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 using a variety of different datasets.

ALGORITHMS

MSD RADIX SORT

MSD (most significant digit) radix sort is a divide-and-conquer algorithm that partitions the strings based on their character at a given position. The comparison position starts from 0 and increases with one at every recursion level. No position, then, is visited twice; and if the algorithm does not attempt to partition buckets of size 1 or consisting entirely of strings shorter than the recursion depth, each string is visited at most one more time than the length of its shortest distinguishing prefix. Thus, the partitioning takes at most $O(L(R) + n)$ time, where $L(R)$ is the sum of the LCP array.
However, efficient implementations require the buckets to be implemented as an array of linked lists in order to avoid the overhead of binary search tree insertions and lookups. This allows true constant time insertion to buckets, but wastes time and memory if the strings use only a fraction of the alphabet for which MSD radix sort allocates space. Likewise, if the number of strings is smaller than the size of the alphabet, standard comparison based string sorting algorithms outperform MSD radix sort.
Our implementation uses a fixed alphabet size of 256 and falls back to ternary quicksort when the size of the bucket drops below it.

QUICKSORT

String quicksort operates by recursively paritioning the collection of strings corresponding to their lexicographical order compared to the pivot string.

Good pivot selection seeks to reduce the maximum depth of the recursion tree. For quicksort we selected the pivot as the median of the first, the last, and the middle element of an array of strings.

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 the strings using a single character comparison. To do this multikey quicksort maintains the index of the character currently being compared. The resulting partitioning is almost the same as in the ternary quicksort, multikey quicksort keeps an extra partition for strings with the currently compared character being their last.
The sets are then recursively partitioned again and again until each partition contains only one string. After that the strings are returned in the reverse order of recursion resulting in a sorted array of strings.
Quicksort sorts strings in time $O(L(R) + n \log n)$.

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 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.

REFERENCES

[1] U. Lauther and T. Lukovszki. Space efficient algorithms for the Burrows-Wheeler backtransformation. In *Proc. 13th Annual European Symposium on Algorithms*, volume 3669 of *LNCS*, pages 293–304. Springer, 2005.

[2] J. Seward. Space-time tradeoffs in the inverse B-W transform. In *Proc. IEEE Data Compression Conference*, pages 439–448. IEEE, 2001.

[3] U. Lauther and T. Lukovszki. Space efficient algorithms for the Burrows-Wheeler backtransformation. In *Proc. 13th Annual European Symposium on Algorithms*, volume 3669 of *LNCS*, pages 293–304. Springer, 2005.

[4] J. Seward. Space-time tradeoffs in the inverse B-W transform. In *Proc. IEEE Data Compression Conference*, pages 439–448. IEEE, 2001.

[5] U. Lauther and T. Lukovszki. Space efficient algorithms for the Burrows-Wheeler backtransformation. In *Proc. 13th Annual European Symposium on Algorithms*, volume 3669 of *LNCS*, pages 293–304. Springer, 2005.

[6] J. Seward. Space-time tradeoffs in the inverse B-W transform. In *Proc. IEEE Data Compression Conference*, pages 439–448. IEEE, 2001.