CS 6505: Computability & Algorithms

Lectures for Week 4, Feb. 1-5, 2010

Merge Sort

Merge Sort is an algorithm for sorting an arbitrary list of items using only comparisons. The algorithm is as follows:

```
MergeSort(L_{1..n})
    if |L| = 1 then return L
    else return Merge(MergeSort(L_{1...⌊n/2⌋}),
        MergeSort(L_{⌈n/2⌉..n}))

Merge(A,B)
    O←Empty List
    for i=1 to min(|A|,|B|)
        if A_1 < B_1
            Append A_1 to O
            A ← A_{2...|A|}
        else
            Append B_1 to O
            B ← B_{2...|B|}
    if |A| > 0
        Append A to O
    if |B| > 0
        Append B to O
```

Now we will analyze the worst case running time complexity of Merge Sort.

Time Complexity

We can make a tree of the running of the algorithm on the list. Each level contains the whole list. The whole list in one piece is on the top level, two halves on the second level, four quarters on the third level, and on the bottom level, individual elements.

Clearly the depth of this tree is $O(\log n)$. At each level, each piece has to be merged into the larger pieces one level up one element at a time. Thus, n elements have to be copied at each level of the tree. Therefore, the time complexity is $O(n \log n)$.

Big-Oh Notation

We've seen the big-oh notation several times, but in the future (including the remainder of this lecture) we will also use big-theta and big-omega notation. Here are the definitions of these notations:

Notation	Definition
$g(n) = O(f(n))$	$\exists c, n_0$ s.t. $\forall n > n_0$, $c \cdot g(n) < f(n)$
$g(n) = \Omega(f(n))$	$\exists c, n_0$ s.t. $\forall n > n_0$, $c \cdot g(n) > f(n)$
$g(n) = \Theta(f(n))$	$\exists c_1, c_2, n_0$ s.t. $\forall n > n_0$, $c_1 \cdot f(n) < g(n) < c_2 \cdot f(n)$

Lower Bound for Comparison Sorting

Can we possibly do better than $O(n \log n)$ time for sorting using comparisons only? No. Here's a proof:

Assume we have a list of the first n naturals. We want a lower bound on the worst case, and having repeated elements can only help us, and when all elements are distinct, we may as well map them them to the naturals.

Assume we have a comparison sorting algorithm that runs in time $f(n)$. If at every point in our sorting algorithm we stored where we moved each element from and to, we could reverse all these directions to create an "unsorter" that takes the first n naturals in order and returns that particular permutation. This means that our sorting algorithm provides enough information to pinpoint which permutation a list is.

Since our algorithm uses only comparisons to determine the order of elements, we can lower bound the running time by the number of comparisons performed. Since each comparison returns only two results ("less than" or "greater than"), we get 1 bit of information from each comparison.

How many bits of information do we need to specify a particular permutation (which we have shown our algorithm collects enough information to do)?

We can number all of the $n!$ permutations from 1 to $n!$ and specify them by their number, but we need at least $\log_2(n!)$ bits to specify the number $n!$. Thus, we need at least $\log_2(n!)$ comparisons. But $\log_2(n!) = \Theta(\log n!) = \Theta(n \log n)$ by Stirling's Approximation (below). Thus, the number of comparisons needed (and hence the running time) is $\Theta(n \log n)$

Stirling's Approximation

$\log n! \sim n \log n - n = \Theta(n \log n)$

Master Theorem (weak version)

If we have a recurrence of the form

$$T(n) = \begin{cases} aT(\frac{n}{b}) + n^c & n > 1 \\ d & n = 1 \end{cases}$$

then Θ-bounds on the non-recurrent solution is given by

$$T(n) = \begin{cases} \Theta(n^c) & \log_b a < c \\ \Theta(n^c \log n) & \log_b a = c \\ \Theta(n^{\log_b a}) & \log_b a > c \end{cases}$$

"Natural" Merge Sort

Merge Sort can be done sequentially on, for instance, a slow tape archive (or a TM) by starting at the bottom of the partitioning tree and working up. For instance, if one has four tapes, unsorted list on tape A, one can scan sequentially copying elements to tape B until the next element is less than the current element, then continue copying elements to tape C until this happens again, then merge the lists on B and C onto tape D before repeating this process. Repeat until all elements on tape A are on tape D.

Now go backwards on tape D merging onto tape A in reverse order. Repeat until the list is sorted.

This process uses no more comparisons in the worst case than the standard merge sort if implemented properly and runs much faster with nearly-sorted lists. Moreover, it avoids linear seek time penalties on tapes or in sequential access data structures like linked lists.

k-th Order Selection

We look at an algorithm for finding the k-th smallest element in a list.

How long does it take to find the max (n-th smallest)? We may not see the maximum until we see the end of the list, so we will need to scan all n entries. Thus, it takes $O(n)$ time.

What about the time to look up the k-th smallest element?

An obvious idea is to scan the entire input for the minimum and remove it. Then we could scan for the minimum of the remaining $n-1$ elements, which would give us the second largest. We could repeat this, giving us a $O(kn)$ algorithm.

Now what about the median? The median is the $\lceil \frac{n}{2} \rceil$-th element in the array. The above algorithm gives us an $O(n^2)$ method. However, this is clearly not optimal since there is an obvious $O(n \log n)$ method.

Using Randomness

If we have access to randomness, we could try this algorithm:

First, randomly select an element, say p, from the n-element array. Now compare the random element p (for "pivot") against all of the elements in the array, sorting elements that it is greater than to its left and those that it is less than to its right. If the element p is only greater than $k < \frac{n}{2}$ elements, then it's less than the median. So we recurse on the elements to the right of p searching for the $\frac{n}{2} - k$-th element. If the element p is greater than $k > \frac{n}{2}$ elements, then

p is greater than the median. So we recurse on the elements to the left of p. Otherwise, we can just output p.

We haven't introduced randomization as a resource in this class, but we can show that this will eliminate $\frac{1}{4}$ elements on average each time.[1]

Suppose that we had a technique that eliminated $1/4$ of the elements at each stage. Then we would be left with $3/4$ of the nodes at each stage. If we iterate over the entire list to gauge how good our pivot was, then we need O(n) time at each stage. So we could use the following analysis:

$$\begin{aligned} T(n) &= n + T(3n/4) \\ &= n + 3n/4 + T(3/4 * 3n/4) \\ &= n/(1 - 3/4) \\ &= 4n \end{aligned}$$

In other words, this was a geometric series that we evaluated. So at each stage, we iterate over our entire list and removed a small portion of our list. Even though we removed a small portion of the list, we still ran in linear time.

Now we will show an algorithm that guarantees we eliminate a constant fraction without using randomness, and so also gives an upper bound of $O(n)$ time. This algorithm is not obvious; for a long time, many thought that $O(n \log n)$ was the best possible.

Without Randomness

We now give the algorithm.[2]

We proceed as follows. Suppose that we have n elements. Now divide them into partitions of 5 elements. Now take the median of each 5-element set and collect these into a set of medians. We'll now find the median of the set of medians using this same algorithm.

What does the median of medians tell us? Well, of the $\frac{n}{5}$ medians, $\frac{n}{10}$ of those medians are less than the median of medians. Now consider the 5-element partitions with medians less than the median of medians. Both the median and the 2 elements less than the median will be smaller than the median of medians. Counting these two elements with the medians themselves, we have $\frac{3n}{10}$ elements less than the median of medians. The same analysis goes for the number of values greater than the median of medians.

Thus, we can eliminate $\frac{3n}{10}$ elements from consideration, making our list at most $\frac{7}{10}$ as long.

Now, we can apply the partitioning function above to recurse and find the median, or another k-th smallest element we seek.

[1] For more information, see CLRS section 9-2.

[2] For more info, see CLRS section 9-3

Time Complexity

Let's analyze the time complexity as follows. First, finding medians of $\frac{n}{5}$ 5-element sets takes time $c\frac{n}{5} = O(n)$. Finding the median of these medians takes time $T(\frac{n}{5})$. Second, once we have the median of medians, we require $O(n)$ to repartition the list around the pivot and time at most $T(\frac{7n}{10})$ to recurse on the set that contains the element we seek.

So our recursion is

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

This recursion can't be attacked by the Master Theorem, but we already know to expect a linear time algorithm, so let's just guess that our algorithm runs in time cn. Substituting, we get

$$\begin{aligned} cn &= \frac{cn}{5} + \frac{7cn}{10} + O(n) \\ &= \frac{9cn}{10} + kn \end{aligned}$$

for some k. This equation is true if $c = 10k$. The existence of a constant c for which the algorithm runs in time cn proves that the algorithm runs in $O(n)$.

Variations

What happens if we partition into groups of size 7? First of all, the step of finding the median of each partition will take longer, which increases the constant in the $O(n)$ term in the recurrence. Using the same analysis as above, we get a recurrence of

$$T(n) = T(\frac{n}{7}) + T(\frac{10n}{14}) + O(n).$$

Knowing that it's a linear function, we see the recurrence is:

$$T(n) = \frac{6cn}{7} + O(n).$$

Notice that $\frac{6}{7}$ is closer to $\frac{3}{4}$ than $\frac{9}{10}$. It is clear that $\frac{3}{4}$ is the lowest we could ever get (even though the constant factor in the $O(n)$ term will blow up unacceptably high before we ever got there).

What happens if we use groups of three? The recurrence becomes

$$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + O(n).$$

Assuming this runs in linear time gives us the equation

$$cn = cn + kn,$$

which clearly has no solution for c when k is positive. In other words, the problem does not get smaller each time we recurse, so we have to do linear work at all $\log n$ levels of the recursion tree, resulting in an $O(n \log n)$ algorithm.