CISC 380 **Instructor: Dr. Sarah Miracle**

Homework 2

Student: Nick Hillemeier and Haoyi Shi

Purpose:

- Practice solving recurrence relations.
- Learn to design (and analyze) algorithms using a divide and conquer approach.

General Homework Policies:

- This homework assignment is due by the start of class on Monday, March 2. In order to submit your homework, you must do two things.
  1. Submit the `pdf` (it should generated by modifying the LaTeX template provided) and the completed `Assignment2.java` file through Canvas. Before submitting please zip all the files together and submit one `.zip` file. See below for more detailed submission instructions.
  2. Bring a paper print-out of the pdf to class.

  Late assignments will not be accepted and will receive a 0!
- You will be assigned a partner for this assignment. Only one assignment should be submitted and you and your partner will both receive the same grade. Make sure to include your partner's name on the homework. Your assignment should be a true joint effort, equally created, and understood by both partners.
- Getting *ANY* solutions from the web, previous students etc. is *NOT* allowed.[1]
- Copying code from anywhere or anyone is not allowed (even previous code you have written). Allowing someone to copy your code is also considered cheating.[1]
- You are not allowed to discuss this assignment with anyone except for your partner (if you have one) or the instructor.[1]
- Your work will be graded on correctness and clarity. Write complete and precise answers and show all of your work (there is a detailed grading rubric included at the end of the assignment).
- Questions marked (*PRACTICE*) will not be graded and do not need to be submitted. However it's highly recommended that you complete them.

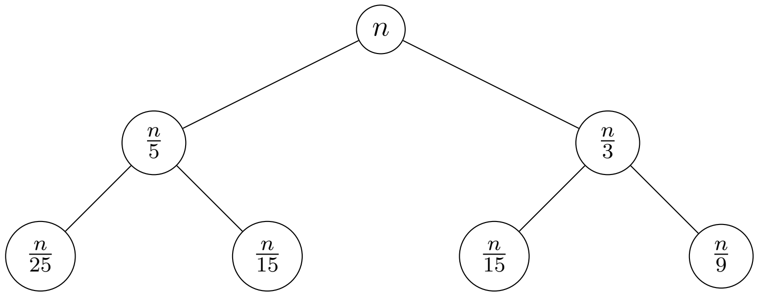
[1]*See the section of the syllabus on academic dishonesty for more details.*

Homework Problems:

1. (3 points) 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 should give a Θ bound. Show your work and justify your answers (this should include justifying why your answer is a lower AND an upper bound).

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

Solution: To solve this recurrence, we first can create a tree to see if we can find the number of levels to our tree and how much work is done at each level.



Now that we have a tree, we can figure out the levels and how much work is done. At the first level, n work is done. At the second level, $\frac{n}{5} + \frac{n}{3} = \frac{8}{15}n$ work is done. At the third level $\frac{n}{25} + \frac{n}{15} + \frac{n}{15} + \frac{n}{9} = \frac{64}{225}n$ work is done. Here, we can see that each level does $(\frac{8}{15})^i n$ work, where i is the level in the tree. Now we need to find how many levels in the tree there are. On the left side of the tree, there are $\log_5 n$ levels, and on the right side there are $\log_3 n$ levels. Since they are not the same, we do not have a full tree, so we need to bound $T(n)$ to solve the recurrence. So, the upper bound will be $T(n) \leq \sum_{i=0}^{n/3}(\frac{8}{15})^i$ and the lower bound will be $T(n) \geq \sum_{i=0}^{n/5}(\frac{8}{15})^i$. Now, since $\frac{8}{15}$ is less than 1, we can use the infinite geometric series to solve this. So, $T(n) \leq \sum_{i=0}^{\infty} \frac{1}{1-\frac{8}{15}} = \frac{15}{7}$ and $T(n) \geq \sum_{i=0}^{\infty} \frac{1}{1-\frac{8}{15}} = \frac{15}{7}$. Since these are the same, the upper and lower bounds are the same, so we will have a theta bound. Now, since we know the levels, we can multiply by the work to solve the recurrence. So, $\frac{15}{7} * (\frac{8}{15})^i n = \Theta(n)$. So, $T(n) = \Theta(n)$.

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

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

2. (7 points) 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: For our solution, we have three base cases. If the length of the list is of length one, then return the element, if the length of the list is of length two, then check to see if the elements are equal, and if they are return it, and if the list is of length three, check to see if two out of the three elements are the same, and if they are, return it. Now for larger arrays, we split the array in half and recursively find the dominant entry in each half. Once we find that, if the left recursive dominant entry equals the right recursive dominant entry, we return it, since if the dominant entry for both sides is the same, it's the dominating entry of the entire array. Otherwise, we loop through the array from the start index passed to the end index passed and check each element of the array against the two dominating entries, and increment a counter if the element matches the left dominant entry, or increment a different counter if the element matches the right dominant entry. After the loop completes, we check both counters to see if they are greater than half the size of the array, and if one of the counters is, we return it, since that means that one of the dominant entries occurs more than half of the time in the array. If we can't find a dominating entry, we return the minimum integer value, so that if there isn't a dominating entry, the result will be the minimum int value.

(b) Provide well-commented pseudocode.

Solution:

```
function dominatingEntry(A[0.....n-1], start, end)
    if (end - start = 0)
        return A[start]             //Returns if only one element
    else if (end - start = 1)
        if A[start] = A[end]
            return a[Start]         //Returns if there are to elements
                                    //and they are equal
    else if (end - start = 2)
        if (array[start] = array[end] OR array[start] = array[start + 1])
            return array[start]     //Returns if the first two or the
                                    //first and last elements of three
                                    //are the same
        else if (array[start+1] = array[end]
            return array[end]       //Returns if the last two elements
                                    //of three are the same
    else
        mid = (end+start)/2         //Gets the mid point
        leftCount = 0
        rightCount = 0              //Sets left and right counts to 0
        dominantLeft = dominatingEntry(A, start, mid)   //Recursive
        dominantRight = dominatingEntry(A, mid+1, end)  //Calls
        if (dominantLeft = dominantRight)
            return dominantLeft     //Returns if the left dominant
                                    //and right are the same
        for (i=0, i < end-start+1, i++) //Loops through array
            if (dominantLeft = A[i])
                leftCount++         //Increments if the element at the
                                    //ith position is equal to the
                                    //value returned by dominantLeft
            else if (dominantRight = A[i])
                rightCount++        //Increments if the element at the
                                    //ith position is equal to the
```

```
                                  //value returned by dominantRight
        if (leftCount > (end-start+1)/2)
            return dominantLeft   //Returns if leftCount is greater
                                  //than half the size of the array
        else if (rightCount > (end-start+1)/2)
            return dominantRight  //Returns if rightCount is greater
                                  //than half the size of the array
    return minimumInt             //Only returns if there isn't a
                                  //dominating entry for the input
```

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

Solution: The big-O bound for our algorithm is $O(n \log n)$, and our algorithm is also $\Theta(n \log(n))$. For our runtime, we have base cases that take constant time, two recursive calls that take half the size of the array as input, a loop that loops from the start index to the end index of the array passed to the function, and if statements and return statements that are constant time. Our worst case is that we have to loop through the full length of our input array. So, our recurrence relation is $T(n) = 2T(\frac{n}{2})+n$. We can use the Master Theorem to solve this recurrence, where $a = 2$, $b = 2$, and $k = 1$. So, with $a <=> b^k$ as our guide, $2 = 2^1$, so our solution will be $\Theta(n^k \log n)$. With $k = 1$, $T(n) = \Theta(n \log n)$. This is correct because we keep on splitting the input size in half, which means we have $\log n$ levels, and we have one loop that runs at the worst case through the entire length of the array, which is n, so when we multiply those together we get $\Theta(n \log n)$.

(d) (*EXTRA CREDIT*) Implement your pseudo-code from part (b). Add code to the `dominant` method in the `Assignment2.java` file provided with the assignment. Note that for simplicity you are implementing this algorithm for an array of integers. However, you are still not allowed to compare elements in the array only to check for equality (i.e. no $A[i] > A[j]$).

3. (5 points) Recall the maximum subarray problem discussed in class (given (possibly negative) integers $a_1, a_2, \ldots, a_n$, find the maximum value of $\sum_{k=i}^{j} a_k$). You will design and implement a **divide and conquer** algorithm to solve this problem. Your algorithm should divide the array in half as in Merge Sort and have running time $\Theta(n \log n)$. Complete the partial code given to you in the `maxSubArrayRecursive` method within file `Assignment2.java`. Note that if the array contains only negative numbers then the answer should be negative. For example, if the input is `[-1, -2, -3]` then your method should return -1 (i.e. a subarray has length at least 1).

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 is a correct algorithm? We will discuss further in class.

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. As a comment at the top of the `bruteHill` method explain your algorithm in words and analyze its running time. Then implement your algorithm by adding code to the `bruteHill` method.

(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.

Code is given in the `Assignment2HillDriver.java` file that compares the performance of the two methods. You should run this code and think about the output. Note that the code does not thoroughly test your methods - this is something you should do!

Additional Programming Instructions:

Note that your code will be automatically run on a standard set of test cases. In order to ensure that you do not lose points, follow the instructions below.

- Your code must compile without any errors using the version of Java on the lab computers. If your code does not compile you will not receive any points for the assignment.
- Do not modify any of the methods signatures (i.e. name, return type or input type). Note that you are always welcome (and encouraged) to add additional methods but these will not be run directly by the test code.
- You are not allowed to use packages (e.g. no statement `package ...` at the top of your file).
- No extra folders or files in your submission. Zip up only the files you need to submit not the folder they are in.

Grading Criteria:

Your work will be graded on both correctness **and clarity**. Write complete and precise answers and show all of your work. Your pseudo-code and proofs should follow the guidelines posted on Canvas and discussed in class.

Evaluation Rubric for Problem 1

Component	Requirement for Full Credit
Solution Correctness (1pt)	The answer is correct.
Written Explanation (2pt)	Your solution includes a logical argument explaining why your answer is correct. The explanation is clear, correct and complete. It includes full sentences and is easily understood by any student in the class. Think of this as a proof and remember what you learned in Math 128.

Evaluation Rubric for Problem 2

Component	Requirement for Full Credit
Solution Correctness (3pts)	The algorithm will work correctly on all inputs and uses a divide and conquer strategy.
Written Description & Justification (2pt)	The explanation of the algorithm and why it is correct is clearly written and easily understandable. It includes full sentences and should be easily understood by any student in the class.
Pseudo-code (1pt)	The pseudo-code is clearly written following the guidelines discussed in class.
Running Time Analysis (1pt)	The solution gives a recurrence relation and solves it. It is clearly written and correct (for the algorithm given!).
Implementation (EXTRA CREDIT) (1pt)	Your code will be run on a standard set of test cases. Make sure to test your code thoroughly! Note that you will lose points if you do not follow the general style guidelines given in the syllabus or if your implementation changes the running time of your pseudo-code. You will receive NO credit if your code does not implement your pseudo-code (which must use a divide and conquer approach) or if you compare elements in the array directly.

Evaluation Rubric for Problem 3.

Component	Description
Style & Documentation (1 pts)	I'll be watching for style issues as well as correct output. See the syllabus for some general style guidelines (e.g. your code should be well-documented).
Correct Output on Test Cases* (4 pt)	Your code will be run on a standard set of test cases.

*Note that if your code does not implement a divide and conquer algorithm (using the given starter code) with the appropriate running time, you will lose credit regardless of whether the test cases execute correctly.

Evaluation Rubric for Problem 4.

Component	Description
Style & Documentation (1 pts)	I'll be watching for style issues as well as correct output. See the syllabus for some general style guidelines (e.g. your code should be well-documented).
brute force Description & Big-O Analysis (1 pt)	Your *brute force* algorithm explanation should be in full sentences and contain as much detail as pseudo-code. Make sure to justify your analysis as well as given a bound (i.e., explain where your bound comes from and why it is correct).
divide & conquer Big-O Analysis (1 pt)	Make sure to justify your analysis as well as given a bound. This must include stating (and justifying) a recurrence relation as well as solving it. If applicable, you may use the Master Theorem but make sure to state which case you are using and show your work.
Correct Output on Test Cases* (2 pt)	Your *brute force* code will be run on a standard set of test cases.
Correct Output on Test Cases* (2 pt)	Your *divide & conquer* code will be run on a standard set of test cases.

*Note that if your code does not implement the appropriate algorithm you will receive no credit regardless of whether the test cases execute correctly.