Design and Analysis of Algorithms
Georgia Tech, Spring 2014 **Homework 1** **Instructor: Chris Peikert**

These solutions are being provided **for your personal use only**. They are not to be shared with, or used by, anyone outside this class (Spring 2014 section of Georgia Tech CS 3510A). Deviating from this policy will be considered a violation of the GT Honor Code.

1. *Asymptotics.* For each pair of functions $f(n)$, $g(n)$ given below, state and justify whether $f(n) = o(g(n))$, $f(n) = \Theta(g(n))$, $g(n) = o(f(n))$, or none of these.

   (a) $f(n) = n^2 \log^4(n)$; $\quad g(n) = n^4 \log(n)$

   Solution: We have $f(n) = o(g(n))$ because $f(n)/g(n) = \log^3(n)/n^2$, which goes to 0 as $n \to \infty$. In fact, $\log^c(n)/n^\epsilon$ goes to 0 for any constants $c, \epsilon > 0$, or in other words, polynomial terms (like n^ϵ) always dominate polylogarithmic terms (like $\log^c(n)$) asymptotically.

   (b) $f(n) = \lg(n) = \log_2(n)$; $\quad g(n) = \ln(n)$ (the natural logarithm)

   Solution: We have $f(n) = \Theta(g(n))$ because $\log_2(n) = \ln(n)/\ln(2)$, so $f(n)$ is just a (nonzero) constant multiple of $g(n)$. (Note that the same argument works for $\log_b(n)$ for any fixed base b.)

   (c) $f(n) = 1.01^n$; $\quad g(n) = n^{100}$ (also, compare $f(n)$ to $g(n)$ for moderate values of n)

   Solution: We have $g(n) = o(f(n))$ because $1.01^n = n^k$ for $k = \frac{n \log(1.01)}{\log n}$, and the exponent k grows to ∞ as n grows (while the exponent 100 stays fixed). Thus $\lim_{n\to\infty} f(n)/g(n) = \infty$. However, note that for all $n \leq 100{,}000$, we have $f(n) < g(n)$, so $f(n)$ does not begin to dominate $g(n)$ until n grows to be quite large.

   (d) $f(n) = n$; $\quad g(n) = \begin{cases} n & \text{if } n \text{ is even} \\ n^2 & \text{if } n \text{ is odd} \end{cases}$

   Solution: None of the three cases apply, because $n = \Theta(n)$ but $n = o(n^2)$. The notation $f(n) = \Theta(g(n))$ means that $f(n)$ is between two constant multiples of $g(n)$ for *all* large enough value of n, which is not the case here due to the odd values of n. Similarly, the notation $f(n) = o(g(n))$ means that $\lim_{n\to\infty} f(n)/g(n) = 0$, which is also not the case here due to the even values of n.

2. Do CLRS Problem 2-4 ("Inversions").

   Solution:

   (a) The inversions (index pairs) are $(1, 5)$, $(2, 5)$, $(3, 4)$, $(3, 5)$, and $(4, 5)$.

   (b) The reverse-sorted array $(n, n-1, n-2, \ldots, 1)$ has the most inversions, because *every* pair (i, j) for $i < j$ is an inversion (so there can't be another array with more inversions). The number of inversions is just the number of (i, j) pairs with $i < j$. For each value of $i = 1, \ldots, n$, there are exactly $n - i$ values of j. Thus the number of inversions is $\sum_{i=1}^{n}(n - i) = \sum_{i'=0}^{n-1} i' = (n-1)n/2$, which is $\Theta(n^2)$.

(c) Let $I(A)$ denote the number of inversions in an array A. We claim that the running time of InsertionSort on an array A of length n is $\Theta(I(A) + n)$. Observe that if $I(A)$ is not too large, i.e., only $O(n)$, then the runtime is only $\Theta(n)$, which is asymptotically better than that of MergeSort. But if $I(A)$ is large, say $\Theta(n^2)$ as in part (a), then the runtime is $\Theta(n^2)$.

We prove the claim using induction. Recall that the main loop of InsertionSort iteratively sorts the array prefixes $A[1 \dots i]$ for $i = 1, \dots, n$. Our inductive hypothesis is that the work done in sorting $A[1 \dots i]$ is $\Theta(I(A[1 \dots i]) + i)$. Indeed, this is true for the base case of $i = 1$, because $A[1 \dots 1]$ has 0 inversions and the first iteration of the main loop takes constant time. Now consider the work done in sorting $A[1 \dots i + 1]$: it's the work done in sorting $A[1 \dots i]$, which we know from the inductive hypothesis, plus the work done in the $(i + 1)$st iteration of the main loop. That iteration does $\Theta(1)$ work entering the loop, plus a number of comparisons and swaps that's exactly the number of elements in $A[1 \dots i]$ that are strictly larger than $A[i + 1]$. This is exactly the number of inversions of form $(\star, i + 1)$ in the *original* (unsorted) array A, because the number of inversions of that form is the same for any rearrangement of the elements of $A[1 \dots i]$. Since the number of inversions in the original $A[1 \dots i + 1]$ is exactly the number of inversions in the original $A[1 \dots i]$, plus the number of inversions of the form $(\star, i + 1)$ in A, we conclude that the inductive hypothesis holds for $i + 1$, and thus for n, and this proves the claim.

(d) We'll modify MergeSort so that in addition to sorting its input array, it also returns the number of inversions that were in that array. The base case is easy to modify: just return 0 on an array of length ≤ 1. To see how we should modify the algorithm in the case where we recursively call MergeSort on two halves of the array, we first analyze the kinds of inversions (index pairs) (i, j) that are possible: either i and j are both in the same half (left or right), or i is in the left half and j is in the right half. (We can't have i in the right and j in the left, because that would mean $i > j$.) By our inductive hypothesis, the number of inversions within the left and right halves will be returned by the respective recursive calls to MergeSort, so we just need to count the number of inversions where i is in the left half and j is in the right half. We can do this by modifying the Merge routine to return the number of those inversions. For each element x in the (sorted) left half, we want to know how many elements N_x in the (sorted) right half are strictly smaller than x. But notice that at the moment when we place x into its final position, the current index $j \in \{1, \dots, \lceil n/2 \rceil\}$ into the (copied) right half of the array is exactly $1 + N_x$, because all elements to the left of that index are $< x$ and all elements at or to the right of it are $\geq x$. Therefore, each time we place some x from the left half, the modified Merge algorithm increments a counter by $j - 1$, and returns the final value of the counter.

To summarize, we modify MergeSort so that it additionally returns the sum $X + Y + Z$, where X and Y were the outputs returned by the recursive calls to MergeSort on the left and right halves respectively, and Z is the output of the modified Merge routine. By inspection, the Merge routine still takes $\Theta(n)$ time and we still have the runtime recurrence $T(n) = 2T(n/2) + \Theta(n)$, which solves to $T(n) = \Theta(n \log n)$.

3. Solve the following recurrences, giving your answer in Θ-notation. Assume a base case of $T(n) = 1$ for $1 \leq n \leq 4$. Show your work.

   (a) $T(n) = T(n - 4) + \lg(n)$

Solution: We can repeatedly invoke (or "unwind") the recurrence to get

$$T(n) = \lg(n) + \lg(n-4) + \lg(n-8) + \cdots + 1,$$

where there are $\Theta(n)$ terms (about $n/4$ to be more precise) in the sum. Since each term is $\leq \lg(n)$, we get $T(n) \leq \lg(n) \cdot \Theta(n)$, and so $T(n) = O(n \lg n)$. However, this is only an *upper* bound on $T(n)$ (hence the $O(\cdot)$ expression), and bounding each term by $\lg(n)$ might yield "too loose" of an answer. (For example, if we very foolishly bounded each term by n^{100}, we would get the upper bound $T(n) = O(n^{101})$, but obviously this is not very helpful for getting a *tight* $\Theta(\cdot)$ bound!)

To get a $\Theta(\cdot)$ expression for $T(n)$ we also need to give a *lower* bound. Hoping that our bounds above were not too loose, we can conjecture that $T(n) = \Theta(n \lg n)$. Then it remains to prove that $T(n) = \Omega(n \lg n)$, or in other words $T(n) \geq c \cdot n \lg n$ for some constant $c > 0$ and all large enough n. To prove this, we can ignore the terms in the rightmost half of the above sum (which are all relatively small, but positive) and consider just the left half, which contains the larger terms. There are $\Theta(n)$ (about $n/8$ to be more precise) terms which are all at least $\lg(n/2) = \lg(n) - 1$. Therefore $T(n) = \Omega(n \lg n)$, as needed.

(b) $T(n) = 5T(n-3)$

Solution: We can unwind the recurrence to get $T(n) = 5 \cdot 5 \cdots 5 \approx 5^{n/3}$, because there are about $n/3$ factors of 5 in the expansion. Note that there are not exactly $n/3$ terms, but this count is only off by one or two, which just amounts to a constant factor that gets absorbed by the $\Theta(\cdot)$ notation. Therefore we have $T(n) = \Theta(5^{n/3})$.

Note that while the expression $T(n) = \Theta(5)^{n/3}$ is also technically correct, it is much less informative than the answer given above, because $\Theta(5)$ could equally well represent 5, 100, $1/2$, or any other positive constant. As an exercise, verify that $100^{n/3} \neq \Theta(5^{n/3})$, and that $(1/2)^{n/3} \neq \Theta(5^{n/3})$.

(c) $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + \Theta(n)$ (Hint: work out the recursion tree.)

Solution: By filling out the recursion tree we see that each level represents $c \cdot n$ work, and the levels contain instances of size n, $\sqrt{n} = n^{1/2}$, $\sqrt{n^{1/2}} = n^{1/4}$, $\sqrt{n^{1/4}} = n^{1/8}$, etc. So the only question is how many levels there are, or equivalently, how many times h must we take a square root (staring at n) before reaching a base case (say, 2)? Observe that the size at level i of the tree (where the root is level 0) is $n^{(1/2^i)}$. So we just need to solve for the height h in the equation

$$n^{1/2^h} = 2 \iff n = 2^{(2^h)} \iff \lg(n) = 2^h \iff \lg(\lg(n)) = h.$$

Therefore, $T(n) = \Theta(n \log \log n)$.

Note that we *cannot* apply the Master Theorem to this recurrence, because we have $a = b = \sqrt{n}$ which are not constants. And indeed, applying the Master Theorem would give the *incorrect* answer $T(n) = \Theta(n \log n)$.

4. *Partial sorting.* The ACME corporation claims to have developed a comparison-based algorithm PSort that "partially sorts," in the following sense: given an input array $A[1 \ldots n]$, the algorithm rearranges it so that the smallest one-tenth of the elements are placed in sorted order in the subarray $A[1 \ldots n/10]$, whereas the largest nine-tenths of the elements can remain in *any order* in the remaining positions of the array. ACME claims that PSort is comparison-based, correct on all inputs, and that its worst-case runtime is only $f(n) = O(n)$.

   (a) Assuming that ACME's claims are correct, give the most efficient algorithm you can that uses PSort as a subroutine to *fully* sort an input array, and argue that it is correct.

   > **Solution:** Our algorithm SortWithPSort($A[1 \ldots n]$) is as follows: if $n < 10$ (in which case PSort isn't guaranteed to sort anything), then sort A using any method. Otherwise, call PSort(A) on the entire array, then call SortWithPSort($A[n/10 \ldots n]$), i.e., recurse on the top $9/10$ of the array (which may still be unsorted).
   >
   > Correctness of the base case is obvious. Correctness of the recursive case follows by induction: the smallest one-tenth of the elements are sorted and put in the correct positions by PSort, and by the inductive hypothesis the recursive call to SortWithPSort sorts the remaining nine-tenths of the elements, yielding a fully sorted array.

   (b) Give a recurrence for the worst-case runtime $T(n)$ of your algorithm, and give an asymptotic solution in closed form.

   > **Solution:** The total runtime is equal to the runtime of PSort plus the runtime of the algorithm on the remaining nine-tenths of the array. This yields the recurrence $T(n) = T(9n/10) + O(n)$, with a base case of $T(i) = O(1)$ for $i < 10$. By the master theorem with $a = 1$ and $b = 10/9$ (so $\log_b(a) = 0$), the closed form is $T(n) = O(n)$.

   (c) Can ACME's claims possibly be true? Why or why not?

   > **Solution:** ACME's claims cannot be true! The runtime lower bound for comparison-based sorting algorithms is $\Omega(n \log n)$. But if ACME's claims were true, by using PSort as a subroutine we will have obtained a comparison-based algorithm that runs in only $O(n)$ time, which is impossible! Therefore, ACME's claims cannot be true.

5. *Dominant elements.* In this problem, array elements are objects that can be compared for equality, but do not have any natural ordering, so they cannot be sorted. That is, for elements a, b we can test whether $a = b$ (as a basic operation), but not whether $a < b$.

   A *dominant element* in an array $A[1 \ldots n]$ is one which appears strictly more than $n/2$ times. Therefore, an array can have only one dominant element, or none at all. In this problem you will devise a divide-and-conquer algorithm that finds the dominant element in a given array, if one exists.

   (a) Briefly describe a property of dominant elements that naturally leads to a divide-and-conquer algorithm.

Solution: If you split the array into two nearly equal-sized arrays (or more generally, two subarrays of any size), a dominant element of the whole array must also be dominant in at least one of the two halves. Thus, dividing the array into two nearly equal halves and checking for dominant elements within those, plus a little extra work, will solve the problem.

(b) Write your algorithm in pseudocode, and briefly argue why it is correct.

Solution: $\mathsf{DOMINANT}(A[1 \ldots n])$

```
if(n == 1)
    return A[1]
else
    D1 = DOMINANT(A[1 ... n/2])
    D2 = DOMINANT(A[n/2 ... n])
    if(D1 == D2)
        return D1
    else
        if(D1 != null)
            if(number of occurrences of D1 in A[1 ... n] > n/2)
                return D1
        if(D2 != null)
            if(number of occurrences of D2 in A[1 ... n] > n/2)
                return D2
        return null
```

Except in the trivial base case, the algorithm breaks the array into 2 nearly equal subarrays, then considers 3 possibilities. If each subarray has the same dominant element, or neither has a dominant element, the same is true for the entire array. If only one subarray has a dominant element, or each subarray has a different dominant element, each element needs to be checked against the dominant element(s) to determine how many times that element occurs in the entire array. If it occurs more than $n/2$ times, it is the dominant element. Since there can be only one dominant element, there isn't a problem in checking for dominant elements when the subarrays each have different dominant elements.

(c) Give a recurrence for the worst-case runtime $T(n)$ of your algorithm, and give an asymptotic solution in closed form.

Solution: $T(n) = 2T(n/2) + \Theta(n)$, which in closed form is $T(n) = \Theta(n \log n)$. Notice that in the worst case, the non-recursive steps really do take $\Theta(n)$ work, because when $D1$!= $D2$ and $D1$!= null we must pass over $\Theta(n)$ array elements to count the number of occurrences of $D1$.