CIS 675 (Spring 2021) Disclosure Sheet

Name: Yuchen Wang
HW # two

[✓ **Yes**] **No** Did you consult with anyone on parts of this assignment, including other students, TAs, or the instructor?

Yes [✓ **No**] Did you consult an outside source (such as an Internet forum or a book other than the course textbook) on parts of this assignment?

If you answered **Yes** to one or more questions, please give the details here:

I asked sara (TA) and prof. Sucheta questions on Q2 and Q3.

By submitting this sheet through my Blackboard account, I assert that the information on this sheet is true.

This disclosure sheet was based on one originally designed by Profs. Royer and Older.

Homework 2:
Due March 4 at midnight Eastern time. Submit your solutions typed and in a pdf document. To receive full credit, explain your answers.

If you collaborate with another student or use outside sources, please list those students' names and the URL/title/etc. of the sources that you referred to. Collaboration is permitted, but you must write up your own solutions.

1. Suppose that you are implementing a version of MergeSort in which the subarrays *cannot* be created by using pointers to the original array; instead, you must allocate new space and copy over the appropriate elements one by one. Assume that allocating space does not take any time.

1. Analyze the running time of this version of MergeSort.
   - $T(n) = 2T(n/2) + c1 + c2n = 2T(n/2) + O(n), a = 2, b = 2, d = 1$. As $log_b a = log_2 2 = 1 = d$. Hence, $T(n) = nlogn$.
   - We have $2T(n/2)$, as a = 2 and b = 2. a = 2: There are two recursive calls generated in each recursive case. b = 2: Input data is divided into left half and right half in each recursive case, so we have input data size for recursive calls as half of the recursive calls in higher recursive level.
   - c1 stands for the constant time consumed at each recursive level aside the merging process
   - c2 stands for the time consumption of comparing two array elements in the merging process, and there is total n comparison in each recursive level.
2. Analyze the *space usage* (i.e., in memory) of this version of MergeSort. You only need to consider the space used by arrays and sub-arrays, not temporary helper variables. Assume that an array of size k requires k units of space. Assume that nothing is ever deleted.
   - The space usage of each recursive level = the size of original input array = n. MergeSort cuts input array in half at each recursive level, thus the number of input array doubles every level and the

size of input array shrinks to half. However, the total space usage of each recursive level remains unchanged.

- There are $log_2 n$ level in total, because input arrays are cut in half at every recursive level until the base case is encountered(array size = 1).
- Hence, the total space usage $= log_2 n * n = nlog_2 n = O(nlog_2 n)$ logically.
- However, under the real world situation, the space usage is actually $\mathbf{O(n)}$, because the mergeSort does not generate its recursive branches in parallel. At run time, it needs space of n at root level and second level, n/2 at third level, n/4 at fourth level, ..., and 2 at leaf level. Hence, aside the root level, the need of space is a geometric series with common ratio of $\frac{1}{2}$ and it takes totally $n\frac{1-(\frac{1}{2})^{logn}}{1-\frac{1}{2}} = 2n(1-\frac{1}{n}) = O(n)$. And as root level need space of n, the overall space complexity is still $O(n)$.

2. Suppose you are using Binary Search to find a particular element k in a sorted array A. In the worst case, Binary Search takes O(log n) time to find the desired element, where n is the length of A (i.e., if we get all the way down to a subarray of size 1 before finding k). However, in the best case, the element you are searching for might be the first one that you check (i.e., if the value you are searching for is the middle element of the array), so the best-case running time is $O(1)$. In the *median* case, how many elements do you need to check? (In other words, if you make a list showing how many checks need to be performed before finding k, what would the median of this list be?). Report your answer in big-O terms. You must explain your answer. Hint: Think about how many elements are found with just 1 check. How many are found with 2 checks? 3? i?

- The number of searching demanded with binary search can be inducted as: $1, 2, 2, 3, 3, 3, 3, 4.......logn$. It is a sequence which the value of term increases by 1 every **P** terms, and **P** belongs to a geometric series with common ratio of 2. This sequence has n terms, because we have n element can be found with binary search.
- $\mathbf{T(n/2)}$ = the number of element the medium case needs to check, I use x to represent $\mathbf{T(n/2)}$ in the proof below:

  - With geometric series sum formula, we have
  - $\frac{1-2^x}{1-2} = \frac{n}{2}$
  - $2^x = \frac{n+2}{2}$
  - $x = log_2 \frac{n+2}{2}$
  - $x = O(logn)$

- Hence, $T(n/2) = O(logn)$

3. Suppose you are using an instrument to measure the depth of a body of water. You are in a boat on top of a large 'valley' in the lake, and want to know how deep this valley is. Assume this depth is an integer. Each time you fire the instrument, you specify a maximum distance d, and the instrument tells you whether the bottom of the valley is more or less than d units away (but it does not tell you the depth directly). Using this instrument is very expensive, so you want to minimize the number of times you use it. One way to use this instrument would be to initially set $d = 1$, then if the lake is deeper than that, set $d = 2$, and so on, until you find the depth. However, this algorithm will require many uses of the instrument. Design an efficient algorithm to determine how to set the d values so that you find the depth with as few uses as possible. How many times will the instrument be used?

- **Pseudocode of the algorithm is attached at end of the document.**
- **Claim**: **try(d)** returns true if $d >= ValleyDepth$
- **T1(n)** = the number of instrument use to guess the depth range which the valley's depth n belongs to = the times of **try()** processed in **FindDepthRange()** on the valley with depth of n.
  - **T1(n)** $= log_2 n = O(log_2 n)$.
  - **Explanation**: the value of max starts from 1, and doubles itself every while loop iteration till surpassing the value of n. Hence, **try()** is called $log_2 n$ time in while loop, as it takes $max\ log_2 n$ iteration to surpass n. At the end, it becomes to $O(log_2 n)$.
- **T2($\frac{n}{2}$)** = the number of instrument use to detect the depth of the valley over the range from $\frac{n}{2}$ to n = the times of **try()** processed in **BinarySearch()** on the valley with depth of n.

  - **T2($\frac{n}{2}$)** = 2T2($\frac{n}{4}$) = $log_2\frac{n}{2} = O(log_2n)$.
  - **Explanation**: This is a binary search, and it starts with input data size of $max - min = n - \frac{n}{2} = \frac{n}{2}$. Input data size shrink to half every iteration in binary search, thus, in the worst case, it takes $log_2\frac{n}{2}$ times to find the intended value on $\frac{n}{2}$ original data size. At the end, it becomes to $O(log_2n)$.

- **T(n)** = the total number of instrument use to detect the depth of the valley n = the times of **try()** processed during a call on **FindValley-Depth()** on the valley with depth of n.
  - **T(n)** = T1(n) + T2($\frac{n}{2}$) = $O(log_2n) + O(log_2n) = O(log_2n)$.

4. Suppose you have an unsorted array with some duplicates. You want to count the number of elements that are a duplicate of some other number. Design an efficient algorithm to compute this value. What is its running time?

- My algorithm can be broken into following steps
  1. Loop through the input array **'InputArray'**. Record the smallest number **min** and largest number **max**. This step takes $O(n)$ running time.
  2. Create a new integer array **'CounterArray'** with length of $max - min + 1$ and set all element of **'CounterArray'** to 0. This step takes $O(1)$ running time.
  3. First element of **'CounterArray'** represents **min** and the last element represents **max**. Create a new integer variable **counter**, and set it to 0. Loop through **'InputArray'** again, and set the value of specific **'CounterArray'** element to 1 when encountering the corresponding element in **'InputArray'**, and increase **counter** by one when the value of specific **'CounterArray'** element is already being 1. This step takes $O(1)$ running time.
  4. At the end we have 'the number of elements that are a duplicate of some other number' represented by **counter**. And the overall running time **T(n)** = $O(n) + O(1) + O(n) = O(n)$

Extra Credit. Suppose you have a sequence of sticks in a line. Assume these posts are numbered $1, ..., k$. You want to find the tallest stick and the shortest stick. However, you forgot your measuring scale, and the only means you have of determining this is by pulling out two sticks from the line, holding them in front of you, and seeing which one is shorter and which is taller. After this, you return them to their original places in the line. At no point do you learn their actual heights: all you can do is bring two sticks forward and see which one of the two is shorter/taller. Assume each such process takes 1 minutes.

(a) One thing you could do is go through the line sequentially, keeping track of indices of the tallest/shortest you've seen so far. At each iteration, you pull out stick i and compare to the tallest stick so far and the shortest stick so far, to see if i is the new tallest/shortest. What is the total running time (give this as an exact function of k, not big-O).

- **T(k)** = the total running time of question a algorithm on k sticks $= 2(k-2)+1 = (2k-3)$ minutes

    1. We compare stick 1 and 2, then we have the shortest and the tallest stick. This step takes 1 minute
    2. We then compare the rest of $(k-2)$ sticks with the shortest and the tallest stick, and keep updating the shortest and the tallest stick. This step takes $2(k-2)$ minutes.
    3. Hence, it takes totally $2(k-2)+1 = (2k-3)$ minutes to run the algorithm.

(b) Describe an algorithm that requires fewer than $1.5k$ minutes in the worst case.

- **T(k)** = the total running time of my algorithm on k sticks $= 1 + \frac{k-2}{2} + (k-2) = 1 + \frac{k}{2} - 1 + k - 2 = (1.5k-2)$ minutes

    1. We compare stick 1 and 2, then we have the shortest and the tallest stick. This step takes 1 minute
    2. We then group the rest of $(k-2)$ sticks into group of two sticks, and compare the two sticks in each group to find the shortest and

the tallest stick in the group. This step takes $\frac{k-2}{2}$ minutes, as we have $\frac{k-2}{2}$ comparisons and each comparison takes 1 minute.

3. we then compare the current shortest with the shortest in each group, and current tallest with the tallest in each group. We keep doing this to find the global shortest and the tallest stick. This step takes $(k-2)$ minutes.
4. Hence, it takes totally $1+\frac{k-2}{2}+(k-2) = 1+\frac{k}{2}-1+k-2 = (1.5k-2)$ minutes to run the algorithm.

The Algorithm for Question Three:

```
function FindValleyDepth( )
    $min = 0$, $max = 1$
    FindDepthRange($min$, $max$)
        return BinarySearch($min$, $max$)
end function

function FindDepthRange(unit min, unit max)
    while $True$
        if $try(max) == True$:
            return $max, min$
        else:
            $min = max$
            $max = 2max$
end function

function BinarySearch(unit min, unit max)
    if $try(\frac{max+min}{2}) == True$:
        if $try(\frac{max+min}{2} - 1) == False$:      //This is the base case.
            return $\frac{max+min}{2}$
        else:
            return BinarySearch($min$, $\frac{max+min}{2} - 1$)
    else:
        return BinarySearch($\frac{max+min}{2} + 1$, $max$)
end function
```