⇨ Tackling the first problem:-

Initially, I tried the brute-force algorithm.

eg. $\begin{bmatrix} 1 & 2 & 3 \\ 9 & 2 & 8 \\ 1 & 5 & 3 \end{bmatrix}$

I created a program where I started from (0,0) then scanned (0,1) & (1,0) and navigated to the cell which as least cost.

$$A[i][j] \longrightarrow \min(A[i+1][j], A[j+1][i])$$

Within few minutes I got to know the approach was wrong.

Secondly ~~And finally~~ I tried a method which worked. We go to cell and update it with cost to reach it.

eg in $\begin{bmatrix} 1 & 2 & 3 \\ 9 & 2 & 8 \\ 1 & 5 & 3 \end{bmatrix}_{A_{ij}} \longleftrightarrow \begin{bmatrix} 1 & 3 & 6 \\ 10 & 2 & 8 \\ 11 & 5 & 3 \end{bmatrix}$

[The only way to reach the topmost row & column is to travel horizontally/vertically].

Now we can reach A[1][1] in two ways:

(a) (0,0) → (1,0) → (1,1)

(b) (0,0) → (0,1) → (1,1)

As $A_{(0,1)}$ & $A_{(1,0)}$ are already updated with their costs, we simply add the min of (A[0][1] & A[1][0]) to A[1][1] and get the min. cost of cell A[1][1].

$$\therefore \begin{bmatrix} 1 & 3 & 4 \\ 10 & 2 & 8 \\ 11 & 5 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 3 & 4 \\ 10 & 1 & 12 \\ 11 & 10 & (13) \end{bmatrix}$$

→ A[2][2] gives the min cost path.

$$\begin{bmatrix} 1 & 2 & 3 \\ 9 & 2 & 8 \\ 1 & 5 & 3 \end{bmatrix}$$

1+2+2+5+3 = 13

↓ Original Matrix

I found a problem here that though the time reduces to milliseconds, we lose the initial matrix. Luckily in this task, we need not require the original matrices costMartrix A, B later.

Same logic for B (costMatrixB).

So after coding the logic above the costMatrixA changes to costMatrixA where each of its elements costMatrixA[i][j] stores the minimum cost path from cMA[0][0] to CMA[i][j]. Similarly for CMB.

But this only solves the first part of problem.

FINAL SOLUTION: PART-1:

I realised that in the part (ii), productMat was product of matrices.

$$\begin{bmatrix} - & - & - \\ - & - & - \end{bmatrix}_A \times \begin{bmatrix} - & - \\ - & - \\ - & - \end{bmatrix}_B$$

⇨ Tackling the first problem:-

Initially, I tried the brute-force algorithm.

eg. $\begin{bmatrix} 1 & 2 & 3 \\ 9 & 2 & 8 \\ 1 & 5 & 3 \end{bmatrix}$

I created a program where I started from (0,0) then scanned (0,1) & (1,0) and navigated to the cell which as least cost.

$$A[i][j] \longrightarrow \min(A[i+1][j], A[j+1][i])$$

Within few minutes I got to know the approach was wrong.

Secondly ~~And finally~~ I tried a method which worked.

We go to cell and update it with cost to reach it.

eg in $\begin{bmatrix} 1 & 2 & 3 \\ 9 & 2 & 8 \\ 1 & 5 & 3 \end{bmatrix}_{A_{ij}} \longleftrightarrow \begin{bmatrix} 1 & 3 & 6 \\ 10 & 2 & 8 \\ 11 & 5 & 3 \end{bmatrix}$ [The only way to reach the topmost row & column is to travel horizontally/ vertically].

Now we can reach A[1][1] in two ways:

(a) (0,0) → (1,0) → (1,1)

(b) (0,0) → (0,1) → (1,1)

As $A_{(0,1)}$ & $A_{(1,0)}$ are already updated with their costs, we simply add the min of (A[0][1] & A[1][0]) to A[1][1] and get the min. cost of cell A[1][1].

$$\therefore \begin{bmatrix} 1 & 3 & 4 \\ 10 & 2 & 8 \\ 11 & 5 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 3 & 4 \\ 10 & 1 & 12 \\ 11 & 10 & (13) \end{bmatrix} \rightarrow A[2][2] \text{ gives the min cost path.}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 9 & 2 & 8 \\ 1 & 5 & 3 \end{bmatrix} \quad 1+2+2+5+3 = 13$$

↓ Original Matrix

I found a problem here that though the time reduces to milliseconds, we lose the initial matrix. Luckily in this task, we need not require the original matrices costMartrix A, B later.

Same logic for B (costMatrixB).

So after coding the logic above the costMatrixA changes to costMatrixA where each of its elements costMatrixA[i][j] stores the minimum cost path from cMA[0][0] to CMA[i][j]. Similarly for CMB.

But this only solves the first part of problem.

FINAL SOLUTION: PART-1:

I realised that in the part (ii), productMat was product of matrices.

$$\begin{bmatrix} - & - & - \\ - & - & - \end{bmatrix}_A \times \begin{bmatrix} - & - \\ - & - \\ - & - \end{bmatrix}_B$$

where A[i][i] stores the minimum cost in going from [i][i] to [sizen-1][sizen-1]

Here in my second sol^n I had to make changes I had to reverse the steps.

eg. $\begin{bmatrix} 1 & 2 & 3 \\ 9 & 2 & 8 \\ 1 & 5 & 3 \end{bmatrix} \longleftrightarrow \begin{bmatrix} 1 & 2 & 14 \\ 9 & 2 & 11 \\ 9 & 8 & 3 \end{bmatrix}$

Just the reverse path.

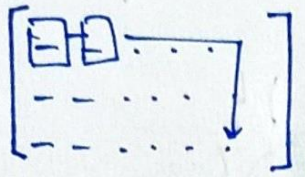
$\begin{bmatrix} 13 & 12 & 14 \\ 11 & 10 & 11 \\ 9 & 8 & 3 \end{bmatrix} \Rightarrow$ X matrix

Each cell in X matrix X[i][i] stores the minimum cost path from [i][i] to [sizen-1][sizen-1].

Property exploited: Path cost from pt x to pt y is same as the path cost from y to x.

So now in the ProductMat, instead of calling f^n FindMinCostA(i,k,sizen) repeatedly, we can direct use the value of X[i][k]. (which has already been calculated in the FindMinCostA function.

PROBLEM-1 SOLVED (Part-A).



Minimum path required.

sizen x sizen

eg. $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ We have to traverse from (0,0) to (1,1) and add the numbers that come along the path.

in 2x2 matrix two paths are possible

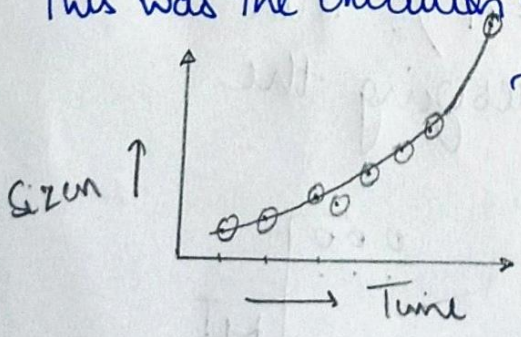
(0,0) + (0,1) + (1,1) = 7 ✓ ∴ 7 is answer.

(0,0) + (1,0) + (1,1) = 8

Initially the code given in question was of exponential time as it involved recursion.

Time	0.287	0.321	0.4	0.39	0.702	1.198	3.8	54
Value of sizen	1	5	10	11	13	15	16	18

This was the execution time vs value of sizen.



Plotting on desmos gives an exponential looking graph.

So the first problem is to find a way to find minimum cost without using recursion.

The second problem was to optimize the product of matrices.

The regular code is:

```
for(  ){
    for(  ){
        for(  ){
```

This has complexity $O(n^3)$.

```
}}}
```

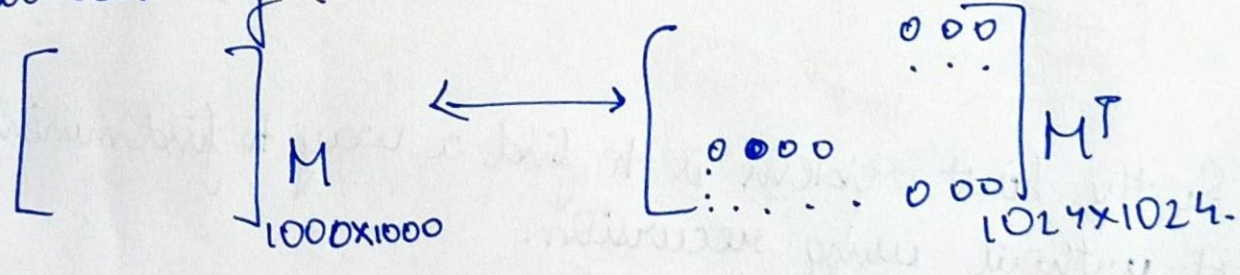
This part takes 5-6 seconds for n=1000.

So the problem boils down to how to multiply matrices efficiently.

Try 1: Using Strassen algorithm.

(Tried to read the Strassen algo).

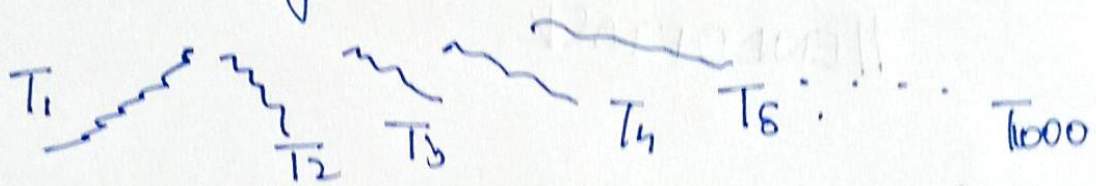
Drawback: The Strassen O is $O(n^{2.80})$. This would bring down the time to 2 seconds but my target was 1 sec. Also Strassen algorithm works only for power of 4 (ie sysm $= 4^n$). This problem can be solved by resizing the matrices eg.



1000x1000 matrix can be transformed into 1024x1024 matrix by adding zeros to extra rows & columns. No the M^T can be multiplied but I dropped the idea as time would reduce to 2 seconds only.

It didn't take time to realise that the only way was multithreading.

Multithreading means using different threads to perform an operation. The initial idea was clear. Each row-column multiplication of two matrices was independent; so why to execute it using same thread, use diff threads to multiply.



I can create 1000 threads for thousand colum-row operations (like a dot product). The idea seemed easy but took a lot of time to implement it.

Try 2: I tried it using python as threading is easy in it. I learnt about threads.
t1.join(); Thread(target='functionname'), args) etc. what are daemon threads? etc. But could not implement it. I was getting errors which I could not understand.

Try3: Tried learning multithreading in C++.

→ used chrono library
(for accurate time measurements).

→ used multithreading in CPP

We need to check on the optimize -O3 flag by compiler (as shown in image) in the main document.

//ENDOFTASK.