Lecture 17: Quicksort

Doina Precup
With many thanks to Prakash Panagaden and Mathieu Blanchette

February 12, 2014

So far, we discussed two sorting algorithms: selection sort, which is $O(n^2)$, and merge sort, which is $O(n \log n)$. However, neither of them is used when you have to sort databases with millions of examples. Instead, the most popular solution in practice is quicksort.

1 Quicksort algorithm

The basic idea of the algorithm is that of divide and conquer: we want to partition the array into two parts, sort each one (recursively, of course), and then get the result. But unlike merge sort, where getting the result means the extra work of merging, here we want to just *have* the result when the recursive calls are finished. The algorithm will work **in place**, which means that no new array or data structure is created. All the work takes place in the same array, with only $O(1)$ temporary extra memory being allocated. This is a big advantage if you want to handle lots of data.

The pseudocode of the algorithm is as follows:

Algorithm qsort(a, p, r)
Input: An array a and two indices p, r
Output: The portion of the array between p and r will be sorted as a side effect of the algorithm
if $p < r$ **then**
  int $q \leftarrow$ partition(a, p, r)
  qsort(a, p, q)
  qsort$(a, q + 1, r)$

The first call will be qsort$(a, 0, n - 1)$ where n is the number of elements in the array. All the interesting work is actually in the partition method. Its goal is to move around the elements such that all the ones *before* the index q are less than or equal to some quantity x, while all the ones *after* index q are greater than x. This means that if we sort the two halves of the array independently, the resulting array will also be all sorted (a fact that we will prove shortly).

The pseudocode for the partition algorithm is as follows:

Algorithm partition(a,p,r)
Input: An array a and indices p and r within the array

Output: An integer j between p and r such that all elements with indices $\leq j$ are smaller than or equal to all the elements with indices $> j$
$x \leftarrow a[p]$ //this will be the quantity used to partition the array
$i \leftarrow p - 1$ //two indices i,j will determine which elements to swap
$j \leftarrow r + 1$
while true do
  repeat $i \leftarrow i + 1$ **until** ($i > r$ **or** $a[i] \geq x$)
  repeat $j \leftarrow j - 1$ **until** ($j < p$ **or** $a[j] \leq x$)
  if $i < j$ **then** swap$(a[i], a[j])$
  else return j

To see what is going on, consider the example of the following array:

5	3	2	8	1	2	7	6

The value of x will be 5, so we will partition the array into a sub-array that has elements ≤ 5 and one that has elements ≥ 5. We start with the two indices ("fingers") outside of the array and we move them towards the middle of the array, until we get to the following situation:

5	3	8	2	1	2	7	6
i					j		

At this point, we swap the respective elements and the array becomes:

2	3	8	2	1	5	7	6
i					j		

The indices continue moving until they get to the following position:

2	3	8	2	1	5	7	6
		i		j			

At this point, we swap again and the array becomes:

2	3	1	2	8	5	7	6
		i		j			

The indices continue moving until the following position:

2	3	1	2	8	5	7	6
			j	i			

Here the indices have crossed so we stop, and j is returned. Note that all elements with indices $\leq j$ are $\leq x = 5$, and all the elements to the right of j are ≥ 5. So if these two halves of the array are now sorted independently, no merging will be necessary in the end.

2 Proof of correctness

Now let's try to prove (by induction, of course) that the algorithm works.

Base case: Consider an array of just one element. Quicksort will not do anything, as it should (the array is sorted)

Induction step: We assume that the recursive calls work correctly (put your faith in the recursion!). So after these, the two halves of the array are sorted. Assume also that the partition method works correctly (we will show this below). Now consider two elements of a with indices $i < j$.

Are they in the correct order?

- If $i < j \leq q$ then $a[i] \leq a[j]$ by the induction hypothesis
- If $q < i < j$ then $a[i] \leq a[j]$ by the induction hypothesis
- If $i \leq q < j$ or $i < q \leq j$ then, by using the fact that partition works correctly, we know that $a[i] \leq x \leq a[j]$.

So we checked that any two elements must be in the correct order.

Now let's prove that partition works correctly. First, we need to check that i and j do not go out of bounds. To do this, note that we start outside and the first move is correct. Second, there will be at least one element $\leq x$ that is put in the second half of the array (this is the first element). So i, which always increases, will have to hit this element at some point and it will stop, before it goes out of bounds (even without the first condition). For j we can make a similar argument; either an element $\leq x$ exists (or was swapped) in the first half, in which case j will stop on it, or if not, j will stop on the first element of the array (and will not go out of bounds).

Second, we will show that the algorithm has a **loop invariant**: whenever we go through the while loop, it is true that all elements with indices $\leq i$ are $\leq x$ and all the ones with indices $> j$ are $\geq x$. Since this is true all the time, it will also be true when the algorithm terminates, which will conclude our proof. This property is clearly true based on the two loops in which i and j move. When i and j stop, the property is violated for those respective elements, so it is fixed with a swap.

3 Complexity

The partition algorithm is $O(n)$ where $n = r - p + 1$ is the number of elements under scrutiny. To see this, note that the indices move by one every time, and never return to a previous position, so at most they can "migrate" from one end to the other.

For the whole algorithm, assuming that the partition algorithm cuts the array into a side of size k and one of size $n - k$, we have the following recurrence:

$$T(n) = T_{partition}(n) + T(k) + T(n - k)$$

As seen before, $T_{partition}(n) \in O(n)$ so we have:

$$T(n) = cn + T(k) + T(n - k)$$

The worst case of the algorithm is when the recursive calls split the array into 1 element and $n - 1$ elements at every step. In this case, we have:

$$\begin{aligned} T(n) &= cn + T(1) + T(n-1) \\ &= cn + T(1) + c(n-1) + T(1) + T(n-2) = c(n + (n-1)) + 2T(1) + T(n-2) \\ &= \dots \\ &= c(n + (n-1) + \dots + 2 + 1) + nT(1) \end{aligned}$$

As we showed when discussing selection sort, the first term is $O(n^2)$. the second term is $O(n)$. So, in this case, the complexity is $O(n^2)$/ This case is achieved when the array is already sorted. Of course, this is easy to detect in practice, so the worst case can be avoided.

In the best case, each recursive call would split the array roughly in half, yielding the recurrence:

$$T(n) = cn + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) = cn + 2T\left(\frac{n}{2}\right).$$

As seen when we analyzed merge sort, this yields $O(n \log_2 n)$.

In the "average" case, we would expect that the array is divided into some fraction, but not necessarily half. To do the analysis, we need to a=make some assumption about how k looks on average. For example, suppose that k is uniformly distributed between 1 and $(n-1)$ (i.e., all values of k are equally likely). Then, on average, we have:

$$T(n) = cn + \frac{1}{n}\sum_{k=1}^{n}[T(k) + T(n-k)]$$

In COMP-251, you will show that this yields $2n \ln n$, or $1.4n \log_2 n$. In other words, in the average case, quick sort is only 40% slower than in the best case

Quicksort has been analyzed a lot, and good approaches exist to **randomize** the choice of the element by which to partition, in order to achieve good performance. A simple approach is to "shuffle" elements by picking pairs at random and swapping them (similarly to how you shuffle a deck of cards). Note that as long as we do this a number of times that is *constant* with respect to the size of the array, this shuffling will not affect O(), even if this number is very big. An alternative approach is to partition not by the first element in the array, but by the median of a subset of elements, in order to avoid the worst-case for arrays that are sorted or nearly-sorted.