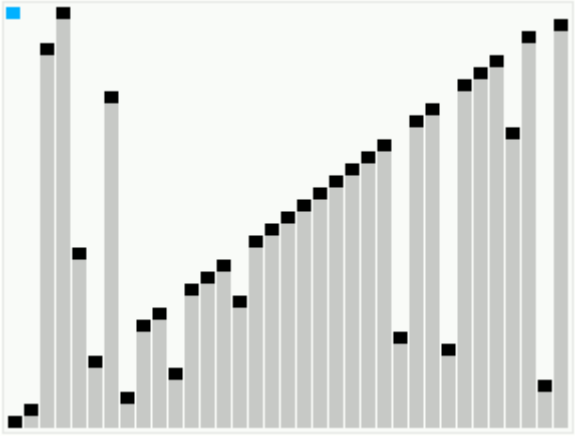
Smoothsort

In computer science, **smoothsort** is a comparison-based sorting algorithm. A variant of heapsort, it was invented and published by Edsger Dijkstra in 1981.[1] Like heapsort, smoothsort is an in-place algorithm with an upper bound of O(n log n),[2] but it is not a stable sort.[3] The advantage of smoothsort is that it comes closer to O(n) time if the input is already sorted to some degree, whereas heapsort averages O(n log n) regardless of the initial sorted state.

Smoothsort



Smoothsort operating on an array which is mostly in order. The bars across the top show the tree structure.

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$
Best-case performance	$O(n)$
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

Contents

Overview

Like heapsort, smoothsort first transforms the input array into an implicit heap data structure, then produces the sorted array by repeatedly extracting the largest remaining element, whose location is determined by the heap structure, and the restoring the structure on the remaining elements. Whereas heapsort uses an implicit tree structure on the array in which a parent node always come before its descendants, so that the initial element is the root of the tree, smoothsort uses an implicit tree structure where a parent node always comes after its descendants. This has some important consequences, such as the fact that not all initial portions of the array correspond to a complete tree; however, it can help to avoid unnecessary displacements as in heapsort, where every element has to pass through the initial position (the root) just before getting swapped to its final position. Indeed, the algorithm is organized so that at the point where an element gets extracted from the heap structure, it is already at the rightmost position among the remaining elements, which is its proper place in the sorted array, so no swap is needed to bring it there.

The tree structure defined on array positions is fixed and independent of the array contents. It can be extended to all natural numbers, and doing so there is no root, since every position gets to be a child of a parent further on; however some positions are leaves in an absolute sense. This contrasts with the tree structure used for heapsort, where the initial position is the global root, but there are no absolute leaves, since every position gets two children when sufficiently many positions are added. In the smoothsort structure, every position *i* is the root of a unique subtree, whose nodes form an interval that ends at *i*. An initial interval of positions, and in particular the set of all positions in a given array, might be such an interval corresponding to a subtree, but in general decomposes as a union of a number of successive such subtree intervals, which Dijkstra calls "stretches". Any subtree rooted at a position whose parent lies beyond the initial interval considered gives a stretch in the decomposition of that interval, which decomposition is therefore unique. When a new position following the sequence of stretches is added, one of two things can be the case: either the position is a leaf and adds a stretch of length 1 to the decomposition, or it combines with the last two stretches, becoming the parent of their respective roots, thus replacing the two stretches by a new stretch containing their union plus the new (root) position.

Different rules could be used to determine for each position which of the two possibilities applies. For instance, one could stipulate that the last two stretches are combined if and only if they have equal size, in which case all subtrees would be perfect binary trees. Dijkstra's formulation of smoothsort uses a different rule, in which sibling subtrees never have equal sizes (except when both are 1), and the sizes of all subtrees are among a set of values that he calls Leonardo numbers (they are closely related to Fibonacci numbers). The general outline of the smoothsort procedure can be defined independently of the rule used to define the implicit tree structure, though the details of its steps depend on that rule. Dijkstra points out[1] that it would have been possible to use perfect binary trees (of size 2^k-1); this would lead to the same asymptotic efficiency,[2] but a constant factor in efficiency would be lost due to the on average greater number of stretches that a range of positions breaks up into.

The rule Dijkstra uses is that the last two stretches are combined if and only if their sizes are *successive* Leonardo numbers L(i+1), L(i) (in decreasing order), which numbers are recursively defined as:

- L(0) = L(1) = 1
- L(k+2) = L(k+1) + L(k) + 1

As a consequence, the size of any subtree is a Leonardo number. The sequence of sizes stretches decomposing the first *n* positions, for any *n*, can be found in a greedy manner: the first size is the largest Leonardo number not exceeding *n*, and the remainder (if any) is decomposed recursively. The sizes of stretches are decreasing, strictly so except possibly for two final sizes 1, and avoiding successive Leonardo numbers except possibly for the final two sizes.

In the initial phase of sorting, an increasingly large initial part of the array is reorganized so that the subtree for each of its stretches is a max-heap: the entry at any non-leaf position is at least as large as the entries at the positions that are its children. In addition, the subsequence of root (rightmost) entries of the stretches is kept increasing; this ensures in particular that the final root holds a maximal entry among the interval considered so far. In the second phase one shrinks the interval of interest back to smaller and smaller initial parts of the array, maintaining the same relations as before as the decomposition of the part considered into stretches evolves. At the point where a position disappears from the part under consideration due to shrinking, its entry is maximal among those in the part; as this is true at each shrinking step, those entries are left to sit in their proper position in the sorted result.

The rearrangements necessary to ensure the required relations at all times, and thereby realize the eventual sorting of the array, are as follows. During the first "growing" phase, the heap relation must be enforced whenever two final stretches are combined with a new root to a single stretch; then (also in the case where a singleton stretch was added) the new root has to be compared to any previous roots of stretches to ensure that the subsequence remains increasing, basically by performing an insertion step on the subsequence by repeated swaps with predecessors; and finally, if the new root was moved backwards in this insertion, the max-heap property must be restored for the subtree where it ended (because the entry at the root position has decreased). During the "shrinking" phase, the only worry is keeping the subsequence of roots of stretches increasing as positions are added to and removed from the subsequence. This automatic whenever a stretch of length 1 is removed; however when a longer stretch breaks up and leaves two smaller stretches in its place, the values at their roots are unrelated to the entries at preceding roots, so preserving increase of the subsequence requires two insertion steps, each one (as before) followed by restoring the max-heap property in the stretch where the new entry ends up, unless it remained in its final position.

To this general plan, a few refinements are applied to arrive at Dijkstra's sorting procedure. Notably, in the growing phase, ensuring the max-heap property at the root of a newly formed stretch can be combined with ordering it relative to the root of the preceding stretch: if the latter is larger than the new root, then either it also dominates both children of the new root, in which case swapping the roots also repairs the max-heap property, or else swapping the new root with its larger child also repairs the relation with the previous stretch root (but the repair of the max-heap property still needs to propagate down the heap). All in all, at most one series of swaps needs to propagate through the structure during each growing step; by contrast, a shrinking step may as mentioned involve either no such series or two of them.

Implementation of the implicit tree structures could be easily done by computing once and for all a list of Leonardo numbers and tables for parent and child relations (which do not depend at all on the values to be sorted). But in order to make this qualify as an in-place sorting algorithm, Dijkstra maintains in a slick way a fixed number of integer variables in such a way that at each point in the computation gives access to the relevant information, without requiring any tables. While doing so does not affect complexity, it does take up a large part of the code for the algorithm, and makes the latter more difficult to understand.

Operations

While the two phases of the sorting procedure are opposite to each other as far as the evolution of the sequence-of-heaps structure is concerned, the operations they need to perform to ensure the invariants of the structure have much in common. All these operations are variations of the "sift" operation in a binary max-heap, which restores the heap invariant when it is possibly violated only at the root node. Restoring the increase among the roots of the stretches is obtained by considering the root of each stretch except the first to have as an additional child (Dijkstra uses the term "stepson") the root of the stretch to its left.

Growing the heap region by incorporating an element to the right

When an additional element is considered for incorporation into the sequence of stretches (list of disjoint heap structures) it either forms a new stretch of its own added to the sequence, or it combines the two rightmost stretches by becoming parent of both their roots and forming a new stretch that replaces the two in the sequence. Which of the two happens depends only on the sizes of the stretches currently

present (and ultimately only on the index of the element added); Dijkstra stipulated that stretches are combined if and only if their sizes are L(k+1) and L(k) for some k, i.e., consecutive Leonardo numbers; the new stretch will have size L(k+2).

After a new element *x* is incorporated into the region of heap structures, as the root of a final stretch that is either a new singleton or obtained by combining two previous stretches, the following procedure called "trinkle" will restore the required relations. The element *x* is repeatedly swapped with the root of the previous stretches, as long as this condition holds: there is a stretch immediately to the left of the stretch of *x*, whose root is greater than *x*, and *also* greater than both of the children of *x*, if *x* has them. Note that at this time no ordering relation between *x* and its children has yet been established, and if the largest of its children exceeds *x*, then it is that child that will become the root once the max-heap property is ensured. After thus moving *x* into the appropriate stretch, the heap property of the tree for that stretch is established by "sifting down" the new element to its correct position. This is the same operation as used in heapsort, adapted to the different implicit tree structure used here. It proceeds as follows: while *x* is not at a leaf position, and its greater child exceeds *x*, swap *x* with that child and continue sift-down.

Because there are $O(\log n)$ stretches, whose tree structures have depth $O(\log n)$, the complexity of trinkle is bounded by $O(\log n)$.

The trees used are systematically slightly unbalanced (any left subtree has depth one more than the corresponding right subtree, except when both are singletons); still, the sift-down operation remains somewhat simpler than one that can handle general binary heaps, since each node has either two children or none; there are fewer cases to distinguish.

Optimization

With respect to the description above, Dijkstra's algorithm implements the following improvement.

The decision of how to handle the incorporation of a new element is postponed until the type of the next element is inspected: if either that next element is the parent of the current element (having it as right child), or is a leaf but the parent of the current element still is within the range of the array being sorted (when it comes along it will take the current element as left child), then instead of trinkle a simple sift within its subtree is done, avoiding comparison with previous roots. For the final element of the array trinkle is always applied. Thus during the beginning of the growing phase, one is most often just applying sift-down in a single stretch; only those "stepson" relations are considered (by trinkle) that will continue to hold until the end of the growing phase.

Shrinking the heap region by separating the rightmost element from it

During this phase, the form of the sequence of stretches goes through the changes of the growing phase in reverse. No work at all is needed when separating off a leaf node, but for a non-leaf node its two children become roots of new stretches, and need to be moved to their proper place in the sequence of roots of stretches. This can be obtained by applying trinkle first for the left child, and then for the right child.

Optimization

In this description, one knows that at the position where trinkle starts the heap property is already valid. So I can simplify by, instead of doing the tests that trinkle needs to do, just compare and possibly swap the element with the root before it (if any), and afterwards apply trinkle at its new position if a swap was actually done. Since a swap may invalidate the heap property at the new position where the smaller root moved to, the simplification only applies to the first step.

Analysis

Smoothsort takes $O(n)$ time to process a presorted array and $O(n \log n)$ in the worst case, and achieves nearly-linear performance on many nearly-sorted inputs. However, it does not handle all nearly-sorted sequences optimally. Using the count of inversions as a measure of un-sortedness (the number of pairs of indices i and j with $i < j$ and $A[i] > A[j]$; for randomly sorted input this is approximately $n^2/4$), there are possible input sequences with $O(n \log n)$ inversions which cause it to take $\Omega(n \log n)$ time, whereas other adaptive sorting algorithms can solve these cases in $O(n \log \log n)$ time.[2]

The smoothsort algorithm needs to be able to hold in memory the sizes of all of the trees in the Leonardo heap. Since they are sorted by order and all orders are distinct, this is usually done using a bit vector indicating which orders are present. Moreover, since the largest order is at most $O(\log n)$, these bits can be encoded in $O(1)$ machine words, assuming a transdichotomous machine model.

Note that $O(1)$ machine words is not the same thing as *one* machine word. A 32-bit vector would only suffice for sizes less than L(32) = 7049155. A 64-bit vector will do for sizes less than L(64) = 34335360355129 ≈ 2^{45}. In general, it takes $1/\log_2(\varphi) \approx 1.44$ bits of vector per bit of size.

References

1. Dijkstra, Edsger W. *16 Aug 1981 (EWD-796a)* (http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF) (PDF). E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. "One can also raise the question why I have not chosen as available stretch lengths: ... 63 31 15 7 3 1 which seems attractive since each stretch can then be viewed as the postorder traversal of a balanced binary tree. In addition, the recurrence relation would be simpler. But I know why I chose the Leonardo numbers:" (transcription (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD07xx/EWD796a.html))
2. Hertel, Stefan (13 May 1983). "Smoothsort's behavior on presorted sequences" (http://scidok.sulb.uni-saarland.de/volltexte/2011/4062/pdf/fb14_1982_11.pdf) (PDF). *Information Processing Letters*. **16** (4): 165–170. doi:10.1016/0020-0190(83)90116-3 (https://doi.org/10.1016%2F0020-0190%2883%2990116-3).
3. Brown, Craig (21 Jan 2013). "Fastest In-Place Stable Sort" (http://www.codeproject.com/Articles/26048/Fastest-In-Place-Stable-Sort). Code Project.

External links

- Commented transcription of EWD796a, 16-Aug-1981 (https://www.enterag.ch/hartwig/order/smoothsort.pdf)