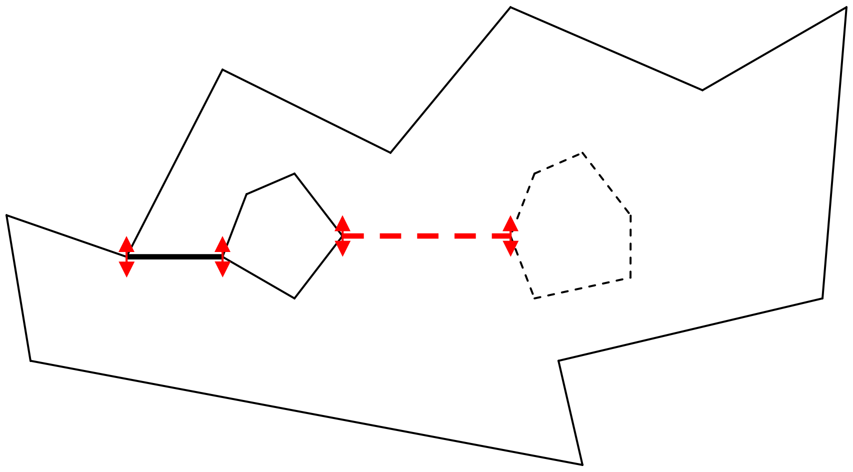
Solution for Homework 3

3.1

(1) To perform triangulation, the first step is to partition it into monotone polygons with respect to a certain direction, y-axis, for example. In this step, we can sort the "event" points and treat them as start, end, split, merge or regular vertex, regardless whether the polygon has hole or not, and can always get a set of monotone polygons at the end. Then, we just perform triangulation on these monotone polygons. Therefore, polygon with holes still admits triangulation.

(2) Suppose we are given a polygon with 1 hole, 2 holes, 3 holes, and so on, and induce from these.

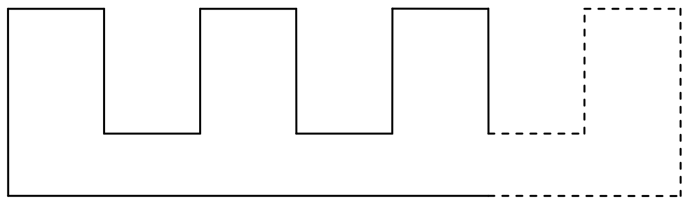


One-hole:
As we have proved it has triangulation, there must exist at least a diagonal that connect the outer vertex and inner vertex in this triangulation. Otherwise, the vertex of inner edges is not belonging to any triangle. We mark this diagonal with black bold line.
Imagine the scenario that the polygon breaks apart at this diagonal a bit (shown by red arrow), resulting in a new SIMPLE polygon with n+2 vertices, which according to Theorem. 3.1, has n+2-2=n triangles after triangulation. Apparently the original polygon with one hole also has n triangles after triangulation.
Two-hole:
If the polygon has 2 holes, its triangulation should has at least one diagonal connecting Hole-2 to Hole -1, or to the outer edge. The argument is similar to 1-hole case. Suppose the diagonal is connected to Hole 1 (shown in red bold dash line), and perform the breaking apart step again using both diagonal as boundaries. We get a new SIMPLE polygon with n+2+2 vertices, thus the polygon, as well as the old two-hole polygon, has n+2+2-2=n+2 triangles after triangulation.
…..
To induce from these analysis, we say that a complex polygon with n vertices and h holes will have

n+2*h-2 triangles all together after triangulation.

3.2

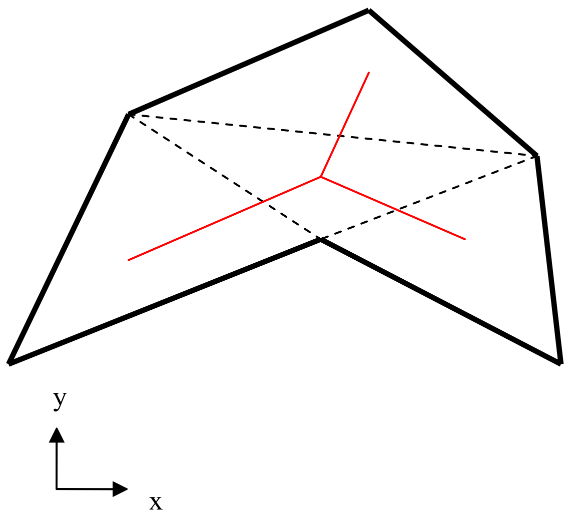
There are a lot of examples to justify that $\left\lfloor \frac{n}{4} \right\rfloor$ cameras are sometimes necessary to guard simple rectilinear polygons. The simplest case is a rectangle. Another typical case is as shown as below:



Each time we add a new corner (shown by dash line), the number of edges increases by 4, and we need a new camera to monitor it. It's easy to see the number of cameras needed is always n/4.

3.3

This argument is not true. A simple case is shown below:



This polygon (shown in black bold lines) is monotone with respect to x-axis. Its triangulation is shown by black dash lines, and the dual graph shown by read lines. It's easy to see the dual graph has degree of 3 at the center node.

3.5

Suppose we are given the dual graph of the triangulation of the simple polygon. Thus it will be a tree (for argument see the top of Page 48 of the textbook). Then we can perform a depth-first traversal on the graph and 3-color it. To traverse the whole graph and color each vertex requires $O(n)$ time all together. The pseudo-code as following:

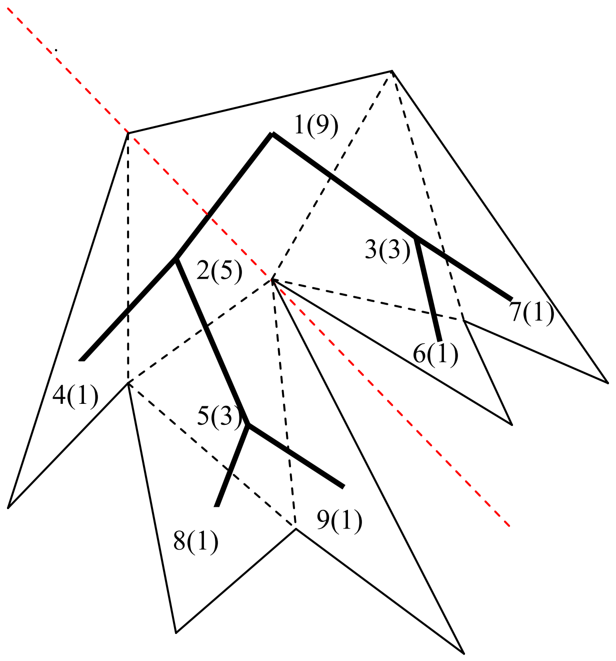
```
Input: dual graph G, each node of which associated with a triangle of the triangulation of the polygon
Output: a 3-color plan for the polygon
Initialization:
Mark each node of G as "Unvisited", and its father node as "NULL"
NodeStack=[select any one node from G];
Node = NodeStack.pop();
Traversal:
DFS_Traversal (G, Node):
    // color 3 vertices of Node, according to the color of Node's father; at beginning, Node has no
    // father, thus 3-color its vertices with red, blue, yellow arbitrarily
    Color(Node);
    Mark Node as "Visited";
    // expand nodes that are neighbors to the current one, and never visited before
    SonNodes[Degree] = Expand(G, Node);
    IF SonNodes = []:
        IF NodeStack == []:
            RETURN;
        ELSE:
            Node = NodeStack.pop();
    ELSE:
        Node = SonNodes[1];
        NodeStack.push(SonNodes[2→Degree]);
    DFS_Traversal(G, Node);
    RETURN;
```

3.6

As shown in class and textbook, to partition a simple polygon into monotone polygons, and to triangulate monotone polygons require $O(n\log n)$ time all together. Having a triangulation, we can construct the corresponding dual graph in linear time, because a simple polygon has $n-2$ triangles. Now suppose we've got the dual graph of the triangulation, and the number of nodes **m** = **n**-2. Each edge in the graph represents a diagonal of the triangulation. Instead of deciding how many vertices for split we can decide how many nodes in dual graph for split. In particular, to have at most

$\lfloor 2n/3 \rfloor + 2$ vertices, we need to split at most $\lfloor 2(m+2)/3 \rfloor + 2 - 2 = \lfloor 2(m+2)/3 \rfloor$ nodes in the dual graph.

For example, in the following polygon, n=11, m= 9, thus $\lfloor 2(m+2)/3 \rfloor = 7$. If we split at the edge between Node 1 and 2, as shown by the red line, we get 4 nodes for the left sub-polygon, and 5 nodes for the right, which is a satisfied solution



First, we can use a **dynamic program** method to calculate for each node, how many descent nodes, including the node itself, it has for its sub-tree. For the above dual tree, for example, the results are shown in parentheses beside the node number. This algorithm requires $O(m)$ time, which is shown below:

```
Input: dual graph G, in the form of a tree
Output: number of nodes (including itself) in sub-tree for each node
Initialization:
For each node, set the number of nodes at its sub-tree as 0 (Node.SubTreeNodeNum = 0);
Node = Root node of G;
Recursion:
SubTreeNode_Calc(G, Node):
    // expand descent nodes of the current one
    DescNodes[Degree] = Expand(G, Node);
    IF DescNodes = []:    // leaf node
        Node.SubTreeNodeNum = 1
```

```
ELSE:
    FOR i = 1 → Degree:
        Node.SubTreeNodeNum = 1+ SubTreeNode_Calc(G, DescNodes[i]);
RETURN
```

Then, we just perform a **depth first search** from the root node. The first time when we find a node which satisfies:

$$\text{Node.SubTreeNodeNum} \in \left[m - \lfloor 2(m+2)/3 \rfloor,\quad \lfloor 2(m+2)/3 \rfloor \right],$$

we just return it with its father node, and split the polygon at the diagonal corresponding to the edge between these two nodes. Thus we are done. The DFS of course requires $O(m)$ time.

In summarize, the algorithm requires $O(n\log n)+O(n)+O(m)+O(m)=O(n\log n)$ time.

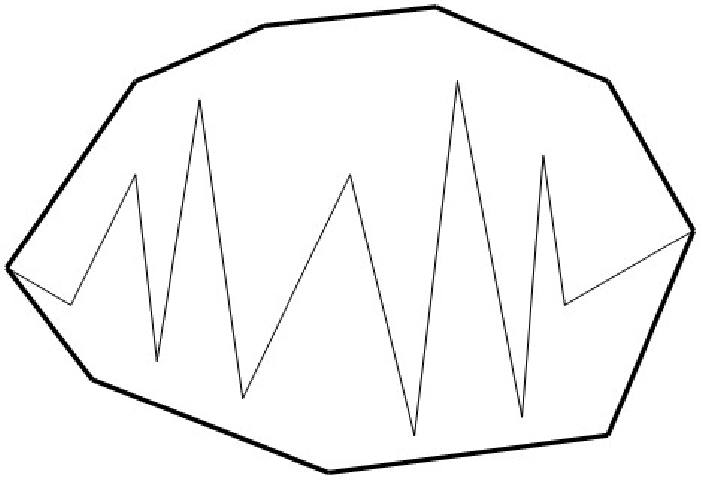
3.10

Yes, it can.

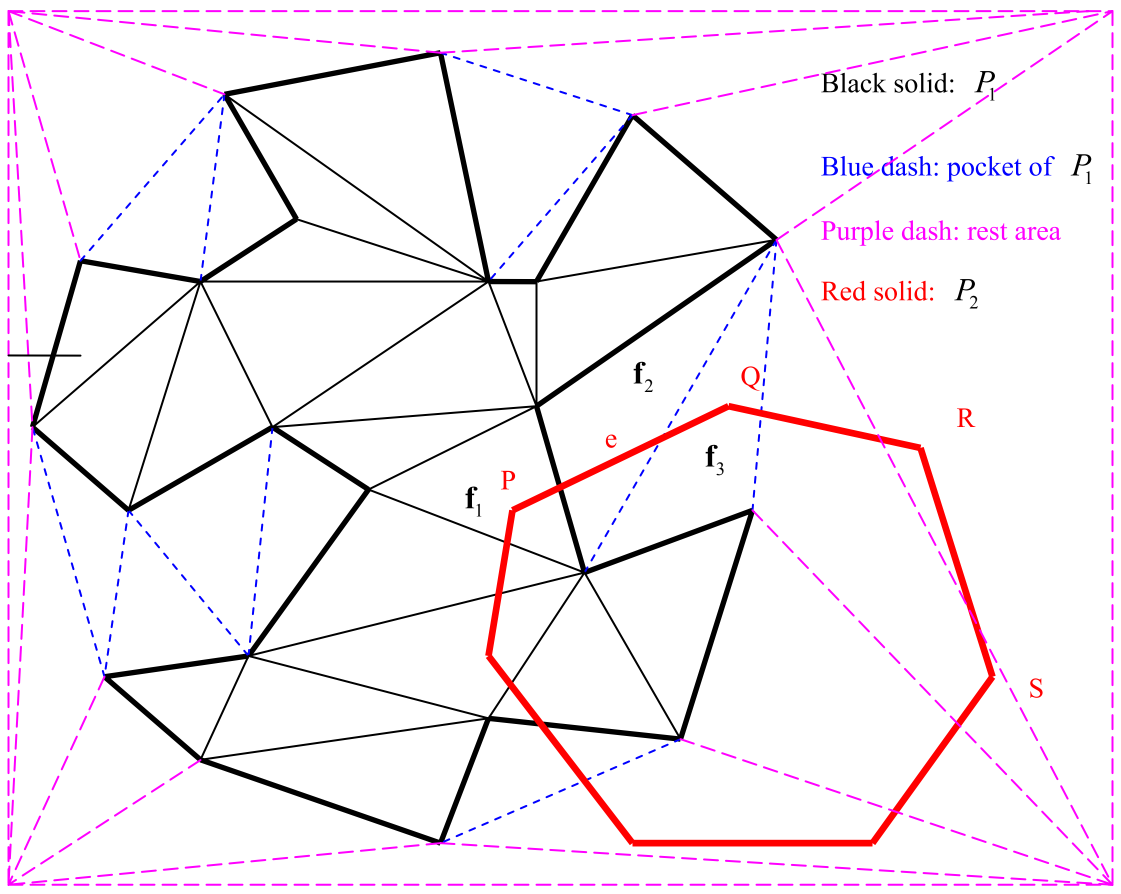
First, calculate the convex hull of the n point, which requires $O(n\log n)$ time. In this step, we already have a sorted vertices list according to x-coordinates. We can then connect the inner vertices from left to right one by one, thus split the convex hull into two monotone polygons.

Second, triangulate both monotone polygons, which can be done in $O(n)$ time.

This approach is shown by the following figure:



3.12



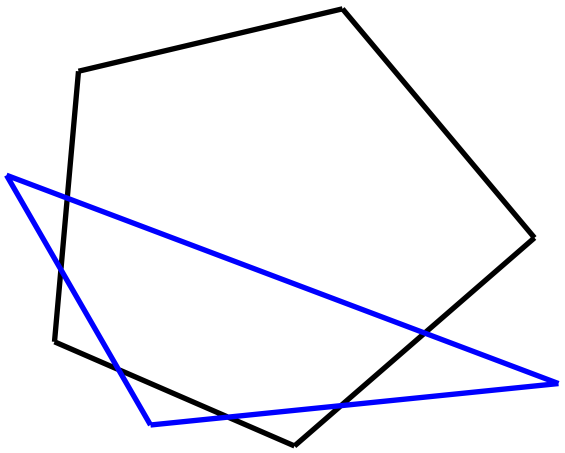
Suppose we have a sufficiently large rectangular, that covers P_1, its pocket, and P_2. It is divided into 3 parts: triangulations inside P_1 (black solid lines), triangulations outside P_1 but inside its convex hull or, inside its pockets (blue dash lines), and triangulations of the rest area (green dash line), all of which are represented in double-connected edge lists. Then it's easy to find the total number of triangles inside the rectangular is bounded by $(n-2)+(n+4+2-2)=O(n)$

Step 1: arbitrarily start from an end-point **P** of an edge **e** of polygon P_2. As the plane rectangular is divided into $O(n)$ triangles, we can decide which triangle (thus which area) **P** belongs to in $O(n)$ time with brute force. Now suppose this face is $\mathbf{f}_1$

Step 2: march from **P** along **e**, and decide whether **P** and $\mathbf{f}_1$ intersect at some edge. As $\mathbf{f}_1$ is a triangle, we can make decision in $O(1)$ time. If the answer is no, we simply jump to the other endpoint Q, and check the next edge. Suppose the answer is yes, as the figure shows, and **e** cross into a new face $\mathbf{f}_2$. We can easily detect $\mathbf{f}_2$ by calling the Twin() method of the half-edge that belongs to $\mathbf{f}_1$ and crosses **e**. We then check whether **e** and $\mathbf{f}_2$ intersect. We do this again and again until the end point Q. In this way, we determine all the triangles that have intersections to **e**. Suppose the intersection happens k times. Thus this step is finished in $O(1+k)$ times.

Step 3: remember we finished the investigation of edge **e** and come to point **Q**. We thus perform the similar actions to Step 2 iteratively, from **Q** to **R**, **R** to **S**, and so on, until we returned to **P** and find all the intersection events. So all together, Step 2 and 3 cost $O\left(m+\sum_{i=1}^{m} k_i\right)=O(m+k)$ time, in which k is the total intersection numbers P_2 has to the triangulation of the whole 3 triangulation areas.

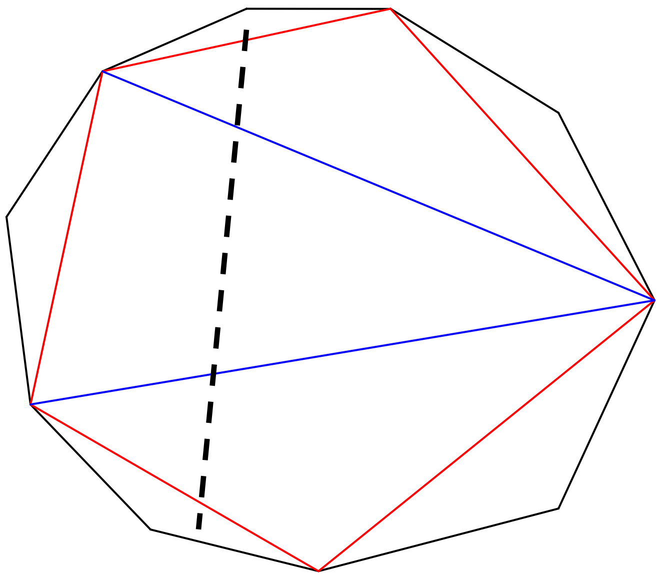
Step 4: with all the intersection points, and their face belonging information to P_1, its pocket, or the rest, we can perform the overlay actions shown in Step 2, Page 38 of the textbook to update the subdivision so as to get the intersection between P_1 and P_2, which cost $O(k)$ time.



In general, the algorithm requires $O(n)+O(m+k)+O(k)=O(n+m+k)$ time. However, as any triangle can intersect a **convex polygon** for at most 6 times (twice for each edge, shown above), the intersection of triangulation of P_1 with its pocket can intersect P_2 for at most $6 \cdot 2(n-2)$ time, which means k is still in $O(n)$. Therefore, the algorithm is in $O(n+m)$ time.

3.13

For convex polygons, we start at one vertex s, we connect its two neighboring vertices with a diagonal. We then perform similar actions to vertices until one round is finished. At this point, a set of triangles is formed, so we "throw" them away and perform the connection for second round. We do this until we are left with a triangle. This process is shown below, with different colors representing different round:



In this process, a new convex polygon is formed at each round, which can be intersected by an interior line twice. See the bold dash line for example. However, the number of vertices decreases by 50% at each round, so we can at most perform this work for $\log n$ times. Therefore, the interior line can be intersected by $2\log n$ diagonals when the algorithm finishes. Then we are done with this method to get $O(\log n)$ stabling numbers.