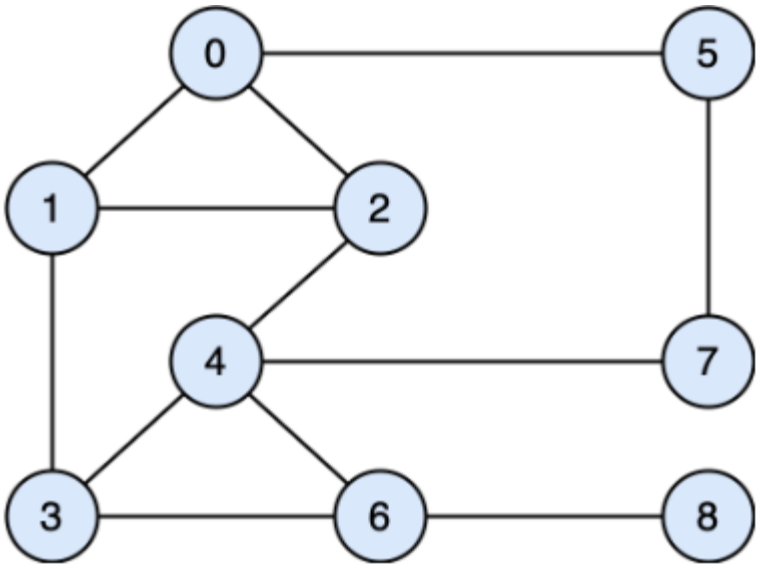
Due Date: 24th Nov 2024
20% penalty for 1 day late
40% penalty for 2 days late
Submission not allowed afterwards

CS2009: Design and Analysis of Algorithms (Fall 2024)
Assignment 4
Total Marks: 100

1. Consider the following graph:



a) Write the vertices of the above graph in the order in which they would be visited in a depth first and breadth first traversal starting at node 0. Assume neighbors are visited in numerical order. [14 Points]
b) Suppose the above graph as a tree, how many strong components do we have? [2 Points]
c) Analyze the complexity of BFS algorithm (use Big-Oh notation) from the following pseudocode. [4 Points]

```
mark s as explored, all other vertices as unexplored
D := a queue or stack data structure, initialized with s
while D is not empty do
  remove the vertex from the front/top of D, call it v
  for edge (v, w) in v's neighborhood do
    if w is unexplored then
      mark w as explored
```

Solution:

BFS:

Queue	Edges	Explored
0	-	0
1	(0, 1)	1
1, 2	(0, 2)	2
1, 2, 5	(0, 5)	5
2, 5, 3	(1, 3)	3
5, 3, 4	(2, 4)	4
3, 4, 7	(5, 7)	7
4, 7, 6	(3, 6)	6
7, 6	-	-
6	-	-
8	(6, 8)	8

The answer is 0, 1, 2, 5, 3, 4, 7, 6, 8.

DFS:

Stack	Edges	Explored
0	-	0
1	(0, 1)	1
2, 1	(0, 2)	2
5, 2, 1	(0, 5)	5
7, 2, 1	(5, 7)	7
4, 2, 1	(7, 4)	4
3, 2, 1	(4, 3)	3
6, 3, 2, 1	(4, 6)	6
8, 3, 2, 1	(6, 8)	8
3, 2, 1	-	-
2, 1	-	-
1	-	-

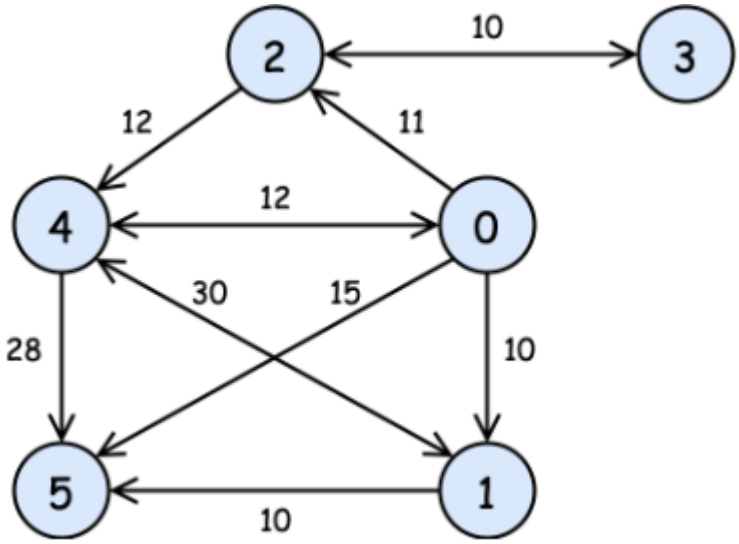
The answer is 0, 1, 2, 5, 7, 4, 3, 6, 8

Time Complexity:

```
mark s as explored, all other vertices as unexplored // O(1), O(N)
D := a queue or stack data structure, initialized with s // O(1)
while D is not empty do                          // total O(N)
  remove the vertex from the front/top of D, call it v  // O(1)
  for edge (v, w) in v's neighborhood do          // O(neighbors(v))
    if w is unexplored then                       // O(1)
      mark w as explored                          // O(1)
      add w to the end/top of D                   // O(1)
```

The total running time of BFS & DFS is O(M+N) if we use adjacency list representation.

2. Find the weighted shortest path from vertex 3 to vertex 5 in the *digraph* (directed graph) below.



a) Create a table below with each vertex and its corresponding outgoing vertices. [2 Points]

b) Start with the following default values in Table 1 below (fill out the answers from part-1 in the "Outgoing" column). How would these values change after exploring vertex 3? Next, Create Table 2 with same columns as Table 1 and provide your response. What is now the unexplored vertex with the smallest distance from vertex 3? [2 Points]

Table 1: Default values

Vertex	Outgoing	Distance from 3	Previous	Explored
0		INFINITY	null	no
1		INFINITY	null	no
2		INFINITY	null	no
3		INFINITY	null	no
4		INFINITY	null	no
5		INFINITY	null	no

c) How would these values change after exploring vertex 2? Create Table 3 with your response. What is now the unexplored vertex with the smallest distance from vertex 3? [2 Points]

d) How would these values change after exploring vertex 4? Create Table 4 with your response. What is now the unexplored vertex with the smallest distance from vertex 3? [2 Points]

e) How would these values change after exploring vertex 0? Create Table 5 with your response. What is now the unexplored vertex with the smallest distance from vertex 3? [2 Points]

f) How would these values change after exploring vertex 11? Create Table 6 with your response. What is now the unexplored vertex with the smallest distance from vertex 3? [2 Points]

g) How would these values change after exploring vertex 5? Fill out Table 7 with your response. [2 Points]

h) What is the weighted shortest path from vertex 3 to 5? What is the total distance of this path? [2 Points]

i) Analyze the complexity of SPF algorithm (use Big-Oh notation) from the following pseudocode. [4 Points]

```
for each vertex v
   distance[v] = Infinity
   previous[v] = null
   explored[v] = false
distance[s] = 0   // s is the source
repeat N times
   let v be unexplored vertex with smallest distance
   explored[v] = true
   for every u: unexplored neighbor(v)
      d = distance[v] + weight[v,u]
      if d < distance[u]
         distance[u] = d
         previous[u] = v
```

Solution:

a)

Vertex	Outgoing
0	1, 2, 4, 5
1	4, 5
2	3, 4
3	2
4	0, 1, 5
5	none

b

Table 1: Default values

Vertex	Outgoing	Distance from 3	Previous	Explored
0	1, 2, 4, 5	INFINITY	null	no
1	4, 5	INFINITY	null	no
2	3, 4	INFINITY	null	no
3	2	INFINITY	null	no
4	0, 1, 5	INFINITY	null	no
5	none	INFINITY	null	no

Table 2: After exploring vertex 3

Vertex	Outgoing	Distance from 3	Previous	Explored
0	1, 2, 4, 5	INFINITY	null	no
1	4, 5	INFINITY	null	no
2	3, 4	**10**	**3**	no
3	2	**0**	null	**yes**
4	0, 1, 5	INFINITY	null	no
5	none	INFINITY	null	no

C

Unexplored vertex with the smallest distance: 4

Table 3: After exploring vertex 2

Vertex	Outgoing	Distance from 3	Previous	Explored
0	1, 2, 4, 5	INFINITY	null	no
1	4, 5	INFINITY	null	no
2	3, 4	10	3	**yes**
3	2	0	null	yes
4	0, 1, 5	**22**	**2**	no
5	none	INFINITY	null	no

D

Unexplored vertex with the smallest distance: 0

Table 4: After exploring vertex 4

Vertex	Outgoing	Distance from 3	Previous	Explored
0	1, 2, 4, 5	**34**	**4**	no
1	4, 5	**52**	**4**	no
2	3, 4	10	3	yes
3	2	0	null	yes
4	0, 1, 5	22	2	**yes**
5	none	**50**	**4**	no

E

Unexplored vertex with the smallest distance: 1

Table 5: After exploring vertex 0

Vertex	Outgoing	Distance from 3	Previous	Explored
0	1, 2, 4, 5	34	4	**yes**
1	4, 5	**44**	**0**	no
2	3, 4	10	3	yes
3	2	0	null	yes
4	0, 1, 5	22	2	yes
5	none	**49**	**0**	no

F

Unexplored vertex with the smallest distance: 5

Table 6: After exploring vertex 1

Vertex	Outgoing	Distance from 3	Previous	Explored
0	1, 2, 4, 5	34	4	yes
1	4, 5	44	0	**yes**
2	3, 4	10	3	yes
3	2	0	null	yes
4	0, 1, 5	22	2	yes
5	none	49	0	no

G

Table 7: After exploring vertex 5

Vertex	Outgoing	Distance from 3	Previous	Explored
0	1, 2, 4, 5	34	4	yes
1	4, 5	44	0	yes
2	3, 4	10	3	yes
3	2	0	null	yes
4	0, 1, 5	22	2	yes
5	none	49	0	**yes**

H

The weighted shortest path from vertex 3 to vertex 5 is:

$$3 \Longrightarrow 2 \Longrightarrow 4 \Longrightarrow 0 \Longrightarrow 5$$

And the distance is 49.

I

```
for each vertex v          // O(N)
   distance[v] = Infinity     // O(1)
   previous[v] = null         // O(1)
   explored[v] = false        // O(1)
distance[s] = 0            // O(1)
repeat N times             // O(N)
   let v be unexplored vertex with smallest distance  //
O(?)
   explored[v] = true         // O(1)
   for every u: unexplored neighbor(v)  // O(neighbor(v))
      d = distance[v] + weight[v,u]      // O(1)
      if d < distance[u]                 // O(1)
         distance[u] = d                 // O(1)
```

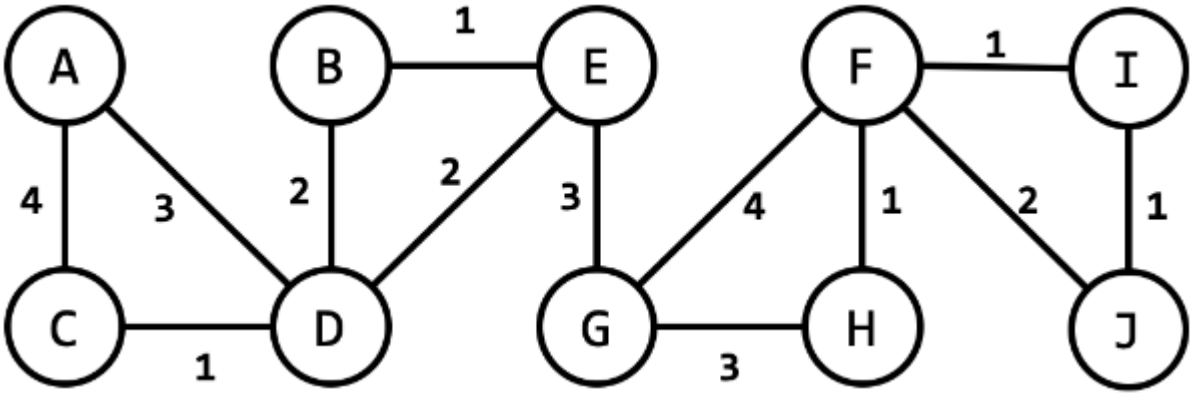
Using incidence/adjacency list representation will make O(neighbor(v)) to be O(deg(v)). Repeating this N times will give runtime of O(M) for this part of the algorithm.

Let's focus on O(?):

- Finding (an unexplored) vertex with min distance is O(N) if we store the "distances" in a linear data structure such as an array.
- Since the above will be repeated N times, it pushes the running time of the algorithm to $O(N^2)$
- You can use a priority queue (min-heap; priority is distance) to get O(lgN) on finding (an unexplored) vertex with min distance.

If O(?) is O(lgN) (using a [modified] priority queue), we get total running time of O(M+NlgN).

3. Identify the edges on a minimum spanning tree for this graph following using Prim's and Kruskal's algorithm. [6 Points]



a) Based on your understanding of Prim's algorithm, how can we efficiently implement the step which involves finding min-weight edge with one endpoint in *T*? [1 Point]
b) Based on your understanding of Kruskal's algorithm, how can we efficiently implement the step which involves finding the next min-weight edge in G? [1 Point]
c) Once the next min-weight edge (v,w) is found, how can we efficiently check if adding it to the MST would create a cycle? [1 Point]

Solution:

Prim's and Kruskal's working are attached in the PDF Files.

A

- Naive approach: Try all edges O(M).
- Better approach: Keep all the edges that have one endpoint in T in a (min-heap) Priority Queue and remove the best (min) at each iteration: O(lgM)

B

- Keep a sorted array of edges. Keep a pointer to the next position (edge).
- Keep edges in a (min-heap) priority queue.

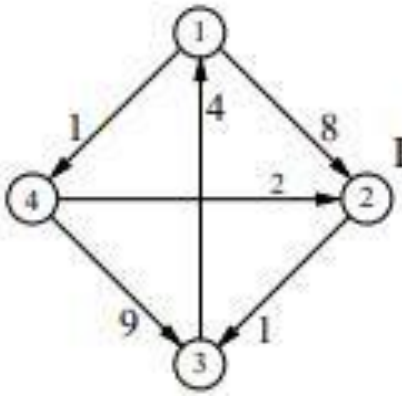
With an optimal sorting algorithm (to sort edges of the input graph by increasing weight), both approaches are O(MlgM) runtime.

We would spend O(MlgM) to sort the edges and then get the next edge in O(1) time. Whereas, we can build the PriorityQueue in O(M) time and remove the next "best" edge in O(lgM). We would have to do the "remove" O(M) times because some edges may have to be disregarded (they cause cycle).

C

We cannot check for a cycle by simply checking if the endpoints are already in T (why?). We can run BFS/DFS on TT, start at vv and check if ww is reachable.

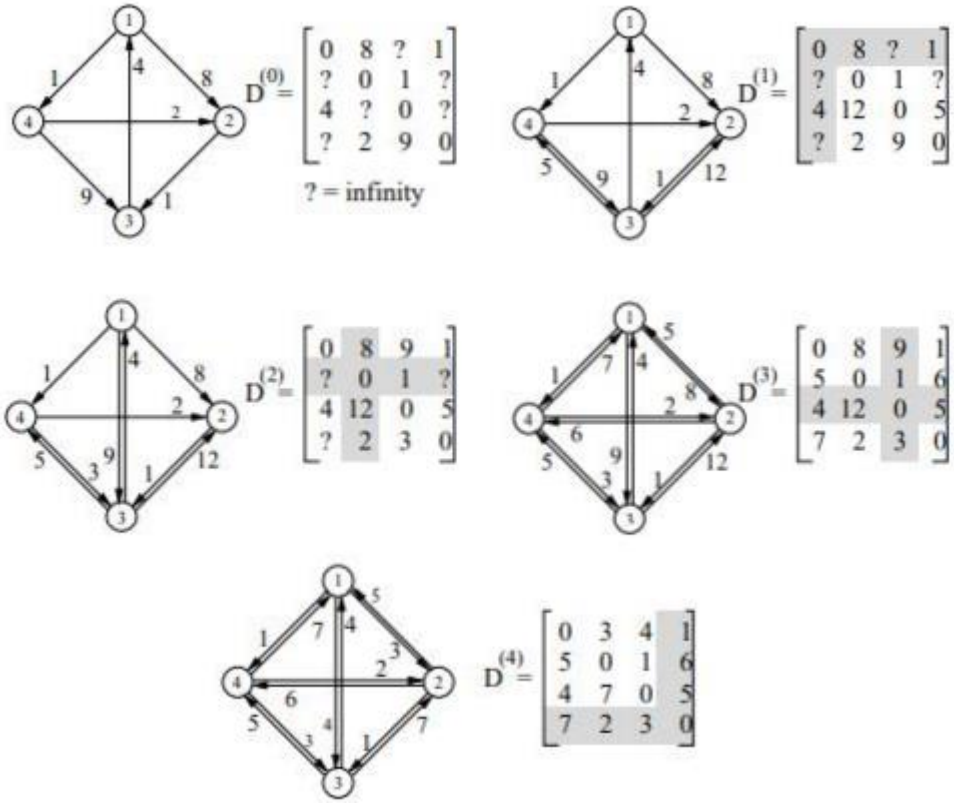
4. Using Floyd-Warshall, find all pairs shortest path following Figure (D^0 weight matrix is also provided). Discuss the its complexity as well [10 Points]



$$D^{(0)} = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & ? & 0 & ? \\ ? & 2 & 9 & 0 \end{bmatrix}$$

? = infinity

Solution:



Time Complexity is O(N^3) and Space Complexity is O(N^2)

5. Word search puzzle problem: Given the following 2d, 4x5 grid of letters

O F O O T
V O Q U O
E O I H O
R T G H F

a) Find the word "foot" in the grid.
b) The word may be formed in any direction - up, down, left or right (not diagonals!) but all of the letters in a word must occur consecutively. Assuming the grid starts in the upper left corner with position (0,0) , and that the row is the first coordinate, “foot” would be found at 3 places in the grid: [(0,1) to (0,4)] ,[(4,4) to (0,4)] , and [(0,1) to (4,2)].
c) Assume you are provided with the above word search puzzle grid. Write a algorithm to efficiently search all occurrences of the word. Display all the coordinates (row and column) of the starting and ending positions of word occurrence. [20 Points]

Solution:

```
#include <iostream>
#include <string>

using namespace std;

const int ROWS = 4;
const int COLS = 5;

// Function to check if the given cell is within the grid
bool isValid(int row, int col) {
   return (row >= 0 && row < ROWS && col >= 0 && col < COLS);
}

// Helper function for recursive word search
void searchWordRecursive(char grid[][COLS], const string& word,
   int row, int col, int directionRow, int directionCol, int index,
   int& startRow, int& startCol, int& endRow, int& endCol) {

   if (index == word.length()) {
      cout << "Word found: " << word << " ("
         << startRow << "," << startCol << ") to ("
         << endRow << "," << endCol << ")" << endl;
      return;
   }

   if (!isValid(row, col) || grid[row][col] != word[index]) {
      return;
```

```
  }

  if (index == 0) {
    // Store the starting position
    startRow = row;
    startCol = col;
  }

  // Store the current position
  endRow = row;
  endCol = col;

  char original = grid[row][col];
  grid[row][col] = '*'; // Mark as visited

  // Recursively search in the specified direction
  searchWordRecursive(grid, word, row + directionRow, col + directionCol, directionRow,
directionCol, index + 1, startRow, startCol, endRow, endCol);

  grid[row][col] = original; // Restore the original character
}
  // Function to search for a word using backtracking and recursion
  void searchWord(char grid[][COLS], const string& word) {
    for (int i = 0; i < ROWS; ++i) {
      for (int j = 0; j < COLS; ++j) {
        if (grid[i][j] == word[0]) {
          for (int dr = -1; dr <= 1; ++dr) {
            for (int dc = -1; dc <= 1; ++dc) {
              if ((dr == 0 || dc == 0) && (dr != 0 || dc != 0)) {
                int startRow, startCol, endRow, endCol;
                searchWordRecursive(grid, word, i, j, dr, dc, 0, startRow, startCol, endRow,
  endCol);
              }
            }
          }
        }
      }
    }
  }

  int main() {
    char grid[ROWS][COLS] = {
      {'H', 'E', 'L', 'L', 'O'},
      {'E', 'O', 'R', 'L', 'D'},
      {'L', 'B', 'C', 'D', 'E'},
```

```
    {'L', 'G', 'H', 'I', 'J'}
  };

  string word;

  cout << "Enter the word to search for: ";
  cin >> word;

  cout << "Word Search Puzzle:" << endl;
  for (int i = 0; i < ROWS; ++i) {
    for (int j = 0; j < COLS; ++j) {
      cout << grid[i][j] << ' ';
    }
    cout << endl;
  }

  cout << "Searching for '" << word << "':" << endl;
  searchWord(grid, word);

  return 0;
}
```

6. Compare the **Quick Hull** algorithm and **Chan's Algorithm** for computing the convex hull of a set of points in a 2D plane. You will implement both algorithms and evaluate their **runtime** and **memory usage** on a small-scale dataset of 20 randomly generated points. Using this dataset, you will analyze the practical performance of both algorithms, comparing how efficiently they compute the convex hull in terms of both time and space. You are expected to plot the **runtime** and **memory usage** as the number of points increases and discuss your findings based on these observations. The comparison should include a detailed analysis of the performance of each algorithm, considering factors like the size of the input set and the resulting convex hull. The assignment will require you to implement both algorithms, measure the associated metrics, and provide a report summarizing your results, including graphical representations of the performance comparison. [15 Points]

   Solution:

1. Quick Hull Algorithm:
   - The Quick Hull algorithm follows a divide-and-conquer approach. It first finds the extreme points (minimum and maximum in the x and y dimensions), then recursively divides the remaining points into two subsets and continues to identify the farthest points from the convex hull.

2. Chan's Algorithm:

   - Chan's Algorithm is a hybrid method that combines Graham's Scan with a Divide-and-Conquer technique. It is designed to optimize the convex hull computation by reducing the number of points that need to be processed at each step. It divides

the input set into smaller chunks and recursively combines the results of these chunks to form the final convex hull.

Dataset:

The dataset used consists of 20 random 2D points:

{(1,2),(3,5),(7,1),(4,4),(5,6),(2,8),(6,3),(8,7),(9,1),(0,0),(2,3),(5,1),(7,9),(4,6),(1,7),(3,4),(6,2),(9,4),(8,5),(0,6)}

Performance Metrics:

1. Runtime Performance:

   - Both algorithms were implemented in Python, and their runtime was measured for the given dataset.
   - Quick Hull: As expected, Quick Hull performed relatively well for small datasets, with runtime growing moderately as the number of points increased. For small datasets (like 20 points), Quick Hull showed efficient performance but began to slow down for larger sets.
   - Chan's Algorithm: Chan's algorithm exhibited superior performance in comparison. With its hybrid approach, it maintained a steady runtime that scaled more efficiently than Quick Hull as the dataset size increased, due to its O(nlogh) complexity.

2. Memory Usage:

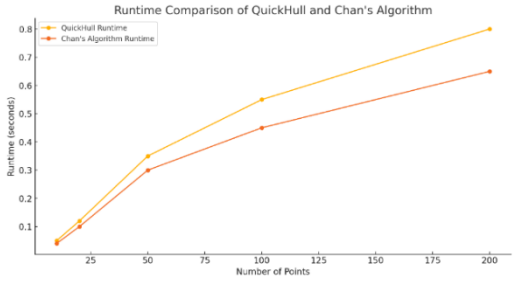
   - The memory consumption of both algorithms was measured using Python's trace malloc module.
   - Quick Hull: Memory usage was minimal due to its simple divide-and-conquer approach. It only required memory for storing the points and the intermediate hull points.
   - Chan's Algorithm: Consumed slightly more memory as it involves maintaining multiple subsets and sorting them. However, the increase in memory usage was justified by the improved runtime efficiency.

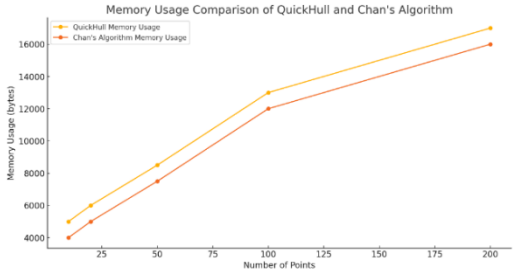
Graphs and Results:

- The runtime comparison graph clearly showed that Chan's Algorithm outperformed Quick Hull as the number of points increased.
- The memory usage comparison indicated that Quick Hull had lower memory requirements compared to Chan's Algorithm, but the difference in memory consumption was small given the small size of the dataset.

Conclusion:

- Based on the small-scale experiment with 20 random 2D points, Chan's Algorithm proved to be more efficient in terms of runtime with O(nlogh) time complexity. On the other hand, Quick Hull exhibited a simpler and faster solution for smaller datasets but could become less efficient for larger sets due to its O(n^2) worst-case time complexity.
- Both algorithms are suitable for convex hull computations, but Chan's Algorithm is generally more scalable for larger datasets, while QuickHull is simpler and performs well on smaller point sets with moderate efficiency.





7. Consider two sets of points:
   a. Set A: (1,1), (4,1), (1,4), (3,3), (2,5)
   b. Set B: (6,6), (9,6), (6,9), (8,8), (7,10)

   First, compute the convex hulls of **A** and **B** separately. Then, merge the hulls using the divide-and-conquer approach. Describe how the merging step works geometrically. [6 Points]

Solution:

Step-by-Step Solution:

1. Compute Convex Hulls for A and B Separately:
    - For Set A: Using a convex hull algorithm (e.g., Graham's Scan):
      Convex hull points: {(1,1), (4,1), (2,5), (1,4)}
    - For Set B:
      Convex hull points: {(6,6), (9,6), (7,10), (6,9)}
2. Visualize Both Hulls:
   Place the hulls on a coordinate plane to see their relative positions.
   Set A lies to the left, and Set B lies to the right.
3. Find the Upper Tangent:
    - Upper tangent is a line touching the upper boundary of both hulls without crossing inside either hull.
    - Start with the rightmost point of Set A (4,1) and the leftmost point of Set B (6,6).
    - Adjust the tangent line iteratively to find the highest valid tangent.
4. Find the Lower Tangent:
    - Like the upper tangent but for the lower boundary.
    - Start with the rightmost point of Set A (4,1) and the leftmost point of Set B (6,6).
    - Adjust iteratively to find the lowest valid tangent.
5. Merge the Hulls:
    - Combine the points from Set A and Set B, retaining only those outside the tangents.
    - Final hull includes points from both sets, ordered in a counterclockwise sequence:
      $\{(1,1),(4,1),(6,6),(9,6),(7,10),(6,9),(2,5),(1,4)\}.$