This homework is due by the ***start of class on Friday, January 29nd***. You can either submit the homework via the course page on T-Square or hand it in at the beginning of class. Since this is a short homework, this is an ideal time to learn to use Latex.

Homework Policies:

- Your work will be graded on correctness and clarity. Write complete and precise answers.
- You may collaborate with up to *three* other classmates on this problem set. However, you must *write your own solutions* in your own words and *list your collaborators*.
- You may portions of our two textbooks that we have covered. The purpose of these homeworks is NOT only to teach you the algorithms that we cover, but also to teach you how to problem-solve. Therefore using *ANY* outside resources like the internet, previous students etc. is *not* allowed!
- **Optional** questions are for you to practice and learn the basics before you advance. They may ask you to follow the steps of an algorithm covered in class, or review material from an earlier class. Solutions to these problems will not be graded, but you must be are able to do them, and you are responsible for material that is addressed by these optional questions.
- **Basic** questions *must be solved and written up alone*; you may not collaborate with others. These will help you build the framework needed to solve the harder questions. Please do these first, before meeting in groups, for both you and your group's benefit.
- **Group** questions may be solved in collaboration with up to ***three*** other classmates. However, you must still *write up your own solutions* alone and in your own words. These will form the bulk of your homework questions.
- **Bonus** questions are like group questions, but they will be more challenging and will be graded more rigorously. Bonus points are tallied separately from your normal points, and will be applied to your grade *after* the curve. Bonus points are worth more than normal points, and can have a significant positive affect on your grade. Most importantly, bonus problems are fun!

1. (Basic) Solve the following recurrences using the master theorem. You may give the answers in Big-Θ notation.
   (a) $T(n) = 7T(n/4) + n$.

   Solution: Using the Master Theorem - $\log_4 7 > 1$, so the answer is $T(n) = O(n^{\log_4 7})$. Or by the recursion tree method, we get 7^i calls per level, $n/4^i$ is the argument, so we do $n/4^i$ work per call, and there are $\log_4 n$ levels.
   That means the total work done is

   $$\sum_{i=0}^{\log_4 n} 7^i n/4^i = n \sum_{i=0}^{\log_4 n} (7/4)^i = n\frac{(7/4)^{\log_4 n+1} - 1}{7/4 - 1} = O(n\frac{7^{\log_4 n}}{4^{\log_4 n}}) = O(n^{\log_4 7})$$

   (b) $T(n) = 7T(n/4) + n^2$.

Solution: Using the Master Theorem - $\log_7 4 < 2$, so the answer is $T(n) = O(n^2)$.
Or by the recursion tree method, we get 7^i calls per level, $n/4^i$ is the argument, so we do $(n/4^i)^2$ work per call, and there are $\log_4 n$ levels.
That means the total work done is

$$\sum_{i=0}^{\log_4 n} 7^i (n/4^i)^2 = n^2 \sum_{i=0}^{\log_4 n} (7/16)^i$$

$1 \leq \sum_{i=0}^{\log_4 n} (7/16)^i \leq \sum_{i=0}^{\infty} (7/16)^i = 16/9$ so our final answer is $O(n^2)$.

(c) $T(n) = 64T(n/3) + \sqrt{n}$.

Solution: Using the master theorem, $\log_3 64 > 1/2$, so the final answer is $T(n) = n^{\log_3 64}$.

2. (Group) Solve the following recurrences WITHOUT using the master theorem. You should describe or draw a tree, derive a series, and the find the summation of that series. You should give your answers in Big-Θ notation for parts b) and c). Show your work.

(a) $T(n) = T(n-1) + 2^n$.

Solution: We'll think about this as if doing the recursion tree method.
Each call spawns one other call with the argument decreased by one. This means that the ith level of this tree will have 1 node with argument $n - i$, and does 2^{n-i} "work", and there will be n levels total.
That means the total work done is

$$\sum_{i=0}^{n} 2^{n-i} = \sum_{j=0}^{n} 2^j = \frac{2^{n+1} - 1}{2 - 1} = 2^{n+1} - 1 = \Theta(2^n)$$

(b) $T(n) = 5T(n/4) + n^2$.

Solution: By the recursion tree method, we get 5^i calls per level, $n/4^i$ is the argument, so we do $(n/4^i)^2$ work per call, and there are $\log_4 n$ levels.
That means the total work done is

$$\sum_{i=0}^{\log_4 n} 5^i (n/4^i)^2 = n^2 \sum_{i=0}^{\log_4 n} (5/16)^i$$

$1 \leq \sum_{i=0}^{\log_4 n} (5/16)^i \leq \sum_{i=0}^{\infty} (5/16)^i = 16/11$ so our final answer is $\Theta(n^2)$.

(c) (Bonus) $T(n) = T(an) + T(bn) + n$ for constants $a, b > 0$ and $a + b < 1$. Note, we want a big-Θ answer, so you need upper and lower bounds.

Solution: At each level, we have 2^i calls, but wait, they are all different! At the top level, we do n work. At the next level, we do $an + bn$ work. In fact, you can see that at each level, we do $(a+b)^i n$ work, if the level is complete. You know that there is at least one level, and certainly at most infinity levels. So the total amount of work $T(n)$ that you do

$$n \leq n\sum_{i=0}^{0}(a+b)^i \leq T(n) \leq n\sum_{i=0}^{\infty}(a+b)^i = \frac{n}{1-a-b}$$

So, $T(n) = \Theta(n)$ because it is bounded between two multiples of n!

3. (Group) We have an array of objects $A[1 \dots n]$, and the task is to design an efficient algorithm that decides if there is an element that comprises more than half of the elements of the array. You should return a such an element, or a special NULL object if there is none. These are abstract objects, so there can be no comparisons of the form "$A[i] > A[j]$" - you can only test if "$A[i] = A[j]$", in constant time. Therefore, you can't sort.

   (a) Show how to solve this problem in $O(n \log n)$ time using a divide and conquer approach. This about how would you use the solutions to the subproblems to consider a solution to the main problem.

Solution: Our Algorithm with either return an element x, or will say that there is no dominant entry.

Assume A has a dominating entry x. It follows that x must also be the dominant entry of at least one of the two half-arrays by the pidgeonhole principle - if there are more than n/2 copies of x in the array A, then at least one of the two half-arrays bucket must contain more than n/4 of them.

So if we recurse on the the two half arrays, at least one of the two recursive calls will return the correct element x. We don't know for certain that the element x is indeed dominant in the entire array, but we can iteratively check every element of array against x, and double check that there are at least n/2 copies present in the entire array. We perform this check up to twice (once for each returned element from the recursive calls), so this check takes $O(n)$ time total.

Thus the recurrence relation for this algorithm is $T(n) = 2T(n/2) + O(n)$, which leads to an $O(n \log n)$ running time.

   (b) (Bonus) Find a linear time way to do this, prove that it is correct, and analyze it's runtime.

Solution: There is a bug in this problem related to handling the extra entry in odd arrays. See the discussion on Piazza.

The idea behind the proof of correctness relies on the idea if an element is the dominant element of an array, then it will remain the dominant entry after one iteration of the pair/cancel algorithm. For an even length array, say there are $d > n/2$ dominant entries, and $n - d$ non-dominant entries. Say $2k \leq d$ of the dominant entries occur in k pairs, $d - 2k$ are unpaired with $d - 2k$ non-dominant entries, and the remaining

$n + 2k - 2d$ are non-dominant. Thus after one step, we are left with k dominant entries and $n/2 - d + k$ non-dominant entries. As $2d > n$, there are strictly fewer non-dominant entries as dominant entries.

The amount of work we do is O(n) at each level, and we decrease the size of the problem by at least half, so our recurrence relation is $T(n) < T(n/2) + O(n)$, so that $T(n) = O(n)$.

There are two possible ways to fix the algorithm to remove the bug. The first was discussed on Piazza.

The second, simpler, solution is to simply examine this extra entry. Check against the entire array to see if the extra entry is the dominant entry in $O(n)$ time. If the entry isn't the dominant entry, we can toss it out and use the previous argument. If it is the dominant entry, then we are done! We still only add $O(n)$ complexity to each phase, so the runtime is the same order.

After we are done, we do a final check to make sure the entry we are returning is the dominant entry (we might otherwise return a false positive).

4. (Group) Recall the Merge subroutine of the MergeSort algorithm, which takes as input two sorted lists L_1, L_2, of size m and n, and outputs a list L with a combination of the elements in $O(m + n)$ time. Prove that the Merge algorithm will always return a *sorted* list; that is for any two adjacent elements, $L[i] \leq L[i + 1]$. You may want to directly compare elements, and you may have a couple of cases. We're looking for an airtight proof.

Solution: We are give two sorted lists L_1, L_2. Since they are sorted,

$$L_1[0] \leq L_1[1] \leq L_1[2].... \leq L_1[p].. \leq ..L_1[m-2] \leq L_1[m-1]$$

and similarly

$$L_2[0] \leq L_2[1] \leq L_2[2].... \leq L_2[q].. \leq ..L_2[n-2] \leq L_2[n-1].$$

We denote L as the list that our merge algorithm constructs from L_1 and L_2. To prove that the merge algorithm returns a sorted list, it suffices to show

$$L[0] \leq L[1] \leq L[2].....L[m+n-2] \leq L[m+n-1]$$

Let us prove this by Induction.

Statement : The list returned by merge algorithm, L, is sorted. That is for any $t \geq 1$, $L[t] \geq L[t-1]$

Base case: Show that the statement holds for $t = 1$.

By the merge algorithm $L[0] = \min\{L_1[0], L_2[0]\}$, which implies that $L[0] \leq L_1[0]$ and $L[0] \leq L_2[0]$, and that $L[0]$ is a minimum of both lists. Therefore $L[1] \geq L[0]$ for any $L[1]$. Hence the statement is true for $t = 1$.

Inductive step:

By inductive hypothesis, let us assume that the statement holds for $t \leq k$. Now let us show that the statement holds for $t = k + 1$.

Assume that the merge algorithm is comparing $L_1[p]$ and $L_2[q]$ to decide which element to insert as the $k + 1$ element. Without loss of generality let us assume $L_1[p] \leq L_2[q]$, and $L[k + 1] = L_1[p]$. From the merge algorithm, we know $L[k]$ must either be $L_1[p - 1]$ or $L_2[q - 1]$.

If $L[k] = L_1[p - 1]$, then we know $L_1[p] \geq L_1[p - 1]$, so $L[k + 1] \geq L[k]$.

If $L[k] = L_2[q - 1]$, then we know the previous step in merge must have compared $L_1[p]$ and $L_2[q - 1]$, and chosen $L_2[q - 1]$. This implies $L_1[p] \geq L_2[q - 1]$, so $L[k + 1] \geq L[k]$.

We have therefore shown that $L[k + 1] \geq L[k]$

5. (Group) You're given an array of n numbers. A *hill* in this array is an element $A[i]$ that is at least as large as it's neighbors on either side. In other words, $A[i] \geq A[i - 1]$ and $A[i] \geq A[i + 1]$. (for the boundaries, position $i = 1$ is a hill if $a[1] \geq a[2]$, resp. $i = n$ is a hill if $a[n] \geq a[n - 1]$.)

   (a) Give a brute force algorithm that can find a hill. Explain your algorithm in words and analyze its running time.

   Solution: Go through every possible element and check if it's a hill or not. This takes $O(1)$ work per element, so $O(n)$ time total. (Alternatively, search for a global maximum in $O(n)$ time - this is definitely a hill.)

   (b) Give a divide and conquer algorithm, with proof of correctness and analysis of running time, that will return *some* hill in the array in $\log(n)$ time.

   Solution: If $n \leq 2$, then return the larger (or only) element in the array. Otherwise compare the two elements in the middle of the array, $A[\frac{n}{2} - 1]$ and $A[\frac{n}{2}]$. If the first is bigger, then recurse in the first half of the array, otherwise recurse in the second half of the array.

   Why is this a valid solution? We will give a proof by induction. You should be able to see why the base case when $n \leq 2$ is correct.

   Assume that the algorithm works correctly for all size inputs smaller than n. We need to show that it works for size n. Assume without loss of generality that we recurse in the first half of the array, $A[0 \ldots \frac{n}{2} - 1]$.

   By assumption, the recursive call correctly returns a hill in its respective subarray - meaning that it returns an element in that sub array that is bigger than its neighbors. We have two cases:

   - This element is not on the boundary of the subarray - i.e. not element $A[\frac{n}{2} - 1]$. Then this element is also a hill in the larger array, as it's still $\geq$ its neighbors.
   - This element is $A[\frac{n}{2} - 1]$. Since this was a hill in the first subarray, we know that $A[\frac{n}{2} - 1] \geq A[\frac{n}{2} - 2]$. Since we decided to choose the first half of the array to recurse $A[\frac{n}{2} - 1] \geq A[\frac{n}{2}]$. Thus this is indeed a hill!

We could have made the mirror argument if we had chosen the second array to recurse in.

By induction, we see that the element returned will be a hill, and the algorithm is correct. We do $O(1)$ work before making the recursive call, so the recurrence relation is $T(n) = T(n/2) + O(1)$, which leads to an $O(\log n)$ running time.

(c) Similarly, a $k - hill$ is an element that is at least as large as its k neighbors on each side. By this definition, a *hill* is just a $1 - hill$. Modify the above divide and conquer algorithm so that it will find a $k - hill$. Analyze its running time, in terms of n and k.

Solution: We modify the algorithm above very slightly. In part b, we chose the half of the array with the larger of the two elements in the middle - there were the two boundary elements of the two subproblems.

For this algorithm, we instead look at the $2k$ elements in the middle, with $A[\frac{n}{2} - k \dots \frac{n}{2} - 1]$ being the k right most boundary elements on the first half of the array, and $A[\frac{n}{2} \dots \frac{n}{2} + k - 1]$ being the k left most elements on the second half of the array. We recurse down whichever side has the maximal element (if there are ties, it doesn't matter). If the array is small $\leq 2k$, then we can brute force find the maximal element in $O(k)$ time.

How does our proof of correctness change? The base case is brute forced to be correct.

By the inductive hypothesis, the recursive call correctly returns a hill in its respective subarray - meaning that it returns an element in that sub array that is bigger than its k neighbors in that subarray. Again, assume wlog that we chose the first half. We have two cases:

- This element is not on the boundary of the subarray - i.e. more than k away from the midpoint. Then this element is also a hill in the larger array, as it's still $\geq$ its k neighbors.
- This element is within k elements of the midpoint. Again, it's a hill in the subarray it's $\geq$ all its k neighbors to the left, as well as any neighbors it has to its right before the midpoint. Since we decided to choose this half of the array to recurse, it is also $\geq$ all the k elements on the left side of the other subarray. Thus, it is larger than AT LEAST k elements to its right as well. Thus, it's a hill in the larger array.

By induction, the algorithm is correct. At each step, we need to find the maximum of $O(k)$ elements, which takes $O(k)$ time. Our recurrence relation is then $T(n) = T(n/2) + O(k)$, which leads to an $O(k \log n)$ algorithm.

(d) (Bonus) Give a divide and conquer algorithm, with proof of correctness and analysis of running time, that will find a 1-hill in a *two*-dimensional nxn array. Here a 1-hill needs to be at least as large as its four neighbors above, below, to the left, and to the right. You need to at least asymptotically beat the brute force algorithm.

Solution: This is again the same idea as above - we instead cut our square into four quarters. The boundary points are the union of a 2xn and a nx2 strip of entries crossing the middle of the original square. Like in parts b and c, we choose a maximal element among these boundary points and recurse in the associated quarter. If you've followed the solutions to parts b and c, you can see why this algorithm is correct.

Consider the maximal element in the quarter that we recurse in. If it is not on the boundary, then it is an internal node that must be a hill. If it is on the boundary, then it must be this maximal element chosen in the above step, since that is the maximal element on the entire boundary. It is at least as large as any boundary point, and at least as large as any point inside the quarter -¿ thus this maximal element is a hill. In either case, we recurse on a quarter that definitely contains a hill, which is all that we need for this algorithm to be correct.

Before recursing, all we do is find the maximum of all the $4n - 4$ elements crossing through the center of the square (count them!), which leads to a recurrence relation of $T(n) = T(n/2) + O(n)$ (remember, n is the side length), which leads to an $O(n)$ running time by the Master Theorem.

There are many seemingly viable algorithms for this problem, but most of them have bugs.