COS 226 **Precept 6 Notes** September 24, 2021
Fall 2021
Tarjan

Divide-and-conquer recurrences

We reviewed the proof the MergeSort runs in $\Theta(n\log n)$ time. This algorithm uses *divide and conquer*: it divides the problem into two smaller subproblems, solves these by applying itself recursively, and then combines the solutions to the two subproblems to obtain the solution to the original problem. In the case of MergeSort, each subproblem is half the size of the original (to within 1), and the combining step merges the two sorted lists that are the solutions to the two subproblems, thereby producing a sorted list that solves the original problem.

One way to do the analysis is to think about the tree of recursive subproblems. At each level of the recursion each input value is in exactly one subproblem. The time spent to do the merge in a subproblem is linear in the size of the subproblem. Thus the total time to do all the merges at a given level of the recursion tree is $\Theta(n)$, where n is the number of input values in the original problem. Since the maximum subproblem size decreases by at least a constant factor with each recursive call, the number of recursion levels is $\Theta(\log n)$. The total time is the total time per level ($\Theta(n)$) times the number of levels ($\Theta(\log n)$), and is thus $\Theta(n\log n)$. (If n is a power of two, as Kevin assumed, the subproblem size decreases by a factor of two at each level, but if n is arbitrary, the subproblem size decreases by at least a factor of 3/2: the worst-case is from 3 to 2.)

If n is an exact power of 2, the recurrence that gives the running time of mergesort is $T(1) = \Theta(1)$, $T(n) = 2T(n/2) + \Theta(n)$ for $n > 1$. This recurrence is especially easy to understand, since the total size of the recursive subproblems is the same at each level of the recurrence. In other situations this is not the case. We explored a situation (see the section on in-place mergesort below) in which the recurrence governing the running time is $T(1) = \Theta(1)$, $T(n) = 3T(n/2) + \Theta(n)$ for $n > 1$. If we ignore the constant factors, the recurrence becomes $T(1) = 1$, $T(n) = 3T(n/2) + n$ for $n > 1$. In this case the sum of the subproblem sizes does not remain constant but grows by a factor of 3/2 with each level of recursion.

To solve this recurrence let us assume for simplicity that n is a power of 2. (The solution we shall get holds for arbitrary n, but the proof is messier.) Then the number of levels of recursion is $\lg n$, where lg denotes the base-2 logarithm. Since the total size of the subproblems grows by a factor of 3/2 with each level of recursion, the total size of the subproblems at level k of the recurrence is $n(3/2)^k$. Ignoring the constant factor, the total time is the sum of this value over all levels of the recursion. This sum is at most a constant factor times the last term (why?) which is $n(3/2)^{\lg n} = n^{\lg 3}$, which is about $n^{1.5846}$. Instead of getting a linearithmic result, we get a result that grows as a power of n greater then 1 (but less than 2).

On the other hand, the recurrence $T(1) = 1$, $T(n) = T(n/2) + n$ for $n > 1$ has a linear solution, as we can see by expanding: $T(n) = n + n/2 + n/4 + \ldots < 2n$. This recurrence arises for example in analyzing the best-case running time of the QuickSelect algorithm for finding the median of a set of n numbers, or more generally finding the k^{th} smallest out of a set of n numbers.

2-sums, 3-sums, *k*-sums

We also explored the *k*-sum problem. In the *k*-sum problem, one is given a set of *n* numbers and a fixed value *k*, and asked to determine whether there are *k* of the numbers that sum to zero. More generally one can ask for *all* sets of *k* of the numbers that sum to zero. Important special cases are $k = 2$ and $k = 3$. To simplify the discussion, I'll assume we are just interested in determining whether *any* subset of size *k* sums to zero, but the methods I'll discuss extend to solving the more general problem of finding *all* subsets of size *k* that sum to zero. In stating time bounds I'll assume that *k* is a fixed constant.

The obvious way to solve the problem is to generate all *k*-size subsets and test each one. This takes time proportional to the number of subsets, which is $\Theta(n^k)$. In lecture, Kevin described how to use sorting and binary search to improve the worst-case running time for 3-sum: (1) Sort the numbers. (2) For each distinct pair of numbers *x* and *y*, use binary search in the sorted array to search for $-(x + y)$. This algorithm takes $\Theta(n\log n)$ for the sorting plus $\Theta(\log n)$ time to test each of the $\Theta(n^2)$ pairs, for a total of $\Theta(n^2\log n)$ time.

We can reduce this bound by a factor of log*n* by observing that the searches for the various pairs are related, and we can do better by doing $\Theta(n)$ searches at the same time, using linear instead of binary search. Normally, linear search would be much more time-consuming than binary search, but because we can exploit the dependence among the searches if we use linear search, we gat an overall logarithmic speedup.

The idea works even in the simplest version of the problem, 2-sum. After sorting the numbers, we initialize *x* to be the smallest number and *y* to be the largest. If $x + y = 0$, we stop. If $x + y < 0$, we replace *x* be the next bigger number of the list, and repeat. If $x + y > 0$, we replace *y* by the next smaller number of the list, and repeat. We stop when we find a solution or when $x = y$ (assuming for simplicity that all numbers are distinct).

Exercise: Prove that this algorithm is correct. Hint: suppose x^*, y^* is a solution. Prove that the algorithm will find a solution when $x \leq x^*$.

The total number of times *x* or *y* changes is less than *n*, so the search time is $\Theta(n)$. The sorting time is $\Theta(n\log n)$, so the sorting time dominates the search time, but we still get an improvement over the naïve $\Theta(n^2)$-time algorithm. For 3-sum, we only need to do the sorting once. Then, for each possible value of *x*, we run the 2-sum linear search to look for a pair *y* and *z* such that $x + y + z = 0$. (In the 2-sum search, our target for $y + z$ is $-x$, not zero.). If we want all solutions rather than a single one, we need to constrain the searching so that $x < y < z$: for each new value of *x*, initialize *y* to be the number after *x*.

This idea reduces the time for the *k*-sum problem to $\Theta(n^{k-1})$ if $k \geq 3$. We can reduce the time even further if $k \geq 4$, although at the cost of lots of additional storage space. Suppose *k* is at least four and even. Compute all possible sums of $k/2$ of the numbers. Sort these sums. Now run the 2-sum algorithm on this huge sorted list. In the linear search, *x* is the sum of the smallest $k/2$ candidate numbers and *y* is the sum of the largest $k/2$ of the candidate numbers. One needs to add a way to eliminate spurious solutions, in which *x* and *y* represent non-disjoint sets of numbers. The worst-case running time of this method is $\Theta(n^{k/2})$, but the worst-case space is also $\Theta(n^{k/2})$, making it of questionable practical usefulness.

Exercise: Give an efficient way to eliminate spurious solutions.

Open-ended questions: What happens if k is odd? Is there an even faster method?

The idea of computing sums of half-size subsets was used by Horowitz and Sahni in an algorithm for the general subset sum problem: is there *any* subset, of any size k, that sums to zero? There is quite a bit of literature on this problem, and I encourage you to explore it if you are interested.

In-place MergeSort?

We explored a way of making MergeSort into an in-place algorithm (one that uses only a constant number of extra storage locations other than the input array). Suppose we had a *partial merge* algorithm that, given an array of size n split into a front half H_1 and a back half H_2, each sorted, rearranges the array into a front half A, completely sorted, and a back half B, in unknown order but such that every item in A is no greater than every item in B. Suppose further that the partial merge algorithm takes linear time and is in-place. Then we could implement MergeSort in-place as follows: Given the input array, apply MergeSort recursively to the front half of the array and to the back half of the array. Then do a partial merge. Finally, apply MergeSort recursively to the back half of the array. This sorts the array. (Why?). The recurrence governing the worst-case running time of this algorithm is $T(1) = \Theta(1)$, $T(n) = 3T(n/2) + \Theta(n)$ for $n > 1$, which we saw in the first part of these notes has a solution of $\Theta(n^{1.58+})$.

Thus if we had a linear-time, in-place partial merge algorithm, we could implement MergeSort in place with a sub-quadratic running time. Of course, we need to design a partial merge algorithm. There is such an algorithm, but it is not so simple, I think, so I'll leave the design of such an algorithm to you as a challenge problem. If you want to accept this challenge, I suggest you begin by designing a partial merge algorithm that will result in the first quarter of the array containing the smallest quarter of the items in sorted order. If you can do this, you should be able to bootstrap this algorithm so that for any fixed fraction $f < 1$, it will put the smallest fn items in the front of the array in sorted order and run in linear time, but with a constant factor in the running time that depends (badly) on n. Such a result implies that for any $\varepsilon > 0$, MergeSort can be implemented to run in place with a running time of $\Theta(n^{1+\varepsilon})$.

Unfortunately, this does not attain the ultimate goal, which is to get an in-place version of MergeSort that runs in $\Theta(n\log n)$ time. This goal was achieved by Kronrod using much more complicated ideas. Here is a link to a more recent paper that simplifies Kronrod's algorithm: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.1155&rep=rep1&type=pdf.