Sorting, Information, and Recursion

Charles Davi

August 20, 2021

Abstract

Below we introduce two theorems that show a set of real numbers is sorted if and only if (1) the distance between adjacent pairs in the resultant ordering is minimized, and (2) if and only if the amount of information required to encode the resultant ordering, when expressed as a particular class of recurrence relations, is minimized. As such, these two theorems demonstrate an equivalence between sorting and minimizing information.

1 Introduction

In Section 2, we show that sorting a set of numbers is equivalent to minimizing the distance between adjacent entries in the resultant sequence of numbers. In Section 3, we show that this is in turn equivalent to minimizing the amount of information used to encode the set as a particular class of recurrence relations that is related to the derivative of a function. The theorem in Section 2 also implies a more general notion of sorting that would apply to all mathematical objects for which there is a measure of distance $F : S \times S \to \mathbb{R}$, where S is the set of objects in question. That is, if it is possible to compare every pair of objects in a set, and map the difference between each pair to the real line, then we can define a partial order on the set in question, as a directed graph, that is the analog of sorting the set, where node a points to node b if $b - a \geq 0$, and the distance $|b - a|$ is minimized over the set. It is this latter criteria that differentiates this method from an arbitrary partial order on a set.

2 Sorting and Distance

Theorem 2.1. *A sequence of real numbers $(a_1, a_2, \ldots, a_k)$ is sorted if and only if the distance $|a_i - a_{i+1}|$ is minimized for all i.*

Proof. Assume the sequence is sorted in ascending order, and that the theorem is false. Note that the proof for the case of a descending sequence is analogous. Because the sequence is sorted, the distance $|a_1 - a_2|$, must be minimized for a_1, since by definition, all other elements are greater than a_2. By analogy, the distance $|a_{k-1} - a_k|$ must also be minimized for a_k. Therefore, it follows that there must be some a_m, for which either -

$$(1)\ |a_i - a_m| < |a_i - a_{i+1}|,$$

or,

$$(2)\ |a_{i+1} - a_m| < |a_i - a_{i+1}|.$$

Said in words, because we have already eliminated the first and last entries in the sequence as entries that could cause the theorem to be false, in order for the theorem to be false, there must be some pair of adjacent entries, and some third entry, with a distance to one of those two entries that is less than the distance between the pair itself. Because assuming that $a_m < a_i$, or that $a_m > a_{i+1}$, simply changes the indexes, it must be the case that $a_i < a_m < a_{i+1}$, which in

turn contradicts the assumption that the sequence is sorted, which completes the first half of the proof.

Now we will prove by induction that if $|a_i - a_{i+1}|$ is minimized for all i, then the sequence is sorted:

Assume we start with a single entry, a_i. Then, we want to insert some new entry, a_j, in order to generate a sequence in ascending order, since a proof for descending order is analogous. Because there are only two entries, a_i is the nearest neighbor of the new entry a_j (i.e., the distance between a_i and a_j is minimized). Because the criteria of the theorem is only that adjacent entries have minimized distances, we can place the inserted entry on either side of an existing entry. So we choose that if $a_j > a_i$, then we insert a_j to the right of a_i, and otherwise, we insert a_j to the left of a_i.[1]

Now assume the theorem holds for some number of insertions $k \geq 2$. It follows that this will imply a sorted sequence $(a_1, a_2, \ldots, a_k)$. If there is more than one nearest neighbor for a given insert (i.e., two entries of equal distance from the insert entry) that are both equal in value, then we find either the leftmost such entry, or the rightmost such entry, depending upon whether the inserted entry is less than its nearest neighbor, or greater than its nearest neighbor, respectively. If the insert is equidistant between two unequal adjacent entries, then we insert it between them.

Now assume we insert entry a_m, which will be insertion number $k + 1$. Further, assume that following the process above causes the resultant sequence of $k + 1$ entries to be unsorted, solely as a result of this insertion (note we assumed that the sequence is sorted beforehand), and further, assume that the indexes are such that $(a_1, a_2, \ldots, a_k)$ is the correct set of indexes for the sorted sequence prior to the insertion of a_m.

Now assume the process above places a_m at the front of the sequence. Because the sequence is unsorted, it must be that $a_m > a_1$, but this is impossible, according to the process above, which would place it to the right of a_1. Now assume the process above places a_m at the end of the sequence. Because the sequence is unsorted, it must be that $a_m < a_k$, but again, the process above dictates an insertion to the left.

Therefore, since by assumption the sequence is unsorted, it must be the case that a_m is inserted between two entries in the sequence a_i, a_{i+1}. By the process

[1] Note that if the application of such a rule is not done consistently, then the distance between adjacent entries will not be minimized. For example, insert the vector $(2, 3, 1)$ in that order, minimizing distance, but alternating the rule of insertion, and we could, e.g., end up with $(2, 1, 3)$, which does not minimize the distances between adjacent entries. In order to simplify the proof, we assume the application of certain consistent rules, since all alternatives would be eliminated as violating the requirement that the distances between adjacent entries must be minimized.

described above, it must also be the case that the distance to at least one of them is minimized, so assume that a_i is the nearest neighbor of a_m.

Again, since by assumption, the sequence is out of order, it must be the case that either -

$$(1)\ a_i > a_m,$$

or

$$(2)\ a_m > a_{i+1}.$$

In case (1), again, the process above dictates insertion to the left of a_i, which eliminates this case as a possibility. In case (2), it must be that a_{i+1} is the nearest neighbor of a_m, which leads to a contradiction. The cases where a_{i+1} is the nearest neighbor of a_m are analogous, which completes the proof. □

Conjecture 2.2. *There is no deterministic sorting algorithm that can provably generate a total order on a multi-set of numbers (i.e., allowing for duplicates) in less than $O(N)$ steps;*[2]

Conjecture 2.3. *There is no sorting algorithm that can reliably generate a total order on a multi-set of numbers (i.e., allowing for duplicates) in less than $O(\log(N))$ steps,*[3]

in each case, where N is the cardinality of the set.

[2]Note that the algorithm presented in the second half of Theorem 2.1 would have a linear runtime on a machine capable of performing all independent calculations in parallel. See the code attached.

[3]Attached is code for two sorting algorithms, one deterministic, with an $O(N)$ runtime, that is based upon the algorithm presented in the proof for Theorem 2.1, and another with a $O(\log(N))$ runtime, in each case, when run on a machine capable of performing all independent calculations in parallel. However, the $O(N)$ algorithm would generate only a partial order on a set with duplicate entries (i.e., a multi-set), though the code includes annotations that explain how the exact algorithm presented in the proof for Theorem 2.1 could be implemented in $O(N)$ runtime. The $O(\log(N))$ algorithm generates a total order, regardless of whether or not there are duplicates. However, the $O(\log(N))$ algorithm does have some small probability of generating error, because it makes use of random noise to handle duplicate entries. The error would arise only when two duplicate entries are given exactly the same amount of noise.

All of the attached code can actually be run in Octave and Matlab, and in each case includes a sample dataset.

3 Sorting, Information, and Recursion

Note that if for example, we're given a sequence of numbers $(1, 2, 3)$, then we can express that sequence as another sequence $(1, 1, 1)$, where we add each entry in the latter vector in order, i.e., $(((1) + 1) + 1)$.[4] If we're instead given a set of numbers, and first sort the set of numbers in question, we will minimize the amount of information required to encode the set as a recurrence relation of this type, which follows trivially from Corollary 3.1 below.

We can then generalize this to vectors, expressing the sequence $(v_1, \ldots, v_k)$ as another sequence $(\Delta_1, \ldots, \Delta_k)$, where $\Delta_1 = v_1$, and $\Delta_i = v_i - v_{i-1}$, for all $i > 1$. Measuring the amount of information required to represent this sequence of difference vectors however requires a notion of the amount of information required to represent an individual vector.

Definition: *the logarithm of a vector*, $\log(v_1) = v_2$, is such that,

$$2^{||v_2||} = ||v_1||,$$

and,

$$\frac{v_1}{||v_1||} = \frac{v_2}{||v_2||}.$$

That is, raising 2 to the power of the norm of v_2 produces the norm of v_1, and both vectors point in the same direction. Note that because $||v_2|| = \log(||v_1||)$, and $v_2 = \log(v_1)$, it follows that $||v_2|| = ||\log(v_1)||$.

We have then,

$$\overrightarrow{H} = \sum_{\forall i} \log(\Delta_i), \tag{1}$$

where $||\overrightarrow{H}||$ is always a real number.[5]

[4] Note that if we're given a sequence of numbers in the range of a function $(f(1), f(2), f(3), \ldots)$, then taking the difference between adjacent entries gives the slope of the line connecting adjacent points in the range of the function. This in turn allows us to reconstruct the range of the function given these differences.

[5] Note that a real number sequence that contains a decreasing pair of adjacent entries will produce a complex valued output to the logarithm function. This creates an asymmetry between strictly increasing sequences, and all others, since the logarithm of 0 is not defined, absent additional assumptions. Interestingly, accumulation over a set of positive numbers is as a general matter strictly increasing. In particular, the quantity of time that has elapsed, is therefore always strictly increasing, if correctly observed. Finally, note this would not be an

If each v_i is a vector, then $\overrightarrow{H}$ is a vector analog of the measure of the amount of information required to represent a sequence as a recurrence relation of the type described above, which I suspect will have applications to physics, in particular, thermodynamics.[6] However, in the case of either a sequence of vectors or real numbers $(v_1, \ldots, v_k)$, the actual number of bits required to encode the sequence as a recurrence relation is instead given by,

$$\bar{H} = \sum_{\forall i} \log(||\Delta_i||).^{7} \tag{2}$$

Corollary 3.1. *A sequence of real numbers $(a_1, a_2, \ldots, a_k)$ is sorted if and only if $\bar{H}$ is minimized.*

Proof. Assume that the corollary is false, and that the sequence is unsorted, but $\bar{H}$ is minimized. Now sort the sequence, and recalculate $\bar{H}$. By Theorem 1.1 above, it must be the case that each argument to the logarithm function is minimized, and therefore the sum over each logarithm is minimized, which in turn implies that $\bar{H}$ is minimized, which contradicts our assumption that $\bar{H}$ is minimized when the sequence is unsorted.

Now assume instead that the corollary is false, and that the sequence is sorted, but $\bar{H}$ is not minimized. By Theorem 1.1, it must be the case that each argument to the logarithm function is minimized, and therefore the sum over each logarithm is minimized, which in turn implies that $\bar{H}$ is minimized, which contradicts our assumption that $\bar{H}$ is not minimized, which completes the proof. □

Let $s = \{v_1, \ldots, v_k\}$ be a set of vectors, such that $v_i \in \mathbb{R}^n$, for all i.

Theorem 3.2. *$||\overrightarrow{H}||$ is maximized when $\Delta_i = \Delta_j$ for all $\{i, j\}$.*

issue for sequences of vectors, given the definition of $\log(v)$ above, which will always produce real number vectors, given a real number vector as an argument.

[6] For example, if we begin with a set of observed velocity vectors for a system, and they are all equal in magnitude and direction, implying rectilinear velocity, then sorting them and calculating $\overrightarrow{H}$ will produce the zero vector. If however, we have a system with a complicated motion, then we will find a non-zero vector answer. This is superficially consistent with thermodynamic entropy, that increases as a function of the ostensible randomness of a system.

[7] Note that once the magnitude and direction of a vector are specified, the vector in question is specified, without any other information being necessary. We can specify the magnitude of a vector v using $\log(||v||)$ bits, and can specify the direction in any plane as a portion of the length of the circumference of the applicable unit circle, which requires at most $\log(2\pi)$ bits. As such, to specify a vector, we first specify the unit vector that points in the correct direction, by specifying the angle of the vector in each plane of some basis of the space in question, which will in turn define a unit vector in the correct direction, and then extend that vector using the stated norm, which will together identify the vector. Because the norm is unbounded, whereas the amount of information required to specify direction is bounded for any given dimension, we ignore the amount of information required to specify direction.

Proof. We begin by proving that,

$$\bar{H} = \sum_{\forall i} \log(||\Delta_i||),$$

is maximized when the norms of all Δ_i are equal.

So assume this is not the case, and that as such, there is some $||\Delta_i|| > ||\Delta_j||$, and so we can then restate $\bar{H}$ as,

$$\bar{H} = \sum_{\forall k \neq \{i,j\}} \Big(\log(||\Delta_k||) \Big) + \log(||\Delta_i||) + \log(||\Delta_j||).$$

Now let $L = ||\Delta_i|| + ||\Delta_j||$, and let $F(x) = \log(L - x) + \log(x)$. Note that if $x = ||\Delta_j||$, then $F(x) = \log(||\Delta_i||) + \log(||\Delta_j||)$. Let us maximize $F(x)$, which will in turn maximize $\bar{H}$, by taking the first derivative of F, with respect to x, which yields,

$$F' = \frac{1}{x} - \frac{1}{L - x}.$$

Setting F' to zero, we find $x = \frac{L}{2}$, which in turn implies that $F(x)$ has an extremal point when the arguments to the two logarithm functions are equal to each other. The second derivative is negative for any positive value of L, and because L is the sum of the norm of two vectors, L is always positive, which implies that F is maximized when $||\Delta_i|| = ||\Delta_j||$. Since we assumed that $\bar{H}$ is maximized for $||\Delta_i|| > ||\Delta_j||$, we have a contradiction. And because this argument applies to any pair of vectors, it must be the case that all vectors have equal magnitudes.

For any set of vectors, the norm of the sum is maximized when all vectors point in the same direction, and taking the logarithm does not change the direction of a vector. Therefore, if $||\overrightarrow{H}||$ is maximized, it must be the case that all Δ_i point in the same direction. Note that if all such vectors point in the same direction, then,

$$||\overrightarrow{H}|| = \sum_{\forall i} ||\log(\Delta_i)|| = \bar{H},$$

which completes the proof. □

```
%================================================================================
%================================================================================
%VECTORIZED SORTING ALGORITHM (LINEAR RUNTIME, UNIQUE ENTRIES ONLY)
%================================================================================
%================================================================================
%COPYRIGHT CHARLES DAVI 2021

%================================================================================
%GENERATES DATASET
%================================================================================
num_items = 5; %the number of random numbers
max_num = 50; %the maximum random number, the min being 1
dataset = randi(50,[num_items 1])

dataset = unique(dataset); %ensures only unique entries
num_items = size(dataset,1);

%scrambles the dataset-------------------------------------------------------
scramble = randperm(num_items);
dataset = dataset(scramble)

%================================================================================
%SORTS DATASET
%================================================================================

%Note the min operater can be implemented in log(N) steps as follows:
%take the difference between corresponding entries in List(1:N-1) - List(2:N).
%If, e.g., (entry 1 - entry 2) is greater than zero, then entry 1 cannot be the min.
%So remove entry 1, and all such other entries, and repeat, until you have either
%one element remaining, or a set of equal elements.

%The loop condition is in either case the list not changing size,
%which has a constant runtime - just check the stack memory for the list.
%Therefore, the loop for the min operator runs log(N-K) times, where N is the
%number of entries in the list, and K is the number of duplicates.
%Each step of the loop has a constant runtime.

%Finally, note that if you wanted the leftmost, or rightmost index among the minima,
%using this approach, you would simply take the first or last entry in memory,
%which has a constant runtime. As a result, you could implement the algorithm in
%the second half of Theorem 2.1 in linear time.

[IGNORE min_index] = min(dataset); %the nearest neighbor of the min item is NULL

%the following steps have a constant runtime in parallel
num_rows = size(dataset,1);
temp_matrix = repmat(dataset', [1 1 num_rows]);
ref_dataset = shiftdim(temp_matrix,2);
diff_matrix = dataset.-ref_dataset;

%this step has a linear runtime
zero_indeces = 1:num_rows+1:num_rows*num_rows;

%the remaining steps other than the min operator have a constant runtime
diff_matrix(zero_indeces) = Inf; %sets the zero entries to infinity

positive_entries = find(diff_matrix > 0); %these cannot be the nearest neighbors
diff_matrix(positive_entries) = Inf;
diff_matrix = abs(diff_matrix);
```

```
[a b] = min(diff_matrix)

nearest_neighbors = reshape(b,[num_rows 1])

%=========================================================================
%GENERATES DI-GRAPH EDGES
%=========================================================================

edge_matrix = [dataset dataset(nearest_neighbors)];
edge_matrix(min_index,2) = Inf %ignores the nearest neighbor of the min item
```

```
%================================================================================
%================================================================================
%VECTORIZED SORTING ALGORITHM (LOGARITHMIC RUNTIME)
%================================================================================
%================================================================================
%COPYRIGHT CHARLES DAVI 2021

%================================================================================
%GENERATES DATASET
%================================================================================
num_items = 5;
max_num = 50;
dataset = randi(max_num,[num_items 1]);

%adds noise which allows us to remove entries that match to themselves
%this in turn allows for duplicate entries in log(N) time
temp_dataset = dataset;
dataset = dataset + rand(num_items,1);
%================================================================================
%SORTS DATASET
%================================================================================

%Note the min operator can be implemented in log(N) steps as follows:
%take the difference between corresponding entries in List(1:N-1) - List(2:N).
%If, e.g., (entry 1 - entry 2) is greater than zero, then entry 1 cannot be the min.
%So remove entry 1, and all such other entries, and repeat, until you have either
%one element remaining, or a set of equal elements.

%The loop condition is in either case the list not changing size,
%which has a constant runtime - just check the stack memory for the list.
%Therefore, the loop for the min operator runs log(N-K) times, where N is the
%number of entries in the list, and K is the number of duplicates.
%Each step of the loop has a constant runtime.

%Finally, note that if you wanted the leftmost, or rightmost index among the minima,
%using this approach, you would simply take the first or last entry in memory,
%which has a constant runtime. As a result, you could implement the algorithm in
%the second half of Theorem 2.1 in linear time.

[IGNORE min_index] = min(dataset); %the nearest neighbor of the min item is NULL

%the following steps have a constant runtime in parallel
num_rows = size(dataset,1);
temp_matrix = repmat(dataset', [1 1 num_rows]);
ref_dataset = shiftdim(temp_matrix,2);
diff_matrix = dataset.-ref_dataset;

zero_indeces = find(diff_matrix == 0);
diff_matrix(zero_indeces) = Inf; %sets the zero entries to infinity

positive_entries = find(diff_matrix > 0); %these cannot be the nearest neighbors
diff_matrix(positive_entries) = Inf;
diff_matrix = abs(diff_matrix);

%as explained above, this has a log runtime
[a b] = min(diff_matrix)

%this has a constant runtime
nearest_neighbors = reshape(b,[num_rows 1])
```

```
%==========================================================================
%GENERATES DI-GRAPH EDGES
%==========================================================================
dataset = temp_dataset; %resets the dataset to the original
edge_matrix = [dataset dataset(nearest_neighbors)];
edge_matrix(min_index,2) = Inf %effectively NULL
```