CSE548, AMS542: Analysis of Algorithms, Fall 2017 Date: Oct 25

Homework #1

Group Number: 52

Name	SBU ID	% Contribution
Rahul Bhansali	111401451	33.33%
Kiranmayi Kasarapu	111447596	33.33%
Neha Indraniya	111499447	33.33%

Collaborating Groups

Group Number	Specifics Number (e.g., specific group member? specific task/subtask?)
	Shayan -- Question 2(b)
56	Kedar -- Question 2(b)
19	Chaitanya Kalantri -- Question 2(b), 1(c)
26	Akhil Bhutani -- Question 1(a), 1(b) 1(c), 2(b)
26	Aditya - Question 2(b)

External Resources Used

	Specifics (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

Task 1. [60 Points] Most Profitable Trade Routes
(a) [10 Points] 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.

Let:

The solution to M_L (leftmost n-1 columns) = M_L_profit
The solution to M_T (topmost m-1 rows) = M_T_profit
The solution to M = M_profit

Both M_L_profit and M_T_profit store their solution as a tuple: (buyValue, profit)
where:

1. **buyValue:** is the minimum possible buying price starting from (1,1) upto that position.
2. **profit:** is the maximum possible profit that can be earned starting from (1,1) upto that position.

So, M_L_profit is the solution of the grid starting at (1,1) and ending at (m,n-1)
And M_T_profit is the solution of the grid starting at (1,1) and ending at (m-1,n)

Now, to calculate the solution at M, there are three cases:

1. **M is inaccessible**: then there exists no solution for M, so just return (+infinity,-infinity) which denotes that there is no answer possible.
2. **M is empty**: in this case the ***profit*** value at this cell will simply be the maximum of the profit of the cell just to its left and of the cell just to its top (i.e. $M_{L\text{ and}}$ M_T). The ***buyValue*** for this cell will be the minimum of the ***buyValue*** of the cell just to its left and of the cell just to its top (i.e. $M_{L\text{ and}}$ M_T).
3. **M is a shop**: in this case the ***profit*** value will be the **maximum** of four possible values:
   a. Profit value of the left cell
   b. Profit value of the top cell
   c. Shop Value of that cell – buyValue of left cell
   d. Shop Value of that cell – buyValue of top cell

   The ***buyValue*** of this cell will be the **minimum** of three possible values:
   a. Shop Value of that cell
   b. buyValue of the left cell
   c. buyValue of the top cell

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.
Hence, proved.

(b) [5 Points] Explain how you will use the observation from part (a) to solve problem M in Θ (mn) time and Θ (mn) extra space.

Consider a 2-D array dp[][] of size mXn where each element of the array (dp[i][j]) stores a tuple (buyValue, profit) where buyValue represents the minimum buy value found up to that cell (i.e. starting from (1,1) to (i,j)) and profit represents the maximum possible profit up to that cell. From the observation in previous part, we know that the solution to a particular cell (i,j) can be found if we know the solution to the cell just above it (i-1,j) and the solution to the cell just left to it (i,j-1).
So, if we know the solutions to each cell in the leftmost column and each cell in the topmost row, then we can start calculating the answers for all the other cell positions by traversing row by row starting from 2nd row till the last row.

Based on above observations, here is the pseudocode:
dp[i][j] -> this is an element of the 2D-array where each element stores a tuple (buyValue, profit).
buyValue -> this is the minimum buying price obtained so far starting from (1,1) to (i,j)
profit -> this is the maximum profit possible starting from (1,1) to (i,j)
grid[i][j] -> this denotes the grid. Its value is equal to -1 if that cell is inaccessible, 0 if its empty and the shop value if there's a shop.

The final answer will be stored is the *profit* value of the (buyValue,profit) dp[m][n] tuple.

```
def findMaxProfit(m,n,grid,dp):
       infinity = 1000
       buyValue = +infinity
       profit = -infinity
       #if F is inaccessible return
       if grid[m][n]<0:
       return (buyValue,profit)
       dp[1][1] = (buyValue, profit)
       #Initialize the values of topmost row and leftmost column
       for i in range(2,n+1):
       #Shop
       if grid[1][i]>0:
              # it is not possible to make a larger profit by selling at this
position
              if grid[1][i]< buyValue:
              buyValue = grid[1][i]
             #a profit can be made at this position. So check if its larger than the
current maximum profit and update
              else:
              if(grid[1][i]-buyValue>profit):
                     profit = grid[1][i]-buyValue
              #fill dp[0][i] cell with the updated buyValue and profit values
              dp[1][i] = (buyValue, profit)
       #Empty
       elif grid[1][i]==0:
              #the buyValue and profit value will be the same as the previous row
cell
              dp[1][i] = dp[1][i-1]
       #Inaccessible
       elif grid[1][i]<0:
              #all remaining row cells are inaccessible so set (buyValue,profit) =
(+infinity,-infinity) and break
              for j in range(i,n+1):
              dp[1][j]=(+infinity,-infinity)
               break
       buyValue = +infinity
       profit = -infinity
       for i in range(2, m+1):
       # Shop
       if grid[i][1] > 0:
              # it is not possible to make a larger profit by selling at this
position
              if grid[i][1] < buyValue:
                buyValue = grid[i][1]
              # a profit can be made at this position. So check if its larger than
the current maximum profit and update
              else:
              if (grid[i][1] - buyValue > profit):
                     profit = grid[i][1] - buyValue
                # fill dp[0][i] cell with the updated buyValue and profit values
                dp[i][1] = (buyValue, profit)
        # Empty
        elif grid[i][1] == 0:
                # the buyValue and profit value will be the same as previous column
cell
                dp[i][1] = dp[i-1][1]
        # Inaccessible
        elif grid[i][1] < 0:
                # all remaining row cells are inaccessible so set (buyValue,profit) =
(+infinity,-infinity) and break
                for j in range(i, m+1):
                dp[j][1] = (+infinity, -infinity)
                break
        #Now run the main loop          <- This loop consumes
                                           maximum time since
                                        it runs for mxn iterations.

        for i in range(2,m+1):
            for j in range(2,n+1):
                #extract the (buyValue,profit) of left and top cells
                lval, lprofit = dp[i][j - 1]
                tval, tprofit = dp[i - 1][j]
                # Shop
                # the profit will be a maximum of the profit values to the left cell,
top cell,
                # or the profit generated at that point and the buyValue will be
updated to the minimum of left cell, top cell
                # and grid[i][j] value
                if grid[i][j]>0:
                buyValue = min(lval,tval,grid[i][j])
                profit = max(lprofit,tprofit,grid[i][j]-lval,grid[i][j]-tval)
                dp[i][j]=(buyValue,profit)
                #Empty
                elif grid[i][j]==0:
                dp[i][j] = (min(lval,tval),max(lprofit,tprofit))
                #Inaccessible
                elif grid[i][j]<0:
                dp[i][j] = (+infinity,-infinity)
        #backtrack
        i=m
        j=n
        path=""
        while i>=1 and j>=1:
        #top cell is inaccessible
        if grid[i-1][j]<0:
                path+="E"
                j=j-1
                continue
        #left cell is inaccessible
        if grid[i][j-1]<0:
               path+="S"
               i=i-1
               continue
        lval, lprofit = dp[i][j - 1]
        tval, tprofit = dp[i - 1][j]
        #empty
        if grid[i][j]==0:
               bVal,profit = dp[i][j]
               if profit==lprofit:
               path+="E"
               j=j-1
               continue
               if profit==tprofit:
               path+="S"
               i=i-1
               continue
        #shop
        if grid[i][j]>0:
               bVal,profit=dp[i][j]
              if profit ==lprofit or profit==grid[i][j]-lval:
               path+="E"
               j=j-1
               continue
               if profit==tprofit or profit==grid[i][j]-tval:
               path+="S"
               i=i-1
               continue
        path=path[:-1]
        print path[::-1]
        return dp[m][n]
```

Based on the above code,

Time Complexity:

we can see that the biggest loop runs mxn times (as highlighted in bold in code), when we are filling each cell of the dp array, i.e. time complexity is of the order **Θ(mn)** time.

Space Complexity:

The memory that we are allocating is the dp array of size mxn i.e. space complexity is of **Θ(mn).**
Hence, proved.

(c) [35 Points] Suppose m = n = 2k for some integer k > 0. Design a recursive divide-and conquer algorithm to solve problem M in Θ n 2 time and Θ (n) extra space. Find and solve the recurrences for running time and space usage.

The data structures we'll be using comprise of 8 1-D arrays to store the border cell values in each recursion:
top: stores the (buyValue,profit) value of the cells in topmost row.
left: stores the (buyValue,profit) value of the cells in leftmost column.
right: stores the (buyValue,profit) value of the cells in rightmost column.
bottom: stores the (buyValue,profit) value of the cells in bottommost row.
mRow1: stores the (buyValue,profit) value of the cells in middle row which is n/2th row. (last row of the top quadrant).
mRow2: stores the (buyValue,profit) value of the cells in middle row which is (n/2)+1 th row. (first row of the bottom quadrant).
mCol1: stores the (buyValue,profit) value of the cells in middle column which is n/2th column. (last column of the left quadrant).
mCol2: stores the (buyValue,profit) value of the cells in middle column which is (n/2) +1th column. (first column of the right quadrant).

Let's call the list of above 8 1-D arrays as **BorderArrays**

Approach:

1. Firstly, we'll calculate the (buyValue, profit) tuples of each cell. But we will store in memory only if the cells belong to any of the above array (BorderArrays). Therefore, total space will be 8XN = Θ(N) space.

buyValue is the minimum buying price up to that cell starting from (1,1).
profit is the maximum possible profit up to that cell starting from (1,1)

2. Secondly, we need the path. We already have the border cell values for the complete grid. Find which quadrant the current position belongs to and go to step 1. We do this recursively till we hit the base case:
3. The base case is a grid of size of 1X1. At this point we already know the (buyValue, profit) values for the adjacent top and left cells as we have stored the values of the border cells. So, we can determine which cell was used to come to the current cell. We accordingly append this path value (E or S) to the path string.
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.

Pseudocode:

#forwardTracker computes (buyValue, profit) for each cell but stores (buyValue, profit) values of only the BorderArrays (mentioned above). Also, the value at each cell can be computed by storing in memory the solution of only the previous row and the current row i.e. $\Theta(n)$ space.

forwardTracker(n, grid, topProfitRow, leftProfitCol):

```
1.     Initialize prevRow to topProfitRow
2.     Initialize BorderArrays to empty arrays.
3.     for i -> 1 to n:
4.         row -> []
5.         for j -> 1 to n:
6.              if j==1:
7.                  row[j] = solve(leftProfitCol[i],prevRow[j], grid[i][j])
8.              else:
9.                  row[j] = solve(row[j-1], prevRow[j], grid[i][j])
10.           BorderArrays = populateBorderCells(row[j],i,j,n)
11.       prevRow = row
12.       free(row)
13.  return BorderArrays
```

path = “”   <- Global Variable initialized to empty string. It will store the final path

#pathFinder does two things: It recursively backtracks and stores the path from currentPosition till it reaches a “border cell” and returns this “border cell” so that the appropriate next grid can be called in the recursion. There will be three such recursions as one sub-grid can be eliminated once we know which sub-grid to choose.

pathFinder(n, grid, startPosition, currentPosition, parentBorderArrays):

```
1.     if currentPosition == (1,1)
2.         return
3.     if n==1:
4.         s, nextPosition = getDirection(
           parentBorderArrays->left, parentBorderArrays->top,
           currentPosition, grid)
5.          path = s+path
6.          return nextPosition

7.     gridToRecurse, newStartPos = findQuadrant(startPosition,
   currentPosition, n)
8.     topProfitRow, leftProfitCol = findRowAndCol(startPosition,
   currentPosition, parentBorderArrays)
9.     childBorderArrays = forwardTracker(n/2, gridToRecurse,
   topProfitRow, leftProfitCol )
10.  currentPosition = pathfinder(n/2, gridToRecurse, newStartPos,
   currentPosition)
```

```
        11.  gridToRecurse, newStartPos = findQuadrant(startPosition,
           currentPosition)
        12.  topProfitRow, leftProfitCol = findRowAndCol(startPosition,
           currentPosition, parentBorderArrays)
        13.  childBorderArrays = forwardTracker(n/2, gridToRecurse, topProfitRow,
           leftProfitCol )
        14.  currentPosition = pathfinder(n/2, gridToRecurse, newStartPos,
           currentPosition)

        15.  gridToRecurse, newStartPos = findQuadrant(startPosition,
           currentPosition)
        16.  topProfitRow, leftProfitCol = findRowAndCol(startPosition,
           currentPosition, parentBorderArrays)
        17.  childBorderArrays = forwardTracker(n/2, gridToRecurse, topProfitRow,
           leftProfitCol )
        18.  currentPosition = pathfinder(n/2, gridToRecurse, newStartPos,
           currentPosition)
```

```
findTradeRoute(grid, 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.     parentBorderArrays = forwardTracker(n, grid, row, col)
4.     pathfinder(n, grid, (1,1) , (n,n), parentBorderArrays)
5.     print path
```

Helper Functions used:

1. **solve(left, top, pos)** -> This function returns the appropriate (buyValue, profit) tuple based on the (buyValue, profit) values of left, top and pos.
    left is the adjacent left cell of pos.
    top is the adjacent top cell of pos.
    The logic used is similar to that used in solving the previous question.
2. **populateBorderCells(tuple, i, j, n)** -> This checks whether position (i,j) belongs to any of the eight **BorderArrays** array and appends the *tuple* to that array. *tuple* is the (buyValue, profit) value of position (i,j).
3. **free(row)** -> This deletes the row array.
4. **getDirection(leftCol, topRow, currentPosition, grid)** -> This returns a tuple (s, nextPosition) where s is either E or S based on whether currentPosition was reached from the adjacent left cell or adjacent top cell. nextPosition is that cell position i.e. left or top.
5. **findQuadrant(startPosition, currentPosition, n)** -> This returns a tuple (gridToRecurse, newStartPosition) where gridToRecurse is the sub-grid (top left, top right, bottom left, bottom right) where currentPosition lies. newStartPosition is the top left co-ordinate of that sub-grid.

```
6. findRowAndCol(startPosition, currentPosition, parentBorderArrays) -> This
   returns which row and column should be returned from the parentBorderArrays
   (which is of type BorderArrays) based on the startPosition and currentPosition.
```

Space Complexity:

For a grid of size nxn:

forwardTracker -> 8xn since 8 arrays of size n are stored.

pathfinder -> The space usage recurrence is given below:

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

Explanation:
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, that space is freed from the stack and is not stored in the memory i.e. we use space to store BorderArrays of only the current sub-problem.

Solving:
S(n) = S(n/2) + 8n
S(n) = S(n/4) + 8n/2 + 8n
S(n) = S(n/8) + 8n/4 + 8n/2 + 8n
S(n) = S(n/16) + 8n/8 + 8n/4 + 8n/2 + 8n
.
.
.
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)
S(n) = 8n*2*(1-1/n)
S(n) = 16n – 16
S(n) = Θ(n)
Hence Proved.

Time Complexity:

For a grid of size nxn:

forwardTracker ->

Line 5 runs nxn times => Θ(n^2) and inner loop lines runs in Θ(1) time. Therefore, overall runtime is Θ(n^2).

pathfinder ->
1. We solve exactly three sub-problems of size n/2 in each recursion. (3T(n/2))
2. Each recursion also calls forwardTracker to compute and store the BorderArrays -> Θ(n^2).
3. Base Case (n=1) runs in Θ(1) time

Thus the recurrence relation is as follows:

T(n) = 3T(n/2) + Θ(n^2)

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

Task 2:
(a) **[10 Points] 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) time.**

In an n x n grid, there are a maximum of n^2 objects, since at each cell, we are considering only one mass objects, i.e even if multiple objects are present in each cell, we are adding all the masses of each object and considering as a single object.
We are asked to find, forced exerted on each object by other objects in the entire grid.

Here is an outline of the algorithm:
On each object at cell (i, j), maximum of n^2 -1 objects(i.e all object except object at (i, j)) forces will be acting on it. We have to calculate the forces by the given formula

$$F_{p,q} = \frac{m_p m_q}{r^2}$$

The above equation represents the force acting on object p, by object q. The resultant force $F_{p,q}$ will have an x-component and y-component i.e $F_{p,q(x)}$ and $F_{p,q(y)}$. There are only 2 components here since the universe is considered as flat 2D universe.

So to calculate force acting on object (i,j), the total force would be:

$$F_{(i,j)(x,y)} = \sum_{a=\{1..n\}-\{i\},\ b=\{1..n\}-\{j\}} \frac{m_{(i,j)} m_{(a,b)}}{r^2}$$

The summation here is vector addition, So each force $\frac{m_{(i,j)}m_{(a,b)}}{r^2}$ has a x-component and y-component, lets represent them by $F_{(i,j)(a,b)x}$ and $F_{(i,j)(a,b)y}$, so the total force reduces down to

$$F_{(i,j)} = \sum_{a=\{1..n\}-\{i\},\ b=\{1..n\}-\{j\}} F_{(i,j)(a,b)x} + \sum_{a=\{1..n\}-\{i\},\ b=\{1..n\}-\{j\}} F_{(i,j)(a,b)y}$$

So to calculate force at each cell(i.e on one object at cell (i,j)), we have n^2 multiplications and $2n^2$ additions. So the total cost for calculating force for each object is $n^2 + 2n^2 = 3n^2$. We have n^2 objects. To calculate force on each of these n^2 objects the cost will be $n^2 \times 3n^2 = 3n^4$. So the brute force solution takes $O(n^4)$ time to calculate the force on every object.
Since it is asked to calculate force on unit-mass objects, after calculating the force $F_{(i,j)}$ by $m_{(i,j)}$ which takes another n^2 time operation. Or while calculating the force $F_{(i,j)(x,y)}$ use $m_{(i,j)}$ as 1. In either case the complexity still remains as $O(n^4)$ (as $n^2 < n^4$ and we can ignore lower order terms)
A simple pseudo-code algorithm is shown below.

```
#define G

Distance( i, j, a, b )
   1. $r_x$ <- a - i
   2. $r_y$ <- b - j
   3. return $\sqrt{r_x^2 + r_y^2}$

Force( $m_{i,j}$,$m_{a,b}$,r )
   1. return G x $m_{i,j}$ x $m_{a,b}$ / $r^2$

GetDirection( i, j, a, b, r)
   1. return $\frac{(a-i)}{r}\hat{x} + \frac{(b-j)}{r}\hat{y}$

CalculateForce( S, n )

   1. Let F[0..n-1,0..n-1] be a new matrix
   2. for i <- 0 to n-1
   3.   for j <- 0 to n-1
   4.     $f_{(i,j)x}$ = 0, $f_{(i,j)y}$ = 0
   5.     $m_{i,j}$ = S[i,j]
   6.     for a <- 0 to n-1
```

```
 7.        for b <- 0 to n-1
 8.           if a != i and b != j
 9.                r <- Distance(i,j,a,b)
10.                $m_{a,b}$ <- S[a,b]
11.                force = Force($m_{i,j}$,$m_{a,b}$,$r_{x,y}$)
12.                Force[a,b].value += force
13.                Force[a,b].direction =
14.                            getDirection(i,j,a,b,r)
```

So in the above algorithm, lines 8-14 takes O(1).
The innermost for loop at line 7 is executed n^4 times since it is nested under 3 for loops.
So the time complexity of this algorithm is $n^4 * O(1) = O(n^4)$

(b) **[40 Points] Explain how you will solve the problem given in part (a) in $O(n^2 \log n)$time.**

In this question, we have to improve the above algorithm and solve it in $O(n^2\log n)$.
The solution boils down to, how much force each object say at cell(i, j) contributes to every other cell.
Let us consider the grid will all unit masses. We can calculate the contribution of force of the object at cell(i, j) has on every other object/cell. This can be done in $O(n^2)$ time. It takes only order n^2 time because, there are a total of n^2 objects, we have to calculate force for n^2 objects only. For calculating force with one object it takes constant amount of time, because, masses are known and to calculate distance between 2 cells, lets say (i, j) and (i', j') is given by the following formula

$$distance = \sqrt{(i'-i)^2 + (j'-j)^2}$$

Which again takes only constant amount of time i.e O(1)
We can calculate the force vector and store it in the resultant matrix itself.
The force vector is defined as follows

$$\widehat{F} = \frac{(i'-i)}{r}\widehat{x} + \frac{(j'-j)}{r}\widehat{y}$$

Again this is calculated in O(1) for each cell.
Now to calculate the force contributed by object at (i, j) at every other cell will be $n^2 \times O(1) = O(n^2)$
Simple algorithm is as follows:

```
ForceExertedByObject( S, n, i, j )
   1. Force = [0...n, 0...n]
```

```
for a <- 0..n-1
   for b <- 0..n-1
      if a != i and b != j
         r <- Distance(i,j,a,b)
         $m_{a,b}$ <- S[a,b]        //we consider it to be 1
         Force[a,b].value = Force(1,1,r)
         Force[a,b].direction = getDirection(i,j,a,b,r)

This takes on O($n^2$) time.
```

So we have a matrix as follows:
Say we are calculating the force contributed by object at index (2,2) on every other object
Since every object is considered to be unit mass, $m_x m_y = 1$ and the force magnitude at every cell reduces to G/r^2 where r is the distance between cell (2,2) and itself(The current cell for which we are calculating the force)
Lets see the calculation for index (0,0),

$$r^2 = (2-0)^2 + (2-0)^2 = 8$$
$$F = \frac{G}{r^2} = \frac{G}{8}$$
$$Direction = \frac{2}{\sqrt{8}}\widehat{x} + \frac{2}{\sqrt{8}}\widehat{y} = \frac{\widehat{x}}{\sqrt{2}} + \frac{\widehat{y}}{\sqrt{2}}$$

Similarly we can calculate for other cells. The Force magnitude values and directions are shown below for each cell

index	0	1	2	3
0	$\frac{G}{8}$ $\frac{\widehat{x}}{\sqrt{2}} + \frac{\widehat{y}}{\sqrt{2}}$	$\frac{G}{5}$ $\frac{2\widehat{x}}{\sqrt{5}} + \frac{\widehat{y}}{\sqrt{5}}$	$\frac{G}{4}$ $\frac{\widehat{x}}{2} + \frac{0\widehat{y}}{2}$	$\frac{G}{5}$ $\frac{\widehat{x}}{5} - \frac{\widehat{y}}{\sqrt{5}}$
1	$\frac{G}{5}$ $\frac{\widehat{x}}{\sqrt{5}} + \frac{2\widehat{y}}{\sqrt{5}}$	$\frac{G}{2}$ $\frac{\widehat{x}}{\sqrt{2}} + \frac{\widehat{y}}{\sqrt{2}}$	$\frac{G}{1}$ $\frac{\widehat{x}}{1} + \frac{0\widehat{y}}{1}$	$\frac{G}{2}$ $\frac{\widehat{x}}{\sqrt{2}} - \frac{\widehat{y}}{\sqrt{2}}$
2	$\frac{G}{4}$ $\frac{0\widehat{x}}{2} + \frac{2\widehat{y}}{2}$	$\frac{G}{1}$ $\frac{0\widehat{x}}{1} + \frac{\widehat{y}}{1}$	1	$\frac{G}{1}$ $\frac{0\widehat{x}}{1} - \frac{\widehat{y}}{1}$
3	$\frac{G}{5}$ $-\frac{\widehat{x}}{\sqrt{5}} + \frac{2\widehat{y}}{\sqrt{5}}$	$\frac{G}{2}$ $-\frac{\widehat{x}}{\sqrt{2}} + \frac{\widehat{y}}{\sqrt{2}}$	$\frac{G}{1}$ $-\frac{\widehat{x}}{1} + \frac{0\widehat{y}}{1}$	$\frac{G}{2}$ $-\frac{\widehat{x}}{\sqrt{2}} - \frac{\widehat{y}}{\sqrt{2}}$

Matrix $\widehat{S}$

Now in the above matrix, if we increase the magnitude of mass for the object at(2,2), say by ‘m’, the force at each cell will just increase by a factor m and the remaining directions will be the same, i.e. no change in direction.
If we have a 2D universe, lets say 2x2 and we have one celestial object at each cell(i,j), and each value represents the mass of the object. If we convolute the above calculated matrix on each cell, we can find the contribution of each object in every other cell, increased by a factor of the mass of the object and the directions. Place the cell(2,2) on every cell in the below matrix.

Let's look at one scenario. When cell $\widehat{S}$ (2,2) from above matrix is placed on S(0,0). The contribution of this object’s force on remaining of the 3 cells (0,1), (1,0) and (1,1) will be $(\frac{G}{1}, \frac{0\widehat{x}}{1} - \frac{\widehat{y}}{1})$, $(\frac{G}{1}, -\frac{\widehat{x}}{1} + \frac{0\widehat{y}}{1})$ and $(\frac{G}{2}, -\frac{\widehat{x}}{\sqrt{2}} - \frac{\widehat{y}}{\sqrt{2}})$. Since the mass of the object at (0,0) in below matrix is ‘1’, the force does not increase. Lets call this force contributed by object (0,0) as $F_{00}(i,j)$ at each of the remaining cells

Where $F_{00}(0,1) = (\frac{G}{1}, \frac{0\widehat{x}}{1} - \frac{\widehat{y}}{1})$

$F_{00}(1,0) = (\frac{G}{1}, -\frac{\widehat{x}}{1} + \frac{0\widehat{y}}{1})$

$F_{00}(1,1) = (\frac{G}{2}, -\frac{\widehat{x}}{\sqrt{2}} - \frac{\widehat{y}}{\sqrt{2}})$

Index	0	1
0	1	2
1	3	4

Matrix S

So the resultant force matrix looks something like this

Index	0	1
0		$(\frac{G}{1}, \frac{0\widehat{x}}{1} - \frac{\widehat{y}}{1})$
1	$(\frac{G}{1}, -\frac{\widehat{x}}{1} + \frac{0\widehat{y}}{1})$	$(\frac{G}{2}, -\frac{\widehat{x}}{\sqrt{2}} - \frac{\widehat{y}}{\sqrt{2}})$

Now place the cell (2,2) of matrix $\widehat{S}$ an (0,1) cell of S. So the force contributed by this object at (0,1) on every other cell (0,0), (1,0) and (1,1) is

$(\frac{G}{1}, \frac{0\widehat{x}}{1} + \frac{\widehat{y}}{1})$, $(\frac{G}{2}, -\frac{\widehat{x}}{\sqrt{2}} + \frac{\widehat{y}}{\sqrt{2}})$, $(\frac{G}{1}, -\frac{\widehat{x}}{1} + \frac{0\widehat{y}}{1})$, and each magnitude of force multiplied by the mass of the object at (0,1) which is 2, say these forces are $F_{01}(i,j)$
Now the force matrix looks as shown below.

Similarly if we continue imposing matrix $\widehat{S}$'s (2,2) on each element of S, we will find the force exerted by each object on every other object.
So for any matrix or 2D universe in this case, to calculate force exerted on each element of matrix S which is a nxn matrix, we need to compute 2n x 2n unit-mass force matrix $\widehat{S}$ and do 2D convolution with original matrix. Finally add the forces in each cell and we get the force matrix.

Index	0	1
0	$F_{01} = (\frac{2*G}{1}, \frac{0\widehat{x}}{1} + \frac{\widehat{y}}{1})$	$F_{00} = (\frac{G}{1}, \frac{0\widehat{x}}{1} - \frac{\widehat{y}}{1})$
1	$F_{00} = (\frac{G}{1}, -\frac{\widehat{x}}{1} + \frac{0\widehat{y}}{1})$ + $F_{01} = (\frac{2*G}{2}, -\frac{\widehat{x}}{\sqrt{2}} + \frac{\widehat{y}}{\sqrt{2}})$	$F_{00} = (\frac{G}{2}, -\frac{\widehat{x}}{\sqrt{2}} - \frac{\widehat{y}}{\sqrt{2}})$ + $F_{01} = (\frac{2*G}{1}, -\frac{\widehat{x}}{1} + \frac{0\widehat{y}}{1})$

This would take again $O(n^4)$ time complexity, as imposing element (2,2) in the above case, on every element of S matrix takes n^2 time, since n^2 objects are there. And in each convolution step, we perform n^2 operations of finding the the forces on every other object. So $n^2 * n^2 = O(n^4)$.

For simplicity we can construct 2 matrices from $\widehat{S}$, one for x-component and y-component and convolute them separately, in order to avoid adding x and y component every time, i.e. basically the direction vector.
We can calculate x-components separately and y-components separately and combine the 2 matrices together at the end. This is simple matrix addition and can be done in $O(n^2)$ time. From now on we will calculate x-component separately and y-component separately for simplicity and understanding purposes

Since we reduced this problem, into a convolution problem, it is simply multivariate polynomial multiplication of elements of matrix S and matrix $\widehat{S}$. We can convert this bivariate polynomial into a univariate. The maximum degree of polynomial constructed from $\widehat{S}$ will be (2n*2n) as there are 2n * 2n elements in $\widehat{S}$. So multiplying 2 polynomials of degree $4n^2$ can be done using Fast Fourier Transform(FFT) which takes $4n^2\log(4n^2)$ time which $O(n^2\log(n))$ time.

We will show a simple example where convolution and polynomial multiplication gives the same result for 2x2 matrix. We will use the same matrix S.
Construct a bivariate polynomial S(x,y) from the matrix S as follows. Powers of x and y are the values of (i,j), in which cell they belong.

$$S(x,y) = x^0y^0 + 2x^0y^1 + 3x^1y^0 + 4x^1y^1 = 1 + 2y + 3x + 4xy$$

Computing $(S(x,y))^2$. The resultant polynomial is as follows

$$S(x,y)S(x,y) = (1 + 2y + 3x + 4xy)(1 + 2y + 3x + 4xy)$$
$$=$$
$$S(x,y)^2 = 1 + 4y + 6x + 20xy + 4y^2 + 9x^2 + 16xy^2 + 24x^2y + 16x^2y^2$$
$$=$$
$$x^0y^0 + 4x^0y^1 + 6x^1y^0 + 20x^1y^1 + 4x^0y^2 + 9x^2y^0 + 16x^1y^2 + 24x^2y^1 + 16x^2y^2$$

We will consider only the terms till degree n, i.e degree 1 in bivariate form and ignore the others, so we consider only ($x^0y^0 + 4x^0y^1 + 6x^1y^0 + 20x^1y^1$)
So the resultant matrix becomes

Index	0	1
0	1	4
1	6	20

We do not need to consider other higher order terms as we are not interested in terms which are outside the matrix. In this question also, we just want to find forces acting on each cell's object. We do not want to consider any forces that are acting outside the 2D grid.
The convolution of the matrix S with itself will now simple be similar to univariate polynomial convolution as explained in class, therefore it will yield the same result.

We will construct 2 polynomials S(x,y) and $\widehat{S}$ (x,y)
(for $\widehat{S}$ (x,y) we are considering only one component at a time. So Fij will be force in either x-direction or y-direction depending on for which directing we are calculating. We can do the same for other direction also. It will just be 2 times the time complexity obtained for computing in 1 direction.)

$$S(x,y) = \sum_{i=0}^{2n}\sum_{j=0}^{2n} a_{x,y}x^iy^j \text{ where } a_{x,y} = 0, \text{ if } i,j > n$$

$= m_{ij}$, otherwise

$$\widehat{S}(x,y) = \sum_{i=0}^{2n}\sum_{j=0}^{2n} a_{x,y}x^i y^j \text{ where } a_{x,y} = F_{ij}$$

The two polynomials are bivariate polynomials, Multiplying both and the coefficients of the resultant bivariate polynomial, gives us the force acting on each object/cell as described above for 2x2 matrix and ignoring higher order terms.

As mentioned above if we convert them to univariate polynomial and do polynomial multiplication using FFT, and if we can infer the coefficients of resultant univariate polynomial to a bivariate polynomial resultant coefficients, then we get the forces acting on each object, and this can be achieved using FFT multiplication which will take $O(n^2\log(n))$ time

Lets see how to convert bivariate to univariate and infer the coefficients in less than $O(n^2\log(n))$.
We are trying to convert multiplication of 2 bivariate polynomials of degree n to multiplication of 2 univariate polynomials of degree bound $O(n^2)$.

Let 2 bivariate polynomials be

$$f(x,y) = \sum_{i,j}^{n} f_{ij}x^i y^j \text{ and } g(x,y) = \sum_{i,j}^{n} g_{ij}x^i y^j$$

$$f(x,y)g(x,y) = \sum_{i,j,i',j'=0}^{n} f_{ij}g_{i'j'}x^{i+i'}y^{j+j'}$$

Let construct a univariate polynomial for f(x,y), call it f(z)

$$f(z) = \sum_{i,j=0}^{n} f_{ij}z^{Ni+j} \text{ and } g(z) = \sum_{i,j=0}^{n} g_{ij}z^{Ni+j} \quad (\text{for some } N)$$

$$h(z) = f(z)g(z) = \sum_{i} h_i z^i$$

The maximum degree of f(z) and g(z) will be (Nn + n) = (N+1)n. So we can compute f(z)g(z) in O((N+1)nlog((N+1)n)) ~ O(Nnlog(Nn))
We can show that for N > 2n, we can infer the coefficients of f(x,y)g(x,y) from f(z)g(z)

Lets choose N = 2n+1
From the above h(z) equation, we can get the following:

$$h(z) = f(z)g(z) = \sum_{i,j=0}^{n} f_{ij}x^{Ni+j} \sum_{i',j'=0}^{n} g_{ij}x^{Ni'+j'} = \sum_{i} h_i z^i$$

We can show that

$$f(x,y)g(x,y) = \sum_{i,j=0}^{2n} h_{Ni+j} x^i y^j$$

We will show this using an example.
Let take a bivariate polynomial $p(x,y) = 1 + 2y + 3x + 4xy$. This polynomial is of degree 2 in each variable.
Multiplying $p(x,y) * p(x,y)$ gives us the following:

$$p(x,y)^2 = 1 + 4y + 6x + 20xy + 4y^2 + 9x^2 + 16xy^2 + 24x^2y + 16x^2y^2$$

Now we'll contruct a univariate polynomial as described above:
Lets choose N = 2n + 1 = 2*2 +1 = 5 and construct $p(z) = \sum_{i,j=0}^{n} p_{ij} z^{Ni+j}$

$$p(z) = 1 + 2z + 3z^5 + 4z^6$$

$$p(z)^2 = 1 + 4z + 6z^5 + 20z^6 + 4z^2 + 9z^{10} + 16z^7 + 24z^{11} + 16z^{12}$$

We can see the $p(z)^2$ and $p(x,y)^2$ are almost similar.
We can convert $p(z)^2$ to $p(x,y)^2$, by simply applying inverse for each for (x,y) degree.
For eg: to get coefficient of x^0y^0 , calculate degree of z which is equal to Ni + j = N*0 + 0 = 0
So in the above $p(z)^2$ coefficient of z^0 will be coefficient of x^0y^0
Similarly let's calculate for x^1y^1 , degree of z will be N*1 + 1 = 5*1 +1 = 6
So coefficient of z^6 will be coefficient of , which is 20.
Same way we can construct all coefficients and convert back to the bivariate polynomial.

Converting from bivariate to univariate would take only O(n^2) time as the maximum degree will be O(n^2) only. In the above example for bivariate degree 2, the univariate degree was 12. Since N = 2n+1 will suffice,The maximum degree of univariate polynomial can be (N+1)n = (2n+2)n = O(n^2)

In our case, we do not need all coefficients, we need only n^2 coefficient for our force calculation, as higher order terms from resultant bivariate polynomial are not required. So to get only the required n^2 terms, it takes O(n^2) time.

So the Pseudo algorithm is as follows for the above explained steps:

```
ConvertToUnivariate( S, n )
   1. Initialize f(x,y) = 0
```

```
  2. Construct a bivariate polynomial from the each cell of S
  3. for i <- 0..n
  4.    for j <- 0..n
  5.       f(x,y) += s[i][j]$x^i y^j$
  6. N <- 2n+1
  7. Scan f(x,y) and construct p = $\sum_{i,j=0}^{n} f_{ij} z^{Ni+j}$
  8. Return p

GetCoefficients( S(z), n )
  1. coefficients[1..n,1..n]
  2. N = 2 * n + 1
  3. for i <- 0..n
  4.    for j <- 0..n
  5.       degree_z = N * i + j
  6.       coefficients[i,j] <- s[ Degree_z ]
  7. return coefficients

Combine( X, Y, n )
  1. force = [1..n,1..n]
  2. for i <- 0..n
  3.    for j <- 0..n
  4.       force[i,j] = X[i,j]$\hat{x}$ + Y[j,j]$\hat{y}$
  5. return force

FindForce( S, n )
  1.   S' = [1..2n,1..2n] // all initialized to 1
  2.   $\hat{S}$ = ForceExertedByObject(S',n, n)
  3.   Split $\hat{S}$ into $\hat{S}_i$ and $\hat{S}_j$    // split into x & y component
  4.                                // matrice
  5.   $S_i(z)$ = ConvertToUnivariate( $\hat{S}_i$,2n )
  6.   $S_j(z)$ = ConvertToUnivariate( $\hat{S}_j$,2n )
  7.   S(z) = ConvertToUnivariate( S,2n )
  8.   $S_x(z)$ = FFT($S_i(z)$, S(z), 2n)
  9.   $S_y(z)$ = FFT($S_j(z)$, S(z), 2n)
  10.  $force_x$ = GetCoefficients($S_x(z)$, n)
  11.  $force_y$ = GetCoefficients($S_y(z)$, n)
  12.  force = Combine( $force_x$, $force_y$, n)
  13.  return force
```

Analysing the running time

Functions `ConvertToUnivariate, GetCoefficients and Combine` takes O(n^2) time each.
Function `FindForce` takes
Line 2 - takes O(n^2)
Line 3 - takes O(n^2)
Line 5-7 takes 3 * O($(2n)^2$)
Line 8-9 takes 2 * O($(2n)^2\log((2n)^2)$)
Line 10-11 takes 2 * O(n^2)
Line 12 takes O(n^2)
Total time taken is O($n^2\log(n)$) as this is the dominating term from all the steps.

Task3:
(a) [40 Points] Give an algorithm that can compute A × B in o($n^{2.41}$) time when k = 2. You must clearly prove that the time complexity of your algorithm is, indeed, o($n^{2.41}$).

Answer:

As B is a rotator matrix we see that there is a symmetry across the diagonal and hence it can be divided into two matrices: upper triangular and lower triangular matrix. For example, see below : -

$$\begin{array}{|c|c|c|}\hline a1 & a2 & a3 \\ \hline a2 & a3 & a1 \\ \hline a3 & a1 & a2 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline a1 & 0 & 0 \\ \hline a2 & a3 & 0 \\ \hline a3 & a1 & a2 \\ \hline \end{array} + \begin{array}{|c|c|c|}\hline 0 & a2 & a3 \\ \hline 0 & 0 & a1 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

Hence for this matrix B we have two triangular matrices: upper matrix U and lower matrix L: -

B=L+U

Suppose we have another matrix A: -

[p1 p2 p3

q1 q2 q3

r1 r2 r3]

If we multiply A*B to get matrix C, we can do it by: -

C=A*B

C=A*(L+U)

C=A*L + A*U

Hence, it is sum of two matrices. Let us go further into the first multiplication term A*L:-

For the first row of A*L the terms will be: -

$$\begin{bmatrix} p1*a1+p2*a2+p3*a3 & p2*a3+p3*a1 & p3*a2 \\ \vdots & \ddots & \vdots \\ & & \cdots \end{bmatrix}$$

Hence any row here can also be seen as the convolutions of the sequences {p1 , p2 , p3} and {a1, a2, a3}. Refer to the chart below:-

```
p1  p2  p3
a1  a2  a3

p1  p2  p3
    a1  a2

p1  p2  p3
        a1
```

So to obtain the first row of the resultant matrix, it is sufficient to convolute (p1,p2,p3) with (a1, a2, a3). We learnt that convolution is similar to multiplication of 2 polynomials, which take nlogn time for size n vector. Now, we have to find all rows of the resultant matrix. In the above example, if we convolute (q1,q2,q3) with (a1,a2,a3), we get the 2nd row and so on. Each convolution takes nlogn time, there are n rows to be calculated. So, the total time taken for multiplying normal matrix with rotator matrix is n * nlogn = O(n^2logn).

In the question, we are given that, the rows may not be in circular manner (the shift is not always the same) as we took in our example. Still. it would not change the time complexity, because we are each convolution gives us n numbers(elements of the row), we just have to place those number accordingly to the shift in the original matrix, which should not be more than O(n) time for a given row. So the time complexity remains O(n^2logn). Lets call this P(n) = O(n^2logn)

For any matrix A of size n*n when it is multiplied with another matrix of size n*n which has √n * √n rotator submatrices, the resultant matrix C which will also be of size √n * √n submatrices. In our question, we have k=2, hence for any matrix of size n*n, there will be √n * √n rotator submatrices,

Therefore total number of multiplication will be $(\sqrt{n})^3$. For example, please see the example below: -

$$\begin{array}{|c|c|}\hline a & b \\ \hline c & d \\ \hline \end{array} \quad * \quad \begin{array}{|c|c|}\hline e & f \\ \hline g & h \\ \hline \end{array} \quad = \quad \begin{array}{|c|c|}\hline ae+bg & af+bh \\ \hline ce+dg & cf+dh \\ \hline \end{array}$$

Here for two matrix of order 2 x 2 which has 4 elements we need 8 multiplication which is ($(\sqrt{4})^3$ which is basically $n^{3/2}$ multiplications). Similarly for matrix of size 4 we will need 64 multiplications.

Since there are n(√n * √n) submatrices now in our case, we can treat each matrix as an element. So combining these matrices together is similar to multiplication of 2 matrixes of (√n * √n). The number of multiplications would be $n^{3/2}$

Now total time complexity of the original problem will be

$$T(n) \ = \ n^{3/2} * p(\sqrt{n}) + \theta(n^2)$$

where p is cost of multiplying a rotator matrix as defined above

For multiplication of one rotator matrix of size m*m the complexity will be $m^2 log\ m$.

So, $p(m) = \ m^2\ log\ m$.

Substituting this to T(n): -

$T(n) \;=\; (n^{3/2} * ((\sqrt{n})^2 log\ \sqrt{n}) + \theta(n^2)$

$=> \; T(n) \;=\; 1/2\ (n^{5/2} * (log\ n)) + \theta(n^2)$

We calculated above considering, that there will be 8 sub-multiplication problems for every matrix multiplication, but, as we have seen in Strassen's algorithm multiplication we can do it in 7 matrix multiplication itself. So, we can further optimize using Strassen's algorithm.

$T(n) \;=\; ((\sqrt{n})^{log7} * n \;* log\ \sqrt{n}) + \theta(n^2)$

$=> \; T(n) \;=\; ((\sqrt{n})^{2.81} * n \;* log\ \sqrt{n}) + \theta(n^2)$

$=> \; T(n) \;=\; ((n)^{2.81/2} * n \;* log\ \sqrt{n}) + \theta(n^2)$

$=> \; T(n) \;=\; 1/2\ ((n)^{1.405+1} log\ n) + \theta(n^2)$

$=> \; T(n) \;=\; 1/2\ ((n)^{2.405} log\ n)$

As $((n)^{2.405} log\ n) <\ \theta(n^2)$ hence,

$=> \; \theta\ (n) \;=\; ((n)^{2.405} log\ n)$

This is always less than $o\ (n)^{2.41}$. Hence proved.

Note: We can compute similarly for upper triangular matrix, and add the resultant 2 matrices. Since adding 2 matrices is O(n^2). The above complexity dominates

3 (b) What if k $\neq$ 2? Extend your algorithm and derive its time complexity for general (positive integral values of) k.

Answer:

Here, as we are generalizing k, as mentioned in the question there are $(n/n^{1/k}) * (n/n^{1/k})$ matrices of size $(n^{1/k}) * (n^{1/k})$. So, as we have seen in part a of the question cost of one rotator matrix multiplication is $p(m) = \; m^2\ log\ m$ Hence we can write the total complex for generalized k as: -

$T(n) = (n^2/n^{2/k})^{3/2} * p(n^{1/k}) + \theta(n^2)$

$=> \; T(n) = (n^{(2-2/k)*(3/2)} \;* p(n^{1/k}) + \theta(n^2)$

Substituting value of cost function p(n):-

$=> \ T(n) = (n^{(2-2/k)*(3/2)} \ * ((n^{1/k})^2 log(n^{1/k})) + \theta(n^2)$

$=> \ T(n) = (n^{(3-3/k)} \ * ((n^{2/k}) \ log(n^{1/k})) + \theta(n^2)$

$=> \ T(n) = 1/k \ (n^{(3-3/k)+2/k} \ * log(n)) + \theta(n^2)$

$=> \ T(n) = 1/k \ (n^{(3-1/k)} \ * \ log(n)) + \theta(n^2)$

As $(n^{(3-1/k)} \ * \ log(n) \ > \theta(n^2)$, for any integer k, hence we can write the complexity as: -

$=> \ \theta(n) = \ (n^{(3-1/k)} \ * \ log(n))$

We calculated above considering, that there will be 8 sub-multiplication problems for every matrix multiplication, but, as we have seen in Strassen's algorithm multiplication we can do it in 7 sub-matrix multiplication itself. So, we can further optimize using Strassen's algorithm.

$T(n) \ = \ ((n/n^{1/k} \ )^{log7} * (n^{1/k})^2 \ * log \ (n^{1/k})) + \theta(n^2)$

$=> \ T(n) \ = \ ((n^{1 \, - \, 1/k})^{2.81} * (n^{1/k})^2 \ * log \ (n^{1/k})) + \theta(n^2)$

$=> \ T(n) \ = \ ((n^{2.81 \, - \, 2.81/k \, + \, 2/k} \ * log \ (n^{1/k})) + \theta(n^2)$

$=> \ T(n) \ = \ 1/k \ ((n^{2.81 \, - \, 0.81/k} \ * log \ (n)) + \theta(n^2)$

As $((n^{2.81 \, - \, 0.81/k} \ * log \ (n)) > \theta(n^2)$ hence,

$=> \ \theta \ (n) \ = \ ((n^{2.81 \, - \, 0.81/k} \ * log \ (n))$