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 3.10: 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 $\Omega(DP(\mathcal{R}) + n \log n)$ 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).

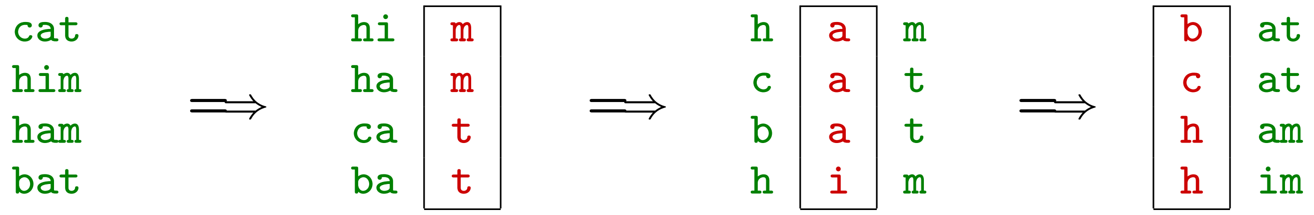
The LSD radix sort algorithm is very simple.

Algorithm 3.11: LSDRadixSort($\mathcal{R}$)

```
Input: Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings of length $m$ over the alphabet [0..$\sigma$).
Output: $\mathcal{R}$ in increasing lexicographical order.
  (1) for $\ell \leftarrow m-1$ to 0 do CountingSort($\mathcal{R}$,$\ell$)
  (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 is replaced by $S_i[\ell]$). The time complexity is $\mathcal{O}(|\mathcal{R}| + \sigma)$.
- The stability of counting sort is essential.

Example 3.12: $\mathcal{R} = \{$`cat`, `him`, `ham`, `bat`$\}$.



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

Theorem 3.13: 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 $DP(\mathcal{R})$ 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 3.14: 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 3.15: MSDRadixSort($\mathcal{R}, \ell$)
Input: Set $\mathcal{R} = \{S_1, S_2, \ldots, S_n\}$ of strings over the alphabet $[0..\sigma)$
       and the length $\ell$ of their common prefix.
Output: $\mathcal{R}$ in increasing 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 3.16: MSD radix sort sorts a set $\mathcal{R}$ of n strings over the alphabet $[0..\sigma)$ in $\mathcal{O}(DP(\mathcal{R}) + n \log \sigma)$ 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.
- The algorithm does not access any symbols outside the distinguishing prefixes. Thus the total time spent in this kind of calls is $\mathcal{O}(DP(\mathcal{R}))$.

This still leaves the time spent in the calls to string quicksort. The calls are for sets of size smaller than σ and no string is included two calls. Therefore, the total time over all calls is $\mathcal{O}(DP(\mathcal{R}) + n \log \sigma)$.

$\square$

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

Standard 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 using symbol comparisons where the constant time is in balance with the information value of the result.

String mergesort takes the opposite approach. It replaces a standard string comparison with the operation LcpCompare(A, B, k):

- The return value is the pair (x, ℓ), where $x \in \{<, =, >\}$ indicates the order, and ℓ is the length of the longest common prefix (lcp) of strings A and B, denoted by $lcp(A, B)$.

- 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.

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

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

Lemma 3.17: 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, $lcp(A,C) = \ell$.

$\square$

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

Lemma 3.18: Let $A \le B, B' \le C$ be strings.

(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 3.17, $lcp(A,B) = \min\{lcp(A,B'), lcp(B',B)\} \le lcp(A,B')$, which is a contradiction. $\square$

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

```
  (1) if $|\mathcal{R}| = 1$ then return $\{(S_1, 0)\}$
  (2) $k \leftarrow \lfloor n/2 \rfloor$
  (3) $\mathcal{P} \leftarrow$ StringMergesort($\{S_1, S_2, \ldots, S_k\}$)
  (4) $\mathcal{Q} \leftarrow$ StringMergesort($\{S_{k+1}, S_{k+2}, \ldots, S_n\}$)
  (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, 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 3.20: 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 $(T_j, \ell_j)$ 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 $(T_J, \ell_j)$ 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 $(T_J, \ell_j)$ to $\mathcal{R}$; $j \leftarrow j + 1$
(15) return $\mathcal{R}$
```

Lemma 3.21: 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 3.18.
  - $lcp(S_i, T_j) = lcp(X, T_j)$ because by Lemma 3.17 $lcp(X, T_j) = \min\{lcp(X, S_i), lcp(S_i, T_j)\}$.

  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 3.22: String mergesort sorts a set $\mathcal{R}$ of n strings in $\mathcal{O}(DP(\mathcal{R}) + n \log n)$ 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. The lcp value stored with a string S cannot become larger than $dp_{\mathcal{R}}(S)$. Therefore, the extra time spent in LcpCompare is bounded by $\mathcal{O}(DP(\mathcal{R}))$.

$\square$

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