Range Queries

- Last time, we considered orthogonal rectangular range queries where we’re given a set of n points P that we’d like to preprocess for range queries. Each query Q is an axis-aligned rectangle, and we’d like to report information about all points in P intersection Q. These queries can be expressed using just two vertices at opposite extremes of the rectangle. Here’s rectangle $R(a, b) = [a_1, b_1] \times [a_2, b_2]$:

- We looked at the typical way to handle these kinds of data structures: find collection of canonical subsets $\{P_1, \ldots, P_k\}$ with each $P_i$ in P is chosen so that any intersection of P and an allowable range shape can be formed from the disjoint union of canonical subsets. The subsets from the list may overlap, but the subsets for a particular range query do not.

- Finally, we’d looked at how to solve the problem in 1D by building a partition tree. Specifically, we used a balanced binary tree with points as its leaves. Each node was associated with the canonical subset of its descendent leaves. We could augment internal nodes with counts, maxes, etc. for their canonical subsets to allow for range queries in $O(\log n)$ time plus however long it takes to actually report.

- Here’s pseudocode for doing a counting query. It differs a bit from the algorithm I described last time, but is maybe easier to understand. This code assumes each internal node describes the range of its leaves.
kd-trees

- So what about the plane or even higher dimensions?
- We'll start with a data structure called the kd-tree, designed by Bentley ['75].
- So originally, this stood for k-dimensional tree. But using k for dimension is confusing, and people forgot that's how the name worked. So now we say things like 2 or 3-dimensional kd-tree.
- kd-trees are partition trees. At each node, we subdivide its point set by splitting them evenly based on their x-coordinate or y-coordinate.
- Each node t stores
  - t.cut-dim: which way we'll split the points (maybe use 0 for x and 1 for y)
  - t.cut-val: at which x or y coordinate should we split the points
  - t.weight: the total weight or number of points in t’s subtree (or whatever else you’d like to report about all of the points)
- So if t.cut-dim is 0 and t.cut-val is 400, we'll store points of x-value < 400 in the left subtree and points of higher x-value in the right subtree. We’ll break ties so that the two subtrees are as evenly split as possible.
- Nodes with one point are the leaves. They store that point as t.point.
- If you zoom out a bit, here is the picture you see. Each node represents a rectangular region of space called a cell. The root’s cell can be thought of as a big rectangle surrounding all the points. When you split a node’s points, it’s like you’re splitting its cell into two smaller cells for that node’s children.
- These nested cells are sometimes called a hierarchical decomposition of space.
Now, there’s a few ways you can pick cut dimension and cut value. The standard way is to alternate between x and y as the cut dimension as I drew, and always set cut-val to be the median value so you split the points into two equal subsets. Picking different ways of picking dimension and value result in other types of binary space partition trees (BSP trees).

You can build a 2D kd-tree in $O(n \log n)$ time by first making two sorted lists of points by x-coordinate and y-coordinate. Then you can search for the median coordinate for each split and split up the lists to recursively build the two subtrees in time linear in the number of points in a subtree. You get a recurrence like $T(n) = 2T(n/2) + n$ which solves to $O(n \log n)$.

It’s a balanced binary tree with $O(n)$ leaves, so its size is $O(n)$.

We can do range counting with the following procedure:

```
int range-count(Range Q, KDNode u)
(1) if (u is a leaf)
   (a) if (u.point $\in Q$) return u.weight,
   (b) else return 0 /* or generally, the semigroup identity element */
(2) else /* u is internal */
   (a) if (u.cell $\cap Q = \emptyset$) return 0 /* the query does not overlap u’s cell */
   (b) else if (u.cell $\subseteq Q$) return u.weight /* u’s cell is contained within query range */
   (c) else, return range-count(Q, u.left) + range-count(Q, u.right).
```

Basically, return the whole count if the whole cell lies in the range. Or return 0 if the whole cell lies outside the range. Otherwise, search deeper and combine your answers.

If you want to report, then instead of returning the weight, you return a list of all the node’s descendant leaves.

How long does a query take? Call a node expanded if it and both children are visited by the recursive counting algorithm. Except for the root, each visited node has an expanded parent, so the running time is proportional to the number of expanded nodes.

A cell is stabbed if it overlaps the range without being contained within it. There are more stabbed cells than expanded cells, so we’ll just count those.

Lemma: Any horizontal or vertical line stabs $O(\sqrt{n})$ cells of the tree.

Proof:
- Suppose the line is vertical. Consider a node with cutting dimension x. The line stabs at most one of its children, and if it fails to stab a child, then it won’t stab any
descendent of that child.

- However, a node with cutting dimension y may have both children stabbed.
- Therefore, each node with cutting dimension x has at most two grandchildren stabbed. In general, the number of stabbed nodes increases by a factor of at most two every two levels of the tree.
- Let $S(n)$ be the maximum number of stabbed nodes by the vertical line. $S(n) = 2$ if $n \leq 4$ and $S(n) = 1 + 2T(n/4)$ otherwise.
- This solves to $S(n) = O(2^\left\lfloor \log_4 n \right\rfloor) = O(n^{1/2}) = O(\sqrt{n})$.

- OK, for query range $Q$ to stab a cell, that cell must be stabbed by at least one of the four line segments bounding $Q$. But each of those stabs $O(\sqrt{n})$ cells, so $O(\sqrt{n})$ cells are stabbed or expanded. A counting query takes $O(\sqrt{n})$ time. A reporting query takes $O(\sqrt{n} + k)$ time.
- Note these are worst case bounds. It’s possible for a query to take far less time, and kd-trees typically perform well in practice.
- What about higher dimensions? For those, the kd-tree splits along each of the $d$ dimensions, the 0th, the 1st, the 2nd, and so on then back to repeat the list. It still uses $O(n)$ space and can be built in $O(n \log n)$ time if $d$ is a constant.
- However, the worst case running time of a counting query increases to $O(n^{1-1/d})$. In short, a stabbing line avoids only one subtree for every $d$ levels.

**Orthogonal Range Trees**

- Next, we’ll use a bit more space to substantially reduce those worst case time bounds.
- An orthogonal range tree uses $O(n \log^{\left\lfloor d/2 \right\rfloor} n)$ space and performs counting queries in $O(\log^{\left\lfloor d/2 \right\rfloor} n)$ time.
- We’ll start with a slightly simplified version for the plane that uses $O(n \log n)$ space and has $O(\log^2 n)$ query time. Later, we’ll see how to get the query time down to $O(\log n)$.
- The key idea is not to mix sorting by $x$ and $y$ value anymore, but to separate them using what is called a multi-level search tree.
- Say we have a query rectangle $Q = [x_{lo}, x_{hi}] \times [y_{lo}, y_{hi}]$. Let $Q_1 = [x_{lo}, x_{hi}] \times R$ be a vertical strip and $Q_2 = [y_{lo}, y_{hi}] \times R$ be a horizontal strip. Query range $Q$ is simply their intersection.
- We’ll find all points in $Q_1$ first, and then we’ll dig deeper to find the subset of these points that lie in the intersection.
- We already saw a way to search $Q_1$, we can use a partition tree for one-dimensional range queries.
- In the tree, each node $u$ was associated with a canonical subset $P_u$. A search consists of finding a set of nodes with disjoint canonical subsets making up $Q_1$ intersect $P$.
- Since every point of each canonical subset lies in $Q_1$, we can safely report all of its points.
that lie in \( Q_2 \) as well by restricting our search to each canonical subset of \( Q_1 \) independently.

- And to make that efficient, we'll store an auxiliary search tree for the canonical subset \( P_u \) of each node \( u \). The canonical subsets are disjoint, so results from searching these auxiliary search trees will be disjoint as well and we can safely add or merge them.

So in summary, a 2-dimensional range tree consists of two levels: an \( x \)-range tree \( T \) where each node \( u \) points to an auxiliary \( y \)-range tree \( T_u \) over its canonical subset \( P_u \).

- For a query, we find a collection of \( O(\log n) \) nodes \( u \) whose canonical subsets have the correct \( x \)-values for our range. For each \( P_u \), we do a second search in \( T_u \) to count or report the points of \( P_u \) with the correct \( y \)-values.

```c
int range2D(Node u, Range2D Q, Range1D C=[x0,x1]) {
    if (u is a leaf) // hit the leaf level?
        return (Q contains u.point ? 1 : 0) // count if point in range
    else if (Q’s x-range contains C) { // Q’s x-range contains C
        y0, y1 = [-infinity, +infinity] // initial y-cell
        return range1Dy(u.aux, Q, [y0, y1]) // search auxiliary tree
    }
    else if (Q.x is disjoint from C) // no overlap
        return 0
    else
        return range2D(u.left, Q, [x0, u.x]) + // count left side
              range2D(u.right, Q, [u.x, x1]) // and right side
}
```
There are $O(\log n)$ searches in auxiliary trees, each taking $O(\log n)$ time, so the query time is for counting is $O(\log^2 n)$. For reporting, we can report the leaves from each auxiliary search tree in time proportional to the number of points in it, so reporting $k$ points total takes $O(\log^2 n + k)$ time.

But how much space do we use? Each point appears at most once in each auxiliary search tree, and it appears in one auxiliary search tree for each ancestor of its leaf in $T$. So each point appears $O(\log n)$ times for a total space usage of $O(n \log n)$.

We can build the data structure in $O(n \log n)$ time as well by doing so in a bottom-up manner. Whenever we want to build $T_u$ for a node $u$, we look at the auxiliary structures for its two children. They already have their points sorted by $y$-value, so we can merge them and build $T_u$ in time $O(|P_u|)$. So everything ends up taking time linear in the size of the data structure.

In $d$-dimensions, you have a top level tree for the first coordinates, and then auxiliary $(d-1)$-dimensional range trees at each node to take care of the remaining coordinates. You get a structure of size $O(n \log^{d-1} n)$ with $O(\log^d n)$ query time—that’s one log factor per level of the search tree.

### Cascaded Search

- Can we do better? It turns out we can speed up the queries by eliminating some redundant work using a technique called cascading search. It’s a special case of a more general technique called fractional cascading.
- The problem is that we’re doing a lot of separate $O(\log n)$ time searches even though each of them uses the same pair of search keys, $y_{lo}$ and $y_{hi}$.
- The main idea is that once we know where the lowest point higher than $y_{lo}$ lies in one canonical subset $P_u$, we should immediately know the lowest such point in the canonical subsets for the children of $u$. 

- [Diagram of cascaded search tree]
Let’s assume the auxiliary structures aren’t trees, but instead arrays sorted by y-value which we’ll call auxiliary lists. We’ll also stick to reporting queries for now. If you can find the least point in the array for P_u above y_lo, you can easily find all the other points of P_u in the range by just walking forward along the array and stopping as soon as you’ve gone too far.

Say we have a node v in the top level tree with two children nodes v’ and v”. Let A be the auxiliary list for v and A’ and A” be the lists for its children. A is the disjoint union of A’ and A”. For each element in A, we’ll add a pointer to the least element A’ which has higher y-value. We’ll also add one to the least element in A” which has higher y-value.

For a query, we’ll start by searching the root’s node’s list for the lowest point higher than y_lo.

Now, as we do our search for points in Q_1, we’ll be able to in only O(1) time per node follow those pointers we added and learn the lowest point higher than y_lo for every auxiliary list for every node we touch.

Meaning we spend no additional time doing searches in auxiliary lists once we know the canonical subsets for Q_1.

Reporting now takes O(log n + k) time: we find the starting positions for all the auxiliary lists in O(log n) time total and then walk along them in O(k) time total to return the points.

Counting can be done in O(log n) time, but you have to be a bit more clever since we’re working with arrays instead of trees for the auxiliary structures. In short, find differences between lowest and top indices of the auxiliary lists to learn how many elements there are. This idea can be generalized to any kind of group operation over the elements, but not semi-group operations. For those, you’ll need to iterate over all the canonical subsets.

Unfortunately, cascading search only works at the lowest level of a d-dimensional range tree, because it crucially depends upon you only needing to search for a constant number of things in the auxiliary structure. Query times in d dimensions go down to O(log^(d-1) n).