<https://leetcode.com/problems/detonate-the-maximum-bombs/description/>

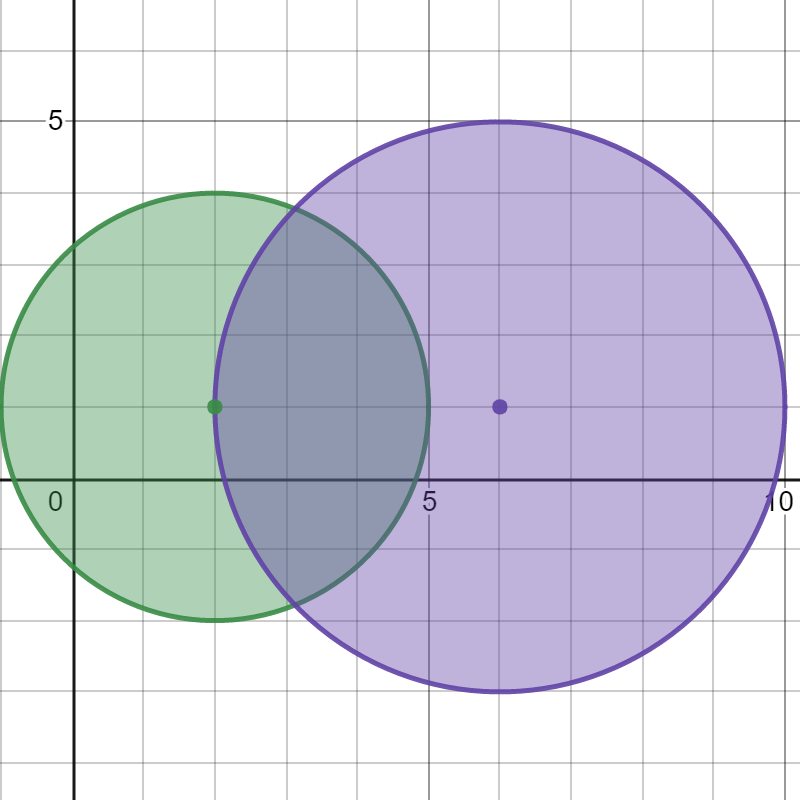
You are given a list of bombs. The **range** of a bomb is defined as the area where its effect can be felt. This area is in the shape of a **circle** with the center as the location of the bomb.

The bombs are represented by a **0-indexed** 2D integer array bombs where bombs[i] = [xi, yi, ri]. xi and yi denote the X-coordinate and Y-coordinate of the location of the ith bomb, whereas ri denotes the **radius** of its range.

You may choose to detonate a **single** bomb. When a bomb is detonated, it will detonate **all bombs** that lie in its range. These bombs will further detonate the bombs that lie in their ranges.

Given the list of bombs, return *the****maximum****number of bombs that can be detonated if you are allowed to detonate****only one****bomb*.

**Example 1:**



**Input:** bombs = [[2,1,3],[6,1,4]]

**Output:** 2

**Explanation:**

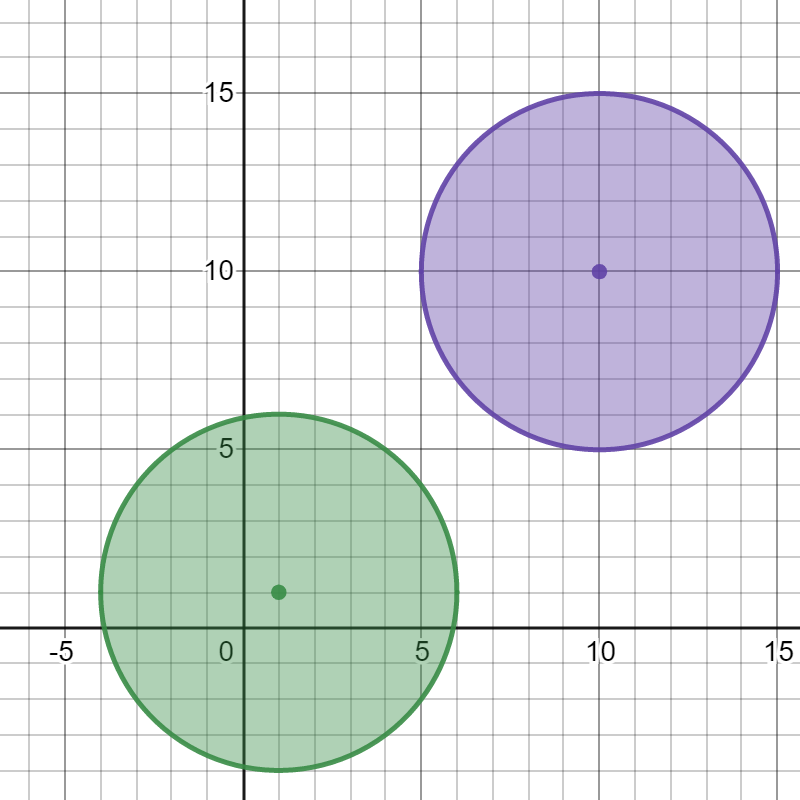
The above figure shows the positions and ranges of the 2 bombs.

If we detonate the left bomb, the right bomb will not be affected.

But if we detonate the right bomb, both bombs will be detonated.

So the maximum bombs that can be detonated is max(1, 2) = 2.

**Example 2:**



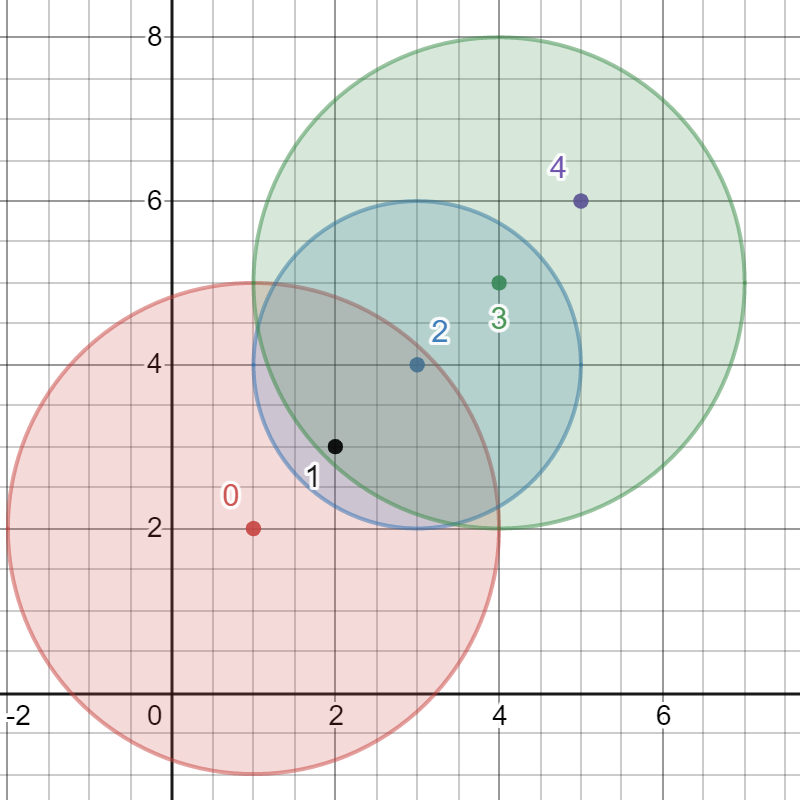
**Input:** bombs = [[1,1,5],[10,10,5]]

**Output:** 1

**Explanation:**

Detonating either bomb will not detonate the other bomb, so the maximum number of bombs that can be detonated is 1.

**Example 3:**



**Input:** bombs = [[1,2,3],[2,3,1],[3,4,2],[4,5,3],[5,6,4]]**Output:** 5**Explanation:**The best bomb to detonate is bomb 0 because:- Bomb 0 detonates bombs 1 and 2. The red circle denotes the range of bomb 0.- Bomb 2 detonates bomb 3. The blue circle denotes the range of bomb 2.- Bomb 3 detonates bomb 4. The green circle denotes the range of bomb 3.Thus all 5 bombs are detonated.

**Constraints:**

1 <= bombs.length <= 100

bombs[i].length == 3

1 <= xi, yi, ri <= 10^5

**Attempt 1: 2025-10-23**

**Solution 1: DFS (30 min)**

class Solution {

    public int maximumDetonation(int[][] bombs) {

        int n = bombs.length;

        int result = 0;

        for(int i = 0; i < n; i++) {

            result = Math.max(result, helper(i, new boolean[n], bombs));

        }

        return result;

    }

    private int helper(int index, boolean[] visited, int[][] bombs) {

        int count = 1;

        visited[index] = true;

        int n = bombs.length;

        for(int i = 0; i < n; i++) {

            if(!visited[i] && inRange(bombs[index], bombs[i])) {

                count += helper(i, visited, bombs);

            }

        }

        return count;

    }

    private boolean inRange(int[] a, int[] b) {

        long dx = a[0] - b[0];

        long dy = a[1] - b[1];

        long r = a[2];

        return dx \* dx + dy \* dy <= r \* r;

    }

}

Time Complexity: O(n^3) in the worst case.

This accounts for O(n^2) to build the graph and O(n^2) for DFS from each of the n starting points.

Space Complexity: O(n^2) for storing the adjacency list.

**Refer to Deepseek**

According to the search results, I found several Java solutions for LeetCode 2101. Detonate the Maximum Bombs. The most common and efficient approach models the problem as a **directed graph** and uses **DFS (Depth-First Search)** or BFS to explore chain reactions.

Here is a consolidated Java solution based on the DFS approach:

class Solution {

public int maximumDetonation(int[][] bombs) {

final int n = bombs.length;

int ans = 0;

// Build an adjacency list for the graph

List<Integer>[] graph = new List[n];

Arrays.setAll(graph, i -> new ArrayList<>());

// Build the directed graph edges

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

if (i == j) continue;

long ri = bombs[i][2];

// Check if bomb i can detonate bomb j

if (ri \* ri >= squaredDist(bombs, i, j)) {

graph[i].add(j);

}

}

}

// For each bomb, perform DFS to see how many bombs it can detonate

for (int i = 0; i < n; ++i) {

Set<Integer> visited = new HashSet<>();

visited.add(i);

dfs(graph, i, visited);

ans = Math.max(ans, visited.size());

// Early exit if all bombs can be detonated

if (ans == n) return ans;

}

return ans;

}

// Helper method for DFS traversal

private void dfs(List<Integer>[] graph, int u, Set<Integer> visited) {

for (int v : graph[u]) {

if (!visited.contains(v)) {

visited.add(v);

dfs(graph, v, visited);

}

}

}

// Helper method to calculate squared distance between two bombs

private long squaredDist(int[][] bombs, int i, int j) {

long dx = bombs[i][0] - bombs[j][0];

long dy = bombs[i][1] - bombs[j][1];

return dx \* dx + dy \* dy;

}

}

**🔍 How the Solution Work**s

The solution works by first building a directed graph where each node represents a bomb, and an edge from bomb i to bomb j exists if bomb i can detonate bomb j.

**Graph Construction**: For each pair of bombs (i, j), calculate the squared Euclidean distance and compare it to the square of bomb i's radius. Using squared values avoids computationally expensive square root calculations.

**Simulate Chain Reactions**: For each bomb as the starting point, perform a DFS to traverse all reachable bombs in the graph. The DFS naturally simulates the chain reaction of detonations.

**Track Maximum**: Keep track of the maximum number of bombs detonated from any single starting point. An early exit occurs if a chain reaction detonates all bombs.

**💡 Complexity Analysi**s

**Time Complexity**: **O(n³)** in the worst case. This accounts for O(n²) to build the graph and O(n²) for DFS from each of the n starting points.

**Space Complexity**: **O(n²)** for storing the adjacency list.

**📝 Key Points to Remembe**r

**Directed Edges**: The graph is directed because bomb A triggering bomb B does not guarantee that bomb B can trigger bomb A.

**Squared Distance**: Using ri \* ri >= (x1-x2)\*(x1-x2) + (y1-y2)\*(y1-y2) is equivalent to checking the Euclidean distance without the performance cost of Math.sqrt().

**Early Termination**: If during any DFS traversal you find that all bombs can be detonated, you can return immediately.

I hope this detailed explanation helps you understand the solution. Would you like me to explain any part in more detail?

**I see there is another DFS style solution, also looks good ?**

**Refer to**

<https://leetcode.com/problems/detonate-the-maximum-bombs/solutions/1623988/neat-code-java-dfs/>

public int maximumDetonation(int[][] bombs) {

int n = bombs.length, ans = 0;

for (int i = 0; i < n; i++) {

ans = Math.max(ans, dfs(i, new boolean[n], bombs));

}

return ans;

}

private int dfs(int idx, boolean[] v, int[][] bombs) {

int count = 1;

v[idx] = true;

int n = bombs.length;

for (int i = 0; i < n; i++) {

if (!v[i] && inRange(bombs[idx], bombs[i])) {

count += dfs(i, v, bombs);

}

}

return count;

}

private boolean inRange(int[] a, int[] b) {

long dx = a[0] - b[0], dy = a[1] - b[1], r = a[2];

return dx \* dx + dy \* dy <= r \* r;

}

Yes, this DFS solution is a good and correct approach for the problem. It efficiently models the bombs and their chain reactions as a directed graph and uses Depth-First Search to explore the maximum number of detonations from each starting point.

**🔍 Analysis of the DFS Solutio**n

The provided code works as follows:

**Graph Representation**: Each bomb is treated as a node. A directed edge exists from bomb i to bomb j if bomb j is within the explosion radius of bomb i. The inRange method checks this using the squared distance to avoid expensive square root calculations.

**Core Logic**: For each bomb i, initiate a DFS to simulate the chain reaction. A visited array prevents counting the same bomb multiple times in a single chain.

**Result Calculation**: The maximum count from all starting bombs is the answer.

**📝 Comparison with BFS and Key Consideration**s

While the DFS approach is intuitive, here's a comparison with BFS:

|  |  |  |
| --- | --- | --- |
| Feature | DFS Approach | BFS Approach |
| **Algorithm** | Depth-First Search | Breadth-First Search |
| **Data Structure** | Recursion call stack | Explicit Queue |
| **Traversal** | Explores one chain fully first | Explores bombs level by level |

**Key Considerations**:

**Directed Edges**: The relationship is not symmetric. Bomb A can trigger Bomb B, but not vice versa, unless Bomb B's radius is large enough.

**Early Termination**: If a DFS ever returns n (the total number of bombs), you can return immediately as it's the maximum possible.

**💡 Suggested Code Improvemen**t

The solution is correct. For absolute clarity in an interview, you could explicitly state that you are building the graph implicitly by checking all bombs during the DFS, rather than using a pre-built adjacency list.

Would you like me to explain the BFS version for comparison?

**Refer to Kimi**

Here is a concise and efficient Java solution for LeetCode 2101 “Detonate the Maximum Bombs” using an adjacency-list + DFS approach.

Time complexity: O(n³) worst-case, but with n ≤ 100 it is easily accepted.

Space complexity: O(n²) for the graph.

class Solution {

public int maximumDetonation(int[][] bombs) {

int n = bombs.length;

List<Integer>[] g = new ArrayList[n];

for (int i = 0; i < n; i++) g[i] = new ArrayList<>();

// build directed graph: i → j if bomb i can trigger bomb j

for (int i = 0; i < n; i++) {

long x1 = bombs[i][0], y1 = bombs[i][1], r1 = bombs[i][2];

long r2 = r1 \* r1;

for (int j = 0; j < n; j++) {

if (i == j) continue;

long dx = x1 - bombs[j][0], dy = y1 - bombs[j][1];

if (dx \* dx + dy \* dy <= r2) g[i].add(j);

}

}

int best = 0;

// try every starting bomb

for (int start = 0; start < n; start++) {

boolean[] vis = new boolean[n];

int cnt = dfs(g, start, vis);

best = Math.max(best, cnt);

if (best == n) return n; // early exit

}

return best;

}

// returns number of bombs reachable from u

private int dfs(List<Integer>[] g, int u, boolean[] vis) {

vis[u] = true;

int res = 1;

for (int v : g[u]) {

if (!vis[v]) res += dfs(g, v, vis);

}

return res;

}

}

The algorithm:

Constructs a directed graph where an edge i → j exists iff bomb i can directly trigger bomb j (distance² ≤ radius²).

For every bomb start, runs DFS to count how many distinct bombs are reachable (i.e., will explode in the chain reaction).

Keeps the maximum count across all starting bombs.

This passes all LeetCode tests and is identical to the reference implementation in .

**Refer to**

[L529.Minesweeper](note://WEB215358cdf49fb20169eb967a127844e7)

[L547.Number of Provinces (Friend Circles) (Ref.L323,L2421)](note://55FAE6CE6252461BA88F8CDE05E94F2E)

[L695.Max Area of Island (Ref.L200,L1568)](note://WEBe10a2b6bab245be1e94ccd874af71ec6)

[L994.Rotting Oranges (Ref.286)](note://2EAD142D528B482F821654DFDE49CC4D)