Lecture 8: Quicksort Analysis

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

Mergesort limitations?

Mergesort divides the input array A[*low*..*high*] into two pieces of similar size without looking at the contents. We get pieces A[*low*..*mid*] and A[(*mid* + 1)..*high*] merely by using the location *mid* $\leftarrow$ (*low* + *high*) / 2.

But, after sorting the two pieces, we have to merge them, costing us an extra n operations.

Can we arrange it so that we don't have to merge at the end? This is the strategy that Quicksort uses.

Quicksort intuition

Quicksort is based on the following subtle idea. In a sorted **output** array, the *value* of the middle key also divides the array into two pieces of similar size: all keys to the left are smaller and all keys to the right are bigger. Can we use this median value to help with the sorting?

Well, if we had an easy way to pick some x so that about half the values in the input array A[*low*..*high*] are $\leq$ x and the rest $\geq$ x, then we could stick x into cell A[*mid*] and rearrange the entries so that smaller elements are put on the left of x and larger entries are put on the right of x.

This would not yet be the sorted output array, but we could recursively apply the same process to the pieces A[*low*..*mid*] and A[$(mid + 1)$..*high*]. The recursive descent stops when we reach a piece of length ≤ 1, because an array of length 0 or 1 is sorted already.

Quicksort algorithm

```
Quicksort(A, low, high)
1: if A[low..high] has 2 or more elements then
2:     Partition(A, low, high), returning the index mid {everything $\geq$ the chosen pivot
       is to the right of mid }
3:     Quicksort(A, low, mid)
4:     Quicksort(A, mid + 1, high)
```

To sort A[0..(*n* - 1)], the initial call is Quicksort(A, 0, n - 1).

Quicksort provides the control structure for the sort, but the work of comparing and rearranging the keys is done by the (non-recursive) Partition algorithm.

Partition

```
Partition(A, *low*, *high*)
x $\leftarrow$ A[*low*]
i $=$ *low* - 1
j $=$ *high* $+$ 1
**loop**
    **repeat**
        j $\leftarrow$ j - 1
    **until** A[j] $\leq$ x
    **repeat**
        i $\leftarrow$ i $+$ 1
    **until** A[i] $\geq$ x
    **if** i $<$ j **then**
        swap A[i] and A[j]
    **else**
        return j
```

Partition analysis

To analyse Partition, we can use a simple counting argument as we did for Merge:

- i starts at low-1
- j starts at high+1
- each time through the outer loop (line 4), j is decremented at least once (line 6) and i is incremented at least once (line 9) and we do a comparison (lines 7 and 10) for each decrement or increment.
- Then either a swap at line 12 or a return at line 14.
- Either way, we keep going until we hit the return at line 14 which we are guaranteed to do since eventually $i \geq j$.
- Because we are only adding and subtracting 1, $i \geq j$ after O(high-low) operations.
- So Partition is $O(n)$.

Quicksort analysis

Let's call the time function for Quicksort, T, then:

$$T(1) = 1$$
$$T(high - low) = high - low + T(mid - low + 1) + T(high - mid)$$
$$T(n) = n + T(mid - low + 1) + T(high - mid)$$

Let's have a look at two cases:

1. Worst case: mid = low
2. Best case: mid = (low+high)/2

Quicksort - worst case

When mid = low, the time function for Quicksort becomes:

$$T(1) = 1$$
$$T(n) = n + T(1) + T(n-1)$$
$$T(n) = 1 + n + T(n-1)$$

Substitution and iteration

Applying subsitution and iteration:

$$\begin{aligned}
T(n) &= 1 + n + T(n-1) \\
&= 1 + n + (1 + n - 1 + T(n-2)) \\
&= 1 + 2n + T(n-2) \\
&= 1 + 2n + (1 + n - 2 + T(n-3)) \\
&= 0 + 3n + T(n-3) \\
&= 0 + 3n + (1 + n - 3 + T(n-4)) \\
&= -2 + 4n + T(n-4) \\
&= 1 - (1 + 2 + 3 + \cdots + k - 2) + kn + T(n-k) \\
? &= 1 - \frac{(k-1)(k-2)}{2} + kn + T(n-k) \\
? &= \frac{k(2n-k+3)}{2} + T(n-k)
\end{aligned}$$

Substitution and iteration

Set $k = n - 1$ to get T(1) on the right:

$$\begin{aligned} T(n) ?&= \frac{(n-1)(2n-(n-1)+3)}{2} + 1 \\ ?&= \frac{(n-1)(n+4)}{2} + 1 \end{aligned}$$

Which we can prove by induction (left as an exercise).

So in the worst case Quicksort is $O(n^2)$. That's not good.

The worst case occurs exactly when Quicksort is run on an already sorted array.

We can avoid the worst case by choosing a better pivot. There are a few options, but the most common are: choose a random element as the pivot; choose the median of 3 elements (e.g. first, middle, last) as the pivot.

Quicksort best case

What about when mid = (low+high)/2? In that case, mid splits the data in two equal parts, so the time function is:

$$T(1) = 1$$
$$T(n) = n + T(n/2) + T(n/2)$$

which is the same as for Mergesort. So in the best case, Quicksort is $O(n \log n + n)$.

Uneven, but not worst case

What if we're always somewhere between the worst and best case? Is Quicksort good or bad on average?

Consider a pivot that always gives us a 90%/10% split of the data (sounds pretty bad). In that case, the time function is:

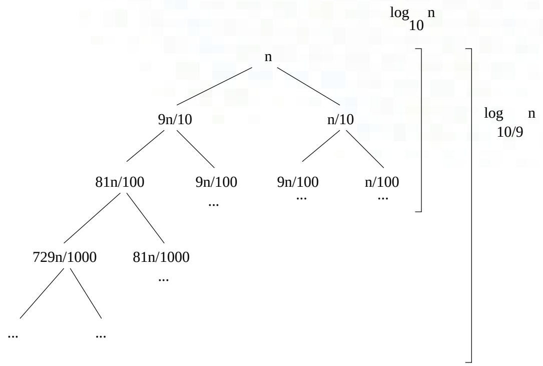
$$T(n) = T(9n/10) + T(n/10) + n$$

It is difficult to solve this problem via iteration and substition, but we can use a recursion tree instead.

Quicksort recursion tree

Since each layer of the tree does at most n things and there are at most $\log_{10/9} n$ layers, total time is still $O(n \log n)$.

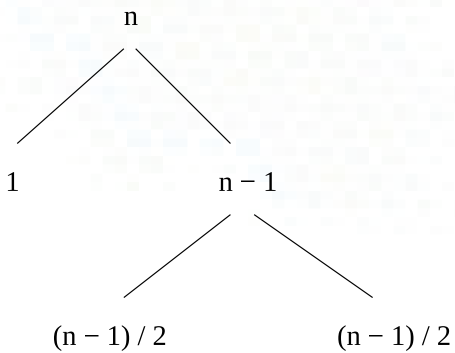
This is true for any split that **keeps the same proportion**, even a 99 to 1 split. (But note that a split of $n - 1$ to 1 is not of constant proportion, which is why we got $O(n^2)$ in that case.)



Proportional splits

What about alternating a perfect split with a bad split?

In this case, the tree is at most twice as high as for all perfect splits, so still $O(n \log n + n)$.



Quicksort versus Mergesort

- Mergesort is $O(n \log n)$ in the worst case.
- Quicksort is $O(n^2)$ in the worst case, but $O(n \log n)$ for most cases given a good implementation.
- Mergesort copies all the data into a new array during Merge, so requires an extra n words of memory.
- Quicksort can sort in place.
- In place Mergesorts are possible, but these are then $O(n^2)$ in the worst case.

Relevant parts of the textbook

Chapter 7 discusses Quicksort and the performance of Quicksort is discussed in Section 7.2. Section 7.4 does a more rigorous analysis of Quicksort.

We haven't covered Heapsort this semester, but that is discussed in some detail in Chapter 6.

Also of interest is Chapter 8 which discusses sorting in linear time. Mergesort and Heapsort are essentially optimal for comparison sorts, so to sort in linear time, we must give up doing comparisons, and this usually means we have to either make other assumptions, or use more memory.