RECURSIVE FUNCTIONS FOR SORTING ALGORITHMS

VICTOR YODAIKEN

Abstract. Mostly for my own amusement, I wanted to see if it was possible to specify basic sorting algorithms in ordinary algebra without "formal methods" or either pseudo or genuine code. Functional programming people try to do this using functional programming languages and it's all been done in Lisp, but the goal here was to just use recursive functions[1] and sequences and to see whether it made things clearer or not. I think it makes things clearer but you can judge for yourself. The paper starts with basic methods and selection sort, then quicksort, insertion sort, and merge sort before looking at radix sort. There is a discussion in section 6 and a pedantic appendix with all the definitions and obvious lemmas in section 7 . There's code at `https://github.com/yodaikenv/sorting`.

1. Sequences and Selection Sort Version 1

All sequences in this note are finite. We can write a sequence directly, $s = [x_1, \dots x_n]$ or as a finite map $s : i \mapsto x_i$ with a domain (pre-image) that is an index set $\{1, \dots n\}$. I'm going to go back and forth between these two notations as convenient. The empty sequence is Nil. The number of elements of a sequence is $Length(s)$ (so $Length(Nil) = 0$ and $Length([x]) = 1$, etc.). The standard operations on sequences are $s' = appendl(a, s)$ which appends a to sequence s on the left so $s'(1) = a$ and $s'(i + 1) = s(i)$ for $i > 0$, $concat(s, u)$ which concatenates the sequences in order, $delete(s, i)$ which produces a sequence obtained by removing element i from sequence s and $swap(s, i, j)$ which permutes the two indicated elements (all of these are defined in tedious detail in the pedantic appendix, section 7).

If each element in a sequence is associated with some measure, say the sequence s is *in (ascending) order* if for any i and j in the index set of s, if $i < j$ then $measure(s(i)) \leq measure(s(j))$. A sequence of length less than 2 is trivially in order. By default, "in order" means "in ascending order" here. Recall that a permutation of a set X is a one-to-one map from X onto itself. A sequence w is a permutation of a sequence s iff there is some permutation π of the index set of s so that $s(i) = w(\pi(i))$ for all $0 < i \leq Length(s)$ and $Length(w) = Length(s)$. The problem of sorting is the problem of producing an ordered permutation of a sequence efficiently. Note that if s is in order and $measure(s(1)) \geq measure(x)$ then $appendl(x, s)$ is in order.

The simplest sorting method is called selection sort and it constructs a sorted sequence by repeatedly moving the least element from the original sequence and appending it to the new sequence. Since there is no requirement that each element of a sequence has a unique measure, say the least element is the element with the

Date: Nov 21-June 6 2017.
Key words and phrases. programming, sequences, sorting.

minimum index that has a measure less than or equal to the measures of all other elements of the sequence and let $least(s)$ be the index of that element [1].

$$Least(s) = \text{minimum } i : (\forall 0 < j \leq Length(s))(s(i) \leq s(j))$$

$$Selection(s) = \begin{cases} s \text{ if } Length(s) < 2; \\ appendl(s(Least(s)), Selection(delete(s, Least(s))) \text{ otherwise} \end{cases}$$

So if the measure is the element value,

$$\begin{aligned} &Selection([1,3,5,0]) = appendl(0, Selection([1,3,5]) \\ &= appendl(0, appendl(1, Selection([3,5]))) \\ &= appendl(0, appendl(1, appendl(3, Selection([5])))) \\ &= appendl(0, appendl(1, appendl(3, [5]))) \\ &= [0,1,3,5] \end{aligned}$$

Each recursive application of $Selection$ reduces s by one element until it disappears and the recursion stops. But finding the least element requires looking at each element of the remaining unordered sequence and suppose each comparison is one step. If $Length(s) = n$ then there are $n+(n-1)\cdots+1$ comparisons so selection sort takes around n^2 steps. A sort of a million element list then takes a trillion steps. If each step is just one nanosecond, that would take around 20 minutes. I'll return to selection sort below, but let's go to the main event.

2. Quicksort

Quicksort has a reputation for being slippery — and if you try to understand it from code or pseudo-code, it is. It's easier to understand as two different algorithms: a high level algorithm in which the sequence is partitioned into finer and finer divisions and a low level algorithm for accomplishing the partition by swapping.

2.1. High level.

Definition 2.1. A map f on some set of finite sequences[2] is a *quicksort partition* function if and only if for every non-empty sequence s with an appropriate measure, $f(s) = (u, v, w)$ where:

1. u, v, and w are sequences and at least $v \neq Nil$.
2. $concat(u, concat(v, w))$ is a (not necessarily ordered) permutation of s,
3. Every element of v has the same measure (v is called the "pivot"),
4. The measure of every element of u is less than or equal to the measure of the elements of v
5. The measure of every element of w is greater than the measure of the elements of v.

If f is a quicksort partition, quicksort can be defined by

$$qs(s) = \begin{cases} s & \text{if } Length(s) < 2; \\ concat(qs(u), concat(v, qs(w)) & \text{otherwise where } f(s) = (u, v, w) \end{cases}$$

Lemma 2.2. *$qs(s)$ is an ordered permutation of s if f is a quicksort partition map.*

[1]see definition 7.9 for a more formal definition of Least.

[2]You can call it an operator if you want, but maps on maps are common in computer science.

Proof. If $Length(s) < 2$ then $qs(s)$ is trivially an ordered permutation of s. If $s = [s(1), s(2)]$ then for $f(s) = (u, v, w)$, it must be that $concat(u, concat(v, w))$ is an ordered permutation by inspection of cases. If the property holds for sequences of length n or less, consider a sequence s of length $n + 1$. Let $f(s) = (u, v, w)$ and note that $concat(u, concat(v, w))$ is a permutation of s so we just have to show that it is ordered. By definition, v is in order, and we know both u and w are length less than $n+1$ so $qs(u)$ and $qs(v)$ are ordered permutations of u and v, respectively (by the induction hypothesis). Therefore $z = concat(qs(u), concat(v, qs(w))$ must be a permutation of s. And concatenating three ordered sequences that have properties $3, 4, 5$ produces an ordered sequence (see 7.19). □

2.2. **Partitioning by swapping.** The true heart of quicksort is the standard algorithm for partitioning. This algorithm involves picking $s(1)$ as the sole pivot element and then determining two numbers: L and R. R is the greatest (rightmost) index of an element that has measure less than or equal to the measure of the pivot.L is the number of elements starting at element 1 that have measure less than or equal to the measure of the pivot (which includes the pivot itself). If $L < R$ then the sequence looks like

$$s = [x_1 = pivot, \dots x_L, x_{L+1}, \dots x_R, \dots x_n]$$

Then $s(R)$ can be swapped with $s(L+1)$ which will make L increase. Let p be the pivot element.

$$R(s,p) = \max i \text{ s.t. } measure(s(i)) \leq measure(p)$$

$$L(s,p) = \max i \text{ s.t. forall } j \leq i, measure(s(j)) \leq measure(p)$$

(see definitions 7.12 and 7.11 for algorithmic definitions). Partitioning is then just repeated swapping until L and R are equal at which point partitioning is done and all elements x_i for $i > L$ have greater measure than the pivot element.

$$s = [x_1 = pivot, \dots x_{L=R}, \dots x_{R+1}, \dots x_n].$$

Let $L = L(s, s(1))$ and $R = R(s, s(1))$ to simplify the expressions.

$$part(s) = \begin{cases} s & \text{if } Length(s) < 2; \\ part(swap(s, L+1, R)) & \text{if } L < R \\ (u, [s(1)], w) & \text{otherwise} \\ & u = Nil \text{ if } L = 1 \\ & u = [s(2), \dots s(L)] \text{ if } L > 1 \\ & w = Nil \text{ if } L = Length(s) \\ & w = [s(L+1), \dots, s(Length(s))] \\ & \text{if } L < Length(s) \end{cases}$$

This map *part* meets the definition of a quicksort partition map.

Each partition by *part* takes something like $Length(s)$ steps, counting each swap as a step. If the pivots are chosen well, the whole process can take around $\log_2 Length(s) \times Length(s)$ steps to complete quicksort. If they are chosen badly they process can take $Length(s)^2$ steps. Imagine, for example that s is ordered in reverse order to start. If s is all or mostly elements of a single measure, then expanding v can improve behavior.

3. Insertion Sort

In insertion sort, the first element of s is removed on each step and that element is inserted - in order - into the new ordered sequence w.

$$insertionsort(s) = insertall(Nil, s)$$

$$insertall(u, s) = \begin{cases} u & \text{if } s = Nil; \\ insertall(insert(u, s(1)), s') & \text{if } s = appendl(s(1), s') \end{cases}$$

$$insert(u, x) = \begin{cases} appendl(x, u) & \text{if } u = Nil \text{ or } measure(x) \leq measure(u(1)) \\ appendl(u(1), insert(u', x)) & \text{if } u = appendl(u(1), u') \text{ and } measure(x) > measure(u(1)) \end{cases}$$

Insertion sort is exponential, just like selection sort, because each insert may require a complete pass through the sorted sequence depending on the randomness of the initial sequence and on the way sequences are represented[3].

4. Mergesort

Two ordered sequences can be merged into a single ordered sequence.

$$merge(s, u) = \begin{cases} concat(s, u) & \text{if } u = Nil \text{ or } s = Nil \\ appendl(s(1), merge(s', u)) & \text{if } measure(s(1)) \leq measure(u(1)) \\ & \text{ and } s = appendl(s(1), s') \\ appendl(u(1), merge(s, u')) & \text{otherwise if } u = appendl(u(1), u'). \end{cases}$$

Since $merge(s, u)$ reduces the sum of the lengths of the sequence arguments by one on each recursion, it takes at most $Length(s) + Length(u)$ steps to complete (this could be less if one of the sequences reduces faster than the other).

Proving that if s and u are ordered then $merge(s, u)$ is ordered is, surprise, another simple induction proof based on $Length(s) + Length(u)$. Note that if s is in order and $measure(x) \leq measure(s(i))$ for any $0 < i \leq Length(s)$ it must follow that $appendl(x, s)$ is in order.

Proof. If either $s = Nil$ or $u = Nil$ the result is trivial. If neither is empty $Length(s) + Length(u) \geq 2$. Otherwise $s = appendl(s(1), s')$ for some s' and $u = appendl(u(1), u')$ for some u'. Suppose the property holds when $Length(s) + Length(u) = n$ for $n \geq 2$ and suppose $Length(u) + Length(s) = n + 1$. But then either

$$merge(s, u) = appendl(s(1), merge(s', u)$$

or

$$merge(s, u) = appendl(u(1), merge(s, u'))$$

and in either case the right hand merged sequences must be ordered by the induction hypothesis and the result of the concatenation is ordered as noted above. □

[3]If the sequence is represented directly, since it is in order, the insertion point can be found in $\log_2(Length)$ steps, but then it may take *Length* swaps to make room for the new element. It would be interesting to work out what happens if a balanced binary tree is used to hold the sorted sequence.

Mergesort then follows from the observation that single element sequences are trivially in order. Given a sequence s, divide it into 2 parts and do that recursively until we have single element sequences, then merge until the full ordered sequence has been constructed. Suppose $split(s) = (u, v)$ splits s into 2 sequences of equal or near equal length so that $concat(u, v) = s$ and $Length(v) \leq Length(u) \leq 1 + Length(v)$ (see 7.13).

Merge sort then composes the "merge" recursively:

$$msort(s) = \begin{cases} s & \text{if } Length(s) < 2 \\ merge(msort(u), msort(v) & \text{where } (u, v) = split(s)\text{otherwise} \end{cases}$$

It takes $\log_2 Length(s)$ steps to merge down the final splits and each of those takes $Length(s)$ steps so if $n = Length(s)$ then the cost is $n \log_2(n)$ - just as it says in all those introductory algorithms texts!

The problem with mergesort is that for certain kinds of encodings of sequences, the merge operation copies a lot of data. The expression $appendl(u(1), merge(s, u'))$ looks easy, but if the sequences are encoded as adjacent memory cells, then the obvious implementation is copy out the first element of s to some temporary location, then shift each element of s right one position to make space at the beginning and then put the copy of $u(1)$ in the vacant space on the left.

If memory is really tight – if we are writing this code in 1955 or on an embedded microcontroller - then that makes a difference and each merge step from the second sequence takes $Length(r)$ copies! For most cases, if the sequence is encoded as linked list or if there is sufficient memory available, mergesort is very efficient, but the analysis done in the 1950s is still very influential so the less reliable but more memory frugal quicksort is often used as the default sort method.

5. Bucket and Radix sort

Bucket sort can sort a sequence s in $Length(s)$ steps without doing a single comparison. Suppose that the sequence is of global temperature measurements on some day and we want to sort by the mean global temperature in whole degrees centigrade that day. Still now, pre-carbonization, in 2017 we can assume that those temperatures are between 0 and 100 (the globe is never all frozen and not boiling either). Move down the sequence and copy each element is into a "bucket" (set) associated with its measure. We need only 101 buckets. When the whole sequence has been classified, construct a sequence by concatening the elements in bucket order. We can do even better. Before sorting by temperature, first bucket sort the sequence that the sequence by date the same way. Suppose we have 5 years of measurements, so a little over 1800 days (buckets). Then sort by temperature, but keep each bucket in order - the date order. The result will be a sequence where the first element is the earliest measure of the lowest temperature. And all this in $2n$ steps, if collapsing the buckets is discounted and collapsing buckets should be fast. A sorting method where the second sort does not reorder elements of equal measure is called "stable".

Each bucket is a sequence so we need a sequence of sequences: that is a sequence $\alpha : \{1, \dots n\} \to X$ where the elements of X are also sequences. Let's define an operation to append an element to one of the sequence elements:

$$\text{if } \beta = \mathit{InnerAppend}(\alpha, i, a) \text{ then } \beta(j) = \begin{cases} \alpha(j) & \text{if } j \neq i \\ \mathit{appendl}(a, \alpha(j)) & \text{otherwise;} \end{cases}$$

To concatenate the elements of a sequence of sequences:

$$\mathit{concatn}(\alpha, i) = \begin{cases} \mathit{Nil} & \text{if } i > \mathit{Length}(\alpha) \\ \mathit{concat}(\alpha(i), \mathit{concatn}(\alpha, i+1)) & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathit{bucket}(s) &= \mathit{concatn}(\mathit{classify}(\mathit{Nil}, s), k) \\ \mathit{classify}(\alpha, \mathit{Nil}) &= \alpha \\ \mathit{classify}(\alpha, \mathit{appendl}(a, s')) &= \mathit{classify}(\mathit{InnerAppend}(\alpha, \mathit{measure}(a), a), s') \end{aligned}$$

In practice, the limitation of bucket sort is that large numbers of buckets make it expensive. Suppose we want to sort a list of numbers in the range $0 < i < 2^{64}$. In some cases, we need 2^{64} buckets. Certainly, sorting a million element sequence, using 2^{64} sequences for the buckets as part of the sorting process might be impractical. In many cases, this problem can be addressed by smart data structures - for example, we know that if the list is only a million elements in length, it will have at most have a million distinct measure values. The other method is to take advantage of stability and do multiple sorts a positional digit representation of the measure. That is called radix sort.

One example is that a non-negative integer encoded as 32 bits can also be viewed as four digits of base 256, each corresponding to 8 bits. If we bucket sort with 256 buckets, 4 times, we can sort a sequence with measure expressed as a 32 bit number. Say a single passes of $n = \mathit{Length}(s)$ steps to classify and then concatenate each collection of buckets, times 4 digits so $4n$. And each step of classification is generally computationally cheaper than a comparison.

6. Discussion

This is an exercise in ”informal methods”. The idea is not to create some tightly locked validation of methods we know work anyways, but to characterize them mathematically as an aid to understanding and intuition. The other objective was to start to get some idea of where introducing state is necessary. Researchers in functional programming languages argue that “pure functions” simplify and clarify programming and discard all that messy implicit state modification in inferior functional languages. But doing this exercise makes me think state is a less of a binary phenomenon - there is implicit state in a recursive function as it spirals down. Variables even change value: for $f(0) = 1, f(n+1) = f(n) * n$ and $f(2)$ n first is set to 1 and then 0. The function definition describes an algorithm that repeatedly runs in different states. In that sense, recursion permits inclusion of state, but only simple state - once we get changes in multiple variables, traditional recursive functions become awkward.

Consider what happens if we use ”linked lists” to represent sequences. Suppose we have sequence s of values and sequence u of links with $\mathit{Length}(u) = \mathit{Length}(s)+1$ and, initially, $u(i) = i-1$. Then we can define $\mathit{index}(1) = u(1)$ and $\mathit{index}(n+1) = u(\mathit{index}(n))$. A synthetic sequence v can be defined by $v(i) = s(\mathit{index}(i))$. But to describe moving a link, we need to define assignment of a value to a sequence: so if $w = \mathit{assign}(s, i, x)$ then $w(j) = x$ if $j = i \leq \mathit{Length}(s)$ and $w(j) = s(j)$ otherwise.

Trying to juggle multiple changes to multiple sequences starts to become awkward in the recursive function framework.

7. Details

7.1. **Pedantic definitions.** In what follows $n \geq 0$.

Definition 7.1. The empty sequence *Nil* is a sequence. If s is a sequence and a is a value, then $appendl(a, s)$ is a sequence. There are no other sequences.

Definition 7.2.

$appendr(a, Nil) = appendl(a, Nil).appendr(a, appendl(b, s)) = appendl(b, appendr(a, s))$

Definition 7.3. Function application of sequences.

$$(appendl(a, s))(1) = a, \quad (appendl(a, s))(n) = s(n-1) \text{ for } n > 1$$

Note that $s(0)$ is not defined.

Definition 7.4.

$$Length(Nil) = 0, \quad Length(appendl(a, s)) = 1 + Length(s)$$

Definition 7.5.

$$concat(Nil, s) = s, \quad concat(appendl(a, u), s) = appendl(a, concat(u, s))$$

Definition 7.6.

$delete(Nil, n) = s, \quad delete(appendl(a, s), 1) = s, delete(appendl(a, s), n) = appendl(a, delete(s, n-1))$ for $n > 1$

Note that $delete(s, 0)$ is not defined.

Definition 7.7.

$$\begin{aligned}
&replace(s, b, 0) = s \\
&replace(Nil, b, n) = Nil \\
&replace(appendl(a, s), b, 1) = appendl(b, s) \\
&replace(appendl(a, s), b, n+2)) = appendl(a, replace(s, b, n+1))
\end{aligned}$$

Definition 7.8. . If $0 < i < j \leq Length(s)$ then:

$$swap(s, i, j) = replace(replace(s, s(j), i), s(i), j)$$

Definition 7.9. If *measure* is a map to some totally ordered set with $<$ the total order:

$$Least(s) = FindLeast(0, 0, s)$$

$$FindLeast(i, j, s) = \begin{cases} i & \text{if } j > Length(s); \\ Findleast(i, j+1, s) & \text{if } measure(s(i)) \leq measure(s(j)) \\ & j \leq Length(s) \\ Findleast(i+1, j+1, s) & \text{if } measure(s(i)) > measure(s(j)) \\ & j \leq Length(s) \end{cases}$$

Definition 7.10.

$$Flatten_{i=k}^{n}\alpha(i) = \begin{cases} Nil & \text{if } k > n \\ & \text{ or } \alpha(k) \text{ is not a sequence} \\ concat(\alpha(k), Flatten_{i=k+1}^{n}\alpha(i)) & \text{otherwise} \end{cases}$$

Definition 7.11.

$$R(\mathit{Nil}, p) = 0$$
$$R(\mathit{concat}(s', [x]), p) = \begin{cases} \mathit{Length}(s') & \text{if } \mathit{measure}(x) > \mathit{measure}(p) \\ R(s', p) & \text{otherwise.} \end{cases}$$

Definition 7.12.

$$L(s, p) = nL(s, p, 0)$$
$$nL(\mathit{Nil}, p, i) = i$$
$$nL(\mathit{appendl}(x, s'), p, i) = \begin{cases} i & \text{if } \mathit{measure}(x) > \mathit{measure}(p) \\ nL(s', p, i+1) & \text{otherwise.} \end{cases}$$

Definition 7.13.

$$\mathit{split}(s) = \mathit{nSplit}(\mathit{Nil}, s)$$
$$\mathit{nSplit}(u, s) = \begin{cases} (u, s) & \text{if } \mathit{Length}(s) \le \mathit{Length}(u) \\ \mathit{nsplit}(\mathit{appendr}(s(1), u), s') & \text{otherwise where} s = \mathit{appendl}(s(1), s'); \end{cases}$$

Lemma 7.14. *$\mathit{split}(s) = (u, v)$ where $\mathit{Length}(u) - \mathit{Length}(v) \le 1$ and $\mathit{concat}(u, v) = s$.*

Proof. Let's prove $\mathit{nsplit}(u, v) = (u', v')$ where $\mathit{concat}(u', v') = \mathit{concat}(u, v)$ and $\mathit{Length}(u') - \mathit{Length}(v') \le 1 + \mathit{Length}(u)$. If this is true then $\mathit{split}(s) = \mathit{nsplit}(\mathit{Nil}, s) = (u, v)$ where $\mathit{concat}(u, v) = s$ and $\mathit{Length}(u) - \mathit{Length}(v) \le 1 + 0$.

Induction on length of v. If $v = \mathit{Nil}$ then If $\mathit{nslplit}(u, \mathit{Nil}) = (u, \mathit{Nil})$ which fits. If the lemma is true for v with length less than or equal to n and suppose $v = \mathit{appendl}(a, v')$. Then if $\mathit{Length}(v) < \mathit{Length}(u)$ the result holds trivially. Otherwise $\mathit{nsplit}(u, v) = \mathit{nsplit}(\mathit{appendr}(a, u), v')$ and the induction hypothesis takes us over the line. □

7.2. **Pedantic Proofs.**

Lemma 7.15. *If s is a sequence and a is some value, $s' = \mathit{appendl}(a, s)$ is the unique sequence satisfying:*

$$s'(i) = \begin{cases} a & \text{if } i = 1; \\ s(i-1) & \text{if } i > 1 \end{cases}$$

Proof. (I told you it was pedantic). $(\mathit{appendl}(a, s))(1) = a$ by definition. Suppose the lemma holds for $n \ge 1$ and consider $n + 1$ which must be greater than 1. By definition $(\mathit{appendl}(a, s')(n+1) = s'(n+1-1) = s'(n)$. QED. □

Lemma 7.16. *If $u = \mathit{replace}(s, b, n)$ and $0 < n \le \mathit{Length}(s)$ then:*

$$u(i) = \begin{cases} s(i) & \text{if } n \ne i \\ b & \text{if } n = i \end{cases}$$

Proof. Cleary $s \ne \mathit{Nil}$ or $0 < n$ is false so $s = \mathit{appendl}(a, s')$ for some a and s'. Suppose $n = 1$. Then $u = \mathit{replace}(s, b, 1) = \mathit{replace}(\mathit{appendl}(a, s'), 1) = \mathit{appendl}(b, s')$. And so, by the previous lemma $u(1) = b$ and $u(i) = s(i)$ for $i \ne 1$. Suppose the property holds for $0 < n \le k$ and consider $n = k + 1$. Then $u = \mathit{replace}(s, b, k + 1)) = \mathit{replace}(\mathit{appendl}(s(1), s'), b, k+1) = \mathit{appendl}(s(1), \mathit{replace}(s', b, k)$. Let $u' = \mathit{replace}(s', b, k)$. Then $u(1) = s(1)$, and $u(i) = u'(i-1)$ for $i > 1$ by lemma 7.15.

But by the induction hypothesis $u'(i-1) = s'(i-1)$ if $i-1 \neq k$ and $u'(k) = b$. QED □

Lemma 7.17. *$swap(s,i,j)$ is a permutation of s if $s \neq Nil$ $swap(s,i,j)$ is defined.*

Proof. Let

$$u = swap(s,i,j) = Replace(Replace(s, s(j), i), s(i), j).$$

Let $u' = replace(s, i, s(j))$. Obviously $Length(u') = Length(s)$ for $u = replace(u', j, s(i)$ we must have $Length(u) = Length(u') = Length(s)$. By lemma 7.16

$$u(n) = \begin{cases} s(i) & \text{if } n = i \\ u'(n) & \text{otherwise} \end{cases}$$

and by the same lemma this expands to

$$u(n) = \begin{cases} s(j) & \text{if } n = i \\ s(i) & \text{if } n = j \\ s(n) & \text{otherwise} \end{cases}$$

□

Lemma 7.18. *$Selection(s)$ is an ordered permutation of s.*

Proof. There is nothing to prove if $Length(s) < 2$. Suppose the proposition true for sequences of length n and suppose $Length(s) = n+1$. Then $Selection(s) = concat([s(Least(s)], Selection(delete(s))$. We know $Selection(delete(s))$ must be ordered and a permutation of $delete(s)$ because $Length(delete(s)) = n$. But

$$concat([s(Least(s)], delete(s)])$$

is obviously a permutation of s so $concat[s(Least(s))], Selection(delete(s))]$ must also be a permutation of s and it has to be in order if $Selection(delete(s))$ is in order and $s(Least(s))$ is really the least element. □

Lemma 7.19. *If u and s are ordered sequences according to some measure, and the measure of every element of u is less than or equal to the measure of every element of s then $w = concat(u,s)$ is ordered.*

Proof. Suppose the contrary and for some $i < j$ we have $w(i) > w(j)$. If i and j are both less than or equal to $Length(u)$ or both greater than $Length(u)$ then the assumption that u or s is ordered is false.If $0 < i \leq Length(u)$ and $Length(u) < j \leq Length(w)$ then the assumption that every element of u has a measure less than or equal to the measure of every element of s must be false. □

7.3. **A pedantic explanation of radix sort.** Any non-negative integer n can be expressed in positional notation in base b as a sequence of length $\lceil \log_b(n) \rceil$ with each element holding a non-negative integer $0 \leq d < b$. Let $s = digits(n,b)$ be the sequence of such digits so that $n = \sum_{i=1}^{\lceil \log_b n \rceil} s(i) \times b^{Length(s)-i}$. So $digits(123, 10) = s = [1,2,3]$. And $123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$. Suppose n and k are non-negative integers and $s = digits(n,b)$ and $u = digits(k,b)$. Note that $n > k$ if and only if either $Length(s) > Length(u))$ or $Length(s)) = Length(u)$ and there is some $j < Length(u)$ so that $s(j) > u(j)$ and $\forall 0 < k < j, s(k) = u(k)$. So $124 > 123$ because the lengths of the two sequences decimal are equal and the third digit of the first number is greater than the third digit of the second number and all lower indexed digits are equal.

Let $digit(i, r, j)$ be the j^{th} digit (counting from 0 on the right) of the encoding of integer i in base r.

$$digit(i, r, j) = (\lfloor i/r^j \rfloor \bmod r$$

For example $digit(123, 10, 0) = 3$ and $digit(123, 10, 1) = 12 \bmod 10 = 2$. This computation is very efficient on digital computers when r is a power of 2. Radix sort just applies bucket sort for each digit

$$m_j(x) = digit(x, r, j)$$

$$Rsort(s, r, j, k) = \begin{cases} s & \text{if } Length(s) < 2 \\ & \text{or } j = k \\ Rsort(bucket(s, m_j, r), r, j+1, k) & \end{cases}$$

Then $Rsort(s, r, 0, k)$ will sort by k digits starting from the lowest order digit. Of course it is faster if we combine multiple passes into one.

References

[1] Roezsa Peter. *Recursive Functions in Computer Theory.* New York, NY, USA: Halsted Press, 1982. ISBN: 0470271957.

Austin Texas.
E-mail address: `victor.yodaiken@gmail.com`