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

# Homework 2

Student: Haoyuan Du, Keghan Halloran

**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 deadline given in Canvas. Late assignments will not be accepted and will receive a 0.
- Submit both 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 the *Additional Programming Instructions* section below for more detailed instructions.
- 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.
- You are not allowed to consult outside sources, other than the textbook, your notes, the Java API and the references linked from Canvas (i.e., no looking for answers on the internet).[1]
- 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.

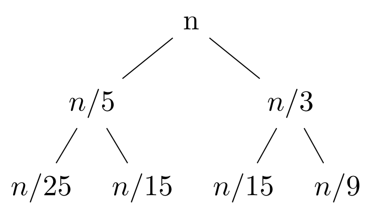
[1]*See the section of the syllabus on academic dishonesty for more details.*

**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 $\Theta$ 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:**



According to the recursion tree, we know that the leftmost branch will be the shortest branch because n is being divided by 5 each level, as opposed to being divided by 3 on each level of the rightmost branch. Because of this, we know that the leftmost branch is the lower bound of T(n), and the rightmost branch, which is the longest, is the upper bound of T(n).
For the second level, $\frac{8}{15}n$ work is done. For the third level, $(\frac{8}{15})^2n$ work is done. So, each level, $(\frac{8}{15})^in$ work is done. We also know that the left most branch will have $\log_5 n$ levels and the right most branch will have $\log_3 n$ levels.
The equation for total work done is as follows. Number of levels $*$ work is done per level = Total Work. Thus, for the lower bound, $T(n) \geq n * \sum_{i=0}^{\log_5 n}(\frac{8}{15})^i$, which is $\frac{\frac{8}{15}^{\log_5 n+1}-1}{\frac{8}{15}-1}$, since $\frac{8}{15} < 1$, the answer will be $\frac{1}{1-\frac{8}{15}}$, which is $\frac{15}{7}n$. Similarly, for the upper bound, $T(n) \leq n * \sum_{i=0}^{\log_3 n}(\frac{8}{15})^i$, which is $\frac{15}{7}n$. Therefore, we get a $\theta$ bound of $\theta(n)$.

(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:** Our algorithm has a base case and a recursive part. In the base case, if the length of current array is 1, then we return the only element. If the length of current array is 2, then return the first element.
In the recursive part, we divide the current array by 2 parts if its length is greater than 2 and execute the recursion. The recursion will return both LEFT sub-array and RIGHT sub-array values. Then we use a for loop to go through the current array and figure out how many times each value occurs respectively. If one of the values makes up more than half of the array, we return this value. Otherwise, we return null.
Our algorithm is correct, because if a value is a dominant entry, even after we divide the whole array into pieces of length of 2 or length 1, at least one of the sub-arrays will return the dominant entry, because of the "Pigeonhole Principle". Once the dominant entry or entries from the subarrays are returned, the for loop goes through the parent array of the subarrays and counts the number of occurences of the dominant entry/entries from the sub arrays. If one of them is dominant in the parent array, it

will be returned, otherwise, we return null.

(b) Provide well-commented pseudocode.

**Solution:**

```
function DominantEntry(A[$a_0$,...$a_{n-1}$], low, high)        //low and high are integers
    if (high - low = 0)
        return A[low]        //if the array only has one element, return this element
    else if (high - low = 1)
        return A[low]        //if the array has two elements, return the left one
    else
        left = DominantEntry(A[$a_0$,...$a_{n-1}$], low, (low+high)/2)        //recursive
        right = DominantEntry(A[$a_0$,...$a_{n-1}$], (low+high)/2+1, high)        //calls
        for i=low, i $\leq$ high, i++ do        //loop in current array
            if (left != null)
                if (left = A[i])
                    countLeft++        //count left occurrences
            if (right != null)
                if (right = A[i])
                    countRight++        //count right occurrences
        if (countLeft > $(high - low + 1)/2$)        //return the dominant if
            return left                              //occurrences > half array
        else if (countRight > $(high - low + 1)/2$)
            return right
        else
            return null                              //if no dominant, return null
```

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

**Solution:** Our algorithm is $\theta(nlogn)$. Firstly, our base case takes constant time. Then we have two recursive calls, each call is half of the array. Next, we have a for loop which in the worst case will be executed n times. Finally, the if statement takes constant time. Thus, the recurrence is $T(n) = 2 * T(n/2) + n$. We can use the master theorem to solve this. a = 2, b = 2, k = 1, so 2 = $2^1$, which is $\theta(n^k logn)$. Since k=1, our $T(n) = \theta(nlogn)$. This is reasonable because we call the recursive calls twice, each recursive call takes a half of current array. Meanwhile, the for loop loops through the low to high index which is also half of the current array. The worst case of the for loop is when it iterates the whole original array which is n. So our result is $\theta(nlogn)$.

(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 a valid solution? Think about why this 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. 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.

- Your solution should not print anything unless explicitly instructed to.

---

**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.