Problem Set 6 Solutions

Exercise 6-1. Do exercise 14.1-5 on page 307 of CLRS.

Solution: First find the rank of x. Add i to this value, and find the element with this rank. This takes $2O(\lg n) + 1$ time.

Exercise 6-2. Do exercise 14.2-2 on page 310 of CLRS.

Solution: Yes, since the black height of a node can be computed from the information at the node and its two children. According to Theorem 14.1 (page 309 of CLRS) insertion and deletion can be still performed in $O(\lg n)$ time.

Exercise 6-3. Do exercise 14.3-1 on page 316 of CLRS.

Solution: Assume that before the rotation x has α as its left child and y as its right child, and y has β as its left child and γ as its right child. The rotation changes the pointers as in the LEFT-ROTATE procedure as shown on page 278 of CLRS. In addition, at the end it sets $\mathit{max}[y] \leftarrow \mathit{max}[x]$ and $\mathit{max}[x] \leftarrow \max(\mathit{max}[\alpha], \mathit{max}[\beta], \mathit{high}[\mathit{int}[x]])$ in that order. This takes $O(1)$ time.

Exercise 6-4. Do exercise 33.1-4 on page 946 of CLRS.

Solution: For each point, sort the others by their polar angle relative to that point, and check if any two adjacent points in the sorted order have the same angle. For each point, we need $O(n)$ time to compute the polar angles of all the other points, $O(n \lg n)$ time to sort them and $O(n)$ time to check whether any two adjacent points have the same angle. We repeat the process for $O(n)$ points, which gives us a total running time of $O(n^2 \lg n)$.

Exercise 6-5. Do exercise 33.2-1 on page 946 of CLRS.

Solution: We can show this by construction: consider a regular polygon with n sides. If n is even, for each side of the polygon there is exactly one other side parallel to it (think of a square, or a hexagon). If n is odd, there are no parallel sides (think of a triangle or a pentagon). In any case, of all pairwise $n(n-1)/2$ combinations of sides, there are at most $n/2$ pairs of sides which are parallel to each other. The remaining $n(n-1)/2 - n/2$ are not parallel. So if we extend them sufficiently in both directions, at some point they will intersect (pairwise). Thus we will have n segments with $n(n-1)/2 - n/2 = \Theta(n^2)$ intersections.

Problem 6-1. Overlapping rectangles

VLSI databases commonly represent an integrated circuit as a collection of rectangles. Assume that each rectangle is rectilinearly oriented (sides parallel to the x- and y-axis), so that a representation of a rectangle consists of its minimum and maximum x- and y-coordinates.

(a) Give an $O(n \lg n)$-time algorithm that decides whether a set of rectangles so represented contains two rectangles that overlap. Your algorithm need not report all intersecting pairs, but it must report that an overlap exists if one rectangle entirely covers another, even if the boundary lines do not intersect. (*Hint:* Move a "sweep" line across the set of rectangles by replacing one of the two spatial dimensions with time. At all times maintain the collection of rectangles pierced by the sweep line.)

Solution:

Idea: The main idea here is to move a sweep line from left to right, while maintaining the set of rectangles intersected by the line in an interval tree.

Details: First sort the x-coordinates of the rectangles. Scan the sorted x-coordinate list from lowest to highest. When an x-coordinate of a left edge is found, add the interval corresponding to the y-coordinates of that edge into the interval tree and check for overlap. When an x-coordinate of a right edge is found, delete the interval corresponding to the y-coordinates of that edge from the interval tree. The interval tree always contains the set of "open" rectangles intersected by the sweep line.

(b) Argue that your algorithm indeed runs in $O(n \lg n)$ time in the worst case.

Solution: This implementation requires $O(n \lg n)$ time to sort the x-coordinates, and $2n \cdot O(\lg n)$ to maintain the interval tree, for a total running time of $O(n \lg n)$.

Problem 6-2. GPS map display

After graduating from MIT, you decide to join GiPSy, a startup that hopes to make money by selling an affordable hand-held GPS (Global Positioning System) device. The device picks up time-stamped messages from 4 geostationary satellites and uses them to calculate its precise coordinates on the globe by taking into account the orbital position of each satellite (included in the messages) and the time it took each message to reach the device.

The device has a rectangular LCD screen which displays the user's exact location on the map, as well as all the nearby streets; see Figure 1. The device must be able to update the map display in real time because the user may be moving and may zoom in or out. GiPSy's first model will only work in cities whose streets are arranged in a grid (such as Manhattan). If the first model proves to be a success in those markets, GiPSy hopes to secure the necessary funding to develop a model that can work in all cities.

Your task is to figure out how the city map should be preprocessed and stored on the device, in order to quickly answer queries about what streets are in the vicinity of the user. More precisely, the problem is defined as follows:

- A ***road*** $r = \langle (r_{x1}, r_{y1}), (r_{x2}, r_{y2}) \rangle$ is a line segment, either horizontal or vertical, specified by the coordinates of its endpoints, (r_{x1}, r_{y1}) and (r_{x2}, r_{y2}). Thus, for every road $r = \langle (r_{x1}, r_{y1}), (r_{x2}, r_{y2}) \rangle$, we have either $r_{x1} = r_{x2}$ (vertical) or $r_{y1} = r_{y2}$ (horizontal).

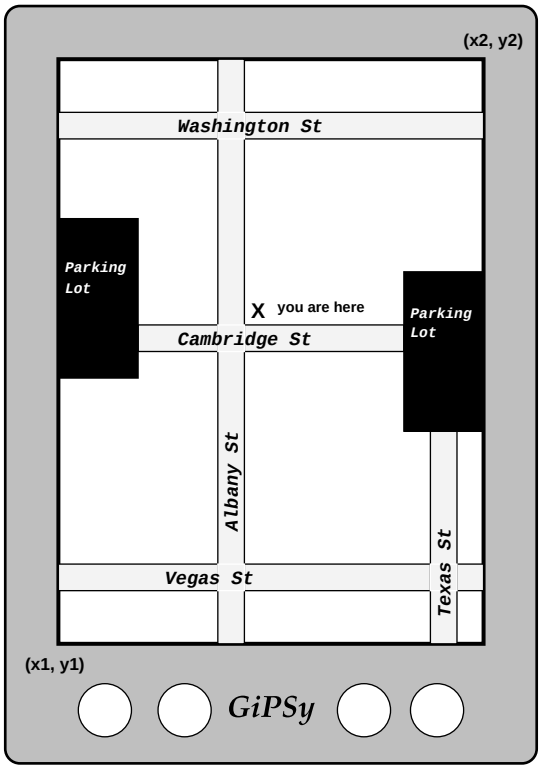

Figure 1: A sketch of a GiPSy device. Washington St., Vegas St., and Albany St. are Type-2 roads. Cambridge St. and Texas St. are Type-1 roads. The view rectangle is determined by the coordinates of the lower-left and upper-right corners, as shown.

- A ***map*** $M = \{r^1, r^2, \ldots, r^n\}$ is a set of n roads.
- A ***view rectangle*** $V = \langle (V_{x1}, V_{y1}), (V_{x2}, V_{y2}) \rangle$ specifies the rectangular region that should be displayed by giving the coordinates of the rectangle's lower-left and upper-right corners, (V_{x1}, V_{y1}) and (V_{x2}, V_{y2}) respectively.
- A road r is ***visible*** in the view rectangle V if it intersects the interior of the rectangle V. There are two types of visible roads r: can appear on the rectangular display:

  Type 1: One or both endpoints of the road r are inside the view rectangle V.

  Type 2: The road r crosses the view rectangle V but both its endpoints lie outside the view rectangle V.
- The goal of a ***clipping query*** is to report all the visible roads for a given view rectangle V.

Because the map M does not change often, we are free to spend a reasonable amount of time preprocessing the map M into a data structure that supports queries efficiently, using a reasonable amount of auxiliary space.

In the problem parts that follows, you will often be called upon to "give an efficient method" for supporting a particular type of query. For each such problem part, you must do the following:

1. Give an efficient algorithm for preprocessing the map M into a data structure.
2. Give an efficient algorithm for using this data structure to answer the query for a given view rectangle V.
3. Analyze the worst-case preprocessing time, worst-case query time, and worst-case space occupied by the data structure. In all cases, the analysis should be in terms of the total number n of roads on the map M and the number k of visible roads in the view rectangle V.

Optimizing the query time is most important; the preprocessing time is secondary.

(a) Give an efficient method for finding all Type-1 roads. (*Hint:* Use a two-dimensional range tree.)

Solution: Place the endpoints of the roads into a two-dimensional range tree. A range query on the view rectangle will return exactly the desired roads. For n roads, there are $2n$ endpoints, so the running time is asymptotically the same as for 2-D range trees: $O(n \lg n)$ preprocessing time, $O(\lg^2 n + k)$ query time, and $O(n \lg n)$ space.

The rest of the problem is about finding Type-2 roads. Without loss of generality, we focus on finding horizontal Type-2 roads.

A horizontal road r ***straddles*** a view rectangle V if it crosses the left edge of V. Thus, every straddling road r is visible. Depending on whether the right endpoint of a straddling horizontal road r is inside the view rectangle, a straddling horizontal road r may be Type-1 or Type-2.

(b) Suppose that we knew how to compute the set of straddling horizontal roads for a given map M and view rectangle V. Give an efficient algorithm to convert the set of straddling horizontal roads into the set of Type-2 horizontal roads.

Solution: Part (a) identifies all of the Type-1 horizontal roads. Mark these roads. Now run through the set of straddling horizontal roads and remove any roads that have been marked. Then we obtain the set of Type-2 horizontal roads, because every Type-2 horizontal road is a straddling horizontal road. The running time is proportional to the number of straddling horizontal roads, which is at most k.

Thus, our goal is to identify which horizontal roads are straddling. We use the following characterization. A horizontal road $r = \langle (r_{x1}, r_y), (r_{x2}, r_y) \rangle$ straddles the view rectangle $V = \langle (V_{x1}, V_{y1}), (V_{x2}, V_{y2}) \rangle$ if it satisfies two properties:

1. $r_{x1} \le V_{x1} \le r_{x2}$, i.e., the road r crosses the vertical line extending the left edge of the view rectangle V. We say that r is ***horizontally straddling***.
2. $V_{y1} \le r_y \le V_{y2}$, i.e., the road r falls within the vertical extent of the view rectangle V. We say that r is ***vertically straddling***.

(c) Draw a picture showing a view rectangle V and an example of each of the following kinds of visible horizontal roads:

1. straddling and Type-2,
2. straddling but not Type-2,
3. horizontally straddling but not vertically straddling,
4. vertically straddling but not horizontally straddling, and
5. neither horizontally straddling nor vertically straddling (but still visible).

Solution: See Figure 2

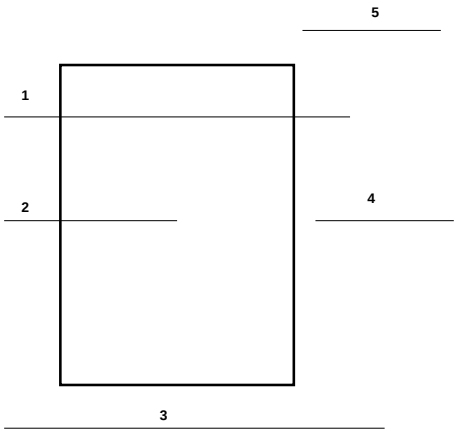

Figure 2: Different kinds of horizontal roads

A simple approach for computing the set of straddling horizontal roads is to

- compute the set of vertically straddling horizontal roads, and
- remove from this set any roads that are not horizontally straddling.

(d) Give an efficient method for finding all vertically straddling horizontal roads. (*Hint:* Use a one-dimensional range tree.)

Solution: Place the endpoints of all horizontal roads in a 1-D range tree based on their y-coordinates. We can then use this range tree to find all the roads whose y-coordinates lie within the vertical extent of the view rectangle V, that is $V_{y1} \leq r_y \leq V_{y2}$.
The preprocessing time to build the range tree is $O(n \lg n)$, and the tree requires $O(n)$ space. Each subsequent query takes time $O(\lg n + p)$ where p is the number of candidate roads on the map M.

Once we have the set of vertically straddling horizontal roads, we can simply run through the list and remove any roads that are not horizontally straddling. This filtering results in the set of straddling horizontal roads.

(e) Analyze the worst-case running time of this filtering algorithm.

Solution: The running time is linear in the number of vertically straddling horizontal roads, which can be as high as $\Theta(n)$.

(f) Explain why this approach is slow in the worst case.

Solution: The worst case occurs when all roads on the map are vertically straddling. In that case, the running time is $\Theta(n)$, even though few of these roads may actually be horizontally straddling as well (and hence visible).

Our final goal is to develop a faster method for computing the set of straddling horizontal roads.

(g) Show that instead of explicitly enumerating all vertically straddling horizontal roads in part (d), we can find $O(\lg n)$ nodes in the range tree whose descendants contain all of the vertically straddling horizontal roads and no other roads. Give an efficient algorithm to find these $O(\lg n)$ nodes.

Solution: If we insert all horizontal roads into a 1-D range tree based on their y-coordinates, then as we have seen in lecture, there exist $O(\lg n)$ subtrees in the range tree whose nodes contain the result of the query and nothing else. In order to find the nodes on which these $O(\lg n)$ subtrees are rooted, we can use the procedure 1-D RANGE QUERY presented in lecture, which runs in $O(\lg n)$ time.

(h) Give an efficient method for finding all the straddling horizontal roads. (*Hint:* Combine interval trees with range trees.)

Solution: As we have seen above, the range tree based on the y-coordinates of the horizontal roads returns $O(\lg n)$ subtrees per query. These subtrees contain all the *vertically* straddling horizontal roads. In order to find which of those roads are also horizontally stradddling (and hence in view of the rectangle), we can "pre-emptively" construct an interval tree for each one of the nodes of the range tree. The interval tree associated with each node stores the x-coordinate intervals of all the roads in the entire subtree rooted at that node. This way, when we get the $O(\lg n)$ subtrees after issuing a query to the range tree (based on y-coordinate), we can issue separate queries to the interval trees of each subtree root to find all the roads which intersect the horizontal range of the rectangle (that is, the interval $[V_{x1}, V_{x2}]$).

Query time: Since each interval tree stores at most $O(n)$ points, each query to the interval tree takes time $O(k \lg n)$ where k is the number of roads returned by that interval tree. Note that the time taken for a query that returns no roads ($k = 0$) is $O(\lg n)$ and not $O(0)$. So a more precise description of the running time would be $O(\lg n + k \lg n)$.

We issue $O(\lg n)$ such queries, for a total running time of $O(lg^2 n + K \lg n)$ where K is the total number of straddling roads.

Pre-processing Time: Building each interval tree takes time $O(m \lg m)$ where m is the number of nodes it contains. So we need $O(n \lg n)$ time to build the interval tree for the root of the range tree, $O\left(2(n/2)\lg(n/2)\right)$ for the roots of its subtrees etc. Since

$$\begin{aligned} n\lg n + 2\left(\frac{n}{2}\lg\frac{n}{2}\right) + 4\left(\frac{n}{4}\lg\frac{n}{4}\right) + \ldots 0 &= n\left[\lg n + \lg\frac{n}{2} + \lg\frac{n}{4} + \ldots + 0\right] \\ &= n\left[\lg n + \lg n - 1 + \lg n - 2 + \ldots + \lg n - \lg n\right] \\ &\approx n\left[(\lg n)(\lg n) - \frac{(\lg n)^2}{2}\right] \\ &= n\frac{\lg^2 n}{2} \end{aligned}$$

the time it takes to initially build all these trees is is $O(n \lg^2 n)$.

This can also be seen by considering the recurrence

$$T(n) = 2T(n/2) + \Theta(n \lg n)$$

which falls under case 2 of the Master Theorem and gives us the same answer.

Space: Storing these interval trees takes $O(n \lg n)$ space since we have $O(n)$ points and each point is stored in exactly one interval tree at each level of the original range tree (similar analysis as for 2-D range trees).

(i) Conclude by analyzing the entire method for answering clipping queries.

Solution: We need $O(n \lg n)$ time to build the range tree and $O(n \lg^2 n)$ time to build the interval trees, for a total of $O(n \lg^2 n)$ time for the pre-processing stage.

The space we need for the range tree is $O(n)$, and we need another $O(n \lg n)$ for the interval trees, for a total of $O(n \lg n)$.

Each query to the range tree takes $O(\lg n)$ and returns $O(\lg n)$ subtrees. As we saw above, the time it takes takes to query the associated $O(\lg n)$ interval trees is $O(\lg^2 n + K \lg n)$ where K is the total number of visible roads. Thus the total running time for each query is $O(\lg^2 n + K \lg n)$.