2021 ASEE ANNUAL CONFERENCE
Virtual Meeting | July 26–29, 2021 | Pacific Daylight Time
Paper ID #34126

Linear Transform Sort

Mr. Soren Peter Henrichsen, Utah Valley University

Soren Henrichsen is a student at Utah Valley University. His interests include artificial intelligence, algorithms, robotics, machine learning, and statistics.

Dr. Reza Sanati-Mehrizy, Utah Valley University

Reza Sanati-Mehrizy is a professor of the Computer Science Department at Utah Valley University, Orem, Utah. He received his M.S. and Ph.D. in Computer Science from the University of Oklahoma, Norman, Oklahoma. His research focuses on diverse areas such as Database Design, Data Structures, Artificial Intelligence, Robotics, Computer-Aided Manufacturing, Data Mining, Data Warehousing, and Machine Learning.

Dr. Afsaneh Minaie, Utah Valley University

Afsaneh Minaie is a Professor and Chair of Engineering Department at Utah Valley University. She received her B.S., M.S., and Ph.D. all in Electrical Engineering from University of Oklahoma. Her research interests include gender issues in the academic sciences and engineering fields, Embedded Systems Design, Mobile Computing, Wireless Sensor Networks, Nanotechnology, Data Mining and Databases.

Linear Transform Sort

Abstract

This paper outlines the linear transform sort and its ideal and worst use cases. In general, a transform sort generates new keys in the range from 0 to n by a mathematical transformation, then uses those keys to sort the data set. We will prove the functionality of the linear transform sort and show that the complexity is $O(n)$ for flat data distributions, and $O(n^2)$ in the worst case. Linear transform sorting may be useful for flat data distributions but is mostly a proof of concept for future transform sorting algorithms, such as adaptive transformations or specific transformations for normally distributed data or other known data distributions.

Introduction

Many commonly used sorting algorithms are comparison based, determining elements' relative position by comparing them, and absolute position by their relative position. The limit on average complexity for the best comparison sorting algorithms on randomly generated unsorted lists is $O(nlog(n))$, though they may perform better or worse for particular inputs[1, 2].

Non-comparison sorts exist, notably radix sort and bucket sort [3]. These sorts determine absolute position directly from operations on the list elements, so the time taken depends on the size, length, and distribution of the input data set. While these sorts can be faster in theory, they can be slower in practice because for radix sort the complexity also scales with the key size [7], and bucket sort is influenced by the data distribution.

This paper will explore a new non-comparison sort. The linear transform sort uses a linear transformation to generate new keys between 0 and n and uses those keys to sort the data using a recursive bucket sort. The idea of using new computed keys is adapted from the Schwartzian Transform [4], where a sorting key is extracted from other extraneous data all at once instead of repeatedly during runtime.

This sort is a proof of concept that transformation can adapt the input of non-comparison sorts to increase speed of sorting. We will outline the linear transform sort algorithm, inductively prove its functionality, outline best and worst cases, present test data, and propose improvements for further research.

Algorithm

linear transform sort(S)

Input: a list of integers or doubles S

Output: a sorted list

- find largest key value (max(S)), smallest key value (min(S)), and number of entries (n)
- if max(S) = min(S) end
- else:
    - assign each element a new key based on a transformation of their value or old key. For this implementation targeted to evenly distributed data, we will use the transformation

$$H(x) = \left\lfloor \frac{n(x - min(S))}{(max(S) - min(S))} \right\rfloor$$

    - create n+1 buckets with indices 0 through n
    - assign each element to the bucket corresponding to their new key
    - recurse linear transform sort on each bucket

Note: While this implementation of transform sort acts very similar to bucket sort for a linear transform function, it will act differently with a transform function optimized for a different data set.

Proofs

Quicksort can be inductively proven to be able to sort any size list [6]. We have found experimentally that linear transform sort can sort lists up to at least n = 100,000. We will inductively prove that the linear transform sort algorithm above can sort an arbitrary list S with size n. First, we will outline the base cases and prove a supporting lemma, then we will prove the transformation assigns new keys only in the interval [0, n], then we will show that sorting holds across bucket boundaries, and finally we will inductively prove that the linear transform sort algorithm above can be used to sort any size list.

Base case: Where size n = 0 or n = 1, the list is sorted.

Alternate base case: Where size n < 1 and max(S) = min(S), all values are the same hence the list is sorted.

Lemma 3.1: if $\lfloor a \rfloor > \lfloor b \rfloor$, then $a > b$

By the definition of floor we have integers j, k so that $k + 1 > a \geq k$ and $j + 1 > b \geq j$

We can prove $k \geq j + 1$ by contradiction: if $j + 1 > k$, $\lfloor a \rfloor \geq \lfloor b \rfloor$ which is a contradiction.

Hence $a \geq k \geq j + 1 > b$ Hence by the transitive property of inequalities, $a > b$

a) Transform

We will prove that the key transform function $H(x) = \left\lfloor \frac{n(x-min(S))}{(max(S)-min(S))} \right\rfloor$ generates new key values in the range [0, n].

Consider three elements of S: max(S), min(S), and arbitrary element s.

max(S) and min(S) are the largest and smallest elements of S, so min(S) ≤ s ≤ max(S)

$$H(min(S)) = \left\lfloor \frac{n(min(S)-min(S))}{(max(S)-min(S))} \right\rfloor$$

$$H(min(S)) = \left\lfloor \frac{n(0)}{(max(S)-min(S))} \right\rfloor = 0$$

$$H(x) = \left\lfloor \frac{n(max(S)-min(S))}{(max(S)-min(S))} \right\rfloor$$

We will prove that $H(min(S)) \leq H(s) \leq H(max(S))$. We work by contradiction in parts:

1) Proof: $H(min(S)) > H(s)$ → contradiction

   It is shown above that $H(min(S)) = 0$

   Assume $H(min(S)) > H(s)$

   Hence $0 > \left\lfloor \frac{n(s-min(S))}{(max(S)-min(S))} \right\rfloor$

   For this to be true, one or more of the following must be true:

$$\begin{cases} n < 0 \\ s < min(S) \\ max(S) < min(S) \end{cases}$$

   Any of these would create a contradiction.

   Hence $H(s) \geq H(min(S))$ for all s.

2) Proof. $H(s) > H(max(S))$ → contradiction

   It is shown above that $H(max(S)) = \lfloor n \rfloor = n$

   Assume $H(s) > H(max(S))$

$$\left\lfloor \frac{n(s-min(S))}{(max(S)-min(S))} \right\rfloor > \lfloor n \rfloor$$

Using Lemma 3.1 we then have $\frac{n(s-min(S))}{(max(S)-min(S))} > n$

$$n(s - min(S)) > n(max(S) - min(S))$$
$$n * s - n * min(S) > n * max(S) - n * min(S)$$
$$n * s > n * max(S)$$
$$s > max(S)$$

Any element being strictly greater than the maximal element is a contradiction.

Hence $H(max(S)) \geq H(s)$

Therefore:

- For any element in S, it holds that $0 = H(min(S)) \leq H(s) \leq H(max(S)) = n$ or in other words, no element can have a key value outside [0, n].
- Since the floor function yields integers, we have n + 1 possible key values. We can then assign the elements of S to one of n + 1 buckets corresponding to the n + 1 possible key values.

b) Sorting holds across bucket boundaries

We will prove that sorting holds across bucket boundaries, or symbolically,

$$\forall\, a, b \in S\ ,\ H(a) > H(b) \Rightarrow a > b$$

We work directly.

Assume $H(a) > H(b)$

$$\left\lfloor \frac{n(a-min(S))}{(max(S)-min(S))} \right\rfloor > \left\lfloor \frac{n(b-min(S))}{(max(S)-min(S))} \right\rfloor$$

Using Lemma 3.1, we then have:

$$\frac{n(a-min(S))}{(max(S)-min(S))} > \frac{n(b-min(S))}{(max(S)-min(S))}$$
$$a - min(S) > b - min(S)$$
$$a > b$$

Hence any element in a higher-indexed bucket is greater than any element in a lower-indexed bucket.

c) Using the inductive assumption

Using the above proofs and the inductive assumption that arrays with a number of elements in the range [0, n − 1] can be sorted with linear transform sort, we can prove that linear transform sort works for an array S with n elements.

We have shown that the transform step generates new keys to place the elements of S into n+1 buckets.

From equations (1) and (2) in 3.2.1 above, we see that at least 1element is in bucket 0 and one element is in bucket n.

We also know from 3.2.1 that all elements of S are placed in one of the buckets in the range [0, n].

Hence buckets 0 and n could have any number of elements in the range [0, n − 1] and any other bucket could have any number of elements in the range [0, n − 2].

By the inductive assumption these buckets can be sorted using linear transform sort.

Since each bucket is sorted, and we know from 3.2.2 that sorting holds across bucket boundaries, we have sorted S.

Hence assuming the algorithm works for any data set with size in the range [0, n − 1], the algorithm works for any data set of size n.

Complexity

Best case

The best case for this sort is a discrete uniform distribution, where the data points are evenly distributed across the entire range of the data set. For example:

(Figure 1: Discrete Uniform Distribution)

In this case, the algorithm will:

1. find n, if $n \leq 1$ end
2. find max(S) and min(S), if max(S) = min(S) end
3. generate a new key for each element
4. create n + 1 buckets corresponding to the keys, and put each element in its corresponding bucket (in the best case, each element will have its own bucket, leaving one bucket empty)

5. recurse on each bucket (each recursion will end on step 1 because in each bucket there are only zero or one elements)

The time complexity will scale with n, as each data point is evaluated a constant number of times. The space complexity will also scale with n.

Worst case

The worst case for this sort is when at each step, all data points except one will be in the same bucket, so that with each recursion only one data point is removed. For example:

(Figure 2: Sample worst case distribution)

In theory, anything that diverges faster than F(x) = x! will use the worst case sorting time. Anything that diverges faster than F(x) = x and slower than F(x) = x! will slow the sort, giving a time complexity between n and n^2.

In the worst case, the algorithm will:

1. find n, if $n \leq 1$ end
2. find max(S) and min(S), if max(S) = min(S) end
3. generate a key for each element with the transformation
4. create n + 1 buckets corresponding to the keys, and put each element in its corresponding bucket (There will be one element in bucket n and the rest of the element in bucket 0, or vice versa)
5. recurse on each bucket (each recursion will result in one data point sorted and a recursion with $n - 1$ elements. The algorithm will terminate after n recursions)

In the worst case, the time complexity will scale with n^2, as each data point is operated on n times for each recursion, and each recursion only sorts one data point.

Normal use

In normal use cases, transform sorting is less time complex when the data distribution matches the transform function. It gets more time complex when the data distribution does not match the transform function, so each recursion sorts out only a few data points.

For linear transform sort:

- Duplicate data points take a constant number of operations to sort (k operations per data point)

- Data points far from the max or min can take up to one full recursion to sort - (up to k ∗ n operations per data point)
- Counterintuitively, in some cases adding more intermediate or duplicate data points can make the sort faster. Consider [1, 3, 12, 60] (recursion depth 4) and [1, 3, 5, 12, 25, 60] (recursion depth 3).

Running tests on several random and non-random data sets to compare timsort, quicksort, and linear transform sort showed that the runtime for quicksort and linear transform sort was almost the same at n = 10000 across several different data sets. Timsort is still around a hundred times faster. Further work on refining this implementation and new transform algorithms is needed before transform sorts have any practical application.

Implementation

```
import math
# Global variable to avoid flattening complexity
sorted_list = []
# supporting functions

def getmax_val(lyst):
    max_val = -math.inf
    for i in range(len(lyst)):
        if lyst[i] > max_val:
            max_val = lyst[i]
    return max_val

def getmin_val(lyst):
    min_val = math.inf
    for i in range(len(lyst)):
        if lyst[i] < min_val:
            min_val = lyst[i]
    return min_val


# main functions for linear transform sort

def linear_transform(entry, max_val, min_val, num_entries):
    return (math.floor(num_entries*(entry-min_val)/(max_val-min_val)))

def linear_transform_sort_helper(sublyst):
```

```
    global sorted_list
    num_entries = len(sublyst)
    if num_entries == 0:
        return # this bin has no contents
    if num_entries == 1:
        sorted_list.append(sublyst[0])
        return # sorting complete on this bin
    min_val = getmin_val(sublyst)
    max_val = getmax_val(sublyst)
    if min_val == max_val:
        for value in sublyst:
            sorted_list.append(value)
        return # sorting complete on this bin
    else: # sorting is necessary
        bins = []
        for i in range(num_entries + 1): # create n+1 bins
            bins.append([])
        for i in range(num_entries): # place entries into bins
            cast_location = linear_transform(
            sublyst[i], max_val, min_val, num_entries
            )
            bins[cast_location].append(sublyst[i])
        for i in range(num_entries + 1): # recurse on each bin
            linear_transform_sort_helper(bins[i])

def lt_sort(lyst):
    global sorted_list
    sorted_list = []
    linear_transform_sort_helper(lyst)
    return sorted_list
```

Conclusion and Results

We compared linear_transform_sort to timsort [5] and quicksort [6] over several different data set types and sizes. In general, linear transform sort performed better than quicksort at n < 10000 and for duplicate, linear, and squared data. Timsort performed much better than linear_transform_sort in every case. Included are the results over 100 trials at n = 10,000.

Evaluating sorts with a random integer data set:
(random.randint(0, 10^6))

Average time for linear_transform_sort:	0.02537
Average time for quicksort:	0.02538
Average time for timsort:	0.00025

Evaluating sorts with a random noninteger data set:
(random.uniform(0, 10^6))

Average time for linear_transform_sort:	0.02526
Average time for quicksort:	0.02623
Average time for timsort:	0.00026

Evaluating sorts with a data set of perfect squares:
[1, 4, 9, 16, 25... ...100,000,000]

Average time for linear_transform_sort:	0.01235
Average time for quicksort:	0.02551
Average time for timsort:	0.00015

Evaluating sorts with a duplicate data set:
[42, 42, 42, 42...]

Average time for linear_transform_sort:	0.00135
Average time for quicksort:	0.02573
Average time for timsort:	4.648e-06

Further research

- Transform functions can be derived for data fitting the normal distribution and other common distributions
- Transform sorts can be compared with other non-comparison sorts for real-world applications
- Transform sorting could be proven to be stable
- Dynamic transformation algorithms could be implemented for use when the data distribution is irregular or not known (e.g. strings of varying lengths)

- Other sorting algorithms such as counting sort or radix sort could be studied for post-transformation use.

References

1. Background on sorting algorithms: https://en.wikipedia.org/wiki/Sorting_algorithm, accessed on March 2021.
2. Further background on sorting algorithms: https://www.cs.ubc.ca/~tsiknis/cics216/notes/Unit7-SortingAlgorithms.pdf, accessed on January 2021.
3. Bucket sort: https://www.geeksforgeeks.org/bucket-sort-2/ , accessed on December 2020.
4. Schwartzian transform: http://www.stonehenge.com/merlyn/UnixReview/col64.html, accessed on January 2021.
5. Python list.sort() or timsort, https://docs.python.org/3/howto/sorting.html, accessed on January 2021.
6. Quicksort and proof of functionality https://www.cs.mcgill.ca/~dprecup/courses/IntroCS/Lectures/comp250-lecture17.pdf , accessed on February 2021.
7. Sorting Algorithms, http://syllabus.cs.manchester.ac.uk/ugt/2019/COMP26120/SortingTool/radix_sort_info.html, accessed on January 2021.