Journal of Statistical Software

May 2015, Volume 65, Code Snippet 1. *http://www.jstatsoft.org/*

LazySorted: A Lazily, Partially Sorted Python List

Naftali Harris
Stanford University

Abstract

LazySorted is a Python C extension implementing a partially and lazily sorted list data structure. It solves a common problem faced by programmers, in which they need just part of a sorted list, like its middle element (the median), but sort the entire list to get it. **LazySorted** presents them with the abstraction that they are working with a fully sorted list, while actually only sorting the list partially with quicksort partitions to return the requested sub-elements. This enables programmers to use naive "sort first" algorithms but nonetheless attain linear run-times when possible. **LazySorted** may serve as a drop-in replacement for the built-in `sorted` function in most cases, and can sometimes achieve run-times more than 7 times faster.

Keywords: partial sorting, lazy evaluation, Python.

1. Introduction

Many important statistical quantities are functions of order statistics. Order statistics like the median, the quartiles, or the deciles are by themselves useful for data summary purposes. Slightly more complicated functions of the order statistics include truncated means, which compute the mean of a series of points from say the 5th to the 95th percentiles, making them more robust than ordinary means.

Linear time algorithms exist to compute a particular order statistic, both in expectation (Hoare 1961), and in worst-case analysis (Blum, Floyd, Pratt, Rivest, and Tarjan 1973). Armed with such a linear time selection algorithm, all of these above quantities can be computed in linear time as well. (The truncated mean can be computed by determining the 5th and 95th percentiles in linear time, and then passing through the array once to compute the mean of all values between these percentiles). Despite the presence of these algorithms, however, it is a common pattern to see programmers simply sorting the entire list and then working with the computed order statistics to compute these sorts of quantities.

The **LazySorted** package (Harris 2014) is a C extension for Python (van Rossum *et al.* 2011) which presents the programmer with a data structure that is conceptually a sorted list. However, the data structure is only partially sorted physically. Whenever the programmer requests elements from it, it uses quicksort partitions to partially sort itself just enough to return the requested elements. It also keeps track of the partial sorting it has done so that later calls may use work done in earlier calls. This allows the programmer to interact with the **LazySorted** object in the naive way, as though it were a sorted list, but actually end up employing efficient algorithms.

We have carefully written **LazySorted** to support Python 2.5, 2.6, 2.7, 3.1, 3.2, 3.3, and 3.4, and have also thoroughly unit-tested it with all these versions. The source code is freely available under a BSD license from `https://github.com/naftaliharris/lazysort/`. It may also be installed through PyPI with `$ easy_install lazysorted` or `$ pip install lazysorted`.

2. Application programming interface (API)

The **LazySorted** object has a constructor that implements the same interface as Python's built-in `sorted` function. The resulting **LazySorted** object supports most of the non-mutating methods of a Python list.

In particular, the constructor has the signature

```
LazySorted(iterable, key = None, reverse = False)
```

with the same argument meanings as for the `sorted` function. The object supports the `__iter__`, `__contains__`, `__getitem__`, `__index__`, and `__count__`, and `__len__` methods, which respectively allow one to iterate through the data structure, check if the data structure contains a particular element, select an item from the data structure by its index, find the smallest index of a particular item in the data structure, count the occurrences of a particular element in the data structure, and determine the data structure's length, just as in a Python list.

Like the `sorted` function, **LazySorted** does not alter the iterable it is asked to sort.

These careful considerations make **LazySorted** a drop-in replacement for the `sorted` function in most cases. There are, however, a few differences between **LazySorted** and the built-in `sorted` function and list objects that it imitates.

Most significantly, the **LazySorted** object is immutable. There are two reasons for this: First, if **LazySorted** were to support the mutating `insert`, `append`, or `extend` methods, then to keep the **LazySorted** object sorted the added elements would need to be put into their sorted location. That would make these methods behave differently in **LazySorted** objects than they do in ordinary Python list's, potentially leading to confusion and programming errors. Second, since the **LazySorted** container is implemented as an array, insertion of elements into the middle of the array would require $O(n)$ time. This is not competitive with other data structures like skip-lists or balanced binary search trees, for which insertion costs $O(\log n)$.

A second difference between **LazySorted** and the Python built-ins it imitates is that **LazySorted** sorts are not stable, (i.e., it does not preserve the order of elements that compare equal), while those in Python are. This is due to the nature of the partition operation. To make **LazySorted** sorting stable, however, one need only employ the standard trick of sorting the original elements paired together with their original indices.

A third difference is that **LazySorted** has an extra method, `between(i, j)`, which returns a list of all the items whose sorted indices are between indices `i` and `j`, but not necessarily in any order. This allows one to compute truncated means efficiently, for example.

A final, small difference, is that **LazySorted** implements the Python 3 versions of the `sorted` and list APIs. This means that it does not support the `cmp` argument in its constructor, and does not support the `__getslice__` method, both of which were removed in Python 3.

3. Example

For most use cases, one simply uses **LazySorted** as one would the `sorted` function. For example, the following is a function that efficiently computes the largest or smallest k elements of an iterable:

```
from lazysorted import LazySorted

def top_k(xs, k, key = None, reverse = False):
    ls = LazySorted(xs, key = key, reverse = reverse)
    return ls[0:k]
```

We will see an example of how **LazySorted** may be used to create a median function later.

4. Implementation

In short, **LazySorted** works by partially sorting subsets of the list with quicksort partitions, and keeping track of the pivot elements.

A quicksort partition is a routine that operates on a list in place. It takes a pivot element from the list, and moves all of the elements less than that element to the left, and all elements greater or equal than it to the right. This may be achieved in linear time and constant space by passing through the list once and swapping elements (see Section 4.3 for the explicit algorithm). Quicksort (Hoare 1962) is a sorting algorithm that partitions the data into two subsets with the partition algorithm, and then recursively sorts the two parts with quicksort. Quickselect (Hoare 1961) on the other hand, is an algorithm for finding the kth smallest element of a list. It works by picking a pivot element and partitioning the data, as in quicksort. Then the algorithm recurses into the larger or smaller part of the list, depending on whether k is larger or smaller than the index of the pivot element.

There are two key observations we can make from these algorithms: First of all, if we are only interested in part of a sorted list, we only need to recurse into the part we are interested in after doing a partition. Second of all, after doing some partitions, the list is partially sorted, with the pivots all in their sorted order and the elements between two pivots guaranteed to be bigger than the pivot to their left and smaller than the pivot to their right.

LazySorted stores the location of the pivots resulting from each of its partition operations. When elements are requested from the **LazySorted** list, we first check the pivot store to see how well we can bound the location of the requested elements. Then, we need only search for the elements within bounded locations.

4.1. Partitioning optimizations

As detailed by Sedgewick (1978), there are a number of optimizations one can employ to improve the practical performance of quicksort. We employ the well-used median-of-three trick for pivot selection, in which the pivot used for the partitioning is determined by selecting three random elements and choosing the pivot to be their median. This leads to pivots that are closer to the middle of the array, decreasing the total number of partitions that need be completed. Still greater improvement could perhaps be attained by using the optimal pivot selection strategies described by Martínez and Roura (2001).

The second optimization we use, also suggested by Sedgewick (1978), is to default to insertion sort when the array to sort is small. Insertion sort works by starting with the leftmost element of the array, which is by itself a sorted array of length one. The algorithm then grows this sorted subarray from left to right by inserting each new element into its sorted place, until the entire array is sorted. Despite the fact that insertion sort runs in $O(n^2)$ expected time, using insertion sort on small lists leads to speed-ups because it has less overhead compared to quicksort and partitioning.

4.2. Storing the pivots

As observed above, the past pivots describe the partial structure of the **LazySorted** object. To keep track of this partial structure, we therefore keep track of the pivots. Now, between two pivots, the data may either be sorted or unsorted. If the data is known to be sorted, (perhaps from an earlier insertion sort, for example), then we want to remember this in case we need an element from this range in the future. So associated with each pivot are also attributes that describe whether the data to its left and right is known to be sorted or not. For completion, we also have faux-pivots at indices -1 and n, bordering the list, so that every element in the list is bounded by two pivots.

We store the pivots in a balanced Binary Search Tree for fast, $O(\log n)$ look-ups, insertions, and deletions. The implementation we chose is a Treap, (Aragon and Seidel 1989; Seidel and Aragon 1996), due to its ease of implementation and reputation for good performance under a number of insertions and deletions, as is common and expected in this application.

Throughout the code, we make sure to remove redundant pivots. For example, if we have pivots at indices 100, 200, and 300, and know from the pivot attributes that the data between 100 and 200 is sorted, and that the data between 200 and 300 is also sorted, then the pivot at index 200 is redundant. We can and would remove it, leaving us with just pivots 100 and 300, with sorted data between them. Removing redundant pivots like this ensures efficient behavior when the list is mostly sorted. For example, in the special case when the list is fully sorted, we will have no pivots except for the faux-pivots at indices -1 and n. Consequently, we recognize that the list is fully sorted in $O(1)$ time, and can therefore compute a requested order statistic in $O(1)$. Had there been n pivots in the sorted list, one for each index, then it would have taken $O(\log n)$ to determine that the requested order statistic was already in its sorted order, and hence $O(\log n)$ to return it.

A simple argument shows why the pivot storage does not accumulate much computational overhead: The worst case is that there are $O(n)$ pivots, (actually perhaps $\frac{n}{8}$ or $\frac{n}{16}$ depending on how high the threshold for insertion sort is set). If another selection operation is requested, this will result in another $O(\log n)$ partition operations, (though probably substantially fewer if there really were $O(n)$ pivots already there, since the partitioning could begin at a much smaller portion of the list). These $O(\log n)$ partitions each require $O(\log n)$ time to insert the resulting pivot points into the binary search tree (BST), yielding an overhead of at most $O(\log^2 n)$, which is not much compared to the $O(n)$ cost of computing all the partitions.

In fact, after starting with an unsorted list, the first selection operation adds $O(\log n)$ pivots to the pivot store, each of which then only costs $O(\log \log n)$, (since only $O(\log n)$ pivots are being added). This means that the pivot overhead for a single selection operation on an unsorted list is even less, $O(\log \log n \log n)$.

As a matter of practice, profiling our code has indicated that for reasonably sized lists, **LazySorted** spends the overwhelming majority of its time computing partitions, showing that the computational overhead of keeping track of pivots is indeed negligible. (Of course, keeping track of the pivots does require an extra $O(n)$ extra space in the worst case).

4.3. Prefetching

Python objects in the reference CPython implementation live in the heap and are passed around by reference. This means that to compare two Python objects, their pointers must be dereferenced and the resulting data compared based on their types. For larger lists that do not fit into cache, this presents a data-locality problem – as Python objects are compared, pointers are dereferenced to locations all throughout the heap, leading to a large number of cache misses. This leads to slow behavior for larger lists. To combat this, in our partition function we employ the `__builtin_prefetch` built-in function for GCC and Clang, which requests that the cache loads the referenced memory into cache in preparation for later use.

The following simplified excerpt of our partition function demonstrates its use; note that we prefetch memory from three steps forward in the list. This value was determined to be optimal by experimentation. Adding this prefetching operation yielded substantial speedups for large lists.

```
int partition(LSObject *ls, int left, int right) {
    PyObject **ob_item = ls->xs->ob_item;
    int piv_idx = pick_pivot(ls, left, right);
    PyObject *pivot = ob_item[piv_idx];

    SWAP(left, piv_idx);
    int last_less = left;

    for (int i = left + 1; i < right; i++) {
        __builtin_prefetch(ob_item[i + 3]);
        IF_LESS_THAN(ob_item[i], pivot) {
            last_less++;
            SWAP(i, last_less);
        }
    }

    SWAP(left, last_less);
    return last_less;
}
```

The invariant preserved by the partitioning in the for-loop is that the object at index `last_less`, and everything to its left, are less than the pivot or the pivot itself.

5. Other partial sort implementations

To the best of our knowledge, no other software package, in any language, implements this sort of data structure that permits arbitrary functions of the order statistics to be computed efficiently. However, algorithms for computing special cases of these functions certainly exist, and have been implemented in popular Python packages. In this section we discuss these implementations.

5.1. Python built-ins

Python comes with the built-in functions `sorted`, `min`, and `max`, with efficient C implementations that, respectively, sort iterable collections and find their extreme values. The `sorted` function uses the Timsort algorithm (Peters 2002), which can run faster than $O(n \log n)$ on lists with some partial ordering.

5.2. Package NumPy

The popular **NumPy** package (Oliphant *et al.* 2009) has a median and more general percentile function. Up until **NumPy** 1.7, they were implemented naively, by sorting the array and picking off the relevant elements. As of **NumPy** 1.8, however, they are implemented using linear time partitioning algorithms in C.

5.3. Module heapq

The **heapq** module in the standard library implements a priority queue, allowing one to efficiently compute the largest or smallest k elements of a collection of n elements in $O(n \log k)$ time by passing through the data once and maintaining a priority queue of the top k elements seen so far. The **heapq** module implements these algorithms explicitly in the `heapq.nlargest` and `heapq.nsmallest` functions, which are based on C implementations in the `_heapq` module.

5.4. Module statistics

As proposed in PEP 450, Python 3.4 contains a **statistics** module in the standard library, which contains a median function. However, this median function is computed using a complete sort based upon the `sorted` function, giving it a suboptimal $O(n \log n)$ run-time. As an example of the common programming practices employed by programmers, below is the source code of the release, (with the docstring omitted).

```
def median(data):
    data = sorted(data)
    n = len(data)
    if n == 0:
        raise StatisticsError("no median for empty data")
    if n%2 == 1:
        return data[n//2]
    else:
        i = n//2
        return (data[i - 1] + data[i])/2
```

Note that the implementation of the median function with the **LazySorted** package would be exactly the same, except that the `data = sorted(data)` line would be replaced by `data = LazySorted(data)`.

6. Performance

In this section, we compare the performance of **LazySorted** to different sorting and partial sorting implementations. In particular, we compare **LazySorted** to the `sorted` built-in, the **heapq** module, and to various sorting functions from **NumPy** 1.8, all under Python 2.7. These are run on a machine running Linux Mint 14 (nadia), with an Intel i5-3210M processor, (3M cache, up to 3.10 GHz), and 8 GB of RAM.

Our experiments compared the average time to compute various quantities on differently sized lists, with the list size varying from 10^1 to 10^7 by $10^{\frac{1}{2}}$ multiples, rounded to an odd integer to make the median unambiguous. The lists were composed of `float`'s generated with `random.random`, and then shuffled with `random.shuffle`. The reason for this shuffle is to destroy any data locality arising from the memory allocator. (Indeed, without this shuffle **LazySorted** and it competitors run noticeably faster).

6.1. Median

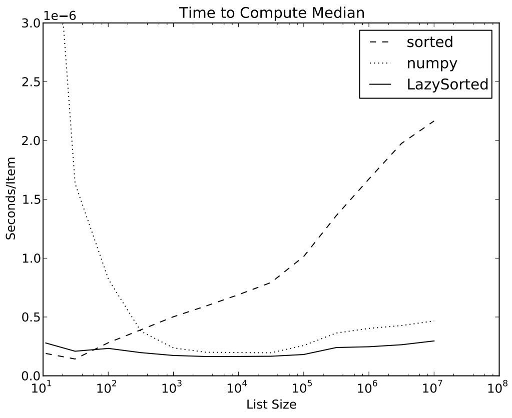
We compared the time taken to compute the median by **LazySorted**, a full sort with `sorted`, and `numpy.median`, summarizing the results in Figure 1a. The algorithms used for **LazySorted** and `sorted` were the same naive algorithm except that `sorted` was replaced by **LazySorted**.

Unsurprisingly, **LazySorted** is significantly faster than the naive full sort for all but small lists. Indeed, for the largest list with size 10,000,001, **LazySorted** is 7.3 times faster. However, **LazySorted** is also faster than `numpy.median`, which also uses a linear time algorithm. For the largest list, **LazySorted** is 1.6 times faster, although it is only 1.2 times faster for lists from sizes 3,163 to 31,623.

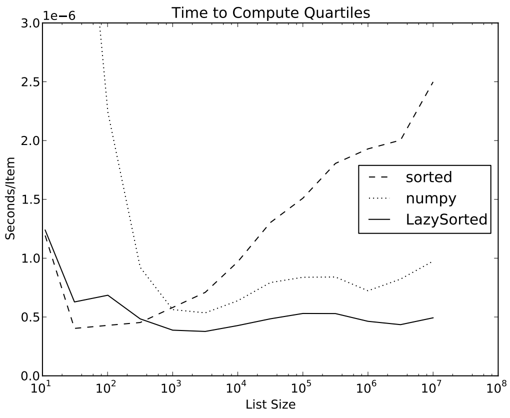
One unexpected observation from this analysis is that the `numpy.median` function has very, very poor performance for small lists. On lists of size 11, for example, it runs 18 times slower than **LazySorted**, and 26 times slower than a full sort with `sorted`.

6.2. Quartiles

We employed three different algorithms to compute the quartiles in this section. First, we used a full sort with the sorted function, and then picked off the elements with indices closest to the theoretical quartile indices, (which need not be integers). Second, we used the same algorithm except with **LazySorted** instead of full sorting. Third, we used the `numpy.median` and `numpy.percentile` functions sequentially, together with the ordinary `min` and `max` functions. Applying these linear time algorithms five times to compute the five quartiles yields a



(a) Medians.



(b) Quartiles.

Figure 1: Computing medians and quartiles.

linear time algorithm, but it is not expected to be as fast as **LazySorted** because later sorting does not take advantage of work done earlier. We summarize the results in Figure 1b.

For even moderately sized lists, not sorting the entire list yields substantial performance improvements over a complete sort for computing these quartiles. At lists of size 317, using **LazySorted** is almost as fast as `sorted`, (which takes 0.94 times the time), by size 1001, **LazySorted** is 1.5 times faster, and at the largest list size, 10,000,001, **LazySorted** is 5.1 times faster.

As before, **NumPy** has major performance issues dealing with small lists, running 15 times slower than **LazySorted** for lists of size 11, but then becoming faster until it is from 1.5 times to 2.0 times slower than **LazySorted** as the list size grows.

6.3. Top 10

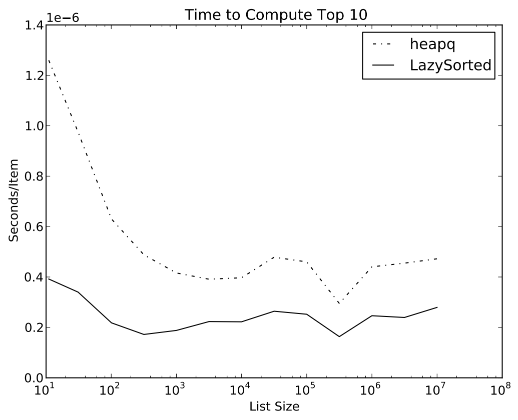
We also compared the performance of **LazySorted** and `heapq.nsmallest` in determining the top 10 smallest elements from the list.

As we see from Figure 2a, **LazySorted** performs better for all list sizes. `heapq.nsmallest` has some minor trouble for small lists, starting and remaining about 3 times slower than **LazySorted** until lists of size 1,001. For larger lists, **LazySorted** remains about 1.8 times faster. This performance improvement is somewhat surprising considering that the heap-based algorithm used to compute the top 10 elements is perhaps the most natural linear time algorithm for it.

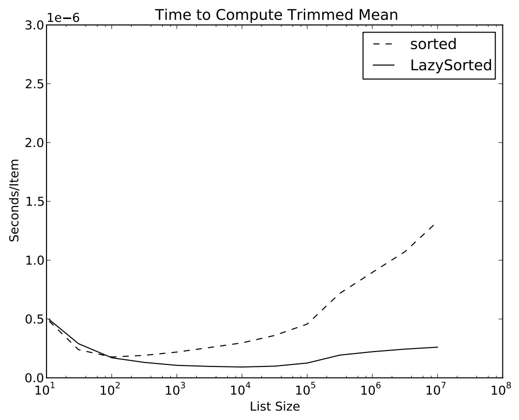
6.4. Trimmed mean

Finally, we compared the performance of **LazySorted** and `sorted` for determining the 0.05-trimmed mean, that is, the mean of the elements between the 5th and 95th percentiles. For **LazySorted**, we did this with the `between` method.

As seen in Figure 2b, `sorted` performs slightly better for small lists, but **LazySorted** begins to do substantially better on moderately sized and large lists, eventually becoming 5.1 times faster for lists of size 10,000,001.



(a) Top 10.



(b) 0.05-trimmed mean.

Figure 2: Computing the top 10 elements and the trimmed mean.

7. Conclusion

LazySorted is a software package allowing programmers to compute functions of an ordered list with the naive "sort first" algorithm, while still achieving efficient run-times. For all but small lists, this can lead to substantial speed-ups over complete sorts without changing the naive algorithms. Despite being designed for computing arbitrary functions of order statistics, **LazySorted** is still noticeably faster than functions from **NumPy** and **heapq** that are designed specifically to compute particular functions of order statistics.

Acknowledgments

I would like to thank Balasubramanian Narasimhan for his generosity in reviewing this document and his helpful suggestions for improving it.

References

Aragon CR, Seidel RG (1989). "Randomized Search Trees." In *30th Annual Symposium on Foundations of Computer Science, 1989*, pp. 540–545.

Blum M, Floyd RW, Pratt V, Rivest RL, Tarjan RE (1973). "Time Bounds for Selection." *Journal of Computer and System Sciences*, **7**(4), 448–461.

Harris N (2014). ***LazySorted**: A Partially and Lazily Sorted List Data Structure.* Python extension module version 0.1.1, URL https://pypi.python.org/pypi/lazysorted/.

Hoare CA (1962). "Quicksort." *The Computer Journal*, **5**(1), 10–16.

Hoare CAR (1961). "Find (Algorithm 65)." *Communications of the ACM*, **4**(7), 321–322.

Martínez C, Roura S (2001). "Optimal Sampling Strategies in Quicksort and Quickselect." *SIAM Journal on Computing*, **31**(3), 683–705.

Oliphant T, *et al.* (2009). ***NumPy**: A Python Library for Numerical Computations.* URL http://www.scipy.org/Numpy.

Peters T (2002). *Timsort.* Http://bugs.python.org/file4451/timsort.txt.

Sedgewick R (1978). "Implementing Quicksort Programs." *Communications of the ACM*, **21**(10), 847–857.

Seidel R, Aragon CR (1996). "Randomized Search Trees." *Algorithmica*, **16**(4–5), 464–497.

van Rossum G, *et al.* (2011). *Python Programming Language.* URL http://www.python.org/.

Affiliation:

Naftali Harris
Department of Statistics
390 Serra Mall
Stanford University
Stanford, CA 94305-4065, United States of America
E-mail: naftali@stanford.edu
URL: http://www.naftaliharris.com/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 65, Code Snippet 1 *Submitted:* 2014-02-19
May 2015 *Accepted:* 2014-08-20