cmi | CHENNAI MATHEMATICAL INSTITUTE

https://www.cmi.ac.in

End Semester Examination, Aug–Nov 2023

Name: Aritra Majumder	Roll Number: MCS202304
Subject: Algorithms	Date: 4th Jan
Course & Year: (MSC 2023)	Total No. of Pages:

—— Begin here ——

(All the times, I'm writing length function (method) as size function (method) ✓)

125/125

(a)

```
~~LCMMREC(D):~~
    ~~if D.Size ≤ 2.~~
        ~~return 0~~
    ~~n = D.Size - 1~~
    ~~current Best = ∞~~
    ~~for i = 1 →~~
        ~~currentBest = min(currentBest, $D_0 D D_n$ + LCMMREC(D(0:k))~~
                                      ~~+ LCMMREC(D(k+1:n))~~

    ~~return currant Best~~
```

```
~~LCMMREC(D):~~
    ~~if length(D) ≤ 2:~~
            ~~return 0.~~
    ~~n = length(D) - 1~~
    ~~current Best = ∞~~
    ~~for i = 1 → n-1~~
        ~~currentBest = min(currentBest,~~
```

e)

```
LCMM(D):
    n = length(D) - 1
    M[1:n, 1:n] = ∞
    for i=1 → n.
        M[i,i]=0
    for l=2 to n:
        for i=1 to n-l+1:
            j = i+l
            for k = 1 to (j-1)
                M[i,j] = min(M[i,j], M[i,k] + M[k+1
                         + $D_{i-1} D_k D_j$ )
```

initialize 2D Matrix M that has two axes with Indices from 1 to n to ∞

LCMMREC(D):

n

(a)

```
LCMMREC(D):
    n = length(D) - 1

    return MODLCMMREC(D, 1, n)


MODLCMMREC(D, l, r):

    if l == r:
        return 0
    current Best = ∞
    for K = l to (r-1)
        currentBest = min(currentBest, MODLCMMREC(D, l, K)
                                       + MODLCMMREC(D, K+1, r)
                                       + D[l-1]·D[K]·D[r].

    return currentBest.
```

10/10

✓

(b) correctnes proof:

we will prove using induction on number of matrices.

Base case: (l==n) or there is only one matrix

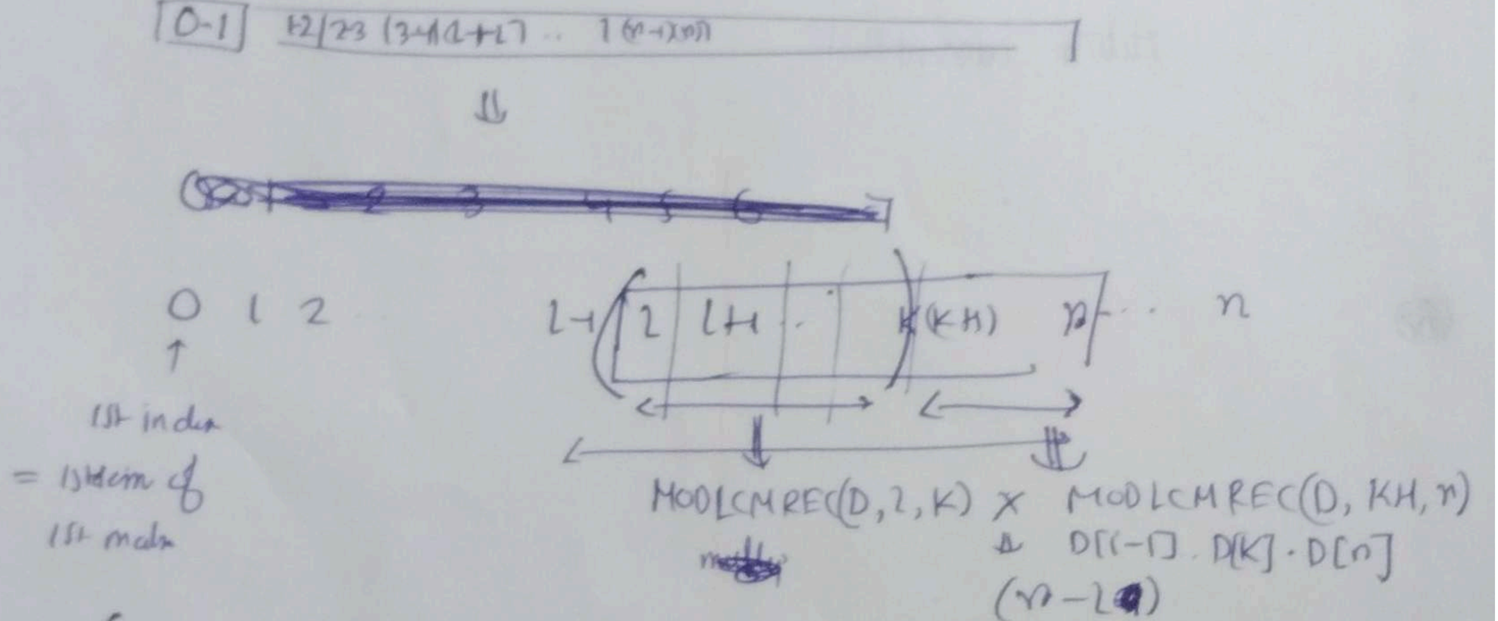
any cost

we dont need to multiply one matrix.

So, return 0

~~Induction hypothesis if the multipl~~

Induction step: lets say we have to calculate the LCMMREC(D) for (l → n)th matrix of length p(say) or MODLCMREC(D, l, n)


[0-1] 1-2 2-3 (3-4)(l+1) .. (n-1)(n)
0 1 2
1st index = 1st item of 1st matrix
l-1 l l+1 . k (k+1) n . . n
MODLCMREC(D, l, K) × MODLCMREC(D, K+1, n)
+ D[l-1] . D[K] . D[n]
(n-l)

So. now we can split the matrices into positions and if we can calculate the mun of all such division ie, MODLCMREC(D, l, k) and MODLCMREC(D, k+1, n) then we can multiply such two matrices in D[l-1]·D[k] D[n]

By the ind hyp the smaller instances give correct ans ✓

So, our algorithm is correct

$\frac{10}{10}$

(c) ~~T(n)~~

$T(1) = 1$

$T(n) = 1 + \sum_{i=1}^{n-1} (T(i) + T(n-i) + c)$ ✓

2nd (n-i) matrix multiplication

constant multiplication

first i matrix multiplication

$\frac{10}{10}$

What are you counting?.

d) $T(n) = 1 + \sum_{i=1}^{n-1} (T(i) + T(n-i) + c)$

$\geq T(1) + T(2) \quad + T(n-1) + T(n-1) + \cdots \quad T(1)$

where did the c go?.

$\geq 2T(n-1)$

$\therefore T(n) \geq 2T(n-1) + 2T(n-2) + \quad T(1)$

$= 2(T(n-1) + T(n-2) + \cdots \quad T(1))$

Assuming, $T(n) = \alpha^n$

~~$T(n) \geq 2(\alpha^{n-1} + \quad \alpha^1)$~~

~~$\geq 2\alpha(\alpha^0 + \cdots \quad \alpha^{n-1})$~~

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

$\alpha^{0})$

$T(n) = 2(\alpha^{n-1} + \alpha^{n-2} +$

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

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

$\geq 2\alpha^{n-1}$

~~$\alpha^{n} \geq 2\alpha^{n-1}$~~

also $T(n) \geq 2T(n-1)$

$\frac{T(n)}{T(n-1)} \geq 2$

$\frac{T(n-1)}{T(n-2)} \geq 2$

$\therefore T(n) \geq 2^{n}$

$T(n) = \Omega(2^{n})$

The question asked for a $\Theta()$ bound

0/10

PT

(M is a 2D Matrix with indices 1 to n in both cases initialised to ∞)

22.

```
e)  LCMM(D):
        M = [1:n, 1:n] initialised to ∞.        // n^2
        n = length(D) - 1                       // 1

        for i = 1 → n
            M[i,i] = 0                          // n

        for l = 2 → n                           // n
            for i = 1 to n-l+1                  // n^2
                j = i+l-1                       // n^2
                for k = i → j-1                 // n^3
                    M[i,j] = min(M[i,j], M[i,k] + M[k+1,j]
                                 + D[i-1]·D[k]·D[j]     // n^3

        return M[1,n].   // 1
```

(for g.1.g)

10/10

(1 → n) indices assumption

(f) ~~we will get~~ it would take 0 cost to multiply a single matrix

So, $M[i,i]=0$ for all matrices

for ~~for~~ all the matrices from (i to j)

$M[i,j]$, we can split the multiplication

into $\underset{(i-1)}{} \underline{i}) \underline{(i+1)} \ldots \underline{(j-1)}) j$ these parts

but then we need to make sure that smaller instances like $M[i,K]$ or $M[K',j]$ are solved before when $K<j$, $K'>i$.

~~or the~~, So, we first iterate over length of matrix array, then finding the optimal splitting to update $M[i,j]$

10/10

first loop: iterates over lengths so that smaller ~~matrices~~ length problem are calculated first.

L calculate $M[i, i+L-1]$ for all L

2nd loop: for length (L)

✓

try to calculate $M[i, i+L-1]$ with i varying from 1 to $n-L+1=j$

L calculate all $M[i, i+L-1]$ for all i

3rd loop: tries to find the optimal splitting K from ith matrix to $j-1$ th matrix such that

$$M[i,j] = \min_{K=i}^{j-1} \left(M[i,K] + M[K+1,j] + D[i-1]D[K]D\right.$$

↑ calculate optimally

(g) from the question (e), we get that

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

$= n^3 + 2n^2 + 2n + 1$

$\geq n^3$

$T(n) = \Omega(n^3)$

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

now $T(n) \leq 2n^3$ for all $n \geq 1000$

as ~~$T(n) = 2000n^3$~~

$T(n) =$ ~~$2000n^2 +$~~ $1000n^2 + 2n^2 + 2n + 1$

$< 2000n^2$

for $n \geq 100$

$\therefore T(n) \leq 2n^3$ for all $n \geq 1000$

$\therefore T(n) = O(n^3)$

✓

$\frac{10}{10}$

$\therefore T(n) = \Theta(n^3)$

$\therefore$ Algorithm runs in polynomial (n) time

2. (a)

```
HasRepeats (A):
    $\hat{A}$ = MergeSort (A)
    n = length ($\hat{A}$)

    for i = 1 to n-1:
            if $\hat{A}[i] == \hat{A}[i-1]$
                    return true
    return false
```

✓

$\frac{10}{10}$

(b) in comparison based ~~(sorting algorithm)~~ model, we can only ask this type of question

a > b
a == b   and combination them
a < b ⟶ answer will be (0,1)

So, the adversary would follow this simple strategy

the user wants to sort an A array that the adversary has.

He can only ask this type of question

(i:j)
⇒ if A[i] < A[j] •

for this question, let adversary has already declared all the elements are unique, to handle worst case as in that case A[i] ≤ A[j] question would not be of any help

Now Adversary follows the following strategy

He keeps all possible permulations of the array A as all the elements are unique, (let's say there are n elements)

So (n! total permutation) $L_0 = n!$ number of permulation

now, whenever the person asks some (i:j) question he calculates L_{yes} as all the permulations st

$A[i] > A[j]$

L_{No} as " " " "

$A[i] < A[j]$

∴ Now

$L_{yes} \cup L_{No} = L_0$

$L_{yes} \cap L_{No} = \phi$

$\max(|L_{yes}|, |L_{No}|) \geq \frac{|L_0|}{2}$

∴ if $|L_{yes}| \geq \frac{|L_0|}{2}$

then $L_1 = L_{yes}$

else $L_1 = L_{No}$

Now, iteratively adversary processes $L_2, L_3, L_4 \ldots L_K$

until L_K has only one element 17 (then the user has found the answer

Why should the algorithm wait till this happens?

$|L_i| \geq |L_{i-1}|/2$

$|L_1| \geq \frac{|L_0|}{2}$, $|L_2| \geq \frac{|L_1|}{2} \geq \frac{|L_0|}{4}$ ⋮ $|L_K| \geq \frac{|L_0|}{2^K}$

So. now, $|L_k| \geq \frac{|L_0|}{2^k}$

or $2^k \geq \frac{|L_0|}{|L_k|}$

or $2^k \geq n!$

or $k \geq \log(n!)$

What prevents the algorithm from stopping when the adversary has more than one permutation ?.

$\frac{10}{20}$

now $n! = n\cdot(n-1)\cdots(\frac{n}{2})\cdots 1 \geq \underbrace{\frac{n}{2}\cdot\frac{n}{2}\cdot\frac{n}{2}\cdots}_{n/2}$

$\therefore n^n \geq n! \geq (\frac{n}{2})^{n/2}$

$\therefore n\log n \geq \log n! \geq \frac{n}{2}(\log n - 1)$

$\therefore$

$k \geq \frac{n}{2}\log n - \frac{n}{2}$

$k = \Omega(n\log n)$.

[0]*K = initializes K sized array with all elements (0)

1

```
4)(a) DynArr:   // data structure
      n=0       // initialization of number of elements
      A=[]      // Initialization of an empty array

   Insert (x):

      if n==0:
          B=[0]    // initialize an array with 1 element as 0
          A=B      // ~~copy elements of B to A~~ change reference
          A[n]=x   // (n=n+1)
          n++.
   else if n == (A.size)   // A is ~~all the elements are~~ full
          B=[0]*(3n)   // creating array of triple size

          for i=1 to n
              B[i]=A[i]   // copy element

          A=B.        // (changing the reference, assume
          A[n]=x      //  AO time is taken here.)
          n++

   else   A[n]=x
          n++

Retrieve (i):
   if i >= n:
     error
     array index out of bound

   else return A[i]
```

✓ 10/10

(4)(b) Insert() has worst case cost $O(n)$ as when array (trip6 the size C) is full, Insert() will create an array of ~~add dene~~ and copy elements. So it will take $O(n)$ time

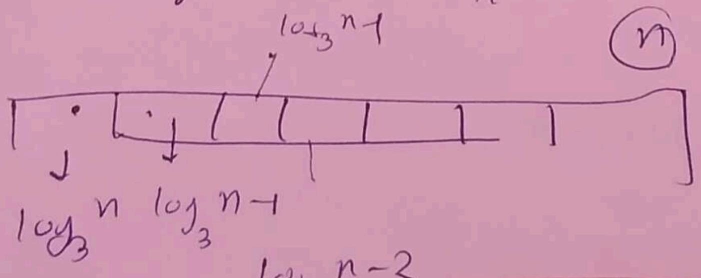
$\frac{10}{10}$ ✓

from the crude analysis above, it seem like calling insert() n times will take $n * O(n)$ time $= O(n^2)$

But, after noticing that, insert() will take $O(n)$ time very rarely (whenever the size is some power of 3).

In that case, we can make our analysis better to get a better amortized cost.

after n operations, there are very few indices those are initialized many times and elements copied many times



Not clear.

$$So, \ \Sigma^i C = 1(\log_3 n) + 2(\log_3 n - 1) + 8(\log_3 n - 3) - + 0$$

$$n + \frac{2n}{3}(0)$$

$$= O(n)$$

in AGP series

why milar to heapify() takes $O(n)$ time instead of $O(n \log n)$ time)

(c) To insert n elemn we need to call Insert On times.

we will use aggregate analysis to show that it has constant amortized cost

So $\sum_{i=1}^{n} c_i = n + (3^0 + 3^1 + \cdots 3^{\log_3 n}) \cdot 4$

Explanation: to insert n elements in the array we need to insert n times.

when the array size is power of 3, we need 3 times more operation of the size to create a new array and one time more operation to copy all of them

So 4 times more operation wherever the array size increase

$$\therefore \sum_{i=1}^{n} c_i = n + (3^0 + \cdots 3^{\log_3 n}) \cdot 4$$

$$= n + 4 \cdot \frac{3^{\log_3 n + 1} - 1}{3 - 1}$$

$$= n + 6(n - 1)$$

$$< 7n$$

$\frac{10}{10}$

$\therefore$ ~~$T(n) = \frac{7n}{n}$~~

$$\therefore T(n) < \frac{7n}{n} = 7$$

$\therefore$ ~~$T(n) = 0$~~

it has amortized cost of O(1)

4(a) we will now use accounting method: to analyze

on every insertion, element x is inserted with 7 credits b)

and the total credit is maximum when the array is full

and the ~~x~~ (each element has 0 credit right after the expansion

and before the insertion of the new element).

Claim: only 2/3 rd last elements of the full array. each has 6 credits on it

proof: right after the insertion to the full array, we assume that each element has 0 credits and now last 2/3 rd elements will have 6 credits on it as they will be inserted with 1 cost, remaining will be 6.

Now, when expansion happens, array was full and lets say the array size was $(n) = 3K$ for some K.

now $2K$ elements has 6 credits each $= 12K$ credits

now, we need to create an array of $3n$ size which will take $9K$ credits and there are $3K$ elements which need to be shifted to new array.

It will take another $3K$ credit.

So, $12K$ credit will be consumed and surplus will be 0.

So, Insert(x) n time has constant ~~time~~ amortized cost

10/10

3)(b) $G = ((V,F), c, s, t)$.

$f : E \to N$

10/10

Now let us construct G_f (residual graph) in the following way,

an edg $e = (U \to v)$ has capacity c_e and flow f_e through that edge

the in G_f we will add $e'_f = (V \to U)$ ✓

with $c'_{e'_f} = f_e$

and $e_f = (U \to v)$ with $c'_{e_f} = c_e - f_e$ if $c'_{e_f} \neq 0$. ✓

$$c'(v \to U) = \begin{cases} f(U \to v) & \text{if } f(U \to v) \neq 0 \\ 0 & \text{otherwise} \end{cases}$$ ✓

$$c'(U \to v) = \begin{cases} 0 & \text{if } f(U \to v) = c(U, v) \\ c(U \to v) - f(U \to v) & \text{otherwise} \end{cases}$$

upon constructing G_f, we need to check if t is reachable from s or not. ✓ [No 0 edges are present in G_f, assume they are removed]

t reachable from s in G_f $\Rightarrow$ f is not max flow ✓

t not " " s in G_f $\Rightarrow$ f is max flow

```
MaxFlowPath ( G = (V, E), c, s, t)
    for v ∈ V
        v.flow = 0
        v.parent = NIL
    s.flow = ∞ , S = ∅
    Q = an empty max-priority Queue
    for v ∈ V
        enqueue v in Q, keyed by v.flow
    while Q is not Empty
        v = Extract - Max (Q)
        S = S ∪ {v}
        for each edge v → x
            newflow = min (v.flow, c (v → x))
            if newflow > x.flow
                x.flow = newflow
                x.parent = v
                IncreaseKey (Q, x, x.flow)
    ( we can get the flow
                by calling t.flow)
    reqPath = [t]
    cur = t
    while (cur != s)
        cur = cur.parent
        reqPath.~~pushback~~.append (cur)
```

Missing: description of the data structure.

18/20

~~20/20~~

PTO

```
revPath = reverse (revPath, 0, length (revPath) -1)

Return revPath
```

~~reverse (arr)~~

~~if length~~

~~if length (arr) <= 1~~

~~return arr~~

~~n = length (arr)~~

~~temp = arr [n-1]~~

~~arr [0]~~

~~arr [n-1] = arr [0]~~

~~arr [0] = temp~~

```
reverse (arr, l, r)

    if l == r || l >= r

        return arr.

    temp = arr [l]
    arr [l] = arr [r]
    arr [r] = temp

    return reverse (arr, l+1, r-1)
```