SIMPLE ALGEBRA FOR SORTING ALGORITHMS

VICTOR YODAIKEN

Abstract. Fundamental programming concepts described with basic algebra and recursive functions. This is a working draft with more methods to be added - working through quicksort now.

1. Introduction

It's odd that all the descriptions of basic programming operations, such as sorting, rely on pseudo code or complex formal logic. All we are doing is modifying finite sequences so it seems like we should be able to use ordinary algebra. I'm going to start by defining basic operations on finite sequences and then look at the usual algorithms for sorting them - all without either using any pseudo code or any formal methods.

This is somewhat unorthodox, but it's convenient to define a finite sequence of length n as a map $s : \{1...n\} \to X$ where X is often left unspecified[1].The requirement is that the map must be total - that is defined on every element of $\{1..n\}$. The more standard notation of $\langle x_1, \dots x_n\rangle$ is generally a bit clumsier here (see the definition of "cut" below).

- Write $|s|$ for the length of the sequence so if $s : \{1...n\} \to X$, then $|s| = n$.
- Let e be the empty sequence - the map from the emptyset. Sensibly $|e| = 0$.
- Write $\langle x\rangle$ for the sequence with one element $\langle x\rangle : \{1\} \to \{x\}$ so $\langle x\rangle(1) = x$.
- Write $s * t$ to concatenate sequences so if $z = s * t$ then:

$$z(i) = \begin{cases} s(i) & \text{if } i \leq |s| \\ t(i - |s|) & \text{otherwise} \end{cases}$$

- When it is clear from context that x is an element and s is a sequence, I will write xs to abbreviate $\langle x\rangle * s$.

If B is a boolean valued map on pairs of elements of X and s is a sequence, say that s is ordered by B if and only if $B(s(i), s(i + 1))$ for all $0 < i < n$. Note that both e and $\langle x\rangle$ are always in order. In general, B will define a partial order, but I'm not sure if that's the boundary of interesting predicates.

Lemma 1.1. *If t is ordered by B and $|t| > 0$ and $B(x, t(1))$ then the sequence $s = xt$ is ordered by B.*

Date: June 6 - 16 2017.

Key words and phrases. programming, sequences, sorting.

[1]The more usual algebra definition is that for a finite set X, the set X^* is the free monoid of strings with e as the identity. The representation of strings as maps produces an isomorphic free monoid so I will just assert the usual properties (such as associativity) without further notice.

Proof. We know $B(x, t(1))$ so since $s(1) = x$ and $s(2) = t(1)$, we know that $B(s(1), s(2))$. If $i > 1$ then $i-1 \geq 1$ and $s(i) = t(i-1)$. But we know that for $i > 1$ we have $B(t(i-1), t(i))$ since t is ordered. So we must have $B(s(i), s(i+1))$. □

We don't need any special mathematics here, but in computer science functions that modify or describe the effects of modifications on other functions are common - due to a fundamental property of computing. So next I'm going to define what it means for a function to be a sorting or ordering function. These functions will map a sequence (which we have already defined as a type of map) and an ordering predicate, to a new sequence that has the same elements permuted into the order defined by the ordering predicate. The ordering predicate (boolean function) is an argument so that "sorting" can be considered without committing to what objects we are sorting: $f(s, B)$ sorts s using B and and it's up to B to come up with a measure for elements of the sequence s.

Definition 1.2. A sequence s is a "permutation" is true for of sequence t if and only if both the sequences have the same length and they have the same distribution of elements or, more formally: $|s| = |t|$ and there is an onto map $\gamma : \{1...|s|\} \to \{1...|s|\}$ so that

$$s(p(i)) = t(i) \text{ for all } 0 < i \leq |t|.$$

(Alternatively, you could consider sorting a map s relative to B to be the same problem as finding the element γ of the symmetric group S_n so that $B(s(\gamma(i)), s(\gamma(i+1))$ for all $0 < i < |s|$).

Definition 1.3. f orders sequences according to a boolean predicate B iff $t = f(s, B)$ is ordered by B and t is a permutation of s.

2. Algorithms: the beginning

Now we can look at algorithms. The simplest sorting algorithm repeatedly removes the best (according to B) element of a sequence and glues those together to build a new sequence. Let

Definition 2.1. $Best(s, B)$ = the least $i \leq |s|$ so that for all $0 < j \leq |s|$ where j is not equal to i, we have $B(s(i), s(j))$

This definition is cheating, because it does not provide an algorithm, but the algorithm is easy to define (see 5.1).

Define another operation to cut elements from a sequence:

For any sequence s where $0 < i \leq |s|$, $t = cut(s, i)$ is a sequence:

$$t(j) = \begin{cases} s(j) & \text{if } 0 < j < i; \\ s(j+1) & \text{if } j \geq i \end{cases}$$

Note that $|cut(s, i)| = |s| - 1$ if $|s| > 0$ and $0 < i \leq |s|$.

If s is a sequence, $Best(s, B)$ is the index of its "best" element, $s(Best(s, B))$ is the value of that element, and $cut(s, Best(s, B))$ is the sequence obtained by removing the best element. Now we can sort a sequence recursively by

$$Sort(s, B) = \begin{cases} e & \text{if } s = e \\ \langle s(Best(s, B)) \rangle * Sort(cut(s, Best(s, B)), B) & \text{otherwise} \end{cases}$$

The time complexity of this is easy: each Best takes as many steps as the length of the sequence and we have to recurse $|s|$ times, so it takes around $(|s|)^2 + |s|)/2$ steps.

Should we prove it works? Proof is by induction (or recursion) on the length of the sequence.

Proof. If $s = e$, no problem. Suppose the property holds for sequences of length n and s has length n+1. Then cutting s gives some t with length n and $Sort(t, B) = Sort(cut(s, Best(s, B)), B)$ must be sorted by B by the inductive hypothesis and the lemma plus the definition of $Best$ finishes the proof. □

Mergesort is more fun.

3. Mergesort

Mergesort is based on the observation that if you merge two ordered sequences, in order, you get a sequence that is still in order.

$$Merge(e, e, B) = e$$
$$Merge(xq, e, B) = xq$$
$$Merqe(e, yq, B) = yq$$

$$Merge(xs, yt) = \begin{cases} \langle x \rangle * Merge(s, yt, B) & \text{if } B(x, y) \\ \langle y \rangle * Merge(xs, t, B) & \text{otherwise} \end{cases}$$

Observe that $Merge(s, t, B)$ reduces the two sequence arguments by one on each recursion so it takes at most $|s| + |t|$ steps to complete (this could be a lot less if one of the sequences reduces faster than the other).

Let's prove that if s and t are B ordered then $Merge(s, t, B)$ is B ordered. This is, surprise, another simple induction proof based on $|s| + |t|$.

Proof. We only have to work the case where $|s| > 0 and |t| > 0$ because otherwise it's trivial. So the interesting cases are where the parameters are $s = xq$ and $t = yz$. If the two suffixes are empty then it's easy. Suppose the property holds when $|q| + |z| = n$ and suppose $|q| + |z| = n + 1$. Simple - using lemma1. □

And note merge takes $|s| + |t|$ steps. Now is where merge sort gets cool. We think of our sequence of elements as a sequence of sequences and then merge those recursively. Each pass reduces the number of sequences by 1/2 so there are $\log_2(|q|)$ steps at most.

Sequences of sequences are a little weird so let's look at them in a bit more detail. To distinguish between sequence of atoms or scalars and sequences of sequences, I'll use capital letters for the latter. A sequence $Q = \langle q \rangle$ is a single element sequence with element q so $Q(1) = q$. A sequence $T = \langle \langle x \rangle \rangle$ has a single element, the single element sequence $\langle x \rangle$ so $T(1) = \langle x \rangle$ and $(T(1))(1) = x$. The sequence e has no elements but the sequence $\langle e \rangle$ has a single element which is the empty sequence and $\langle e \rangle(1) = e$. Also $(q * s)(1) = q(1)$ but $(\langle q \rangle * s)(1) = q$.

Let $Seq(e) = \langle e \rangle$ and let $Seq(\langle x \rangle * q) = \langle \langle x \rangle \rangle * Seq(q)$. Note the first element of the new sequence is a single element sequence. Seq turns an n element sequence into a sequence of n single element sequences. This is really no work - we are just

looking at the same sequence in a different way. Now we need to recursively reduce the sequence of sequences.

$$MergeMap(e, B) = e$$
$$MergeMap(\langle q \rangle, B) = q$$
$$MergeMap(\langle q \rangle * \langle t \rangle * Z, B) = \langle merge(q, t, B) \rangle * MergeMap(Z, B)$$

The third equation tells us that a sequence of at least two sequence elements $S = \langle s \rangle * \langle q \rangle * Z$ is reduced to $z * MergeMap(Z, B)$ where $z = Merge(q, t, B)$.

If $|s| = n$ then the time complexity of $MergeMap(Seq(s), B)$ is $(log_2 n)n$.

4. Quicksort

Quicksort[2] is a desperately unintuitive algorithm as usually presented because the basic idea and the detailed fiddling with arrays are not separated. It's not at all clear that in most practice the array fiddling is much of an engineering optimization anymore — although it certainly was back in the 1950s when the algorithm was first discovered. I want to split the algorithm up into two parts: the actual sort and the in-place optimizations so it is a little more comprehensible[3].

The basic idea is pretty simple and is almost merge sort in reverse. Instead of splitting the sequence and merging sorted sequences together, Quicksort keeps sorting the sequence into components that are better and not better than some "pivot" element. That is, we pick some pivot element $s(p)$ and then roam through s, throwing the elements better than $s(p)$ to the front and the elements that are not to the back. Then we carry out the same process on each of the split off sequences. The first effort is crude. The key part is to filter a sequence using B and some pivot value x.

Suppose $quickL(s, x, B)$ reduces s to only those elements that are better than x by B.

$$quickL(s, x, B) = \begin{cases} e & \text{if } s = e \\ s(1) * quickL(cut(s, 1), x, B) & \text{if } B(s(1), x) \\ quickL(cut(s, 1), x, B) & \text{otherwise} \end{cases}$$

Then $quickR(s, x, B)$ tosses out the other elements of s

$$quickR(s, x, B) = \begin{cases} e & \text{if } s = e \\ s(1) * quickR(cut(s, 1), x, B) & \text{if } \neg B(s(1), x) \\ quickR(cut(s, 1), x, B) & \text{otherwise} \end{cases}$$

$$quick(s, j, B) = \begin{cases} e & \text{if } s = e \text{ or } j > |s| \\ quick(quickL(cut(s, j), s(j), B), 1) & \\ *\langle s(j) \rangle & \\ *quick(quickR(cut(s, j), s(j), B), 1) & \text{otherwise} \end{cases}$$

[2]Quicksort is an example of how important naming is in computer science. Obviously, Quicksort is "quicker" than mergesort or at least it sounds quicker.

[3]The treatment here is similar to what is found in Haskell quicksort examples, but does not require category theory. The reader can take that as positive or negative, depending on personality.

The weakness of Quicksort is readily apparent from this definition: if the pivots divide sequences into unbalanced parts there can be up to n recursive applications instead of $\log_2 n$. Suppose the original sequence has only elements of one value, for example. In that case either $quickR$ or $quickL$ will be mostly empty. Or suppose the sequence is already ordered by B. The first problem can be cured by putting all elements matching the pivot value in the center.

$$quickC(s,x,B) = \begin{cases} e & \text{if } s = e \\ s(1) * quickC(cut(s,1),x,B) & \text{if } s(1) = x \\ quickC(cut(s,1),x,B) & \text{otherwise} \end{cases}$$

Then we have to discard pivots from the left and right.

$$quickL'(s,x,B) = \begin{cases} e & \text{if } s = e \\ s(1) * quickL'(cut(s,1),x,B) & \text{if } s(1) \neq x \text{ and } B(x,s(1)) \\ quickL'(cut(s,1),x,B) & \text{otherwise} \end{cases}$$

$$quickR'(s,x,B) = \begin{cases} e & \text{if } s = e \\ s(1) * quickR'(cut(s,1),x,B) & \text{if } s(1) \neq x \text{ and } \neg B(x,s(1)) \\ quickR'(cut(s,1),x,B) & \text{otherwise} \end{cases}$$

$$quick(s,j,B) = \begin{cases} e & \text{if } s = e \text{ or } j > |s| \\ quick(quickL'(cut(s,j),s(j),B),1) & \\ *quickC(s,x,B) & \\ *quick(quickR'(cut(s,j),s(j),B),1) & \text{otherwise} \end{cases}$$

collects all matching values in center. There's still the problem of a sorted sequence and questions about how to pick pivots. I've picked the first element, which is one common method, but there's nothing sacred about it. To prove this second version of quicksort works, we can extend the first lemma to sequences and then show that each of the three segments are ordered by B.

Lemma 4.1. *If q and s are ordered by B and $q \neq e$ and $s \neq e$ and $B(q(|q|), s(1))$ then $q * s$ is ordered by B.*

Proof. By induction on $|q|$ from 1 which simply needs lemma 1.1. Supposing the property for q where $|q| = n$, suppose $q =< x > *r$ where $|r| = n$. Then we know $r * s$ is ordered by B and $(< x > *q) * s =< x > (q * s)$ so using lemma 1.1 again we are done. □

Now all we have to prove is that the three subsequences are ordered by B. The center sequence consists only of the first pivot, so it's ordered. The other two are, as usual, easy to cover using recursion on sequences. (more later).

5. Appendix

Definition 5.1.

$$Best(s,B) = BestN(s,B,1,s(1))$$

$$BestN(s,B,n,x) = \begin{cases} x & \text{if } n \geq |s| \\ BestN(s,B,n+1,x) & \text{if } B(x,s(n+1)) \\ BestN(s,B,n+1,s(n+1)) & \text{otherwise} \end{cases}$$

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