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 replace 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 ∈ {<, =, >} indicates the order, and ℓ = 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.

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

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

**Lemma 1.31:** Let A, B, B' and C be strings such that A ≤ B ≤ C and A ≤ B' ≤ 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 ≥ B'. Then by Lemma 1.30, lcp(A, B) = min{lcp(A, B'), lcp(B, C)} ≤ lcp(A, B'), which is a contradiction. □

Intuitively, the above result makes sense if you think of lcp(·, ·) as a measure of similarity between two strings. The higher the lcp, the closer the two strings are lexicographically.

**Algorithm 1.33:** StringMerge(P, Q)

Input: Sequences P = ((S₁, k₁), ..., (Sₙ, kₙ)) and Q = ((T₁, l₁), ..., (Tₗ, lₗ))

Output: Merged sequence R

1. R ← ∅; i ← 1; j ← 1
2. while i ≤ n and j ≤ l do
3.   if kᵢ > lⱼ then append (Sᵢ, kᵢ) to R; i ← i + 1
4.   else if kᵢ < lⱼ then append (Tⱼ, lⱼ) to R; j ← j + 1
5.   else if kᵢ = lⱼ then
6.     (x, h) ← LcpCompare(Sᵢ, Tⱼ)
7.     if x = "=" then
8.       append (Sᵢ, kᵢ) to R; i ← i + 1
9.       ℓᵢ ← h
10.    else
11.       append (Tⱼ, lⱼ) to R; j ← j + 1
12.      ℓⱼ ← h
13.    while i ≤ n do append (Sᵢ, kᵢ) to R; i ← i + 1
14.    while j ≤ l do append (Tⱼ, lⱼ) to R; j ← j + 1
15.   return R

**Theorem 1.35:** String merge sort sets a set R of n strings in O(lcpΣP(R) + log n) time.

**Proof.** If the calls to LcpCompare took constant time, the time complexity would be O(lcpΣP(R)) by the same argument as with the standard mergesort. Whenever LcpCompare makes more than one, say i + 1, symbol comparisons, one of the lcp values stored with the strings increases by 1. Since the sum of the final lcp values is exactly lcpΣP(R), the extra time spent in LcpCompare is bounded by O(lcpΣP(R)).

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 1.32:** StringMergesort(R)

Input: Set R = {S₁, S₂, ..., Sₙ} of strings.

Output: R sorted and augmented with LCPₖ values.

1. if |R| = 1 then return ((S₁, 0))
2. m ← ⌊n/2⌋
3. P ← StringMergesort(S₁, S₂, ..., Sₘ)
4. Q ← StringMergesort(Sₘ₊₁, Sₘ+₂, ..., Sₙ)
5. return StringMerge(P, Q)

The output is of the form

((T₁₁, ℓ₁₁), ..., (Tₘ⁺ₗ, ℓₘ⁺ₗ))

where ℓᵢ = lcp(Tᵢ, Tᵢ₊�) for i > 1 and ℓ₁ = 0. In other words, ℓᵢ = LCPₖ(Pᵢ).

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 1.34:** 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 if R = ∅ then k₁ = lcp(X, S₁) and ℓ₁ = lcp(X, T₁)).

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 kᵢ > ℓⱼ then lcp(Sᵢ, Tⱼ) > lcp(X, Tⱼ) and thus
  - Sᵢ < Tⱼ by Lemma 1.31.
  - lcp(Sᵢ, Tⱼ) = lcp(X, Tⱼ) because, by Lemma 1.30, lcp(X, Tⱼ) = min{lcp(X, Sᵢ), lcp(X, Tⱼ)}.

Hence, the algorithm chooses the smaller string and maintains the invariant. The case kᵢ ≥ ℓⱼ is symmetric.

- If kᵢ = ℓⱼ, then clearly lcp(Sᵢ, Tⱼ) ≥ kᵢ, and the call to LcpCompare is safe, and the smaller string is chosen. The update ℓᵢ ← h or kᵢ ← 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 1.36:** Binary search

Input: Ordered set R = {k₁, k₂, ..., kₙ}, query value x.

Output: The number of elements in R that are smaller than x.

1. left ← 0; right ← n + 1
2. while left < right do
3.   mid ← ⌊(left + right + 1)/2⌋
4.   if kₘid < x then left ← mid
5.   else right ← mid
6. return 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(\log n log 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 1.37:** Let A, B, B' and C be strings such that A ≤ B ≤ C and A ≤ B' ≤ C. Then lcp(B, B') ≥ lcp(A, C).

**Proof.** Let \( B_{min} = \min\{B, B'\} \) and \( B_{max} = \max\{B, B'\} \). By Lemma 1.30, \( lcp(A, C) = \min\{lcp(A, B_{min}), lcp(B_{min}, C)\} \)
\( \leq lcp(A, B_{min}) = \min\{lcp(A, B_{max}), lcp(B_{max}, C)\} \)
\( \leq lcp(B_{min}, B_{max}) = lcp(B, B') \)
\( \square \)

**Algorithm 1.38:** 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. \( \text{lcp} \leftarrow 0 \)
3. \( \text{if} \ lcp > \text{lcp}(S_{\text{left}}, P) \)
4. \( \text{while} \ \text{right} - \text{left} > 1 \)
5. \( \text{mid} \leftarrow \left\lfloor (\text{left} + \text{right})/2 \right\rfloor \)
6. \( \text{mllcp} \leftarrow \text{lcp}(S_{\text{left}}, P) \)
7. \( \text{rlcp} \leftarrow \text{lcp}(S_{\text{mid}}, P) \)
8. \( \text{if} \ \text{llcp} > \text{rlcp} \)
9. \( \text{else} \ \text{left} \leftarrow \text{mid}; \text{lcp} \leftarrow \text{mllcp} \)
10. \( \text{return} \ \text{left} \)

*The average case query time is now \( O(\log n) \) (exercise).*

**Algorithm 1.39:** Skewed 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. \( \text{lcp} \leftarrow 0 \)
3. \( \text{if} \ \text{lcp} = \text{lcp}(S_{\text{left}}, P) \)
4. \( \text{while} \ \text{right} - \text{left} > 1 \)
5. \( \text{if} \ \text{llcp} > \text{lcp} > \text{rlcp} \)
6. \( \text{mid} \leftarrow \left\lfloor (\text{left} + \text{right} + 1)/2 \right\rfloor \)
7. \( \text{mllcp} \leftarrow \text{lcp}(S_{\text{left}}, P) \)
8. \( \text{rlcp} \leftarrow \text{lcp}(S_{\text{right}}, P) \)
9. \( \text{if} \ \text{llcp} > \text{rlcp} \)
10. \( \text{else} \ \text{left} \leftarrow \text{mid}; \text{lcp} \leftarrow \text{mllcp} \)
11. \( \text{return} \ \text{left} \)

**Lemma 1.40:** Let A, B, B' and C be strings such that A ≤ B ≤ C and A ≤ B' ≤ 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 1.31. Then by Lemma 1.30, lcp(A, B') = \min\{lcp(A, B), lcp(B, B')\}.

In case (e), we use Lemma 1.30: lcp(B, B') ≥ \min\{lcp(A, B), lcp(B, B')\} = lcp(A, B)
lcp(B, B') ≥ \min\{lcp(B, C), lcp(B, C')\} = lcp(B, C)

Thus lcp(B, B') ≥ \max\{lcp(A, B), lcp(B, C)\}.

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 \( lcp(S_{\text{left}}, P) \) and \( lcp(P, S_{\text{right}}) \).

By Lemmas 1.30 and 1.37, lcp\( (P, S_{\text{mid}}) \geq \min\{lcp(S_{\text{left}}, P), lcp(P, S_{\text{right}})\} \)
Thus we can skip \min\{lcp(S_{\text{left}}, P), lcp(P, S_{\text{right}})\} first characters when comparing \( P \) and \( S_{\text{mid}} \).

We can improve the worst case complexity by choosing the midpoint closer to the smaller lcp value:

- If \( \text{llcp} - \text{rlcp} > 1 \), choose the middle position closer to the right.
- This is achieved by choosing the midpoint as weighted average of the left position and the right position. The weights are \( d = \text{lcp} - \text{rlcp} \).
- If \( \text{llcp} - \text{rlcp} > 1 \), choose the middle position closer to the left in a symmetric way.
- The worst case time complexity of the resulting algorithm (shown on the next slide) is \( O(m \log \log n) \). The proof is omitted here.
- The lower bound on string binary searching time has been shown to be \( \Theta\left(\frac{m \log \log n}{\log \log \log n} + m + \log n\right) \).

There is a complicated algorithm achieving this time complexity.

The lower bound above assumes that no other information besides the ordering of the strings is given. We can further improve string binary searching by 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 \( S_{\text{mid}} \).
- We have already compared \( P \) against \( S_{\text{left}} \) and \( S_{\text{right}} \), and we know \( lcp(S_{\text{left}}, P) \) and \( lcp(P, S_{\text{right}}) \).

In the unskewed algorithm, the values \( \text{left} \) and \( \text{right} \) are fully determined by \( \text{mid} \) independently of \( P \). That is, \( P \) only determines whether the search ends up at position \( \text{mid} \) at all, but if it does, \( \text{left} \) and \( \text{right} \) are always the same. Thus, we can precompute and store the values

\[
\text{LLCP}[\text{mid}] = \text{lcp}(S_{\text{left}}, S_{\text{mid}})
\]
\[
\text{RLCP}[\text{mid}] = \text{lcp}(S_{\text{mid}}, S_{\text{right}})
\]

Now we know all lcp values between \( P, S_{\text{left}}, S_{\text{mid}}, S_{\text{right}} \) except \( \text{lcp}(P, S_{\text{mid}}) \). The following lemma shows how to utilize this.

**Algorithm 1.41:** String binary search (with precomputed lcps)

**Input:** Ordered string set \( R = \{S_1, S_2, \ldots, S_n\} \), arrays LLCP and 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. \( \text{lcp} \leftarrow 0 \)
3. \( \text{if} \ \text{lcp} = \text{lcp}(S_{\text{left}}, P) \)
4. \( \text{while} \ \text{right} - \text{left} > 1 \)
5. \( \text{mid} \leftarrow \left\lfloor (\text{left} + \text{right})/2 \right\rfloor \)
6. \( \text{if} \ \text{LLCP}[\text{mid}] > \text{lcp} \leftarrow \text{left} \leftarrow \text{mid}; \text{lcp} \leftarrow \text{RLCP}[\text{mid}] \)
7. \( \text{else} \ \text{if} \ \text{RLCP}[\text{mid}] > \text{lcp} \leftarrow \text{right} \leftarrow \text{mid}; \text{lcp} \leftarrow \text{LLCP}[\text{mid}] \)
8. \( \text{else} \ \text{llcp} \leftarrow \text{mllcp}; \text{rlcp} \leftarrow \text{mllcp} \)
9. \( \text{else} \ \text{left} \leftarrow \text{mid}; \text{lcp} \leftarrow \text{mllcp} \)
10. \( \text{return} \ \text{left} \)

The following algorithm shows how to utilize this.
Theorem 1.42: 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 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 $O(\log(\Sigma \text{mid}(R) \setminus \{\text{mid}(R)\}) + 1)$ time. Thus the arrays $\text{LLCP}$ and $\text{RLCP}$ can be computed in $O(\Sigma LCP(R) + n)$ time and stored in $O(n)$ space.

The main while loop in Algorithm 1.41 is executed $O(\log n)$ times and everything except LcpCompare on line (11) needs constant time.

Other comparison-based data structures such as binary search trees can be augmented with lcp information in the same way.

Automata

Finite automata are a well known way of representing sets of strings. In this case, the set is often called a (regular) language.

- No insertions and deletions
- No satellite data, i.e., data associated to each string

Any good hash function must depend on all characters. Thus computing $H(S)$ needs $\Omega(|S|)$ time, which can defeat the power of hash functions. An example is the Karp–Rabin hash function.

The parameters $q$ and $r$ have to be chosen with some care to ensure that collisions are rare for any reasonable set of strings.

- A hash value can be computed once, stored, and used many times.
- Some hash functions can be computed more efficiently for a related set of strings. An example is the Karp–Rabin hash function.

The original choice is $r = s$ 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. (But a bad case for this is a Thue-Morse sequence.)

- If $q$ and $r$ were both powers of two, then only the last $\lceil \log_2 q \rceil \log_2 r$ 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.

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 $\text{LCP}_y$ and its sum $\Sigma LCP(R)$.
- Lcp-comparison technique.

Hashing and Fingerprints

Hashing is a powerful technique for dealing with sets based on mapping each string to an integer using a hash function:

$$H : \Sigma^* \to [0, q) \subset \mathbb{N}$$

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 $S = s_0 s_1 \ldots s_{n-1}$ over an integer alphabet is

$$H(S) = (\sum_{i=0}^{n-1} (r^i s_i) \mod q \mod m/q) \mod q$$

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:

$$((x + y) \mod q) = ((x \mod q) + (y \mod q)) \mod q$$

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 $r^{-1}$ modulo $q$, and we can also compute $H(A) = (H(AB) - H(B)) \cdot (r^{-1})^{|B|} \mod q$.

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

Selected Literature

- Trie:
  Compact trie:
  Ternary trie and string quicksort:
  MSD radix sort in $O(\Sigma LCP(R) + n + \epsilon)$ time:
• String mergesort:

• Complexity of string binary search without precomputed lcp information:

• LCP array and string binary search using lcp information:

• Karp–Rabin hashing: