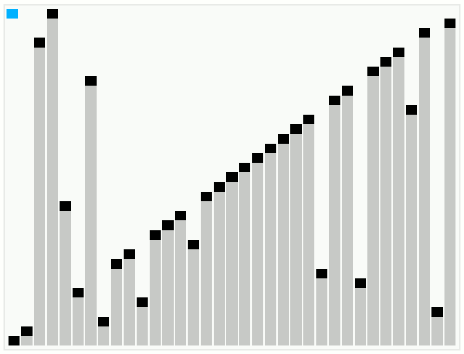
Smoothsort

From Wikipedia, the free encyclopedia

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



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

Contents

Overview

Like heapsort, smoothsort builds up an implicit heap data structure in the array to be sorted, then sorts the array by continuously extracting the maximum element from that heap. Unlike heapsort, Dijkstra's original formulation of smoothsort does not use a binary heap, but rather a custom heap based on the Leonardo numbers; it was later shown that the algorithm can be reformulated in terms of a binary heap without loss of asymptotic efficiency.[2] The heap structure consists of a string of heaps, the sizes of which are all Leonardo numbers, and whose roots are stored in ascending order. The advantage of this custom heap over binary heaps is that if the sequence is already sorted, it takes only $O(n)$ time to construct and deconstruct the heap, hence the better runtime.

Breaking the input up into a sequence of heaps is simple – the leftmost nodes of the array are made into the largest heap possible, and the remainder is likewise divided up. It can be proven [4] that:

- Any array of any length can so be divided up into sections of size L(x).
- No two heaps will have the same size. The string will therefore be a string of heaps strictly descending in size.
- No two heaps will have sizes that are consecutive Leonardo numbers, except for possibly the final two.

Each heap, having a size of L(x), is structured from left to right as a sub-heap of size L(x − 1), a sub-heap of size L(x − 2), and a root node, with the exception of heaps with a size of L(1) and L(0), which are singleton nodes. Each heap maintains the heap property that a root node is always at least as large as the root nodes of its child heaps (and therefore at least as large as all nodes in its child heaps), and the string of heaps as a whole maintains the string property that the root node of each heap is at least as large as the root node of the heap to the left.

The consequence of this is that the rightmost node in the string will always be the largest of the nodes, and, importantly, an array that is already sorted needs no rearrangement to be made into a valid series of heaps. This is the source of the adaptive qualities of the algorithm.

The algorithm is simple. We start by dividing up our unsorted array into a single heap of one element, followed by an unsorted portion. A one-element array is trivially a valid sequence of heaps. This sequence is then grown by adding one element at a time to the right, performing swaps to keep the sequence property and the heap property, until it fills the entire original array.

From this point on, the rightmost element of the sequence of heaps will be the largest element in any of the heaps, and will therefore be in its correct, final position. We then reduce the series of heaps back down to a single heap of one element by removing the rightmost node (which stays in place) and performing re-arrangements to restore the heap condition. When we are back down to a single heap of one element, the array is sorted.

Operations

Ignoring (for the moment) Dijkstra's optimisations, two operations are necessary – increase the string by adding one element to the right, and decrease the string by removing the right most element (the root of the last heap), preserving the heap and string conditions.

Grow the string by adding an element to the right

- If the last two heaps are of size L(x + 1) and L(x) (i.e., consecutive leonardo numbers), the new element becomes the root node of a bigger heap of size L(x+2). This heap will not necessarily have the heap property.
- If the last two heaps of the string are not consecutive Leonardo numbers, then the rightmost element becomes a new heap of size 1. This 1 is taken to be L(1), unless the rightmost heap already has size L(1), in which case the new one-element heap is taken to be of size L(0).

After this, the heap and string properties must be restored, which is usually done via a variant of insertion sort. This is done as follows:

1. The rightmost heap (the one that has just been created) becomes the "current" heap
2. While there is a heap to the left of the current heap and its root is larger than the current root *and* both of its child heap roots
   - Then swap the new root with the root on the heap to the left (this will not disturb the heap property of the current heap). That heap then becomes the current heap.
3. Perform a "filter" operation on the current heap to establish the heap property:
   - While the current heap has a size greater than 1 and either child heap of the current heap has a root node greater than the root of the current heap
     - Swap the greater child root with the current root. That child heap becomes the current heap.

The filter operation is greatly simplified by the use of Leonardo numbers, as a heap will always either be a single node, or will have two children. One does not need to manage the condition of one of the child heaps not being present.

Optimisation

- If the new heap is going to become part of a larger heap by the time we are done, then don't bother establishing the string property: it only needs to be done when a heap has reached its final size.
  - To do this, look at how many elements are left after the new heap of size L(x). If there are more than L(x − 1) + 1, then this new heap is going to be merged.
- Do not maintain the heap property of the rightmost heap. If that heap becomes one of the final heaps of the string, then maintaining the string property will restore the heap property. Of course, whenever a new heap is created, then the rightmost heap is no longer the rightmost and the heap property needs to be restored.

Shrink the string by removing the rightmost element

If the rightmost heap has a size of 1 (i.e., L(1) or L(0)), then nothing needs to be done. Simply remove that rightmost heap.

If the rightmost heap does not have a size of 1, then remove the root, exposing the two sub-heaps as members of the string. Restore the string property first on the left one and then on the right one.

Optimisation

- When restoring the string property, we do not need to compare the root of the heap to the left with the two child nodes of the heaps that have just been exposed, because we know that these newly exposed heaps have the heap property. Just compare it to the root.

Analysis

Smoothsort takes $O(n)$ time to process a presorted array and $O(n \log n)$ in the worst case. However, its performance does not degrade to $O(n \log n)$ as smoothly as Dijkstra claimed; there are possible input sequences with $O(n \log n)$ inversions (elements at indices i and j with $i < j$ and $A[i] > A[j]$) that cause it to take $\Omega(n \log n)$ time, whereas earlier 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 heaps in the string. Since all these values are distinct, this is usually done using a bit vector. Moreover, since there are at most O(log n) numbers in the sequence, these bits can be encoded in O(1) machine words, assuming a transdichotomous machine model.

References

1. Dijkstra, Edsger W. *Smoothsort – an alternative to sorting in situ (EWD-796a)*. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (original (http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF); transcription (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD07xx/EWD796a.html))
2. Hertel, Stefan (1983). "Smoothsort's behavior on presorted sequences" (PDF). *Information Processing Letters*. **16** (4): 165–170. doi:10.1016/0020-0190(83)90116-3.
3. http://www.codeproject.com/Articles/26048/Fastest-In-Place-Stable-Sort
4. Smoothsort Demystified (http://www.keithschwarz.com/smoothsort/). Keithschwarz.com. Retrieved on 2010-11-20.

External links

- Commented transcription of EWD796a (http://www.enterag.ch/hartwig/order/smoothsort.pdf)