0951339dgodhia@cs.stonybrook.edu
0974349rumshah@cs.stonybrook.edu

Analysis Of Algorithms Assignment 1

Darshan Godhia

Rushabh M. Shah

1. Task 1: Laser Game

1.1. Given:

- A game configuration of a $n \times n$ square board, where $n = 2^k$ for some integer $k \geq 0$, with three types of game pieces - generators, receptors and deflector.
- The generators are of 5 types - red (r), green (g), blue (b), yellow (y), and violet (v), and direct laser beams horizontally towards the right, worth some points $\in [1, 100]$.
  During any given game, n generators will be used.
- The receptors get activated when a laser beam of a specific color (i.e. r/g/b/y/v) hits its sensor. Each receptor is also worth some points $\in [1, 100]$.
  During any given game, n receptors will be used.
- There is one beam deflector which moves through the grid, which mirrors the horizontally incident laser beam vertically upwards. Thus it is used to direct beams from the generators, placed on the left to hit the receptors placed on the top of the column.
- At each step of the play the deflector must be moved to a neighboring cell either to the left or above or diagonally to the top-left of the current cell.
- The deflector is initially placed inside the cell at the bottom-right corner of the grid and the game ends when the deflector reached the top-left corner cell of the grid.
- As soon as a receptor becomes activated, both the receptor and the activating generator are removed from the board and the player collects all the points associated with those two game pieces.

1.2. Problem:

Given an initial state of the board with all the generators and receptors placed and choses, the goal is to move the deflector from the bottom-right corner cell to the top-left corner cell - a deflector path, in a way that maximizes the total number of points he can collect.

1.3. Explain how in $\Theta(n^2)$ time and space, you can find the best reflector path.

Solution:
Dynamic programming gives a solution to the given problem in $\Theta(n^2)$ runtime and space complexity. We can apply dynamic programming to this problem because it has both - optimal substructure and overlapping subproblems.
Let $Points[i, j]$ be the number of points that can be obtained on cell in the i^{th} row, j^{th} column.
So, we have the following relationship to compute $Points([i, j])$,

$$Points[i,j] = \begin{cases} generator(i).points + receptor(j).points, & \text{if } generator(i).color == receptor(j).color \\ 0, & otherwise \end{cases}$$

Let $Value[i, j]$ be the optimal number of points that can be obtained at destination (i, j) starting from the bottom right cell.

We can arrive at any point (i, j) from three possible directions:
1. Cell to the right of (i, j), i.e. from $(i, j + 1)$.
In this case, we can either activate (i, j) OR $(i, j+1)$, since they share a common generator.
2. Cell to the bottom of (i, j), i.e. from $(i + 1, j)$.
In this case, we can either activate (i, j) OR $(i+1, j)$, since they share a common receptor.
3. Cell to the bottom right of (i, j), diagonally, i.e. from $(i + 1, j + 1)$.
In this case, we can activate both (i, j) and $(i+1, j+1)$, as they have mutually exclusive generators as well as receptors.

To obtain a dynamic programming solution, we need to optimize this subproblem. We determine the optimal movement to arrive at any point (i, j).
We calculate optimal $Value[i, j]$ for all (i, j) using the following relationship.

$$Value[i,j] = max \begin{cases} Value[i, j+1], \\ Value[i+1, j], \\ Points(i,j) + Value[i+1, j+1] \end{cases} \tag{1}$$

The solution to the *reflector path problem* is simply $Value[0, 0]$.
To find the solution:

- iterate over $i \leftarrow n \ to \ 1$
    - iterate over $j \leftarrow n \ to \ 1$
        - compute and store $Value[i, j]$

After calculating $Value[0, 0]$, we can print the actual path followed, by backtracking from $(0, 0)$ to (n, n), by matching $Value[i, j]$ with one of $Value[i, j+1]$, $Value[i + 1, j]$ and
$Points(i, j) + Value[i + 1, j + 1]$).

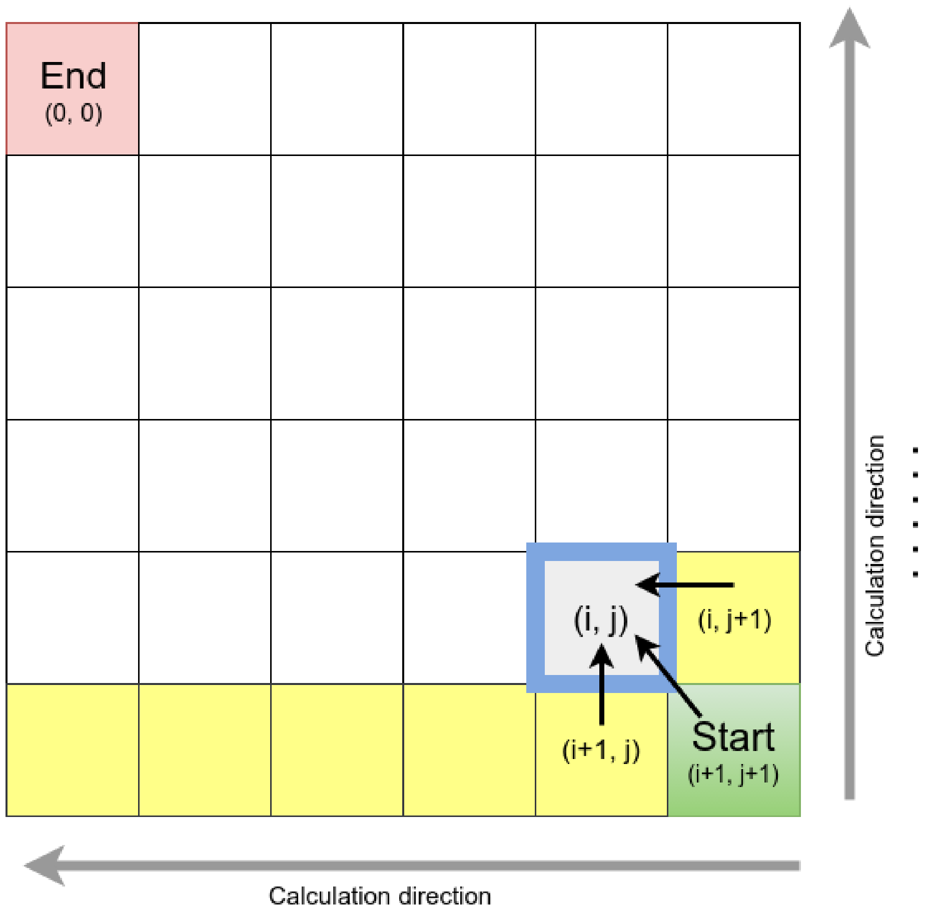


Figure 1. Possibilites to reach (i, j)

Analysis:
The $Value[i,j]$ computation loops over $n \times n$ items, computing and storing a total of $n \times n$.
Thus it takes $\Theta(n^2)$ time and space.

The printing step can be done in $\Theta(n)$.

In this way, we solve the problem in $\Theta(n^2)$ time and space complexity.

1.4. Explain how can you generalize the algorithm from previous part to use $\Theta(n^{1+\epsilon})$ space and run in time $\Theta(n^{3-\epsilon})$ for any given $\epsilon \in [0,1]$.

Observe that when $\epsilon = 0$, this algorithm uses $\Theta(n)$ space, but runs in $\Theta(n^3)$ time.

In the algorithm used in the previous approach, we see that we are using a 2D matrix to store the maximum points that can be obtained at any point.
But, on close observation we find out that we can use lesser space at the cost of more computing time. This can be better understood by considering the following boundary case:
Let's say we are restricting the space to be used to $\mathcal{O}n$.
In this case we can maintain two separate n sized arrays to store the maximum points that can be obtained at any given row and column.
i.e.,
R[i] = Max points that can be obtained in row i until now
C[j] = Max points that can be obtained in column j until now

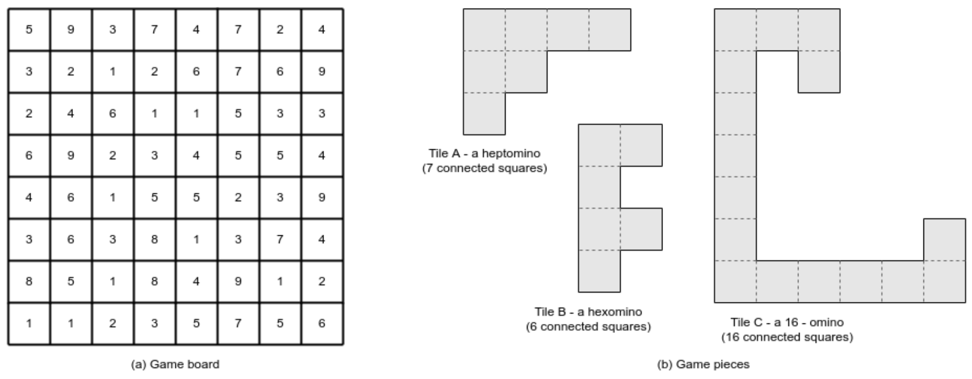
Both the arrays are of size n and are initialized to 0.

1.5. Design a recursive divide-and-conquer algorithm to find the best deflector path in $\Theta(n^2)$ time and $\Theta(n)$ space. Find and solve the recurrences for running time and space usage.

To get a solution in $\Theta(n^2)$ time and $\Theta(n)$ space complexity, we build upon the solution in section 1.3.
There, we saved the entire $n \times n$ matrix for $Value[i,j]$, however, a careful observation of the referencing of $Value[i,j]$ in computation shows that we only ever use two consecutive rows of the $Value[i,j]$ matrix.

We make use of this fact to reduce the space complexity of the algorithm, to a linear asymptotic bound.

The runtime recurrence of this solution stays the same as section 1.3.
We apply divide and conquer to calculate $Value[i,j]$ as



5	9	3	7	4	7	2	4
3	2	1	2	6	7	6	9
2	4	6	1	1	5	3	3
6	9	2	3	4	5	5	4
4	6	1	5	5	2	3	9
3	6	3	8	1	3	7	4
8	5	1	8	4	9	1	2
1	1	2	3	5	7	5	6

Figure 2. Task 2: A game of Polyominos

$$Value[i,j] = max \begin{cases} Value[i, j+1], \\ Value[i+1, j], \\ Points(i,j) + Value[i+1, j+1]; \end{cases}$$

The above relation

2. Task 2: A Game of Polyominos

2.1. Given:

- A $n \times n$ square board, where $n = 2^k$ for some integer $k \geq 0$.
- A game piece. Each game piece is a geometric figure formed by putting one or more square cells edge to edge, where each cell is exactly equal to a cell of the board. Each such polyomino will be composed of $m \in [1, n^2]$ square cells, and must also completely fit inside the game board.
- Each square of the game board has a number written on it which is the number of points a player wins if she can occupy that cell using one of her game pieces.
- The newly placed polyomino must not cover an already occupied cell.
- The entire polyomino does not need to be placed completely inside the board, it must cover at least one board cell.
- All four orientations of a polyomino that keep its sides parallel to the sides of the board are legal.

2.2. Problem:

Find a location on the board to legally place and orient the game piece, such that the total number of points in all cells covered by it is maximized.

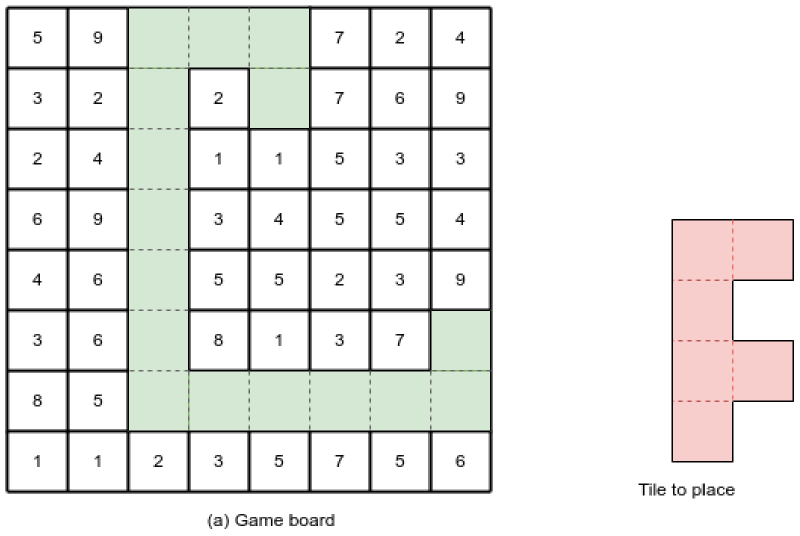


Figure 3. Starting state of the placement problem

2.3. Show that the problem can be trivially solved in $\mathcal{O}(n^4)$.

2.3.1. Solution:

- This problem is similar to the *protein docking* problem.
- The trivial solution therefore is as follows:
- For all possible translational placement locations:

  1. For all possible orientation possibilities:

     (a) Place the tile one the board in all possible locations, with all possible orientations.
     (b) Check whether the placement is legal.
     (c) Calculate the points acquired in the current setting.

- Solution is the configuration with the maximum number of points acquired.
- **Algorithm:**

```
for each rotation of piece_grid:
  for each row in game_board:
      for each column in game_board:
        place piece_grid
```

- Starting from the left-lowermost corner cell, we start placing the tile in each cell and translate it, one cell at

- At each translation we also perform a constant number of rotations of the tile.

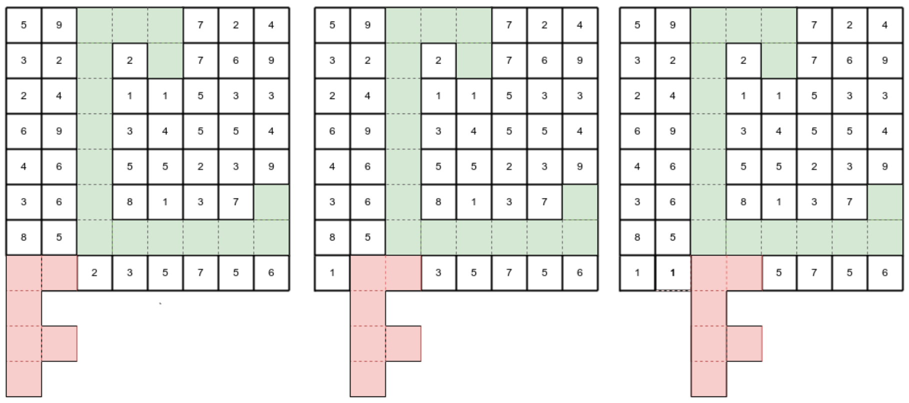


Figure 4. Translating and fitting the tile linearly

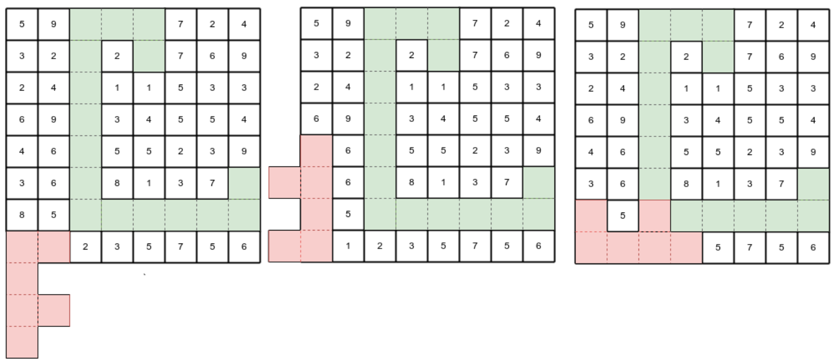


Figure 5. Rotation at every translation

- After each placement we need to check whether the tile overlaps with any already placed game piece, as it would be illegal. We also need to calculate the points obtained from the placement. In the naive approach, we simply visit every cell of the tile, find if there is any overlap, and if not, add up the points obtained due to the cell in the current placement.

- **Completeness and Subproblems:**

  - Translations.
    Since every input tile can completely fit on the board, it's highest dimensionality is $n \times n$.
    Thus, to get every possible translational possibility in placement, we need to translate the tile in $n + n - 1$ positions horizontally, and $n + n - 1$ positions vertically.
    So, in the worst case, the total number of translational combinations to check are:
    $$\begin{aligned} Translate(n) &= \mathcal{O}(n + n - 1 * n + n - 1) \\ &= \mathcal{O}(2n - 1 * 2n - 1) \\ &= \mathcal{O}(4n^2 - 4n - 1) \\ &= \mathcal{O}(4n^2) \\ &= \mathcal{O}(n^2) \end{aligned}$$
  - Rotations.
    Translational positioning of the tile is not enough. We also need to check for all the rotational possibilities. This can be done by checking for all the $four$ rotations of the tile at every translational placement. Thus, the complexity introduced by rotation itself is:
    $$\begin{aligned} Rotate(n) &= \Theta(4) \\ &= \Theta(1) \end{aligned}$$
  - Check legality of placement and point calculations:
    At every $Translational \times Rotational$ placement of the tile, we need to check whether the tile placement is legal. Thus we need to go over each and every cell of the tile, and check whether it overlaps any non-empty cell. To calculate the points obtained by the placement, we need to calculate the sum of all the empty cells covered by the placed tile. Both these operations can be done together in a single pass over all the cells of the tile. Given that, the length of the tile in the range of $[1, 2, \ldots, n^2]$, so in the worst case, we need to check all the n^2 cells of the tile. So the complexity of this step is:
    $$AddDetect(n) = \mathcal{O}(n^2)$$

- **Analysis:**
  The runtime of the *Naive Approach* can be represented as:

$$
\begin{aligned}
T(n) &= Translate(n) * Rotate(n) * AddDetect(n) \\
Translate(n) &= \mathcal{O}(n^2) \\
Rotate(n) &= \Theta(1) \\
AddDetect(n) &= \mathcal{O}(n^2) \\
T(n) &= \mathcal{O}(n^2) * \Theta(1) * \mathcal{O}(n^2) \\
&= \mathcal{O}(n^4)
\end{aligned}
$$

$$T(n) = \mathcal{O}(n^4) \tag{2}$$

2.4. Show that the problem can be solved in $\mathcal{O}(n^2 \log n)$.

2.4.1. Solution:

As commented earlier, the given problem is very similar to that of *Protein Docking*, which itself was a version of *polynomial multiplication*. The similarity arises from the fact that, both the solutions essentially depend upon *Rotational and Translational* search. The *polynomial multiplication* problem has a solution in $n\log(n)$ time in the form of convolution, using *Fast Fourier Transformation*. The given problem, can be written as *polynomial multiplication*, where the values in the board form the first polynomial, and the tile cell configuration forms another polynomial.

- **Analysis:**
  - Polynomial 1: The game board has a dimensionality of $n \times n$. Therefore, we need $n \times n$ elements to completely represent it in the polynomial form.
    Length of polynomial 1: $\Theta(n \times n) = \Theta(n^2)$
  - Polynomial 2: The tile has a length in the range of $[1, 2, \ldots, n^2]$. So, the length of the second polynomial is bound by this length.
    Length of polynomial 2: $\mathcal{O}(n^2)$
  - Polynomial multiplication of two polynomials of length N can be done in runtime complexity of $N \log N$, using *Fast Fourier Tranformation*.
  - Here, we have two polynomials of length $\mathcal{O}(n^2)$, and we have to find their multiplication.
    Using *FFT* technique, we can do them in:

$$
\begin{aligned}
T(N) &= \Theta(N \log(N)) \\
N &= \mathcal{O}(n^2) \\
T(N) &= \mathcal{O}(n^2 \log(n^2)) \\
\log(n^2) &= 2\log(n) \\
T(N) &= \mathcal{O}(2n^2 \log(n)) \\
&= \mathcal{O}(n^2 \log n)
\end{aligned}
$$

$$T(n) = \mathcal{O}(n^2 \log n) \tag{3}$$

3. Multiplying Fractal Matrices

Given:

$\Delta - Fractal$: A $cell(i, j)$ will definitely contain a zero provided the bitwise AND of i and j is non-zero

3.1. Sketch Fractal Matrices

3.1.1. Fractal Matrix of size 2x2:

Here X represents any non zero number

$$\Delta_{2x2} = \begin{bmatrix} X & X \\ X & \end{bmatrix}$$

3.1.2. Fractal Matrix of size 4x4:

$$\Delta_{4x4} = \begin{bmatrix} X & X & X & X \\ X & & X & \\ X & X & & \\ X & & & \end{bmatrix}$$

3.1.3. Fractal Matrix of Size 8x8

$$\Delta_{8x8} = \begin{bmatrix} X & X & X & X & X & X & X & X \\ X & & X & & X & & X & \\ X & X & & & X & X & & \\ X & & & & X & & & \\ X & X & X & X & & & & \\ X & & X & & & & & \\ X & X & & & & & & \\ X & & & & & & & \end{bmatrix}$$

3.1.4. Fractal Matrix of Size 16x16

$$\Delta_{16x16} = \begin{bmatrix}
X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X \\
X & & X & & X & & X & & X & & X & & X & & X & \\
X & X & & & X & X & & & X & X & & & X & X & & \\
X & & & & X & & & & X & & & & X & & & \\
X & X & X & X & & & & & X & X & X & X & & & & \\
X & & X & & & & & & X & & X & & & & & \\
X & X & & & & & & & X & X & & & & & & \\
X & & & & & & & & X & & & & & & & \\
X & X & X & X & X & X & X & X & & & & & & & & \\
X & & X & & X & & X & & & & & & & & & \\
X & X & & & X & X & & & & & & & & & & \\
X & & & & X & & & & & & & & & & & \\
X & X & X & X & & & & & & & & & & & & \\
X & & X & & & & & & & & & & & & & \\
X & X & & & & & & & & & & & & & & \\
X & & & & & & & & & & & & & & &
\end{bmatrix}$$

Inverse Fractals

3.1.5. Inverse Fractal of Size 2x2

$$\nabla_{2*2} = \begin{bmatrix} & X \\ X & X \end{bmatrix}$$

3.1.6. Inverse Fractal of Size 4x4

$$\nabla_{4*4} = \begin{bmatrix} & & & X \\ & & X & X \\ & X & & X \\ X & X & X & X \end{bmatrix}$$

3.1.7. Inverse Fractal of Size 8x8

$$\nabla_{8*8} = \begin{bmatrix}
 & & & & & & & X \\
 & & & & & & X & X \\
 & & & & & X & & X \\
 & & & & X & X & X & X \\
 & & & X & & & & X \\
 & & X & X & & & X & X \\
 & X & & X & & X & & X \\
X & X & X & X & X & X & X & X
\end{bmatrix}$$

3.1.8. Inverse Fractal of Size 16x16

$$\nabla_{16*16} = \begin{bmatrix}
 & & & & & & & & & & & & & & & X \\
 & & & & & & & & & & & & & & X & X \\
 & & & & & & & & & & & & & X & & X \\
 & & & & & & & & & & & & X & X & X & X \\
 & & & & & & & & & & & X & & & & X \\
 & & & & & & & & & & X & X & & & X & X \\
 & & & & & & & & & X & & X & & X & & X \\
 & & & & & & & & X & X & X & X & X & X & X & X \\
 & & & & & & & X & & & & & & & & X \\
 & & & & & & X & X & & & & & & & X & X \\
 & & & & & X & & X & & & & & & X & & X \\
 & & & & X & X & X & X & & & & & X & X & X & X \\
 & & & X & & & & X & & & & X & & & & X \\
 & & X & X & & & X & X & & & X & X & & & X & X \\
 & X & & X & & X & & X & & X & & X & & X & & X \\
X & X & X & X & X & X & X & X & X & X & X & X & X & X & X & X
\end{bmatrix}$$

3.2. Running time of $n \times n$ Δ fractal matrix and a standard matrix Y

The standard recursive matrix multiplication has 8 matrix multiplications and 4 matrix sums (using *Divide and Conquer*).
The running time of standard recursive multiplication is given by $\Theta(n^3)$
This follows from its recurrence relation given by:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 8T(\frac{n}{2}) + \Theta(n^2), & otherwise \end{cases}$$

But in case of Δ-Fractals, we observe that the lower left element (for a 2×2 matrix) or the lower left matrix (for all $k \times k$ matrices, where $0 < 2^k < n$)

Thus we see that 8 multiplications are reduced to 6 multiplications since two of them are $zero'd$ out.
The recurrence relation can therefore be rewritten as -

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 6T(\frac{n}{2}) + \Theta(n^2), & otherwise \end{cases}$$

Solving this by Masters Theorem we get -

$$\begin{aligned} T(n) &= \Theta(n^{log_b a}) + \Theta(n^2) \\ &= \Theta(n^{log_2 6}) \qquad \ldots \{Here, a = 6, b = 2\} \\ &= \Theta(n^{2.585}) \end{aligned}$$

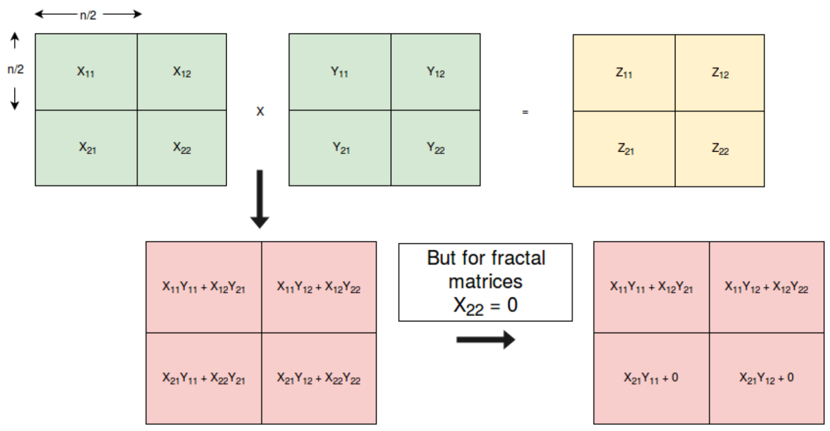


Figure 6. Recursive Matrix Multiplication

3.3. How will you modify the layout of the matrices so that the standard iterative algorithm can compute the product in part (b) within the same asymptotic time bound as the standard recursive algorithm

The standard iterative approach simply parses through all the elements given in the data structure and calculates the value of element at each position.
The algorithm for that is as follows:

```
for i from 1 to n
  for j from 1 to n
    result[i][j] = 0
    for k from 1 to n
      result[i][j] += arr[i][k] + arr[k][j]
```

This has a runtime of $\Theta(n^3)$. Thus in order to obtain the same time complexity as of the recursive approach, we need a data structure such that we can easily skip the multiplications of the elements which had zero values.

This gives the intuition of a data structure that only stores the non - zero values of a given matrix - **Sparse Matrix Representation**

Consider a simple representation of the sparse matrix - list of lists :

Let the Δ matrix be

$$\begin{bmatrix} X_{11} & X_{12} \\ X_{21} & 0 \end{bmatrix}$$

Consider the Sparse Matrix Representation of X and Y. Each element of a sparse matrix is represented as a (key, value) pair.
For X the key is the column number of the element whereas for Y the key is the row index.

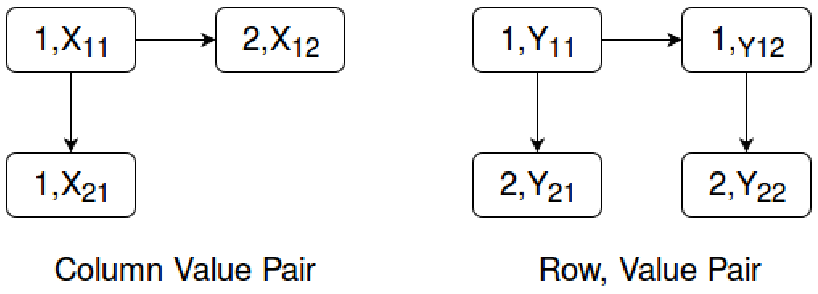


Figure 7. Sparse Matrix Representation of Δ Fractal Matrix and Standard Matrix

```
while row in in X:
  X_row = row
  while column in Y:
    Y_col = column
    result[key1][key2] = 0

    while X_row.right!= null && Y_column.down != null:
      if(X_row.key == Y_column.key):
        result[X_row][Y_col] += X_row.value * Y_col.value
        X.row = X.row.right
        Y.col = Y.col.down
      else:
        if(X_row.key < Y_col.key):
          X_row = X_row.right
        else:
          Y.col = Y.col.down
    column = coumnl.right
row = row.down
```

Using this representation we will skip the elements guaranteed to be zero and hence the number of multiplications will be same as that of the recursive algorithm, i.e for each nxn matrix multiplication we will have at most 5 multiplications.

Hence the time complexity can be found by solving the same recurrence relation as that of the Recursive divide and conquer.

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a}) + \Theta(n^2) \\ &= \Theta(n^{\log_2 6}) + \Theta(n^2) \\ &= \mathcal{O}(n^{2.585}) \end{aligned}$$

3.4. Running time of Strassen Multiplication for the same

In Strassen Algorithm the following computations are used:..

Sums

$X_{r1} = X_{11} + X_{12}$
$X_{r2} = X_{21} + X_{22}$
$X_{c1} = X_{11} - X_{21}$
$X_{c2} = X_{12} - X_{22}$
$X_{d1} = X_{11} + X_{22}$
$Y_{r1} = Y_{11} + Y_{12}$
$Y_{r2} = Y_{21} + Y_{22}$
$Y_{c1} = Y_{11} - Y_{21}$
$Y_{c2} = Y_{12} - Y_{22}$
$Y_{d1} = Y_{11} + Y_{22}$

Products

$P_{11} = X_{11} * Y_{c2}$
$P_{22} = X_{22} * Y_{c1}$
$P_{r1} = X_{r1} * Y_{22}$
$P_{r2} = X_{r2} * Y_{11}$
$P_{c1} = X_{c1} * Y_{r1}$
$P_{c2} = X_{c2} * Y_{r2}$
$P_{d1} = X_{d1} * Y_{d1}$

But for Δ Fractal Matrices the value of X_{22} is always going to be zero. Substituting this value we get:

Sums

$X_{r1} = X_{11} + X_{12}$
$X_{r2} = X_{21}$
$X_{c1} = X_{11} - X_{21}$
$X_{c2} = X_{12}$
$X_{d1} = X_{11}$
$Y_{r1} = Y_{11} + Y_{12}$
$Y_{r2} = Y_{21} + Y_{22}$
$Y_{c1} = Y_{11} - Y_{21}$
$Y_{c2} = Y_{12} - Y_{22}$
$Y_{d1} = Y_{11} + Y_{22}$

Products

$P_{11} = X_{11} * Y_{c2}$
$P_{22} = 0$
$P_{r1} = X_{r1} * Y_{22}$

$P_{r2} = X_{r2} * Y_{11}$
$P_{c1} = X_{c1} * Y_{r1}$
$P_{c2} = X_{c2} * Y_{r2}$
$P_{d1} = X_{d1} * Y_{d1}$

As we can see the number of additions in Strassen decrease by 3, but this would not affect the overall time complexity as that factor is still bounded by **add term here**

The change in the time complexity will however occur due to the decrease in the number of multiplications by 1 ($P_{22} = 0$)
Thus the recurrence relation is:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 6T(\frac{n}{2}) + \Theta(n^2), & otherwise \end{cases}$$

Solving this by Masters Theorem we get -

$$\begin{aligned} T(n) &= \Theta(n^{log_b a}) + \Theta(n^2) \\ &= \Theta(n^{log_2 6}) \\ &= \mathcal{O}(n^{2.585}) \end{aligned}$$

3.5. Repeat parts (b) - (d) if Y is also a $n \times n$ Δ Fractal Matrix

By extending the rules established for $MatrixX$, we know that if Y is also a Δ-Fractal matrix then $Y_{22} = 0$
Using this we can find the following:

1. Time Complexity of Standard Recursive Algorithm for Computing XY
   Figure 6 in B part shows the recursive divide and conquer computations that are required in order to multiply the matrices.

   Since both X and Y are Δ-Fractal Matrices, we can zero out the X_{22} and Y_{22} terms. Thus we are now left with only 5 multiplications for each $N \times N$ matrix multiplication.

   The recurrence relation would be -

   $$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 5T(\frac{n}{2}) + \Theta(n^2), & otherwise \end{cases}$$

   Solving this by Masters Theorem we get -

$$\begin{aligned} T(n) &= \Theta(n^{log_b a}) + \Theta(n^2) \\ &= \Theta(n^{log_2 5}) \\ &= \mathcal{O}(n^{2.3219}) \end{aligned}$$

2. Layout modification so that standard iterative works in the same time as of the recursive implementation

   Similar to the case of matrix multiplication when only X was given to be Δ - Fractal Matrix, we can represent both the matrices using Sparse Matrix Representation.

   Thus X and Y can both be represented as Figure 7

   On applying Sparse Matrix Multiplication Algorithm, the time complexity of the multiplication turns out to $\Theta(n^{2.3219})$ as the number of multiplications will at most be 5.

3. Strassen Matrix Multiplication Algorithm

   For Strassen Algorithm, in addition to P_{22}, in this case P_{r1} will also zero out since now $Y_{22} = 0$
   Thus the number of multiplications reduces to 5. The time complexity is :

   $$\begin{aligned} T(n) &= \Theta(n^{log_b a}) + \Theta(n^2) \\ &= \Theta(n^{log_2 5}) \\ &= \mathcal{O}(n^{2.3219}) \end{aligned}$$

3.6. Repeat parts (b) - (d) assuming Y to be an n x n inverse fractal

Firstly, note that the inverse fractal is just like the Δ-Fractal Matrix the only difference being the X_{11} is always 0 instead of X_{22}.

Consider a Δ-Fractal Matrix X and a Inverse Fractal Matrix Y
For the following parts we can therefore use $X_{22} = Y_{11} = 0$

1. Standard Recursive Algorithm
   Substituting $X_{22} = Y_{11} = 0$ in Figure 6, the matrix reduces to

   Thus the number of multiplications reduces to 4. The recurrence relation is

   $$\begin{aligned} T(n) &= \Theta(n^{log_b a}) + \Theta(n^2) \\ &= \Theta(n^{log_2 4}) + \Theta(n^2) \\ &= \mathcal{O}(n^2 log(n)) \end{aligned}$$

2. Layout modification so that standard iterative works in the same time as of the recursive implementation. Using sparse multiplication again, we will avoid the multiplication of zero elements and hence the number of multiplications is equal to the pairs of non zero elements.

The number of multiplications is hence at most 4 per $n \times n$ matrix multiplication.

Time Complexity:

$$
\begin{aligned}
T(n) &= \Theta(n^{log_b a}) + \Theta(n^2) \\
&= \Theta(n^{log_2 4}) + \Theta(n^2) \\
&= \mathcal{O}(n^2 log(n))
\end{aligned}
$$

3. Strassen Matrix Multiplication Algorithm

   If X_{22} and Y_{11} are both zero, we can eliminate two of the product equations from Strassen Equations:
   $P_{22} = 0$
   $P_{r2} = 0$
   No other term gets eliminated in the Strassen Products and hence we still need to do 5 multiplications. Thus the time complexity can be found out as follows:

$$
\begin{aligned}
T(n) &= \Theta(n^{log_b a}) + \Theta(n^2) \\
&= \Theta(n^{log_2 5}) \\
&= \mathcal{O}(n^{2.3219})
\end{aligned}
$$

   Here we see that Strassen will in fact perform worse than the standard recursion algorithm.

Created with Madoko.net.