HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI
MATEMAATTIS-LUONNONTIETEELLINEN TIEDEKUNTA
MATEMATISK-NATURVETENSKAPLIGA FAKULTETEN
FACULTY OF SCIENCE

String Sorting in Python – Comparison of Several Algorithms

Onni Koskinen, Arturs Polis, and Lari Rasku

Comparison-based sorting is one of the most mature subfields of CS research. However, the more well-known of such algorithms have been designed with the expectation that the objects they sort can be compared in constant time. When used to sort objects that require linear-time comparison operations, such as strings, they perform a lot of wasteful work that leads to suboptimal performance. For maximum efficiency, *string sorting algorithms* are needed.

We have implemented a family of three different string sorting algorithms in Python and compared their performance against Python's native Timsort [1] using four different datasets with two variants each.

ALGORITHMS

String processing algorithms distinguish themselves from naive comparison algorithms by maintaining knowledge of the lengths of the *longest common prefixes* (LCP) of pairs of input strings as they sort them, which they use to avoid redundant comparisons. The *LCP array* of a set of strings, by extension, is defined as follows:

> Given an ordered set of strings $S_1 < ... < S_n$, $LCP[1] = 0$ and $LCP[i]$ is the length of the longest common prefix of strings S_i and S_{i-1} when $i > 1$.

i	S_i	$LCP[i]$
1	actor	0
2	allocate	1
3	alpha	2
4	beta	0
5	byproduct	1

Observe that were one to confirm that the set of strings in the above example is indeed sorted, one would have to check exactly the highlighted characters plus one for every string, with the first string excepted. In fact, $\Omega(L(R) + n)$ represents the lower bound for any algorithm that must access symbols one at a a time, where $L(R)$ is the sum of the LCP array for a set of strings R with n elements. In comparison, the average lower bound for sorting strings using only naive comparisons is $\Omega(n(\log n)^2)$.

MSD RADIX SORT

MSD radix sort first partitions the strings into different buckets based on what their first symbol is, then recursively partitions *those* buckets based on what the second symbol is, and so on. When only single-element buckets or buckets containing only strings shorter than the recursion depth are left, the results are concatenated and output.

MSD radix sort never needs to process a symbol twice, technically giving it $O(L(R) + n)$ complexity assuming a finite alphabet. However, the complexity is dominated by the bucket container data structure: if σ is the size of the alphabet and if the buckets are stored in a binary search tree, each addition takes $O(\log \sigma)$ time. If they are stored in an array or a hash table, merging takes $\Theta(\sigma)$ time.

Our implementation uses fixed arrays of size 256 but falls back to string quicksort when the size of the bucket falls below the size of the alphabet, avoiding wasteful allocation and merging. This gives it a time complexity of $O(L(R) + n \log \sigma)$.

BURSTSORT

Burstsort partitions the input strings into growable but limited-sized buckets that are afterwards sorted using an auxiliary sorting algorithm. Unlike many distribution sorts, burstsort reads the input sequentially in order to avoid cache misses. The partitioning limits the input length for the auxiliary sort and can thus improve its cache-efficiency as well. Burstsort's worst case complexity depends on the implementation details.

Burstsort uses a *burst trie* to store the buckets and maintain their order. This data structure is a trie whose leaf nodes are growable buckets containing pointers to the suffixes of the inserted strings while the path from the root to the leaf node via single-character edges represents the prefix of the strings just as with a regular trie.

A new operation, *burst*, is needed when a full bucket is already at the maximum capacity and a new string is being inserted into it. Burst creates a new bucket for each character in the alphabet and redistributes the suffixes in the bursting bucket into them.

Our implementation uses a fixed alphabet of 256 symbols, a maximum bucket size of 32768 strings, and uses in-place multikey quicksort to sort the buckets.

QUICKSORT

String quicksort operates by recursively partioning the list of strings corresponding to their lexicographical order compared to the pivot string. It sorts strings in time $O(L(R) + n \log n)$.

Good pivot selection reduces the maximum depth of the recursion tree. For quicksort we selected the pivot as the median of the first, the last, and the middle string in the list.

Ternary quicksort partitions the strings based on whole string comparison. Resulting sets are:

- Equal to the pivot
- Lexicographically smaller than the pivot
- Lexicographically larger than the pivot

Multikey quicksort partitions using single character comparison, an extra partition for strings with the currently compared character being their last is created.

Both algorithms recursively partition the strings until each partition contains only one string. Number of string comparisons in ternary quicksort is of $O(n \log n)$, with overall time complexity depending on the choice of the pivot.

DATASETS

With the exception of the URLS dataset [2], all datasets were retrieved from the Pizza & Chili Corpus [3]; the URLS dataset is the one used by Ranjan Sinha in his original burstsort paper [4]. The algorithms were tested on a sample of 100 and 200 megabytes with each dataset.

DNA

The DNA dataset consists of sequences of nucleotide codes, all exactly 3732300 characters in length. This is by far the easiest dataset, having the smallest number of strings and the smallest LCP array sum; very little of the extremely long strings is actually required for sorting them.

PROTEINS

The PROTEINS dataset consists of sequences of amino acid codons of varying lengths. It can be perceived as a more challenging variant of the DNA dataset, having two orders of magnitude more strings and an LCP array sum four whole orders of magnitude larger while retaining a similar alphabet.

URLS

The URLS dataset consists of several web addresses. Due to most common URLs having similar prefixes, as well as the dataset containing several duplicate URLs, this dataset has the highest LCP array sum, though not significantly higher than the WORDS dataset.

WORDS

The WORDS dataset is a modification of the ENGLISH dataset of the Pizza & Chili Corpus, constructed by splitting each word on its own line in order to make our algorithms sort individual words instead of entire lines. The dataset thus consists of very many very short strings, with a few outliers due to formatting markup in the source file. The dataset also ranks second highest in LCP array sum size and highest in alphabet size, due to common words appearing hundreds of times in the text and some loan words using characters not in the English alphabet.