CMPSC 465 Data Structures & Algorithms
Fall 2024 Mingfu Shao and Debarati Das

HW 3

Due September 20th (Friday), 11:59 pm

Formatting: Each problem should begin on a new page. When submitting in Gradescope, try to assign pages to problems from the rubric as much as you can. Make sure you write all your group members' names. For the full policy on assignments, consult the syllabus.

1. (0 pts.) Acknowledgements. List any resources besides the course material that you consulted in order to solve the assignment problems. If you did not consult anything, write "I did not consult any non-class materials." The assignment will receive a 0 if this question is not answered.

2. (15 pts.) Problem 1.

   You are given a 2D matrix of integers, where each row and each column is sorted in non-decreasing order. Your task is to search for a target integer within the matrix using an efficient search strategy. The matrix is defined as follows:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

   where m is the number of rows and n is the number of columns.

   The goal is to find the position of the target integer in the matrix, or determine that it does not exist. You must propose a search strategy that leverages the properties of the matrix to minimize the number of comparisons.

   Input

   - A 2D matrix A of size $m \times n$, where each row and column is sorted in non-decreasing order.
   - An integer value x, the target value to be searched for in the matrix.

   Output

   - If the target is found, return its position in the matrix as a tuple (i, j).
   - If the target is not found, return -1.

   Objectives

   1. (5 pts.) Assume that $m = 1$, i.e., the input matrix A is essentially a sorted array of size n, and the required output is the position j where $A[j] = x$ or -1 is x is not in A. Design an algorithm for this special case and analyze its time complexity. (Hint: consider binary search.)
   2. (10 pts.) Design an algorithm for this problem (the general case where A is of size $m \times n$) and analyze its time complexity. (Hint: consider how you can efficiently divide the search space (the matrix) by focusing on sections where the target is most likely to be, while eliminating areas where the target cannot possibly be located.)

Solution:

Part 1: Special Case ($m = 1$). In this case, the matrix has only one row, which is essentially a sorted array of size n. To find the target value x, we can use binary search.

Algorithm for Part 1: Binary Search for 1D Array

- Initialize left $= 0$, right $= n - 1$
- While left $\leq$ right
  - Set mid $= \left\lfloor \frac{\text{left+right}}{2} \right\rfloor$
  - If A[mid] $= x$, return mid
  - If A[mid] $< x$, set left $=$ mid $+ 1$
  - If A[mid] $> x$, set right $=$ mid $- 1$
- End while
- Return -1

End algorithm

Part 2: The idea is to divide the matrix into four quadrants, recursively eliminate the quadrants where the target cannot be, and search in the quadrant that could contain the target. We define recursive function search_matrix(A, x, top, left, bottom, right), that searches for x in the submatrix of A from (top, left) to (bottom, right), and returns the position where x locates or -1 if x cannot be found in the submatrix.

Algorithm for Part 2: Divide and Conquer for 2D Matrix

- Function search_matrix(A, x, t, l, b, r)
  - If $t > b$ or $l > r$: return -1
  - If $t = b$: scan the row and check if x presents and return accordingly
  - If $l = r$: scan the column and check if x presents and return accordingly
  - Set $i = \left\lfloor \frac{t+b}{2} \right\rfloor$, $j = \left\lfloor \frac{l+r}{2} \right\rfloor$
  - If $A[i][j] = x$, return (i, j)
  - If $A[i][j] < x$
    - P = search_matrix$(A, x, t, j + 1, i, r)$; If $P \neq -1$: return P; # top-right quadrant
    - P = search_matrix$(A, x, i + 1, l, b, j)$; If $P \neq -1$: return P; # bottom-left quadrant
    - P = search_matrix$(A, x, i + 1, j + 1, b, r)$; If $P \neq -1$: return P; # bottom-right
  - End
  - If $A[i][j] > x$
    - P = search_matrix$(A, x, t, l, i - 1, j - 1)$; If $P \neq -1$: return P; # top-left quadrant
    - P = search_matrix$(A, x, i, l, b, j - 1)$; If $P \neq -1$: return P; # bottom-left quadrant
    - P = search_matrix$(A, x, t, j, i - 1, r)$; If $P \neq -1$: return P; # top-right
  - End
  - Return -1
- End function search_matrix
- Call search_matrix$(A, x, 1, 1, m, n)$ to perform the search

End algorithm

Time Complexity Analysis. The divide-and-conquer approach divides the $m \times n$ matrix into four quadrants and recursively searches up to three of these quadrants based on comparisons. The size of each quadrant is $m \times n/4$. Let $T(mn)$ be the time complexity for searching in an $m \times n$ matrix. The recurrence relation is:

$$T(mn) = 3 \cdot T\left(\frac{mn}{4}\right) + \Theta(1)$$

Here you may regard mn as a single variable. Applying the master theorem with $a = 3$, $b = 4$, and $d = 0$, we compute:

$$\log_b a = \log_4 3 \approx 0.79$$

Since $\log_b a > d$, the time complexity is:

$$T(mn) = \Theta\left((mn)^{\log_4 3}\right) = \Theta\left((mn)^{0.79}\right)$$

Solution 2 for part 2

1. Start from the bottom-left corner of the matrix, i.e., $A[n][1]$.
2. If the current matrix number is equal to the target, return the position. If it is smaller than the target, move to the right entry ($A[p][q] \to A[p][q+1]$). If the matrix entry is larger, move to the top entry ($A[p][q] \to A[p-1][q]$).
3. Repeat step 2 until either $p < 1$ (out of matrix bounds) or $q > m$ (out of matrix bounds). If this occurs, the target is not in the matrix. Return -1.

The pseudocode for this solution is as follows:

```
Function search_matrix2(A, target x, n, m):
    p ← n;
    q ← 1;
    while p ≥ 1 and q ≤ m do
        if A[p][q] = x then
            return (p, q);
        else
            if A[p][q] > x then
                p ← p − 1;
            else
                q ← q + 1;
    return -1;
```

Time Complexity Analysis:
Since we are only moving up or right, the worst-case scenario is that we traverse from $[n][1]$ to $[1][n]$. Thus, the time complexity is $O(m + n)$.

3. (15 pts.) Problem 2.

In class, we have learned how to find a pair of points that minimizes the distance between them from a given set of n points on a 2D plane. The brute force solution for this problem runs in $O(n^2)$, and we can improve the solution to $O(n \log n)$ using the divide-and-conquer approach.

Now, however, you are more interested in finding the "smallest triangle". Specifically, you want to find a set of three points from the given N points, where point i has coordinates (x_i, y_i), that forms

a triangle with the smallest perimeter. Let $\langle i, j, k\rangle$ represent the triangle formed by points i, j and k, and $d(i, j, k)$ denote its perimeter. To simplify the problem, you may assume that no two points share the same x-coordinate or y-coordinate, and no three points are collinear.tm

a. (3 pts.) Brute force approach: Write pseudocode for a brute force method. The time complexity of this approach should be $O(n^3)$.

b. (8 pts + 4 bonus pts.) Merging step: We now apply the divide and conquer approach. Suppose we have divided all points into a left half P_L and a right half P_R, and have already identified the "smallest triangle", denoted as $\langle \ell_0^*, \ell_1^*, \ell_2^*\rangle$ in P_L and $\langle r_0^*, r_1^*, r_2^*\rangle$ in P_R. Let d^* be the minimum of $d(\ell_0^*, \ell_1^*, \ell_2^*)$ and $d(r_0^*, r_1^*, r_2^*)$. Let point q be the rightmost point in P_L.

   (a) (2 pts.) Prove that if a triangle with a perimeter smaller than d^* exists, then all three points must lie within the band between the lines $x = x_q - d^*/2$ and $x = x_q + d^*/2$.

   (b) (2 pts.) Prove that, for each point i in the band, if there exist points j and k that can form a smaller triangle with i, then points j and k must satisfy $y_i - d^*/2 \le y_j, y_k \le y_i + d^*/2$.

   (c) (4 bonus pts.) Suppose that we divide the band into $(d^*/2) \times (d^*/2)$ boxes, prove that the number of points in each box is at most a constant c.

   (d) (4 pts.) Describe how you would determine if there are three points in the band that form a triangle with a perimeter smaller than d^*. You can assume that B, the list of points in the band sorted in ascending order by their y-coordinates, is given. You can also assume that the constant c above is given. Your algorithm should run in $O(n)$ time.

c. (5 pts.) Complete algorithm and time complexity. Describe the complete algorithm, and analyze its time complexity.

Solution:

a. In a brute-force solution, we check all possible triplets to find the triangle with the minimum perimeter:

```
Algorithm Brute-Force (P[1 · n])
    init d* as the minimum perimeter and (i*, j*, j*) as the optimal 3 points
    For i = 1 to (n − 2)
        For j = i + 1 to (n − 1)
            For k = j + 1 to n
                Compute d(i, j, k), and if it is smaller than d*:
                let d* be this smaller perimeter and let (i*, j*, k*) = (i, j, k).
            End
        End
    End
    Return (i*, j*, k*) and d*
End algorithm
```

   This algorithm certainly runs in $\Theta(n^3)$ time.

b. (a) Assume that triangle $\langle i, j, k\rangle$ has a perimeter less than d^*. Clearly, this triangle must span both sides. Without loss of generality, assume point i locates on the left side of q and points j and k locates on the right side of q. Recall the triangle inequality: the sum of the lengths of any two sides of a triangle is greater than the length of the third side. Hence,

the length of any side of the triangle, namely $dist(i,j)$, $dist(i,k)$, and $dist(j,k)$, must not exceed $d^*/2$. Since points i and j are on different sides of q, they must both be in the band as otherwise their distance is beyond $d^*/2$. The same argument can be applied for i and k. Combined, all three points must be in the band.

(b) The above proof can be used here as well. Let $\langle i,j,k\rangle$ be a triangle with perimeter smaller than d^*. As argument above, $dist(i,j) < d^*/2$ and $dist(i,k) < d^*/2$. Hence, the vertical distance between points i and j must be smaller $d^*/2$, which gives $y_i - d^*/2 \leq y_j \leq y_i + d^*/2$. Similarly we can prove $y_i - d^*/2 \leq y_k \leq y_i + d^*/2$.

(c) We can divide each $(d^*/2) \times (d^*/2)$ box into 9 smaller, $(d^*/6) \times (d^*/6)$, boxes. We show that each smaller box cannot have more than 2 points inside (including the boundary). Suppose conversely there exists three points (i,j,k) in one smaller box. We have that $dist(i,j), dist(i,k), dist(j,k)$ must be all upper bounded by the diagonal of the smaller box which is $\sqrt{2} \cdot d^*/6$. Hence its perimeter $d(i,j,k) \leq 3 \cdot \sqrt{2} \cdot d^*/6 < d^*$. This is a contradiction as each smaller box locates entire on one side of q, violating the condition that the minimum perimeter of a triangle formed by any 3 points from P_L or P_R is d^*. Thus, the number of points in a $(d^*/2) \times (d^*/2)$ box should not exceed $2 \times 9 = 18$.

(d) Based on b(b), in order to search for possible triangle with smaller perimeter than d^*, for each point i in B, we only need to check (distinct) point j and k that are above point i and satisfy $y_j, y_k \leq y_i + d^*/2$. All such points j and k must be in a box either at the same layer with poins i, or in a box one layer above, since the height of each box is $d^*/2$. There are at most 4 such boxes (each layer contains 2) where points j and k can be. Hence, we only need to check points j and k within $4c$ positions in B after point i, since b(c) proved that each box contains at most c points.

```
Merge-Smallest-Triangle (B, d*, c)
    init d* as the minimum perimeter and (i*, j*, j*) as the optimal 3 points
    For (i = 1; i ≤ |B| − 2; i++)
        For (j = i + 1 → 4c)
            For (k = j + 1 → 4c)
                Compute d(B[i], B[j], B[k])
                Compute d(B[i], B[j], B[k]), and if it is smaller than d*:
                let d* be this smaller perimeter and let (i*, j*, k*) = (i, j, k).
            End
        End
    End
    Return (i*, j*, k*) and d*
End algorithm
```

The above algorithm performs "Compute" $16 \cdot c^2 \cdot |B|$ times. This is $\Theta(n)$ since c is a constant and that $|B| \leq n$.

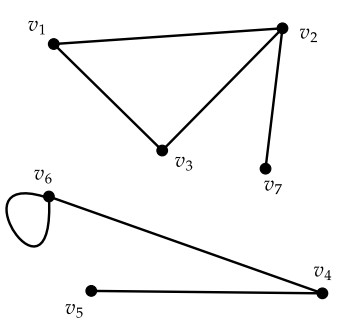
c. Complete algorithm: In each recursion, we divide the points into left and right halves, then recursively process each half separately. Afterward, we call Merge-Smallest-Triangle to combine the results from the sub-recursions. Here is the pseudocode for this algorithm:

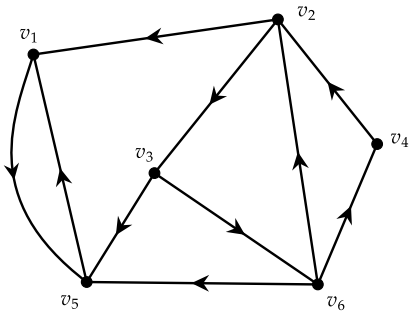
```
Find-Smallest-Triangle (P)
    Divide point set P into P_L and P_R

    {< ℓ*_0, ℓ*_1, ℓ*_2 >, d*_L} = Find-Smallest-Triangle (P_L)
    {< r*_0, r*_1, r*_2 >, d*_R} = Find-Smallest-Triangle (P_R)
    d* = min(d*_L, d*_R)

    Construct B
    {< s*_0, s*_1, s*_2 >, d*_s} = Merge-Smallest-Triangle (B, d*)

    If d*_s < d*
        Return {< r*_0, r*_1, r*_2 >, d*_s}
    Else if d*_L < d*_R
        Return {< ℓ*_0, ℓ*_1, ℓ*_2 >, d*_L}
    Else
        Return {< r*_0, r*_1, r*_2 >, d*_R}
    End if
End algorithm
```

Time complexity: Note that, similar to Lecture 06, the points piped to Find-Smallest-Triangle should be two lists, sorted in their x-coordinates and y-coordinates. The construction of the two lists for P_L, P_R, and the construction of B can all be done in $\Theta(n)$ time. The merge-step, Merge-Smallest-Triangle, can also be done in $\Theta(n)$ time as we showed in b(c). The recurrence is therefore $T(n) = 2 \cdot T(n/2) + \Theta(n)$, which gives $T(n) = \Theta(n \log n)$.

4. (12 pts.) Problem 3.

    1. (3 pts.) Draw an undirected graph with 4 vertices and 4 edges, and its adjacency matrix contains exactly 6 1s, or show that such graph does not exist.
    2. (3 pts.) Draw a directed graph with 4 vertices and 4 edges, and its adjacency matrix contains exactly 6 1s, or show that such graph does not exist.
    3. (3 pts.) Show all connected components in the undirected graph given below (left).
    4. (3 pts.) Show all connected components in the directed graph given below (right).





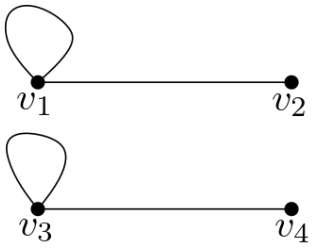


Figure 1: A solution for 3.1

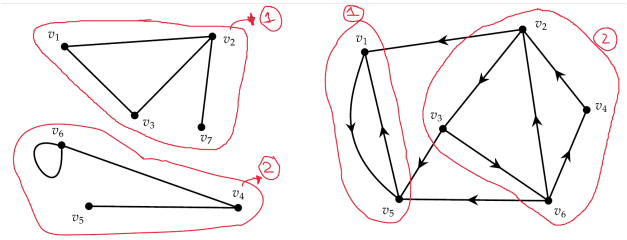


Figure 2: Connected components of 3.3 and 3.4

Solution.

1. Let s be the number of self-loop edges in an undirected graph and t be the number of non-self loop edges in the graph. Then the question states that:

$$s + t = 4$$
$$s + 2t = 6$$

   which has the solution $(s, t) = (2, 2)$. Thus any graph with 4 vertices, 2 self-loop edges and 2 non-self loop edges satisfies the problem. Fig 1 gives one such graph.

2. This can't happen. Proof. Let M be the adjacency matrix of a directed graph G. Then by the definition of an adjacency matrix $m_{ij} = 1$ iff (i, j) is an edge of G. Hence the no. of edges in G must equal the no. of 1s in M, contradicting the question.

3. There are 2 strongly connected components, given by the subgraphs induced on the vertex sets $\{v_1, v_2, v_3, v_7\}$ and $\{v_4, v_5, v_6\}$ respectively. See Fig 2.

4. There are again 2 strongly connected components, given by the subgraphs induced on the vertex sets $\{v_1, v_5\}$ and $\{v_2, v_3, v_4, v_6\}$. See Fig 2.