ICS 161: Design and Analysis of Algorithms Lecture notes for January 25, 1996

Selection and order statistics

Statistics refers to methods for combining a large amount of data (such as the scores of the whole class on a homework) into a single number of small set of numbers that give an overall flavor of the data. The phrase *order statistics* refers to statistical methods that depend only on the *ordering* of the data and not on its numerical values. So, for instance, the average of the data, while easy to compute and very important as an estimate of a central value, is not an order statistic. The **mode** (most commonly occurring value) also does not depend on ordering, although the most efficient methods for computing it in a comparison-based model involve sorting algorithms. The most commonly used order statistic is the **median**, the value in the middle position in the sorted order of the values. Again we can get the median easily in O(n log n) time via sorting, but maybe it's possible to do better? We'll see that the answer is yes, O(n).

We will solve the median recursively. Sometimes, especially in relation to recursive algorithms, it is easier to solve a more general problem than the one we started out with. (Making the problem more difficult paradoxically makes it easier to solve.) This is true because if the problem is more general it may be easier for us to find recursive subproblems that help lead us to a solution.

The more general problem we solve is **selection**: given a list of n items, and a number k between 1 and n, find the item that would be kth if we sorted the list. The median is the special case of this for which k=n/2.

We'll see two algorithms for this general problem, a randomized one based on quicksort ("quickselect") and a deterministic one. The randomized one is easier to understand & better in practice so we'll do it first.

But before we get to it, let's warm up with some cases of selection that don't have much to do with medians (because k is very far from n/2).

Second-best search

If k=1, the selection problem is trivial: just select the minimum element. As usual we maintain a value x that is the minimum seen so far, and compare it against each successive value, updating it when something smaller is seen.

```
min(L)
{
x = L[1]
for (i = 2; i <= n; i++)
    if (L[i] < x) x = L[i]
return x
}
```

What if you want to select the second best?

One possibility: Follow the same general strategy, but modify min(L) to keep two values, the best and second best seen so far. We need only compare each new value against the second best, to tell whether it is in the top two, but then if we discover that a new value is one of the top two so far we need to tell whether it's best or second best.

```
second(L)
{
x = L[1]
y = L[2]
if (y < x) switch x, y
for (i = 3; i <= n; i++)
    if (L[i] < y) {
    y = L[i]
    if (y < x) switch x, y
    }
return y
}
```

Although this algorithm is pretty easy to come up with, some interesting behavior shows up when we try to analyze it.

- In the **worst case**, the list may be sorted in decreasing order, so each of the n-2 iterations of the loop performs 2 comparisons. The total is then 2n-3 comparisons.
- In the **average case** (assuming any permutation of L is equally likely) the first comparison in each iteration still always happens, but the second only happens when L[i] is one of the two smallest values among the first i. Each of the first i values is equally likely to be one of these two, so this is true with probability 2/i. The total expected number of times we make the second comparison is

  $$\sum_{i=3}^{n} 2/i = 2 \ln n + O(1)$$

  The ln is a natural logarithm. The sum (for i from 1 to n) of 1/i, known as the *harmonic series*, is ln n + O(1) (this can be proved using calculus, by comparing the sum to a similar integral). Therefore the total expected number of comparisons overall is n+O(log n).

This small increase over the n-1 comparisons needed to find the minimum gives us hope that we can perform selection faster than sorting.

Heapselect

We saw a randomized algorithm with n + O(log n) comparison expected. Can we get the same performance out of an unrandomized algorithm?

Think about basketball tournaments, involving n teams. We form a complete binary tree with n leaves; each internal node represents an elimination game. So at the bottom level, there are n/2 games, and the n/2 winners go on to a game at the next level of the tree. Assuming the better team always wins its game, the best team always wins all its games, and can be found as the winner of the last game.

(This could all easily be expressed in pseudo code. So far, it's just a complicated algorithm for finding a minimum or maximum, which has some practical advantages, namely that it's *parallel*

(many games can be played at once) and *fair* (in contrast, if we used algorithm min above, the teams placed earlier in L would have to play many more games and be at a big disadvantage).

Now, where in the tree could the second best team be? This team would always beat everyone except the eventual winner. But it must have lost once (since only the overall winner never loses). So it must have lost to the eventual winner. Therefore it's one of the log n teams that played the eventual winner and we can run another tournament algorithm among these values.

If we express this as an algorithm for finding the second best, it uses only n + ceil(log n) comparisons, even better than the average case algorithm above.

If you think about it, the elimination tournament described above is similar in some ways to a binary heap. And the process of finding the second best (by running through the teams that played the winner) is similar to the process of removing the minimum from a heap. We can therefore use heaps to extend idea to other small values of k:

```
heapselect(L,k)
{
heap H = heapify(L)
for (i = 1; i < k; i++) remove min(H)
return min(H)
}
```

The time is obviously O(n + k log n), so if k = O(n/log n), the result is O(n). Which is interesting, but still doesn't help for median finding.

Quick select

To solve the median problem, let's go back to the idea of using a sorting algorithm then finding the middle element of the sorted list. Specifically, look at quicksort:

```
quicksort(L)
{
pick x in L
partition L into L1<x, L2=x, L3>x
quicksort(L1)
quicksort(L3)
concatenate L1,L2,L3
}
```

We could have a selection algorithm that called quicksort explicitly before looking at the middle element. Instead let's put the "look at the middle element" line into the quicksort pseudocode:

```
select(L)
{
pick x in L
partition L into L1<x, L2=x, L3>x
quicksort(L1)
quicksort(L3)
concatenate L1,L2,L3
return kth element in concatenation
}
```

This is not a recursive algorithm itself since it does not call itself (although it does call quicksort, which is recursive). Just like quicksort, it takes time O(n log n).

But notice: if k is less than the length of L1, we will always return some object in L1. It doesn't matter whether we call quicksort(L3) or not, because the order of the elements in L3 doesn't make a difference. Similarly, if k is greater than the combined lengths of L1 and L2, we will always return some object in L3, and it doesn't matter whether we call quicksort on L1. In either case, we can save some time by only making one of the two recursive calls. If we find that the element to be returned is in L2, we can just immediately return x without making either recursive call. We can also save a little more time (not very much) by not doing the concatenation, instead directly looking at the right place in L1, L2, or L3.

```
select(L)
{
pick x in L
partition L into L1<x, L2=x, L3>x
if (k <= length(L1)) {
    quicksort(L1)
    return kth element in L1
} else if (k > length(L1)+length(L2)) {
    quicksort(L3)
    return (k-length(L1)-length(L2)) element in L3
} else return x
}
```

So far this is an improvement (it makes fewer calls to quicksort) but it is still an O(n log n) algorithm. One final observation, though, is that the code inside each if statement sorts some list and then returns some position in it. In other words, it solves exactly the same sort of selection problem we started with! And we could make the same improvements (of only doing one out of two recursive calls) to the two remaining calls to quicksort, simply by replacing these pieces of code by a recursive call to the selection routine.

```
quickselect(L,k)
{
pick x in L
partition L into L1<x, L2=x, L3>x
if (k <= length(L1))
    return quickselect(L1,k)
else if (k > length(L1)+length(L2))
    return quickselect(L3,k-length(L1)-length(L2))
else return x
}
```

Analysis of quickselect

Quickselect always makes a recursive call to one smaller problem. If we pretended it was always a problem of half the size, we would get a recurrence T(n)=O(n)+T(n/2)=O(n), but of course it's not. In the worst case, the recursive call can be to a problem with only one fewer element (L3 could be empty, and L2 could only have one element in it). This would instead give us a recurrence T(n)=O(n)+T(n-1)=O(n^2). So in the worst case this algorithm is very bad.

Instead we want to analyze the average case, which is much better. To perform the average case analysis rigorously, we would form a randomized recurrence with two parameters n,k. The worst case turns out to be when k=n/2, so to keep things simple, I'll just write a one-variable recurrence assuming that worst case. (This assumption only makes the analysis sloppier, but if it gives some bound f(n) you know it will at least not be worse than f(n) but it might actually be better.)

Now we always eliminate |L2|+min(|L1|,|L3|) objects from the list at each step. (except when we return x, in which case the algorithm terminates). Roughly, this min is equally likely to be any number from 1 to n/2. So the recursive call to quickselect is equally likely to involve a list of any size from n/2 to n. (If k were different from n/2, this would only make some smaller sizes more likely and some larger sizes less likely, which can be used to prove that k=n/2 really is the worst case.)

As usual with expected case analysis we get a recurrence involving a sum over possible choices of the probability of making that choice, multiplied by the time it would take if we made the choice. As before let's measure things in comparisons. The recurrence is:

$$T(n) = n-1 + \sum_{i=n/2}^{n} 2/n\ T(i)$$

How do we analyze something like this? We know it should come out to O(n), but we don't know what the constant factor should be.

If we knew the correct constant factor c, we would be able to prove inductively that T(n) is at most cn. The method would be simply to plug in ci for T(i) on the left side, grind through the sums, and verify that the result is at most cn. (One would also have to prove a base case for the induction, of course.)

Note that this sort of proof needs an explicit constant; it isn't enough to plug in O(i) on the left side and verify that the result is O(n) on the right side. [Exercise: if T(n)=n+T(n-1), why doesn't an inductive proof show that T(n)=n+O(n) (by induction hypothesis) which is O(n)? Hint: the constant factor hidden by the O-notation needs to truly be constant and not something that can grow each time you induct.]

If we knew c we could perform an explicit induction proof that T(n)=cn. Even though we don't know c, we can still work through the sums, and get something of the form T(n)=f(c) n. What this tells us is that the explicit induction proof works when f(c) is at most c. So by working through the sums with an unknown value of c, we get information that helps us determine c.

Let's try this strategy for quickselect.

$$T(n) = n-1 + \sum_{i=n/2}^{n} 2/n\ T(i)$$

$$\leq n-1 + \sum_{i=n/2}^{n} 2/n\ ci$$

$$= n-1 + \frac{2c}{n} \sum_{i=n/2}^{n} i$$

$$= n-1 + \frac{2c}{n} \left(\sum_{i=1}^{n} i - \sum_{i=1}^{n/2-1} i \right)$$

$$= n-1 + 2c/n\ (n^2/2 - n^2/8 + O(n))$$

$$= (1+3c/4)n + O(1)$$

The induction works (with a large enough base case to swamp the O(1) term) whenever c is greater than 4. So quickselect uses roughly 4n comparisons in expectation.

Quicker selection

Floyd and Rivest noticed that by choosing the pivot point x more carefully, one can get a much better algorithm. I'll just describe this with only sketchy analysis. It's not in the book and I won't test on it, but something like this is what you should be using if you want medians for large data sets. If you want to read more about it, a good recent reference is "Average Case Selection", by Walter Cunto and Ian Munro, in the Journal of the ACM (vol. 36, no. 2, April 1989, pages 270-279).

```
sampleselect(L,n,k)
{
given n,k choose parameters m,j "appropriately"
pick a random subset L' having m elements of L
x = sampleselect(L',m,j)

partition L into L1<x, L2=x, L3>x
if (k <= length(L1))
    return sampleselect(L1,k)
else if (k > length(L1)+length(L2))
    return sampleselect(L3,k-length(L1)-length(L2))
else return x
}
```

The basic idea is that the closer x is to the kth position, the more items we'll eliminate in the final recursive call. By taking a median of a sample, instead of just choosing randomly, we're more likely to get something closer to the kth position.

How to choose m? Typically it should be some small fraction of n, so that the O(m) time used in the first recursive call is small. We'll pick a more exact value after doing the analysis.

How to choose j appropriately? We want it to be very likely that the j-median of L' is close to the k-median of L -- that way as much as possible gets removed from the recursive calls. More specifically, if k is small we want the j-median of L' to be slightly greater than the k-median of L (so that the stuff that gets removed is likely to be on the larger side) and conversely if k is large we want j to be a little small. So

```
j = k(m/n) + fudge
```

where fudge is positive when k is small, negative when k is large. In general the right value of fudge is O(sqrt(m) log n) for reasons that would take some complicated probability theory to explain.

The result: after the top level call, we'll probably have eliminated either most of the items larger than the kth position, or most of the items smaller than it. So in the recursive calls, k is very likely to be near 1 or near n (within O(n log n / sqrt m) of it) since adjacent samples in L' are likely to be separated by roughly n/m positions in L.

After the second level of recursion, we'll probably have eliminated most of the items on the other side of the kth position. There are likely to be many fewer than n items left -- O(n log n /sqrt m) of them to be precise. So the bulk of the time happens in those first two calls. In the first one we do at most n comparisons, and in the second we are likely to only do at most min(k,n-k).

Putting this into a recurrence we get

```
T(n)=n+min(k,n-k)+2T(m)+T(n log n / sqrt m)
```

for the total time (with high probability). We also can show that T(n)=O(n) (or just use quickselect in the recursive calls). So instead of analyzing this recurrence as a recurrence (which would mean figuring out what "with high probability" starts meaning when you iterate the recurrence several times) we just replace T(n) by O(n) in the formula above:

```
T(n)=n+min(k,n-k)+O(m)+O(n log n / sqrt m)
```

We still have a free parameter m. To finish the description of the algorithm and its analysis, we need to choose m. If we make m too large, the O(m) term will dominate the formula. If we make m too small, the other term will dominate. To make m "just right", we make the terms roughly equal to each other:

```
m = n log n / sqrt m

m^(3/2) = n log n

m = (n log n)^(2/3)
```

The result is an algorithm that uses n + min(k,n-k) + O((n log n)^(2/3)) comparisons (with high probability, or in expectation).

This turns out to be about as good as possible for a randomized algorithm. But it remains an open problem exactly how many comparisons are needed without randomization. (Of theoretical significance only since in practice randomization is good.) The current best algorithms use roughly 2.95 n comparisons but are quite complicated. Next time we'll see a simpler (but still complicated) method with O(n) comparisons.
