Lecture 18: Thinking about Memory Usage

11:00 AM, Mar 3, 2021

Contents

These notes are partial, to help those attending lab this afternoon. More complete notes will be posted later today.

1 In-place Quicksort

What if we had to sort in constant space (so no creating additional lists as you go)? Instead, you would have to sort within the array you already have, somehow moving elements around within the array. Such algorithms are called *in-place* algorithms.

For today's demonstration we will use one of the most widely-used algorithms: *quicksort*. We'll get the intuition of quicksort by first looking at a functional implementation, then we'll do an in-place version using arrays.

1.1 Quicksort, functional implementation

The idea of quicksort is straightforward: given a list to sort, we choose any one element, then partition the remaining elements into two pieces: those smaller than the pivot and those larger than the pivot. We sort the pieces recursively, then append the sorted-smaller items, then the pivot, then the sorted larger items. Here's the code in Racket:

```
(define (quicksort L)
  (cond
   [(empty? L) empty]
   [else (append (quicksort (smaller-than (first L) (rest L)))
                 (list (first L))
                 (quicksort (larger-than (first L) (rest L))))]))

(define (smaller-than x L)
  (filter (lambda (y) (< y x)) L))

(define (larger-than x L)
  (filter (lambda (y) (>= y x)) L))
```

1.2 Quicksort, in-place version

Today we will discuss an implementation of this very same algorithm, but using mutable arrays rather than immutable lists. That is, we won't create a new array with each recursive call; instead we will continually modify the same array, until it is sorted.

Here is an example of quicksorting an array, in-place:

- Initial Array:

33	157	18	155	32	17	51

- Step 1: Choose 51 as the pivot element.
- Step 2: Move the elements less than 51 to the left part of the array, and move the elements greater than or equal to 51 to the right part of the array:

33	18	32	17	155	157	51
L	L	L	L	R	R	pivot

- Step 3: Put the pivot element in between:

33	18	32	17	51	157	155
L	L	L	L	pivot	R	R

  and then recursively sort the left and right parts of the array:

17	18	32	33	51	155	157
L	L	L	L	pivot	R	R

- Final (Sorted) Array:

17	18	32	33	51	155	157

Interestingly, when we sort an array in place (and, more generally, when you do any sort of in-place operation), we need not return anything. The effect of quicksorting an array in place is not to produce a new sorted array; rather, it is to modify the given unsorted array.

The most interesting part of in-place quicksort implementation is the partitioning scheme. How would we go about moving all the elements less than the pivot to the left part of the array, and all the elements greater than or equal to the pivot to the right part of the array, in place (and efficiently)? There are a couple of approaches; we will show one of them.

1.3 Partitioning

Use two indices: `left`, which is initialized to index the first element of the array, and `right`, which is initialized to index the last element of the array. The array is then traversed (*sans* the pivot element) by moving the left index to the right and the right index to the left, while maintaining the following properties:

- All data stored at indices less than `left` have values less than the pivot
- All data stored at indices greater than or equal to `right` have values greater than or equal to the pivot.

Here is a simple iterative algorithm that does this:

1. Increment `left` if it indexes a cell whose value is less than the pivot value, otherwise leave `left` in place.
2. Decrement `right` if it indexes a cell whose value is greater than or equal to the pivot, otherwise leave `right` in place.
3. If `a[left]` $\geq$ `pivot` $>$ `a[right]`, then swap `a[left]` and `a[right]`, and then increment `left` and decrement `right`.
4. Repeat until `left` $>$ `right`, at which point the entire array has been processed.

For example, consider the following array in which the pivot is 33. Initially, `left` indexes 51 and `right` indexes 32.

51	31	155	18	181	157	32
left						right

Because $51 \geq 33 > 32$, swap 51 and 32, then increment `left` and decrement `right`.

32	31	155	18	181	157	51
	left				right	

Now, since $31 < 33$, increment `left`, and since $157 \geq 33$, decrement `right`.

32	31	155	18	181	157	51
		left		right		

Since $155 > 33$, `left` cannot be further incremented, unless there is first a swap. But `right` can be decremented, since $181 \geq 33$.

32	31	155	18	181	157	51
		left	right			

And now there should be a swap, since $155 \geq 33 > 18$. After swapping, increment `left` ($18 < 33$) and decrement `right` ($155 > 33$).

32	31	18	155	181	157	51
		right	left			

At this point, `left` exceeds `right`, meaning the entire array has been processed. Observe: the values to the left of `left` are less than the pivot, while values to the right of `right` are greater than or equal to the pivot.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.