Fibonacci Sort

Lucilla Blessing

Abstract

This paper presents a novel merge-based sorting algorithm named "Fibonacci sort", which, while not particularly efficient relative to standard merge sort, has a remarkable connection to the Fibonacci sequence despite never explicitly calculating Fibonacci numbers. It also generalizes to an abstract description of general merge sorts through a sequence of eliminating merge stops, where a description of Fibonacci sort results in an astounding recursive integer sequence related to the Fibonacci numbers.

0 Introduction

The following algorithm is curiously named "Fibonacci sort":

```
function fibonacciSort($s$: array) is
    $n := s$.length
    let $v$ be a new list
    initialize $v$ to $[-1, 0, 1]$
    $lo, mi, hi :=$ last three values of $v$
    while $\neg(mi = 0 \ \wedge \ hi = n)$ do
        if $(hi = n \ \vee \ (mi \neq 0 \ \wedge \ 2 \cdot (hi - mi) \geq mi - lo))$ then
            merge($s$, $lo$, $mi$, $hi$)
            remove next-to-last value of $v$
        else
            append $hi + 1$ to the end of $v$
        end
        $lo, mi, hi :=$ last three values of $v$
    end
end
```

Why that name? The algorithm doesn't compute the Fibonacci sequence in any obvious way, nor does it seem to sort in a way dictated by the Fibonacci sequence somehow. And yet the Fibonacci numbers feature prominently in its analysis, specifically in the sequence of merges that it performs; and it also produces the Zeckendorf representation of the length of its input array in the process. Let's see.

1 Preliminaries

Actually, to make analysis easier, we will start by considering an algorithm with different flow control. Equivalence of the two algorithms (in the sense that they both perform the same sequence of steps and in the same order) will be discussed at the end of this paper.

```
function fibonacciSort(s: array) is
    $n := s$.length
    let $v$ be a new list
    initialize $v$ to $[-1, 0, 1]$
    $lo, mi, hi :=$ last three values of $v$
    while $hi \neq n$ do
        append $hi + 1$ to the end of $v$
        update $lo, mi, hi$
        while $(mi \neq 0 \ \wedge \ 2 \cdot (hi - mi) \geq mi - lo)$ do
            merge($s$, $lo$, $mi$, $hi$)
            remove next-to-last value of $v$
            update $lo, mi, hi$
        end
    end
    while $mi \neq 0$ do
        merge($s$, $lo$, $mi$, $hi$)
        remove next-to-last value of $v$
        update $lo, mi, hi$
    end
end
```

Here the `merge` procedure is defined as follows:

```
function merge($s$: array; $lo$, $mi$, $hi$: $\mathbb{N}$) is
    let $t$ be a new array of length $hi - lo$
    $\texttt{i}, \texttt{j} := lo, mi$
    for k in $0 \leq \texttt{k} < hi - lo$ do
        if $(\texttt{i} < mi \ \wedge \ (\texttt{j} = hi \ \vee \ s[\texttt{i}] \leq s[\texttt{j}]))$ then
            $t[\texttt{k}] := s[\texttt{i}]$;   i++
        else
            $t[\texttt{k}] := s[\texttt{j}]$;   j++
        end
    end
    for k in $0 \leq \texttt{k} < hi - lo$ do
        $s[lo + \texttt{k}] := t[\texttt{k}]$
    end
    delete the array $t$
end
```

Fibonacci sort is a *merge sort*: it sorts an array by *merging* progressively longer runs (sub-arrays which are already sorted) into a single run that contains the elements of both. The `merge` procedure, which takes an array s and three so-called 'merge stops' lo, mi, hi ($lo < mi < hi$), assumes that the sub-arrays $s[lo \mathrel{..} mi]$ and $s[mi \mathrel{..} hi]$ are already sorted, and merges them into a single sorted sub-array, writing the result back to $s[lo \mathrel{..} hi]$. (Here $lo \mathrel{..} hi$ means the range of indices `k` for which $lo \leq \texttt{k} < hi$, i.e. left inclusive, right exclusive. In particular, $lo \mathrel{..} mi$ and $mi \mathrel{..} hi$ are adjacent and disjoint.)

Since 1-element sub-arrays are always trivially sorted, many merge sorts (Fibonacci sort included) start by assuming that *only* those sub-arrays are already sorted (this assumption cannot be made more optimistic without looking at the array, since it *does* hold for an array of pairwise unequal elements in descending order), and the fact that a merge can be done in $\mathcal{O}(hi - lo)$ time is the fundamental fact ensuring that merge sorts can have $\mathcal{O}(n \log n)$ complexity. (We take for granted that `merge` as defined above really does merge, and does so in $\mathcal{O}(hi - lo)$ steps; it's a standard implementation.)

In Fibonacci sort, the list v controls the locations of 'merge stops' – boundary points between sub-arrays that are known to be sorted. The leading entry -1 only serves to ensure that v always has a minimal length of 3, so that all of lo, mi, hi are always defined. Its exact value doesn't matter, but throughout this paper we will stick to the assumption that it is equal to -1; it will serve as a kind of 'signaling value' indicating that v has its minimal length.

The general idea of the algorithm is that v saves merge stops, and the algorithm collects new merge stops from the left towards the right side of the array. As long as the end of the array hasn't been reached, it adds a new merge stop to the end representing a one-element sub-array (since those are guaranteed to be sorted) and then tries to merge as many as possible 'conditionally'. When the end has been reached, it merges all the remaining sub-arrays together 'unconditionally' until the entire array is sorted.

The essential point about Fibonacci sort is the condition "$2 \cdot (hi - mi) \geq mi - lo$" necessary for a 'conditional merge' to happen, saying that the right sub-array must be at least half the length of the left sub-array. This condition is weaker than for binary merge sort, which always only merges two sub-arrays of exactly the same length. But it also isn't relaxed to no size condition at all; if it were, then every new singleton sub-array would be merged instantly, and the algorithm would essentially work like insertion sort, with $\mathcal{O}(n^2)$ complexity. The condition for Fibonacci sort strikes a sort of balance between these two extremes.

We will begin analysis of this algorithm with some elementary properties as well as verifying that all its steps are always well-defined. Firstly, we begin with a simple statement ensuring that v always has at least three elements (and thus lo, mi, hi are never `nil`), and also exposing some basic properties of v. The length of v will be denoted as $|v|$.

Lemma

$|v| \geq 3$ always holds, and the two leading elements of v are always $v[0] = -1$ and $v[1] = 0$. Moreover, (except for the first pair of entries,) v is always strictly increasing, meaning $v[\mathtt{i}+1] > v[\mathtt{i}]$ for all (valid) $\mathtt{i} \in \mathbb{N}, \mathtt{i} \geq 1$.

Proof: We will use "induction over steps": verifying that the initial value of v satisfies the desired conditions, and that any modifications made to v over the course of the program leave them still satisfied. (This technique is common to many proofs in this paper.)

- The initial value of v, namely $[-1, 0, 1]$, has length 3, has -1 and 0 as its two leading elements, and $[0, 1]$ is strictly increasing.

- When an entry is added to the end of v, this entry has the value $hi+1$, where hi is the former last entry of v. Therefore, after the addition, $hi+1$ is the new last entry and hi is the next-to-last entry, and they satisfy $hi+1 > hi$, so v remains strictly increasing. If the former length of v was at least 3, then the current length also is. Lastly, the first two elements are not modified.

- A merge can only happen if $mi \neq 0$. If $|v| = 3$, then $mi = v[1] = 0$, so a merge is impossible; therefore, a merge can only happen if $|v| \geq 4$, in which case $mi = v[\mathtt{i}] > v[1] = 0$ for some $\mathtt{i} \geq 2$. Thus after the merge, $|v| \geq 3$. mi is the element that is deleted, but it's not one of the first two elements, since its index is at least 2. The sequence remains strictly increasing after the removal because of the transitivity of $<$.

■

Another important result is that at the beginning of each iteration of the first **while** loop, the triplets in v (collections of three adjacent elements, omitting the initial index $v[0]$) are "non-squeezy", meaning that they *don't* satisfy the merge condition.

Lemma

At the beginning of each iteration of the first **while** loop, the triplets of v are all non-squeezy, meaning $2 \cdot (v[\mathtt{i}+2] - v[\mathtt{i}+1]) < v[\mathtt{i}+1] - v[\mathtt{i}]$ for all (valid) $\mathtt{i} \in \mathbb{N}, \mathtt{i} \geq 1$. More generally, throughout the entirety of the first **while** loop, there is always at most one squeezy triplet in v, at the very end; so that if it *does* hold for some $\mathtt{i} \geq 1$ that $2 \cdot (v[\mathtt{i}+2] - v[\mathtt{i}+1]) \geq v[\mathtt{i}+1] - v[\mathtt{i}]$, then $\mathtt{i} = |v| - 3$.

Proof: "Induction over steps":

- For the initial value of v, the statement is vacuously true, as there is only one non-initial gap.

- When an entry is added to the end of v, all the previous triplets were not squeezy, so now there is at most one squeezy triplet, at the end. We then begin an inner **while** loop of merging for as long as $|v| \geq 4$ and the last triplet is squeezy. Each merge can be thought of as removing the last three elements of v and then reïnserting the first and last of those three back into v. This eliminates the last squeezy triplet, but it potentially introduces a new one – though, again, only at most one. This continues until the inner **while** loop terminates when a merge is not possible, either because $|v| = 3$ or there is no more squeezy triplet at the end; in both cases, all the remaining triplets are now non-squeezy.

■

Finally, during the first **while** loop, the "gaps" of v (differences between adjacent elements, again ignoring the first pair $v[1] - v[0]$) are *decreasing* in size.

Lemma

Throughout the first **while** loop, the gaps of v are always decreasing, meaning $v[\mathtt{i}+2] - v[\mathtt{i}+1] \leq v[\mathtt{i}+1] - v[\mathtt{i}]$ for all (valid) $\mathtt{i} \in \mathbb{N}, \mathtt{i} \geq 1$. Moreover, the only case where the inequality is not strict is if both sides have the value 1.

Proof: "Induction over steps":

- For the initial value of v, the statement is vacuously true.
- When an entry is added to the end of v, it forms a gap of size 1. Because v itself is strictly increasing, the previous gap had to have had a size of at least 1, thus the gaps of v remain (not necessarily strictly) decreasing.
- When a merge happens, if $|v| = 4$, then after the merge it is 3 and we have a vacuous truth (and if $|v| = 3$, then a merge cannot even take place). Thus we may assume $|v| \geq 5$. Thus v has at least four trailing elements unequal to -1, and they satisfy the decreasing property of gaps. Symbolically:

  $$v = [\ldots, a, b, c, d] \quad \text{with} \quad b - a \geq c - b \geq d - c.$$

  In the first **while** loop, in order for a merge to happen, the additional condition $2 \cdot (d - c) \geq c - b$ must be met. Moreover, $2 \cdot (c - b) \geq b - a$ must *not* hold, since no triplets apart from possibly the final one may be squeezy. Combining the inequality $2 \cdot (c - b) < b - a$ with $c - b \geq d - c$ yields

  $$\begin{array}{ccccc} & & 2 \cdot (c-b) & < & b-a \\ & & 2 \cdot c - 2 \cdot b & < & b - a \\ b + d & \leq & 2 \cdot c & < & 3 \cdot b - a \\ b + d + a & \leq & 2 \cdot c + a & < & 3 \cdot b \\ d + a & & & < & 2 \cdot b \\ d - b & & & < & b - a, \end{array}$$

  so that the gaps of v remain decreasing after c is removed, ensuring even strict inequality.

■

2 The Fibonacci in the sort

We can now analyze the connection between Fibonacci sort and the Fibonacci sequence, the entire reason behind its name. From here onwards, let Fib be the Fibonacci sequence; that is, $\mathsf{Fib}[0] = 0$, $\mathsf{Fib}[1] = 1$, and for all $\mathtt{i} \geq 2$, $\mathsf{Fib}[\mathtt{i}] = \mathsf{Fib}[\mathtt{i} - 1] + \mathsf{Fib}[\mathtt{i} - 2]$.

Fib obeys a kind of "squeeze property": namely, for all $\mathtt{i} \geq 0$, it holds both that $\mathsf{Fib}[\mathtt{i}] \leq \mathsf{Fib}[\mathtt{i}+1]$ and that $2 \cdot \mathsf{Fib}[\mathtt{i}] \geq \mathsf{Fib}[\mathtt{i}+1]$, where the former is true even with strict inequality for $\mathtt{i} \geq 2$, and the latter is true with strict inequality for $\mathtt{i} \geq 3$. (The proof is simply that the latter inequality is equivalent to $\mathsf{Fib}[\mathtt{i} - 1] \leq \mathsf{Fib}[\mathtt{i}]$.)

The key to the connection between Fibonacci numbers and Fibonacci sort is a kind of converse of this squeeze property: namely, if $\mathsf{Fib}[\mathtt{i}]$ and $\mathsf{Fib}[\mathtt{j}]$ are Fibonacci numbers that satisfy $\mathsf{Fib}[\mathtt{i}] < \mathsf{Fib}[\mathtt{j}]$ and $2 \cdot \mathsf{Fib}[\mathtt{i}] \geq \mathsf{Fib}[\mathtt{j}]$, then they are adjacent.

Lemma

Let $\mathtt{i} \geq 2$, $\mathtt{j} \geq 2$. If $\mathsf{Fib}[\mathtt{i}] < \mathsf{Fib}[\mathtt{j}]$ and $2 \cdot \mathsf{Fib}[\mathtt{i}] \geq \mathsf{Fib}[\mathtt{j}]$, then $\mathtt{j} = \mathtt{i} + 1$.

Proof:

- Starting from index 1, Fib has only positive integer entries, since $\mathsf{Fib}[1] = 1$, $\mathsf{Fib}[2] = 1$, and all further entries are sums of positive integers, thus also positive integers.
- Starting from index 2, Fib is strictly increasing (meaning $\mathsf{Fib}[\mathtt{i} + 1] > \mathsf{Fib}[\mathtt{i}]$ for all $\mathtt{i} \geq 2$), since $\mathsf{Fib}[\mathtt{i} + 1] - \mathsf{Fib}[\mathtt{i}] = \mathsf{Fib}[\mathtt{i} - 1]$, which is always a positive integer for $\mathtt{i} \geq 2$.
- We now show that the squeeze property does *not* hold for a pair of Fibonacci numbers two indices apart. Let $\mathtt{i} \geq 2, \mathtt{j} \geq 2$. If $\mathsf{Fib}[\mathtt{i}] < \mathsf{Fib}[\mathtt{j}]$, then by the strict increasing property, we have $\mathtt{i} < \mathtt{j}$. Suppose that $\mathtt{j} = \mathtt{i} + 2$, then $\mathsf{Fib}[\mathtt{i}+2] = \mathsf{Fib}[\mathtt{i}+1] + \mathsf{Fib}[\mathtt{i}] > \mathsf{Fib}[\mathtt{i}] + \mathsf{Fib}[\mathtt{i}]$ and therefore $2 \cdot \mathsf{Fib}[\mathtt{i}] < \mathsf{Fib}[\mathtt{j}]$, so the squeeze property does not hold. Similarly, if $\mathtt{j} = \mathtt{i} + n$ for even higher $n > 2$, $\mathsf{Fib}[\mathtt{j}]$ will be even bigger. Therefore, a necessary condition for the squeeze property to hold is that $\mathtt{j} = \mathtt{i} + 1$, which completes the proof.

$\blacksquare$

A crucial corollary is that if f_0 and f_1 are (any) two Fibonacci numbers satisfying the squeeze property, then $f_0 + f_1$ is also a Fibonacci number. Indeed, it means f_0 and f_1 are adjacent Fibonacci numbers, and the sum of two adjacent Fibonacci numbers is itself a Fibonacci number.

Now we're finally able to make the connection. As an appetizer, let's first look at the sequence of states of v when sorting an array of length 8.

v	gaps of v
`[-1, 0, 1]`	`[1, 1]`
`[-1, 0, 1, 2]`	`[1, 1, 1]`
`[-1, 0, 2]`	`[1, 2]`
`[-1, 0, 2, 3]`	`[1, 2, 1]`
`[-1, 0, 3]`	`[1, 3]`
`[-1, 0, 3, 4]`	`[1, 3, 1]`
`[-1, 0, 3, 4, 5]`	`[1, 3, 1, 1]`
`[-1, 0, 3, 5]`	`[1, 3, 2]`
`[-1, 0, 5]`	`[1, 5]`
`[-1, 0, 5, 6]`	`[1, 5, 1]`
`[-1, 0, 5, 6, 7]`	`[1, 5, 1, 1]`
`[-1, 0, 5, 7]`	`[1, 5, 2]`
`[-1, 0, 5, 7, 8]`	`[1, 5, 2, 1]`
`[-1, 0, 5, 8]`	`[1, 5, 3]`
`[-1, 0, 8]`	`[1, 8]`

Perhaps the clearest observation is that whenever $|v| = 3$, the three terms of v are -1, 0, and a Fibonacci number; and that whenever a merge happens, the last three terms of v are of the form $[c, c + f_0, c + f_1]$, where f_0 and f_1 are Fibonacci numbers. In fact, an even more surprising fact is true: *all* the non-initial gaps in v are *always* Fibonacci numbers. This theorem is actually the entry point to all the other statements about the presence of Fibonacci numbers in v, so let's start here.

Theorem gaps of v are Fibonacci

During the first **while** loop, for all $\texttt{i} \geq 1$, $v[\texttt{i} + 1] - v[\texttt{i}]$ is a Fibonacci number.

Proof: "Induction over steps":

- The initial state $[-1, 0, 1]$ satisfies $1 - 0 = 1 \in \mathsf{Fib}$.
- Whenever an item is added to the end of v, it's equal to the value of the *previous* last item plus one, so the new gap is the number 1, which, again, is in Fib.
- When a merge happens, call the last three entries lo, mi, hi. By the induction hypothesis, $mi - lo =: a \in \mathsf{Fib}$ and $hi - mi =: b \in \mathsf{Fib}$. Moreover, $2 \cdot b \geq a$, because we've passed the check for a 'conditional merge', and either $b < a$ or $b = a = 1$. In both cases, $a + b$ is a Fibonacci number. But this is none other than $hi - lo$, which is the gap that remains after mi has been removed.

■

From this theorem, we can deduce two of our observations as immediate corollaries.

Corollary

Whenever a merge happens, the last three terms of v are of the form $[c, c+f_0, c+f_1]$, where $f_0, f_1 \in \mathsf{Fib}$.

Proof: It follows from the proof of the theorem above that $hi - mi =: b$ and $mi - lo =: a$ are, in particular, *consecutive* Fibonacci numbers; thus $mi - lo = a$ and $hi - lo = a + b$ are also (consecutive) Fibonacci numbers. ■

Corollary

When $|v| = 3$, then there exists $f \in \mathsf{Fib}$ such that $v = [-1, 0, f]$.

Proof: "Induction over steps":
The base case $[-1, 0, 1]$ satisfies the claim because $1 \in \mathsf{Fib}$.
Adding a new term increases $|v|$ to at least 4.
When a merge happens and $|v|$ becomes 3, it must have come from $[-1, 0, a, b]$ because the first two terms are always -1 and 0. From the previous corollary, a and b are (consecutive) Fibonacci numbers. In particular, $b \in \mathsf{Fib}$, so after the merge, $v = [-1, 0, b]$ satisfies the claim. ■

Finally, something astonishing happens if we examine what happens right after the first **while** loop is complete, but before the second **while** loop starts: It turns out that in that moment, the gaps of v encode a representation of the Zeckendorf decomposition of the length of the list, n.

Theorem

After the end of the first **while** loop, the sequence of gaps of v, with the leading 1 removed, is the Zeckendorf representation of n (the length of the array being sorted) in descending order.

Proof:

- We know all the gaps of v are Fibonacci numbers and that they are decreasing.
- Every pair of consecutive entries in the gaps of v after the first **while** loop cannot consist of consecutive Fibonacci numbers (let alone two of the same number), since if they were, then the corresponding triplet would be squeezy. In particular, the entries are pairwise distinct.
- The sum of their values is a telescoping sum, giving $v[|v|-1]-v[1] = n-0 = n$, where $v[|v| - 1] = n$ holds because we've reached the end of the first **while** loop.
- Since the Zeckendorf representation of a positive integer is unique, the proof is complete.

■

3 Properties of the sort

Now that we've explained the "Fibonacci" part of the name "Fibonacci sort", it's time to move on to the "sort" part. We'll prove that this algorithm actually *sorts* at all, and moreover, that it does so in $\mathcal{O}(n \log n)$ time.

First we need a lemma that verifies that v really does what it's meant to do:

Lemma

The entries of v, with $v[0] = -1$ deleted, encode the sub-arrays of the list s which are guaranteed to be sorted. In particular, for all (valid) $\mathtt{i} \in \mathbb{N}, \mathtt{i} \geq 1$, the sub-array $s\big[v[\mathtt{i}] \mathrel{..} v[\mathtt{i}+1]\big]$ is sorted.

Proof: "Induction over steps" once more:

- In $[-1, 0, 1]$, the only relevant gap is $(0, 1)$, which corresponds to the trivially true claim that the (singleton) sub-array $s[0 \mathrel{..} 1]$ is sorted.
- When an item is added to the end of v, its value is one more than the previously last entry, which corresponds to the trivially true claim that some singleton sub-array $s[z \mathrel{..} z+1]$ is sorted.
- When a merge is performed, denote the last three entries of v by lo, mi, hi. By the induction hypothesis, $s[lo \mathrel{..} mi]$ and $s[mi \mathrel{..} hi]$ are sorted. We perform $\mathtt{merge}(s, lo, mi, hi)$. Thus, after the merge, $s[lo \mathrel{..} hi]$ is sorted, which is exactly what v encodes when mi is removed.

■

From this, it's only a quick corollary to prove that Fibonacci sort does indeed sort:

Theorem Fibonacci sort is a sort

At the end of the algorithm, s is sorted.

Proof:

- At the end of the first **while** loop, the final element of v is n.
- At the end of the second **while** loop, $v = [-1, 0, n]$.
- It follows that $s[0 \mathrel{..} n]$ is sorted then. But $s[0 \mathrel{..} n] = s$.

■

Still, even if Fibonacci sort does sort, so does bogosort – the tricky part is to sort *fast*. Fibonacci sort does indeed achieve the optimal asymptotic complexity of $\mathcal{O}(n \log n)$, the same as ordinary merge sort – as we'll see next.

We'll start with a lemma about the possible intersections of integer ranges caused by merges.

Lemma

If two merges (lo, mi, hi) and (lo', mi', hi') happen at any time over the course of the first **while** loop, then the ranges lo .. hi and lo' .. hi' are either disjoint or one is a subset of the other.

Proof: We know $lo < hi$ and $lo' < hi'$. We enumerate all possible cases:

`     l--h` `L--H` ✓	`    l---h` `L---H` ✓	`   l----h` `L----H` ✗	`  l-----h` `L-------H` ✓	`  l---h` `L-------H` ✓
`l-------h` `L---H` ✓		`l-------h` `L-------H` ✓		`l---h` `L-------H` ✓
`l-------h` `  L---H` ✓	`l-------h` `  L-----H` ✓	`l----h` `   L----H` ✗	`l---h` `    L---H` ✓	`l--h` `     L--H` ✓

where $(\texttt{l}, \texttt{h}) := (lo, hi)$ and $(\texttt{L}, \texttt{H}) := (lo', hi')$.

From the enumeration of cases, the claim is true in all cases marked with ✓and false in all cases marked with ✗. It follows that if the claim is false, then

$$\text{either} \quad lo' < lo < hi' < hi \quad \text{or} \quad lo < lo' < hi < hi'.$$

Assume without loss of generality that (lo, mi, hi) is the merge that happened first. Then, no more entry x with $lo < x < hi$ can ever occur in v again, since all items before lo are already less than lo and all items after hi will be greater than hi. Therefore, (lo', mi', hi') can never happen, since in that case $lo < lo' < hi$ or $lo < hi' < hi$ would follow; a contradiction. ■

This lemma lets us conclude that any two merges of the same size must be disjoint:

Corollary

If two distinct merges (lo, mi, hi) and (lo', mi', hi') happen at any time over the course of the first **while** loop, and $hi - lo = hi' - lo'$, then lo .. hi and lo' .. hi' are disjoint.

Proof: The lemma above tells us that they must be either disjoint or one must be a subset of the other. But in the second case, $lo = lo'$ and $hi = hi'$. If two merges with the same start- and end-point happen at different times, then there would have to be a mi' satisfying $lo < mi' < hi$ after (lo, mi, hi) with $lo < mi < hi$ had already taken place; this contradicts what we have proven in the lemma above. ■

This corollary implies that the number of merges of a *fixed* size l during the first **while** loop is bounded from above by $\lfloor n/l \rfloor$, where n is the length of the list s.

This allows us to prove that the first **while** loop takes no more than $\mathcal{O}(n \log n)$.

Theorem first while loop is $\mathcal{O}(n \log n)$

Assuming the time spent on steps other than merging is negligible, the first **while** loop executes in $\mathcal{O}(n \log n)$ steps, where n is the length of the list s.

Proof:

- All merges are of size $hi - lo$, where (lo, mi, hi) are the last three entries of v at the instant the merge is happening. This number, being equal to $(mi - lo) + (hi - mi)$, the sum of two consecutive Fibonacci numbers, is a Fibonacci number.

- Since the number of merges of size l is bounded from above by $\lfloor n/l \rfloor$, and every merge of (lo, mi, hi) takes $\mathcal{O}(hi - lo)$ steps, the *total* cost of all merges of size l is bounded by $\mathcal{O}(\frac{n}{l} \cdot l) = \mathcal{O}(n)$.

- Therefore, the total cost of *all* merges (of any size) is bounded by the sum over all possible merge sizes of the total cost of merges of that size, which is

$$\begin{aligned} & \mathcal{O}\big((\#\text{ possible merge sizes}) \cdot n\big) \\ = {} & \mathcal{O}\big(\underbrace{(\#\text{ Fibonacci numbers less than } n)}_{\Theta(\log n)} \cdot n\big) \\ = {} & \mathcal{O}(n \log n), \end{aligned}$$

  since $\mathsf{Fib}[k] \sim \varphi^k/\sqrt{5} \in \Theta(\varphi^k)$ and therefore $k \in \Theta(\log_\varphi(\mathsf{Fib}[k]))$, where φ is the golden ratio.

■

This implies that the algorithm takes $\mathcal{O}(n \log n)$ steps if the length of the input list is a Fibonacci number, since in that case, its Zeckendorf representation consists of just that number itself. To prove that the algorithm runs in $\mathcal{O}(n \log n)$ time for *any* input, we find an upper bound for the steps necessary in the second **while** loop.

Theorem second while loop is $\mathcal{O}(n \log n)$

Assuming the time spent on steps other than merging is negligible, the second **while** loop executes in $\mathcal{O}(n \log n)$ steps.

Proof: The number of merges performed in the second **while** loop equals the number of Zeckendorf components minus 1 (since every gap corresponds to one of them; each merge reduces $|v|$ by one; we terminate when there's only one gap left). The cost of each merge is bounded by at most $\mathcal{O}(n)$. The number of Zeckendorf components is bounded by $\frac{1}{2}$ times the number of Fibonacci numbers not exceeding n, which is $\Theta(\log n)$. Together, we obtain $\mathcal{O}(n \log n)$ steps. ■

Therefore the entire algorithm runs in $\mathcal{O}(n \log n)$ time, as desired.

All the results leading up to this point were proven about the alternative version of the Fibonacci sort algorithm. We conclude this section with a remark on the equivalence of the two versions. For $n = 0$ and $n = 1$ both algorithms do nothing, as the lists are already sorted. Assuming $n \geq 2$, both algorithms proceed through the "progress states" in the leftmost column, and can be shown to do equivalent things in each case:

	original	**alternative**
start	$v = [-1, 0, 1, 2]$, beginning of **while**	$v = [-1, 0, 1, 2]$, beginning of inner **while**
$hi < n$; conditional merge possible	merge; go to beginning of **while**	merge; go to beginning of inner **while**
$hi < n$; conditional merge impossible	add 1; go to beginning of **while**	add 1; go to beginning of inner **while**
$hi = n$; conditional merge possible	merge; go to beginning of **while**	merge; go to beginning of inner **while**
$hi = n$; conditional merge impossible; unconditional merge possible	merge; go to beginning of **while**	leave first **while** if in it; unconditional merge; go to beginning of second **while**
$hi = n$; no merge possible ($mi = 0$)	stop	stop

4 Generalized mergesorts, the fairy sequence

The strategy that Fibonacci sort uses to keep track of which runs are sorted gives rise to an idea: what if we *start* a merge sort from a list of currently sorted runs, and simply merge by continuously removing the middle of a triplet of 'merge stops' until no more sub-arrays are left to merge?

In other words, we want to have an analogue of our list v, which will track the currently sorted runs using a list of 'run barriers'. Unlike in Fibonacci sort, however, we'll want it to start out by *already* containing information about all the sorted runs, so that the only changes ever made to it are *removing* elements:

- The only sub-arrays guaranteed to be initially sorted are the singletons, so we want v to start off as $[0, \dots, n]$ (both 0 and n are *inclusive*, so that v initially has length $n + 1$).
- Each merge eliminates one 'run barrier', turning some $[\dots, lo, mi, hi, \dots]$ into $[\dots, lo, hi, \dots]$. Thus, each merge removes one non-initial, non-final element.
- We terminate when v becomes $[0, n]$, since we know this means the entire list is sorted.

A *generalized mergesort* can then be described simply by specifying in what order the run barriers should be eliminated, i.e. an iterator procedure that produces all numbers $\texttt{i} \in \mathbb{N}$ with $0 < \texttt{i} < n$ in some order.

Some examples from known sorting algorithms:

- Bottom-up (binary) mergesort, $n = 2^k$:

$$1, 3, 5, \dots, n-1,\ 2, 6, 10, \dots, n-2,\ 4, 12, 20, \dots, n-4,\ \dots, n/2$$

- Left-to-right (binary) mergesort, $n = 2^k$:

$$1, 3, 2, 5, 7, 6, 4, 9, 11, 10, 13, 15, 14, 12, 8, \dots \qquad \text{(A108918 in the OEIS)}$$

- Insertion sort:

$$1, 2, 3, 4, \dots, n-1$$

A remarkable novel sequence appears when we consider the sequence of merges that Fibonacci sort performs (for $n \in \mathsf{Fib}$, or, alternatively, for $n \to \infty$, which is what we'll consider it as being from now on). Here it is:

$$1, 2, 4, 3, 6, 7, 5, 9, 10, 12, 11, 8, 14, 15, 17, 16, 19, 20, 18, 13, \dots$$

As of Tau Day 2024, there is no OEIS entry for this sequence.

Let's call this sequence the *fairy sequence* and denote it by Fai. We already know how we can compute Fai to any desired length n: just pick $n_0 \geq n, n_0 \in \mathsf{Fib}$ and sort an array of length n_0 and keep track of where you merge. But this sequence also hides a marvelous recursive pattern, and can therefore also be computed from its own previous entries. The claim is that the following algorithm computes Fai:

```
function fairySequence() is
    let Fai be a new empty array
    i := 1
    repeat forever
        for j in $0 \leq j < \mathsf{Fib}[i] - 1$ do
            append $\mathsf{Fai}[j] + \mathsf{Fib}[i + 1]$ to the end of Fai
        end
        append $\mathsf{Fib}[i + 1]$ to the end of Fai
        i++
    end
end
```

Note that $\mathsf{Fib}[0] = 0$ and $\mathsf{Fib}[1] = \mathsf{Fib}[2] = 1$, so the first two iterations of the **repeat** loop have a vacuous **for** loop.

The main idea why this recursive algorithm works is because the merges in Fibonacci sort are themselves highly recursive, as the next lemma shows. If v and w are arrays, we write v .. w for their *concatenation.*

Lemma

Whenever $v = [-1, 0, \mathsf{Fib}[\mathtt{i}]]$ for some $\mathtt{i} \in \mathbb{N}, \mathtt{i} \geq 3$ $(\mathsf{Fib}[\mathtt{i}] \geq 2)$, meaning that $v' :=$ the array of gaps of v is $[1, \mathsf{Fib}[\mathtt{i}]]$, then v' will go on to evolve as v' .. w_0, v' .. w_1, $\ldots$, v' .. $[\mathsf{Fib}[\mathtt{i} - 1]], [1, \mathsf{Fib}[\mathtt{i} + 1]]$, where the w_n are the states of v' from the beginning of Fibonacci sort (that is, $[1, 1]$) until the point where $[1, \mathsf{Fib}[\mathtt{i} - 1]]$, with the leading 1 removed.

The claim of this lemma is a bit of a mouthful, so let's demonstrate it with an example. Say $v' = [1, 5]$. The states of v' starting from $[1, 1]$ up to $[1, 3]$ unfold as follows:

$$[1, 1], \quad [1, 1, 1], \quad [1, 2], \quad [1, 2, 1], \quad [1, 3].$$

With the leading 1s removed, this becomes

$$[1], \quad [1, 1], \quad [2], \quad [2, 1], \quad [3].$$

Thus the lemma says that the next states of v' after $[1, 5]$ will be the concatenations of $[1, 5]$ with the states above; and sure enough, v' goes on to evolve as

$$[1, 5, 1], \quad [1, 5, 1, 1], \quad [1, 5, 2], \quad [1, 5, 2, 1], \quad [1, 5, 3], \quad [1, 8].$$

This lemma includes the claim that v' will reach the state $[1, \mathsf{Fib}[\mathtt{i} - 1]]$ at all; but a proof by induction over $\mathtt{i}$ will handle that just fine. It also implies that if v' has reached the state $[1, \mathsf{Fib}[\mathtt{i}]]$, then it will reach the state $[1, \mathsf{Fib}[\mathtt{i} + 1]]$.

Proof: Induction over i.

- The base case is $\mathtt{i} = 3$, $\mathsf{Fib}[\mathtt{i}] = 2$. In this case, $v' = [1, 2]$, and it evolves into $[1, 2, 1]$, which is the concatenation of $[1, 2]$ with $[1, 1] = [1, \mathsf{Fib}[\mathtt{i} - 1]]$ with the leading 1 removed, and then into $[1, 3] = [1, \mathsf{Fib}[\mathtt{i} + 1]]$. In particular, $[1, \mathsf{Fib}[\mathtt{i} - 1]] = [1, 1]$ has occurred.

- Suppose that the claim has been shown for all $\mathtt{j} < \mathtt{i}$, and suppose that $v' = [1, \mathsf{Fib}[\mathtt{i}]]$. The next thing the algorithm will do is append 1, giving $v' = [1, \mathsf{Fib}[\mathtt{i}], 1]$. The algorithm will go on to append and merge at the end of v' in the same way as it has at the beginning of the algorithm, because all the operations are local and depend only on the last few elements of v' – clearly these states are exactly the concatenations as specified in the lemma, since the algorithm has earlier done those exact things starting from $v' = [1, 1]$ instead. A chain of merges will never propagate past $\mathsf{Fib}[\mathtt{i}]$ until the entry right after $\mathsf{Fib}[\mathtt{i}]$ becomes $\mathsf{Fib}[\mathtt{i} - 1]$, because all the entries of v' are always Fibonacci numbers and $\mathsf{Fib}[\mathtt{i} - 1]$ is the smallest one that is squeezy with respect to $\mathsf{Fib}[\mathtt{i}]$, allowing a merge to propagate. The state $[1, \mathsf{Fib}[\mathtt{i}], \mathsf{Fib}[\mathtt{i} - 1]]$ will occur, by the induction hypothesis; and once it does, the next operation will be another merge, resulting in $[1, \mathsf{Fib}[\mathtt{i} + 1]]$.

■

In particular, since the state $v = [-1, 0, 1]$ occurs (right at the beginning), the lemma implies that for any Fibonacci number f, the state $v = [-1, 0, f]$ will occur.

This already gives us an idea of how the positions of the merge stops will end up recursive just like in the `fairySequence` algorithm; but before we proceed, we need to make sure that their numbers match up. In every iteration of the **repeat** loop, `fairySequence` adds a total of $\mathsf{Fib}[\mathtt{i}]$ new entries ($\mathsf{Fib}[\mathtt{i}] - 1$ from the **for** loop, plus one). We need to make sure that this is the same as the number of merges between the states $v' = [1, \mathsf{Fib}[\mathtt{i} + 1]]$ and $v' = [1, \mathsf{Fib}[\mathtt{i} + 2]]$.

We first need a counting lemma.

Lemma

For all $\mathtt{i} \in \mathbb{N}, \mathtt{i} \geq 3$,

$$\mathsf{Fib}[\mathtt{i}] \quad = 1 + \sum_{\mathtt{j} \in 1\,..\,\mathtt{i}-1} \mathsf{Fib}[\mathtt{j}] \quad = 2 + \sum_{\mathtt{j} \in 2\,..\,\mathtt{i}-1} \mathsf{Fib}[\mathtt{j}].$$

Proof: The base case $\mathtt{i} = 3$ is true, with both sides being 2. For the induction step, note that

$$\sum_{\mathtt{j} \in 1\,..\,\mathtt{i}} \mathsf{Fib}[\mathtt{j}] = \left(\sum_{\mathtt{j} \in 1\,..\,\mathtt{i}-1} \mathsf{Fib}[\mathtt{j}] \right) + \mathsf{Fib}[\mathtt{i} - 1] = \mathsf{Fib}[\mathtt{i}] + \mathsf{Fib}[\mathtt{i} - 1] - 1 = \mathsf{Fib}[\mathtt{i} + 1] - 1.$$

The equivalence of the two sums follows from $\mathsf{Fib}[1] = 1$. ■

Lemma

The number of entries added to `fairySequence` in an iteration of its **repeat** loop, namely $\mathsf{Fib}[\mathtt{i}]$, is equal to the number of merges made between the states $v' = [1, \mathsf{Fib}[\mathtt{i}+1]]$ and $v' = [1, \mathsf{Fib}[\mathtt{i}+2]]$.

Proof: Induction over $\mathtt{i}$.

- One merge happens between $[1,1]$ and $[1,2]$, namely $[1,1,1] \to [1,2]$.
- Suppose the claim has been shown for all $\mathtt{j} < \mathtt{i}$. Denote

  $$f := \mathsf{Fib}[\mathtt{i}], \quad f_p := \mathsf{Fib}[\mathtt{i}-1], \quad f_s := \mathsf{Fib}[\mathtt{i}+1].$$

  We want to count the number of merges from $[1,f]$ to $[1,f,f_p]$ to be $f_p - 1$, because then one more merge to $[1,f_s]$ will yield f_p merges overall.

  We know that $v = [-1,0,f]$ will evolve into $[-1,0,f,1]$, and then, by the lemma above, $[-1,0,f,2]$, $[-1,0,f,3]$, $[-1,0,f,5]$, and so on, through all the Fibonacci numbers up to $[-1,0,f,f_p]$. The induction hypothesis holds for them all, which gives us

  $$\sum_{\mathtt{j} \in 1\,..\,\mathtt{i}-2} \mathsf{Fib}[j] = \mathsf{Fib}[i-1] - 1 = f_p - 1$$

  merges overall.

■

In particular, the last merge, namely

$$\big[1, \mathsf{Fib}[\mathtt{i}+1], \mathsf{Fib}[\mathtt{i}]\big] \to \big[1, \mathsf{Fib}[\mathtt{i}+2]\big]$$

corresponds to the final addition of an entry, outside the **for** loop.

Theorem fairy sequence recursive algorithm

The algorithm `fairySequence` is well-defined (it never accesses elements of Fai that don't exist yet) and computes Fai.

Proof: Both the sequence of merges and the fairy sequence start with $1, 2$.

From the lemmas above it follows that between the states $v' = [1, \mathsf{Fib}[\mathtt{i}+1]]$ and $v' = [1, \mathsf{Fib}[\mathtt{i}+1], \mathsf{Fib}[\mathtt{i}]]$, the algorithm will have made the same $\mathsf{Fib}[\mathtt{i}] - 1$ merges as at the start, just offset by $\mathsf{Fib}[\mathtt{i}+1]$ units; which is exactly what the **for** loop accomplishes. Finally, the merge from $[1, \mathsf{Fib}[\mathtt{i}+1], \mathsf{Fib}[\mathtt{i}]]$ to $[1, \mathsf{Fib}[\mathtt{i}+2]]$ happens at location $\mathsf{Fib}[\mathtt{i}+1]$.

In an iteration of the **repeat** loop where the algorithm needs $\mathsf{Fib}[\mathtt{i}] - 1$ previous values, Fai already has $\sum_{\mathtt{j} \in 1\,..\,i} \mathsf{Fib}[\mathtt{j}] = \mathsf{Fib}[\mathtt{i}+2] - 1 > \mathsf{Fib}[\mathtt{i}] - 1$ entries from the previous iterations. ■

From the recursive algorithm and our theorem, we obtain some very fancy recurrence relations for the fairy sequence. Let Fai^* be the *two*-indexed fairy sequence, i.e. $\mathsf{Fai}^*[\mathtt{i}] := \mathsf{Fai}[\mathtt{i} - 2]$, or equivalently, Fai^* is Fai with two garbage values, say zeros, prepended to the start. Then we have

$$\mathsf{Fai}^*\big[\mathsf{Fib}[\mathtt{i} + 1]\big] = \mathsf{Fib}[\mathtt{i}],$$

and for all $\mathtt{k} > 0, \mathtt{k} < \mathsf{Fib}[\mathtt{i} - 1]$

$$\mathsf{Fai}^*\big[\mathsf{Fib}[\mathtt{i}] + \mathtt{k}\big] = \mathsf{Fib}[\mathtt{i}] + \mathsf{Fai}^*[\mathtt{k} + 1].$$

Moreover, the following corollaries hold:

Corollary

Let Fai_* be the *one*-indexed fairy sequence, i.e. $\mathsf{Fai}_*[\mathtt{i}] := \mathsf{Fai}[\mathtt{i} - 1]$. The restriction of Fai_* to any domain of the form $1 \mathrel{..} f$, where $f \in \mathsf{Fib}$, is a permutation.

Proof: Those are precisely the merges done in Fibonacci sort until a state $v = [-1, 0, f]$ is reached. If Fibonacci sort is run with a list of length f as input, this is precisely where it will halt, and it will have eliminated all barrier stops in $1 \mathrel{..} f$ precisely once. ■

Corollary

Fai_* is a permutation of $\mathbb{N}_+$.

Proof: Follows immediately from the previous one because for each natural number there is a Fibonacci number greater than it. ■

These two corollaries justify the (perhaps more serious-sounding) name "Fibonacci Gray code" for Fai, because it shares analogous properties with the Gray code sequence

$$0, 1, 3, 2, 6, 7, 5, 4, 12, 13, 15, 14, 10, 11, 9, 8, \ldots \qquad \text{(A003188 in the OEIS)}$$

whose restriction to any domain of the form $0 \mathrel{..} 2^k$ is a permutation, and which is a permutation of $\mathbb{N}$.

However, apart from this rather surface-level similarity, there is no bigger connection between Fai and any kind of Fibonacci equivalent of Gray codes; besides, the name "fairy sequence" sounds a lot cooler.

Fibonacci sort implementation in Nim

The following pages contain a Nim implementation of the Fibonacci sort algorithm, along with terminal printouts of the state of v after every addition and merge. Both the original and alternative algorithm are included; the alternative one additionally distinguishes between conditional and unconditional merges in the printouts.

```
import std/[strutils, terminal]


proc merge[T](s: var openArray[T]; lo, mi, hi: int) =
  if s[mi - 1] <= s[mi]: return
  var t = newSeq[T](hi - lo)
  var i = lo
  var j = mi
  for k in 0 ..< hi - lo:
    if i < mi and (j == hi or s[i] <= s[j]):
      t[k] = s[i]
      inc i
    else:
      t[k] = s[j]
      inc j
  for k in 0 ..< hi - lo:
    s[lo + k] = t[k]
```

```
proc fibonacciSort[T](s: var openArray[T]) =
  let n = s.len
  if n <= 1: return
  var
    lo = -1
    mi =  0
    hi =  1
  var v = @[lo, mi, hi]
  while not (mi == 0 and hi == n):
    if hi == n or (mi != 0 and 2 * (hi - mi) >= mi - lo):
      merge(s, lo, mi, hi)
      v.delete(v.len - 2)
      styledEcho(fgMagenta, "xmerge: ", resetStyle, $v)
    else:
      v.add(hi + 1)
      styledEcho(fgRed, "add ", align($(hi+1), 2), ": ",
      resetStyle, $v)
    (lo, mi, hi) = (v[^3], v[^2], v[^1])


proc fibonacciSortAlt[T](s: var openArray[T]) =
  let n = s.len
  if n <= 1: return
  var
    lo = -1
    mi =  0
    hi =  1
  var v = @[lo, mi, hi]
  while hi != n:
    v.add(hi + 1)
    styledEcho(fgRed, "add ", align($(hi+1), 2), ": ",
    resetStyle, $v)
    (lo, mi, hi) = (v[^3], v[^2], v[^1])
    while mi != 0 and 2 * (hi - mi) >= mi - lo:
      merge(s, lo, mi, hi)
      v.delete(v.len - 2)
      styledEcho(fgGreen, "cmerge: ", resetStyle, $v)
      (lo, mi, hi) = (v[^3], v[^2], v[^1])
  while mi != 0:
    merge(s, lo, mi, hi)
    v.delete(v.len - 2)
    styledEcho(fgCyan, "umerge: ", resetStyle, $v)
    (lo, mi, hi) = (v[^3], v[^2], v[^1])
```