

## CS303 (Spring 2008)- Solutions to Assignment 6

### Exercise 2.3-4

The recurrence would be  $T(n) = cn + T(n - 1)$  for some constant  $c$  (and  $T(1) = c$ ), because the recursive sorting is done on  $n - 1$  elements, and the insertion step takes  $O(n)$  time. For solving this recurrence, we can't use the Master Theorem, but in fact, it is even easier: this is just a complicated way of writing  $cn + c(n - 1) + c(n - 2) + \dots + c$ , so the solution is  $c \cdot \sum_{i=1}^n i = c \cdot \frac{n(n+1)}{2} = \Theta(n^2)$ .

### Exercise 2.3-7

First off, notice that it is easy to solve this problem in time  $\Theta(n^2)$ : simply try all pairs of numbers in  $S$  (and there are only  $\binom{n}{2} = \Theta(n^2)$  pairs), and see whether any of them adds up to  $x$ . But we want to improve the running time to  $\Theta(n \log n)$ . One could try to solve this problem from scratch using Divide&Conquer. A better approach is to notice that once the set is *sorted*, the problem becomes a lot easier. So assume that we start by sorting the set  $S$  into an array  $a$  in time  $O(n \log n)$ , say by using Merge Sort or Heap Sort.

Now suppose that we are trying to find two elements  $a[i]$  and  $a[j]$  that add up to  $x$ . We start by looking at the sum  $a[1] + a[n]$ . If that sum is too small, then clearly *no* sum involving  $a[1]$  can ever add up to  $x$ , so we can safely delete/ignore  $a[1]$ . If the sum is too big, then no sum involving  $a[n]$  can ever add up to  $x$ , so we can ignore  $a[n]$  from now on. This gives the following  $\Theta(n)$  algorithm (once the array is sorted): Start with  $i = 1$  and  $j = n$ . While  $i \leq j$ , compare  $x$  to  $a[i] + a[j]$ . If they are equal, we are done. If  $x > a[i] + a[j]$ , then increase  $i$ , else decrease  $j$ .

The correctness of this algorithm follows from the above explanations. It runs in linear time (constant work per iteration), so the running time is dominated by the sorting, and is  $\Theta(n \log n)$ . In general, for many problems with arrays, an  $O(n \log n)$  algorithm is obtained by first sorting the array, and then running a simple linear time algorithm. For instance, that also helps in detecting if an array contains duplicates. This type of question tends to be very common at job interviews.

### Exercise 4.2-2

In this case, the numbers of elements in each node of the recursion tree are not equal, but it is easy to see that until the first leaf is reached, in the tree, the sum of sizes in each level is still  $n$ , and hence, the total work at each level is  $cn$ . Even if we always look at the smaller subtree (corresponding to  $T(n/3)$ ), we see that the first leaf can be at level no higher than  $\log_3 n$ . Thus, for each of at least  $\log_3 n$  layers, the total work is  $cn$  each, and thus, the solution is at least  $cn \log_3 n = \Omega(n \log n)$ .

While the question didn't ask this, it is equally easy to see that the solution is also  $O(n \log n)$ . The reason is that the total number of layers is at most  $\log_{3/2} n$ , by looking at the number of further layers for the larger subproblem in each case. At each of these layers, the total work is at most  $cn$  (for the lower layers, it's actually less), so the total is at most  $cn \log_{3/2} n = O(n \log n)$ .

### Exercise 4.2-4

One really doesn't need a Recursion Tree for this problem. Notice that if  $a$  is a constant, then  $T(a)$  too is a constant. Thus, the recurrence simplifies to  $T(n) = T(n - a) + c' + cn = T(n - a) + c''n$  for some new constant  $c''$ . Now the recurrence simply unrolls to  $c''n + c''(n - a) + c''(n - 2a) + \dots = c''(\sum_{i=0}^{\lfloor n/a \rfloor} n - ia) = \Theta(n^2)$ . (This problem is really very similar to 2.3-4.)

### Problem 4-1

- (a)  $T(n) = 2T(n/2) + n^3$ . By case 3 of the Master Theorem, the solution is  $\Theta(n^3)$ .
- (b)  $T(n) = T(9n/10) + n$ . By case 1 of the Master Theorem, the solution is  $\Theta(n)$ .
- (c)  $T(n) = 16T(n/4) + n^2$ . By case 2 of the Master Theorem, the solution is  $\Theta(n^2 \log n)$ .
- (d)  $T(n) = 7T(n/3) + n^2$ . By case 3 of the Master Theorem, the solution is  $\Theta(n^2)$ .
- (e)  $T(n) = 7T(n/2) + n^2$ . By case 1 of the Master Theorem, the solution is  $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$ .
- (f)  $T(n) = 2T(n/4) + \sqrt{n}$ . By case 2 of the Master Theorem, the solution is  $\Theta(\sqrt{n} \log n)$ .
- (g)  $T(n) = T(n-1) + n$ . We can't apply the Master Theorem here, as there is no division. Instead, we can see that the recurrence unrolls to  $n + (n-1) + (n-2) + \dots + 1 = \sum_{i=1}^n i = \Theta(n^2)$ .
- (h)  $T(n) = T(\sqrt{n}) + 1$ . This one is a bit more tricky, and requires a variable substitution (which we didn't discuss in detail in class, but it's not that hard to figure out). Suppose that we write  $n = 2^k$ , by defining  $k = \lg n$ . Then, we can define  $T'(k) = T(2^k)$ , and get the recurrence

$$T'(k) = T(2^k) = T(\sqrt{2^k}) + 1 = T(2^{k/2}) + 1 = T'(k/2) + 1.$$

We can now solve this directly (or use case 2 of the Master Theorem), and obtain that  $T'(k) = \Theta(\log k)$  (in fact,  $T'(k) = \lg k$ ). Because  $T(n) = T'(\lg n) = \Theta(\lg \lg n)$ , we have found that  $T(n) = \Theta(\log \log n)$ .

### Problem 4-2

Notice that the input consists of  $n \lg n$  bits describing the  $n$  integers in the array, and we are only allowed to look at  $O(n)$  of those bits. So we can't even look at each number in its entirety once. Here's the idea for solving this problem. By looking at the last bit of each number in the array, we can count the number of even vs. odd numbers in the array. Because we also know how many even and odd numbers there are between  $0, \dots, n$ , we can then tell whether the missing number is even or odd. Once we have discovered that, we can rule out half of the numbers from the array (and never look at them again), and continue. Let's assume we determine that the missing number is even.

How do we continue? We look at the second bit. Again, we can count how many even numbers have the second bit equal to 0 vs. 1, and we know how many even numbers between 0 and  $n$  have the second bit equal to 0 vs. 1, and can thus determine the second bit by looking at only one bit each for all even numbers. After that, we can again ignore half of the remaining numbers, and so on.

Suppose that at some point, there are still  $k$  numbers remaining in the array. If  $k = 1$ , we are done. Otherwise, we can look at one bit each for the  $k$  numbers (which takes time  $O(k)$ ), and in that way, fix one more bit and rule out half of the numbers from further consideration. Thus, the running time recurrence for this algorithm is  $T(k) = k + T(k/2)$ , which by case 3 of the Master Theorem has solution  $T(k) = \Theta(k)$ . Thus, for the entire instance, we only check  $\Theta(n)$  bits.