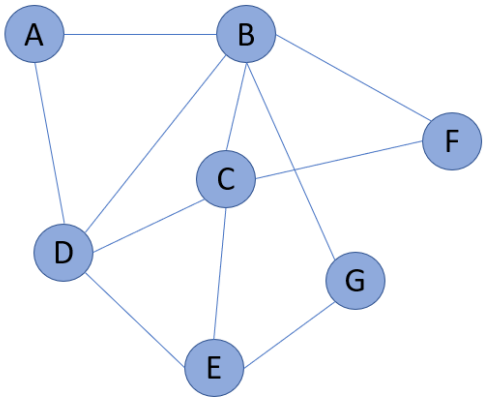
# Solution: Graph Algorithms – I

1. **Write BFS and DFS for a graph**: What would be the BFS and DFS traversal for the below graphs. Write the nodes for BFS and DFS. Start at node A.




This is one of the possible traversals.

BFS : [use queue]

Queue : A, B, D, C, G, F, E

Result : A, B, D, C, G, F, E

DFS : [use stack]

Stack : A, D, E, G, B, F, C

Result : A, D, E, G, B, F, C

2. **Apply BFS/DFS to solve a problem**

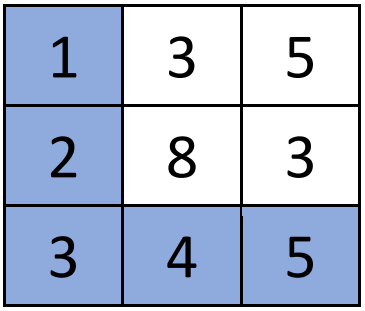
You are given a 3-D puzzle. The length and breadth of the puzzle is given by a 2D matrix puzzle[m][n]. The height of each cell is given by the value of each cell, the value of puzzle[row][column] give the height of the cell [row][column]. You are at [0][0] cell and you want to reach to the bottom right cell [m-1][n-1], the destination cell. You can move either up, down, left, or right. Write an algorithm to reach the destination cell with minimal effort.
How to effort is defined: The effort of route is the maximum absolute difference between two consecutive cells.

Example if a route requires us to cross heights: 1, 3, 4, 6, 3, 1

The absolute differences between consecutive cells is: |1-3| = 2, |3-4|=1, |4-6|=2, |6-3|=3, |3-1|=2; this gives us the values: {2, 1, 2, 3, 2}. The maximum value of these absolute differences is 3. Hence the effort required on this path will be: 3.

Example:

Input: puzzle[][] = [[1, 3, 5], [2, 8, 3], [3, 4, 5]]
Output: 1
Explanation: The minimal effort route would be [1, 2, 3, 4, 5] which has an effort of value 1. This is better than other routes for instance, route [1, 3, 5, 3, 5] which has an effort of 2.




a. Describe the solution to the problem

The absolute difference between adjacent cells A and B can be perceived as the weight of an edge from cell A to cell B. Thus, we could use Dijkstra's Algorithm which is used to find the shortest path in a weighted graph with a slight modification of criteria for the shortest path.

Use a differenceMatrix of size row*col where each cell represents the minimum effort required to reach that cell from all the possible paths from the starting cell. Initialize each cell in differenceMatrix to infinity.

We push the visited adjacent cells in a priority queue. The priority queue holds all the reachable cells sorted by its value in differenceMatrix, i.e the cell with minimum absolute difference of effort with its adjacent cells would be at the top of the queue.

b. Write pseudocode for your solution

```
def minEffort(puzzle):
    rows = len(puzzle)
    cols = len(puzzle[0])

    #Stores min absolute difference of effort
    difference_matrix[rows][cols] initalize to infinity
    difference_matrix[0][0] = 0 #start position it will be 0

    # effort, row, col
    priority_queue = [(0, 0, 0)]

    while priority_queue:
        difference, cur_row, cur_col = pop item from priority_queue

        if difference is more than
```

```
difference_matrix[cur_row][cur_col]) then continue

        if we reach last cell return difference

        process each possible direction
                #In adjacent cell find best new effort
                new_difference = max(difference,
abs(puzzle[new_row][new_col] - puzzle[cur_row][cur_col]))

                # check if it is less than current minimum abs path
value of another path update difference_matrix[new_row][new_col]

                # push this value to the heap (new_difference,
new_row, new_col))
```

c. Implement the algorithm. Name your function **minEffort(puzzle)**. Name your file **MinPuzzle.py**

```
import heapq

def minEffort(puzzle):
    rows = len(puzzle)
    cols = len(puzzle[0])

    #Stores min absolute difference of effort
    difference_matrix = [[float('inf')] * cols for _ in range(rows)]
    difference_matrix[0][0] = 0 #start position it will be 0

    # effort, row, col
    priority_queue = [(0, 0, 0)] #heap to store minimum absolute value
of effort to reach i,j (minPathAbsVal, i, j)

    # directions that we can travel
    moves = [[-1, 0], [1, 0], [0, -1], [0, 1]]  # for traversing
L,R,U,D
    while priority_queue:
        difference, cur_row, cur_col = heapq.heappop(priority_queue)

        if (difference > difference_matrix[cur_row][cur_col]):
            continue

        if (cur_row == rows - 1 and cur_col == cols - 1):
            return difference

        for x, y in moves:
            new_row, new_col = cur_row + x, cur_col + y

            if (0 <= new_row < rows and 0 <= new_col < cols):
                new_difference = max(difference,
abs(puzzle[new_row][new_col] - puzzle[cur_row][cur_col]))  # best new
effort

                if (new_difference <
difference_matrix[new_row][new_col]):  # check if it is less than
```

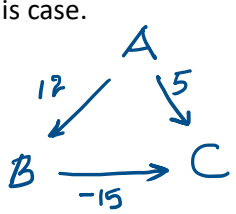
```
current minimum abs path value of another path
                    difference_matrix[new_row][new_col] =
new_difference

                    heapq.heappush(priority_queue, (new_difference,
new_row, new_col))  # push this value to the heap

    return 0

print(minEffort([[1, 3, 5], [2, 8, 3], [3, 4, 5]]))  #1
print(minEffort([[1,2,2],[3,8,2],[5,3,5]])) #2
```

d. What is the time complexity of your implementation?

If m = number of rows and n = number of columns

It will take O(m.n) time to visit every cell in the matrix. The priority queue will contain atmost m*n cells, so it will take O(log(m*n)) time to heapify after every new cell is added to the queue. This will take total of O(mnlog(mn))

3. **Analyze Dijkstra with negative edges**: Analyze with a sample graph and show why Dijkstra does not work with negative edges. Give the sample graph and write your explanation why Dijkstra would not work in this case.




Dijkstra's algorithm starting from A discovers C and B. In the next step, it visits C and marks it as visited.

Since the vertex is marked visited, the algorithm assumes that the path developed to this vertex (A->C) is the shortest.

But actually, the shortest path from A to C is A->B->C (given that the negative weighted edge has some significance).