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 $\Omega(n(logn)^2)$.

MSD RADIX SORT

MSD radix sort first partitions the strings into different buckets based on 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.

Highlight and underline these to illustrate the partitioning.

actor
allocate
alpha
beta
byproduct

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.

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. The strings are returned in the reverse order of recursion resulting in a sorted array of strings, with the number of string comparisons being $O(n \log n)$.

BURST SORT

Burst sort text Burst sort text Burst sort text
Burst sort text Burst sort text Burst sort text Burst sort text Burst sort text
Burst sort text Burst sort text
Burst sort text Burst sort text
Burst sort text

DATASETS

With the exception of the URLs dataset, all datasets were retrieved from the Pizza & Chili Corpus [2]; the URL dataset [4] is the one used by Ranjan Sinha in his original burstsort paper [3]. 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.

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.