APS105: Quicksort

Daniel Zingaro

March 28, 2010

1 Why Quicksort?

We have covered three sorts to this point: selection, insertion, and bubble. They have a few things in common:

- They are all iterative. They use loops rather than recursion.
- They are all similar in terms of efficiency. That is, one of them is not conclusively better than the others. (But as we have seen in class, there are some special cases where one sort does substantially better than the others; for example, insertion sort on an already-sorted list.)
- They are all extremely slow. If you tried sorting millions of elements using these sorting algorithms, it would take forever. They are OK for small amounts of data, but degenerate quickly once a certain threshold is reached.

By contrast, quicksort is a recursive sorting technique that is extremely fast. It's more complicated to implement, and a little easier to get it wrong, but it's probably worth it if you're sorting a lot of data.

2 Idea Behind Quicksort

We can demonstrate the powerful idea of quicksort with an example. Let's say that we're sorting the following list of numbers:

```
8 9 1 6 4 2 3
```

Assume that we choose the value 3, and reorder the array so that all values less than 3 come first, followed by all values that are at least 3. There are many ways to rearrange the array; the most important thing is that values less than 3 are all found before any values that are at least 3. Here is one way to rearrange the array so that this is satisfied:

```
1 3 2 8 9 6 4
```

Rearranging the array in this way is referred to as **partitioning** the array.

Exercise 1: Another Rearrangement

Give another valid way of partitioning the array, again choosing value 3. That is, put all values less than 3 first, then put the values that are at least 3. There are lots of ways to do this!

Now, it is helpful to visualize our array as consisting of two separate pieces:

```
1 3 2 ::: 8 9 6 4
```

The essence of quicksort can be captured by the following idea.

If we can independently sort the left piece of the array, and then sort the right piece of the array, the entire array would be sorted.

That is, we can first sort:
`1 3 2`
and then sort:
`8 9 6 4`
at which point the entire array will be sorted.

How should we sort each of these two pieces? Keep in mind that what we have done here is take one big sorting problem (with seven elements) and produce two smaller, simpler sorting problems (one with three elements, one with four). We can therefore apply exactly the same idea to each subproblem. That is, to sort the left half:
`1 3 2`
we could partition this piece of the array around some value, to yield two further subproblems, such as: `1 2 ::: 3`
Eventually, our subproblems (such as the one-element array `3` here) will be so simple that we can solve them directly, without breaking them down any further. Once we have solved all subproblems for the left piece of the array, we proceed to break down: `8 9 6 4` into smaller sorting problems, until they too are so simple that further partitioning is not required.

It is important to understand that sorting the left half of the array, and sorting the right half of the array, are totally independent. Why? The reason is that when we partition an array, we have split the array into small values and large values. None of the small values can possibly go where any of the large values are, and vice versa. If we independently sort these two sides of the array and then think about pasting them together, the whole array will come out sorted.

Think about this in terms of sorting people. Choose some height, like 5 ft 5 in, put all people less than this height in one group, and all people more than this height in another group. Now, have the two groups put themselves in order based on height. You'll get two rows of people and, if you join the two rows, everyone will be ordered by height.

3 Partitioning an Array

We must be able to partition an array around some value `pivot`, so that all values less than `pivot` come first, followed by all other values. There are many different approaches for doing this. (For example, in the first exercise of this handout, you used whatever strategy you liked to partition the array.) I will expect you to be familiar with the method below (not the one in the textbook).

(The word **pivot** simply refers to the value that partitions the array. The usual terminology in quicksort discussions is to refer to this as the pivot value. For example, if we want all elements less than 9 to come first and then everything else, we say that 9 is the pivot value.)

The algorithm for partitioning an array processes the array from left to right. It uses two indices to remember the boundary between elements that are less than the pivot, those that are greater than or equal to the pivot, and those elements that have not yet been processed.

- We'll use variable `i` to keep track of the border between elements that are `< pivot` and those that are `>= pivot`
- We'll use variable `j` to keep track of the start of the elements we have not yet processed

`< i`	between `i` and `j`	`>= j`
`< pivot`	`>= pivot`	unprocessed

Here is the algorithm for partitioning an array `a` around a pivot:

- We begin with `i` and `j` at the left end of the part of the array we are sorting
- Then, at each step, we have two possibilities
  - If `a[j] >= pivot`, we simply increment `j`
  - Otherwise, `a[j] < pivot`, so we cannot just increment `j`
    - We swap `a[i]` and `a[j]`, and then
    - Increment both `i` and `j`

4 Partition Code

To complement my pseudocode description, here is code that implements the partition procedure. This function partitions the array between indices `low` and `high`, and returns the index of the first element that is `>= pivot`.

```
int partition (int low, int high, int pivot, int a[]) {
  int i = low, j = low;
  while (j <= high) {
    if (a[j] < pivot) {
      swap (a, i, j);
      i++;
    }
    j++;
  }
  return i;
}
```

5 Partition Example

Here is a worked example of how the above partition algorithm would operate on an array. Please follow along with the code above and/or its preceding description.

- Consider array: $[12, 30, 25, 8, 4, 9, 15, 13]$
- Let's partition around pivot 13
- Initial state: `i = 0`, `j = 0`
- Is `a[0] < pivot`? Yes — "swap" `a[0]` with itself and increment `i` and `j`
- New state: `i = 1`, `j = 1`
- Is `a[1] < pivot`? No — increment `j`
- New state: `i = 1`, `j = 2`
- Is `a[2] < pivot`? No — increment `j`
- New state: `i = 1`, `j = 3`
- Is `a[3] < pivot`? Yes — perform swap; increment both
- New array: $[12, 8, 25, 30, 4, 9, 15, 13]$
- New state: `i = 2`, `j = 4`
- Is `a[4] < pivot`? Yes — perform swap; increment both
- New array: $[12, 8, 4, 30, 25, 9, 15, 13]$
- New state: `i = 3`, `j = 5`

- Is `a[5] < pivot`? Yes — perform swap; increment both
- New array: $[12, 8, 4, 9, 25, 30, 15, 13]$
- New state: `i = 4`, `j = 6`
- Is `a[6] < pivot`? No — increment `j`
- New state: `i = 4`, `j = 7`
- Is `a[7] < pivot`? No — increment `j`
- New state: `i = 4`, `j = 8`

So, the final array, after partitioning around pivot value `13` is:
$[12, 8, 4, 9, 25, 30, 15, 13]$
All values less than 13 come first, followed by all values that are at least 13. We are going to come back to this shortly and show how to use quicksort on both of these pieces so that the entire array is sorted.

Exercise 2

Partition the array $[12, 8, 4, 9]$ around pivot value 9.

Partition the array $[12, 8, 4, 9]$ around pivot value 5. (Don't get confused because 5 is not in the array. Continue to use the partition algorithm to put values less than 5 before any values that are greater than or equal to 5.)

6 Complete Quicksort Algorithm

Now, we have a way of partitioning an array so that all values less than the pivot are placed before any value that is $\geq$ the pivot. We use this to write the quicksort algorithm, as follows:

- If we are asked to sort a piece of an array with zero or one elements, we do nothing. An empty array, or an array with one element, is already sorted. This is the base case.
- Otherwise, we are sorting a piece of an array that has at least two elements. In this case:
  - (1) Partition the array around a pivot value to divide the array into those values that are less than the chosen pivot, and those values that are at least as large as the pivot
  - (2) Swap the pivot value so that it sits between both of these sections of the array
  - (3) Recursively call quicksort on the values that are smaller than the pivot, and
  - (4) Recursively call quicksort on the values that are at least as large as the pivot

The chosen pivot can be any value; the code below chooses the rightmost value on each recursive call.

```
void quicksort (int low, int high, int a[]) {
  if (low < high) {
    int pivot = a[high];
    int i = partition (low, high - 1, pivot, a);
    swap (a, i, high);
    quicksort (low, i - 1, a);
    quicksort (i + 1, high, a);
  }
}
```

Exercise 3

The choice of pivot ends up being critical to the efficiency of quicksort. If we choose a pivot that evenly divides the array into two halves, we are happy. If instead we choose a

pivot that divides the array into one huge piece and one small piece, we have not made as much progress. Assuming we use the rightmost element as the pivot value, give an example of an array that will result in a poor pivot being chosen on each recursive call, and hence poor performance for quicksort.

7 Continuing the Example

7.1 Sorting the Entire Array

Let's go back to our partitioned array in section 5. We pivoted around value 13, and ended up with:
$[12, 8, 4, 9, 25, 30, 15, 13]$
To continue the sort, we will now use the ideas of section 6.

Our original array contained more than one element, which is why we partitioned the array around a pivot in the first place. In other words, we have just completed step (1) of the recursive case for quicksort, and now must complete steps (2), (3) and (4) in order to sort the entire array.

Step (2) — swapping the pivot in between those elements that are less and those elements that are at least as large – gives:
$[12, 8, 4, 9, 13, 30, 15, 25]$

Step (3) says that we should now call quicksort on the subarray:
$[12, 8, 4, 9]$
In other words, we stop what we're doing on the entire array, and start the quicksort algorithm from the beginning on this smaller array. When we're finished with this smaller subarray, we'll come back here and do step (4) on the whole array, at which point we'll be finished.

7.2 Sorting $[12, 8, 4, 9]$

We now run quicksort on:
$[12, 8, 4, 9]$
Look back at section 6. The array contains more than one element, so we are not done; instead, we carry out the four steps of the recursive case: (1) partition, (2) swap pivot into the middle, (3) quicksort small values, (4) quicksort large values.

(1) We choose 9 as the element around which to pivot this subarray. Running the partition algorithm from section 5, we get:
$[4, 8, 12, 9]$

(2) Swapping the pivot between the small and large elements gives:
$[4, 8, 9, 12]$

(3) We make a recursive call of quicksort on the subarray:
$[4, 8]$
which causes us to suspend our sorting of $[4, 8, 9, 12]$ right here, sort $[4, 8]$, and then come back and carry out step (4) to sort the elements to the right of 9.

7.3 Sorting $[4, 8]$

Does this subarray contain more than one element? Yes; so we must execute the recursive case of quicksort. Again, four steps:

(1) Partition around pivot element 8. This causes no change:
$[4, 8]$
(2) Swap the pivot into place. Again, no change:
$[4, 8]$
(3) Recursively call quicksort on those elements smaller than the pivot. That is, we call quicksort recursively on:
$[4]$

7.4 Sorting $[4]$

We have reached a base case: an array with one element. We do nothing.

7.5 Returning to Sorting $[4, 8]$

At this level, we have just carried out step (3). We must now carry out step (4) — sorting all elements to the right of 8. There are no such elements, so when we make this recursive call, quicksort will immediately return (the situation is the same as when we asked quicksort to sort [4]).

7.6 Returning to Sorting $[12, 8, 4, 9]$

At this level, we have just finished step (3), with the following array:
$[4, 8, 9, 12]$
We must now do step (4). Recall that our pivot on this level is 9, and step (4) involves quicksorting those elements to the right of 9. We will therefore call quicksort recursively on subarray [12]. Since this subarray contains only one element, quicksort will return immediately.

7.7 Back to the Top

Remember our original partition, where we pivoted around value 13? Well, we have now finished steps (1), (2) and (3) on that array. We can now continue with step (4) on this entire array, picking up from where we were interrupted at the end of section 7.1.

Our current array is:
$[4, 8, 9, 12, 13, 30, 15, 25]$
and we are going to call quicksort recursively on:
$[30, 15, 25]$
The pattern is hopefully clear. We are not asking to sort an array of less than two elements, so the recursive case of quicksort will be evoked. We will therefore carry out the four steps of the recursive case: pivot around 25, swap 25 in between both sections, sort values < 25, sort values ≥ 25 ...

Exercise 4

Finish the quicksort!