Homework #1

(Due: Oct 25)

Group Number: 19

Group Members		
Name	SBU ID	% Contribution
Chaitanya N. Kalantri	111446728	33.33
Neel Paratkar	111483570	33.33
Rohan Karhadkar	111406429	33.33

Collaborating Groups

Group Number	Speci cs (e.g., speci c group member? speci c task/subtask?)
52	Rahul Bhansali, Kiran Kasarapu Discussed about Question 1 and 2.

External Resources Used

	Speci cs (e.g., cite papers, webpages, etc. and mention why each was used)
1.	https://cseweb.ucsd.edu/~slovett/teaching/SP15-CSE190/poly-mult-and-FFT.pdf
2:	https://leetcode.com/problems/unique-paths-ii/description/

Task 1. Most Profitable Trade Route:

(a) Suppose ML denotes the sub grid map consisting of the leftmost n−1 columns of M and MT denotes the one consisting of the topmost m − 1 rows of M. Show that if you know the solutions to both ML and MT you can extend the solution to M in Θ (1) time.

Here, we know:

$M_L = M_{m-1,\,n}$ (leftmost n-1 columns)

$M_T = M_{m,n-1}$ (topmost m-1 rows)

$M_{m-1,\,n}$ is the solution of the grid starting at (1,1) and ending at (m,n-1)

And $M_{m,n-1}$ is the solution of the grid starting at (1,1) and ending at (m-1,n)

Also, we know the solutions till the left n-1 columns and m-1 columns.

Therefore, if we can compute the value of

$M_{m,n} = Max_{Profit}\,(M_{m-1,n}\,,\,M_{m,n-1})$

Some of the additional conditions will be:

1. If there is a shop at that location, then:
   If(state(shop))
   {
   // buyValue = Previous buyValue stored;

   If(buyValue – shopValue > $M_{m,n}$)
   $M_{m,n}$ = buyValue;

   If(buyValue >shopValue)
   buyValue = shopValue;
   }
2. Else: // The location is empty or inaccessible
   //Do nothing

Psuedo Code:

```
for(int i = m-1;i<size; i++)
        for(int j = n-1;j<size;j++)
                if (state(Shop) )
                        if(buyValue – shopValue >  Max_Profit (M_i, j-1 , M_i-1 , j)) // Θ(1)
                                Max_Profit = buyValue – shopValue
                        else
                                if(buyValue > shopValue ) // Θ(1)
                                buyValue = shopValue
                else
                        Max_Profit (M_i, j-1 , M_i-1 , j) // Θ(1)
```

In each of the above three cases, we always know all the values required to calculate the solution at M (by using the (buyValue, profit) of M_L and M_T). Thus, finding the solution to M is a constant time operation and can be done in Θ (1) time.

Only traverse through rows and columns. The computation cost for finding the Max_{profit} at each cell/location is only Θ (1) time.

Hence, proved.

(b) Explain how you will use the observation from part (a) to solve problem M in Θ (mn) time and Θ (mn) extra space.

Here, we are given extra Ø(mn) space.

Hence, initially we will calculate/initialize the values and states of the elements of the first row and first column. Later, we can apply the same method as defined in the (a) part to get all the values in the matrix.

Therefore, at each iteration we will have the leftmost column and top most row data, which we consume the extra given space.

Psuedo Code / Code:

Here,

We will consider

```
arr[][] = (buyValue, maxProfit)  //At each cell we will store  (buyValue, maxProfit).

M = the complete matrix/grid

buyValue = Previously stored buyValue

maxProfit = Max profit encountered till now


//Calculate the maximum profit

maxProfit(M, n, n, arr):

//Initialize the topMost row and leftmost column

//topMostRow

M[1][1] = (buyValue, maxProfit)

for i in range(2,n+1):

//Shop
if M[1][i]>0:


// shopValue is greater than the buyValue.
if M[1][i]< buyValue:
            buyValue = M[1][i]
// Update the buyValue


else:
if(M[1][i]-buyValue>maxProfit):
                maxProfit = M[1][i]-buyValue

// Update the arr
arr[1][i] = (buyValue, maxProfit)

//Empty Location
elif M[1][i]==0:
// Update the arr and make it equal to the previous arr value.
```

```
arr[1][i] = arr[1][i-1]


//Inaccessible Location
elif M[1][i]<0:
//Set all the cells to the right of Inaccessible Location as Inaccessible (kind off)

(buyValue,profit) = (INT_MAX, INT_MIN)

break



inf = 9999

for j in range(i,n+1):
        arr[1][j]=(+inf,-inf)
break


// leftMostColumn

//Repeat the same steps with M[i][1]

for i in range(2,n+1):

//Shop
if M[i][1]>0:


// shopValue is greater than the buyValue.
if M[1][1]< buyValue:
        buyValue = M[i][1]
// Update the buyValue

else:
if(M[i][1]-buyValue>maxProfit):
          maxProfit = M[i][1]-buyValue

// Update the arr
arr[i][1] = (buyValue, maxProfit)

//Empty Location
elif M[i][1]==0:
// Update the arr and make it equal to the previous arr value.
arr[i][1] = arr[i-1][1]


//Inaccessible Location
elif M[i][1]<0:
```

```
//Set all the cells to the right of Inaccessible Location as Inaccessible (kind off) by
(buyValue,profit) = (INT_MAX, INT_MIN) and break

inf = 9999
for j in range(i,n+1):
        arr[j][1]=(+inf,-inf)
break


//Traverse the complete grid

for i in range(2,m+1):
for j in range(2,n+1):
#extract the (buyValue,maxProfit) of left and top cells
leftBuyValue, leftProfit = arr[i][j - 1]
 topBuyValue, topProfit = arr[i - 1][j]

//calculate maxProfit at each cell by checking left and top maxProfit and the shopValue (M[i][j]
of that cell)

//shop
//buyValue is min of left and right buyValue
//maxProfit = max(left and top maxProfit & the left and right buyValue)
if M[i][j]>0:
        buyValue = min(leftBuyValue, rightBuyValue,M[i][j])
        profit = max(leftBuyValue, topBuyValue, M [i][j]- leftBuyValue, M [i][j]-
topBuyValue)
        dp[i][j]=(buyValue,profit)

//Empty location
elif M[i][j]==0:
        arr[i][j] = (min(leftBuyValue, topBuyValue),max(leftProfit, topProfit))

//Inaccessible location
elif M[i][j]<0:
        arr[i][j] = (+inf,-inf)


#In order to get the path of the traversal; we will backtrack
i=m ,  j=n ,  path=""

while i>=1 and j>=1:
// Check where the cells are inaccessible and update the path; and consider the next path
//Top

// E = East, S= South
if M[i-1][j]<0:
```

```
        path+="E"
j=j-1
continue
//Left
if M[i][j-1]<0:
        path += "S"
i=i-1
continue

leftBuyValue, leftProfit = arr[i][j - 1]
      topBuyValue, topProfit = arr[i - 1][j]


if M[i][j]==0:
        buyVal,maxProfit = arr[i][j]
if maxProfit== leftProfit:
            path += "E"
j=j-1
continue
        if profit== topProfit:
            path +="S"
i=i-1
continue


if M[i][j]>0:
        buyVal,profit=arr[i][j]
if profit == leftProfit or maxProfit==M[i][j]-leftVal:
            path += "E"
j=j-1
continue
        if profit== topProfit or maxProfit==M[i][j]-topVal:
            path += "S"
i=i-1
continue

path=path[:-1]
print path[::-1]

return arr[m][n]
```

Time Complexity: As there are 2 for loops; one for the iterating through columns and another iterating through rows. Hence, the time complexity is **Θ(mn)**

Space complexity: The memory utilized by arr[][] is Θ(mn). Hence, the space complexity is **Θ(mn).**

(c) Suppose m = n = 2k for some integer k > 0. Design a recursive divide-and conquer algorithm to solve problem M in $\Theta(n^2)$ time and $\Theta(n)$ extra space. Find and solve the recurrences for running time and space usage.

Work Flow:

1. calculate the (buyValue, profit) tuples of each cell. Only store in memory, if the cells belong to any of the above array (BorderArrays).
2. The base case is a grid of size of 1*1. At this point we already know the (buyValue, maxProfit) values for the adjacent top and left cells as we have stored the values of the border cells. So, we can identify the path through recursion for reaching the cell. And append this path value (E or S) to the path string.
3. For path finding, we will move quadrant by quadrant recursively. Find the quadrant of current position and repeat step 1.
4. After the recursion ends, we will now know which quadrant to solve next i.e. the quadrant from which it entered the current quadrant.
5. Print the path string.

Let:

BorderArrays [bottom, top, left, right, midRow1, midRow2, midCol1, midCol2]: Stores all the border values of each of the four quandrant. This will help us to understand the entry point of the quandrant

// BorderArrays will consume 8n space. However, it's still linear.

Pseudocode:

```
.CheckForward(n, M, topProfitRow,leftProfitCol):

    prevRow = topProfitRow
    BorderArrays = [] //Empty array
    for i in 1 to n:
       row = []
       for j in 1 to n:
           if j == 1:
              row[j] = solve(leftProfitCol[i], prevRow[j], M[i][j])
           else:
    row[j] = solve(row[j-1], prevRow[j], M[i][j])
    BorderArrays = fillBorders(row[j],i,j,n)
    prevRow = row
```

```
    delete row
    return BorderArrays


path = “” // Global Variable initialized to empty string. It will store the final path
//We will keep on appending the path as ‘Direction’ + path
//This will give us the correct path to go from source to destination.
//As in the problem, we will be traversing up from bottom.


path(n, M, startPosition, currentPosition, parentBorder ):

        if currentPosition == (1,1)
            return
        if n==1:
            s, nextPosition = getDirection(
              parentBorder->left, parentBorder->top,
              currentPosition, M)
             path = s+path      //Explained in detailed few lines above, why we did this step
             return nextPosition


        Recurse, newPos = findQuadrant(startPosition, currentPosition, n)
        pos = (startPos, currPos)
        topProfitRow, leftProfitCol = findRowCol(pos, parentBorder)
        insideBorders = checkForward(n/2, Recurse, topProfitRow, leftProfitCol)

        pos = (newPos, currPos)
        currPos = pathfinder(n/2, Recurse, pos)

         pos = (newPos, currPos)
        Recurse, newPos = findQuadrant(pos)
        topProfitRow, leftProfitCol = findRowCol(pos, BorderArrays)
        insideBorder = forwardTracker(n/2, Recurse, topProfitRow, leftProfitCol )

        pos = (newPos, currPos)
        currPos = path(n/2, Recurse, pos)

        Recurse, newPos = findQuadrant(pos)

        topProfitRow, leftProfitCol = findRowCol(pos, BorderArrays)
        insideBorder = forwardTracker(n/2, Recurse, topProfitRow, leftProfitCol )

        pos = (newPos, currPos)
        currPos = pathfinder(n/2, Recurse, pos)
```

routeFinalExtract(M, n):

1. Create 1-D array row (1->n) and initialize it to –infinity.
2. Create 1-D array col (1->n) and initialize it to –infinity.
3. BorderArrays = checkForward(n, M, row, col)
4. path(n, M, (1,1) , (n,n), BorderArrays)
5. print path

Description of the functions used:

1. solve(left, top, pos) : Returns (buyValue, maxProfit) tuple based on the (buyValue, maxProfit) values of left, top and pos.
   The logic used is similar to that used in solving the previous question.

2. fillBorders(tuple, pos, n) : This checks whether position 'pos' belongs to any of the eight BorderArrays array and appends the tuple to that array.

3. getDirection(leftCol, topRow, currPos, M) : This returns a tuple (s, nextPos) where s is either E or S based on whether currPos was reached from the adjacent left cell or adjacent top cell.

4. findQuadrant(pos) : This returns a tuple (Recurse, newPosition) where Recurse is the sub-grid (top left, top right, bottom left, bottom right) where currPos lies. newPos is the top left co-ordinate of that sub-grid.

5. findRowCol(pos, BorderArrays) -> This returns which row and column should be returned from the BorderArrays (which is of type BorderArrays) based on the pos.

Space Complexity:

At every recursion we use 8n space to store the BorderArrays and recursively call the function for a problem of size n/2. After coming out from the recursion, free the space from the stack.

$S(n) = S(n/2) + 8n$

$S(n) = S(n/4) + 8n/2 + 8n$ //Substituting the value of S(n/2) for the original equation

$S(n) = S(n/8) + 8n/4 + 8n/2 + 8n$

Generalizing:

$S(n) = 8n (1+ ½ + ¼ + 1/8 +….. + 1/2^{k-1})$ // since $n=2^k$

$S(n) = 8n (1 – (½)^k)/(1-1/2)$ // Applying GP Formula

$S(n) = 8n*2*(1-1/n)$

$S(n) = 16n - 16$

$S(n) = \Theta(n)$

Hence Proved.

Time Complexity:

For a grid of size nxn:

checkForward: Two for loops.Therefore, overall runtime is $\Theta(n^2)$.

path:

1. We solve exactly three subGrids of size n/2 in each recursion. (3T(n/2))
2. Each recursion also calls checkForward to compute and store the BorderArrays : $\Theta(n^2)$.
3. Base Case (n=1) runs in $\Theta(1)$ time

Thus the recurrence relation is as follows:

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

By applying Master's Theorem (case 3) in above recurrence we get $T(n) = \Theta(n^2)$.

Task 2. Gravity:

(a) Give an algorithm that computes the net gravitional forces acting on unit-mass objects at all cells of the n * n grid in $O(n^4)$

Given
Mass = 1 (Unit mass)
Force = $G(m_1 m_2) / r*r$ (magnitude)

Net gravitational force acting on a single object
= Summation of gravitation force exerted by the object with respect to all the objects in the grid
= For one object, the complete grid is traversed. Which takes $O(n^2)$ time.

Net gravitational force acting on all the object
= In order to traverse all the objects, we need to cover the complete grid. This will take $O(n^2)$ time.
= Similarly to calculate force on each object, it takes $O(n^2)$ time.
= Hence, the net gravitational force acting on all the objects is: $O(n^2 * n^2) = O(n^4)$

Psuedo Code:

```
Force_{i,j} = [][]
for( int i =0; i<m; i++):
        for( int j=0; j<n; j++):
                for( int k =0; k<m; k++):
                        for( int l=0; l<n; l++):
                                r^2 = (x_{i,j} – x_{k,l})^2 + (y_{i,j} – y_{k,l})^2
                                Force_{i,j} = Force_{i,j} + G(m_{i,j} * m_{k,l}) / r^2
```

(b) Explain how you will solve the problem given in part (a) in $O(n^2 \log n)$ time.

In this problem, we will have to multiple the whole matrix with itself; and show that this can be done in $O(n^2 \log n)$ time.

This is can be done by 2D convolution matrix multiplication. Hence, the complete matrix needs to be computed over itself.

Well we know that 2 polynomials can be multiplied in O(nlogn) time using FFT. However, in our problem, we need to multiply 2 matrixes.

Multiplying matrix can be decomposed as the computation of 2N one dimensional DFT.

And each polynomial multiplication will require O(nlogn) time for multiplication.

So, 2D DFT can be effectively multiplied in O(n*n logn) = $O(n^2 \log n)$ time

Detailed proof with a small example:

In order to prove, let us consider a simple 2D matrix:

1	2
3	4

1	2
3	4

Let us multiply this 2 matrixes.

The above matrix multiplication can be done in 2 steps:

1. First convert matrix into bivariate polynomial
2. Then we will convert the bivariate into univariate polynomial
3. Finally, we will apply FFT to reach the solution.

MATRIX -> BIVARIATE POLYNOMIAL -> UNIVARIATE POLYNOMIAL ->

MULTIPLY NOW

First convert matrix into bivariate polynomial

Let us take a simple example,

1	2
3	4

This can be written as: $1 * x^0y^0 + 2 * x^1y^0 + 3 * x^0y^1 + 4 * x^1y^1$

Similarly, when you multiply two matrixes we get,

1	2
3	4

1	2
3	4

$(1 * x^0y^0 + 2 * x^1y^0 + 3 * x^0y^1 + 4 * x^1y^1)^2 =$

$(1 + 2x + 3y + 4xy)^2 = 1 + 4x + 6y + 20xy + 16x^2y + 24xy^2 + 16x^2y^2$

Over here, the terms with degree greater than **xy** are not important, because they will not affect the required result. Practically we can think of them as the effect by external forces on the objects. However, as per the problem, we are only interested in the forces applied by the objects within the grid/matrix.

Hence, we can only consider:

1+ 4x + 6y + 20xy.

Force and Direction:

Now, we will try to figure out the force and direction at each of the cell in the matrix.

In order to calculate the force at each cell, we will have to consider a matrix of 2n*2n matrix.

Base Matrix(n*n) and Bigger Matrix(2n*2n)

1	2
3	4

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

Hence, when we calculate the force that will be applied by coloured cell on each cell of the entire matrix: (in all directions)

Index	0	1	2	3
0	G/8 $\frac{x}{\sqrt{2}}+\frac{y}{\sqrt{2}}$	G/5 $\frac{2x}{\sqrt{5}}+\frac{y}{\sqrt{5}}$	G/4 $\frac{x}{\sqrt{4}}+\frac{y}{\sqrt{4}}$	G/5 $\frac{x}{\sqrt{25}}+\frac{y}{\sqrt{5}}$
1	G/5 $\frac{x}{\sqrt{5}}+\frac{y}{\sqrt{5}}$	G/2 $\frac{x}{\sqrt{2}}+\frac{y}{\sqrt{2}}$	G/1 $\frac{x}{\sqrt{1}}+\frac{y}{\sqrt{1}}$	G/2 $\frac{x}{\sqrt{2}}-\frac{y}{\sqrt{2}}$
2	G/4 $\frac{0x}{4}+\frac{2y}{\sqrt{4}}$	G/1 $\frac{0x}{\sqrt{1}}+\frac{y}{\sqrt{1}}$	1	G/1 $\frac{0x}{\sqrt{1}}-\frac{y}{\sqrt{1}}$
3	G/5 $-\frac{x}{\sqrt{5}}+\frac{2y}{\sqrt{5}}$	G/2 $-\frac{x}{\sqrt{2}}+\frac{y}{\sqrt{2}}$	G/1 $-\frac{x}{\sqrt{1}}+\frac{y}{\sqrt{1}}$	G/2 $-\frac{x}{\sqrt{2}}-\frac{y}{\sqrt{2}}$

Now, the force applied by the ‘1’ (first element) on the entire matrix:

1	G/1 $\frac{0x}{\sqrt{1}} - \frac{y}{\sqrt{1}}$
G/1 $-\frac{x}{\sqrt{1}} + \frac{y}{\sqrt{1}}$	G/2 $-\frac{x}{\sqrt{2}} - \frac{y}{\sqrt{2}}$

Now, we calculate the force exerted of the ‘2’ (second element) on the entire matrix:

G/1 $\frac{0x}{\sqrt{1}} + \frac{y}{\sqrt{1}}$	G/1 $\frac{0x}{\sqrt{1}} - \frac{y}{\sqrt{1}}$
G/1 + G/2 $-\frac{x}{\sqrt{1}} + \frac{y}{\sqrt{1}} - \frac{x}{\sqrt{2}} + \frac{y}{\sqrt{2}}$	G/2 + G/2 $-\frac{x}{\sqrt{2}} - \frac{y}{\sqrt{2}} - \frac{x}{\sqrt{1}} + \frac{y}{\sqrt{1}}$

Similarly, we can calculate the force exerted of the ‘3’ (third element) on the entire matrix.

Similarly, we can calculate the force exerted of the ‘4’ (forth element) on the entire matrix.

Hence, we get the force and direction of all the four elements. And we can ignore all the remaining forces, as the higher order terms(external forces) are not required.

Convert bivariate into univariate polynomial:

In order to multiple two bivariate polynomial; initially, we will convert it to univariate. For that we will perform the following operations:

Let f, g be bivariate polynomials.

Their product is:

$(fg)(xy) = \sum_{i,j,i',j'=0}^{n} f_{I,j}\, g_{i',j'}\, x^{i+i'}\, y^{j+j'}$

$= \sum_{i,j=0}^{2n} \left(\sum_{i'=0}^{\min(n,i)} * \sum_{j'=0}^{\min(n,j)} f_{i',j'}\, g_{i-i',j-j'} \right) x^i y^j$

Now, we would reduce the problem of multiplying two bivariate polynomials of degree n in each variable, to the problem of multiplying two univariate polynomials of degree $O(n^2)$, and then apply the algorithm using the standard FFT.

Let N be large enough to be determined later, and define the following univariate polynomials:

$F(z) = \sum_{i,j=0}^{n} f_{i,j}\, z^{Ni+j}$, $G(z) = \sum_{i,j=0}^{n} g_{i,j}\, z^{Ni+j}$

We can clearly compute F, G from f, g in linear time, and as deg(F), deg(G) ≤ (N + 1)n, we can compute F · G in time O((Nn) log(Nn)). The only question is whether we can infer f · g from F · G.

Lemma: Let N >= 2n + 1. If $H(z) = F(z)G(z) = \sum H_i z^i$; then

$(fg)(xy) = \sum_{i,j=0}^{2n} H_{Ni+j} x^i y^i$

Proof:

$H(z) = F(z)G(z) = \left(\sum_{i=0}^{n} f_{i,j}\, z^{Ni+j} \right) \left(\sum_{i=0}^{n} g_{i,j}\, z^{Ni'+j'} \right)$

$= \sum_{i,j,i',j'=0}^{n} f_{i,j}\, g^{i',j'} z^{N(i+i')+(j+j')}$

We need to show that the only solutions for $N(i + i') + (j + j') = N i^* + j^*$

where $0 \le i, i', j, j' \le n$ and $0 \le i^*, j^* \le 2n$,

are these which satisfy $i+i' = i^*$, $j + j' = j^*$.

As $0 \le j + j', j^* \le 2n$ and $N > 2n$, if we compute the value modulo N we get that

$j + j' = j^*$, and hence also $i + i' = i^*$.

Using this bivariate to univariate method. Let us convert bivariate polynomial of the example into univariate:

Eg: (1+2x+3y+4xy)

Here, n=2;

Formula Used: Univariate power = Ni + j;

N=2n+1;

Therefore, N=5;

Hence, the equation we get:

$2x = z^{(5+0)} = 2z^5$

$3y = z^{(0+1)} = 3z^1$

$4xy = z^{(5+1)} = 4z^6$

Hence, the converted univariate matrix is: $1+2z^5 +3z +4z^6$

Now that we have converted the bivariate to univariate with $O(n^2)$. This is like solving multiplication of 2 univariate polynomials with highest degree of $O(n^2)$.

Using FFT we get the solution as nlogn for n degree. Hence, for n^2 degree, we will get:

$n^2 \log n^2 = 2n^2 \log n$

$O(n^2 \log n)$

Hence, Proved.

Solution 3 a.

Given : Matrix B is a matrix with $(\frac{n}{\frac{1}{n^k}} \times \frac{n}{\frac{1}{n^k}})$ rotator matrices. Here as k = 2, B has $(\sqrt{n} \times \sqrt{n})$ sub matrices which are rotator matrices. Each rotator matrix will be of size $\sqrt{n} \times \sqrt{n}$. Lets represent B in following form:

$$B = \begin{matrix} B_{11} & \cdots & B_{\sqrt{n}\ 1} \\ \vdots & \ddots & \vdots \\ B_{1\sqrt{n}} & \dots & B_{\sqrt{n}\sqrt{n}} \end{matrix}$$

where, Bij is a rotator matrix of size $\sqrt{n} \times \sqrt{n}$.

A is a $n \times n$ matrix. Lets divide it into $\left(\sqrt{n} \times \sqrt{n}\right)$ smaller matrices of size $\left(\sqrt{n} \times \sqrt{n}\right)$. Hence A will be of the form:

$$A = \begin{matrix} A_{11} & \cdots & A_{\sqrt{n}\ 1} \\ \vdots & \ddots & \vdots \\ A_{1\sqrt{n}} & \dots & A_{\sqrt{n}\sqrt{n}} \end{matrix}$$

We need to multiply these two matrices, and get the result C = A x B. The matrix C will be of size $\left(\sqrt{n} \times \sqrt{n}\right)$ and can be represented as:

$$C = \begin{matrix} C_{11} & \cdots & C_{\sqrt{n}\ 1} \\ \vdots & \ddots & \vdots \\ C_{1\sqrt{n}} & \dots & C_{\sqrt{n}\sqrt{n}} \end{matrix}$$

where:

$$C_{ij} = \sum_{1}^{\sqrt{n}} A_{ik} \times B_{kj}$$

We know that, B_{kj} is a rotator matrix.

Let D = A_{ik} and E = B_{kj} . Here D and E are of size $m \times m$ where m = $\sqrt{n}$.

We know that E is a rotator matrix. Any rotator matrix can be converted to a circulant matrix by shifting columns. This can be done in $O(m)$ time. To get the correct required result we have to shift rows of matrix E. For each column i shifted in matrix E by value of z , the row i in matrix D should be shifted by the same value z. This shifting takes $O(m)$ time. After shifting the columns of E appropriately, lets say we get a circulant matrix of E' of the form:

$$\begin{matrix} e_0 & e_{m-1} & \dots & e_1 \\ e_1 & e_0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & e_{m-1} \\ e_{m-1} & \dots & e_1 & e_0 \end{matrix}$$

Circulant matrices have the property that $e_{ij} = e_{i+1,j+1}$. Lets divide E' into Upper Triangle and Lower Triangle Matrices U and L. L can be represented as below:

e_o	0	…	0	0
e_1	e_o	⋱	⋱	0
e_2	⋱	⋱	⋱	⋮

⋮	⋱	e_1	e_o	0
e_{m-1}	…	e_2	e_1	e_0

U can be presented as below:

0	e_{m-1}	…	e_2	e_1
0	0	⋱	⋱	e_2
0	⋱	⋱	⋱	⋮
⋮	⋱	0	0	e_{m-1}
0	…	0	0	0

E = U + L.
Hence D x E = D(U + L) = DxU + DxL

Consider the multiplication R' = DxU:
Let $d_0 = [p_0, p_1, …., p_{m-1}]$ be the first row of the matrix D.
$R'_{0j} = d_0$ x L -> Elements in the first row of R'.

$R'_{00} = p_0 \times e_0 + p_1 \times e_1 + p_2 \times e_2 + \ldots\ldots + p_{m-1} \times e_{n-1}$
$R'_{01} = p_0 \times 0 + p_1 \times e_0 + p_2 \times e_1 + \ldots\ldots + p_{m-1} \times e_{n-2}$
$R'_{02} = p_0 \times 0 + p_1 \times 0 + p_2 \times e_0 + \ldots\ldots + p_{m-1} \times e_{n-3}$
⋮
$R'_{0n-1} = p_0 \times 0 + p_1 \times 0 + p_2 \times 0 + \ldots\ldots + p_{m-1} \times e_0$

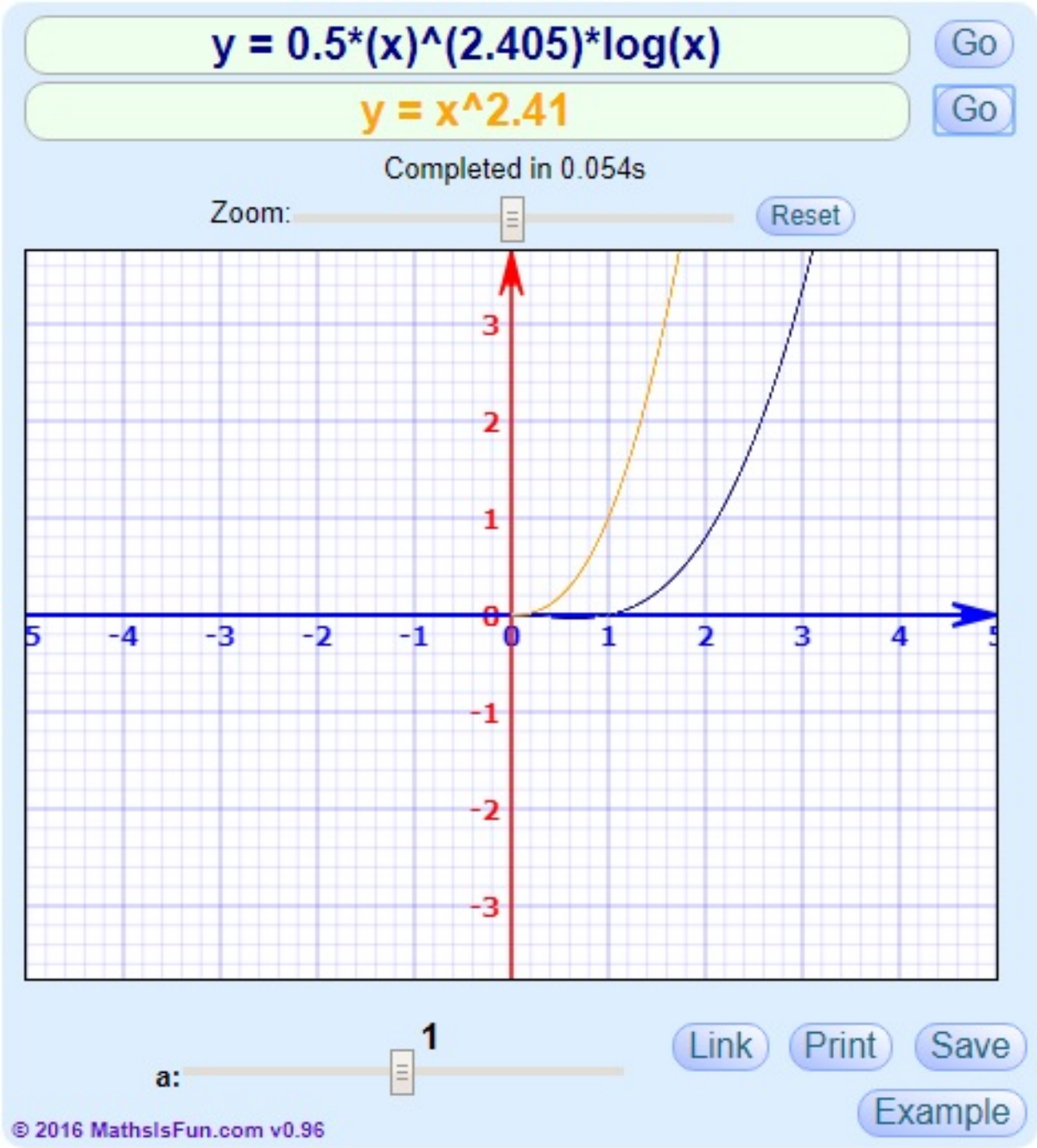
We can see that the values of R'_{0j} can be found from the convolution of two equations (represented in coefficient form) $p = [p_0, p_1, …., p_{m-1}]$ and $e = [e_0, e_1, …., e_{m-1}]$. Hence the row 0 of R' can be found if we know the convolution of p and e. Hence using FFT we can calculate the product of p and e in $O(mlogm)$ time. There are however, m rows in R'. So we will need to calculate the convolution m times. Hence the time required will be $O(m \times mlogm) = O(m^2 \log m)$.
Hence, now we add the result and get DxE in $O(m^2)$. Hence total time will be $O(m^2 \log m) + O(m^2) = O(m^2 \log m)$

Hence time required to multiply a random matrix of size mxm and a circular matrix of size mxm will take $O(m^2 \log m)$ time.

Using Strassen's Matrix multiplication algorithm, we can reduce the number of matrix multiplication and addition.

Using Strassen's Matrix multiplication we can say that, the multiplication of pxp matrix takes $O\left(p^{\log_2 7}\right)$ time. However, here the multiplication cost of each sub matrix is $O(m^2 \log m)$.

So outing values: p = $\sqrt{n}$ and m = $\sqrt{n}$ time complexity = $O\left(n^{0.5x\log_2 7} \times 0.5 \times n \log n\right)$ = $O(n^{2.405} \log n)$. From the graph below we can see that :



$$n^{2.405} \log n = o(n^{2.41})$$

Hence the answer.

Solution 3 B:

We have B consisting of n^2 / $n^{2/k}$ rotatory matrices. From the above solution we can convert A and B into sub matrices of size $n^{1/k}$ x $n^{1/k}$. Let m = $n^{1/k}$ so we can say that we get the

product of the two matrices using convolution in m^2logm time. However, using Strassen's matrix multiplication by dividing the matrices into mxm submatrices, we can say that the cost of multiplication can be reduced by reducing the number of matrix multiplication and additions. The cost of matrix multiplication takes m^2log m in out case while the cost of adding two matrices is m^2. Hence we can see that the cost of multiplication dominates the cost of addition. We consider the matrices to have numbers hence multiplication and addition of constants takes constant time.
But in out case we need to consider the time for multiplication which is $O(m^2 \log m)$.

Out large matrix when divided into smaller matrices of size $n^{1/k}$ x $n^{1/k}$, in itself becomes of size n / $n^{1/k}$ x n / $n^{/k}$. Let p = n / $n^{1/k}$ then the cost of matrix multiplication becomes :

$$O\left(p^{\log_2 7}\right)$$

However, the cost of multiplication is $O(m^2 \log m)$. Hence the final multiplication cost will be: $O\left(p^{\log_2 7}\right)$ x $O(m^2 \log m)$. Which is: $O\left(p^{\log_2 7} * m^2 \log m\right)$.
Substituting, p = n/$n^{1/k}$ and m = $n^{1/k}$, we get

$p^{\log_2 7} * m^2 \log m$ = $n^{(1-\frac{1}{k})\log_2 7}$x $n^{2/k}$ log n = $n^{\frac{2}{k}+(1-\frac{1}{k})\log_2 7}$xlog n

Hence the complexity will be : $O\left(n^{\frac{2}{k}+(1-\frac{1}{k})\log_2 7}\text{xlog n}\right)$