a string S and a string set $\mathcal{R}$, define

$$lcp(S,\mathcal{R}) = \max\{lcp(S,T) \mid T \in \mathcal{R}\}$$
$$lcp(\mathcal{R}) = \sum_{T\in\mathcal{R}} lcp(T, \mathcal{R} \setminus \{T\})$$

In the algorithm for inserting S into $\mathcal{R}$ (Algorithm 1.3), there are two while loops. The first loop follows existing edges as long as possible and the second loop then creates the necessary new nodes and edges. The number of rounds in the first loop is exactly $lcp(S,\mathcal{R})$. Thus the number of new nodes and edges added is $|S| - lcp(S,\mathcal{R})$.

Based on this observation, we will next derive an expression for the size of $trie(\mathcal{R})$. This is not directly based the value $lcp(\mathcal{R})$; we need a more refined measure.

Definition 1.8: Let $\mathcal{R} = \{S_1, S_2, \dots, S_n\}$ be a set of strings and assume $S_1 < S_2 < \cdots < S_n$. Then the LCP array $LCP_{\mathcal{R}}[1..n]$ is defined by

$$LCP_{\mathcal{R}}[i] = lcp(S_i, \{S_1, \dots, S_{i-1}\}) \ .$$

Furthermore, let

$$L(\mathcal{R}) = \sum_{i\in[1..n]} LCP[i] \ .$$

Example 1.9: Let $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$. Then $L(\mathcal{R}) = 7$ and the LCP array is:

LCP	
0	`pot`
3	`potato`
3	`pottery`
0	`tattoo`
1	`tempo`

Theorem 1.10: The number of nodes in $trie(\mathcal{R})$ is exactly $||\mathcal{R}|| - L(\mathcal{R}) + 1$, where $||\mathcal{R}||$ is the total length of the strings in $\mathcal{R}$.

Proof. Consider the construction of $trie(\mathcal{R})$ by inserting the strings one by one in the lexicographical order. Initially, the trie has just one node, the root. As observed earlier, the number of nodes added when inserting S_i is $|S_i| - LCP_{\mathcal{R}}[i]$. Summing up, we get the result. $\square$

The value $lcp(\mathcal{R})$ is perhaps a conceptually simpler measure than $L(\mathcal{R})$. The following result shows that it is asymptotically equivalent.

Lemma 1.11: $L(\mathcal{R}) \leq lcp(\mathcal{R}) \leq 2L(\mathcal{R})$.

The proof is left as an exercise.

We will later see that the array $LCP_{\mathcal{R}}$ is useful as an actual data structure.

Compact Trie

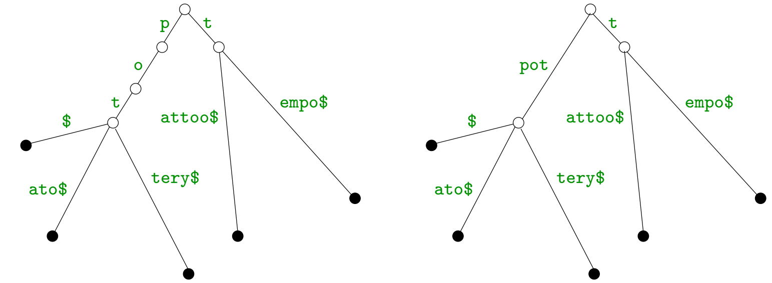
Tries suffer from a large number of nodes, $\Omega(||\mathcal{R}||)$ in the worst case. The space requirement can be problematic, since typically each node needs much more space than a single symbol.

Path compacted tries reduce the number of nodes by replacing *branchless path segments* with a single edge.

- Leaf path compaction applies this to path segments leading to a leaf. The number of nodes is now $|\mathcal{R}| + lcp(\mathcal{R}) - L(\mathcal{R}) + 1$ (exercise).
- Full path compaction applies this to all path segments. Then every internal node (except possibly the root) has at least two children. In such a tree, there is always at least as many leaves as internal nodes. Thus the number of nodes is at most $2|\mathcal{R}|$.

The full path compacted trie is called a *compact trie*.

Example 1.12: Path compacted tries for
$\mathcal{R} = \{\texttt{pot\$}, \texttt{potato\$}, \texttt{pottery\$}, \texttt{tattoo\$}, \texttt{tempo\$}\}$.



The egde labels are factors of the input strings. If the input strings are stored separately, the edge labels can be represented in constant space using pointers to the strings.

The time complexity of the basic operations on the compact trie is the same as for the trie (and depends on the implementation of the child operation in the same way), but prefix and range queries are faster on the compact trie (exercise).

Ternary Trie

The binary tree implementation of a trie supports ordered alphabets but awkwardly. Ternary trie is a simpler data structure based on symbol comparisons.

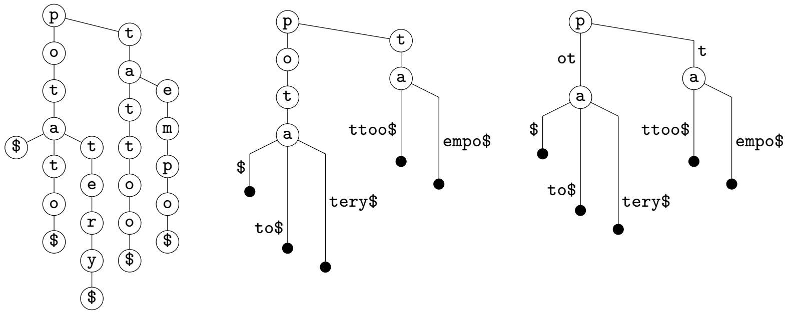
Ternary trie is like a binary search tree except:

- Each internal node has three children: smaller, equal and larger.
- The branching is based on a single symbol at a given position. The position is zero (first symbol) at the root and increases along the middle branches.

Ternary trie has variants similar to σ-ary trie:

- A basic ternary trie is a full representation of the strings.
- Compact ternary tries reduce space by compacting branchless path segments.

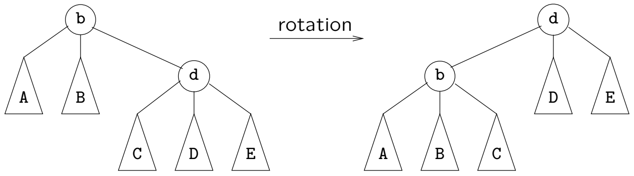
Example 1.13: Ternary tries for
$\mathcal{R} = \{\texttt{pot\$}, \texttt{potato\$}, \texttt{pottery\$}, \texttt{tattoo\$}, \texttt{tempo\$}\}$.



Ternary tries have the same asymptotic size as the corresponding tries.

A ternary trie is *balanced* if each left and right subtree contains at most half of the strings in its parent tree.

- The balance can be maintained by *rotations* similarly to binary search trees.



- We can also get reasonably close to balance by inserting the strings in the tree in a random order.

In a balanced ternary trie each step down either

- moves the position forward (middle branch), or
- halves the number of strings remaining in the subtree (side branch).

Thus, in a balanced ternary trie storing n strings, any downward traversal following a string S passes at most $|S|$ middle edges and at most $\log n$ side edges.

Thus the time complexity of insertion, deletion, lookup and lcp query is $\mathcal{O}(|S| + \log n)$.

In comparison based tries, where the *child* function is implemented using binary search trees, the time complexities could be $\mathcal{O}(|S| \log \sigma)$, a multiplicative factor $\mathcal{O}(\log \sigma)$ instead of an *additive factor* $\mathcal{O}(\log n)$.

Prefix and range queries behave similarly (exercise).

String Sorting

$\Omega(n \log n)$ is a well known *lower bound* for the number of comparisons needed for sorting a set of n objects by any comparison based algorithm. This lower bound holds both in the worst case and in the average case.

There are many algorithms that match the lower bound, i.e., sort using $\mathcal{O}(n \log n)$ comparisons (worst or average case). Examples include quicksort, heapsort and mergesort.

If we use one of these algorithms for sorting a set of n strings, it is clear that the number of *symbol comparisons* can be more than $\mathcal{O}(n \log n)$ in the *worst case*. Determining the order of A and B needs at least $lcp(A, B)$ symbol comparisons and $lcp(A, B)$ can be arbitrarily large in general.

On the other hand, the *average* number of symbol comparisons for two random strings is $\mathcal{O}(1)$. Does this mean that we can sort a set of *random strings* in $\mathcal{O}(n \log n)$ time using a standard sorting algorithm?

The following theorem shows that we cannot achieve $\mathcal{O}(n \log n)$ symbol comparisons for *any* set of strings (when $\sigma = n^{o(1)}$).

Theorem 1.14: Let $\mathcal{A}$ be an algorithm that sorts a set of objects using only comparisons between the objects. Let $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ be a set of n strings over an ordered alphabet Σ of size σ. Sorting $\mathcal{R}$ using $\mathcal{A}$ requires $\Omega(n \log n \log_\sigma n)$ symbol comparisons on average, where the average is taken over the initial orders of $\mathcal{R}$.

- If σ is considered to be a constant, the lower bound is $\Omega(n(\log n)^2)$.

- Note that the theorem holds for any comparison based sorting algorithm $\mathcal{A}$ and any string set $\mathcal{R}$.

- Only the initial order is random rather than "any". Any order could be the correct order, in which case an algorithm that first checks if the order is correct would need to do only $\mathcal{O}(n + L(\mathcal{R}))$ symbol comparisons.

An intuitive explanation for this result is that the comparisons made by a sorting algorithm are *not random*. In the later stages, the algorithm tends to compare strings that are close to each other in lexicographical order and thus are likely to have long common prefixes.

Proof of Theorem 1.14. Let $k = \lfloor (\log_\sigma n)/2 \rfloor$. For any string $\alpha \in \Sigma^k$, let $\mathcal{R}_\alpha$ be the set of strings in $\mathcal{R}$ having α as a prefix. Let $n_\alpha = |\mathcal{R}_\alpha|$.

Let us analyze the number of symbol comparisons when comparing strings in $\mathcal{R}_\alpha$ *against each other*.

- Each string comparison needs at least k symbol comparisons.
- No comparison between a string in $\mathcal{R}_\alpha$ and a string outside $\mathcal{R}_\alpha$ gives any information about the relative order of the strings in $\mathcal{R}_\alpha$.
- Thus $\mathcal{A}$ needs to do $\Omega(n_\alpha \log n_\alpha)$ string comparisons and $\Omega(kn_\alpha \log n_\alpha)$ symbol comparisons to determine the relative order of the strings in $\mathcal{R}_\alpha$.

Thus the total number of symbol comparisons is $\Omega\left(\sum_{\alpha\in\Sigma^k} kn_\alpha \log n_\alpha\right)$ and

$$\sum_{\alpha\in\Sigma^k} kn_\alpha \log n_\alpha \geq k(n-\sqrt{n}) \log \frac{n-\sqrt{n}}{\sigma^k} \geq k(n-\sqrt{n})\log(\sqrt{n}-1)$$
$$= \Omega\left(kn\log n\right) = \Omega\left(n \log n \log_\sigma n\right) \ .$$

Here we have used the facts that $\sigma^k \leq \sqrt{n}$, that $\sum_{\alpha\in\Sigma^k} n_\alpha > n - \sigma^k \geq n - \sqrt{n}$, and that $\sum_{\alpha\in\Sigma^k} n_\alpha \log n_\alpha > (n-\sqrt{n})\log((n-\sqrt{n})/\sigma^k)$ (see exercises). □

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

Theorem 1.15: 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 $\Omega(L(\mathcal{R}) + n\log n)$ 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 $L(\mathcal{R})$ symbol comparisons. No sorting algorithm could possibly do its job with less symbol comparisons. This gives a lower bound $\Omega(L(\mathcal{R}))$.

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 $L(\mathcal{R})$ 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 1.16: TernaryQuicksort(R)
Input: (Multi)set R in arbitrary order.
Output: R in ascending order.
(1) *if* $|R| \leq 1$ *then return* R
(2) select a pivot $x \in R$
(3) $R_< \leftarrow \{s \in R \mid s < x\}$
(4) $R_= \leftarrow \{s \in R \mid s = x\}$
(5) $R_> \leftarrow \{s \in R \mid s > x\}$
(6) $R_< \leftarrow$ TernaryQuicksort($R_<$)
(7) $R_> \leftarrow$ TernaryQuicksort($R_>$)
(8) *return* $R_< \cdot R_= \cdot R_>$

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

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

(1) *if* $|\mathcal{R}| \leq 1$ *then return* $\mathcal{R}$
(2) $\mathcal{R}_\perp \leftarrow \{S \in \mathcal{R} \mid |S| = \ell\}$; $\mathcal{R} \leftarrow \mathcal{R} \setminus \mathcal{R}_\perp$
(3) select pivot $X \in \mathcal{R}$
(4) $\mathcal{R}_< \leftarrow \{S \in \mathcal{R} \mid S[\ell] < X[\ell]\}$
(5) $\mathcal{R}_= \leftarrow \{S \in \mathcal{R} \mid S[\ell] = X[\ell]\}$
(6) $\mathcal{R}_> \leftarrow \{S \in \mathcal{R} \mid S[\ell] > X[\ell]\}$
(7) $\mathcal{R}_< \leftarrow$ StringQuicksort($\mathcal{R}_<, \ell$)
(8) $\mathcal{R}_= \leftarrow$ StringQuicksort($\mathcal{R}_=, \ell + 1$)
(9) $\mathcal{R}_> \leftarrow$ StringQuicksort($\mathcal{R}_>, \ell$)
(10) return $\mathcal{R}_\perp \cdot \mathcal{R}_< \cdot \mathcal{R}_= \cdot \mathcal{R}_>$

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

Example 1.18: 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	$\Longrightarrow$	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

Theorem 1.19: String quicksort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(L(\mathcal{R}) + n \log n)$ time.

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

Proof of Theorem 1.19. 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 $lcp(\mathcal{R}) = \Theta(L(\mathcal{R}))$.
(Exercise: Show that the number is exactly $L(\mathcal{R})$.)

$S[\ell] \neq X[\ell]$**:** 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 1.20: 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 $J[0..n)$.

(1) *for* $i \leftarrow 0$ *to* $\sigma - 1$ *do* $C[i] \leftarrow 0$
(2) *for* $i \leftarrow 1$ *to* n *do* $C[k_i] \leftarrow C[k_i] + 1$
(3) $sum \leftarrow 0$
(4) *for* $i \leftarrow 0$ *to* $\sigma - 1$ *do* // cumulative sums
(5) $tmp \leftarrow C[i]$; $C[i] \leftarrow sum$; $sum \leftarrow sum + tmp$
(6) *for* $i \leftarrow 1$ *to* n *do* // distribute
(7) $J[C[k_i]] \leftarrow k_i$; $C[k_i] \leftarrow C[k_i] + 1$
(8) *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 $\Omega(L(\mathcal{R}) + n \log n)$ 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 (more on this in the exercises).

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 1.21: LSDRadixSort($\mathcal{R}$)
Input: (Multi)set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings of length m over the 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 1.22: $\mathcal{R} = \{\texttt{cat}, \texttt{him}, \texttt{ham}, \texttt{bat}\}$.

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

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

Theorem 1.23: 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 $\mathcal{R}$.

- The weakness of LSD radix sort is that it uses $\Omega(||\mathcal{R}||)$ time even when $L(\mathcal{R})$ 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 1.24: 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 1.25: MSDRadixSort($\mathcal{R}, \ell$)
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($\mathcal{R}, \ell$)
(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($\mathcal{R}, \ell$)
(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 1.26: MSD radix sort sorts a set $\mathcal{R}$ of n strings over the alphabet $[0..\sigma)$ in $\mathcal{O}(L(\mathcal{R}) + n \log \sigma)$ time.

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

- The time excluding the recursive call 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 $lcp(S, \mathcal{R} \setminus \{S\}) + 1$ symbols in S will be accessed by the algorithm. Thus the total time spent in this kind of calls is $\mathcal{O}(L(\mathcal{R}) + n)$.

This still leaves the time spent in the calls for a subsets of size $k < \sigma$, which are handled by string quicksort. No string is included in two such calls. Therefore, the total time over all calls is $\mathcal{O}(L(\mathcal{R}) + n \log \sigma)$.
□

- There exists a more complicated variant of MSD radix sort with time complexity $\mathcal{O}(L(\mathcal{R}) + n + \sigma)$.
- $\Omega(L(\mathcal{R}) + n)$ 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.