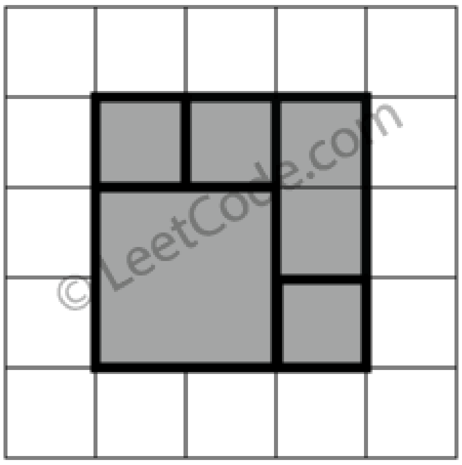
Perfect Rectangle

Given N axis-aligned rectangles where N > 0, determine if they all together form an exact cover of a rectangular region.

Each rectangle is represented as a bottom-left point and a top-right point. For example, a unit square is represented as [1,1,2,2]. (coordinate of bottom-left point is (1, 1) and top-right point is (2, 2)).

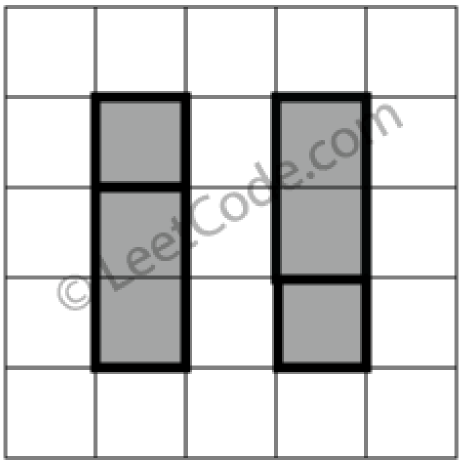
Example 1:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [3,2,4,4],
  [1,3,2,4],
  [2,3,3,4]
]

Return true. All 5 rectangles together form an exac
t cover of a rectangular region.
```



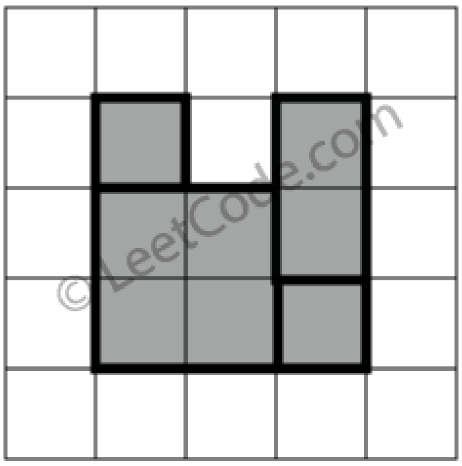
Example 2:

```
rectangles = [
  [1,1,2,3],
  [1,3,2,4],
  [3,1,4,2],
  [3,2,4,4]
]

Return false. Because there is a gap between the tw
o rectangular regions.
```



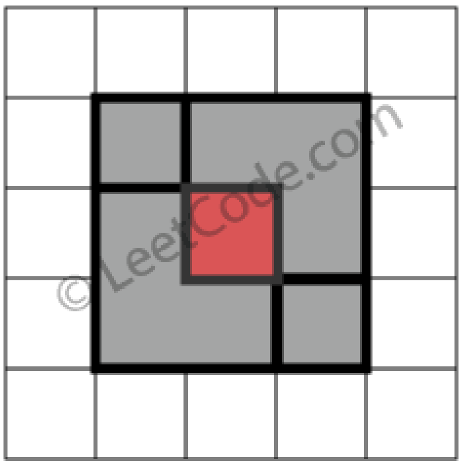
Example 3:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [3,2,4,4]
]

Return false. Because there is a gap in the top cen
ter.
```



Example 4:

```
rectangles = [
  [1,1,3,3],
  [3,1,4,2],
  [1,3,2,4],
  [2,2,4,4]
]

Return false. Because two of the rectangles overlap
 with each other.
```



Solution 1

This is an expanded version of my earlier post 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).

```cpp
// 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 wi
th corner (x, y) at pos
inline bool insert_corner(unordered_map<int, unordered_map<int, int>>& corner_cou
nt, int x, int y, int pos) {
    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[1
2] = valid_interior[15] = true;

    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_i
nterior[mask])
                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
.second != maxy)) {
            if (mask != 3 && mask != 5 && mask != 10 && mask != 12 && mask != 15)
return false;
        }
    }
    return true;
}
```

written by hxtang original link here

Solution 2

The right answer must satisfy two conditions:

1. the large rectangle area should be equal to the sum of small rectangles
2. count of all the points should be even, and that of all the four corner points should be one

'''
public boolean isRectangleCover(int[][] rectangles) {

```
    if (rectangles.length == 0 || rectangles[0].length == 0) return false;

    int x1 = Integer.MAX_VALUE;
    int x2 = Integer.MIN_VALUE;
    int y1 = Integer.MAX_VALUE;
    int y2 = Integer.MIN_VALUE;

    HashSet<String> set = new HashSet<String>();
    int area = 0;

    for (int[] rect : rectangles) {
        x1 = Math.min(rect[0], x1);
        y1 = Math.min(rect[1], y1);
        x2 = Math.max(rect[2], x2);
        y2 = Math.max(rect[3], y2);

        area += (rect[2] - rect[0]) * (rect[3] - rect[1]);

        String s1 = rect[0] + " " + rect[1];
        String s2 = rect[0] + " " + rect[3];
        String s3 = rect[2] + " " + rect[3];
        String s4 = rect[2] + " " + rect[1];

        if (set.contains(s1)) {
            set.remove(s1);
        } else {
            set.add(s1);
        }
        if (set.contains(s2)) {
            set.remove(s2);
        } else {
            set.add(s2);
        }
        if (set.contains(s3)) {
            set.remove(s3);
        } else {
            set.add(s3);
        }
        if (set.contains(s4)) {
            set.remove(s4);
        } else {
            set.add(s4);
        }
    }

    if (!set.contains(x1 + " " + y1) || !set.contains(x1 + " " + y2) || !set.cont
ains(x2 + " " + y1) || !set.contains(x2 + " " + y2) || set.size() != 4) return fa
lse;

    return area == (x2-x1) * (y2-y1);
}
```

''''

written by hu19 original link here

Solution 3

Standard sweep line solution.
Basic idea:
Sort by x-coordinate.
Insert y-interval into TreeSet, and check if there are intersections.
Delete y-interval.

```
public class Event implements Comparable<Event> {
 int time;
 int[] rect;

 public Event(int time, int[] rect) {
  this.time = time;
  this.rect = rect;
 }

 public int compareTo(Event that) {
  if (this.time != that.time) return this.time - that.time;
  else return this.rect[0] - that.rect[0];
 }
}

public boolean isRectangleCover(int[][] rectangles) {
 PriorityQueue<Event> pq = new PriorityQueue<Event> ();
        // border of y-intervals
 int[] border= {Integer.MAX_VALUE, Integer.MIN_VALUE};
 for (int[] rect : rectangles) {
  Event e1 = new Event(rect[0], rect);
  Event e2 = new Event(rect[2], rect);
  pq.add(e1);
  pq.add(e2);
  if (rect[1] < border[0]) border[0] = rect[1];
  if (rect[3] > border[1]) border[1] = rect[3];
 }
 TreeSet<int[]> set = new TreeSet<int[]> (new Comparator<int[]> () {
  @Override
                // if two y-intervals intersects, return 0
  public int compare (int[] rect1, int[] rect2) {
   if (rect1[3] <= rect2[1]) return -1;
   else if (rect2[3] <= rect1[1]) return 1;
   else return 0;
  }
 });
 int yRange = 0;
 while (!pq.isEmpty()) {
  int time = pq.peek().time;
  while (!pq.isEmpty() && pq.peek().time == time) {
   Event e = pq.poll();
   int[] rect = e.rect;
   if (time == rect[2]) {
    set.remove(rect);
    yRange -= rect[3] - rect[1];
   } else {
    if (!set.add(rect)) return false;
    yRange += rect[3] - rect[1];
```

```
        yRange += rect[3] - rect[1];
      }
    }
                  // check intervals' range
    if (!pq.isEmpty() && yRange != border[1] - border[0]) {
                          return false;
     //if (set.isEmpty()) return false;
     //if (yRange != border[1] - border[0]) return false;
    }
  }
  return true;
}
```

written by wddd original link here