Problem Set 1 Solutions

This problem set is due **at 11:59pm** on **Wednesday, February 15, 2017.**

EXERCISES (NOT TO BE TURNED IN)

Asymptotic Analysis, Recursion, and Master Theorem

- Do Exercise 4.3-7 in CLRS on page 87.
- Do Exercise 4.3-9 in CLRS on page 88.

Divide and Conquer Algorithms

- Do Exercise 4.2-3 in CLRS on page 82.
- Do Exercise 9.3-1 in CLRS on page 223.

Problem 1-1. Recurrences and Asymptotics [50 points]

Let $T(n)$ be the time complexity of an algorithm to solve a problem of size n. Assume $T(n)$ is $O(1)$ for any n less than 3. Solve the following recurrence relations for $T(n)$.

(a) [6 points] $T(n) = 9T\left(\frac{n}{3}\right) + n^2$

Solution: By case 2 of the Master Theorem, since $f(n) = n^2$ is $\Theta(n^{\log_3 9})$, $T(n) = \Theta(n^2 \log n)$.

(b) [6 points] $T(n) = 5T\left(\frac{n}{3}\right) + n$

Solution: By case 1 of the Master Theorem, since $f(n) = n$ is polynomially less than $n^{\log_3 5}$, $T(n) = \Theta(n^{\log_3 5})$.

(c) [10 points] $T(n) = 7T(\sqrt[8]{n}) + \log^2 n$.

Solution: We solve this one by changing the variables. Assume that n is a power of 2, and let $n = 2^m$, then

$$T(2^m) = 7T(2^{m/8}) + m^2.$$

If we define $S(m) = T(2^m)$, the recurrence becomes

$$S(m) = 7S(m/8) + m^2.$$

We can use the Master Theorem (case 3), since $\log_7 8 < 2$ and the regularity condition holds (note: $7(m/8)^2 < m^2$). Therefore, $S(m) = \Theta(m^2)$, and $T(2^m) = \Theta(m^2)$. Changing the variable back ($m = \log n$), we have $T(n) = \Theta(\log^2 n)$.

(d) [10 points] $T(n) = T(n/2) + T(n/4) + \Theta(n)$.

Solution: Use a recursion tree. Since the cost at each subproblem is $\Theta(m)$ (where m is the size of the subproblem), we can upper bound it with cm for some constant c. We prove by induction that the cost at the k^{th} level of the tree the cost is at most $(\frac{3}{4})^{k-1}cn$. Consider the base case first: at the top level of the tree, the cost is at most cn as the original problem is of size n. Suppose, by our inductive hypothesis, that at the k^{th} level of the tree the cost is at most $(\frac{3}{4})^{k-1}cn$: then, at the $(k+1)^{th}$ level we have at most a cost of $(\frac{3}{4})^{k-1}cn/2 + (\frac{3}{4})^{k-1}cn/4 = (\frac{3}{4})^{k}cn$. This proves our inductive step. Hence, by induction, the cost at the k^{th} level of the tree the cost is at most $(\frac{3}{4})^{k-1}cn$. The total cost is the sum of the costs at all levels, that is $\sum(\frac{3}{4})^{k-1}cn$, which is a geometric sum converging to $\Theta(n)$.

(e) [18 points] Consider the following functions. Within each group, sort the functions in asymptotically increasing order, showing strict orderings as necessary. For example, we may sort $n^3, n, 2n$ as $2n = O(n) = o(n^3)$.

1. [6 points] $\log n$, $\sqrt{n}$, $(\log n)^2$, $\log \log n$.

   Solution: $\log \log n = o(\log n) = o((\log n)^2) = o(\sqrt{n})$
   A change of variables can be helpful here - if you're not sure whether $(\log n)^2$ is asymptotically bigger or smaller than $O(\sqrt{n})$, replacing n with 2^m and comparing m^2 with $2^{m/2}$ is easier.

2. [6 points] $n^{4/3}$, $n \log n$, n^2, $\log(n!)$.

   Solution: $\log(n!) = O(n \log n) = o(n^{\frac{4}{3}}) = o(n^2)$, using Stirling's approximation for the factorial.

3. [6 points] $n!$, n^n, e^n, $2^{\log n^{\log \log n}}$.

   Solution: $2^{\log n^{\log \log n}} = o(e^n) = o(n!) = o(n^n)$, using Stirling's approximation for the factorial.

Problem 1-2. Counting Dominating Pairs [50 points]

Ben Bitdiddle is working with points in d dimensions. A d-dimensional point P is represented as a d-tuple $P = (p_1, p_2, \ldots, p_d)$ where p_i is its component in the i-th dimension.

Given two d-dimensional points P and Q, we say that P dominates Q if P is has strictly bigger values than Q in each dimension. More formally, P *dominates* Q if $p_i > q_i$ for all $i \in \{1, \ldots, d\}$.

Ben is given two sets of d-dimensional points $S = \{S_1, S_2, \ldots, S_n\}$ and $T = \{T_1, T_2, \ldots, T_n\}$ such that $|S| = |T| = n$. We say that a pair of points (S_i, T_j) such that $S_i \in S$ and $T_j \in T$ is a *dominating pair* if S_i dominates T_j.

(a) [8 points] Suppose $d = 1$. Give an efficient algorithm for computing the number of dominating pairs, (S_i, T_j), from S to T.

Solution:

Approaching the solution: The first step here is to think about what our input looks like. Each point is a single number here, so each point in S dominates all points in T that are smaller than it.

For each point in S, we need to come up with a way to count all these corresponding points in T. An intuitive, naive approach would be to iterate through all points in T. But that is not the best we can do. A useful fact to keep in mind is that comparisons

are transitive. So, if some point dominates another, it will dominate all smaller points too. Therefore, sorting T ensures that any point in S will be greater than some chunk of T and smaller than the other chunk - the separating point between the two chunks will be at the position that we are looking for.

Solution:

For $d = 1$, each point is simply a single number. We start by making the observation that for any given point $S_i \in S$, the number of dominating pairs it creates with points in T is exactly the number of items in T that are smaller than it.

Our solution is therefore to sort both sets S and T. Then for each element in S, do binary search for it in T and count the number of points smaller than it. The total number dominating pairs is then the sum of dominating pairs created by each point in S.

To bound the running time, note that sorting both lists takes $O(n \log n)$ time, and binary searching for each of the n points from S in T takes $O(\log n)$ time per point, for a total of $O(n \log n)$ time.

Alternative Solution: We can perform the mergesort merge routine, and count dominations at each step.

We sort both S and T. Then, we merge the elements of S and T. When we move some element S_i to the output, we will increase the number of dominating pairs by the number of elements of T that S_i dominates. This is simply the number of elements of T smaller than S_i, which is the index of the smallest element of T that has not yet been moved to the output. Thus, we can find the number of dominating elements in $O(n \log n)$, to sort the sets, plus $O(n)$, to do the merge step, which adds up to $O(n \log n)$ time.

(b) [14 points] Now let $d = 2$. Give an efficient divide and conquer algorithm for counting the number of dominating pairs from S to T. For full credit, your algorithm should run in $O(n^{\log_2 3})$ time.

Hint: Consider using the median finding algorithm from lecture, along with your solution for part (a).

Solution:

Approaching the solution: Let's follow the hint here. We are encouraged to use the median finding algorithm from class, which means that the median will somehow be useful for our purposes.

Not only that - the median finding algorithm from class is also a divide-and-conquer algorithm on lists, so we can perhaps take inspiration from there to find a way to approach the problem at hand.

One important quality of the median is that it allows us to do comparisons in bulk, just like we do in the median-finding algorithm. That is, suppose that we have the median of two single-dimensional arrays. Comparing the medians of the two arrays will not tell us everything about how the elements in the two groups compare, but it will tell us that everything smaller than the smaller median must be smaller than everything larger than the larger median. This can save us some time, especially if we want to divide our elements to compare. Recall this intuition as it is quite important. (Also, it is what we use in the median-finding algorithm.) This is the kind of intuition we need here; however, it remains uncertain how we could apply the median finding algorithm on two-dimensional lists.

It turns out that in this case the most naive approach is actually the correct one: we can try finding the medians of the two arrays in just one of the two dimensions. Finding medians and splitting the arrays around them will give us four sub-lists, and the comparisons between these sub-lists will be our divide-and-conquer subproblems. Using the aforementioned median property will then allow us to do save work on some of the subproblems, resulting in the required running time.

Solution:

We solve this using divide and conquer. In this scenario, since we are working with 2 dimensional points, we first focus on the first dimension and perform our divide step of the algorithm in this dimension.

Step 1:

Let S_m be the median point in S according to the first dimension. We partition the points in S around this median S_m based on their first dimension. Let $S_{\text{small}} \subseteq S$ be all the points that have a smaller first dimension than S_m and let $S_{\text{big}} \subseteq S$ be the equal or bigger points. This step takes $O(n)$ time using our median finding algorithm from lecture.

Similarly, let T_m be the median point in T according to the first dimension, and let T_{small} and T_{big} be defined similarly after partitioning around T_m.

Step 2:

We now have four dominating pair counting subproblems of sets of size $\frac{n}{2}$,

1. S_{small} to T_{small}
2. S_{small} to T_{big}
3. S_{big} to T_{small}
4. S_{big} to T_{big}

The total number of dominating pairs is clearly the sum of the count for each subproblem. Instead of recursively solving all four subproblems, we can do something more clever, depending on the relation of S_m and T_m. Let $S_m[1]$ denote the first dimension of S_m and similarly define $T_m[1]$.

Suppose that $S_m[1] \leq T_m[1]$. This implies that the first dimension of everything in S_{small} is smaller than (or equal to) the first dimension of everything in T_{big}. Therefore there cannot be any dominating pairs in subproblem 2, and don't need to recurse on that subproblem.

Suppose now that $S_m[1] > T_m[1]$. This implies that the first dimension of all the points in S_{big} is strictly greater than the first dimension of all points in T_{small}. We can therefore count the number of dominating pairs in subproblem 3 by using our algorithm from part (a) on the second dimension of the points.

In either case, we only need to run 3 recursive calls, and possibly an additional $O(n \log n)$ cost for using part (a). Let $T(n)$ be the overall running time for our algorithm. The recurrence is given by,

$$T(n) = O(n) + 3T\left(\frac{n}{2}\right) + O(n \log n)$$

This solves to $O(n^{\log_2 3})$ by the Master theorem.

Alternative solution: We will solve a slightly different problem, which we can use to solve the one given. The problem is: Given a set R of n points marked as either from set S or from set T, count how many dominating pairs from S to T exist. The difference here is that S and T need not be the same size.

We will find R_m, the median point in R according to the first dimension. We will partition the points in R around R_m based on their first dimension. Let R_{small} be the points that have a smaller first dimension than R_m, and let R_{big} be all the points that have a greater or equal first dimension.

We now have three dominating pair counting subproblems of size $\frac{n}{2}$.

1. R_{small} to R_{small}
2. R_{big} to R_{small}
3. R_{big} to R_{big}

The overall solution is the sum of number of dominating pairs in these three subcases.

The first and third subproblems are of the same type as the original problem, but with $\frac{n}{2}$ elements instead of n elements.

The second subproblem can be handled somewhat differently, and more efficiently. Notice that, because of the division strategy, every element of R_{big} has a larger first dimension than every element of R_{big}. Therefore, the only dominating pairs in this

case will be elements of S in R_{big} dominating elements of T in R_{small}, and we only need to examine the second dimension to find out which ones.

Call the former group S_{big}, and the latter T_{small}. Finding the number of dominating pairs in the second dimension is simply an instance of the one dimensional problem from (a), although possibly with unbalanced S and T sets. Those algorithms still work in this case, with the same running time, so this can be solved in $O(n \log n)$ time.

The runtime of this algorithm is $T(n) = 2T(n/2)$ (first and third subproblems) $+O(n)$ (median finding and partitioning) $+O(n \log n)$ (second subproblem). This becomes $O(n \log^2 n)$, by the second case of the Master theorem.

In fact, we can be even more clever. If we use the sorting and merging solution to the 1 dimensional problem, then we can save more time by presorting the data by its second dimension.

At the beginning of the algorithm, we will sort R according to its second dimension. At each subsequent step, we will maintain R in sorted order. When we partition the subproblems, we will separate R_{small} and R_{big}, but we will keep each set sorted by its second dimension. Likewise, when we separate out S_{big} and T_{small}, we will keep the data sorted by its second dimension.

Thus, whenever we are counting a one-dimensional case, the data will already be sorted by the dimension we are examining, and so we can skip the sorting step, and simply merge the two sets.

Let $S(n)$ be the time it takes to count dominating pairs when R is already sorted. Then $S(n) = 2S(n/2)$ (two subproblems) $+O(n)$ (median finding and partitioning) $+O(n)$ (merging). $T(n) = O(n \log n)$ (sorting) $+S(n)$.

By the second case of the Master theorem, $S(n) = O(n \log n)$, and so $T(n) = O(n \log n)$.

Since the problem that was actually solved is a more general one that the original problem, we can use it to solve the original problem in the same amount of time.

(c) [16 points] Generalize your algorithm from part (b) to any $d \geq 2$. Analyze your running time in terms of n and d.

Solution:

Approaching the solution: Notice that the question is specifically asking to generalize the algorithm from the previous part of the problem. The key idea here is that everything that was done in the previous part can be repeated almost identically - starting from taking the median of the two arrays in the first dimension.

The only hurdle that we need to face here is that now we may be left with a recurrence that depends on d as well as n. While there are many ways to deal with such a recurrence, let's look at what we have. We already found solutions for $d = 1$ and $d = 2$.

We can try finding the solution for $d = 3$, and then see if we can find a pattern that can be proven by induction.

Solution:

Our solution is a generalization of the algorithm from part (b). In this case, we must explicitly state the size of our subproblem in terms of n and d.

Similarly as before, we partition S and T around S_m and T_m, which are the medians according to the first dimension. We therefore have the same four subproblems of size $\frac{n}{2}$ and dimension d as before,

1. S_{small} to T_{small}
2. S_{small} to T_{big}
3. S_{big} to T_{small}
4. S_{big} to T_{big}

However, similarly as before, if $S_m[1] \leq T_m[1]$, then we only need to recursively solve subproblems 2, 3 and 4 since we know there cannot be any dominating pairs in subproblem 1. That would be three recursive calls to problems of size $\frac{n}{2}$ and dimension d.

Alternatively, if $S_m[1] > T_m[1]$, then we already know that according to the first dimension, everything in S_{big} dominates everything in T_{small}, hence we simply need to count the number of dominating pairs according to the remaining $d-1$ dimensions.

This gives us the following recurrence for our running time $T(n, d)$.

$$T(n, d) = O(n) + 3T\left(\frac{n}{2}, d\right) + T\left(\frac{n}{2}, d-1\right)$$

We can solve this recurrence inductively. We already showed in part (b) that we get $T(n, 2) = O\left(n^{\log_2 3}\right)$ when the dimension $d = 2$. Similarly, for $d = 3$, it is easy to see that by the Master theorem, we get $T(n, 3) = O\left(n^{\log_2 3} \log n\right)$. In general, we see that $T(n, d) = O\left(n^{\log_2 3} \log^{d-2} n\right)$.

Alternative Solution:

Using the alternative approach from (b), we can apply that idea to this problem. Notice that given a set R of elements from S and from T with n elements in d dimensions, splitting R at its median iin one dimension gives us three subproblems: R_{big} vs. R_{big}, R_{small} vs. R_{small},R_{big} vs. R_{small}, The first two are subproblems with $\frac{n}{2}$ elements and d dimensions, while the third has up to n elements and $d-1$ dimensions, since we know that all elements of R_{big} have larger first dimensions than R_{small}. This step takes $O(n)$ time, for median finding and partitioning.

Thus, this gives us the recurrence $T(n, d) = 2T(n/2, d)+T(n, d-1)+O(n)$. By case 2 of the Master theorem, if $T(n, d-1) = O(n \log^k n)$, then $T(n, d) = n \log^{k+1} n$, since $n^{\log_b a} = n$ in this case.

If we start from the one dimensional case, which was solved in $O(n \log n)$ time, and induct, we get an algorithm for the d dimensional case that runs in $O(n \log^d n)$ time, $d \geq 1$. If we start from the alternative solution for the two dimensional case, which was solved in $O(n \log n)$ for $d = 2$, and induct, we get an algorithm for the d dimensional case that runs in $O(n \log^{d-1} n$, $d \geq 2$.

We now restrict ourselves to the case $d = 2$. Ben is now interested in finding the number of dominating pairs from S to S. That is, he wants to count the number of pairs (S_i, S_j) such that both $S_i, S_j \in S$ and S_i dominates S_j.

At first glance, it is tempting to simply reuse our algorithm from part (b) and simply use two copies of S. However, Ben is convinced there must be a faster algorithm to solve this problem directly.

(d) [12 points] Give a divide and conquer algorithm for computing the number of dominating pairs from S to S that runs in $O(n \log^2 n)$ time. Recall that we have fixed $d = 2$.

Solution:

Approaching the solution: Again, the key intuition from before is what is needed to proceed here - recall that finding an array's median lets us separate the list in subproblems that can be more easily dealt with.

Solution:

Similarly to our algorithm above, let S_m be the median point in S according to the first dimension. We partition S around S_m according to the first dimension. Can now only need to worry about three possible subproblems for counting dominating pairs,

1. S_{small} to S_{small}
2. S_{big} to S_{big}
3. S_{big} to S_{small}

It is easy to see that the omitted case cannot have any dominating pairs. Out of the three subproblems, we only need to recursively solve 1 and 2. For subproblem 3, we see that everything in S_{big} dominates everything in S_{small}, and therefore we only need to count the number of dominating pairs according to the second dimension only, for this we can use our algorithm from part (a) to solve it in $O(n \log n)$ time.

Overall, we get our final recurrence $T(n) = O(n) + 2T\left(\frac{n}{2}\right) + O(n \log n)$, which solves to $O(n \log^2 n)$ by the Masters theorem.

Alternative solution: The alternative solutions mentioned above essentially result from applying the above solution to the previous problems. However, by applying the presorting trick, the runtime for this algorithm can be reduced to $O(n \log n)$, as mentioned above, in part (b).