Elnor Jonuzi

Selected Algorithms Problems

This document presents three carefully chosen homework problems from my Fall 2024 CSE 373: Analysis of Algorithms course. I selected these problems for their intriguing nature and the thoughtful effort required to solve them. I challenge the reader to solve each problem on their own before reading my solution.

Problem 1: Huddling Penguins

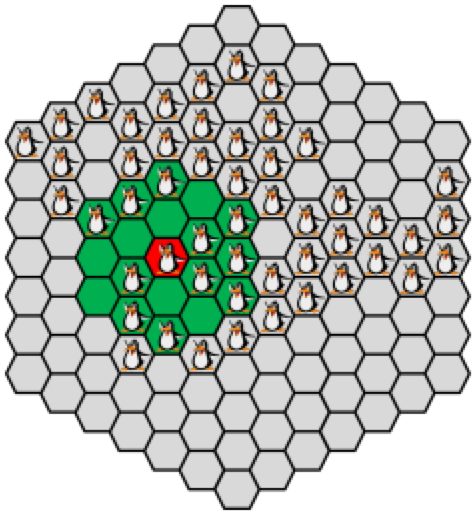
Emperor penguins are known to form dense huddles to survive the brutal cold of Antarctica. It has also been observed that each penguin in a huddle usually positions itself at the center of a regular hexagon in a nearly hexagonal packing of the group. Each hexagonal cell is occupied by at most one penguin.

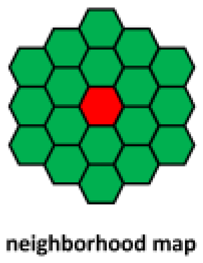
To study the spatial distribution of penguins in a huddle and how that changes over time it is useful to know the number of penguins in the neighborhood of each cell (including the penguin in that cell, if any) of the hexagonal grid. We will call that number the neighbor coefficient of that cell. A neighborhood map specifies the cells in the neighborhood of every cell.

A base-n (n ≥ 1) hexagonal grid is composed of regular hexagonal cells such that each of the six sides of the grid contains exactly n hexagonal cells. Every two adjacent cells share an edge. A base-n hexagonal grid has exactly $3n(n-1)+1$ cells and $3(3n-2)(n-1)$ edges. The distance between two cells is the minimum number of edges one must cross in order to go from one of the cells to the other. The figure below shows a base-7 hexagonal grid and a neighborhood map consisting of all cells (green) within distance 2 of the center cell (red). The neighbor coefficient of the red cell in the hexagonal grid is exactly 12 (= 11 penguins in the green cells + 1 penguin in the red cell).

Now, given a base-n hexagonal grid G_n containing at most one penguin in each cell and a neighborhood map consisting of all cells within distance $r \in [0, 2n-2]$ of the center cell (see figure below), give a $\Theta(n^2)$ dynamic programming algorithm for counting the neighbor coefficient of all $3n(n-1)+1$ cells in G_n. Observe that in this case, the neighborhood map itself is a base-(r+ 1) hexagonal grid, and r may not be independent of n.

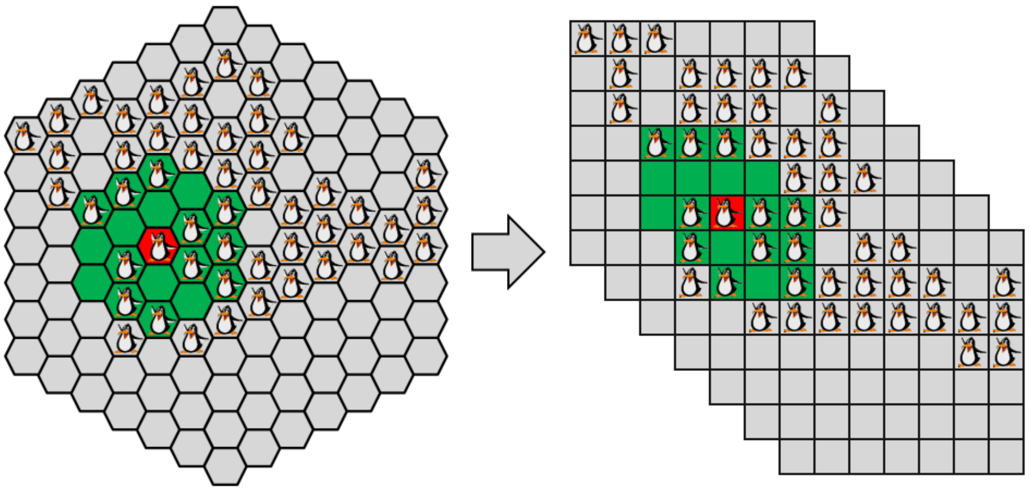
Figure: A base-7 hexagonal grid and a neighborhood map consisting of all cells (green) within distance 2 of the center cell (red). As an example, the neighborhood map is centered on one of the cells of the grid to count its neighbor coefficient which turns out to be 12.





My Solution to Problem 1

Given a base-*n* hexagonal grid *G* and a distance *r* representing the radius of a neighborhood map, here is an algorithm using dynamic programming to calculate the neighbor coefficient of all *3n(n-1)+1* cells in *G*.
The first step of the algorithm is to convert the hexagonal grid into a rectangular grid by shifting the ith column from the left down by i * 0.5 cells. For example:



From here we will process it into three matrices simultaneously: *M, X,* and *Y*. *M* is a $2n - 1$ by $2n - 1$ matrix where each cell (i, j) is 1 if there is a penguin present in the corresponding cell, or 0 otherwise. *X* has the same dimensions as *M*, except each cell (i, j) is the sum of all the cells in matrix *M* corresponding to the rectangular region defined by $\{(0, 0), (i, 0), (j, 0), (i, j)\}$. *Y* has the same dimensions as *X*, except each cell (i, j) is the sum of all the cells in matrix *M* corresponding to the triangular region defined by $\{(i, j), (0, i), (0, i - j)\}$.

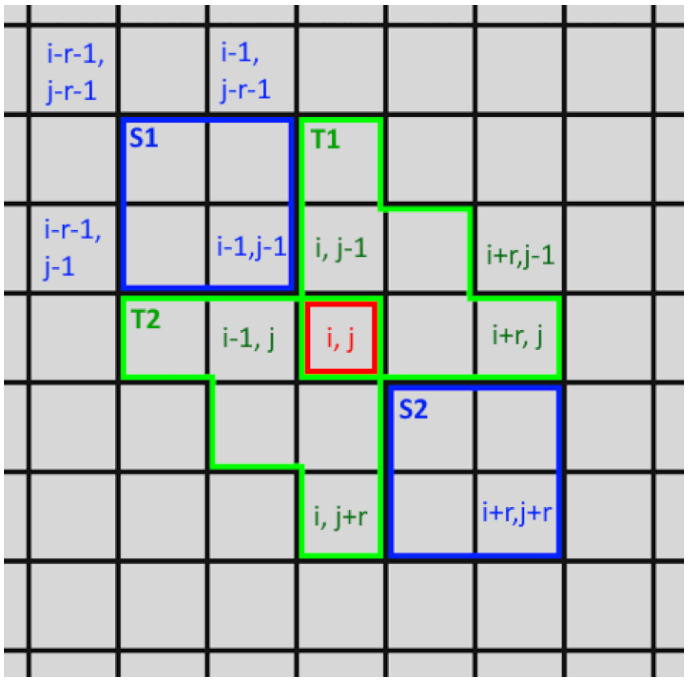
Example of obtaining *X*:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & \textcircled{1} & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 1 & 2 & 3 & 3 \\ 1 & 3 & 5 & 6 & 6 \\ 2 & 5 & 7 & \textcircled{9} & 10 \\ 2 & 6 & 9 & 12 & 14 \\ 3 & 8 & 12 & 16 & 19 \end{bmatrix}$$

Example of obtaining *Y*:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & \textcircled{1} & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 4 & 5 & 3 \\ 2 & 6 & 7 & 7 & 7 \\ 3 & 7 & 10 & \textcircled{11} & 10 \end{bmatrix}$$

In order to calculate the coefficient of
each cell, it helps to divide the neighborhood map into 4 shapes: 2 squares of *r* side length, and 2 right triangles of *r+1* side length. An example is shown where r = 2.



Now we need to loop through all $3n(n - 1) + 1$ cells, and calculate the neighbor coefficient for every cell (i, j) in *M* using matrices *X* and *Y* in the following equations:

$S1 = X[i - 1][j - 1] - X[i - r - 1][j - 1] - X[i - 1][j - r - 1] + X[i - r - 1][j - r - 1]$

$S2 = X[i + r][j + r] - X[i][j + r] - X[i + r][j] + X[i][j]$

$T1 = X[i + r][j] - X[i - 1][j] - Y[i + r][j - 1] + Y[i - 1][j - r - 1]$

$T2 = Y[i][j + r] - Y[i][j + r] - X[i][j - 1] + X[i - r - 1][j - 1]$

$R[i][j] = S1 + S2 + T1 + T2 - M[i][j]$

These equations leverage the overlapping regions of cumulative values to efficiently compute the neighbor coefficients in O(1) time. Indices are checked to ensure they are within bounds; if an index is out of bounds, the value of the attempted cell is treated as *0*. This approach was chosen over padding the matrices to reduce the time complexity of constructing the three matrices initially. The neighbor coefficients are stored in a resultant matrix *R*, which is then converted back into a base-*n* hexagon by shifting the *i*-th column from the left upwards by *0.5*i* cells. The function then returns the resultant hexagonal grid.

Final algorithm pseudocode:

```
countCoeffs(G, n, r):
     // Convert the hexagonal grid to a rectangular grid
     S = convertToRectangularGrid(G)
     M = new int[2n-1][2n-1]
```

```
X = new int[2n-1][2n-1]
Y = new int[2n-1][2n-1]

// Compute matrices M, X, Y. This step is O(n^2).
FOR col FROM 0 TO 2 * n - 1 DO:
     rowLower = col < n ? 0 : (col%n)+1
     rowUpper = col < n ? n+(col%n) : 2*n-1
     FOR row FROM rowLower TO rowUpper DO:
          M[row][col] = S[row][col] has penguin ? 1 : 0
          X[row][col] = get(M, row, col)
          + get(X, row, col-1)
          + get(X, row-1, col)
          - get(X, row-1, col-1)
          Y[row][col] = get(M, row, col)
          + get(X, row-1, col)
          + get(X, row-1, col-1)
          - get(X, row-2, col-1)

R = new int[2n-1][2n-1] // Create the resultant matrix

// Iterate through the 3n(n-1)+1 cells to calculate neighbor
coefficients. This step is O(n^2).
FOR col FROM 0 TO 2 * n - 1 DO:
     rowLower = col < n ? 0 : (col%n)+1
     rowUpper = col < n ? n+(col%n) : 2*n-1
     FOR row FROM rowLower TO rowUpper DO:
          // Calculate neighbor coefficient and store in
          result. This step is O(1)
          S1 = get(X, row-1, col-1)
          - get(X, row-r-1, col-1)
          - get(X, row-1, col-r-1)
          + get(X, row-r-1, col-r-1)
          S2 = get(X, row+r, col+r)
          - get(X, row, col+r)
          - get(X, row+r, col)
          + get(X, row, col)
          T1 = get(X, row+r, col)
          - get(X, row-1, col)
          - get(Y, row+r, col-1)
          + get(Y, row-1, col-r-1)
          T2 = get(Y, row, col+r)
          - get(Y, row, col+r)
          - get(X, row, col-1)
          + get(X, row-r-1, col-1)
          R[row][col] = S1 + S2 + T1 + T2 - M[row][col]
```

```
      // Convert the resultant matrix back to a hexagonal grid. This
      step is O(n^2).
      R_G = convertToHex(R)

      return R_G

// helper pseudo function for safe index handling
get (M, row, col):
      if 0 <= row < M.length and 0 <= col < M[row].length then
            return M[row][col]
      else return 0
```

Efficiency analysis:
The time complexity of obtaining matrices *M, X,* and *Y* is O(n^2) as every cell must be processed and there are 3n(n-1)+1 cells which is $3n^2$-3n+1=O(n^2). The time complexity of obtaining the resultant matrix *R* is O(n^2) as every cell's neighbor coefficient must be obtained, and the time complexity to calculate a cell's neighbor coefficient is O(1). Because every cell must be processed once in the algorithm, the upper and lower bound is O(n^2). Since the upper bound is equal to lower bound this implies the run time complexity is Θ(n^2).

Problem 2: SSSP sells MSTs at a TSP during BFS

SSSP[1] is a place where people go to buy apple[2]. During the BFS[3] this year, they will sell an MST[4] at a TSP[5]. SSSP has set such an APSP[6] for the MSTs that no one can refuse to buy them.

SSSP has many locations in New York, and all locations will offer the deal. Suppose that there are n households $h_1, h_2, \ldots, h_n$ that want to buy the MSTs, and there are m SSSP locations $s_1, s_2, \ldots, s_m$ in New York. For $1 \leq i \leq n$ and $1 \leq j \leq m$, it takes $t_{i,j}$ minutes for the people from h_i to reach s_j if they start at 6am on the day after Thanksgiving. Once they reach their intended s_j they will join the queue waiting outside the store. No one will be allowed to join the queue after the store opens. One household will be served every minute immediately after the opening of the store. So, if people from k households end up waiting in the queue, it will take exactly k minutes for all of them to get their MSTs.

The CS[7] department at SSSP has collected all $t_{i,j}$ values from all n households ahead of time. They have hired you to design an algorithm that given all $t_{i,j}$ values can determine which household (h_i) should go to which store location (s_j) and at what time the stores must open so that the last household to be served can be served as early as possible (measuring time from 6am on the day after Thanksgiving). They require that all stores must open at exactly the same time and every household must be served at some (and exactly one) location. The stores will remain open until all n households are served.

---

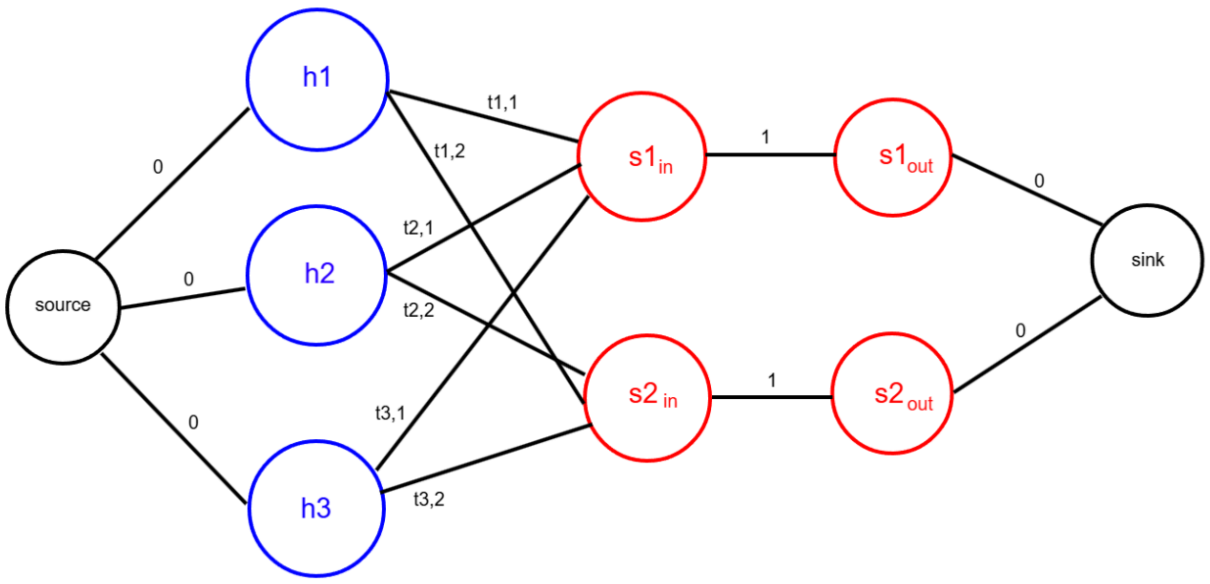
[1]SSSP = Super Store for Superior Products
[2]apple = astonishingly precious products at low expense
[3]BFS = Black Friday Sales
[4]MST = Mind-bogglingly Superior Thing
[5]TSP = Tantalizingly Slashed Price
[6]APSP = Amazingly Puny Sale Price
[7]CS = Customer Service

My Solution to Problem 2

The algorithm I designed to find the optimal matching for houses and stores and the optimal opening time for the stores uses network flow, where each edge has a cost and capacity. The cost is the amount of time it takes for 1 unit to pass through, and the capacity is 1 unit (all edges have a capacity of 1, but their time costs vary).

First we need a source, which supplies all houses with 1 unit, each of which takes 0 time cost to do. Then each house h_i needs to be connected to all stores s_i with an edge that has cost $t_{i,j}$ and capacity 1. Once a unit has arrived at a store, they need to be processed through the entrance and then released. This is best done using 2 nodes to represent the store, one that can accept a unit from all homes, and another that releases a unit into the sink, with an edge in between with a time cost of 1 and capacity of 1 (since it takes 1 minute to process a house).

The goal of the algorithm is to minimize the sink value which holds the accumulated amount of time passed for a selected matching between houses and stores. We have to iterate through all matchings of houses and stores and pick the one with minimum value by constructing the network and running it. Here is an example of a network constructed for three houses and two stores: (only costs are shown but each edge has a capacity of 1).



Algorithm rundown:

- Construct network: $O(n + m)$

- Iterate through all matchings of houses and stores: $O(n * m)$
    - For each iteration, find the minimum time cost. Using Dijkstra's algorithm this can be done in $O(F * (E + VlogV))$ where:
      $F = total\,flow = n$ (n houses)
      $E = number\,of\,edges = n + n * m + m + m = n + nm + 2m$
      $V = number\,of\,vertices = n + 2 * m$

Once the algorithm finds the minimum value for sink, it uses this value and adds it to 6:00 AM to represent the time at which all stores should open since by that point all queued customers will have been processed. It also returns the set of pairings between houses and stores which yielded this minimum sink value as that represents which store each house should go to.

Problem 3: Unshuffling

Given an array of coefficients $a_0, a_1, a_2, \dots, a_{n-1}$ where $n = 2^k$ for some integer $k \geq 0$. When $k > 0$, we would like to unshuffle the array into two subarrays – one containing the even-numbered coefficients $a_0, a_2, \dots, a_{n-2}$ and the other containing the odd-numbered coefficients $a_1, a_3, \dots, a_{n-1}$. Entries in both subarrays appear in exactly the same order as they appeared in the original array.

a) Given an array $A[0{:}n-1]$ of coefficients $a_0, a_1, a_2, \dots, a_{n-1}$, where $n = 2^k$ for some integer $k > 0$, design an efficient recursive divide-and-conquer algorithm for unshuffling the array using only $O(1)$ extra space. In the final unshuffled array, a_i must be stored in $A[\frac{n}{2}(i \bmod 2) + floor(\frac{i}{2})]$, for $i \in [0, n-1]$. The figure below shows an example. Analyze your algorithms running time.

b) Now consider a square matrix $M[0{:}n-1][0{:}n-1]$, where $n = 2^k$ for some integer $k > 0$, and for $0 \leq l \leq n^2 - 1$, coefficient a_l is stored in $M[floor(\frac{n}{2})][l \bmod n]$. Design an efficient algorithm based on recursive divide and conquer which moves all a_l values with $l \bmod 4 = 0$, $l \bmod 4 = 1$, $l \bmod 4 = 2$, and $l \bmod 4 = 3$ to the top-left, top-right, bottom-left, and bottom-right quadrant, respectively. All coefficients in each quadrant must appear in exactly the same order (row by row) as they appeared in the original input matrix. More specifically, a_l must be stored in

$$M[\tfrac{n}{2}(floor(\tfrac{l}{4}) \bmod 2) + floor(floor(\tfrac{l}{4})/\tfrac{n}{2})][\tfrac{n}{2}(l \bmod 2) + (floor(\tfrac{l}{4}) \bmod \tfrac{n}{2})]$$

in the final unshuffled matrix. The figure below shows an example. Your algorithm must use $O(1)$ extra space. Analyze your algorithms running time.

Figure: Unshuffling array $A[0{:}15]$.

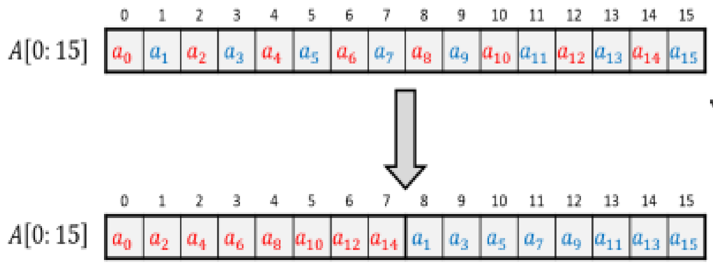
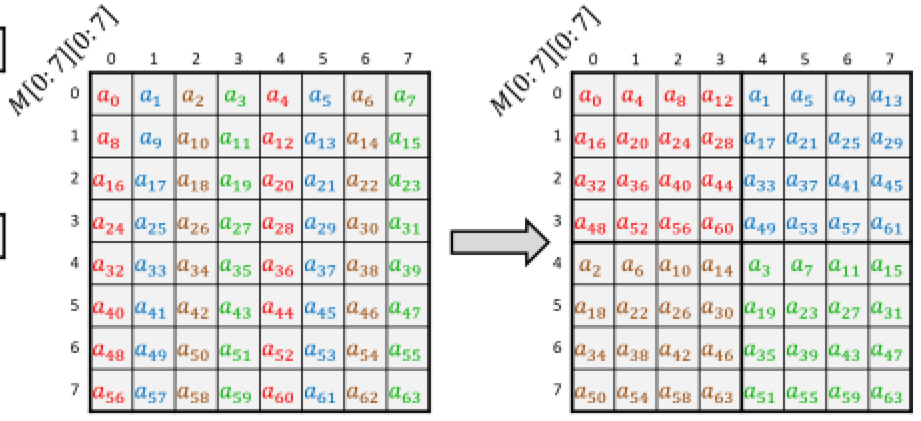

A[0: 15]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15
A[0: 15]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a0 a2 a4 a6 a8 a10 a12 a14 a1 a3 a5 a7 a9 a11 a13 a15

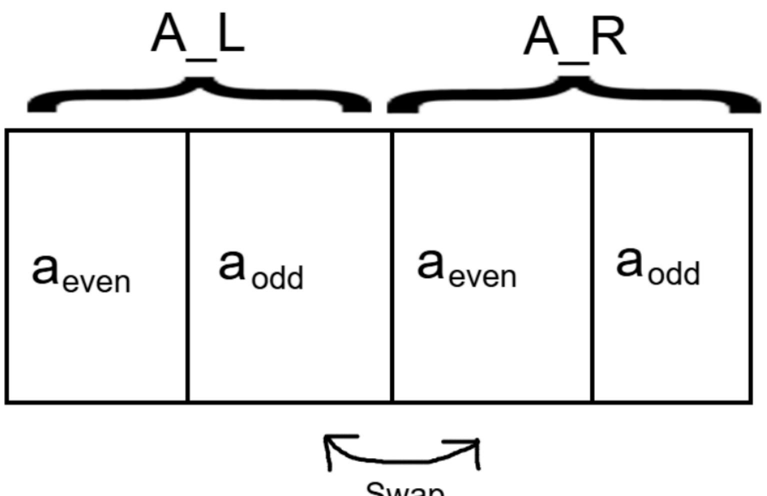
Figure: Unshuffling matrix $M[0{:}7][0{:}7]$.


M[0:7][0:7]
0 1 2 3 4 5 6 7
a0 a1 a2 a3 a4 a5 a6 a7
a8 a9 a10 a11 a12 a13 a14 a15
a16 a17 a18 a19 a20 a21 a22 a23
a24 a25 a26 a27 a28 a29 a30 a31
a32 a33 a34 a35 a36 a37 a38 a39
a40 a41 a42 a43 a44 a45 a46 a47
a48 a49 a50 a51 a52 a53 a54 a55
a56 a57 a58 a59 a60 a61 a62 a63
M[0:7][0:7]
0 1 2 3 4 5 6 7
a0 a4 a8 a12 a1 a5 a9 a13
a16 a20 a24 a28 a17 a21 a25 a29
a32 a36 a40 a44 a33 a37 a41 a45
a48 a52 a56 a60 a49 a53 a57 a61
a2 a6 a10 a14 a3 a7 a11 a15
a18 a22 a26 a30 a19 a23 a27 a31
a34 a38 a42 a46 a35 a39 a43 a47
a50 a54 a58 a63 a51 a55 a59 a63

My Solution to Problem 3

a) Unshuffle divide-and-conquer recursive method, Python implementation:
```
def unshuffle(A, n):
      # Base case: don't unshuffle if cant divide
      # Time complexity: O(1)
      if n <= 2: return

      # Divide into two halves (left and right)
      # Time complexity: O(1)
      h = n//2 # Calculate halve length
      A_L = A[:h] # Obtain left half of A
      A_R = A[h:] # Obtain right half of A
      unshuffle(A_L, h) # Unshuffle left half
      unshuffle(A_R, h) # Unshuffle right half

      # Merge the two unshuffled halves
      # Time complexity: O(n)
      for j in range(n//4):
            A_L[n//4 + j], A_R[j] = A_R[j], A_L[n//4 + j]

      # Combine the halves back into original array
      # Time complexity: O(1)
      A[:h], A[h:] = A_L, A_R
```

Merge step explanation:
The merge step uses a for loop to swap the elements from the right half of the left half (which after calling unshuffle on the left half should contain only odd index elements) with the elements of the left half of the right half (which after calling unshuffle on the right half should contain only even index elements). This is to move all even index elements to the left half of A, which is the desired end result. Diagram demonstration:



To analyze run time complexity we can use the master theorem:

- a = 2 since there are 2 recursive calls
- b = 2 since each problem size is n/2
- d = 1 since the merge step is O(n)=O(n^1)

Since a = b^d (2 = 2^1) this means T(n)=O(n^dlogn)
Thus, the run-time complexity of this algorithm is O(nlogn)

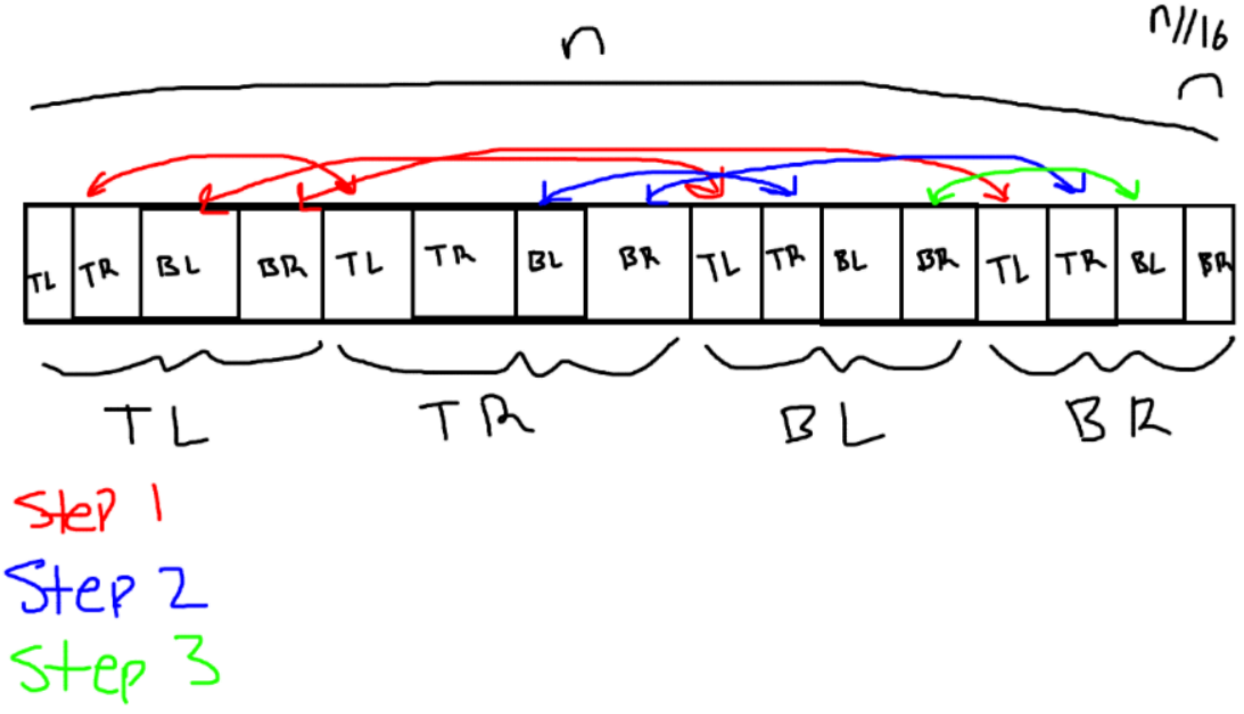
b) Unshuffle divide-and-conquer recursive method, Python implementation:

```
def unshuffle(M, n):
     # Flatten the matrix into a 1-dimensional array
     # Time complexity: O(n^2)
     A = flatten_matrix(M)


     # Unshuffle the 1-dimensional array
     # Time complexity: O(nlogn)
     unshuffle_helper(A, n*n)

     # Reconstruct the array into a matrix
     # Time complexity: O(n^2)
     return reconstruct_matrix(A, n)

def unshuffle_helper(A, n):
     # Base case: don't unshuffle if cant divide
     # Time complexity: O(1)
     if n <= 4: return

     # Divide into four quadrants (A_TL,A_TR,A_BL,A_BR)
     # Time complexity: O(1)
     q = n//4 # Calculate quarter length
     A_TL = A[:q] # Obtain first quarter of A
     A_TR = A[q:2*q] # Obtain second quarter of A
     A_BL = A[2*q:3*q] # Obtain third quarter of A
     A_BR = A[3*q:] # Obtain fourth quarter of A
     unshuffle_helper(A_TL, q)
     unshuffle_helper(A_TR, q)
     unshuffle_helper(A_BL, q)
     unshuffle_helper(A_BR, q)

     # Merge the four unshuffled quarters
     # Time complexity: O(n)
```

```
    x = n//16 # num elements per quadrants quadrant
    for j in range(x): # Merge step 1, part 1
        A_TL[x+j], A_TR[j] = A_TR[j], A_TL[x+j]
    for j in range(x): # Merge step 1, part 2
        A_TL[2*x+j], A_BL[j] = A_BL[j], A_TL[2*x+j]
    for j in range(x): # Merge step 1, part 3
        A_TL[3*x+j], A_BR[j] = A_BR[j], A_TL[3*x+j]
    for j in range(x): # Merge step 2, part 1
        A_TR[2*x+j], A_BL[x+j] = A_BL[x+j], A_TR[2*x+j]
    for j in range(x): # Merge step 2, part 2
        A_TR[3*x+j], A_BR[x+j] = A_BR[x+j], A_TR[3*x+j]
    for j in range(x): # Merge step 3
        A_BL[3*x+j], A_BR[2*x+j]= A_BR[2*x+j], A_TR[3*x+j]

    # Combine the quarters back into original array
    # Time complexity: O(1)
    A[:q],A[q:2*q],A[2*q:3*q],A[3*q:] = A_TL,A_TR,A_BL,A_BR

# Flattens n*n matrix into array of n*n length
# Time complexity: O(n^2)
def flatten_matrix(M):
    A = [] # Initialize result array
    for row in M: # Loop through rows
        A.extend(row) # Append row to array
    return A # Return resultant array

# Reconstructs n*n array into matrix with quadrants
# Time complexity: O(n^2)
def reconstruct_matrix(M, n):
    # Initialize resultant matrix
    M = [[0 for _ in range(n)] for _ in range(n)]
    q = (n*n)//4
    A_TL = A[:q] # Obtain first quarter of A
    A_TR = A[q:2*q] # Obtain second quarter of A
    A_BL = A[2*q:3*q] # Obtain third quarter of A
    A_BR = A[3*q:] # Obtain fourth quarter of A
    h = n//2

    for i in range(h): # Fill top left quadrant
      for j in range(h):
          M[i][j] = A_TL[i * h + j]
    for i in range(h): # Fill top right quadrant
      for j in range(h):
          M[i][j + h] = A_TR[i * h + j]
    for i in range(h): # Fill bottom left quadrant
```

```
            for j in range(h):
                M[i + h][j] = A_BL[i * h + j]
        for i in range(h): # Fill bottom right quadrant
            for j in range(h):
                M[i + h][j + h] = A_BR[i * h + j]

        return M
```

Merge step explanation:
After the algorithm divides the array into 4 quarters and unshuffles each one, it needs to move around the pieces that correspond to the top left, top right, bottom left, and bottom right array. There are 16 pieces (since each quarter is divided into 4 quarters once it is unshuffled), and based on the positioning of these pieces the merge step must perform a swapping algorithm 6 times. This is shown in the following diagram:



The run time complexity of this algorithm is O(n^2) since it requires the matrix to be flattened, and this operation can only be done in O(n^2). However, we can still analyze the run time complexity of the recursive function (unshuffle_helper(M, n)) using the master theorem:

- a = 4 since there are 4 recursive calls
- b = 4 since each problem size is n/4
- d = 1 since the merge step is O(n)=O(n^1)

Since $a = b^d$ ($4 = 4^1$) this means $T(n)=O(n^d \log n) = O(n \log n)$ for unshuffle_helper(M, n).