Falling Squares

On an infinite number line (x-axis), we drop given squares in the order they are given.

The `i` -th square dropped (`positions[i] = (left, side_length)`) is a square with the left-most point being `positions[i][0]` and sidelength `positions[i][1]` .

The square is dropped with the bottom edge parallel to the number line, and from a higher height than all currently landed squares. We wait for each square to stick before dropping the next.

The squares are infinitely sticky on their bottom edge, and will remain fixed to any positive length surface they touch (either the number line or another square). Squares dropped adjacent to each other will not stick together prematurely.

Return a list `ans` of heights. Each height `ans[i]` represents the current highest height of any square we have dropped, after dropping squares represented by `positions[0], positions[1], ..., positions[i]` .

Example 1:

Input: [[1, 2], [2, 3], [6, 1]]
Output: [2, 5, 5]
Explanation:

After the first drop of

```
positions[0] = [1, 2]:
_aa
_aa
-------
```

The maximum height of any square is 2.

After the second drop of

```
positions[1] = [2, 3]:
__aaa
__aaa
__aaa
_aa__
_aa__
--------------
```

The maximum height of any square is 5.
The larger square stays on top of the smaller square despite where its center of gravity is, because squares are infinitely sticky on their bottom edge.

After the third drop of

```
positions[1] = [6, 1]:
__aaa
__aaa
__aaa
_aa
_aa___a
--------------
```

The maximum height of any square is still 5.

Thus, we return an answer of

```
[2, 5, 5]
```

.

Example 2:

Input: [[100, 100], [200, 100]]
Output: [100, 100]
Explanation: Adjacent squares don't get stuck prematurely - only their bottom edge ca n stick to surfaces.

Note:

- `1 .`
- `1 .`
- `1 .`

Solution 1

The idea is quite simple, we use intervals to represent the square. the initial height we set to the square `cur` is `pos[1]`. That means we assume that all the square will fall down to the land. we iterate the previous squares, check is there any square `i` beneath my `cur` square. If we found that we have squares `i` intersect with us, which means my current square will go above to that square `i`. My target is to find the highest square and put square `cur` onto square `i`, and set the height of the square `cur` as

```
cur.height = cur.height + previousMaxHeight;
```

Actually, you do not need to use the interval class to be faster, I just use it to make my code cleaner

```java
class Solution {
    private class Interval {
        int start, end, height;
        public Interval(int start, int end, int height) {
            this.start = start;
            this.end = end;
            this.height = height;
        }
    }
    public List<Integer> fallingSquares(int[][] positions) {
        List<Interval> intervals = new ArrayList<>();
        List<Integer> res = new ArrayList<>();
        int h = 0;
        for (int[] pos : positions) {
            Interval cur = new Interval(pos[0], pos[0] + pos[1] - 1, pos[1]);
            h = Math.max(h, getHeight(intervals, cur));
            res.add(h);
        }
        return res;
    }
    private int getHeight(List<Interval> intervals, Interval cur) {
        int preMaxHeight = 0;
        for (Interval i : intervals) {
            // Interval i does not intersect with cur
            if (i.end < cur.start) continue;
            if (i.start > cur.end) continue;
            // find the max height beneath cur
            preMaxHeight = Math.max(preMaxHeight, i.height);
        }
        cur.height += preMaxHeight;
        intervals.add(cur);
        return cur.height;
    }
}
```

written by leuction original link here

Solution 2

The running time complexity should be $O(n^2)$, since the bottleneck is given by finding the maxHeight in certain range.
Idea is simple, we use a map, and `map[i] = h` means, from `i` to next adjacent x-coordinate, inside this range, the height is `h`.
To handle edge cases like adjacent squares, I applied STL functions, for left bound, we call `upper_bound` while for right bound, we call `lower_bound`.
Special thanks to @storypku for pointing out bad test cases like [[1,2],[2,3],[6,1],[3,3],[6,20]]

```
    vector<int> fallingSquares(vector<pair<int, int>>& positions) {
        map<int,int> mp = {{0,0}, {INT_MAX,0}};
        vector<int> res;
        int cur = 0;
        for(auto &p : positions){
            int l = p.first, r = p.first + p.second, h = p.second, maxH = 0;
            auto ptri = mp.upper_bound(l), ptrj = mp.lower_bound(r);        // find
range
            int tmp = ptrj->first == r? ptrj->second : (--ptrj)++->second;  // tmp
will be applied by new right bound
            for(auto i = --ptri; i != ptrj; ++i)
                maxH = max(maxH, i->second);                                // find
biggest height
            mp.erase(++ptri, ptrj);                                         // era
se range
            mp[l] = h+maxH;                                                 // new
left bound
            mp[r] = tmp;                                                    // new
right bound
            cur = max(cur, mp[l]);
            res.push_back(cur);
        }
        return res;
    }
```

written by JadenPan original link here

Solution 3

use `sq` to store information of squares. `sq[i][0]` is left edge, `sq[i][1]` is length, `sq[i][2]` is highest point. Everytime a new square falls, check if it has overlap with previous `sq`, if so, add the largest `sq[i][2]` to its height.

```python
class Solution(object):
    def fallingSquares(self, positions):
        """
        :type positions: List[List[int]]
        :rtype: List[int]
        """
        if not positions:
            return []
        sq = [[positions[0][0], positions[0][1], positions[0][1]]]
        result = [positions[0][1]]
        max_h = result[0]
        for pos in positions[1: ]:
            h = pos[1]
            l = pos[0]
            add = 0
            for prev in sq:
                if not l >= prev[0] + prev[1] and not l + h <= prev[0]:
                    add = max(add, prev[2])
            sq.append([l, h, h + add])
            max_h = max(max_h, h + add)
            result.append(max_h)
        return result
```

written by BigBiang original link here