A simple and efficient natural merge sort*

Olin Shivers
Georgia Institute of Technology

March 31, 2002

Abstract

Merge-sort algorithms obtain their $O(n \lg n)$ efficiency by balancing the arguments to their merge operations. The standard algorithms for merge-sorting lists achieve this balancing by dynamically allocating extra *administrative* data structures that keep track of the algorithm's progress. The allocation of these administrative data, and their subsequent influence on the algorithm's functioning, complicate the algorithm.

I present a simple algorithm for merge-sorting lists. It is notable because it is both very simple—the core of the algorithm is about five lines of code—and captures all administrative allocation simply in the pattern of recursion, thus requiring *no* allocation, other than the saving of local variables on the recursion stack, and the actual intermediate lists allocated by the merge operations. This means that the "destructive" variant of the algorithm runs entirely in-place, simply rearranging the input list's cells without needing to allocate any other temporary list cells. The number of stack frames required by the algorithm's recursions is also nicely bounded, being logarithmic in the size of the list.

Additionally, the algorithm is *natural*, meaning the leaves of the merge tree are not individual list elements, but rather already-sorted contiguous runs of elements from the input. In other words, the algorithm is *opportunistic*, recursing and merging no more than is necessary. This provides a best-case time of $O(n)$ in addition to the worst-case time of $O(n \lg n)$.

This paper presents and discusses the algorithm, proves the worst-case bound, shows some interesting variants of the basic version, and presents complete, tightly optimised implementations for pure list, destructive list, and vector variants of the sort, in SML, Scheme and C.

---

*Submitted to the *Journal of Functional Programming.*

Contents

1 Opportunistic recursion and "natural" merge sorts

"Natural" merge sorting is an improved variant of the simple merge sort. A natural merge-sort algorithm is one that recursively subdivides the data no more than is necessary—that is, the leaves of the tree of merge operations are maximal-length already-ordered sublists of the original list. For example, a natural merge sort will partition this list of data

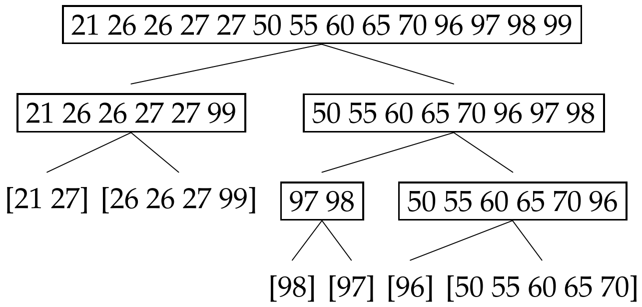
$\boxed{\text{21 27 26 26 27 99 98 97 96 50 55 60 65 70}}$

into the boxed sublists

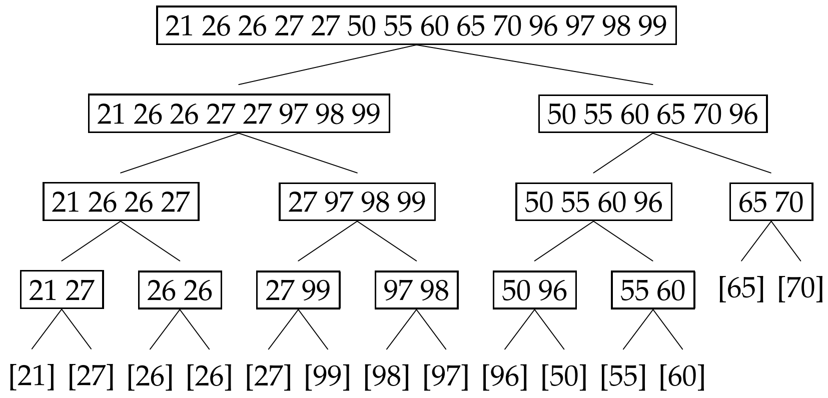
21 27	26 26 27 99	98	97	96	50 55 60 65 70

where each sublist is an already-sorted run of data, and then proceed to merge these sublists to produce the final answer.

Opportunistically partitioning the list in this fashion gives us the following merge tree:



whereas a naive, non-opportunistic merge sort gives us the following complete merge tree, where every leaf is a singleton list:



```
function slowsort(lis) {        (* Bad idea! *)
  accum := []
  for every element x of lis do
    accum := merge([x], accum)
  return accum
  }
```

Figure 1: Unbalanced merge operations make sorting slow.

An opportunistic merge sort allows us to exploit order that already exists in the data, which can pay off handsomely when dealing with real-world data. In the best case, applying an opportunistic merge sort to a completely ordered list of data yields the original list in $O(n)$ time, whereas the naive merge sort always requires $O(n \lg n)$ time.

2 Balancing the merge operations

The key to merge sort's $O(n \lg n)$ running time lies is *balancing* the inputs to the merge operation. Merging two lists of length n_1 and n_2 requires $O(n_1 + n_2)$ steps. Consider, then, what our running time would be if we used the merge-sort algorithm in Figure 1. On the first trip through the loop, the merge operation costs 1 step, the second trip's merge costs 2 steps, and so forth, giving us, all told, a quadratic algorithm. In other words, when our tree of merge operations degenerates into a left-linear structure (that is, all the right children are singleton-list leaf nodes), what we get is an $O(n^2)$ insertion sort.

We get the running time we want by *balancing* our merge operations. We simply ensure that whenever we merge two lists l_1 and l_2, the lengths of the two lists are of roughly the same length—giving us the doubling effect that leads to merge sort's $O(n \lg n)$ running time.

If we are sorting a vector, it is straightforward to do a *top-down* recursion pattern, since we know in advance the length of the vector, and can subdivide as we recurse, ensuring the balance property that we need for the merge operations. Lists are trickier, since we don't know the length in advance, and it takes linear, not constant, time to index the midpoint of a list in order to split it in half. However, restructuring the algorithm in a *bottom-up* fashion allows it to work on-the-fly, efficiently proceeding in a strictly left-to-right manner down the list of elements.

3 Bottom-up merging

Standard algorithms that provide this bottom-up structure do so by dynamically allocating *administrative data-structures* which essentially serve as a work list, allowing the algorithm to determine what to do next. Figure 2 shows the code for one such algorithm, due to O'Keefe [O'Keefe] and adapted from Paulson's introductory text [MLWP]. Without going into the details of this algorithm, note the "worklist" ls, which is a list of lists of elements. Each element of ls is a list of length 2^k for some k; intermediate results are pushed onto, and popped off, merged together, and pushed back onto this list, by the `mergepairs` function. The algorithm cleverly exploits the binary representation of the count variable k to determine when to perform pop/pop/merge/push operations; partial results are kept on ls pending their eventual merging. While this is a very clever algorithm, it's a bit subtle to figure out why the low-order bit of the k value properly drives the control of the merge loop, and the explicit management of the ls value also complicates the algorithm.

4 Using length-counted "improper" lists

Algorithms like samsort are the current state-of-the-art in pure, functional list-sorting algorithms. We can improve on samsort, but before we get to the core of our new algorithm, let's first establish two auxiliary pieces of support code we'll need: the subroutine that picks off contiguous runs of pre-sorted elements, and the one that merges two sorted lists together.

One of the novelties of the algorithm we'll be developing is that it represents some lists with a pointer to the initial list cell and a related integer that gives the length of the list. That is, we do not require the list to be terminated by a special nil tail. If the list is stated to be n elements long, we simply ignore anything that appears in the tail of the list after the first n elements. We do this because the merge-sort algorithm works by repeatedly picking off prefix sublists of the initial list to be sorted. Using a length-count representation allows the algorithm to *share storage* for common prefix sublists of a given list. In contrast, if we require nil-termination, then taking the prefix of a list requires us to copy the prefix, which costs linear time and space.

The routine `getrun`, shown in Figure 3, picks off the longest ordered prefix of elements from its argument `lis`. The parameter `lis` is a nil-terminated list, but `getrun` returns its result using the length-count

```
function samsort(lis) {
  ls := []                      (* List of lists. *)
  k  := 0
  while lis ≠ [] do {
      run, lis := nextrun(lis)
      ls := mergepairs(run::ls, k+1)
      k := k+1
      }
  ls := mergepairs(ls, 0)       (* Do final merges. *)
  return head(ls)
  }

run, rest := function nextrun(xs) {
  case xs of
    x1::x2::xtail =>
      if x1 ≤ x2 then {
        run, rest := nextrun(x2:xtail)
        return x1::run, rest
        }
      else return [x1], x2::xtail
    otherwise => return xs, []
  }

function mergepairs(ls, k) {  (* On each trip through *)
  while true do               (* the while loop, do a *)
      case ls of              (* pop/pop/merge/push on*)
        l1::l2::ls_tail => {  (* ls.                 *)
            if k mod 2 = 1 then return ls
            ls := merge(l1,l2) :: ls_tail
            k := ⌊k/2⌋
            }
        otherwise return ls
  }
```

Figure 2: O'Keefe's smooth applicative merge sort. (The definition of the merge procedure is omitted for space.)

```
len, rest := function getrun(lis) { (* LIS non-empty *)
  x   := head(lis)
  lis := tail(lis)
  run := 1
  while lis ≠ [] and x ≤ head(lis) do {
      x     := head(lis)
      lis   := tail(lis)
      run := run + 1
      }
  return run, lis
  }
```

Figure 3: The auxiliary `getrun` function opportunistically selects pre-sorted initial runs of list elements.

convention—it actually just returns the *length* of the initial prefix, along with the *rest* of the list, *i.e.*, the part that comes after the ordered prefix. It's a simple routine that scans down the list comparing adjacent elements, until it reaches the end of the list or finds a pair of elements that are not ordered, at which point it returns the current list tail and a length-count for the ordered prefix. Note another advantage of using length-counted lists: `getrun` is iterative, running in constant space. In contrast, the `nextrun` function shown with the `samsort` code in Figure 2, which returns a standard nil-terminated result, requires either an $O(n)$ recursion, or a final list-reverse operation.

We also need an auxiliary routine to merge two non-empty sorted lists; this is the `counted_merge` function. The merge is done stably, giving priority to elements from the `a` list. Figure 4 shows the code. It is slightly non-standard since we pass in the argument lists in the length-count representation; otherwise it is unremarkable.

5 The core of the merge sort

What complicates the samsort algorithm is the administrative data structures it allocates, which interact with and drive the control-flow of the algorithm. One of the interesting properties of our new algorithm is that it manages this bottom-up, on-line construction of a balanced merge tree *without* having to heap-allocate any administrative data structures. It uses a different, and appealingly simple, control structure, composed of a loop

```
function cmerge(a, la, b, lb) {
  if la = 0 then return b
  if lb = 0 then return a
  if head(a) ≤ head(b) then
    return head(a) :: cmerge(tail(a), la-1, b, lb)
  else
    return head(b) :: cmerge(a, la, tail(b), lb-1))
  }
```

Figure 4: The auxiliary counted merge function merges two lists in a length-count representation. The length of a is $\mathtt{l_a}$; the length of b is $\mathtt{l_b}$.

inside a local recursion, driven by an integer parameter. This recursive control structure is the central novelty of this algorithm. In a following section, we'll see that the destructive, in-place variant of this algorithm runs without requiring *any* extra storage, except for the stack frames pushed by the recursive calls, with a maximum stack-depth of $O(\lg n)$ frames.

The core of the merge-sort algorithm is the five lines of code in Figure 5. Think of grow as taking three kinds of inputs: (1) a supply of "refined material" (the sorted list s), (2) a supply of "raw material" (the unsorted list u), and (3) a request (the length $\mathtt{l_w}$). It will use the refined and raw materials to construct a sorted list a satisfying the request.

In more detail, the grow procedure has the following contract:

> We pass grow a sorted list s of length l_s, an unsorted list u, and an integer l_w, specifying how long we want the result list to be. List u is nil-terminated, while list s is not required to be.
>
> The procedure will "grow" s up to at least the requested length

```
a, la, v := function grow(s, ls, u, lw) {
  if u = [] or lw ≤ ls then return s, ls, u
  lr, x := getrun(u)
  t, lt, y := grow(u, lr, x, 2^⌊lg ls⌋)
  return grow(cmerge(s,ls,t,lt), ls+lt, y, lw)
  }
```

Figure 5: The core of the merge sort. Note the central pattern of recur/operate/iterate.

l_w by removing elements from u and merging them in with s. It returns the result sorted list a, the length l_a of this list, and v, the remaining unused elements of u that were not merged with s to create a. The answer list a is not required to be nil-terminated.

If u and s are long enough to satisfy the l_w request (that is, if $l_w \leq l_s + \text{length}(u)$) then the procedure must do so (that is, $l_w \leq l_a$). Otherwise, the procedure will use the entire list u (so that $l_a = l_s + \text{length}(u) \leq l_w$).

Note that the contract allows `grow` to pick off, sort, and return a list prefix that is longer than required by l_w. This is the degree of freedom that allows the algorithm to be opportunistic—if we ask for at least seven elements, but the list begins with 10,000 ordered elements, `grow` will just hand back the entire 10,000-element initial run, accomplishing a great deal of work rather quickly (in linear time).

The key feature of the `lw` parameter is that it serves as a *lower bound* on the returned list length—this is how our algorithm achieves balance.

The `grow` procedure works in the following way. If u is empty, then we are finished—there is nothing more to be sorted. Likewise, if $l_w \leq l_s$, then we have already "met quota" with the s list, so we are also done. In either of these cases, we simply return s for the answer list a, and u for the remainder list v.

However, if $l_s < l_w$ and there are elements in u to be processed, we must proceed ahead and do some work. We call `getrun` to pick off an initial, pre-sorted run of elements from u. The run begins with u, and is l_r elements long; x points to the rest of the list. If we're lucky, this run could be the entire list; if we're unlucky, it could be just a single element.

Now, we could merge this run with s, but it might not be long enough for this to be worthwhile. So we need to "grow" our run up until it is long enough to pay for merging it with s—that is, until it is about the same length as s. Fortunately, we have just the procedure we need on hand: `grow`. So, in the fourth line of the algorithm, we make a recursive call to `grow`, passing our initial run (list u, length l_r) as the sorted parameter, along with unsorted list x for further "raw material." Finally, we also need to pass in a lower-bound l_w parameter on this recursive call that will generate a list about as long as s. However, it turns out that simply passing l_s as the lower-bound "want" parameter l_w can cause problems.{Note Problems} What we need is l_s rounded down to the nearest power of two: $2^{\lfloor \lg \texttt{ls} \rfloor}$. The recursive call produces sorted list t, with length l_t, and unsorted remainder y.

In the last line of the algorithm, we merge these two lists to produce a single, longer sorted list, then iterate (or tail-recurse) to further process the remaining unsorted elements.

Parenthetically, you may be wondering why it doesn't damage the balance property that `grow` is allowed to return a much longer list than we need for the merge with s. The answer is that this is acceptable—as long as we generated that longer list *very cheaply.* The `grow` function won't give us extra elements unless it can get them in linear time, in which case we've gotten a "good deal" that justifies locally abandoning balance. We'll cover the details of the run-time analysis in a following section.

Thus, the recursive `grow` call jumps to a right subtree of the merge tree, while the iterative call runs up the left spine of the tree. Note that on every iteration the size of the list roughly doubles (unless we get lucky with a long run of data, in which case it more than doubles). It may help to trace the algorithm by hand on an example list to get a feel for the structure of the recursion and iteration that walks over the merge tree.

6 The top-level interface

All that remains is to write a short "wrapper" procedure that provides the exported interface in terms of the core `grow` function. The wrapper function ensures that the core code is only applied to non-empty lists, and "primes the pump" by picking off an initial run to pass into the first call to `grow`. We force our initial call to `grow` to sort the entire list by passing in an infinite value for l_w (we could pass in the actual length of the input list, but there's no need to pay a linear cost to determine the length of the list when any convenient upper-bound will do just as well).

The top-level procedure also performs one other service: recopying the sorted list. Because the algorithm uses length-counted, improper lists, it may produce a result that is not properly terminated. Hence, one final clean-up pass is needed to copy the result into a fresh, nil-terminated list. The top-level code is shown in Figure 6.

7 Supernatural sorting time

Note the adaptively balancing property of the algorithm: if a recursive call to `grow` gets lucky and returns a larger-than-requested sorted list, the next iteration of `grow` immediately jumps up to the new, larger granularity on its future calls to `grow`. This is not always a characteristic of natural

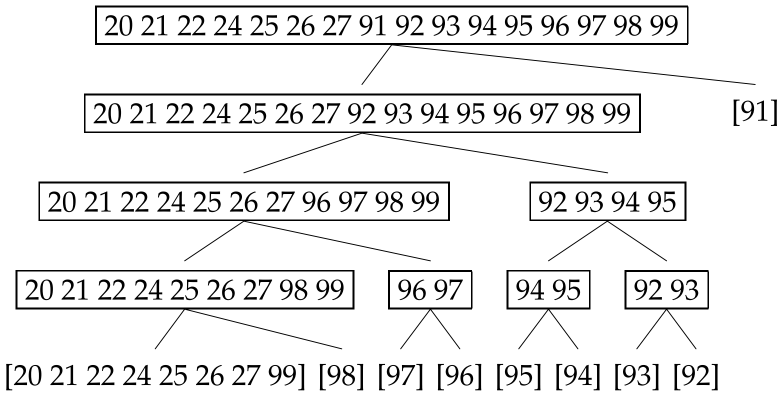
```
function sort(u) {
  if u = [] then return []
  lr, x := getrun(u)
  s, ls, y := grow(u,lr,x,∞)
  return copy(s,ls)
  }

function copy(s,i) {
  if i = 0 then return []
  else return cons(head(s), copy(tail(s), i-1))
  }
```

Figure 6: The top-level "driver" code.

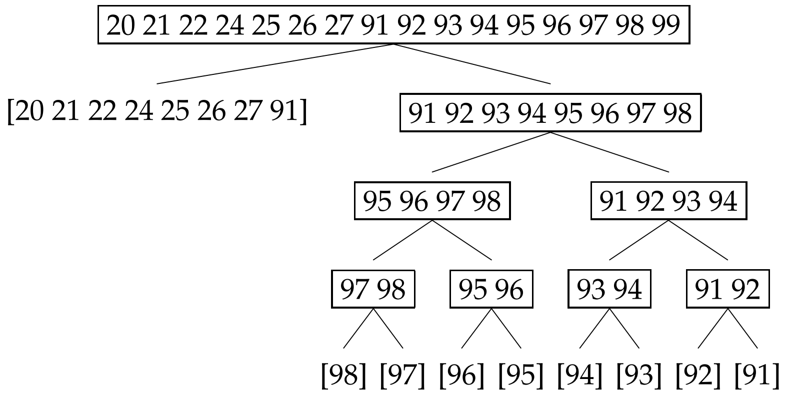
merge sorts. For example, Figure 2's samsort is a natural merge sort, with $O(n \lg n)$ running time. But it is blind to the actual lengths of the runs it finds. Essentially, it splits the input list into its component runs, and then merges these in a simple binary tree with no regard to the actual size of these runs: odd/even adjacent pairs of the leaf nodes are merged together, and so forth. For example, if we have a list of data that breaks down into these runs

20 21 22 24 25 26 27 99	98	97	96	95	94	93	92	91

samsort will treat the initial run of eight elements no differently from a run of a single element, producing the following merge tree:



Although this is balanced enough to give us an $O(n \lg n)$ result, it's easy to see many of these merges are not very balanced. We could do better if we adapted the structure of our merge tree according to the information we discover about the size of the runs. For example, when the `grow` function discovers the initial run of eight elements, it will recursively produce a list of matching size to merge with this initial run. We get the following merge tree:



Here, the individual merge operations are much more balanced, reducing the number of comparisons to sort this example from 56 to 35. This ability to adaptively "shift gears" in response to the size of the runs, changing the granularity of the algorithm on the fly, is going one step better than a simple natural sort.

8 Time bounds

8.1 Establishing the running time

In order to use this algorithm, we need to establish bounds on its running time. We can formalise our notion of "balanced merges" by focussing on the relationship between the height of a merge tree, and the number of list elements it represents. Efficient merge trees are wide and bushy, not narrow and deep. In particular, a good merge tree has the property that the number of list elements it represents dominates 2^{h-1}, where h is the height of the tree.

In order to show our algorithm has $O(n \lg n)$ worst-case time, we must first establish that all the merge trees it handles have this bushy property. To do so, we will need the following trio of simple inductive lemmas about `grow`, which I state without proof (except to note that lemma 3 is a simple consequence of the first two).

Lemma 1: $l_s + \mathrm{length}(u) = l_a + \mathrm{length}(v)$.
(Conservation of data: data are merged from list u with list s, producing list a and remainder v.) □

Lemma 2: $\min(l_s + \mathrm{length}(u), l_w) \leq l_a$.
(Best-effort success guarantee: If there is enough data to satisfy the l_w request, `grow` *will do so. If not,* `grow` *will use up the whole u list.)* □

Lemma 3: $l_s + \mathrm{length}(u) \leq l_w \;\Rightarrow\; l_a = l_s + \mathrm{length}(u)$.
(If we ask for more data than we provide, we get everything.) □

We can now state and prove the critical theorem:

Theorem 1
Let h_s and h_a be the height of the merge trees used to construct the lists s and a passed to and returned from procedure `grow`*, respectively. If $2^{h_s-1} \leq l_s \leq l_w$, then*

(a) $2^{h_a-2} < \min(l_w, l_a)$,

(b) $l_w \leq l_a < l_s + \mathrm{length}(u) \;\Rightarrow\; 2^{h_a-1} \leq l_a$.
(If there is enough data to satisfy the l_w request without running off the end of the u list, the merge tree used to construct the answer list is bushy.)

□

PROOF (BY INDUCTION ON THE LENGTH OF u).
Assume $2^{h_s-1} \leq l_s \leq l_w$. There are three cases:

1. u is the empty list.
   We have run off the end of list u; the algorithm terminates, with $l_a = l_s$, $h_a = h_s$ and $v = u$. Both post-conditions are satisfied by our precondition $2^{h_s-1} \leq l_s \leq l_w$. This establishes the base case of our induction.

2. $l_w \leq l_s$.
   We can satisfy the l_w request simply with list s; no further work is required. Again, the algorithm terminates, with $l_a = l_s$, $h_a = h_s$ and $v = u$, and the post-conditions hold as in the above case.

3. The list u is non-empty and $l_s < l_w$.
   We have not yet satisfied the l_w request, and so the algorithm will have to do some work. Assume, inductively, that the theorem holds for all smaller values of $\mathrm{length}(u)$.

Let $l'_w = 2^{\lfloor \lg l_s \rfloor}$, and $h'_s = \lfloor \lg l_s \rfloor + 1$. That is, l'_w is the largest power of two less than or equal to l_s. Since $2^{h_s-1} \leq l_s$ as well, we know $2^{h_s-1} \leq l'_w$. Thus $2^{h_s-1} \leq l'_w = 2^{h'_s-1}$, so $h_s \leq h'_s$.

There are two cases to consider.

(a) $l_r < l'_w$.
Note that our precondition holds trivially on the recursive call: the run has a height-one merge tree $h_r = 1$, so $2^{h_r-1} \leq l_r$, and our post-conditions hold on the recursive call, by induction: (a) $2^{h_t-2} < \min(l'_w, l_t)$ and (b) $l'_w \leq l_t < l_r + \mathrm{length}(x) \Rightarrow 2^{h_t-1} \leq l_t$. Thus $2^{h_t-2} < l'_w = 2^{h'_s-1} \Rightarrow h_t - 2 < h'_s - 1 \Rightarrow h_t \leq h'_s$.

   i. $\mathrm{length}(u) < l'_w$.
   The recursion is asking for more data than we have, so it will run off the end of the list x, and the returned list will be the entire list u sorted; hence, y will be empty and $l_t = l_r + \mathrm{length}(x) = \mathrm{length}(u)$. This follows from lemma 3. The algorithm will terminate immediately on the iterative call, with $l_a = l_s + l_t = l_s + \mathrm{length}(u)$, and $h_a = 1 + \max(h_s, h_t) \leq 1 + h'_s$. Hence, $2^{h_a-2} \leq 2^{h'_s-1} \leq l_s < l_w$. This, together with $l_s < l_s + l_t = l_a$, establishes post-condition a. Because $l_a = l_s + \mathrm{length}(u)$, the left-hand side of post-condition b's implication is false.

   ii. $l'_w \leq \mathrm{length}(u)$.
   Since $l'_w \leq \mathrm{length}(u) = l_r + \mathrm{length}(x)$, lemma 2 on the recursion gives us $2^{h'_s-1} = l'_w \leq l_t$. Let h_{st} be the height of the merge tree produced by merging s and t. Then $h_{st} = 1 + \max(h_s, h_t) \leq 1 + h'_s$. Suppose $l_s + l_t < l_w$. Then $2^{h_{st}-1} \leq 2^{h'_s} = 2^{h'_s-1} + 2^{h'_s-1} \leq l_s + l_t < l_w$. So we've established our precondition for the iterative call, which will establish our post-conditions by induction.
   On the other hand, suppose that $l_w \leq l_s + l_t$. Then we are done: the algorithm will terminate immediately on the iterative call, with $l_a = l_s + l_t$ and $h_a = h_{st} \leq 1 + h'_s$. So $2^{h_a-2} \leq 2^{h'_s-1} \leq l_s < l_w \leq l_a$, which establishes post-condition a. $2^{h_a-1} \leq 2^{h'_s} = 2^{h'_s-1} + 2^{h'_s-1} \leq l_s + l_t = l_a$, which establishes post-condition b.

(b) $l'_w \leq l_r$.
In this case, the recursion returns immediately, with $l_t = l_r$, and

$h_t = 1$. There are two cases.

i. $l_w \leq l_s + l_t$.
We are finished; the algorithm will terminate immediately on the recursive call: $h_a = 1 + h_s$, and $l_a = l_s + l_t$. $2^{h_a - 2} = 2^{h_s - 1} \leq l_s < l_w \leq l_a$, which establishes post-condition a. $2^{h_a - 1} = 2^{h_s} \leq 2^{h'_s} = 2^{h'_s - 1} + 2^{h'_s - 1} = 2^{h'_s - 1} + l'_w \leq l_s + l_t = l_a$, which establishes post-condition b.

ii. $l_s + l_t < l_w$.
We are not finished; the iterative call will have to do some work. $2^{h_{st} - 1} = 2^{h_s} \leq 2^{h'_s} = 2^{h'_s - 1} + 2^{h'_s - 1} \leq l_s + 2^{h'_s - 1} = l_s + l'_w \leq l_s + l_t < l_w$. This establishes the precondition for the iterative call, which will establish our post-conditions by induction. ■

Our second theorem bounds the number of comparisons operations performed by the `grow` procedure. Let us define $T(l_a, h_a, l_s, h_s)$ to be the number of comparisons performed when applying the `grow` procedure to list s of length l_s, producing result list a of length l_a. The lists s and a were produced by merge trees of height h_s and h_a, respectively.

Theorem 2
$T(l_a, h_a, l_s, h_s) \leq l_a h_a - l_s h_s$.

PROOF (BY INDUCTION ON $l_a - l_s$).
The base case, $l_s = l_a$, is trivial.

To establish the inductive case, $l_s < l_a$, assume the theorem true for all $l'_a - l'_s < l_a - l_s$. Let $h_{st} = 1 + max(h_s, h_t)$ be the height of the merge tree produced by merging s and t.

$$
\begin{aligned}
T(l_a, h_a, l_s, h_s) &= T(l_t, h_t, l_r, 1) + T(l_a, h_a, l_s + l_t, h_{st}) + l_s + l_t - 1 \\
&\leq l_t h_t - l_r + l_a h_a - (l_s + l_t) h_{st} + l_s + l_t - 1
\end{aligned}
$$

The equality on the first line is from the structure of the algorithm's recursion, merge and iteration; note that the cost of merging two lists of length l_s and l_t is at most $l_s + l_t - 1$. The inequality on the second line is by our inductive hypothesis. To finish our proof, we need to show that the right-hand side of this equality is less than or equal to $l_a h_a - l_s h_s$. That is, we need to establish

$$l_t h_t - l_r + l_a h_a - (l_s + l_t) h_{st} + l_s + l_t - 1 \leq l_a h_a - l_s h_s.$$

By some straightforward rearrangement of terms, this is true iff

$$l_t(h_{st} - h_t - 1) + l_s(h_{st} - h_s - 1) + l_r + 1 \geq 0$$

The first two terms are greater than or equal to zero, and $l_r \geq 1$, so we are done. ■

From these two theorems, it is straightforward to establish our desired bound on running time.

Theorem 3
The number of comparisons performed by the `msort` *procedure is* $O(n \lg n)$, *where* n *is the length of the list parameter.* □

PROOF.
The procedure performs a linear number of comparisons to pick off the initial run; we may ignore these. On the initial call to `grow`, $h_s = 1$, so theorem 1 applies. This initial call requests that `grow` sort the entire list—that is, it uses an l_w parameter at least as large as $l_s + \text{length}(u)$. So, by lemma 3, $l_a = l_s + \text{length}(u)$. From theorem 2, this top-level call to `grow` performs at most $l_a h_a - l_s h_s$ comparisons. Theorem 1 gives us $2^{h_a - 2} < l_a$, or $h_a - 2 < \lg l_a$. Thus $h_a < 2 + \lg l_a$, and the number of comparisons performed is

$$\begin{aligned} l_a h_a - l_s h_s &= l_a h_a - l_s \\ &< l_a(2 + \lg l_a) - l_s \\ &= l_a \lg l_a + 2 l_a - l_s \\ &= n \lg n + 2 \lg n - l_s \\ &< n \lg n + 2 \lg n, \end{aligned}$$

which is $O(n \lg n)$. ■

8.2 An open question: Competitiveness

Note that our merge-sort algorithm is, in some sense, an "on-line" algorithm: it makes merge decisions on the fly, as it sequences through the list discovering the sizes of the runs. That is, the algorithm doesn't scan the entire list before deciding how to structure the merge tree.

The left-to-right "on-line" processing order of this algorithm raises the question of how our algorithm's merge trees compare to the best possible merge tree for the data. Imagine a merge-optimal "off-line" sorting algorithm. Let us suppose we are presented with a list of data to merge sort. We could conceivably write an algorithm that would scan the entire list, break it into contiguous runs, and then compute the optimal merge tree for this particular list of data, assuming each merge costs a worst-case $o(n+m-1)$ comparisons. We could feed this tree-shape into the actual merge-sort routine to guide it as it efficiently merges the runs together into the final result.

With this $o(n+m-1)$ metric, it's straightforward to use dynamic programming to determine the shape and cost of the optimal merge tree. If run i has length r_i, and we take $r_{i,j}$ to stand for the sum length of the runs

$$r_{i,j} = \sum_{i \leq k \leq j} r_k,$$

then the minimal cost $C_{i,j}$ to merge together runs i through j is

$$C_{i,j} = \operatorname*{Min}_{i \leq k < j} \Big(C_{i,k} + C_{k+1,j} + r_{i,k} + r_{k+1,j} - 1 \Big).$$

It's interesting to consider how well the merge tree produced by our simple algorithm compares to the optimal one. The question remains open.

9 Sorting destructively

Our algorithm can be easily modified to destructively re-order a list, by side-effecting the list structure. The single change required is to modify the `cmerge` procedure to perform the merge destructively. This gives us a rather simple in-place stable list merge, whose only extra space requirement is the stack space for $\lg n$ recursive calls to the `recur` procedure {Note ConstantSpace}.

A simple destructive merge procedure for length-counted lists is shown in Figure 7. However, we can improve on this simple code in two respects. First, we can implement the merge as a loop instead of a recursion, so that it runs without pushing any stack frames; the simple code above requires $O(n)$ frames. Second, we can reuse our "opportunism" design heuristic, and write the code to avoid doing unnecessary side-effects. These occur when a run of contiguous elements from one of the arguments appears unaltered in the final result; the code above will redundantly alter each list cell in such a run to point to the cell to which it already points. A more sophisticated implementation is written as a pair of scan loops: one that scans across contiguous runs of the `a` list, and one that scans across contiguous

```
function cmerge!(a, la, b, lb) {
  if la = 0 then return b
  if lb = 0 then return a
  if head(a) ≤ head(b) then {
    set_next(a, cmerge!(tail(a), la-1, b, lb))
    return a
    }
  else {
    set_next(b, cmerge!(a, la, tail(b), lb-1))
    return b
    }
}
```

Figure 7: A simple, but inefficient, destructive counted-list merge. (More efficient versions are shown in appendices D and E.)

runs of the `b` list. When a run terminates, a single store is performed to re-point the tail of the run to the head of the other sublist, and we switch over to the alternate scan loop. Avoiding unnecessary writes is especially useful in a system with a high-performance generational garbage collector, since the Achilles heel of these collectors is the overhead added by the collector's write barrier to store operations. Appendices D and E contain complete listings for a destructive list merge sort; the code is tuned for performance, and the list-merge code is the full-blown implementation that runs in constant space and avoids redundant writes.

Using length-counted improper lists pays off in the destructive variant, as well. Instead of eliminating redundant copying, it eliminates the need to nil-terminate intermediate lists, which will subsequently be strung together into longer lists. Again, this is particularly useful for systems with generational GC's (unless the compiler is smart enough to statically detect that nil-termination can be allowed to bypass the write barrier, in which case there's no extra benefit for the generational case over the non-generational case).

Note that the list-scan that searches for the end of the final, sorted list places a heavy load on a computer's memory system, since there is very little computation between pointer loads. The essence of the scan loop is a test/load/decrement/jump loop. This loop issues a load to the memory system every four or so instructions, and the following iteration of the loop is dependent upon this load. Worse, the loads have no locality—they chase the tail of the sorted list through memory.

We can spread out the occurrence of these loads through the computation by arranging for the `grow`, `cmerge`, and `getrun` procedures to return the *final* list element of the result as well as its initial one. The `getrun` operation produces this information "for free," after which it is simply a case of flowing the right pointer value through the recursions and iterations. Once the algorithm is appropriately modified, the top-level driver procedure simply takes this cell as returned by the initial call to `grow`, and places a terminating nil value in it. The code given in appendices D and E shows how this is done.

10 Opportunistically sorting vectors

It is possible to apply the basic idea of this merge sort to vectors. Like the standard vector merge sort, this one requires a temporary "scratch" vector of equal size in which to work. Again, the benefit of doing so is that the resulting sort is natural and adaptive, in comparison to a standard vector merge sort which recurses all the way down to individual elements.

While a complete SML code listing for this sort appears in appendix C, you may find it an amusing puzzle to first work out the details of applying the fundamental idea and basic control structure of this algorithm to the vector case (there are some interesting issues that arise in the vector case).

11 Optimising the round-down operation

Each non-leaf call to our `grow` function computes $l'_w = 2^{\lfloor \lg l_s \rfloor}$. That is, we round l_s down to the largest power of two. How can we carry out this computation efficiently?

The code below shows a simple means of doing so. It passes around a power of two which is a lower bound (but not necessarily the *greatest* lower bound) on l_s, and to double it *up* as necessary. We only need to add one line of code to our basic algorithm:

```
a, la, v := function grow(s, ls, ls2, u, lw) {
  (* ls2 is a power of two less than or equal to ls. *)
  if u = [] or lw ≤ ls then return s, ls, u
  while ls2+ls2 ≤ ls do ls2 := ls2 + ls2
  lr, x := getrun(u)
  t, lt, y := grow(u, lr, x, ls2)
  return grow(cmerge(s,ls,t,lt), ls+lt, ls2+ls2, y, lw)
  }
```

12 Blindly natural merge sort

One way to see the connection between our "supernatural" algorithm and the basic natural sorts, which do not adapt the merge tree to the size of the runs, is to state that the "blindly" natural merge sort is just the adaptively natural merge sort, where we allow the lower bound 2^{h-1} to stand in for the actual lengths of the lists. Tracking the height h of the lists' merge trees rather than their actual lengths builds the same merge tree that O'Keefe's samsort builds:

```
a, la, v := function grow(s, ls, hs, u, hw) {
  if u = [] or hw = hs then return s, ls, u
  lr, x := getrun(u)
  t, lt, y := grow(u, lr, 1, x, hs)
  return grow(cmerge(s,ls,t,lt), hs+1, y, hw)
  }
```

Note that since $\mathtt{h_s}$ is more tightly constrained (it always starts at one, and is only incremented by one), we can change our "quota" comparison from $\mathtt{l_w} \leq \mathtt{l_s}$ to $\mathtt{h_w} = \mathtt{h_s}$, and the top-level driver call can then reliably use zero as its "infinity" $\mathtt{h_w}$ value—a value that can never be reached by starting with height one and incrementing—thus guaranteeing all the unsorted data will be consumed.

13 Complete merge sort as a degenerate case

It's rather pleasant and illuminating, from a pedagogical perspective, that the adaptive algorithm reduces to the naive merge sort, which always recurses all the way down to singleton lists, if we simply define getrun to be lazy instead of greedy:

```
len, rest := function getrun(lis) {
    return 1, tail(list)
    }
```

Alternately, we can strip out any vestige of natural sorting from grow itself, shifting as above to tracking the height of the lists' merge trees, rather than their sizes. If we also dispense with the length-count representation, we are left with an extremely simple core:

```
a, v := function grow(s, h_S, u, h_W) {
  if u = [] or h_W = h_S then return s, u
  t, y := grow([head(u)], 1, tail(u), h_S)
  return grow(merge(s,t), h_S+1, y, h_W)
  }
```

As above, the top-level driver call can use 0 as its infinity $\mathtt{h_W}$ value.

14 Boustrophedon

Our natural sort, as we've developed it, only spots contiguous runs of *increasing* data. So a contiguous run of *decreasing* data would be broken down all the way to its individual elements. We might want to change our algorithm to handle both kinds of runs; this symmetric variation is a well-known technique in natural sorting. All we need to do to manage this is to change our `getrun` function to one that looks at the ordering of the first two elements of its input list. If they increase, the routine scans out the longest increasing prefix; the result can share storage with the input list, if we are using the length-count representation for the answer. If the first two elements decrease, the routine scans out the longest decreasing run, constructing a reversed list as it goes. Figure 8 shows sample code.

Note that the increased complexity of any kind of natural sort only pays off, on average, if there is a skew in the distribution of the inputs. If our input lists have a uniform random distribution, there's no advantage. A bi-directional `getrun` only pays if the likelihood of long runs of decreasing data is greater than pure random chance. Fortunately, real-world data sets frequently do have a fair amount of structure that can be exploited in practice.

15 Refinement types and safely merging counted lists

Our sorting algorithm raises an interesting programming-language issue: in modern "safe" languages such as Java or ML, we are unable to code up a very simple operation in our sort procedure with maximal efficiency. The particular operation where our languages let us down is merging length-counted lists. The counted-merge procedure never does a nil-check before taking the tail of a list. Nonetheless, we know it will never attempt to take the tail of the empty list, as the procedure knows in advance the length of the list. We'd like the compiler to compile the tail operation without doing a preliminary nil check. This unnecessary run-time safety check slows

```
run, runlen, rest := function getrun(lis) {
  case lis of          (* Lis must be non-empty. *)
    [x] => return [x], 1, []
    x1::x2::xrest =>
      len := 2
      prev := x2
      if x1 ≤ x2 then
        (* Scan out increasing sequence. *)
        while xrest ≠ [] and prev ≤ head(xrest) do
            prev := head(xrest)
            len := len+1
            xrest := tail(xrest)
        return lis, len, xrest
      else
          (* Scan out decreasing sequence & reverse. *)
          ans := [x2,x1]
          while xrest ≠ [] and not (prev ≤ head(xrest)) do
              prev := head(xrest)
              ans := prev::ans
              len := len+1
              xrest := tail(xrest)
          return ans, len, xrest
  }
```

Figure 8: A bi-directional, stable `getrun` using the length-count representation for the answer list. Note we must now return the actual run as well as its length, since we can no longer assume the result is a prefix of the input—the decreasing-run case allocates a fresh list.

down the code, which is irritating—part of the reason we are using length-counted lists is to eliminate this cost.

In languages that provide "dangerous" primitives, we can code up the counted-merge procedure without paying for the redundant tests. However, safe languages do not have adequately powerful type or other static annotation facilities to permit us to express in the program text that `(s,ls)` is a list/integer pair such that `ls` is the length of `s`.

Xi and Pfenning have proposed interesting extensions to the type systems of ML-like languages allowing this sort of precision [XiDiss, DepType]. Dependent-type systems of this sort, or static annotation systems such as Arkoudas' Athena [ArkoudasDiss], would allow us to write our counted-merge procedure both safely and efficiently.

16 General tree folding and tree construction

The central idea of our sorting algorithm is just a way to construct balanced trees in an on-line way. We can use this simple recur&iterate pattern, driven by a bound, whenever we have a task of this sort.

For example, suppose we wish to read text from an input stream. Text arrives in chunks of variable size; we would like to read the entire stream to end-of-file and construct a single string containing the results. The four base operations we have are `read_chunk`, which returns the next chunk of text as a string; `append`, which appends two strings, producing a new string; `length`, which returns the length of a string; and `at_eof`, which returns true when we've reached end of file. Assuming `append` requires time and space proportional to the length of its result, it's clear that this loop does $O(n^2)$ amount of copying:

```
ans := ""
while not at_eof() do
  ans := append(ans, read_chunk())
```

However, the obvious variant of our `grow` procedure very efficiently builds the final string:

```
function read_more(s, l_s2, l_w) {
    if at_eof() or length(s) ≤ l_w then return s
    while l_s2 + l_s2 ≤ length(s) do l_s2 := l_s2 + l_s2
    t := read_more(read_chunk(), l_s2)
    read_more(append(t,s), l_s2+l_s2, l_w)
    }
```

I confess I am bemused by the ability of such a simple algorithm to construct such well-balanced trees, without requiring to know in advance the size of the data stream or even when it will cease. The key is the recursion, which allows the algorithm to defer a logarithmic number of merge operations at any given time until sufficient data has arrived or the stream has expired; this is just enough to carry the day.

Other examples where we can use these techniques are unioning a list of sets in an on-line fashion, and building a priority queue from a list of elements. The pattern is also well-suited to operating on network streams or packet sequences.

We can abstract this pattern out into a general "tree fold" combinator that folds an associative, binary operator over a list so that the operator-application tree is a bushy one. When the cost of applying the operator is proportional to the "size" of the operands, this allows us to fold the operator over the list efficiently.

Appendix F shows complete SML definitions for two tree-fold combinators. The second of these is natural and takes contiguous runs of data. The code listing shows one simple and two natural merge-sort functions defined using these combinators, as illustrative examples.

17 Acknowledgments

It's one thing to stumble over an algorithmic trick, but when it comes time to prove things, you have to haul out the big guns. I called on all the ones I know. Jon Bentley taught me about natural sorts and told me inspirational stories. David MacQueen and Bob Tarjan helped me check over the early details of the algorithm. Charles Leiserson privately tutored me in solving recurrences and helped me to solve the $ah_a - sh_s$ recurrence. Matteo Frigo wrote down a one precisely where it was needed. I could not have proved the lower bound without Charles' and Matteo's help.

Chris Okasaki and Simon Peyton-Jones gave me advice on Haskell and pure-functional considerations. Ralf Hinze's web page of pure-functional list sorting algorithms was a great resource—complete code listings and performance measurements for a raft of functional list sorts.

References

[ArkoudasDiss] Kostas Arkoudas.
Denotational Proof Languages.
Ph.D. dissertation, MIT, 2000.

[DepType] Hongwei Xi and Frank Pfenning.
Dependent types in practical programming.
In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '99)*, pp. 214–227, San Antonio, January 1999.

[Knuth3] Donald E. Knuth.
The Art of Computer Programming, volume 3: Sorting and Searching.
Addison-Wesley, 1973.

[MLWP] L. C. Paulson.
ML for the Working Programmer, 2nd *edition.*
Cambridge University Press, 1996. Pages 111–114.

[O'Keefe] R. A. O'Keefe.
A smooth applicative merge sort.
Research paper 182, Department of AI, University of Edinburgh, 1982.

[XiDiss] Hongwei Xi.
Dependent Types in Practical Programming.
Ph.D. dissertation, Carnegie Mellon University, December 1998. Technical Report CMU-CS-91-145, School of Computer Science.

Notes

{**ConstantSpace**}
It is possible to perform a destructive, in-place $O(n \lg n)$-time stable sort of a linked list using only a constant amount of extra storage, instead of the $O(\lg n)$ stack frames required by this algorithm. However, the constant factors in these algorithms render them undesirably slow, and they are also somewhat more complex, so this remains more of a theoretical curiosity than something of practical value. In practice, a $\lg n$ factor can be treated, for all intents and purposes, as a constant less than 64—so from a software engineer's point of view, a fast, simple $O(\lg n)$ algorithm is preferable to a slow, complex $O(k)$ algorithm.

For the persistently curious, several of these constant-space algorithms can be found in the literature. I sketch below the outline of one such algorithm, a variant of Knuth's "Algorithm L" [Knuth3, pp. 163 and exercise 12]. It is a natural, in-place, list-merge sort requiring only constant extra space.

This algorithm makes repeated scans over the list, merging adjacent runs of ordered elements until all the runs have been merged together. When all the runs have been completely coalesced, we are done. Each scan takes linear time and reduces the number of runs in the list by a factor of two (to be precise, $\lceil \frac{n}{2} \rceil$), giving a total run time of $O(n \lg n)$.

At any time during a scan, we partition the list into two pieces: an already-processed prefix, and the as-yet unprocessed tail, $tail$. Let ap be the last list cell of the already-processed prefix (that is, ap's "next" link points to $tail$). A single list scan consists of repeatedly doing the following:

> Pick off the next two runs from $tail$; if $tail$ is empty, or contains only a single run, then this scan is finished. Otherwise, merge the two runs into a single run, set ap's "next" link to point to this new merged run, and update ap to be the last list cell of the new run.

Note that this algorithm performs far too many redundant comparisons scanning and rescanning the list to pick out contiguous runs. In contrast, the opportunistic merge sort that is the main algorithm of this paper retains this information on the stack, once it has done the work to pick out a run—this being the saved information that incurs the $O(\lg n)$ stack-space overhead.

$\{$**Problems**$\}$
Why can't we use l_s for the l_w parameter in the `grow` recursion? Why must we instead round l_s down to the nearest power of two?

Try sorting a list whose runs are of size 8, 7, 6, $\ldots 1$. With l_s used for the lower bound parameter on the recursion, the algorithm exhibits a kind of horizon effect, myopically producing a right-linear tree. See the proof of the central running-time theorem (Theorem 1, page 11) for the details of how the proper choice for the recursive l_w parameter comes into play to ensure that balance is maintained.

Appendices

The following appendices show complete code listings for the following sorts:

- pure list sort, in SML and Scheme;
- destructive list sort, in Scheme and C;
- vector sort, in SML.
- general tree-fold operators and derived merge sorts, in SML.

Online copies of these listings may be found at URL

`http://www.cc.gatech.edu/~shivers/sort/`

A Pure list merge sort: complete SML code

```
fun sort less []  = []                      (* A stable, natural merge sort *)
  | sort less lis =
      let fun getrun(x::xs) =                             (* Pull the next   *)
            let fun lp([], prev, i) = (i, [])             (* run of data     *)
                  | lp(xs as x::xrest, prev, i) =         (* off xs. Return  *)
                      if less(x,prev) then (i, xs)        (* run length &    *)
                      else lp(xrest, x, i+1)              (* following tail. *)
            in lp(xs, x, 1)
            end;

          (* Xs is length lx > 0; ys is length ly > 0. Do a counted merge. *)
          (* The code exploits the lx > 0 & lx > 0 requirement.           *)
          fun cmerge(xs as x::xtail, lx,  ys as y::ytail, ly) =
                if less(y, x)
                then y :: let val ly = ly-1
                          in if ly = 0 then xs
                             else cmerge(xs,    lx,  ytail, ly)
                          end
                else x :: let val lx = lx-1
                          in if lx = 0 then ys
                             else cmerge(xtail, lx,  ys,    ly)
                          end

          fun doubleup(i,j) = (* Double i as far as we can w/o exceeding j. *)
            let fun lp i = let val i2 = i+i
                           in if i2 <= j then lp i2 else i
                           end
            in lp i
            end

          fun grow(s, ls, ls2, [], lw) = (s,ls,[])    (* Core function *)
            | grow(s, ls, ls2, u,  lw) =
                if lw <= ls then (s,ls,u)
                else let val (lr,u2)   = getrun u                        (* Get a run.    *)
                         val ls2       = doubleup(ls2, ls)
                         val (t,lt,u3) = grow(u, lr, 1, u2, ls2)         (* Grow it.      *)
                     in grow(cmerge(s,ls,t,lt), ls+lt, ls2+ls2, u3, lw) (* Merge & loop. *)
                     end

          (* Note: length(lis) or any constant guaranteed *)
          (* to be greater can be used in place of ∞.   *)
          val (lr1, rest) = getrun lis
          val (ans,anslen,_) = grow(lis, lr1, 1, rest, ∞)

          fun first(_,     0) = []
            | first(x::xs, i) = x :: first(xs,i-1)

      in first(ans,anslen) (* Nil-terminate the answer. *)
      end
```

B Pure list merge sort: complete Scheme code

```
(define-syntax mlet ; Multiple-value LET*
  (syntax-rules ()
    ((mlet ((vars exp) rest ...) body ...)
     (call-with-values (λ () exp)
                       (λ vars (mlet (rest ...) body ...))))
    ((mlet () body ...) (begin body ...))))

(define (sort elt< lis)                    ; A stable, natural merge sort
  (if (not (pair? lis)) '()
      (let ((getrun (λ (lis)
                      (let lp ((xs (cdr lis)) (prev (car lis)) (i 1))
                        (if (pair? xs)
                            (let ((x (car xs)))
                              (if (elt< x prev)
                                  (values i xs)
                                  (lp (cdr xs) x (+ i 1))))
                            (values i xs)))))

            (cmerge (λ (a la b lb)
                      (let recur ((x (car a)) (a a) (la la)
                                  (y (car b)) (b b) (lb lb))
                        (if (elt< y x)
                            (cons y (let ((lb (- lb 1)))
                                      (if (= lb 0) a
                                          (let ((b (cdr b)))
                                            (recur x       a la
                                                   (car b) b lb)))))
                            (cons x (let ((la (- la 1)))
                                      (if (= la 0) b
                                          (let ((a (cdr a)))
                                            (recur (car a) a la
                                                   y       b lb))))))))))
        (letrec ((grow (λ (s ls ls2 u lw)
                         (if (or (<= lw ls) (not (pair? u)))
                             (values s ls u)
                             (mlet (((ls2) (let lp ((ls2 ls2))
                                             (let ((ls2*2 (+ ls2 ls2)))
                                               (if (<= ls2*2 ls) (lp ls2*2) ls2))))
                                    ;; LS2 is now the largest power of two <= LS.
                                    ((lr u2)    (getrun u))            ; Get a run.
                                    ((t lt u3)  (grow u lr 1 u2 ls2))) ; Grow it.
                               (grow (cmerge s ls t lt) (+ ls lt)      ; Merge &
                                     (+ ls2 ls2) u3 lw)))))    ;   loop.

                 (first (λ (lis i) (if (= 0 i) '()
                                       (cons (car lis)
                                             (first (cdr lis) (- i 1)))))))

          ;;; Note: (LENGTH LIS) or any constant guaranteed
          ;;; to be greater can be used in place of ∞.
          (mlet (((lr tail)  (getrun lis))
                 ((a la v)   (grow lis lr 1 tail ∞)))
            (first a la)))))) ; Properly nil-terminate answer.
```

C Vector merge sort: complete SML code

```
(* sort less (u, temp, l, r) : unit
*****************************************************************************
** A stable, natural merge sort for arrays.
**
** Sort U[L,R). TEMP is a temporary working vector of size >= R.
**
** recur(l, want)
** Sort U[L,L+LEN) for some LEN
**   where 0 < WANT <= LEN <= (R-L).
**   That is: sort *at least* WANT elements of U[L,R).
**   May put the result into either U[L,L+LEN) or TEMP[L,L+LEN).
**   Will not alter either vector outside this range.
**   Return:
**     - ANSVEC -- the vector holding the value (U or TEMP)
**     - LEN
**     - ANS_IS_U -- tells if ANSVEC is U or TEMP.
**
** lp(s, other, ls, ls2, s_is_u)
** - S[L,L+LS) holds a sorted prefix of U. It's either U or TEMP.
** - OTHER is the vector that *isn't* S.   It's either U or TEMP.
** - LS2 is some power of two <= LS.
** Solve RECUR's problem.
**
** Note that all the <array>_IS_U booleans could be removed; they simply
** enable an optimisation in the internal merge routine.
**
** Uses SML Basis Library Int.min & Array's sub, update & copy.
*)

fun sort less (u, temp, l, r) =
  let fun xor(a:bool, b:bool) = not (a=b)

      fun doubleup(i,j) = (* Double I as far as we can w/o exceeding J. *)
          let fun lp i = let val i2 = i+i
                         in if i2 <= j then lp i2 else i
                         end
          in lp i
          end

      fun getrun(a,l) =                                  (* l < r    *)
          let fun lp(i,x) = if i >= r then i-l           (* x=a[i-1] *)
                            else let val y = sub(a,i)
                                 in if less(y,x) then i-l
                                    else lp(i+1,y)
                                 end
          in lp(l+1, sub(a,l))
          end

      (* Merge S1[L,L+LEN1) and S2[L+LEN1,L+LEN1+LEN2) into
      ** TARGET[L,L+LEN1+LEN2). Merge left-to-right, so that it's OK if
      ** TARGET is S2. TARGET can't be S1. S2_IS_TARGET is true if S2
      ** and TARGET are the same, which allows us to punt the final
      ** blit half of the time.
      *)
```

```
    fun merge(target, s1, s2, l, len1, len2, s2_is_target) =
        let val r1 = l  + len1
            val r2 = r1 + len2

            (* Copy SRC[L,R) to TARGET[TL,...]. *)
            fun blit(src, l, r, tl) =
                copy{src=src, si=l, len=SOME(r-l), dst=target, di=tl}

            fun lp(i, j,x, k,y) =        (* I is next index of TARGET.  *)
                let val i1 = i+1         (* to write. X=S1[J] & Y=S2[K].*)
                in if less(y,x)
                   then let val k = k+1
                        in update(target,i,y);
                           if k < r2 then lp(i1, j,x, k,sub(s2,k))
                           else blit(s1,j,r1,i1)
                        end
                   else let val j = j+1
                        in update(target,i,x);
                           if j < r1 then lp(i1, j,sub(s1,j), k,y)
                           else if not s2_is_target then blit(s2,k,r2,i1)
                           else ()
                        end
                end

        in lp(l, l,sub(s1,l), r1,sub(s2,r1))
        end

    (* This is the core of the sort. *)
    fun recur(l, want) =
        let val len = r-l
            fun lp(s, other, ls, ls2, s_is_u) = (* s <> other *)
              if ls >= want orelse ls = len then (s,ls,s_is_u)
              else let val ls2 = doubleup(ls2,ls)
                       val (t,lt,t_is_u) = recur(l+ls, ls2)
                   in merge(other, s, t, l, ls, lt, xor(s_is_u,t_is_u));
                      lp(other, s, ls+lt, ls2+ls2, not s_is_u)
                   end

        in lp(u, temp, getrun(u,l), 1, true)
        end

in if l < r  (* Don't try to sort an empty range. *)
   then let val (ans,_,ans_is_u) = recur(l, r-l)
        in (* If ANS isn't U, copy it into U. *)
           if ans_is_u then ()
           else copy{src=ans, si=l, len=SOME(r-l), dst=u, di=l}
           end
   else ()
end
```

D Destructive list merge sort: complete Scheme code

```
;;; (sort! < lis)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; A natural, stable, destructive, in-place merge sort.
;;; - natural: picks off maximal contiguous runs of pre-ordered data.
;;; - stable: won't invert the order of equal elements in the input list.
;;; - destructive, in-place: this routine allocates no extra working memory;
;;;   it simply rearranges the list with SET-CDR! operations.

;;; (mlet ((var-list mv-exp) ...) body ...)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; A LET* form that handles multiple values.

(define-syntax mlet
  (syntax-rules ()
    ((mlet ((vars exp) rest ...) body ...)
     (call-with-values (λ () exp)
                       (λ vars (mlet (rest ...) body ...))))
    ((mlet () body ...) (begin body ...))))

;;; (getrun < lis) -> runlen last
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Pick a run of non-decreasing data off of non-empty list LIS.
;;; Return the length of this run, the last cons cell of the run,
;;; and the following list.

(define (getrun elt< lis)
  (let lp ((lis lis) (x (car lis)) (i 1) (next (cdr lis)))
    (if (pair? next)
        (let ((y (car next)))
          (if (elt< y x)
              (values i lis next)
              (lp next y (+ i 1) (cdr next))))
        (values i lis next))))


;;; (merge! < a enda b endb) -> [m endm]
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Destructively merge non-empty lists A & B.
;;; The last cons of A is ENDA. (The cdr of ENDA can be non-nil.)
;;; the last cons of B is ENDB. (The cdr of ENDB can be non-nil.)
;;;
;;; Return the first and last cons cells of the merged list.
;;; The merge is stable.

(define (merge! elt< a enda b endb)
  ;; The logic of these two loops is completely driven by these invariants:
  ;;   SCAN-A: (CDR PREV) = A. X = (CAR A). Y = (CAR B).
  ;;   SCAN-B: (CDR PREV) = B. X = (CAR A). Y = (CAR B).
  (letrec ((scan-a (λ (prev  x a  y b)          ; Zip down A until we
                     (cond ((elt< y x)          ; find an elt > (CAR B).
                            (set-cdr! prev b)
                            (let ((next-b (cdr b)))
                              (if (eq? b endb)
                                  (begin (set-cdr! b a) enda) ; Done.
```

```
                              (scan-b b x a (car next-b) next-b))))

                       ((eq? a enda) (maybe-set-cdr! a b) endb) ; Done.

                       (else (let ((next-a (cdr a)))  ; Continue scan.
                               (scan-a a (car next-a) next-a y b))))))

        (scan-b (λ (prev  x a  y b)          ; Zip down B while its
                  (cond ((elt< y x)          ; elts are < (CAR A).
                         (if (eq? b endb)
                             (begin (set-cdr! b a) enda)      ; Done.
                             (let ((next-b (cdr b))) ; Continue scan.
                               (scan-b b x a (car next-b) next-b))))

                        (else (set-cdr! prev a)
                              (if (eq? a enda)
                                  (begin (maybe-set-cdr! a b) endb) ; Done.
                                  (let ((next-a (cdr a)))
                                    (scan-a a (car next-a) next-a y b)))))))

        ;; This guy only writes if he has to. Called at most once.
        ;; Pointer equality rules.
        (maybe-set-cdr! (λ (pair val) (if (not (eq? (cdr pair) val))
                                          (set-cdr! pair val)))))

    (let ((x (car a))  (y (car b)))
      (if (elt< y x)

          ;; B starts the answer list.
          (values b (if (eq? b endb)
                        (begin (set-cdr! b a) enda)
                        (let ((next-b (cdr b)))
                          (scan-b b x a (car next-b) next-b))))

          ;; A starts the answer list.
          (values a (if (eq? a enda)
                        (begin (maybe-set-cdr! a b) endb)
                        (let ((next-a (cdr a)))
                          (scan-a a (car next-a) next-a y b))))))))



;;; (grow < s ends ls ls2 u lw) -> [a enda la v]
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The core routine.
;;; - < is a comparison function.
;;; - S is a sorted list of length LS > 1, with final cons cell ENDS,
;;;   not properly terminated.
;;; - LS2 is some power of two <= LS.
;;; - U is an unsorted list.
;;; - LW is a positive integer.
;;; Starting with S, and taking data from U as needed, produce
;;; a sorted list of *at least* length LW, if there's enough data
;;; (LW <= LS + length(U)), or use all of U if not.
;;;
;;; GROW takes maximal contiguous runs of data from U at a time;
```

```
;;; it is allowed to return a list *longer* than LW if it gets lucky
;;; with a long run.
;;;
;;; Returns:
;;;   - A:    The result list (not properly terminated)
;;;   - ENDA: The last cons cell of the result list.
;;;   - LA:   The length of the result list
;;;   - V:    The unused tail of U.

(define (grow elt< s ends ls ls2 u lw)
  (let recur ((s s) (ends ends) (ls ls) (ls2 ls2) (u u) (lw lw))

    (if (and (pair? u) (< ls lw))

        (mlet (((ls2) (let lp ((ls2 ls2))
                        (let ((ls2*2 (+ ls2 ls2)))
                          (if (<= ls2*2 ls) (lp ls2*2) ls2))))
               ;; LS2 is now the largest power of two <= LS.
               ((lr endr u2)   (getrun elt< u))
               ((t endt lt u3) (recur u endr lr 1 u2 ls2))
               ((st end-st)    (merge! elt< s ends t endt)))
            (recur st end-st (+ ls lt) (+ ls2 ls2) u3 lw))

        (values s ends ls u))))

(define (sort! elt< lis)
  ;; Note: (LENGTH LIS) or any constant guaranteed
  ;; to be greater can be used in place of ∞.
  (if (pair? lis)
      (mlet (((lr endr rest)  (getrun elt< lis))
             ((a enda la v)   (grow elt< lis endr lr 1 rest ∞)))
        (set-cdr! enda '())
        a)

      '()))
```

E Destructive list merge sort: complete C code

```
/* msort
*******************************************************************************
** A merge sort for linked lists. The sort is
** - in-place & destructive
**   This routine allocates no extra working memory;
**   it simply rearranges the list.
** - natural
**   The routine picks off maximal contiguous runs of pre-ordered data
**   from the list. These runs become the leaves of the merge tree. So
**   it only recurses as far down as needed. Thus, while the worst-case
**   running time is O(n lg n), the routine will process an already-sorted
**   list in O(n) time.
** - stable
**   Equal elements are not disordered. That is, if A comes before B in
**   the input list, and A = B, then A will appear before B in the result
*/

/* This section should be moved into an include file, and that file
** should be included here and by clients, to ensure they synchronise
** on the interface.
*/
typedef void *val_t;    /* Type of the list values. */

typedef struct list {
  val_t val;
  struct list *next;
  } *list_t;

typedef int cmpfun(val_t, val_t);

list_t msort(list_t lis, cmpfun less);

/* The internals start here. */

#include <assert.h>

static const int infinity = 1000000000; /* 1 billion. */

/* Pick a run of data off of non-empty list LIS. Return
** the length of this run and the last element of the run.
*/
static int getrun(list_t lis, cmpfun less, list_t *last)
{
  int i = 1;
  list_t next;
  val_t x;
  assert( lis && last && less );

  next = lis->next;
  x    = lis->val;
  while( next ) {
      val_t y = next->val;
      if( less(y,x) ) break;
      ++i;
```

```
      x    = y;
      lis  = next;
      next = next->next;
      }

  *last = lis;
  return i;
  }


/* Merge non-empty list A (last element ENDA)
** with  non-empty list B (last element ENDB).
** The merge is stable. This code is complex because it
** tightly bummed: it's iterative, not recursive, and it
** makes an effort not to perform unnecessary writes.
**
** Return the head & final elt of the merged result.
*/
static list_t merge(list_t a, list_t enda, list_t b, list_t endb, cmpfun less,
                    list_t *final)
{
  val_t x, y;
  list_t ans, prev;

  assert( a && enda && b && endb && less && final );

  x = a->val;
  y = b->val;

  if( less(y,x) ) {
      ans  = b;
      if( b == endb ) goto doneb;
      prev = b;
      b    = b->next;
      y    = b->val;
      goto scanb;
      }
  else {
      ans  = a;
      if( a == enda ) goto donea;
      prev = a;
      a    = a->next;
      x    = a->val;
      /* Fall through to scana. */
      }
```

```
  /* The logic of these two scan loops is driven by these invariants:
  **   SCANA: prev->next == a. x == a->val. y == b->val.
  **   SCANB: prev->next == b. x == a->val. y == b->val.
  */

scana:
  while( !less(y,x) ) {
      if( a == enda ) goto donea; /* Done. */
      prev = a;                   /* Continue scanning down A. */
      a = a->next;
      x = a->val;
      }

  prev->next = b;
  if( b == endb ) goto doneb;     /* Done. */
  prev = b;                       /* Shift over to B. */
  b = b->next;
  y = b->val;
  /* Fall through to scanb. */

scanb:
  while( less(y,x) ) {
      if( b == endb ) goto doneb; /* Done. */
      prev = b;                   /* Continue scanning down B. */
      b = b->next;
      y = b->val;
      }

  prev->next = a;
  if( a == enda ) goto donea;     /* Done. */
  prev = a;                       /* Shift over to A. */
  a = a->next;
  x = a->val;
  goto scana;

/* Come down here to finish up. */
donea: /* A ran out; tack on remaining B elts & quit. */
  a->next = b;
  *final  = endb;
  return ans;

doneb: /* B ran out; tack on remaining A elts & quit. */
  b->next = a;
  *final  = enda;
  return ans;
  }
```

```
/* The core routine.
** S is a sorted list of length LS > 1, with final element ENDS,
**   not properly terminated.
** LS2 is some power of 2 <= LS.
** U is an unsorted list. LESS is a comparison function.
** Starting with S, and taking data from U as needed, produce
** a sorted list of *at least* length LW, if there's enough data
** (LW <= LS + length(U)), or use all of U if not.
**
** Grow takes maximal contiguous runs of data from U at a time;
** it is allowed to return a list *longer* than LW if it gets lucky
** with a long run.
**
** Returns:
**   - The result list (not properly terminated)
**   - (in *LA) The length of the result list
**   - (in *ENDA) The last element of the result list.
**   - (in *V) The unused tail of U.
**
** It's too bad we have to do multiple-value return in C using
** call-by-reference -- it makes it hard or impossible for a
** compiler to pass values back in registers.
*/
static list_t grow(list_t s, int ls, int ls2, list_t ends,
                   list_t u, int lw, cmpfun less,
                   int *la, list_t *enda, list_t *v)
{
  assert( s && ls > 0 && ends && less && la && enda && v );

  while( ls < lw && u ) {
      list_t t, y, endt;
      int lt, lr = getrun(u, less, &endt);

      do ls2 <<= 1; while( ls2 <= ls );
      ls2 >>= 1;
      /* LS2 is now the largest power of 2 <= LS. */

      t = grow(u, lr, 1, endt, endt->next, ls2, less, &lt, &endt, &y);
      s = merge(s, ends, t, endt, less, &ends);
      ls += lt;
      ls2 <<= 1;
      u = y;
      }

   *la = ls;
   *v  = u;
   *enda = ends;
   return s;
   }
```

```
/* We can use the length of LIS or any constant guaranteed to be greater
** in place of the constant INFINITY below.
*/
list_t msort(list_t lis, cmpfun less)
{
  list_t final, v, ans;
  int len;

  if( !lis ) return 0;
  len = getrun(lis, less, &final);
  ans = grow(lis, len, 1, final, final->next, infinity, less,
             &len, &final, &v);
  final->next = 0; /* Nil-terminate the list. */
  return ans;
  }

#if 0
/* Test harness. Read integers from argv, sort & print out as a list.
** Comparison function compares |n|, so you can see stability.
*/
#include <stdio.h>
#include <stdlib.h>
#define Malloc(type,n)  ((type *) malloc(sizeof(type)*(n)))

static int less(val_t vi, val_t vj) {
  int i = (int) vi,     /* Keep the type */
      j = (int) vj;     /* system happy. */
  if( i < 0 ) i = -i;
  if( j < 0 ) j = -j;
  return i < j;
  }

int main(int ac, char *av[])
{
  int numelts = ac-1;
  struct list *nodevec = 0;
  list_t ans;
  int i;

  if( numelts ) {
      nodevec = Malloc(struct list, numelts);
      if( !nodevec ) {
          fprintf(stderr, "%s: unable to allocate linked list.\n", *av);
          exit(1);
          }

      /* Initialise the linked list from av. */
      for( i=0; i < numelts; ++i ) {
          nodevec[i].val  = (void*) atoi(av[i+1]);
          nodevec[i].next = &nodevec[i+1];
          }
      nodevec[numelts-1].next = 0;
      }

  /* Sort. */
  ans = msort(nodevec, (cmpfun*) less);
```

```
  /* Print linked list. */
  if( !ans ) puts("()");
  else {
      printf("(%d", (int) ans->val);
      while( ans = ans->next )
          printf(" %d", (int) ans->val);
      puts(")");
      }
  }
#endif
```

F SML tree-fold operators and example, derived merge sorts

```
(* Tree-fold combinators: one simple, and one natural. *)

(* tree_fold leaf kons knil xs
 *********************************************************************
 * Construct a balanced binary tree from the elements of xs.
 * - leaf: α→β
 *   Maps an element of xs to a tree leaf.
 * - kons: β×β→β
 *   Constructs a tree from two child trees.
 * - knil: β
 *   The empty tree (which is only used if xs is empty).
 *
 * tree_fold: (α→β) → (β×β→β) → β → α list → β
 * for list-element α, and tree β.
 *)

fun tree_fold leaf kons knil [] = knil
  | tree_fold leaf kons knil (x::xs) =
      let fun grow(tree, height, [],               want) = (tree,[])
            | grow(tree, height, xs as (x::xtail), want) =
                if height = want then (tree,xs)
                else let val (tree2,tail) = grow(leaf x, 1, xtail, height)
                     in grow(kons(tree,tree2), height+1, tail, want)
                     end
      in #1(grow(leaf x, 1, xs, 0)) (* Use zero for ∞. *)
      end
```

```
(* nat_tree_fold getrun kons knil xs:
 *******************************************************************
 * Construct a binary tree from the elements of xs.
 * - getrun: α list → β × int × α list
 *  Picks a run from xs, returning
 *   + the tree for that run,
 *   + the number of elements in it,
 *   + the remainder of xs.
 * - kons: β×β→β
 *   Constructs a tree from two child trees.
 * - knil: β
 *   The empty tree (which is only used if xs is empty).
 * Returns (tree,len) -- the final tree & the number of elements in xs.
 *
 * Note: "length xs" can be used where the following code has an ∞,
 *   as can any integer constant guaranteed to be longer than the length
 *   of any input xs list.
 *
 * nat_tree_fold: (α list → (β × int × α list)) → (β×β→β) → β →
 *                α list → β
 *)

fun nat_tree_fold getrun kons knil [] = (knil,0)
  | nat_tree_fold getrun kons knil xs =

      let fun doubleup(i,j) = (* Double i as far as we can w/o exceeding j. *)
            let fun lp i = let val i2 = i+i
                           in if i2 <= j then lp i2 else i
                           end
            in lp i
            end

          (* ls2 is some    power of 2 ≤ ls.
             ls3 is largest power of 2 ≤ ls. *)
          fun grow(s, ls, ls2, [], lw) = (s,ls,[])
            | grow(s, ls, ls2, u,  lw) =
                if ls >= lw then (s,ls,u)
                else let val (r,lr,tail1) = getrun u
                         val ls3 = doubleup(ls2,ls)
                         val (t,lt,tail2) = grow(r,lr,1,tail1,ls3)
                     in grow(kons(s,t), ls+lt, ls3+ls3, tail2, lw)
                     end

          val (run1, runlen, tail) = getrun xs
          val (ans,  anslen,    _) = grow(run1, runlen, 1, tail, ∞)

      in (ans,anslen)
      end
```

```
(* We can, for example, sort with tree_fold or nat_tree_fold. *)

(* Simple merge sort:
 * msort: (α × α → bool) → α list → α list
 *******************************************************************
 *)
fun merge op< (xs,ys) =
  let fun recur(xs, []) = xs
        | recur([], ys) = ys
        | recur(xs as (x::xtail), ys as (y::ytail)) =
            if y < x then y :: recur(xs,ytail)
            else          x :: recur(xtail,ys)
  in recur(xs,ys)
  end

fun msort less xs =
  tree_fold (fn x => [x]) (merge less) [] xs


(* Natural merge sort
 * natsort1: (α × α → bool) → α list → α list
 *******************************************************************
 * Simple version -- doesn't use length-count representation,
 * so getrun is recursive, not iterative.
 *)
fun getrun op< (x::xs) =
  let fun recur(prev, []) = ([prev], 1, [])
        | recur(prev, ys as y::ytail) =
            if y < prev then ([prev], 1, ys)
            else let val (run,i,tl) = recur(y,ytail)
                 in (prev::run, i+1, tl)
                 end
  in recur(x, xs)
  end

fun natsort1 less xs =
  #1(nat_tree_fold (getrun less) (merge less) [] xs)
```

```
(* Natural merge sort
 * natsort2: (α × α → bool) → α list → α list
 *********************************************************************
 * This one uses the length-count representation -- a "tree" value is an
 * improper list & a length.
 *
 * Unfortunately, the language is not expressive enough for us to make clear
 * that if the length > 0, then the corresponding list is non-nil, so taking
 * the tail of these lists still involves a redundant safety check that slows
 * the code down unnecessarily.
 *)

(* Xs is length lx > 0; ys is length ly > 0. Do a counted merge. *)
(* The code exploits the lx > 0 & lx > 0 requirement.             *)
fun cmerge op< ((xs,lx), (ys,ly)) =
  let fun recur(xs as x::xtail, lx, ys as y::ytail, ly) =
            if y < x
            then y :: let val ly = ly-1
                      in if ly = 0 then xs
                         else recur(xs,    lx, ytail, ly)
                      end
            else x :: let val lx = lx-1
                      in if lx = 0 then ys
                         else recur(xtail, lx, ys,    ly)
                      end
  in (recur(xs,lx,ys,ly), lx+ly)
  end

fun cgetrun op< (xs as x::xtail) =
  let fun lp(prev, i, []) = (i,[])
        | lp(prev, i, ys as y::ytail) =
            if y < prev then (i,ys)
            else lp(y,i+1,ytail)
      val (len,rest) = lp(x,1,xtail)
  in ((xs,len), len, rest)
  end

(* first(lis,i) returns the first i elements of lis.
 * In other words, it converts a list from a length-count
 * representation into a regular, nil-terminated representation.
 *)
fun first(_,       0) = []
  | first(x::rest, i) = x :: first(rest,i-1)

fun natsort2 less xs =
  first(#1(nat_tree_fold (cgetrun less) (cmerge less) ([],0) xs))
```