HW1 : ADA

Q1

a) $f(n) = n^n$, $g(n) = n!$

i) $f(n) \leq c \cdot g(n)$

$n^n \leq c \cdot n!$

$\frac{n^n}{n!} \leq c \cdot \frac{n}{e}$

$\lim_{n \to \infty} \frac{n^n}{n!} = \infty$ as n approaches infinity

The limit goes to infinity, because ~~f(n)~~ n^n grows much faster than $n!$

$\therefore f(n) \neq O(g(n))$

b) ii) We just proved that for sufficiently large n, n^n grows much faster than $n!$ hence it proves that $f(n) = \Omega(g(n))$ which means $f(n) \geq c \cdot g(n)$ will be true for it.

Yes, $f(n) = \Omega\, g(n)$

b) i) $f(n) \leq c \cdot g(n)$ taking C as 1, where c is a constant considering the limit

$n \log n \leq c \cdot (\log n)^n$

$\frac{n \log n}{(\log n)^n}$ Using L'Hopital rule, we can take the derivative.

$\lim_{n \to \infty} \frac{n \log n}{(\log n)^n}$ as limit approaches infinity = $\lim_{n \to \infty} \frac{\log n + 1}{n (\log n)^{n-1}}$

the nuretor grows much slower than the denominator, and limit approaches to zero.

This proves that $n \log n$ is asymtotically bounded by $(\log n)^n$ for very large n.

$\therefore f(n) = O(g(n))$

for $0 \leq c(\log n)^{*n} \leq n^n$, we can't find a constant c which satisfies $n \geq n_0$ for all n.

If could find any constant c for it then $\lim \frac{(\log n)^n}{n \log n}$ would have 0 which is not the case here because we just proved that it goes to infinity.

$\therefore$ it's not in Omega or $f(n) \geq c \cdot g(n)$ is not true.

Q2) Solving linear recurrences given some initial conditions

(a) $B(n) = 2B(n-1) + \frac{n}{2}$ with $B(0) = 1$

$\Rightarrow B(n) = 2(2B(n-2) + \frac{n-1}{2}) + \frac{n}{2}$

$= 2^2 B(n-2) + n + \frac{n}{2} - 1$

$= 2^2 B(n-2) + \frac{3}{2}n - 1$ $\left[\frac{3}{2}n = \frac{2^2-1}{2}n\right]$

$= 2^2(2B(n-3) + \frac{n-2}{2}) + \frac{3}{2}n - 1$

$= 2^3 B(n-3) + 2n + \frac{3}{2}n - 4 - 1$

$= 2^3 B(n-3) + \frac{7}{2}n - 5$ $\left[\frac{7}{2} = (2^3-1)\frac{n}{2}\right]$

$= 2^3(2B(n-4) + \frac{n-3}{2}) + \frac{7}{2}n - 5$

$= 2^4 B(n-4) + 2^2 n + \frac{7}{2}n - 12 - 5$

$= 2^4 B(n-4) + \frac{15}{2}n - 12 - 5$ $\left[\frac{15}{2} = n\left(\frac{2^4-1}{2}\right)\right]$

$= 2^4 B(n-4) + \left(\frac{2^4-1}{2}\right)n - 12 - 5$

$= 2^4 B(n-4) + n\left(\frac{2^4-1}{2}\right) - 17$

$= \vdots \qquad \vdots \qquad \vdots$

$= 2^n(B(n-n)) + n\left(\frac{2^n-1}{2}\right) - c$ $\quad \therefore c$ is a constant $c > 0$

$= 2^n B(0) + n\left(\frac{2^n-1}{2}\right) - c$

$\therefore B(0) = 1$

$$\boxed{B(n) = 2^n + n\left(\frac{2^n-1}{2}\right) - c}$$

(b) $L(n) = L(n-1) + L(n-2)$ with $L(0) = 2$ & $L(1) = 7$

$L(n) - L(n-1) - L(n-2) = 0$ which is a linear homogenous recurrence relation.

Let $L(n) = r^n$

$\Rightarrow r^n - r^{n-1} - r^{n-2} = 0$

$r^{n-2}(r^2 - r - 1) = 0$

$r^2 - r - 1 = 0 \Rightarrow r = \frac{1+\sqrt{5}}{2}, \frac{1-\sqrt{5}}{2}$

Then $L(n) = \alpha_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + \alpha_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$

Using the initial conditions we get the following equations:

$L(0): \alpha_1 + \alpha_2 = 2 \Rightarrow \alpha_1 = 2 - \alpha_2$ (1)

$L(1): \alpha_1\left(\frac{1+\sqrt{5}}{2}\right) + \alpha_2\left(\frac{1-\sqrt{5}}{2}\right) = 7$ (2)

* Simplifying (2):

$\alpha_1 + \alpha_1\sqrt{5} + \alpha_2 - \alpha_2\sqrt{5} = 14$ [substitute (1) now]

$2 - \alpha_2 + (2-\alpha_2)\sqrt{5} + \alpha_2 - \alpha_2\sqrt{5} = 14$

$2 + 2\sqrt{5} - \alpha_2\sqrt{5} - \alpha_2\sqrt{5} = 14$

$2 + 2\sqrt{5} - 2\alpha_2\sqrt{5} = 14$

$1 + \sqrt{5} - \alpha_2\sqrt{5} = 7$

$1 - 7 + \sqrt{5} = \alpha_2\sqrt{5}$

$\Rightarrow \alpha_2\sqrt{5} = -6 + \sqrt{5}$

$\alpha_2 = \frac{-6+\sqrt{5}}{\sqrt{5}} \Rightarrow \alpha_2 = \frac{5-6\sqrt{5}}{5}$

Then $\alpha_1 = 2 - \left(\frac{5-6\sqrt{5}}{5}\right) \Rightarrow \alpha_1 = \frac{5+6\sqrt{5}}{5}$

$$\boxed{\alpha_1 = \frac{5+6\sqrt{5}}{5}, \quad \alpha_2 = \frac{5-6\sqrt{5}}{5}}$$

$$\left[L(n) = \left(\frac{5+6\sqrt{5}}{5}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{5-6\sqrt{5}}{5}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

(c) $X(n+1) = 2X(n) - X(n-1)$ with $X(0) = 0$ & $X(1) = 1$

$\Rightarrow X(n) = 2X(n-1) - X(n-2)$

$X(n) - 2X(n-1) + X(n-2)$ ∴ Let $X(n) = z^n$

$z^n - 2z^{n-1} + z^{n-2} = 0$

$z^{n-2}(z^2 - 2z + 1) = 0$

$z^2 - 2z + 1 = 0 \Rightarrow z = 1, 1$

Since we have repeated roots, our general solution $X(n)$ is of the form: $X(n) = \alpha_1 + \alpha_2$

$X(n) = \alpha_1 (1)^n + \alpha_2 n (1)^n$

$\Rightarrow X(n) = \alpha_1 + \alpha_2 n.$

Using the initial conditions, we get the following equations:

(1) $X(0) = \alpha_1 + \alpha_2 (0) \Rightarrow X(0) = \alpha_1 = 0$

$\alpha_1 = 0$

(2) $X(1) = \alpha_1 + \alpha_2 (1) = 1$

$0 + \alpha_2 = 1 \Rightarrow \alpha_2 = 1$

$\boxed{\alpha_1 = 0, \alpha_2 = 1} \Rightarrow X(n) = n$

(d) $A(n) = 3A(n-1)$ with $A(0) = 1$, then $A(n) = \Theta(3^n)$

$A(n) = 3(3A(n-2))$

$= 3^2 A(n-2)$

$= 3^2 (3A(n-3))$

$= 3^3 A(n-3)$

⋮ ⋮

$A(n) = 3^n A(n-n)$

$= 3^n A(0)$ $[A(0) = 1]$

$A(n) = 3^n$

By the definition of Θ, $f(n)$ is $\Theta(g(n))$ if there exists some constants c_1, c_2, n_0 such that:

$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \quad \forall n \ge n_0.$

In our case, $f(n) = A(n) = 3^n$. Then we can easily find constants c_1 & c_2, & n_0 for which the above inequality holds. Let $c_1 = 1$, $c_2 = 1$, & $n_0 = 0$. Then:

$0 \le 3^n \le 3^n \le 3^n$. Hence, $A(n) = \Theta(3^n)$

$A(n)$

QED!

Question 3

a. $T(n) = 7T(n/7) + 3n + 20$

Using Master Theorem:

$a = 7$, $b = 7$, and $f(n) = 3n + 20 \Rightarrow c = 1$

$c = 1 = \log_7 7 = 1$ ∴ the recurrence falls under case 2

$T = \Theta(n^{\log_b a} \times \log n) = \Theta(n^{\log_7 7} \times \log n) = \Theta(n \lg n)$

b. $T(n) = 16T(n/4) + 100$

Using Master Theorem:

$a = 16$, $b = 4$ and $f(n) = 100 \Rightarrow c = 0$

$c = 0 < \log_b a = \log_4 16 = 2$ ∴ the recurrence falls under case 1

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_4 16}) = \Theta(n^2)$

c. $T(n) = 2T(n/2) + 5n^2 + 2n + 3$

Using Master Theorem:

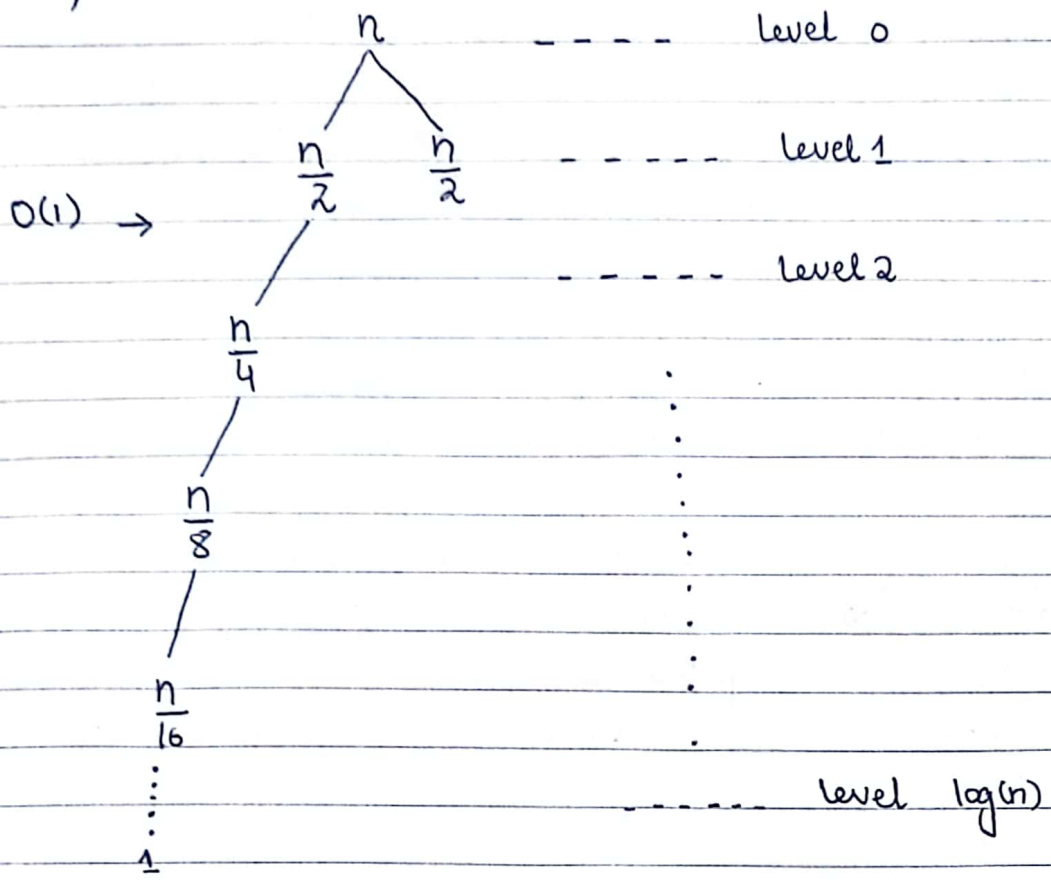
$a = 2$, $b = 2$ and $f(n) = 5n^2 + 2n + 3 \Rightarrow c = 2$

$c = 2 > \log_b a = \log_2 2 = 1$ ∴ the recurrence falls under case 3

$T(n) = \Theta(f(n)) = \Theta(5n^2 + 2n + 3) = \Theta(n^2)$

Q4: 1D Peak finding algorithm

a)



height = log(n)

complexity = O(1) x log(n) = O(logn)

b) Base Case = T(1) → O(1)

Inductive case = Assuming a positive constant c, $x < n$

so $T(k) \leq c \log k$

we have to show that $T(n) \leq c\log n$.

$T(n) = T(\frac{n}{2}) + O(1)$ we are assuming $T(\frac{n}{2}) \leq c \log(\frac{n}{2})$.

$T(n) \leq c(\log n - \log 2) + O(1)$

$T(n) \leq c\log n - c\log 2 + O(1)$

$T(n) \leq c\log n + O(1)$ so, we have shown $T(n) \leq c\log n + O(1)$

∴ Therefore, $T(n) = \log(n)$ holds

Question 5

```
FUNCTION prefix_sums (A)
    DECLARE B[n]
    B[0] = A[0]
    FOR i = 1 to n-1 DO
        B[i] = B[i-1] + A[i]
    END FOR
    RETURN B
END FUNCTION
```

Declaring B[n] takes O(n). Assigning B[0] = A[0] takes O(1). Within the loop each iteration performs an addition operation and an assignment operation. Since there are n-1 iterations, the time complexity is O(n). Returning the array requires constant time, O(1).

Then upper bound of algorithm: $O(n) + O(1) + O(n) + O(1) = O(n)$

The loop must also run for a minimum of n-1 iterations which is linear in n, i.e. lower bound of algorithm = $\Omega(n)$

So for any input size n, the algorithm will take at least c.n operations and at most d.n operations for some constants $c, d > 0$, i.e. the algorithm runs in $\Theta(n)$.

Q6) Finding Local minimum in a complete binary tree containing n nodes. A local minimum may not be unique. Runtime $\Rightarrow O(\log n)$

In a complete binary tree, all levels are completely filled except possible the lowest level which is filled as left as possible. We can design an algorithm to find a local minimum in a complete binary tree, by starting at the root node, & moving to the smaller of the two children, until we reach a node that is smaller than both its child nodes (or the node has no child nodes left). This node will obviously be smaller than its parent as well by our design, thus will be a local minimum.

* For our algorithm, we assume a node has properties of 'value', 'left', and 'right', which correspond to its values, left child & right child respectively. The algorithm is as follows:

```
Find Local Minimum (node)
    if node.left is None & node.right is None
            return node.value
    if node.left/right is None
            if node.value ≤ node.left.value
                    return node.value
            else return Find Local Minimum (node.left)
    if node.left is None
            if node.value ≤ node.right.value
                    return node.value
            else return Find Local Minimum (node.right)
    if node.value ≤ node.left.value and node.value ≤ node.right.value
            return node.value
    else if node.left.value < node.right.value
            return Find Local Minimum (node.left)
    else return Find Local Minimum (node.right)
end
```

The above algorithm compares the current node to its children, & moves to the smaller one. If our node is smaller than both children, it is a local minimum & we return it. Since a single comparison is made on each visited node, & only one node is visited per level of the tree, then our traversal path is at most the height of the tree. A fact: The height of a complete Binary Tree is $\log n$, & it has $(\log n + 1)$ levels, thus our expected traversal is $\log n$. Hence runtime is $O(\log n)$.

Q7:

i) First we will assume that the data is sorted in chronological order.
To see if the events were successful or not, we will use for loop until the length of data array and calculate the difference between previous events and current event.

for loop until n (length of array)
indented → footfall = Curr - Prev
if footfall > 0 then we'll look for another condition

Once we have got footfall for all the events we will run another for loop until the length of array and wherever the footfall is positive ~~and~~ for a consitive days i.e: subarry. Then output the subarray events with date which satisfy this condition.

→ Complexity : n for 1st for loop
n for 2nd for loop = n+n = 2n = O(n)

ii) For this case we are assuming that the data is unsorted.

First we'll sort the data using merge sort, then we'll repeat the same steps as above.

For loop until the length of array, then calculating the foot fall in the loop. After that we'll run another loop and condition it on foot fall. If the foot fall is positive for a consective subarray then we'll output the subarray events with date which satisfy this condition.

→ Complexity : nlogn for Merge Sort, 2n for footfall and subarray
nlogn + 2n = O(nlogn)

Question 8

```
FUNCTION two-max-DC (array):
    IF length(array) == 0 THEN
        RETURN (None, None)
    ELSE IF length(array) == 1 THEN
        RETURN (array[0], array[0])
    ELSE
        mid = length(array) // 2
        left = array[: mid]
        right = array[mid:]
        (left_max1, left_max2) = two_max_DC(left)
        (right_max1, right_max2) = two_max_DC(right)
        first_max = MAX(left_max1, right_max1)
        second_max = MIN(left_max1, right_max1)
        IF left_max2 > second_max and left_max2 > right_max2 THEN
            second_max = left_max2
        ELSE IF right_max2 > second_max THEN
            second_max = right_max2
        END IF
        RETURN (first_max, second_max)
    END IF
END FUNCTION
```

The recurrence relation can be expressed as: $T(n) = 2T(n/2) + O(1)$
This is because the algorithm recursively solves two subproblems of size $n/2$ each and then combines the solutions in constant time.

Using Master Theorem:

$a = 2, b = 2$, and $f(n) = O(1) \Rightarrow c = 0$

$c = 0 < \log_b a = \log_2 2 = 1$ ∴ the recurrence falls under Case 1

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$

Q9) Arcane Comparator in O(n log n)

A possible pseudocode can be as follows:

```
function arcaneComparator(amulet, target)
    if amulet == target
        return true
    else return false
end function.

function countAmulets(amulets, target)
    count = 0
    for each amulet in amulets do
        if arcaneComparator(amulet, target) is true
            count++
    return count
end function

function findMajorityAmulet(amulets)
    if length(amulets) == 1
        return amulets[0]
    mid = length(amulets)/2
    left = findMajorityAmulet(amulets[0...mid])
    right = findMajorityAmulet(amulets[mid+1 ... end])
    if left == right
        return left
    left_count = countAmulets(amulets, left)
    right_count = countAmulets(amulets, right)
    if left_count > right_count
        return left
    else return right
end function
```

The arcaneComparator() function compare two amulets & returns true if they are equal, false if they are not. This takes O(1) time. The countAmulets() function is used to count the number of amulets that are equal to the target amulet which takes O(n) time where n is the number of amulets in the array of amulets. The findMajorityAmulet() function divides the array into two halves, & recursively calls itself till the base criteria is

list, when only 1 amulet is left, which is then returned. It then checks if the left & right halves are equal in which case it is returned. If not, then the function counts the number of times the left & right amulets apear in the list of amulets & returns the amulet that appears more than $n/2$ times.

The runtime is given by the following recurrence relation:

$$T(n) = 2T(n/2) + O(n)$$

since we divide the array into 2 halves, & recursively call the function twice, so we have a branching factor of 2, & subproblems size of 2.

This relation is the same as that of merge sort, which we can solve by Case 2 of the Master Theorem to get the runtime as $T(n) = O(n \log n)$.

Hence the runtime is $O(n \log n)$.