Smoothsort Demystified

Last Major Update: January 7, 2011

A few years ago I heard about an interesting sorting algorithm (invented by the legendary Edsger Dijkstra) called smoothsort with great memory and runtime guarantees. Although it is a comparison sort and thus on average cannot run faster than Ω(n lg n), smoothsort is an adaptive sort, meaning that if the input is somewhat sorted, smoothsort will run in time closer to O(n) than O(n lg n). In the best case, when the input is sorted, smoothsort will run in linear time. Moreover, smoothsort is an in-place sorting algorithm, and requires O(1) auxiliary storage space. Compare this to mergesort, which needs O(n) auxiliary space, or quicksort, which needs O(lg n) space on average and O(n) space in the worst case. In the worst case, smoothsort is an asymptotically optimal O(n lg n) sort. With all these advantages, smoothsort seemed like an excellent sort to code up and add to my archive of interesting code project, and I set out to learn enough about it to build my own implementation.

Unfortunately, I quickly found out that smoothsort is one of the least-documented sorts on the web. Sure, you can find many sites that mention smoothsort or its runtime characteristics, and in a few cases sites that provide an implementation, but hardly any sites explained the full intuition behind how the sort worked or where the runtime guarantees came from. Moreover, Dijkstra's original paper on smoothsort is extremely difficult to read and gives little intuition behind some of the trickier optimizations. After spending a few days reading over existing sites and doing a bit of my own work, I finally managed to figure out the intuition behind smoothsort, as well as the source of many of the complex optimizations necessary to get smoothsort working in constant space. It turns out that smoothsort is actually a generalization of heapsort using a novel heap data structure. Surprisingly, I haven't found this structure mentioned anywhere on the web, and this page may be the first time it's been mentioned online.

This page is my attempt to transfer the algorithm's intution so that the beauty of this algorithm isn't lost in the details of a heavily-optimized sorting routine. Although there is a (fairly important!) proof in the middle of this writeup, most of the intution should be immediate once you see the high-level structure. It's a great algorithm, and I hope you find it as interesting as I do.

Background review: Heapsort

Before I actually go over the intuition behind smoothsort, let's take a few minutes to review a similar algorithm, heapsort. This may seem like an unusual first step, but as you see the evolution of the smoothsort algorithm it will become apparent why I've chosen to present things this way.

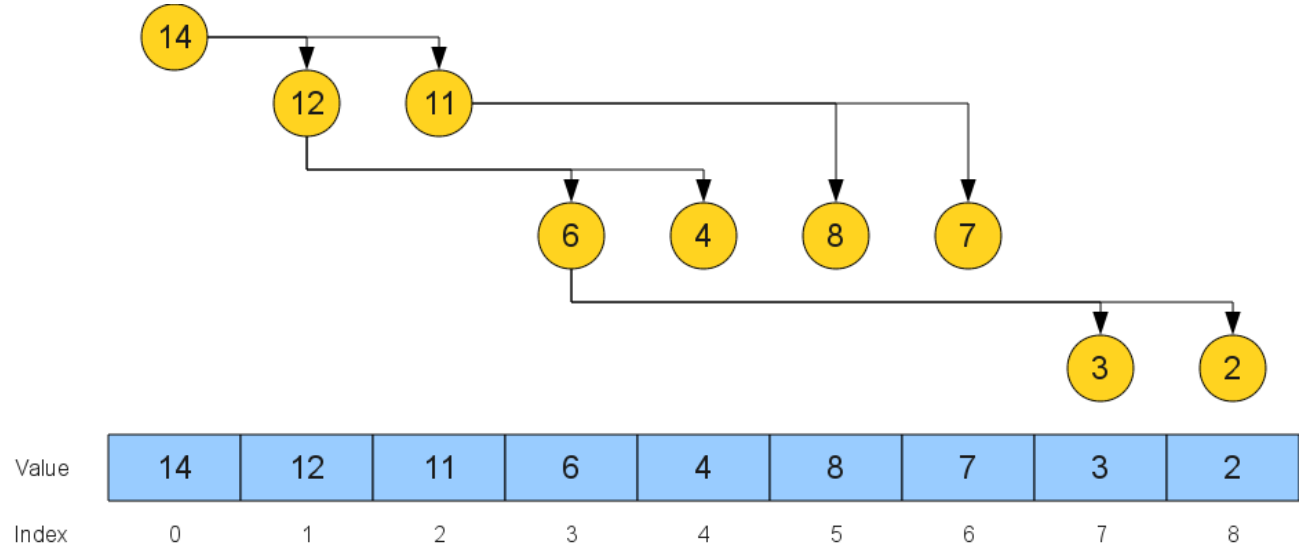
The heapsort algorithm is based on the binary heap data structure. (If you aren't familiar with binary heaps, please read over the link at Wikipedia to get a better sense of how it works before proceeding. It's well-worth your time and I guarantee that you'll like where we're going with this). Conceptually, a binary heap is a complete binary tree where each node is larger than its children. (We could equivalently use min-heaps, where each node is smaller than its children, but the convention in heapsort is to build a max-heap). Because binary heaps are complete, the height of a binary heap over n elements is O(lg n). Moreover, it's possible to build a binary heap in linear time. Finally, given a binary heap, one can remove the maximum element from the heap and rearrange the nodes to restore the heap property in O(lg n) time. This gives a naive implementation of heapsort, which works as follows:

1. Construct a max-heap from the input sequence in O(n) time.
2. Set x to the index of the last spot in the array.
3. While the heap is not yet empty:
    1. Remove the maximum element from the heap.
    2. Place that element at position x, then move x to the previous position.
    3. Rebalance the max-heap.

A sketch of the proof of correctness for this algorithm is reasonably straightforward. We begin by putting all of the elements in the range into the max-heap, and as we dequeue the elements we end up getting everything in the original sequence in descending order. We then place each of these elements at the last unused spot in the array, essentially placing the ascending sequence in reverse order. Since we can build the max-heap in O(n) time and rebalance the heap in O(lg n) time, the overall runtime is O(n + n lg n) = O(n lg n), which is asymptotically optimal.

In practice, though, no one actually implements heapsort this way. The problem is that the above algorithm ends up using O(n) extra memory to hold the nodes in the max-heap. There is a much better way of implementing the heap implicitly in the array to be sorted. If the array is zero-indexed, then the parent of a node at position i is at position $\lfloor (i + 1) / 2 \rfloor - 1$; conversely, a node's

children (if they exist) are at positions 2i + 1 and 2i + 2. Because the computations necessary to determine a node's parent and children can be done in a handful of assembly instructions, there's no need to explicitly store it anywhere, and the O(n) memory that was dedicated to storing links can instead be recycled from the O(n) memory of the input itself. For example, here's a max-heap and its corresponding implicit representation:



The in-place heapsort works by rearranging the array elements into an implicit binary heap with its root in the first position and an arbitrary leaf in the last position. The top element of the heap is then dequeued and swapped into its final position, placing a random element at the top of the binary heap. The heap is then rebalanced and this process repeated until the elements are sorted.

Because the heap is represented implicitly using the array itself, only a constant amount of memory must be used beyond the array itself to store the state of the algorithm (in particular, things like the next position to consider, data for the rebalance heap step, etc.) Consequently, this version of heapsort runs in O(n lg n) time and O(1) auxiliary space. This low memory usage is very attractive, and is one of the reasons that heapsort is so prevalent as a sorting algorithm.

A High-Level Overview of Smoothsort

The one major shortcoming of heapsort is that it always runs in Θ(n lg n). The reason for this has to do with the structure of the max-heap. When we build up the implicit binary heap in heapsort, the largest element of the heap is always on the leftmost side of the array, but ultimately the element must be on the right side. To move the max element to its proper location, we have to swap it with

an essentially random element of the heap, then do a bubble-down operation to rebalance the heap. This bubble-down operation takes Ω(lg n) time, contributing to the overall Ω(n lg n) runtime.

One obvious insight to have is this: what if we build a max-heap but store the maximum element at the *right* of the sequence? That way, whenever we want to move the maximum element into the next open spot at the end of the array, we're already done - the maximum element will now be in the correct position. All we need to do now is rebalance the rest of the elements. Here, we encounter a somewhat cool property. When we take off the root of the heap and "break the heap open" to expose the two max-heaps living under the root. Because each of these are roots of max-heaps, they're bigger than all of the elements of their respective max-heaps, and so one of these two elements is the largest element of what remains. The problem now, though, is that we've fractured our nice max-heap into two distinct max-heaps, and the only way we know how to rebalance them is to swap up a leaf and bubble it down. This is the killer step, since it's almost certainly going to take Ω(lg n) time, forcing our runtime to be Θ(n lg n). We're aiming to get O(n) in the best case, and so this isn't going to work.

Here is the key idea that makes smoothsort possible - what if instead of having a *single* max-heap, we have a *forest* of max-heaps? That is, rather than having a single max-heap, we'll maintain a sequence of max-heaps embedded into the array. That way, it's not a problem if we end up breaking apart one heap into multiple parts without putting it back together. Provided that we don't end up having too many heaps at any one time (say, O(lg n) of them), we can efficiently find the largest element of what remains.

At a high level, smoothsort works as follows. First, we make a linear scan over the input sequence, converting it into a sequence of implicit max-heaps. These heaps will not be binary heaps, but rather an unusual type of heap described below called a *Leonardo heap*. In the course of doing so, we maintain the property that the heaps' top elements are in ascending order, forcing the rightmost heap to hold the maximum of the remaining elements. Once we've done this, we'll continously dequeue the top element of the rightmost max-heap, which is in the correct location since it's in the rightmost unfilled spot. We'll then do some manipulations to reestablish the heaps and the sorted property. These guarantees, plus a bit of clever mathematics, guarantee that the algorithm runs quickly on sorted sequences.

The initial implementation of smoothsort that we'll do will end up having excellent runtime guarantees, but high memory usage (O(n)). We'll then see an

optimization that compresses this down to O(lg n), and finally a theoretically dubious trick that ends up reducing the space requirement to O(1).

Leonardo Trees

Before we can get to the actual smoothsort implementation, we need to discuss the structure of the heaps we'll be building up. These heaps are based on the Leonardo numbers, a sequence of numbers similar in spirit to the better-known Fibonacci numbers. I have actually not seen these heaps used anywhere other than smoothsort, and so for lack of a better name I'll refer to them as *Leonardo heaps.*

The Leonardo numbers (denoted L(0), L(1), L(2), ...) are given by the following recursive formulation:

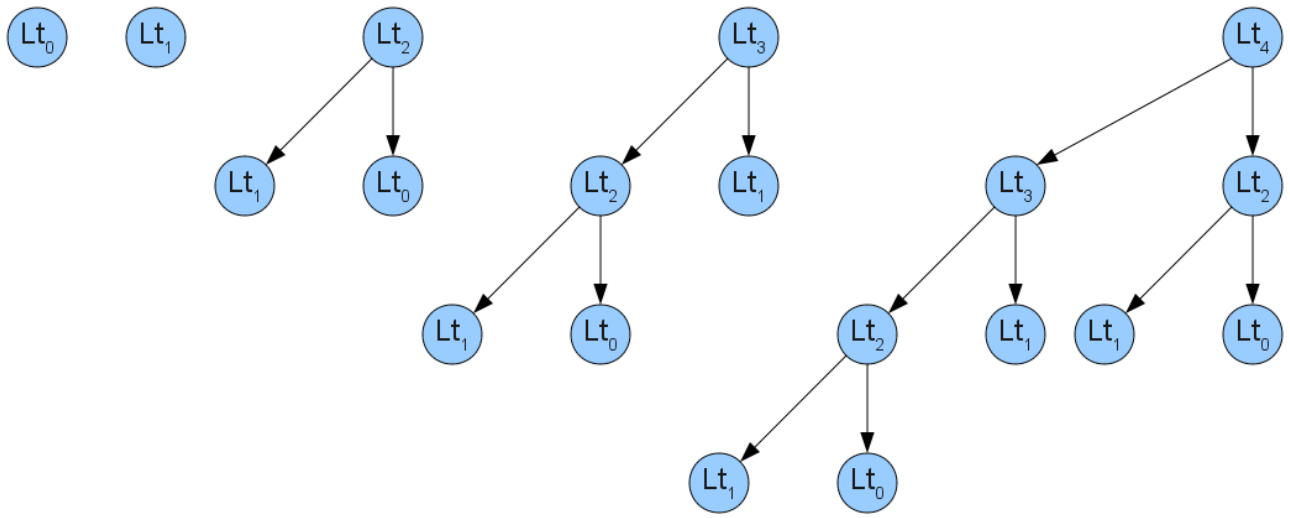
- $L(0) = 1$
- $L(1) = 1$
- $L(n + 2) = L(n) + L(n + 1) + 1$

For reference, the first few Leonardo numbers are 1, 1, 3, 5, 9, 15, 25, 41, 67, and 109.

A *Leonardo tree of order k* (denoted Lt_k) is a binary tree with a rigidly-defined shape. Leonardo trees are defined recursively as follows:

- Lt_0 is a singleton node.
- Lt_1 is a singleton node.
- Lt_{n+2} is a node with two children, Lt_n and Lt_{n+1} (in that order)

You can show with a fairly simple inductive proof that the number of nodes in Lt_k is L(k), hence the name. To make it a bit easier to intuit the structure of these trees, here are some pictures of the first few Leonardo trees:



This seems like a pretty random data structure choice... why is it at all useful? And where does it come from? It turns out that these heaps are not at all chosen randomly. In particular, there is a useful result about Leonardo numbers (and consequently Leonardo trees) that makes them invaluable in the smoothsort algorithm. This is the only proof in this entire writeup, but I strongly encourage you to read it. The proof of this result will be adapted into the main loop of the algorithm, so a good intuitive understanding of what's going here might be helpful later on.

Lemma: Any positive integer can be written as the sum of O(lg n) distinct Leonardo numbers.

Proof: We'll begin by proving the first half of this claim by proving a much stronger claim: for any positive integer n, there is a sequence x_0, x_1, ..., x_k such that:

1. $\sum_i L(x_i) = n$
2. $x_0 < x_1 < \ldots < x_k$.
3. If $x_0 = 0$, then $x_1 = 1$.
4. For any $i > 0$, $x_i + 1 < x_{i+1}$

That's a lot to process, so let's try to give an intuitive feel for what's happening. We're claiming that there is some sequence of Leonardo numbers (indexed by the ascending sequence x_0, x_1, x_2, etc.) that sums up to the number n. Furthermore, the sequence doesn't use the Leonardo number L(0) until first using L(1) (since L(0) = L(1), this makes the proof a lot easier). Finally, if the sum contains two consecutive Leonardo numbers, then those are the smallest

two Leonardo numbers in the sequence. Recall that the Leonardo numbers are defined as $L(n + 2) = L(n + 1) + L(n) + 1$. This last claim states that whenever there are two adjacent Leonardo numbers in the sequence, they must be the smallest numbers in the sequence so that all merges of Leonardo numbers happen at the end. This claim isn't strictly necessary for the correctness proof, but will show up in the smoothsort algorithm and I've included it here.

The proof of this claim is by induction on n. As a base case, if $n = 0$, then take the x's to be the empty sequence and all four claims are satisfied. For the inductive step, assume that for some number n the claim holds and consider the number $n + 1$. Start off by writing $n = L(x_0) + L(x_1) + L(x_2) + ... + L(x_k)$ for some sequence of x's meeting the above criteria. There are then three cases to consider:

Case 1: $x_0 + 1 = x_1$. In this case, note that $L(x_0) + L(x_1) + 1 = L(x_0) + L(x_0 + 1) + 1 = L(x_0 + 2)$. Next, note that $x_0 + 2 = x_1 + 1 < x_2$. If we let $y_0 = x_0 + 2$ and then let $y_i = x_{i+1}$ for $i > 0$, then $\sum_i L(y_i) = y_0 + \sum_{i=1} y_i = L(x_0 + 2) + \sum_{i=2} L(x_i) = 1 + L(x_0) + L(x_1) + \sum_{i=2} L(x_i) = 1 + n$, so the first claim holds. The second claim holds for the y's by the above logic relating $x_0 + 2$ to x_2. Claim (3) holds since $y_0 \neq 0$, and claim (4) because for any $i > 0$, $y_i + 1 = x_{i+1} + 1 < x_{i+2} = y_{i+1}$.

Case 2: $x_0 = 1$, and case 1 does not apply. Since case 1 does not apply, we know that $x_0 + 1 < x_1$. Let $y_0 = 0$, $y_i = x_{i-1}$ for $i > 0$. Then $\sum_i y_i = L(0) + \sum_i x_i = 1 + n$, so the first claim holds. The second claim holds because it held for the x's initially, $x_0 = 1$, and the new first element (y_0) is 0. The third claim holds because $y_0 = 0$ and $y_1 = x_0 = 1$. Finally, because we aren't in case 1, $x_0 + 1 < x_1$, so for any $i > 1$, $y_i + 1 = x_{i-1} + 1 < x_i = y_{i+1}$, so the claim holds for $n + 1$.

Case 3: $x_0 \neq 1$, and case 1 does not apply. We know that $x_0 \neq 0$, since if it were so, by (3) $x_1 = 1$, and so case 1 would apply, a contradiction. Thus $x_0 > 1$. Then let $y_0 = 1$, $y_i = x_{i-1}$ for $i > 0$. Using similar logic to the above (and the fact that $L(0) = L(1) = 1$), the sum of these y's is equal to $n + 1$. Since $x_0 > 1$, the y's are in ascending order. $x_0 \neq 0$, so the third claim does not apply. Finally, since case one did not apply, $x_0 + 1 < x_1$, and so the final claim holds, and the claim holds for $n + 1$

These three cases are exhaustive and mututally exclusive, and so the induction is complete.

Finally, we need to show that each of the sequences described above uses at most O(lg n) Leonardo numbers. To do this, we show that $L(k) = 2 F(k + 1) - 1$, where $F(k + 1)$ is the (k+1)st Fibonacci number. From there, we have that L(k)

$> (2 / \sqrt{5})\varphi^{(k+1)}$, with $\varphi = (1 + \sqrt{5}) / 2$ by the closed-form equation for Fibonacci numbers.. We then have that for any n, if we let $k = \lceil \log_{\varphi}((\sqrt{5}/2)\, n) \rceil = O(\lg n)$, we have that $L(k) > n$. Since n can be written as the sum of unique Leonardo numbers, none of which can be bigger than n (and thus no greater than L(k)), this means that n can be written as the sum of some subset of the first k Leonardo numbers, of which there are only O(lg n).

The proof that $L(k) = 2F(k + 1) - 1$ is by induction on k. For $k = 0$, $2F(1) - 1 = 2 - 1 = 1 = L(0)$. For $k = 1$, $2F(2) - 1 = 2 - 1 = 1 = L(1)$. Now assume that for all $k' < k$, the claim holds. Then $L(k) = L(k - 2) + L(k - 1) + 1 = 2F(k - 1) - 1 + 2F(k) - 1 + 1 = 2(F(k - 1) + F(k)) - 1 = 2F(k + 1) - 1$.

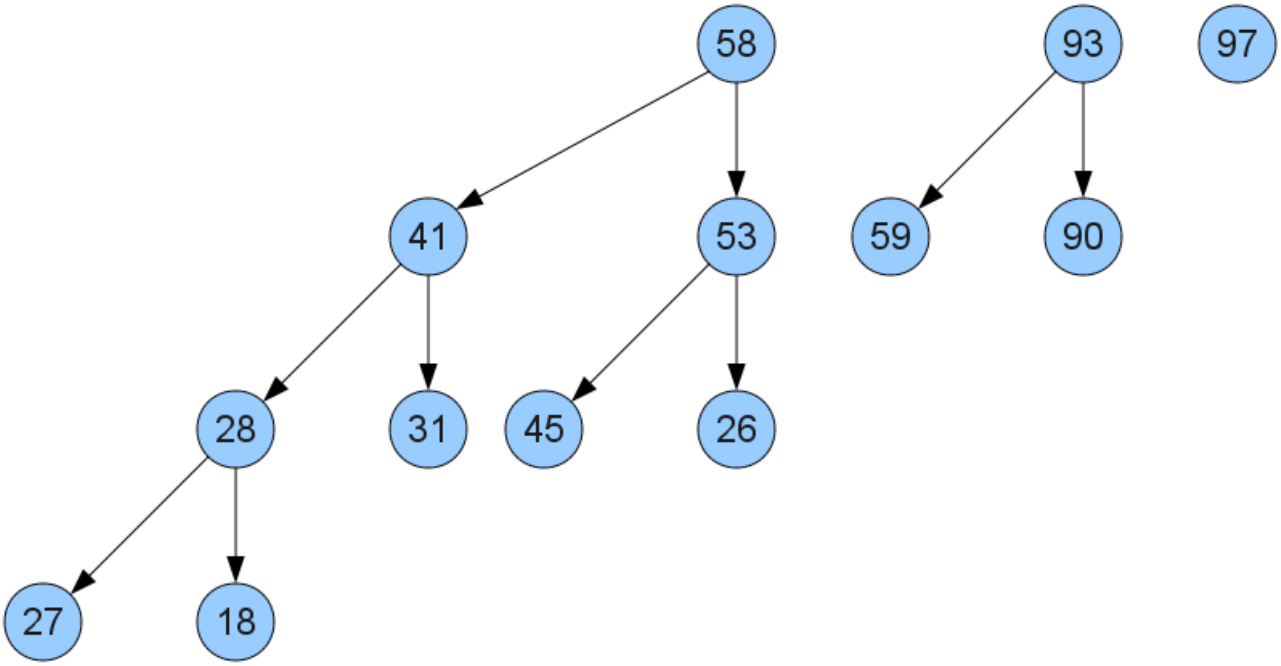
QED

Leonardo Heaps

Much in the same way that you can build a binomial heap using a collection of binomial trees (not required reading, but highly recommended!), you can build a "Leonardo heap" out of a collection of Leonardo trees. A Leonardo heap is an ordered collection of Leonardo trees such that:

1. The sizes of the trees is strictly decreasing. As an important consequence, no two trees have the same size.
2. Each tree obeys the max-heap property (i.e. each node is at least as large as its children)
3. The roots of the trees are in ascending order from left to right.

Here is a sample Leonardo heap:



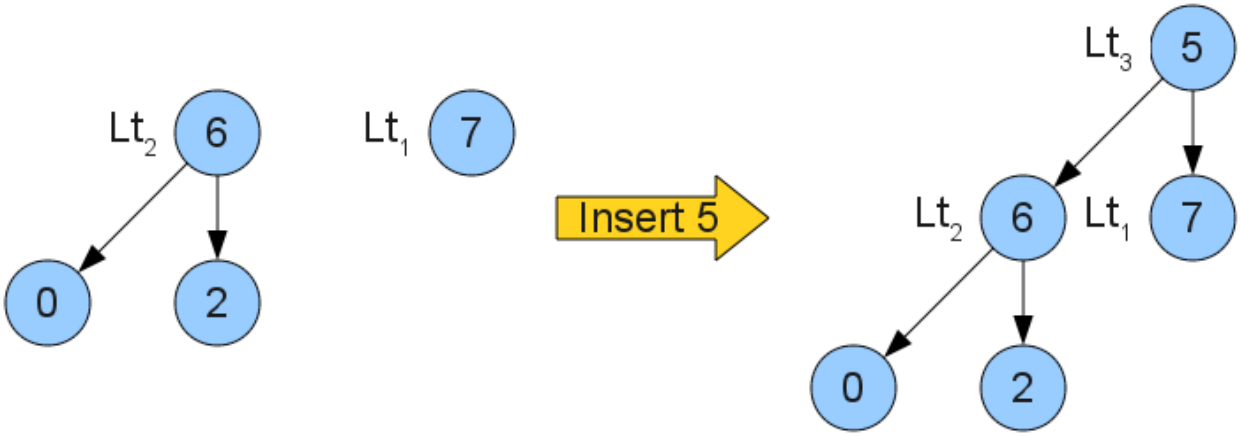
Notice that properties (1) and (3) of Leonardo heaps mean that the smallest heap has the largest root and the largest heap has the smallest root. The roots increase from left to right.

In order for Leonardo heaps to qualify as max-heaps, we'll need to implement some basic functionality on them. In particular, we'll show how to implement heap insert and dequeue-max.

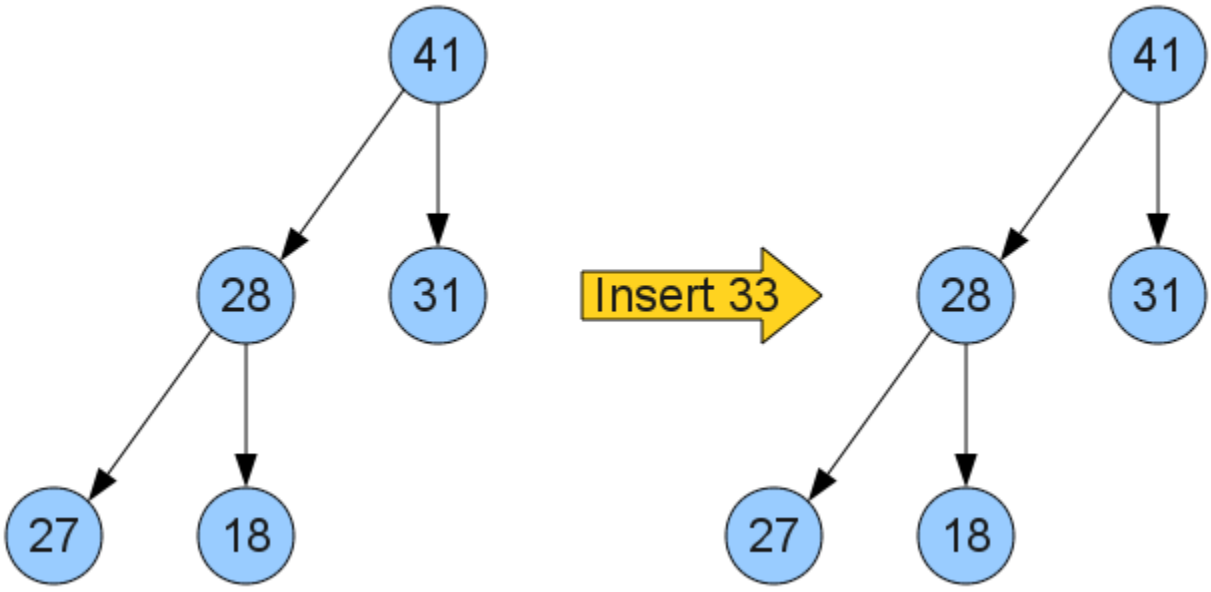
Inserting into a Leonardo heap. There are three steps to inserting into a Leonardo heap. First, we need to ensure that the resulting heap has the correct shape; that it's a collection of Leonardo trees of unique size stored in descending order of size. Next, we need to ensure that the tops of the heaps are sorted in ascending order from left to right. Finally, we'll ensure that each of the Leonardo trees obeys the max-heap property. Let's start by seeing how to get the shape right.

Earlier we proved that each number can be partioned into a sum of descending, unique Leonardo numbers obeying certain properties, and the algorithm for inserting into a Leonardo heap is based on the three cases of the proof. We begin by checking whether the two smallest Leonardo trees correspond to consecutive Leonardo numbers L(k) and L(k + 1). If so, we create a new Leonardo tree of type $Lt_{(k+2)}$ with the inserted element as the root. Otherwise, if the smallest Leonardo tree is of size L(1), we insert the new element as a

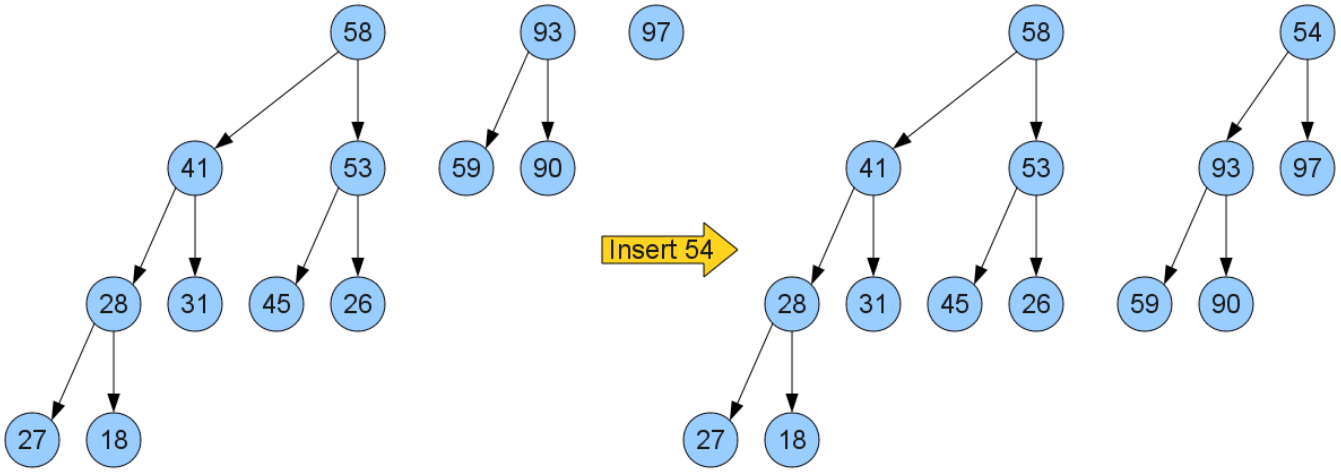
singleton Leonardo tree of size L(0). Finally, if neither other case applies, we insert the new element as a singleton Leonardo tree of size L(1). The proof that this ends up producing a sequence of Leonardo trees of decreasing size is almost identical to the earlier proof, and so I omit it. Here are a few pictures of different insertions into a Leonardo heap:



A simple merge of Lt_2 and Lt_1 with a new node to form Lt_3

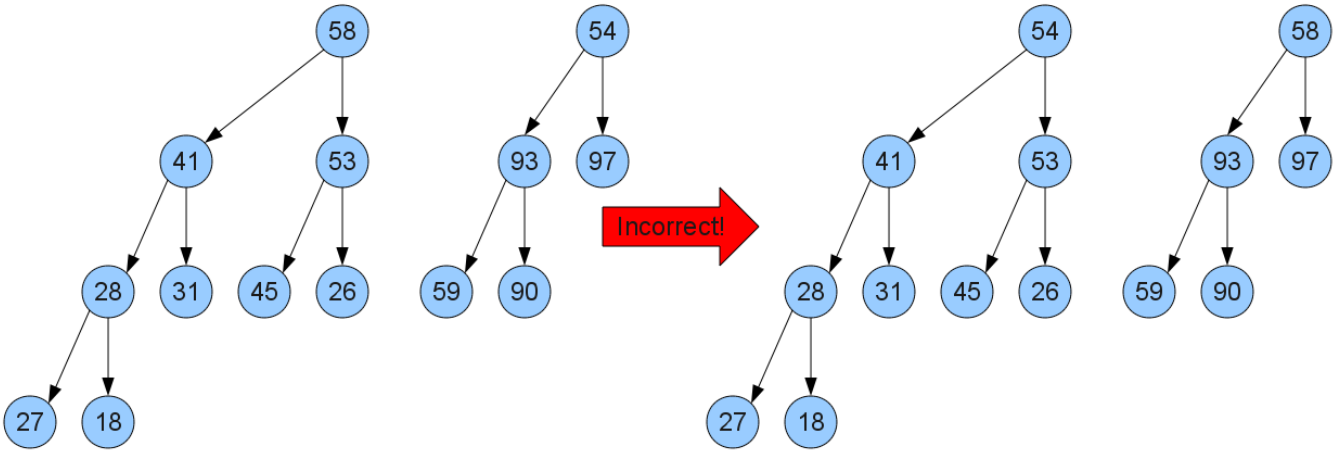


Creating a new Lt_1 in addition to the existing Lt_3.



Merging together the Lt_2 and Lt_1 trees into an Lt_3 tree, ignoring the Lt_4 tree as it isn't one of the two smallest.

Now, let's see how to guarantee that the topmost elements of each heap are sorted in ascending order from left to right. Essentially, this step does an insertion sort of the new value into the list defined by the roots of all of the Leonardo trees, though it's a bit more complex than that. In particular, because the new element is not necessarily the largest element of the tree containing it (because we haven't yet restored the heap property), if we naively swap the new element down until it comes to rest, we can't be guaranteed that the heap property will hold for any of the heaps that were modified. For example, consider this setup, which corresponds to the state of the Leonardo heap in the third of the above examples:



An erroneous swap

Here, the root of the rightmost heap (the number 54) was just added. If we naively insertion sort it down to its proper resting place, then in the end we might need to restore the heap property for each of the heaps we swapped roots with, as seen by the fact that the new root of the rightmost heap is smaller than any of its elements.

The problem with this is that what we really want to do is insertion sort on the values that will *ultimately* be at the roots of the trees. Furthermore, we'd like to do this as efficiently as possible; that is, without reheapifying each tree at every step. Fortunately, we can do this fairly easily. Given a "mostly-heapified" Leonardo tree (one where the root may be out of place, but the rest of the structure is valid), we can guarantee that the node that ultimately ends up being the root of the tree is either the root or the roots of one of its two children. Consequently, we do a modified insertion sort, swapping the root of the preceding tree with the current one only if its root is bigger than the new element *and* the roots of its child nodes. Note that if this is true, after the swap the tree that used to contain the new element is now a correctly-balanced heap, since the new root is bigger than either of the roots of the subtrees.

Finally, once we end up in a situation where the new root is atop the correct heap, we can use the heapify operation originally developed for binary heaps to restore the heap property to that tree. At this point, as mentioned above, all of the trees to the right are valid max-heap Leonardo trees, the current tree is valid, and the trees to the left are all unchanged. Moreover, the roots of the trees are in descending order, since we used an (albeit modified) insertion sort to rearrange them.

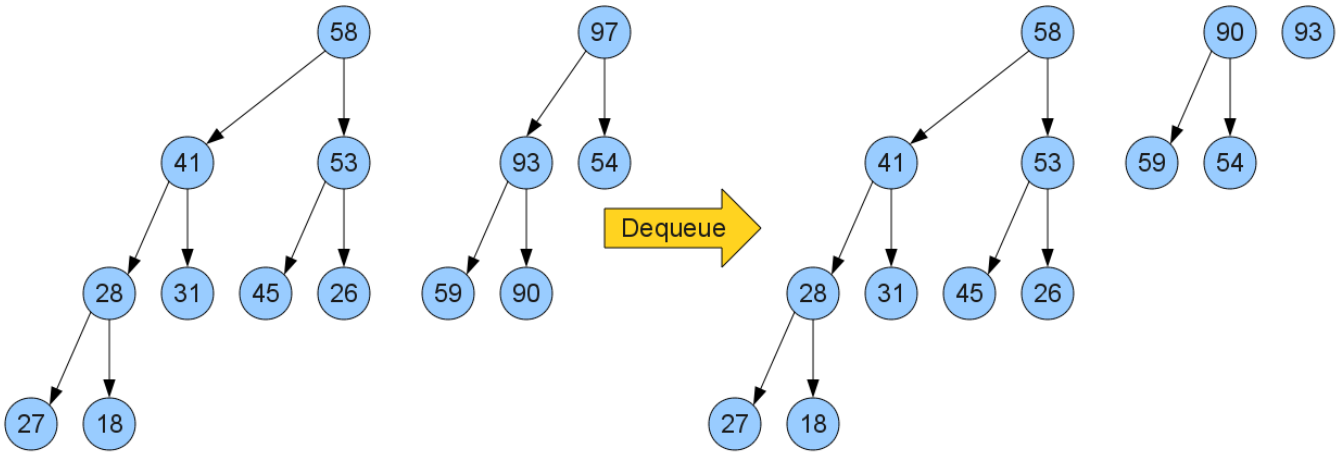
Let's consider the runtime of this operation. Creating a new Leonardo tree from the new element and (possibly) the two preceding trees can be done in O(1). The insertion sort step might move the new element across the tops of at most O(lg n) trees (since, as mentioned before, the partitioning of the n elements into distinct Leonardo numbers uses at most O(lg n) such numbers), and when it finally comes to rest, it's inserted into a Leonardo tree of order at most O(lg n). A quick inductive argument can be shown that the height of a Leonardo tree of size k is O(k), and so this bubble-down step takes at most O(lg n) time, netting an insert time of O(lg n).

However, what if the element we're inserting is the largest element in the heap? In that case, the heap-building runtime is the same, but the time to insertion-sort the element into place is now O(1) instead of O(lg n), and the time to heapify the tree containing the new element is also O(1) since no rearrangements are made. In other words, inserting a new max element into a

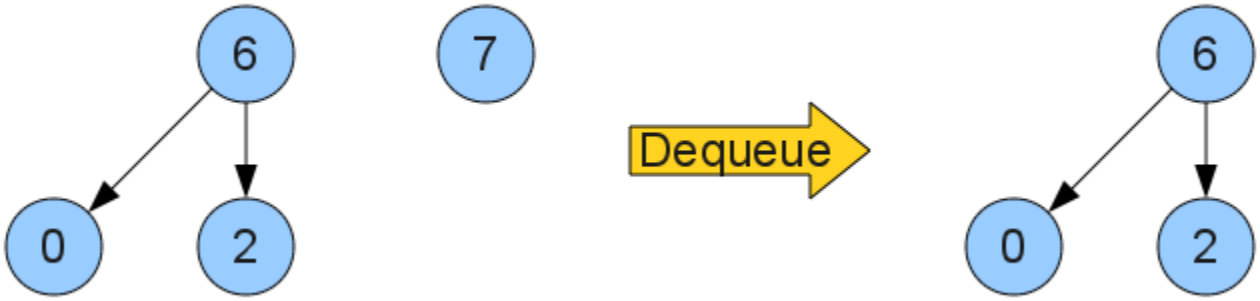
Leonardo heap takes time O(1). This is crucial to getting smoothsort to run in O(n) time on already-sorted inputs.

Dequeuing elements from a Leonardo heap.. This process is similar to the process for building a Leonardo heap, though with a bit more bookkeeping. We know that the largest element is atop the smallest heap, and so we can dequeue it quite easily. There are now two cases to consider, depending on what kind of heap the last element was in. If it was a Lt_0 or Lt_1 heap, then all of the guarantees we had about the heap structure still hold, since all we did was remove a heap from the front of the list. Otherwise, the heap has two child heaps which have just been "exposed" to the rest of the trees. To rebalance the heap, we apply the modified insertion sort algorithm to reposition the root of the leftmost tree, then heapify whichever tree the root ends up in. We then do the same for the rightmost tree. Once this step is complete, all of the heap properties are satisfied and we are done.

Here are some examples of Leonardo heap dequeues:



Dequeuing from this Leonardo heap splits the Lt_3 heap in two and forces a rebalance.



Dequeuing from this heap deletes the last element and does not necessitate a rebalance.

What is the runtime of this algorithm? As mentioned earlier, the insertion-sort-and-heapify operation runs in O(lg n) time in the worst case. However, if the roots of whatever heaps were just exposed are already in the correct position (i.e. neither one needs to be moved)? In this case, the dequeue operation is O(1). This happens if the input is already sorted to some extent, and in particular if the elements fed in to the Leonardo heap were already sorted. Consequently, using a Leonardo heap to sort a range of already-sorted elements takes time O(n).

A First Attempt at Smoothsort

From this definition of Leonardo heaps alone, we can get a first approximation of the smoothsort algorithm. This algorithm, which looks surprisingly similar to regular heapsort, is as follows:

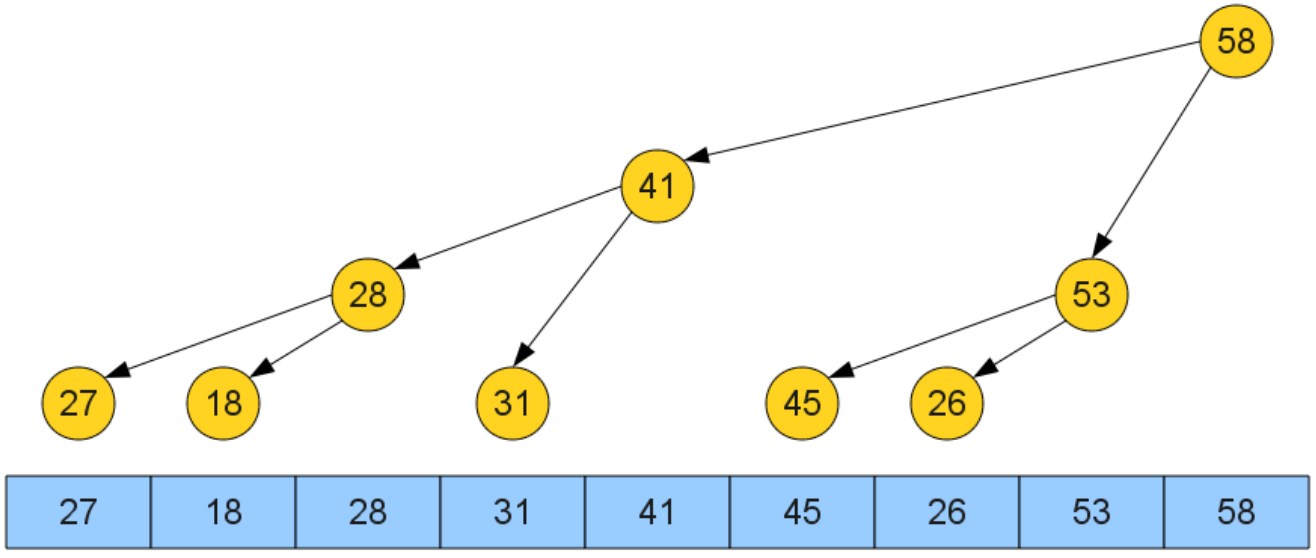
1. Construct a Leonardo max-heap from the input sequence in O(n) time.
2. Set x to the index of the last spot in the array.
3. While the heap is not yet empty:
    1. Remove the maximum element from the heap.
    2. Place that element at position x, then move x to the previous position.
    3. Rebalance the max-heap.

In the worst case, this algorithm runs in O(n lg n), since each insert or removal could run in O(lg n) time. If the input sequence is already sorted, though, the algorithm will run in O(n) time. This current version of smoothsort uses O(n) memory and is not in-place, but is still pretty elegant nonetheless. The rest of this page deals with how to whittle down the memory usage to O(1).

"Mostly Implicit" Leonardo Heaps in O(lg n) Space

The naive heapsort algorithm runs in O(n lg n) and uses O(n) memory to maintain the explicit binary heap. Switching from an explicit representation of the max heap to an implicit representation cuts the memory usage down to O(1) without sacrificing any performance. Can we do the same to the Leonardo heap? The answer is yes, but the method is somewhat indirect. We can move from an explicit Leonardo heap that uses O(n) memory to a "mostly implicit" Leonardo heap that requires only O(lg n) extra space by using the input array to be sorted to encode the heap. From there, only a bit of hacky mathematics stops us from fitting things into O(1).

We'll begin our discussion by talking about a way of implicitly representing a single Leonardo tree using O(1) auxiliary storage space. This ends up not being particularly difficult and can be done inductively. For starters, Lt_0 and Lt_1, the first two Leonardo trees, are both a single node and can easily be represented implicitly in an array. Then, given a Leonardo tree of any other order $k > 1$, we can represent it as the concatenation of its child of size $k - 1$, then its child of size $k - 2$, and then its root node. For example, here is a Leonardo tree of order 4 and its corresponding representation:



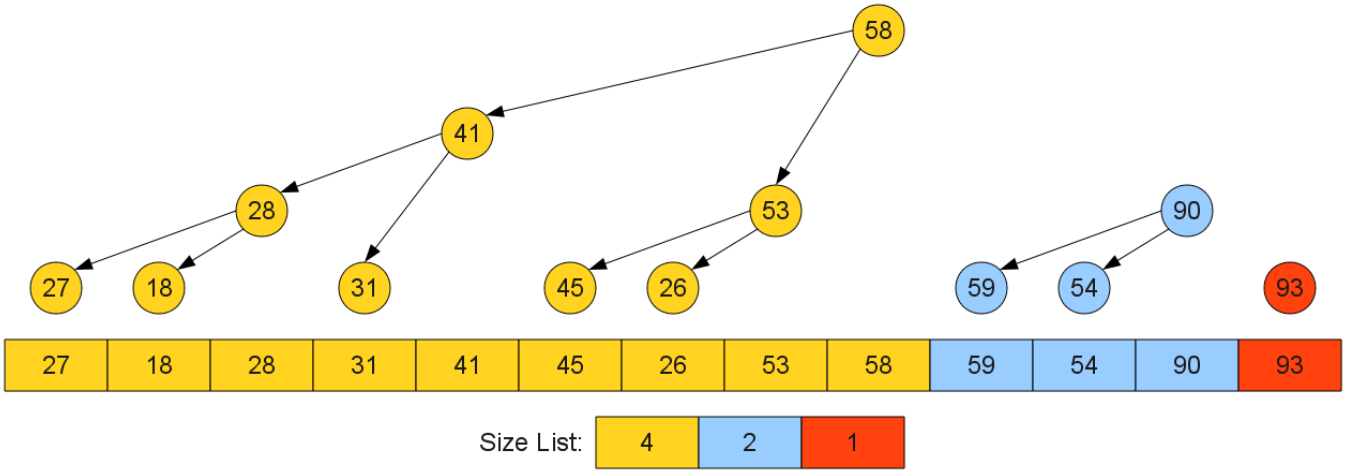
Given such a representation, how do we navigate around in it to get from the root to its subtrees? Well, we know that the root is the rightmost element. If we take one step to the right, we're looking at the root element of the smaller subtree. If we then jump backwards by the size of that tree, we're looking at the root element of the larger of the two subtrees. This gives us an easy procedure

for navigating around implicit Leonardo trees. Assuming we are looking at the encoding of a Leonardo tree of order k in a zero-indexed array of size L(k):

- The root of the tree is at position L(k) - 1.
- The root of the Lt_{k-1} subtree is at position L(k - 1) - 1.
- The root of the Lt_{k-2} subtree is at position L(k) - 2.

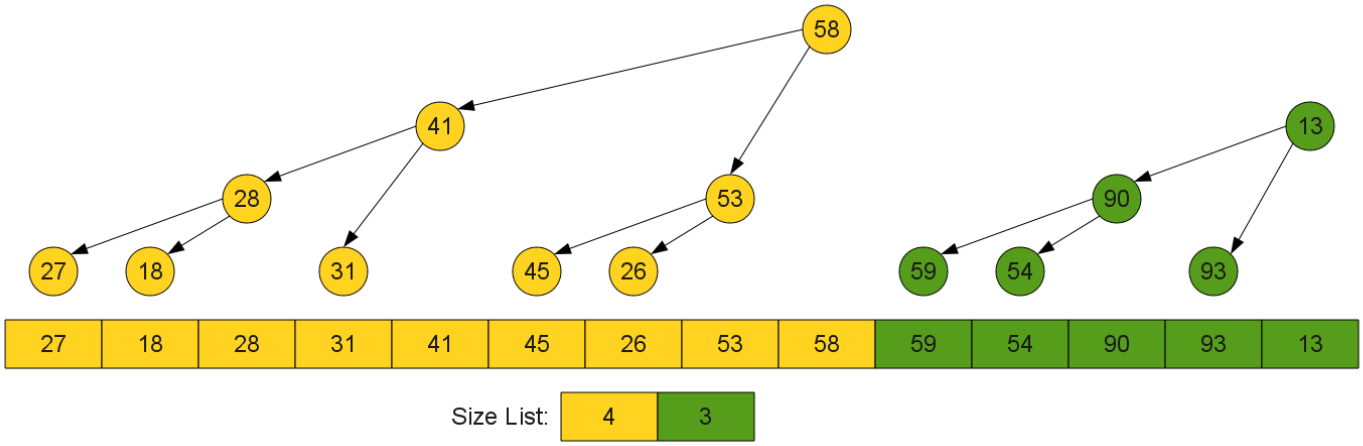
Assuming we have O(1) access to each Leonardo number at a given position, we can descend one level in the tree in constant time. We can guarantee this by memoizing the result of each computation of a value L(k), or by precomputing every single L(k) less than the maximum sequence length representable on the given machine.

However, this discussion only talks about how to represent a single Leonardo tree implicitly, not a forest of them as we've done in a Leonardo heap. Fortunately, with a little extra overhead to track where each representation starts and ends, we can easily adapt it to represent entire Leonardo heaps. The idea is simple - we encode the Leonardo heap implicitly as the concatenation of all of the representations of all of its trees (in descending order of size), along with an auxiliary list storing the sizes of each of the heaps. Because each individual implicit Leonardo tree has its root at the rightmost element, the rightmost element of the entire array must be the root of the smallest heap. The information in the auxiliary list then lets us locate any tree in the heap in O(k) time, where k is the length of the list, by starting at the leftmost tree, then skipping backwards past the lengths of each intermediary tree until we arrive at our destination. For example, here is a Leonardo heap and its corresponding implicit representation:



As mentioned earlier, any Leonardo heap of size n has at most O(lg n) trees in it. This means that the list of tree sizes is therefore of size O(lg n), and we can look up the location of any tree in the heap in O(lg n) time.

One major advantage of this representation is that it naturally supports Leonardo heap insertion and removal. When inserting a new element into the Leonardo heap, we can check in O(1) time whether the two rightmost heaps are mergable by checking whether the last two elements of the size list differ by one. If so, we can merge them implicitly in constant time by adding the new element to the end of the array, then replacing the last two entries of the size list with the size of the resulting heap, as shown here:



From here, we can apply the pseudo-insertion sort and heap rebalance operations with only a constant factor more work to look up the position of the previous heaps and the values of the roots of their child heaps.

The cases where we insert a new tree of type Lt_0 or Lt_1 are also easily accomplished. We can check which type to insert by looking at the last element of the heap list in O(1), then appending the (singleton) representation of these trees to the array. In either case, we add the proper size information to the end of the heap list in O(1).

Perhaps the biggest advantage of this mostly-implicit representation is that it allows for an efficient dequeue max that leaves the largest element of the heap in its proper place in the sorted array. Given a mostly-implicit representation of a Leonardo heap, the maximum element is always in the rightmost spot in the array. To dequeue it, we simply leave it in place, treat the rest of the elements as the remaining Leonardo heap, then do the rebalance operation. Rebalancing is similar to the original case, though we need to make corresponding changes

to the size list in addition to everything else. In particular, on dequeuing an element in a tree of type Lt_0 or Lt_1, we simply discard the last entry from the size list. On dequeuing an element of a tree of order $k > 1$, we represent the newly-exposed trees by replacing the last entry of the size list with two new entries $k - 1$ and $k - 2$ (in that order).

In short, this "mostly-implicit" representation allows us to perform all of the normal operations on a Leonardo heap while reducing the memory usage from O(n) to O(lg n).

Given this mostly-implicit Leonardo heap implementation, we can rewrite our smoothsort implementation accordingly:

1. Construct a mostly-implicit Leonardo max-heap from the input sequence.
2. While the heap is not yet empty:
    1. Remove the maximum element from the heap.
    2. Place that element at the back of the sequence.
    3. Rebalance the max-heap.

The beauty of this sorting algorithm is that, at a high level, it's identical to what we had before. There is a strong connection between priority queues and sorting algorithms at work here - the more we refine our priority queues, the better our sorting algorithms get.

Implicit Leonardo Heaps in O(1) Space

At this point we have an extremely good sorting algorithm: it's adaptive and uses only O(lg n) memory. But to truly round out the algorithm, we need to further cut down on its space usage. Our goal will be to get this entire algorithm working with only O(1) auxiliary storage space. This step is going to be extremely difficult, and will require a combination of clever bitwise hackery and amortized analyses.

The basic idea of this next step is to take the size list and compress it down from using O(lg n) space to using O(1) space by encoding the size list in a specially-modified bitvector. Rather than diving in headfirst and looking at the final result (which is, by the way, fairly terrifying), let's ease into it by reviewing a few simple properties of Leonardo numbers that we've talked about earlier.

If you'll recall, we proved that when partitioning an integer into a sequence of Leonardo numbers, we can do so such that the numbers have **unique order** (i.e. we don't use the same Leonardo number twice). This means that for each Leonardo number, either the number is in the size list or it isn't. This means that if all we care about is whether a tree of a particular order exists in the size list, we can store the answer using a single bit of information Moreover, we know that when splitting a sequence apart into Leonardo trees, those trees always appear in descending order. Consequently, if we knew which trees were in the Leonardo heap, we could recover their order implicitly by simply finding which tree was smallest. This suggests an entirely different approach to storing the size list - a *bitvector* with enough entries to hold all the Leonardo numbers that might reasonably come up during the algorithm's execution.

Before we go into some of the subtleties or complexities involved with using a bitvector, we should first ask an important question - how many bits are we going to need for this vector? We know that for any sequence of length n, there can be at most O(lg n) Leonardo trees in the heap, and so we'll need O(lg n) bits. Amazingly, we can encode all of these O(lg n) bits using O(1) machine words! To see this, we'll first make an assumption that the computer we're on has *transdichotomous memory*. Informally, a machine is transdichotomous if each machine word has size at least Ω(lg n). The logic behind this idea is that each machine word is large enough to store a pointer to any other location in memory. Virtually all computers have this property - on a 32-bit machine, there are 2^{32} addressable bytes, and four bytes collectively can store a pointer anywhere in memory. A similar claim holds for 64-bit machines. Note, however, that this is *not* the same thing as claiming that lg n = O(1). That would be tantamount to saying that the input never gets larger than some size. Rather, the idea is that when we have the input to our problem, we can only run the sorting algorithm on it if we go to a machine that has sufficient space for it, and on that machine we assume that the word size is at least Ω(lg n). In other words, as our problem size grows, so does the size of each word of memory we're using.

The fact that we only have O(lg n) bits in our bitvector, coupled with the fact that each machine's word size is Ω(lg n), means that while the number of *bits* necessary goes up as the problem size increases, the number of *words* needed to encode those bits is a constant. In fact, it's actually a fairly small constant! Rearranging our above math for the index k of the smallest Leonardo number bigger than some n, if we let $k = \lceil \log_\varphi((\sqrt{5}/2)n) \rceil$, then $L(k) > n$. Now suppose that we are working on a machine whose address space is of size 2^i; then if we pick $k = \lceil \log_\varphi((\sqrt{5}/2)\, 2^{i)} \rceil = \lceil i \log_\varphi 2 + \log_\varphi(\sqrt{5} / 2) \rceil \approx \lceil$

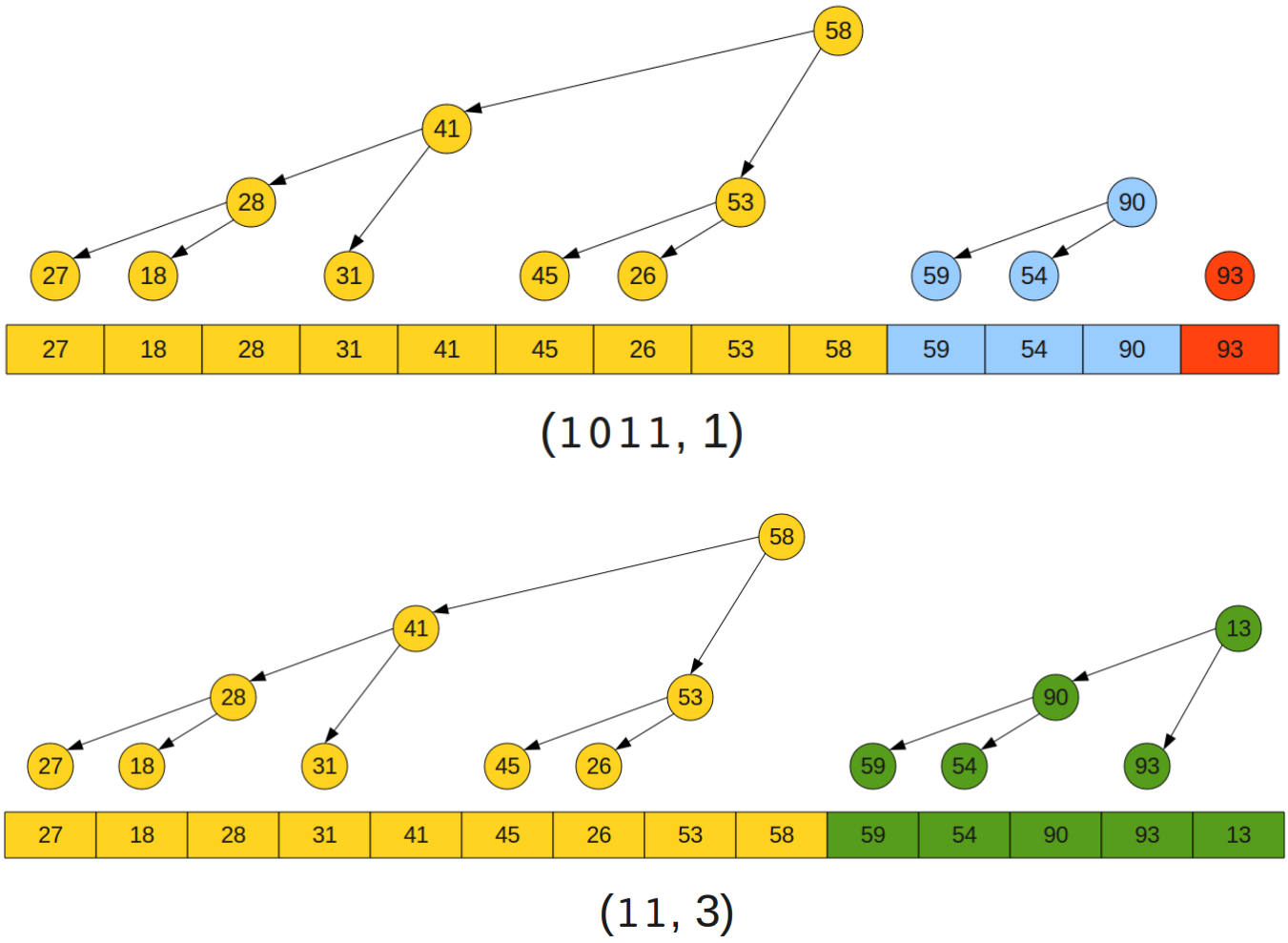
$1.44042009\ i + 0.231855848 \rceil \leq 1.7i$, we only need at most 1.7i bits to encode all the Leonardo numbers that can fit on that machine. If we assume that there there are 2^i words on the machine, each word of which has i bits, then 1.7 machine words suffices! Rounding up, we need only two machine words to store a bit for each possible Leonardo number!

But let's not get ahead of ourselves... we still have a long way to go before we'll get the bitvector working the way we want it to. In particular, we need to look at exactly how we're using the size list in our Leonardo heap to see if we can adapt the operations from an explicit list of sizes to a highly compressed bitvector.

When inserting into a Leonardo heap, we need to perform several key steps. The first step is deciding what the insertion will do - will it merge two old trees into a new tree, or insert a new tree of order one or zero? In order to answer this question, we need to know the order of the smallest tree in the heap. With the original, uncompressed size list representation, this was easily accomplished in O(1); we just looked at the first entry of the list. But with our new bitvector representation, this isn't going to work. In fact, without some sort of optimization, we might have to look at all of the O(lg n) bits to decide which one is the smallest (for example, using a linear search over the bits). Since we're doing n insertions, if we're not careful, this could take time $\Omega(n \lg n)$, eating up our O(n) best-case behavior.

One idea that might come to mind as an easy way to fix this problem would be to keep a pointer into our bitvector indicating what the smallest bit is that's currently set. This then gives us O(1) lookup of the smallest tree, fixing the above problem. For reasons that will become a bit clearer later on, we'll instead opt to use another strategy that will make the analysis easier. If we have a bitvector with a pointer into it indicating where the first non-zero value is, then you can think of the pointer as splitting the bitvector into two parts - a high-order part containing the trees in use, and a low-order part consisting solely of zeros. For example, the bitvector `101011000` gets split as `101011`||`000`, with the trees in use in the upper bits and unused trees in the lower bits. Of course, encoding these zero bits explicitly is a bit redundant; rather than encoding these zeros explicitly, we'll just keep track of how many of them there are. This means that we could encode the bitvector `101011000` as (`101011`, 3), for example. Notice that this second number can also be interpreted as the smallest tree currently in the heap, which is exactly what we set out to do. For notational purposes, I will write out these tuples using ω to mean "some bitstring" and n to mean "some number." For example, when talking about an encoding with a bitstring ending with 1, I might write (ω1, n).

Here are a few examples of implicit Leonardo heaps that use these modified bitvectors to encode their sizes:



In the first of these pictures, the trees have order 4, 2, and 1, which would yield a naive bitvector `10110`. However, since we do not allow trailing zeros, it is encoded as (`1011`, 1). The second picture has a heap with trees of order 4 and 3, whose naive bitvector would be `11000`, but is encoded as (`11`, 3) using our notation.

Now, suppose that we have a bitvector keeping track of the existing tree sizes and suppose that we want to do an insertion. In order for this step to work, we need to be able to discern which of the three cases we're in. This, fortunately, is not particularly difficult.

- Suppose that the last two trees in the heap have indices that differ by one. This means that the encoding of the tree structure must look

like (ω_{011}, n) for some ω and n, since trees of adjacent index can only appear at the end, and there's at most one pair. We can detect this case very easily by just testing the last two bits. If we find that this is the case, we can represent the merged encoding by rewriting it as (ω_1, n + 2), since we merged the last two entries together and made the smallest tree two orders bigger.

- Otherwise, if the last tree in the heap has order one, we can detect this because we'll have an encoding of the form (ω_1, 1). We need to add a tree of order zero, which can be done easily by changing this representation to (ω_{11}, 0).
- Otherwise, we need to add a tree of order one. Given (ω_1, n) for some n, we change this to be ($\omega_{100...01}$, 1) by adding enough zeros to the bitstring to correctly encode the data after saying that the smallest tree has size one.

It shouldn't be too hard to see that each of these operations can be implemented in O(1) using simple shifts and arithmetic.

After we've inserted the node into the Leonardo heap, we need to ensure that its two heap properties hold (that each heap is internally balanced and that the string of heaps is in sorted order). With an explicit size list, we could easily walk across the tops of the heaps since we could, in O(1), look up the size of each of the heaps. However, with our new bitvector approach, we can no longer claim that it takes O(1) to scan across the sizes of the heaps. In particular, suppose that our bitvector is (10101000000001, 1). Even though we've cached the size of the first tree, we can't necessarily find the next tree without repeatedly shifting the bitvector over until we encounter a 1. (Some machines might have special hardware to support this, but we can't necessarily assume this). This means that every time we try looking up a bit, it might take time O(lg n), and since there's O(lg n) bits, it seems initially like this might take $O(\lg^2 n)$ time per element, making the runtime $O(n \lg^2 n)$ in the worst case! This analysis, while correct, is not tight. It's true that any individual "shift to find the next tree" might take time O(lg n), but collectively all of the shifts we would make while inserting a single element into the heap can't take more than O(lg n) time because once we've shifted past an element, we never shift past it again.

However, there's one more thing that we need to worry about. The whole point of developing this smoothsort algorithm was to get a sorting algorithm with best-case O(n) runtime. This means that when building up the Leonardo heap for a sorted list, the runtime must be O(n). If we have to do a potentially large

number of shifts every time we try to check whether the heap is balanced, this guarantee may be compromised. Fortunately, though, we don't need to worry about this. We can always compare the root of the current heap to the root of the previous heap by skipping backwards a number of elements equal to L(k), where k is the order of the current heap. Since we cache this k, if the elements are already sorted, no shifting is necessary. The runtime guarantees are unchanged in this step.

The runtime analysis for the dequeue step is significantly more involved. Every time that we dequeue from this new Leonardo heap, we need to be able to check the size of the rightmost tree (so we know what children to expose, if any) and then need to run up to two rebalance passes. The runtime analysis for the rebalances is identical to the insertion case - each rebalance takes worst-case O(lg n) time and O(1) best-case time - but the logic required to delete the root of the rightmost tree and expose its children is a bit more complicated and the runtime analysis more involved.

There are three cases to consider during deletion:

- Case 1: The root being deleted is of a tree of order at least two. Then our encoding looks like $(\omega 1, n + 2)$ for some ω, n. Exposing the two heaps then converts this to $(\omega 011, n)$.
- Case 2: The root being deleted is of a tree of order zero. Then our encoding must look like $(\omega 11, 0)$ and we convert it to $(\omega 1, 1)$ to expose the tree of order 1 that must be right behind this one.
- Case 3: The root being deleted is of a tree of order one. Then our encoding is of the form $(\omega 100...001, 1)$. After deleting this one from the encoding, we need to shift past all of the zeros to get to the next tree root. This yields an encoding of the form $(\omega 1, 1 + n)$, where n is the number of zeros we shifted past.

In the original algorithm all of these steps ran in O(1). Now, the first two steps clearly run in O(1), but that last step takes a variable amount of time; in particular, it needs to perform one shift for each of the zeros before the next tree. Since there are O(lg n) bits in the representation, initially it might seem like this would mean that the best-case runtime for this algorithm is $\Omega(n \lg n)$, but it turns out that this is not the case. If we use an amortized analysis, we can show that each operation runs in amortized O(1), giving a total runtime of O(n).

This amortized analysis is a bit tricky because the structure of encodings changes so wildly during each of these steps, especially during deletion of a tree of order 1. To prove the time bound, we'll therefore adopt an alternate

approach actually suggested by Dijkstra in his original paper. We know that there will be a total of n deletions from the heap, and each one of those deletions is essentially an insertion step run backwards. Thus if we can bound the total number of shifts done as we insert all n elements, we have a bound for the total number of shifts done during deletion.

Let's define a potential function on the encoding of our heap sizes as $\Phi(\omega, n) = n$. We will count the number of one-position shifts performed on the encoding during insertion, even though in some cases during insertion we can group these shifts together into one bulk shift operation. The reason for not batching shifts together is that it's unclear whether we'll be able to perform those same shifts in reverse during the delete step (in fact, we can't, or we wouldn't need this analysis!)

- Case one: The last two trees have adjacent order, and we transform the encoding from (ω_{011}, n) to $(\omega 1, n + 2)$. To do this transformation, we need to do two shifts to drop off the last two ones from the representation. Moreover, $\Delta\Phi$ in this case is 2, and so the amortized cost of this step is four.
- Case two: The last tree has order one, and we transform the encoding from $(\omega_1, 1)$ to $(\omega_{11}, 0)$. This requires one shift to make space for the new 1 bit, and $\Delta\Phi = -1$ for an amortized cost of zero.
- Case three: The last two trees do not have adjacent order, nor does the last tree have order one. Then we transform (ω_1, n) into $(\omega_{1000\ldots001}, 1)$, where there are n zeros inserted. This requires n shifts (n - 1 for the zeros, and 1 for the one bit) and we have $\Delta\Phi = 1 - n$ for an amortized cost of 1 shift.

From this we see that in all three cases, the amortized cost of an insertion is O(1), and by symmetry the amortized cost of a deletion is O(1) as well. This guarantees that our time bound is as it was before, at least in an amortized sense.

At this point, we have just proven that if we're willing to use a crazy encoding scheme for our Leonardo heap size list, it's possible to get smoothsort working in O(1) memory. We have just developed an adaptive heapsort variant with O(1) memory usage!

A Final Version of Smoothsort

Having done all the research necessary to figure out exactly how Dijkstra's mysterious smoothsort algorithm worked, I've put together a smoothsort

implementation of my own. It uses O(1) memory via the encoding scheme described above. It also contains a few minor optimizations based on Dijkstra's original paper. I will probably update this writeup to explain them when I recover from fully-detailing the O(1) memory version. :-)

Concluding Remarks

This minor quest of mine to understand smoothsort ended up being one of the most interesting research projects I've undertaken. I learned a fair amount about data structures and algorithms in the process (in particular, a much more general framework for heapsort than I had known before). To the best of my knowledge, no one has previously described the Leonardo heap structure detailed on this page explicitly as a heap data structure, though undoubtedly Dijkstra knew of them when putting together smoothsort. To this day I have no idea idea how Dijkstra came up with this algorithm. There are some many unintuitive insights necessary to put the whole thing together, and it has taken me the course of two months to completely and fully appreciate all the complexities of the implementation. In fact, my first analysis of the algorithm completely missed the point of the size list, and ended up not correctly using O(1) space! Moreover, in the course of writing all of this up, I've cemented my understanding of the transdichotomous machine model and of amortized analysis.

I hope that you found this intro to smoothsort and Leonardo heaps interesting and accessible. I hope that this site increases the profile of this particular sort, since prior to reading up on it myself I had never encountered anyone who had even heard the name of the algorithm before. Ideally, this writeup will make it possible to pick up smoothsort without spending several days of effort doing so.

Feel free to email me if you have any comments or questions!