```
var areaOfMaxDiagonal = function (dimensions) {
  if (Array.isArray(dimensions) && dimensions.length === 1 && Array.isArray(dimensions[0]) && Ar
ray.isArray(dimensions[0][0])) {
    dimensions = dimensions[0];
  }

  let maxD2 = -1;
  let bestArea = 0;

  for (const [w, h] of dimensions) {
    const d2 = w * w + h * h;
    const area = w * h;

    if (d2 > maxD2 || (d2 === maxD2 && area > bestArea)) {
      maxD2 = d2;
      bestArea = area;
    }
  }
  return bestArea;
};
```

```
console.log(
  areaOfMaxDiagonal([
    [
      [4, 7], [8, 9], [5, 3], [6, 10], [2, 9], [3, 10], [2, 2], [5, 8], [5, 10],
      [5, 6], [8, 9], [10, 7], [8, 9], [3, 7], [2, 6], [5, 1], [7, 4], [1, 10],
      [1, 7], [6, 9], [3, 3], [4, 6], [8, 2], [10, 6], [7, 9], [9, 2], [1, 2],
      [3, 8], [10, 2], [4, 1], [9, 7], [10, 3], [6, 9], [9, 8], [7, 7], [5, 7],
      [5, 4], [6, 5], [1, 8], [2, 3], [7, 10], [3, 9], [5, 7], [2, 4], [5, 6],
      [9, 5], [8, 8], [8, 10], [6, 8], [5, 1], [10, 8], [7, 4], [2, 1], [2, 7],
      [10, 3], [2, 5], [7, 6], [10, 5], [10, 9], [5, 7], [10, 6], [4, 3], [10, 4],
      [1, 5], [8, 9], [3, 1], [2, 5], [9, 10], [6, 6], [5, 10], [10, 2], [6, 10],
      [1, 1], [8, 6], [1, 7], [6, 3], [9, 3], [1, 4], [1, 1], [10, 4], [7, 9],
      [4, 5], [2, 8], [7, 9], [7, 3], [4, 9], [2, 8], [4, 6], [9, 1], [8, 4],
      [2, 4], [7, 8], [3, 5], [7, 6], [8, 6], [4, 7],
      [25, 60], [39, 52], [16, 63], [33, 56]
    ]
  ])
); // 2028
```

Maximum Area of Longest Diagonal Rectangle - Deep Concept Analysis

Problem Understanding & Mathematical Foundation

The Core Problem

Given multiple rectangles with dimensions [width, height], find the rectangle with the **longest diagonal**. If multiple rectangles have the same longest diagonal, return the one with the **maximum area**.

Why Diagonals Matter

A rectangle's diagonal represents the **maximum distance** between any two points within that rectangle. In many real-world scenarios (shipping boxes, display screens, etc.), the diagonal is a critical constraint.

Deep Dive into Key Concepts

1. Pythagorean Theorem - The Mathematical Foundation

Basic Formula

For a rectangle with width `w` and height `h`:

- Diagonal length = $\sqrt{(w^2 + h^2)}$

Why We Use d^2 Instead of d

```
// Instead of this (prone to floating point errors):
const diagonal = Math.sqrt(w * w + h * h);

// We use this (exact integer comparison):
const d2 = w * w + h * h;
```

Reasoning:

- **Floating Point Issues**: √25 might not exactly equal √25 due to precision
- **Performance**: Avoiding square root calculation saves computation
- **Accuracy**: Integer comparison (25 vs 26) is always exact
- **Mathematical Validity**: If $d_1^2 > d_2^2$, then $d_1 > d_2$ (monotonic property)

Mathematical Proof of Approach

```
If d₁² > d₂², then d₁ > d₂
Proof:
- d₁, d₂ > 0 (diagonals are positive)
- If d₁² > d₂², taking square root of both sides preserves inequality
- Therefore d₁ > d₂
```

2. Greedy Algorithm Pattern - The Strategic Approach

What Makes This Greedy?

A greedy algorithm makes locally optimal choices at each step, hoping to find a global optimum.

Why Greedy Works Here

```
// At each step, we ask: "Is this the best rectangle I've seen so far?"
if (d2 > maxD2 || (d2 === maxD2 && area > bestArea)) {
    // YES: Update our "best so far"
    maxD2 = d2;
    bestArea = area;
}
```

Key Properties:

- **Optimal Substructure**: The best rectangle overall is the best among all rectangles
- **No Backtracking Needed**: Once we find a better rectangle, we don't need to reconsider previous ones

- **Single Pass Efficiency**: We only need to see each rectangle once

Greedy Choice Property

At each rectangle, we make the greedy choice: "Keep this if it's better than what I have." This local choice leads to the global optimum because:

1. We're looking for a single maximum value
2. The comparison criteria are transitive (if A > B and B > C, then A > C)

3. Multi-Criteria Decision Making - The Tie-Breaking Logic

The Decision Hierarchy

```
Primary Criterion: Diagonal Length (d²)
    ↓ (if tied)
Secondary Criterion: Area (w × h)
```

Why This Order Matters

1. **Problem Requirements**: "Longest diagonal" is the primary goal
2. **Tie Resolution**: Among equal diagonals, maximum area provides a meaningful secondary criterion
3. **Deterministic Results**: Ensures consistent output for the same input

The Logic Breakdown

```javascript
if (d2 > maxD2 || (d2 === maxD2 && area > bestArea))
```

Case Analysis:

- `d2 > maxD2`: Found a longer diagonal → **Always update**
- `d2 < maxD2`: Found a shorter diagonal → **Never update**
- `d2 === maxD2 && area > bestArea`: Same diagonal, larger area → **Update**
- `d2 === maxD2 && area <= bestArea`: Same diagonal, smaller/equal area → **Don't update**

4. Array Destructuring & Modern JavaScript Patterns

Destructuring Assignment

```javascript
const [w, h] = dimensions[i];
```

Deep Explanation:

- **Pattern Matching**: Automatically extracts array elements into variables
- **Readability**: `w` and `h` are more meaningful than `dimensions[i][0]` and `dimensions[i][1]`
- **Performance**: Modern JavaScript engines optimize destructuring well
- **Error Prevention**: Clear variable names reduce mistakes

5. Defensive Programming - Handling Edge Cases

The Extra Nesting Check

```
if (dimensions.length === 1 && Array.isArray(dimensions[0][0])) {
    dimensions = dimensions[0];
}
```

Why This Matters:

- **API Flexibility**: Handles both `[[w,h], [w,h]]` and `[[[w,h], [w,h]]]`
- **Error Prevention**: Avoids crashes from unexpected input format
- **Graceful Degradation**: Code works even with slightly malformed input

Algorithm Complexity Analysis

Time Complexity: O(n)

- **Single Loop**: We iterate through each rectangle exactly once
- **Constant Operations**: Each iteration performs O(1) operations
- **No Nested Loops**: No sorting or searching required
- **Optimal for Problem**: We must examine every rectangle at least once

Space Complexity: O(1)

- **Fixed Variables**: Only `maxD2`, `bestArea`, `w`, `h`, `d2`, `area`
- **No Additional Data Structures**: No arrays, objects, or recursion stack
- **In-Place Processing**: We don't create copies of the input

Real-World Applications & Extensions

Practical Scenarios

1. **Package Shipping**: Finding the largest box that fits through a diagonal constraint
2. **Screen Manufacturing**: Optimizing screen size within diagonal limits
3. **Architecture**: Maximizing room area within diagonal building constraints

Possible Extensions

1. **Multiple Criteria**: Adding weight, cost, or other factors
2. **Tolerance Ranges**: "Approximately equal" diagonals within epsilon
3. **3D Version**: Extending to boxes with space diagonals

Common Pitfalls & How the Code Avoids Them

1. Floating Point Precision

Problem: `Math.sqrt(25) === 5.000000001` on some systems **Solution**: Compare `d²` values (integers) instead

2. Tie-Breaking Ambiguity

Problem: What if diagonals are equal? **Solution**: Clear secondary criterion (maximum area)

3. Input Format Assumptions

Problem: Code breaks with unexpected nesting **Solution**: Defensive check for extra nesting

4. Initialization Issues

Problem: Starting with `maxD2 = 0` fails for all negative dimensions **Solution**: `maxD2 = -1` ensures first rectangle is always considered

Mathematical Insights

Why Area as Tie-Breaker Makes Sense

Given two rectangles with the same diagonal:

- Rectangle 1: 3×4 (diagonal^2 = 25, area = 12)
- Rectangle 2: 5×0 (diagonal^2 = 25, area = 0)

Both have the same diagonal, but Rectangle 1 is clearly more useful in practical applications.

The Optimization Landscape

This problem has a **discrete optimization** nature:

- **Feasible Solutions**: All given rectangles
- **Objective Function**: Lexicographic ordering (diagonal first, area second)
- **Global Optimum**: The single best rectangle according to our criteria

The greedy approach works because we're selecting from a finite, pre-defined set rather than constructing a solution from scratch.

https://leetcode.com/problems/maximum-area-of-longest-diagonal-rectangle/