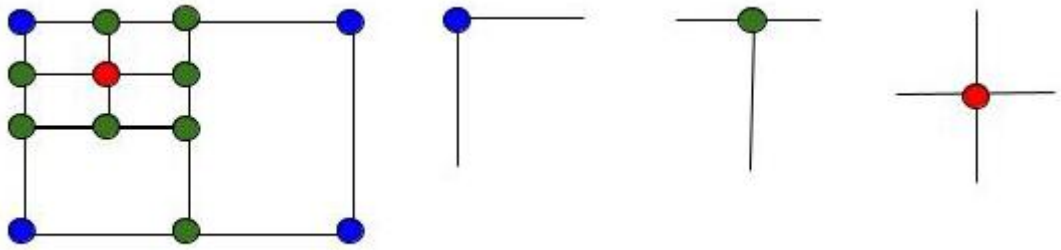
O(n) solution by counting corners with detailed explaination

Last Edit: Mar 6, 2018, 9:51 PM

(/hxtang) hxtang (/hxtang) ★ 98

This is an expanded version of my earlier post (https://discuss.leetcode.com/topic/55874/o-log-n-problem-2-and-o-n-problem-3-solution/3) under the contest discussion board.
The following code passes through not only the OJ but also various test cases others have pointed out.

Idea



Consider how the corners of all rectangles appear in the large rectangle if there's a perfect rectangular cover.

Rule 1: The local shape of the corner has to follow one of the three following patterns

- Corner of the large rectangle (blue): it occurs only once among all rectangles
- T-junctions (green): it occurs twice among all rectangles
- Cross (red): it occurs four times among all rectangles

Rule 2: A point can only be the top-left corner of at most one sub-rectangle. Similarly it can be the top-right/bottom-left/bottom-right corner of at most one sub-rectangle. Otherwise overlaps occur.

Proof of correctness

Obviously, any perfect cover satisfies the above rules. So the main question is whether there exists an input which satisfy the above rules, yet does not compose a rectangle.

First, ***any overlap is not allowed based on the above rules*** because

- aligned overlap like [[0, 0, 1, 1], [0, 0, 2, 2]] are rejected by Rule 2.
- unaligned overlap will generate a corner in the interior of another sub-rectangle, so it will be rejected by Rule 1.

Second, consider the shape of boundary for the combined shape. The cross pattern does not create boundary. The corner pattern generates a straight angle on the boundary, and the T-junction generates a straight border.
So the shape of the union of rectangles has to be rectangle(s).

Finally, if there are more than two non-overlapping rectangles, at least 8 corners will be found, and cannot be matched to the 4 bounding box corners (be reminded we have shown that there is no chance of overlapping).
So the cover has to be a single rectangle if all above rules are satisfied.

Algorithm

- **Step1:** Based on the above idea we maintain a mapping from (x, y)->mask by scanning the sub-rectangles from beginning to end.
    - (x, y) corresponds to corners of sub-rectangles
    - mask is a 4-bit binary mask. Each bit indicates whether there have been a sub-rectangle with a top-left/top-right/bottom-left/bottom-right corner at (x, y). If we see a conflict while updating mask, it suffice to return a false during the scan.
    - In the meantime we obtain the bounding box of all rectangles (which potentially be the rectangle cover) by getting the upper/lower bound of x/y values.
- **Step 2:** Once the scan is done, we can just browse through the unordered_map to check the whether ***the mask corresponds to a T junction / cross, or corner if it is indeed a bounding box corner***.
  (note: my earlier implementation uses counts of bits in mask to justify corners, and this would not work with certain cases as @StefanPochmann points out).

Complexity

The scan in step 1 is O(n) because it loop through rectangles and inside the loop it updates bounding box and unordered_map in O(1) time.

Step2 visits 1 corner at a time with O(1) computations for at most 4n corners (actually much less because either corner overlap or early stopping occurs). So it's also O(n).

```
// pos encoding: 1 - TL 2- TR 4- BL 8-BR
// return false if a conflict in mask occurs (i.e. there used to be a rectangle witl
inline bool insert_corner(unordered_map<int, unordered_map<int, int>>& corner_count,
    int& m = corner_count[x][y];
    if (m & pos) return false;
    m |= pos;
    return true;
}

bool isRectangleCover(vector<vector<int>>& rectangles) {
    // step 1: counting
    unordered_map<int, unordered_map<int, int>> corner_count;
    int minx = INT_MAX, maxx=INT_MIN, miny=INT_MAX, maxy=INT_MIN;
    for (auto& rect : rectangles) {
        minx = min(minx, rect[0]);
        maxx = max(maxx, rect[2]);
        miny = min(miny, rect[1]);
        maxy = max(maxy, rect[3]);
        if (!insert_corner(corner_count, rect[0], rect[1], 1)) return false;
        if (!insert_corner(corner_count, rect[2], rect[1], 2)) return false;
        if (!insert_corner(corner_count, rect[0], rect[3], 4)) return false;
        if (!insert_corner(corner_count, rect[2], rect[3], 8)) return false;
    }

    //step2: checking
    bool valid_corner[16] = {false};
    bool valid_interior[16] = {false};
    valid_corner[1] = valid_corner[2] = valid_corner[4] = valid_corner[8] = true;
    valid_interior[3] = valid_interior[5] = valid_interior[10] = valid_interior[12]

    for (auto itx = corner_count.begin(); itx != corner_count.end(); ++itx) {
        int x = itx->first;
        for (auto ity = itx->second.begin(); ity != itx->second.end(); ++ity) {
            int y = ity->first;
            int mask = ity->second;
            if (((x != minx && x != maxx) || (y != miny && y != maxy)) && !valid_int
                return false;
        }
    }
    return true;
}
```

The above code may be refined by changing the 2D unordered_map to 1D. But such improvements has no effect on complexity.

```
struct pairhash {//double hash function for pair key
public:
    template <typename T, typename U>
    size_t operator()(const pair<T, U> &rhs) const {
        size_t l = hash<T>()(rhs.first);
        size_t r = hash<U>()(rhs.second);
        return l + 0x9e3779b9 + (r << 6) + (r >> 2);
    }
};

bool isRectangleCover(vector<vector<int>>& rectangles) {
    // step 1: counting
    unordered_map<pair<int, int>, int, pairhash> corner_count;
    int minx = INT_MAX, maxx=INT_MIN, miny=INT_MAX, maxy=INT_MIN;
    for (auto& rect : rectangles) {
        minx = min(minx, rect[0]);
        maxx = max(maxx, rect[2]);
        miny = min(miny, rect[1]);
        maxy = max(maxy, rect[3]);

        int& m1 = corner_count[make_pair(rect[0], rect[1])];
        if (m1 & 1) return false; else m1 |= 1;
        int& m2 = corner_count[make_pair(rect[2], rect[1])];
        if (m2 & 2) return false; else m2 |= 2;
        int& m3 = corner_count[make_pair(rect[0], rect[3])];
        if (m3 & 4) return false; else m3 |= 4;
        int& m4 = corner_count[make_pair(rect[2], rect[3])];
        if (m4 & 8) return false; else m4 |= 8;
    }

    //step2: checking
    for (const auto& kv: corner_count) {
        pair<int, int> pos; int mask;
        tie(pos, mask) = kv;
        if ((pos.first != minx && pos.first != maxx) || (pos.second != miny && pos.s
            if (mask != 3 && mask != 5 && mask != 10 && mask != 12 && mask != 15) re
        }
    }
    return true;
}
```

Notes

Comments: 19

Sort By

Type comment here... (Markdown is supported)

Preview | Post

TurtleShip (/turtleship) ★ 4 Aug 28, 2016, 4:32 PM

Thanks for sharing your solution!
I wrote your solution in Java, based on your explanation. I submitted my solution and got accepted. Mine is a bit verbose, but it helped me understand your approach better (I read your actual implementation after I wrote mine; your looks more succinct :D)

Read More

4 Share Reply

StefanPochmann (/stefanpochmann) ★ 20750 Aug 28, 2016, 7:23 AM

It's wrong, fails for example `[[0,0,1,1],[0,2,1,3],[1,1,2,2],[2,0,3,1],[2,2,3,3],[1,0,2,3],[0,1,3,2]]`.
Update: Got fixed, doesn't fail that anymore. I'll try to find a new counterexample :-)

4 Share Reply

SHOW 9 REPLIES

DonaldTrump (/donaldtrump) ★ 412 Sep 25, 2016, 6:32 PM

@hxtang Nice solution. Would you mind sharing how did you come up with the algorithm? What characteristic of the problem gave you the hint of coming up counting the corners? Thanks in advance.

2 Share Reply

orbuluh (/orbuluh) ★ 556 Sep 7, 2016, 12:32 AM

@hxtang Dude, this solution is INSANE, thanks for sharing.
Just put some of my notes fyi and the corresponding python code here.

- Use 1, 2, 4, 8 to represent four corners for each retangles respectively.
  - say BotLeft(bx, by) as 1, BotRight(tx, by) as 2, TopLeft(bx, ty) as 4, TopRight(tx, ty) as 8
- Then the "T" intersection only have four situations

Read More

1 Share Reply

ricky_xuchen (/ricky_xuchen) ★ 0 Nov 10, 2017, 10:14 AM

Re: O(n) solution by counting corners with detailed explaination (/topic/55923/o-n-solution-by-counting-corners-with-detailed-explaination)
Great explanation! Really enjoyed it.
Here are some of my thoughts:
While your rule 2 is definitely valid! Is there an easier way to verify it in implementation? For example, instead of

Read More

0 Share Reply

laughting (/laughting) ★ 0 Dec 21, 2016, 6:41 AM

@hxtang Thanks for your good idea. But I think Rule 1 is not strong enough, because the two methods in step2 meet Rule 1, but the method of using counts of bits in mask is wrong. You probably should update Rule 1 to make it more detail.

0 Share Reply

yu.xiaoxixi (/yuxiaoxixi) ★ 5 Dec 15, 2016, 6:14 PM

Does anybody know how to do it if overlap(example 4) is allowed? It's a Google phone interview question, I can't figure it out…

0 Share Reply

sano1101 (/sano1101) ★ 0 Sep 6, 2016, 10:13 PM

Did anyone think of using a Region QuadTree? Seems like it would make sense here, but I've never built one to know for sure if it would work.

0 Share Reply

soamaaazing (/soamaaazing) ★ 158 Sep 5, 2016, 11:42 AM

Really really smart solution!
I just want to point out that one of the last check may be redundant. It's because we already know the corner-point only has one sub-rectangle, else it won't be a corner-point.:

```
if (mask != 1 && mask != 2 && mask != 4 && mask != 8) return false;
```

Read More

0 Share Reply

zhiqing_xiao (/zhiqing_xiao) ★ 1203 Sep 3, 2016, 8:25 PM

@hxtang
That is smart! And thank you for your detailed explanations.
On the code:

```
pair<int, int> pos;
```

Read More

0 Share Reply

‹ 1 2 ›