Homework 2

Student: Solutions

1. Solve the following recurrences. You can use the Master Theorem for some of these, where applicable, but state which case you are using and show your work. You may give the answers in Big-O notation. Justify your answers.

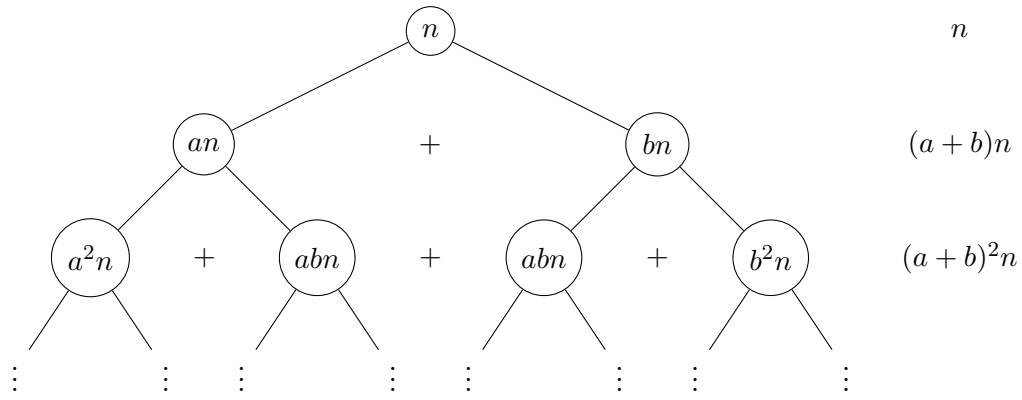
   (a) $T(n) = T(n/5) + T(n/3) + n$

Solution: In this solution we'll actually prove something more general. Specifically that, if $T(n) = T(an) + T(bn) + n$ where $a, b > 0$ and $a + b < 1$ then $T(n) = \Theta(n)$. (Notice that this more general statement applies to the running time of the deterministic median algorithm given in class.)

At each level, we have 2^i calls, but wait! 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! Assume without loss of generality that $a < b$. Then, there are at most $\log_b n$ levels, and at least $\log_a n$ complete levels. But either way,

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

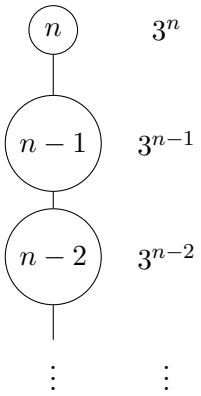
So, $T(n) = \Theta(n)$! Note that the upper bound above comes from looking at the sum to ∞ which is an overestimate of the total bound. The lower bound comes from just looking at the first level of the tree where we do n work which is an underestimate of the total work done.



   (b) $T(n) = 23T(n/4) + \sqrt{n}$.

Solution: We will use the master theorem with $a = 23, b = 4, k = 0.5$ and $b^k = \sqrt{4} = 2$. We have $a > b^k$ so we are in the third case and $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_4 23})$.

   (c) $T(n) = T(n-1) + 3^n$.



Solution:

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 $n-i$, does 3^{n-i} "work", and there will be n levels total.

That means the total work done is

$$\sum_{i=0}^{n} 3^{n-i} = \sum_{j=0}^{n} 3^j = \frac{3^{n+1}-1}{3-1} = O(3^n)$$

2. An array $A[1 \ldots n]$ is said to have a *dominating entry* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a dominating entry, and, if so, to find that entry or element. The elements of the array are abstract objects, and not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Show how to solve this problem in $O(n \log n)$ time using a **divide and conquer** approach.

   (a) Explain your algorithm in words and justify why it is correct.

   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 pigeonhole 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 (assuming there is a dominant element). We don't know for certain that either of the element(s) returned is indeed dominant in the entire array, but we can check this. To check if an element x is dominant in the array, we can loop through and compare each element to x and count the number that are equal in order to determine if there are at least n/2 copies of x present in the entire array.

   (b) Provide well-commented pseudocode.

Solution: Input: array $A = [a_1, \ldots, a_n]$
Output: A dominating entry if one exists. otherwise -1.

```
function DOMINATINGENTRY(A)
    if n = 1 then return a_n
    x = DOMINATINGENTRY([a_1, ... a_{n/2}])
    if ISDOMINATING(x,A) then return (x)
    y = DOMINATINGENTRY([a_{n/2+1}, ... a_n])
    if ISDOMINATING(y,A) then return (y)
    return -1

function ISDOMINATING(x, A = [a_1, ..., a_n])
    if x == -1 then return False
    count = 0
    for (i = 1; i ≤ n; i++) do
        if a_i == x then count = count + 1
    if count >= n/2 then return True
    else return False
```

(c) Analyze your algorithm including stating and solving the relevant recurrence relation (and justifying your analysis).

Solution: We make two recursive calls, one for the left side and one for the right side each of size $n/2$. Outside of the recursive calls, we check if the elements returned are dominant in the entire array. This check involves comparing them to every element in the array and we perform this check up to twice (once for each returned element from the recursive calls), so this 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 (this is the same recurrence as Mergesort which we have solved before).

3. Recall the maximum subarray problem discussed in class (given (possibly negative) integers $a_1, a_2, \ldots, a_n$, find the maximum value of $\sum_{k=i}^{j} a_k$). Design a **divide and conquer** algorithm to solve this problem. Your algorithm should divide the array in half as in Merge Sort.

(a) Explain your algorithm in words and justify why it is correct.

Solution: Our algorithm will return the sum of the maximum subarray. If the input array has size 1 then it will return a_1. Otherwise, there are three possibilities for the maximum subarray; either it is (1) contained in the left half of the array, (2) contained in the right half or (3) crosses the middle. Our algorithm will divide the array in half and recurse on each half of the array. This covers the first two possibilities - (1) and (2). Next we'll find the maximum subarray that crosses the middle (3). To do this we'll find the best starting point on the left side and the best ending point on the right side. To find the best starting point on the left side, we'll start at the middle and iterate

to the left, keeping track of the best starting point so far and the cumulative sum. We'll do the same on the right side to find the best end point. Finally we'll return the maximum of the three subarrays we've found (1), (2) and (3).

(b) Provide well-commented pseudocode.

Solution:
Input: array $A = [a_1, \ldots, a_n]$
Output: The value of the maximum subarray.

```
function MAXSUBARRAY(A)                                          ▷ base case
    if n = 1 then return a_n
                                          ▷ solve the problem recursively on each 1/2
    leftMax = MAXSUBARRAY([a_1, ... a_{n/2}])
    rightMax = MAXSUBARRAY([a_{n/2+1}, ... a_n])
                                          ▷ find the best start point on the left
    bestSoFarL = a_{n/2}
    sumSoFarL = a_{n/2}
    for (i = n/2 − 1; i ≥ 1; i − −) do
        sumSoFarL = sumSoFarL + a_i
        if sumSoFarL > bestSoFarL then bestSoFarL = sumSoFarL
                                          ▷ find the best end point on the right
    bestSoFarR = a_{n/2+1}
    sumSoFarR = a_{n/2+1}
    for (i = n/2 + 1; i ≤ n; i + +) do
        sumSoFarR = sumSoFarR + a_i
        if sumSoFarR > bestSoFarR then bestSoFarR = sumSoFarR
                                          ▷ compute the best sol'n that crosses middle
    crossMax = bestSoFarL + bestSoFarR
                                          ▷ return the max of the three possibilities
    return max(leftMax, rightMax, crossMax)
```

(c) Analyze your algorithm including stating and solving the relevant recurrence relation (and justifying your analysis).

Solution: Outside of the recursive calls, our algorithm does $O(n)$ work - each for loops takes time $O(n)$. We make two recursive calls, one for the left side and one for the right side each of size $n/2$. This gives the following recurrence relation:

$$T(n) = 2T(n/2) + cn,$$

which is the same as merge sort and thus solves to $O(n \log n)$.

4. (6 points) 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. In other words, $A[i] \geq A[i-1]$ and $A[i] \geq A[i+1]$. (If $i = 1$ is a hill then we only need $A[1] \geq A[2]$, resp. if $i = n$ is a hill if $A[n] \geq A[n-1]$.) Consider the

following divide & conquer algorithm that solves the hill problem:

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? Think about why this algorithm is correct.

Solution: (Note that you were not asked to submit an answer to this quesiton.) We will use induction to show the algorithm is always correct. First, we will show that the base case when $n \leq 2$ is correct. If there are 2 elements then the larger is returned. From the definition this is clearly a hill since it is larger than its only neighbor. If there is only one 1 element then it is returned. Again, from the definition this is a hill since it has no neighbors. Notice that we will always return a hill because there is always a hill in our base case (this means every possible array has a hill!).

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.

For this problem, you will implement two different solutions for the hill problem. You will add code to the `Assignment2.java` file provided with the assignment.

(a) Give a **brute force** algorithm that can find a hill.

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 always a hill.)

(b) Implement the divide and conquer algorithm given in Problem 4 by adding code to the `divideAndConquerHill` method. Note that you will need to modify the algorithm given above slightly to return the index of the hill instead of the element. As a comment at the top of the `divideAndConquerHill` method state the recurrence relation and analyze its running time.

Solution: We do $O(1)$ work before making the a single recursive call on half the array, so the recurrence relation is $T(n) = T(n/2) + \Theta(1)$. We will use the master theorem with $a = 1, b = 2, k = 0$ and $b^k = 2^0 = 1$. We have $a = b^k$ so we are in the second case and $T(n) = \Theta(n^k \log n) = \Theta(\log n)$. Note that this is the same recurrence as binary search.