|  |  |
| --- | --- |
| **Rotten Oranges in C++** | |
| #include<bits/stdc++.h>  using namespace std;  class Solution {  public:  //Function to find minimum time required to rot all oranges.  int orangesRotting(vector < vector < int >> & grid) {  // figure out the grid size  int n = grid.size();  int m = grid[0].size();  // store {{row, column}, time}  queue < pair < pair < int, int > , int >> q;  int vis[n][m];  int cntFresh = 0;  for (int i = 0; i < n; i++) {  for (int j = 0; j < m; j++) {  // if cell contains rotten orange  if (grid[i][j] == 2) {  q.push({{i, j}, 0});  // mark as visited (rotten) in visited array  vis[i][j] = 2;  }  // if not rotten  else {  vis[i][j] = 0;  }  // count fresh oranges  if (grid[i][j] == 1) cntFresh++;  }  }  int tm = 0;  // delta row and delta column  int drow[] = {-1, 0, +1, 0};  int dcol[] = {0, 1, 0, -1};  int cnt = 0;  // bfs traversal (until the queue becomes empty)  while (!q.empty()) {  int r = q.front().first.first;  int c = q.front().first.second;  int t = q.front().second;  tm = max(tm, t);  q.pop();  // exactly 4 neighbours  for (int i = 0; i < 4; i++) {  // neighbouring row and column  int nrow = r + drow[i];  int ncol = c + dcol[i];  // check for valid cell and  // then for unvisited fresh orange  if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&  vis[nrow][ncol] == 0 && grid[nrow][ncol] == 1) {  // push in queue with timer increased  q.push({{nrow, ncol}, t + 1});  // mark as rotten  vis[nrow][ncol] = 2;  cnt++;  }  }  }  // if all oranges are not rotten  if (cnt != cntFresh) return -1;  return tm;  }  };  int main() {  vector<vector<int>>grid{{0,1,2},{0,1,2},{2,1,1}};  Solution obj;  int ans = obj.orangesRotting(grid);  cout << ans << "\n";  return 0;  } | **Step 1: BFS Traversal**  The queue will be used to perform BFS, where we process the rotten oranges and spread the rot to adjacent fresh oranges. The variable tm will track the maximum time it takes to rot all oranges.  **First BFS Iteration (Queue: q = { {{0, 2}, 0}, {{1, 2}, 0}, {{2, 0}, 0} }):**   * **Processing rotten orange at (0, 2) at time 0**:   + **Neighbors**:     - (0, 1) is a fresh orange (grid[0][1] == 1), so we rot it and add it to the queue with time 1: q.push({{0, 1}, 1}).   + Updated state:   vis = {  {0, 2, 2},  {0, 1, 2},  {2, 1, 1}  }  q = { {{1, 2}, 0}, {{2, 0}, 0}, {{0, 1}, 1} }   * **Processing rotten orange at (1, 2) at time 0**:   + **Neighbors**:     - (1, 1) is a fresh orange (grid[1][1] == 1), so we rot it and add it to the queue with time 1: q.push({{1, 1}, 1}).   + Updated state:   vis = {  {0, 2, 2},  {0, 2, 2},  {2, 1, 1}  }  q = { {{2, 0}, 0}, {{0, 1}, 1}, {{1, 1}, 1} }   * **Processing rotten orange at (2, 0) at time 0**:   + **Neighbors**:     - (2, 1) is a fresh orange (grid[2][1] == 1), so we rot it and add it to the queue with time 1: q.push({{2, 1}, 1}).   + Updated state:   vis = {  {0, 2, 2},  {0, 2, 2},  {2, 2, 1}  }  q = { {{0, 1}, 1}, {{1, 1}, 1}, {{2, 1}, 1} }  **Second BFS Iteration (Queue: q = { {{0, 1}, 1}, {{1, 1}, 1}, {{2, 1}, 1} }):**   * **Processing rotten orange at (0, 1) at time 1**:   + **Neighbors**:     - (0, 0) is empty (grid[0][0] == 0), so nothing happens.   + Queue remains unchanged:   q = { {{1, 1}, 1}, {{2, 1}, 1} }   * **Processing rotten orange at (1, 1) at time 1**:   + **Neighbors**:     - (1, 0) is empty (grid[1][0] == 0), so nothing happens.   + Queue remains unchanged:   q = { {{2, 1}, 1} }   * **Processing rotten orange at (2, 1) at time 1**:   + **Neighbors**:     - (2, 2) is a fresh orange (grid[2][2] == 1), so we rot it and add it to the queue with time 2: q.push({{2, 2}, 2}).   + Updated state:   vis = {  {0, 2, 2},  {0, 2, 2},  {2, 2, 2}  }  q = { {{2, 2}, 2} }  **Final State:**  After the BFS traversal completes, the queue is empty and the vis array is:  vis = {  {0, 2, 2},  {0, 2, 2},  {2, 2, 2}  }  **Step 2: Checking if All Oranges Are Rotten**   * **Count of Rotten Oranges (cnt)**: The total number of rotten oranges in the grid is cnt = 4 (after BFS propagation). * **Count of Fresh Oranges (cntFresh)**: The initial count of fresh oranges is cntFresh = 4. * **Result**: Since cnt == cntFresh, all fresh oranges have been rotted.   **Step 3: Return the Time**   * The maximum time it took to rot all the fresh oranges is tm = 1.   Thus, the **minimum time required to rot all oranges is 1**. |
| **Output:-**  1 | |