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 meta-mathematics 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 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 quicksort, then moves on to slection sort and merge sort before looking at radix sort.

1. Sequences and Quicksort

The programming language LISP is built on the most fundamental of programming objects - the finite list which can also be called a sequence. Here I want to try for some of the same expressive compactness by describing programming on lists in terms of recursive functions on finite sequences [1] without using any programming languages or pseudo-code or anything like a formal method. 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 the "length" written $Length(s)$ (so $Length(Nil) = 0$ and $Length([x]) = 1$). 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.

Quicksort is one of the most famous and most slippery methods for ordering sequences but it's pretty intuitive and simple at a high level. The more complicated part is a clever method of partitioning a sequence by repeatedly "swapping" elements (in a 2-cycle) - something that can be very efficient for computers. Let's cover one at a time.

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

1.1. **High level.** Concatenation of sequences is straightforward, but it's easy to define:

$$\text{If } w = Concat(s,u) \text{ then } w(i) = \begin{cases} s(i) & \text{if } i \le Length(s) \\ u(i - Length(s)) & \text{otherwise} \end{cases}$$

A map f on some set of finite sequences[1] 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. $Concat(u, Concat(v,w))$ is a (not necessarily ordered) permutation of s,
2. $v \ne Nil$ and every element of v has the same measure,
3. every element of u has a measure less than or equal to the measure of the elements of v
4. every element of w has a measure greater than the measure of the elements of v.

The sequence v is called "the pivot". 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 1.1. *$qs(s)$ is an ordered permutation of s if f is a quicksort partition map.*

Proof. If $Length(s) < 2$ then $qs(s)$ is trivially an ordered permutation of s. If $s = [x,y]$ then if $f(s) = (u,v,w)$, 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 $n+1$. Then if $f(s) = (u,v,w)$ and we know that $Concat(u, Concat(v,w))$ is a permutation of s. 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. Concatenating three ordered sequences that have properties $2,3,4$ produces an ordered sequence. □

1.2. **Partitioning by swapping.** The true heart of quicksort is the standard algorithm for partitioning. This algorithm involves picking $s(1)$ as the 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

$$s = [l_1 = pivot, \dots l_L, x_1, \dots x_R, \dots x_n]$$

Then $s(R)$ can be swapped with $s(L+1)$ which will make L increase. It is simple to define these algorithmically[2] , but here we just need the sense. Let p be the pivot element.

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

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

[2]If you insist: Let $R(Nil,p) = 0$ and let $R(Concat(s',[x]),p) = Length(s')$ when $measure(x) > measure(p)$ and $R(Concat(s',[x]),p) = R(s',p)$ otherwise. Let $L(s,p) = nL(s,p,0)$ and define $nL(Nil,p,i) = i$ and $nL(Concat([x],s'),p,i) = i$ if $measure(x) > measure(p)$ and $nL(s',p,i+1)$ otherwise.

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

Partitioning is then just repeated swapping until L and R are equal at which point partitioning is done.

$$\begin{aligned}
&\text{if } u = \mathit{Swap}(s,j,k)\\
&u(i) = \begin{cases} s(k) & \text{if } i = j\\ s(j) & \text{if } i = k\\ s(i) \text{ if } i \neq j \text{ and } i \neq k \end{cases}\\
&\mathit{Length}(s) = \mathit{Length}(r)
\end{aligned}$$

It should be clear that $\mathit{Swap}(s,j,k)$ is a permutation of s.

Let $\ell = L(s,s(1))$ and $R = R(s,s(1))$

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

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

Each partition by *part* takes something like $\mathit{Length}(s)$ steps, counting each swap as a step. If the pivots are chosen well, the whole process can take around $\log_2 \mathit{Length}(s)\mathit{Length}(s)$ steps to complete quicksort. If they are chosen badly they process can take $\mathit{Length}(s)^2$ 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.

2. Selection and Insertion Sort

"Selection sort" starts with sequences $s_0 = s$ and $w_0 = \mathit{Nil}$. At each step, "the least" element x of s_i is selected and removed to obtain a shorter sequence s_{i+1} and then we let $w_{i+1} = \mathit{Concat}(w,[x])$. Each w_i must be in order, by construction, and when $s_i = \mathit{Nil}$, w_i must be an ordered permutation of s.

Since there is no requirement that each element of a sequence has a unique measure, say the least element is the element with the minimum index that has a measure less than or equal to the measures of all other elements of the sequence.

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

More concretely: let $\mathit{Least}(s) = nb(1,1,s)$ where:

$$nb(i,j,s) = \begin{cases} nb(j,j+1,s) & \text{if } j \leq \mathit{Length}(s) \text{ and } \mathit{measure}(s(j)) < \mathit{measure}(s(i))\\ nb(i,j+1,s) & \text{if } j \leq \mathit{Length}(s) \text{ and } \mathit{measure}(s(i)) \leq \mathit{measure}(s(j))\\ i & \text{otherwise} \end{cases}.$$

$$zap(s) = \begin{cases} Nil & \text{if } Length(s) < 2 \\ Concat([s(1) \dots s(Least(s))], & \\ [s(Least(s)+1) \dots s(Length(s))]) & \text{if } 1 < Least(s) < Length(s) \\ [s(2) \dots s(Least(s))] & \text{if } 1 = Least(s) < Length(s) \\ [s(1), \dots s(Length(s)-1)] & \text{otherwise} \end{cases}$$

$$Selection(s) = \begin{cases} s \text{ if } Length(s) < 2; \\ Concat([s(Least(s))], Selection(zap(s)) \text{ otherwise} \end{cases}$$

Each recursive application of *Selection* reduces s by one element until it disappears and the recursion stops[3]. Selection sort takes n^2 or so steps on real computers because finding the least element in s_i requires comparing the best candidate against all $Length(s_i)$ elements so the number of comparisions is $n + (n-1) + \dots 1 = (n^2 + n)/2$ when $n = Length(s)$.

Lemma 2.1. *$Selection(s)$ 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 $Selection(s) = Concat([s(Least(s)], Selection(zap(s))$. We know $Selection(zap(s))$ must be ordered and a permutation of $zap(s)$ because $Length(zap(s)) = n$. But

$$Concat([s(Least(s)], zap(s)])$$

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

We might as well end this section with an alternative exponential sorting method that is the complement of selection 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.

$$insert(Nil, x) = [x]$$

$$insert(s, x) = \begin{cases} [x] & \text{if } s = Nil \\ Concat(s, [x]) & \text{if } measure(s(Length(s))) \leq measure(x) \\ Concat(insert(u, [x]), [y]) & \text{if } s = Concat(u, y) \text{ and } measure(y) > measure(x) \end{cases}$$

$$insertall(u, s) = \begin{cases} q & \text{if } s = Nil; \\ insertall(insert(u, x), v) & \text{if } s = Concat([x], v) \end{cases}$$

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

[3]For example: $Selection([3,2,1]) = [1] \frown Selection([3,2]) = [1] \frown [2] \frown Selection([3]) = [1,2] \frown [3] \frown Selection(Nil) = [1,2,3]$.

3. Mergesort

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

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

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 $Concat([x], s)$ is in order.

Proof. If either $s = Nil$ or $u = Nil$ the result is trivial. If neither is empty $Length(s){+}Length(u) \geq 2$. 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

$$merge(s,u) = Concat([s(1)], merge(s,u)$$

or

$$merge(s,u) = Concat([q(1)], merge(r,z))$$

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. □

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.

$$Split(s) = \begin{cases} (s, Nil) & \text{if } Length(s) < 2 \\ (u,v) & \text{otherwise} \\ & \text{where} u = [s(1), \ldots s(\lfloor Length(s)/2 \rfloor)] \\ & \text{ and } v = [s(\lfloor Length(s)/2 \rfloor) + 1), \ldots Length(s))] \end{cases}$$

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 $Concat([q(1)], merge(r,z))$ looks easy, but if the sequences are encoded as adjacent memory cells, then the obvious implementation is copy out the first element of q to some temporary location, then copy each element of r down one position to make space at the beginning and then copy the stored first element of q back to the beginning for $Length(q)$ copies.

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.

4. Bucket and Radix sort

Suppose we have a measure m so that for some "small" integer k, and any $0 < i \leq Length(s)$ we have $0 \leq m(s(i)) \leq k$. Small can be different depending on circumstances, but for a simple case suppose that the sequence is of dates and the measure is the mean global temperature in whole degrees centigrade that day. Until the next decade, assume that those temperatures are between 0 and 100 (the globe is never all frozen and not boiling either). Then sorting can be done by classifying in $Length(s)$ steps. Each element is put 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. Suppose the original sequence is already ordered by date and we want to preserve the relative ordering of elements that have equal temperature. In that case, the buckets can be sequences, filled in sequence order. The the first element of the list will be the earliest date with the lowest temperature. The property of not disturbing previous ordering of equal measure elements is called "stability".

Let's use S and W for sequences of sequences - where each $W(i)$ is itself a sequence. Concatenation works as usual since these are just sequences. Suppose $S = [s, u]$ and $w = [x_1, \dots x_n]$. Then $Concat(S, [w]) = [s, u, w]$ but $Concat(S, w) = [s, u, x_1, \dots x_n]$ which would be less useful. It is, however, useful to be able to concatenate a sequence to one of the elements of such a sequence of sequences.

Define

$$\text{if } W = CConcat(S, s, i) \text{ then } W(j) = \begin{cases} Concat(W(i), s) & \text{if } j = i \\ W(j) & \text{otherwise.} \end{cases}$$

Flattening a sequence of sequences concatenates all its elements.

$$flatten(S) = \begin{cases} Nil & \text{if } S = Nil \\ Concat(s, Flatten(S')) & \text{if } S = Concat([s], S') \end{cases}$$

Let s^n be the sequence S with length n so that each $S(i) = s$. Then Nil^n is a sequence of n empty sequences. Suppose the image of m is $\{0, \dots k\}$

$$bucket(s, m, k) = nbucket(Nil^{k+1}, s, m)$$

$$nbucket(S, s, m) = \begin{cases} flatten(S) & \text{if } s = Nil \\ nbucket(CCconcat(S, [x], m(x)), s', m) & \text{if } s = Concat([x], s') \end{cases}$$

If the the measure is in a range that is not relatively small for our computing context: maybe $0 \leq measure(x) < 2^{64}$, we can still use bucket sort in a number of ways including by use of positional notation. For right now, a queue of 2^{64} elements seems too big so we can use repeated bucket sort of digits in some encoding of the integers. At base 256 a number between 0 and 2^{64} can be encoded in a mere 8 digits.

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.

5. 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 $Length(u) = Length(s)+1$ and, initially, $u(i) = i-1$. Then we can define $index(1) = u(1)$ and $index(n+1) = u(index(n))$. A synthetic sequence v can be defined by $v(i) = s(index(i))$. But to describe moving a link, we need to define assignment of a value to a sequence: so if $w = assign(s, i, x)$ then $w(j) = x$ if $j = i \leq Length(s)$ and $w(j) = s(j)$ otherwise. Trying to juggle multiple changes to multiple sequences starts to become awkward in the recursive function framework.

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