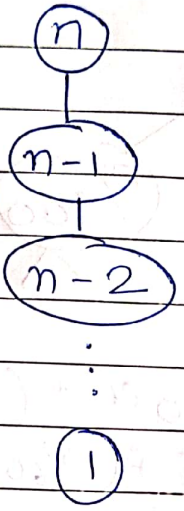
CS 660 HOMEWORK 1

(1) Question 1

a) $T(n) = T(n-1) + n^2$



we can say that by substituting n with n-1

$T(n) = T(n-2) + (n-1)^2 + n^2$

$= T(n-3) + (n-2)^2 + (n-1)^2 + n^2$

$\vdots$

$= T(0) + 1^2 + 2^2 + \dots (n-2)^2 + (n-1)^2 + n^2$ — @

$\because \sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$

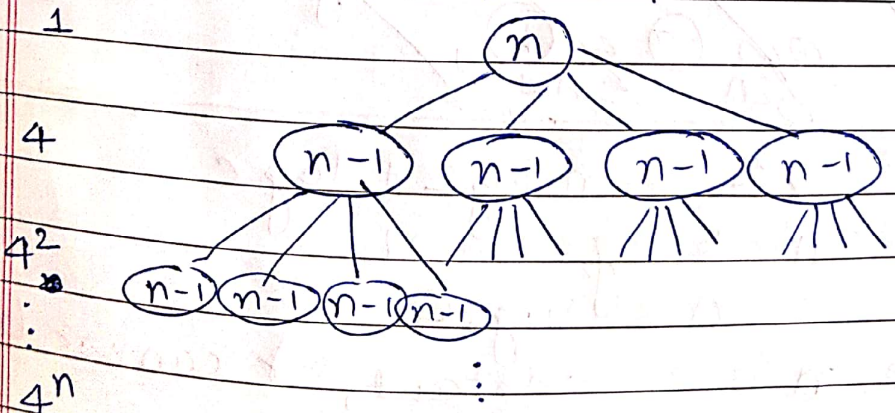
$\therefore 1^2 + 2^2 + 3^2 + \dots n^2 = n(n+1)(2n+1)/6$.

$\therefore$ from @,

$T(n) = n(n+1)(2n+1)/6 + T(0)$

$= O(n^3)$

b) $T(n) = 4T(n-1) + 99$



Now, $1 + 4 + 4^2 + \dots 4^n = 2^{n+1} - 1 = O(4^n)$

$\because 4^n \geq 1 + 4 + 4^2 + \dots 4^{n-1}$

$\therefore 1 + 4 + 4^2 + \dots 4^n \leq 2 \cdot 4^n$

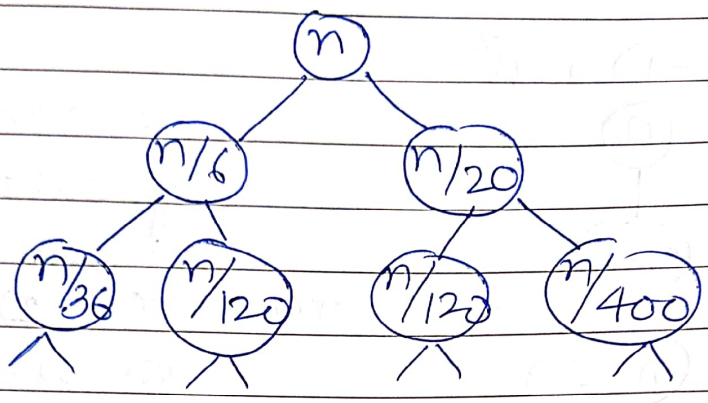
$= O(4^n)$

c) $T(n) = T(n/6) + T(n/20) + O(n)$

Largest n

n/6

$(n/6)^2 \cdot n$



$\leq c \cdot n$

$\leq c(n/6) + c(n/20) \leq \frac{26n}{120}$

$\leq 13n/60$

$\leq 169n/3600 \cdot cn \leq (13/60)^2 \cdot cn$

$\therefore cn + 13/60\, cn + (13/60)^2 cn + \cdots$

$= cn\left(1 + 13/60 + (13/60)^2 + (13/60)^3 + \cdots\right) = cn\left(\frac{1}{1-13/60}\right)$

$\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$ for $0 < \alpha < 1$ $\therefore cn(60/47)$ $O(n)$

d) $2^{2n} \neq O(2^n)$.

We can say that

$O(2^n) = 2 * 2^n$

$= 2^{n+1}$ — (a)

If $2^{2n} = O(2^n)$ — (b)

$\therefore 2^{2n} \leq c \cdot 2^n$.

$\therefore 2^n \leq c$

$\therefore \log 2 \cdot n \leq \log c$

Also from (a) & (b),

$2^{2n} = 2^{n+1} * 2$

$\therefore 2 \cdot n \cdot \log 2 \leq n \cdot \log 2 + \log c$

$2n \leq \log_2 c + n$

$\therefore n \leq \log_2 c$

But since n is a constant, it cannot be true for all $n > n_0$ and c.

e) Using Masters Theorem:

$B(n) = 2B(n/4) + n$, $\{T(n) \leq a\,T(n/b) + c \cdot n^{\alpha}\}$

$\therefore a = 2, \quad b = 4, \quad c = 1, \quad \alpha = 1.$

$2 < 4$

$\therefore T(n) = O(n^{\alpha})$

$= O(n^{1}) = O(n)$

$E(n) = 3 \cdot E(n/3) + n$

$a = 3, \quad b = 3, \quad \alpha = 1$

$3 = 3^{1}.$

$\therefore T(n) = O(n^{\alpha} \log n)$

$= O(n \log n)$

$H(n) = 4H(n/2) + n$

$a = 4, \quad b = 2, \quad \alpha = 1$

$\therefore 4 > 2^{1}$

$\therefore T(n) = O(n^{\log_b a})$

$= O(n^{\log_2 4}) = O(n^{\log 2})$

Q2. Complete Subtree : A tree where each of the nodes have 2 children except the leaf nodes where the leaf nodes are as left as possible.

The algorithm works in O(n) time.

ALGORITHM:

→ The algorithm is a recursive algorithm
→ Initialize the binary tree.
→ Pass the rootnode to the checking function.
→ Check ~~the left~~ if the tree is a full tree or a complete tree.
→ If the tree is full tree, it is a complete tree.

⇒ A tree is full if both of its children are full and both have the same maximum depth.

⇒ A tree is complete if the tree from left child is a full tree and the tree from the right child is a complete tree.

⇒ If the maximum depth of tree from left child > maximum depth of tree from right child by 1, the tree is complete if the tree from left child is complete tree and from right child is full tree.

Pseudo Code:

```
checkCompleteTree (rootNode) {

    x = checkCompleteTree (rootNode → left);
    y = checkCompleteTree (rootNode → right);
```

It is a full tree.

isfull [rootNode] = (isfull[x] &&
isfull[y] &&
maximumDepth(x) == maximumDepth(y))

if (maximumDepth (x) == maximumDepth (y))
isComplete[rootNode] = isfull (x) && isComplete(y)
if (maximumDepth (x) == (maximumDepth (y) + 1))
isComplete[rootNode] = isComplete[x] && isfull[y]

```
if (isComplete == True ) {
print (" the tree is complete ") ;
}
```

The maximum Depth & rootNode are stored in global variables and set in maximum depth function.

```
maxDepth (rootNode) {
    if (rootNode == NULL)
        return 0;
    else {
        leftdepth = maxDepth (rootNode -> left)
        rightdepth = maxDepth (rootNode -> right)
    if ( leftdepth > rightdepth)
        return (leftdepth + 1)
    else return (rightdepth + 1)
```

Q3 Let the two sorted arrays be $a[1 \ldots n]$ and $b[1 \ldots n]$

We will first need to find the median of both the arrays A and B.

ALGORITHM:

→ Let m_1 and m_2 be the median of A and B respectively.

→ If $m_1 == m_2$, $m_1 = m_2$ = the median of union of both the arrays A & B.

→ If $m_1 < m_2$,
the median is in between m_1 to the A[n] and B[1] to m_2.
since on union :
$\therefore$ AUB $[1 \ldots \underbrace{m_1 \ldots m_2}_{} \ldots n]$

→ If $m_2 > m_1$,
the median is in between m_2 to the B[n] and A[1] to m_1.
since on union :
$\therefore$ AUB $[1 \ldots \underbrace{m_2 \ldots m_1}_{} \ldots n]$

To get the median, we simply traverse the array till $n/2$ elements if it is odd, we simply take int$(n/2)$ or if it is even, we take average of middle elements
i.e. $[\text{int}[n/2] + [\text{int}[n/2] - 1]] / 2$

pseudo Code:

MedianUnion (array1, array2) {

m_1 = median(array1);
m_2 = median(array2);
if (m_1 == m_2) {
finalmedian = m_1;
} else if (m_1 < m_2) {
if (sizeof(array1) % 2 == 0)
return MedianUnion(array1[: int(sizeof(Array1)/2)+1],
array2[int(sizeof(Array1)/2-1:])

else return MedianUnion(array1[: int(sizeof(Array1)/2)+1]
, array2[int(sizeof(Array1)/2:])

else if (m_2 > m_1) {
if (sizeof(array1) % 2 == 0)

return medianUnion(array1[int(sizeof(Array1)/2 -1):],
array2[int(sizeof(Array1)/2 +1)]

else return
medianUnion(array1[int(sizeof(Array1)/2):],
array2[: int(sizeof(Array2)/2+1)].

}

e.g. [1, 4, 5] [2, 6, 7].

→ $m_1 = 4$.

$m_2 = 6$.

→ $A_1[4, 5]$ $B_1[2, 6]$.

$m_1 = 5$ $m_2 = 4$

→ A[4] B[6]. $\approx \frac{6+4}{2} \approx 5$

→ . [1, 2, 4, 5, 6, 7]

4.5 ≈ 5.

Q4. To get the majority element in O(n) time, we'll have to assume that the array is sorted (or not as it is usually sorted during median of medians)

ALGORITHM:

→ Assuming that there is a majority element present in the array, it could also be denoted as the median of the array.

→ We apply the median of median algorithm to the array i.e.

⇒ We divide all the elements into groups of 5.

e.g. [1, 4, 5] [2, 6, 7].

→ $m_1 = 4$.
$m_2 = 6$.

→ $A_1[4, 5]$ $B_1[2, 6]$.
$m_1 = 5$ $m_2 = 4$

→ $A[4]$ $B[6]$. $\approx \frac{6+4}{2} \approx 5$.

→ . [1, 2, 4, 5, 6, 7]
4.5 ≈ 5.

Q4. To get the majority element in O(n) time, we'll have to assume that the array is sorted (or not as it is usually sorted during median of ~~medians~~)

ALGORITHM:

→ Assuming that there is a majority element present in the array, it could also be denoted as the median of the array.

→ We apply the median of median algorithm to the array i.e.

⇒ We divide all the elements into groups of 5.

⇒ We sort all the groups of 5 elements in linear time as there are only 5 elements.

⇒ We then find their median ie the middle element of 5.

⇒ The medians would be $m_1, m_2 \dots m_n$ since there are n groups

⇒ We repeat the process and thus find the median of the entire array.

→ Since we assumed that the median was the majority element, we'll have to verify it by counting all the occurrences of the value of median in the array.

→ if (count(median) > (number of elements in array / 2))

the majority-element = median

else ("no majority element") print.

e.g. 50 45 40 35 30 25 20 15 10

250

1, 3, 3, 3, 4, 2, 3, 1

1, 3, 3, 3, 4 | 1, 2, 3

↓ 3 | ↓ 2

$\therefore \frac{3+2}{2} = 5/2 = \boxed{3}$ Median = Majority element

Q5. The solution to this question is similar to the solution of subset of A[1...n] that sum to k.

ALGORITHM :

→ We need 2 variables for this,
MaxUntilNow: to get the local max ending at index i.
MaxSum: to get the overall max sum of the entire subarray.

→ This can be solved in particular by getting the maxUntil now value and adding & comparing it to the current ending index.

for e.g. [a | b | c | d | e]

To check e, we check it with the max sum of subarray from a to d.

→ We repeat this for all n iterations and finally get the max sum of subarray.

Pseudo Code :

```
MaxUntilNow = A[0]
MaxSum = A[0]
    for (int i=1; i<n; i++) {
        MaxUntilNow = max(A[i], A[i] + MaxUntilNow);
        if (maxUntilNow > MaxSum)
            maxSum = maxUntilNow;
    }
    return maxSum;
```

2, -4, 3, -1, 0, 5, 2

① maxSum = 2 maxUntilNow = 2.

② maxSum = 2. ⇐ maxUntilNow = -2

③ maxSum = 3. ⇐ maxUntilNow = 3.

④ maxSum = 3. ⇐ maxUntilNow = 2.

⑤ maxSum = 3 ⇐ maxUntilNow = 2

⑥ maxSum = 7 ⇐ maxUntilNow = 7.

⑦ maxSum = 9. ⇐ maxUntilNow = 9

Q6. Pg 124 Q3(B)

The max product of subarray works similar to the above question, except here, we need to consider the negative elements in a seperate case.

ALGORITHM:

→ We have maxSum, minSum, maxUntilNow, minUntilNow all initialised to a(0).

→ ~~To get the maxUntilNow, we compare it~~ If the element at index is +ve, to get maxUntil, we compare and take max with same ~~prev~~ element or after mult it with prev maxUntil and to get minUntil, we take min with same element or after mult it with prev. minUntil.

→ If the element is -ve, to get maxUntil, we compare & take max with same element or after mult it with prev. minUntil. and to get minUntil, we take min with same element or after mult it with prev. maxUntil.

→ We do the prev step because, we can possibly multiply 2 negative numbers to gain a larger positive number.

→ This can be followed by taking the larger between result and maxSum as the result.

Pseudo code:

```
minSum = a[0], maxSum = a[0]
minUntil = a[0], maxUntil = a[0], result = a[0]
for (int i=1; i < n; i++) {
    if (a[i] > 0) {
    maxUntil = max(a[i], a[i] * maxUntil)
    minUntil = min(a[i], a[i] * minUntil)
    } else {
    maxUntil = max(a[i], a[i] * minUntil)
    minUntil = min(a[i], a[i] * maxUntil)
    }
    maxSum = maxUntil;
    minSum = minUntil;
result = max(result, maxSum);
}
```

{-2, 1, 0, -3, 7, -2, -8}

① mis = -2 mas = -2 miu = -2 mau = -2.

② mis = -2. mas = 1 ⩽ miu = -2 mau = 1.

③ mis = 0 mas = 0 ⩽ miu = 0 mau = 0

④ mis = -3 mas = 0 ⩽ miu = -3. mau = 0.

⑤ mis = -21 mas = 7. ⩽ miu = -21 mau = 7

⑥ mis = -14 mas = 42 ⩽ miu = -14 mau = 42

⑦ mis = -336 mas = 112 ⩽ miu = -336 mau = 112.

result = 112.

Pg 55 : 2b(b).

This is an extension to the Q3 of the assignment

ALGORITHM :

→ After getting the medians of both the arrays a and B namely m_1 and n_1

We compare k with m_1 and n_1

→ Using this info, we can eliminate 1 half of an array.

→ If $m_1 + n_1 > k$,

we can eliminate second half of m_1 if $m_1 > n_1$

or else, eliminate second half of n_1.

Similarly if $m_1 + n_1 < k$,

we can eliminate first half of m_1 if $m_1 < n_1$

or else eliminate first half of n_1.

→ we then adjust k after changing the array size.

```
kthelement ( arr a , arr b , k ) :
    if ( size of (a) == 0 )
        return ( b[k] );
    else if ( size of (b) == 0 )
        return (a[k]);
    else {
    m1 = a[size of (a)/2] , n1 = b[size of (b)/2]
    if ((m1 + n1) > k) {
        if (m1 > n1)
        return ( kthelement (a[: m1], b, k)
    else
        return (kth element( a, b[: n1], k)
    } else {
    if ( m1 > n1)
        return(kthelement (a , b[n1+1 :], k-n1-1)
else return (kthelement (a[m1+1 :], b, k - m1-1 )
```