CS787: Advanced Algorithms **Lecture 2:** Divide & Conquer Algorithms

2.1 Divide and conquer

Divide and conquer algorithms break up a problem into several smaller instances of the same problem, solve these smaller instances, and then combine the solutions into a solution to the original problem. Naturally, for this approach to work, the smaller instances should be simpler than the original problem, and combining their solutions together should be easier to do than solving the original problem. In contrast to greedy algorithms, the correctness of D&C algorithms is usually relatively easy to argue but the running time analysis is more involved. The analysis of the time complexity of these algorithms usually consists of deriving a recurrence relation for the complexity and then solving it.

We give two examples of algorithms devised using the divide and conquer technique, and analyze their running times. Both of these algorithms will deal with unordered lists.

Sorting We describe mergsort—an efficient recursive sorting algorithm. We also argue that the time complexity of this algorithm, $\mathcal{O}(n \log n)$, is the best possible worst case running time a sorting algorithm can have.

Finding the k-th largest element Given an unordered list of elements, we describe how to find the k-th largest element in that list in linear time.

2.1.1 Mergesort

The goal of sorting is to turn an unordered list of n elements into an ordered list. Mergesort is a recursive sorting algorithm. It consists of the following two steps whenever the length of the list n is strictly greater than one. When the length of the list is equal to one, it returns the same list.

1. *Split* the list in two halves that differ in length by at most one element, and sort each half recursively.

2. *Merge* the two sorted lists back together into one big sorted list.

The correctness of the algorithm relies on the merge step. This step can be performed in linear time, and we skip the details.

Let us now analyze the running time. Let $T(n)$ denote the time it takes mergesort to sort a list of n elements. The split step in the algorithm takes constant time and generates two lists of size $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$. When we merge the two smaller lists, we end up with a list of n elements, so the merge step takes $\mathcal{O}(n)$ time. Therefore, we get a recursive definition of $T(n)$ of the form

$$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \mathcal{O}(n).$$

One way of solving this reccurrence is to determine the total amount of work done at every level of recursion. In the case of mergesort, the total work done at any level is a constant times the total size of all the lists at that level, which is n. Therefore, the total work at any level is $\mathcal{O}(n)$. The number of levels, on the other hand, is $\log n$. Therefore, solving this recurrence relation yields

$$T(n) = \mathcal{O}(n \log n)$$

.

Ignoring constants, mergesort is asymptotically the fastest way to sort a list. In particular, any sorting algorithm that uses only comparisons to sort a list of integers must take time at least $\Omega(n \log n)$ in the worst case to sort a list of size n. (See [1] for a proof of this theorem.)

2.1.2 Selection, or, finding the k-th Largest Element

In this problem, we are once again given an unordered list of elements, and want to find the kth largest element. A simple way of solving this problem is to first sort the list and then read off the kth largest element. This takes time $\mathcal{O}(n \log n)$. However, presumably finding only the kth largest element should be simpler than sorting the entire list. For example, we could maintain a list of the k largest elements and populate this list in time $\mathcal{O}(n \log k)$. When k is a small constant, this takes only linear time. We will show that we can perform selection in linear time for an arbitrary k using a divide and conquer approach.

To get intuition for how this problem can be solved, suppose that we could find the median of a list in linear time. We claim that we can then use this as a subprocedure in a divide and conquer algorithm to find the kth largest element. In particular, we use the median to partition the list into two halves. Then we recursively find the desired element in one of the halves (the first half, if $k \leq n/2$, and the second half otherwise). This algorithm takes time cn at the first level of recursion for some constant c, $cn/2$ at the next level (since we recurse in a list of size $n/2$), $cn/4$ at the third level, and so on. The total time taken is $cn + cn/2 + cn/4 + \cdots = 2cn = \mathcal{O}(n)$.

Unfortunately, however, finding the median doesn't seem to be much simpler than finding the kth largest element. The key idea here is that in order to apply the recursion, we don't need an exact median – a near-median would do. In particular, suppose we could find an element at every step such that at least 3/10th of the elements in the list are smaller than it and at least 3/10th of the elements are larger than it, then we could still apply the same divide and conquer approach as above. Assuming each divide step takes linear time, our running time would turn out to be at most $cn + \frac{7}{10}cn + \frac{49}{100}cn + \cdots = 3.33cn = \mathcal{O}(n)$.

Finally, it turns out that we can find a near-median in linear time by again applying recursion. In particular, we divide the list into groups of 5 elements each, find the median in each group in constant time (since each group is of constant size), and then find the median of these medians recursively. The key point to note is that the final step of finding the median of medians applies to a much smaller list – of size $n/5$, and so we still get a small enough running time.

This was just a rough description and analysis of the algorithm. A more formal analysis follows. For simplicity of analysis, we assume that all the list sizes we encounter while running the algorithm are divisible by 5.

Algorithm for selection

1. Divide the list into $n/5$ lists of 5 elements each.
2. Find the median in each sublist of 5 elements.
3. Recursively find the median of all the medians, call it m.
4. Partition the list into elements larger than m (call this sublist L_1) and those no larger than m (call this sublist L_2).
5. If $k \leq |L_1|$, return Selection(L_1, k).
6. If $k \geq |L_1| + 1$, return Selection$(L_2$,$k - |L_1|)$.

The correctness of the algorithm is easy to argue and we will skip the argument. Let us analyse the running time. Note that we make two recursive calls. The first is to a list of size $n/5$. The second is to either L_1 or L_2. How large can these lists be? We argue that these lists can be no larger than $7n/10$ in size. This is because there are $n/10$ medians at step 3 that are smaller than m, and there are three elements in each of the sublists corresponding to these $n/10$ medians that are no larger than the medians, and therefore no larger than m itself. Therefore, L_2 is of size at least $3n/10$, and L_1 is of size at most $n - |L_2| \leq 7n/10$. Likewise we can argue that L_1 is of size at least $3n/10$ and therefore L_2 is of size at most $7n/10$.

So we get the following recurrence for the running time of the algorithm:

$$T(n) = cn + T(n/5) + T(7n/10)$$

where cn is the time taken to construct the list of medians and to partition the list into L_1 and L_2 for an appropriate constant c.

One way of solving this recurrence is to guess that the running time is $T(n) = c'n$ and then check whether the equation is satisfied for some value of c'. Substituting this in the equation we get

$$c'n = cn + \frac{9}{10}c'n$$

which implies $c' = 10c$.

Another way is to count the "work" at every level of recursion as we did in analyzing the running time of mergesort. The reader in encouraged to try that approach and show that it leads to the same answer.

We get the following theorem.

Theorem 2.1.1 *The algorithm given above finds the kth largest element in an unordered list in linear time.*

References

[1] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.