Theorem: If a range space $S = (P, R)$ over a set P of size n has VC-dimension k, then the number of distinct ranges satisfies

$$|R| \leq \sum_{i=0}^{k} \binom{n}{i}.$$

Proof: Here is a sketch of the proof, which is induction on n and k. (Details can be found in the book "The Probabilistic Method," by Alon and Spencer, Wiley, 2000.) Let $g(k, n) = \sum_{i=0}^{k} \binom{n}{i}$. It is easy to prove by induction that this function satisfies the recurrence

$$g(k, n) = g(k, n-1) + g(k-1, n-1).$$

The basis of the induction is trivial. Otherwise, for any $x \in P$, we will decompose the range space into two range spaces. For the first range space we "remove" x from all ranges. Define $S - x = (P - \{x\}, R - x)$, where $R - x = \{r - \{x\} \mid r \in R\}$. Clearly $S - x$ has $n - 1$ elements and its VC-dimension is at most k.

For the second range space we "factor out" x by considering just the ranges of R that are identical except that one contains x and one does not. Define define $S \setminus x = (P - \{x\}, R \setminus x)$, where $R \setminus x = \{r \in R \mid x \notin r, r \cup \{x\} \in R\}$. Clearly, $S \setminus x$ has $n - 1$ elements but (because we have included ranges of R that both include and exclude x) its VC-dimension is at most $k - 1$.

Finally, observe that every subset of R can be put in 1–1 correspondence with the one of the subsets from the union of these two range spaces. (Think about this!) Thus, we have

$$|R| \;=\; |R - x| + |R \setminus x| \;\leq\; g(k, n-1) + g(k-1, n-1) \;=\; g(k, n),$$

which completes the proof.

Canonical Subsets: A common approach used in solving almost all range queries is to represent P as a collection of *canonical subsets* $\{S_1, S_2, \ldots, S_k\}$, each $S_i \subseteq S$ (where k is generally a function of n and the type of ranges), such that any set can be formed as the disjoint union of canonical subsets. Note that these subsets may generally overlap each other.

There are many ways to select canonical subsets, and the choice affects the space and time complexities. For example, the canonical subsets might be chosen to consist of n singleton sets, each of the form $\{p_i\}$. This would be very space efficient, since we need only $O(n)$ total space to store all the canonical subsets, but in order to answer a query involving k objects we would need k sets. (This might not be bad for reporting queries, but it would be too long for counting queries.) At the other extreme, we might let the canonical subsets be the power set of P. Now, any query could be answered with a single canonical subset, but we would have 2^n different canonical subsets to store. (A more realistic solution would be to use the set of all ranges, but this would still be quite large for most interesting range spaces.) The goal of a good range data structure is to strike a balance between the total number of canonical subsets (space) and the number of canonical subsets needed to answer a query (time).

One-dimensional range queries: Before consider how to solve general range queries, let us consider how to answer 1-dimension range queries, or *interval queries*. Let us assume that we are given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ on the line, which we will preprocess into a data structure. Then, given an interval $[x_{lo}, x_{hi}]$, the goal is to report all the points lying within the interval. Ideally we would like to answer a query in time $O(\log n + k)$ time, where k is the number of points reported (an output sensitive result). Range counting queries can be answered in $O(\log n)$ time with minor modifications.

Clearly one way to do this is to simply sort the points, and apply binary search to find the first point of P that is greater than or equal to x_{lo}, and less than or equal to x_{hi}, and then list all the points between. This will not generalize to higher dimensions, however.

Instead, sort the points of P in increasing order and store them in the leaves of a balanced binary search tree. Each internal node of the tree is labeled with the largest key appearing in its left child. We can associate each

node of this tree (implicitly or explicitly) with the subset of points stored in the leaves that are descendents of this node. This gives rise to the $O(n)$ *canonical subsets*. For now, these canonical subsets will not be stored explicitly as part of the data structure, but this will change later when we talk about range trees. This is illustrated in the figure below.

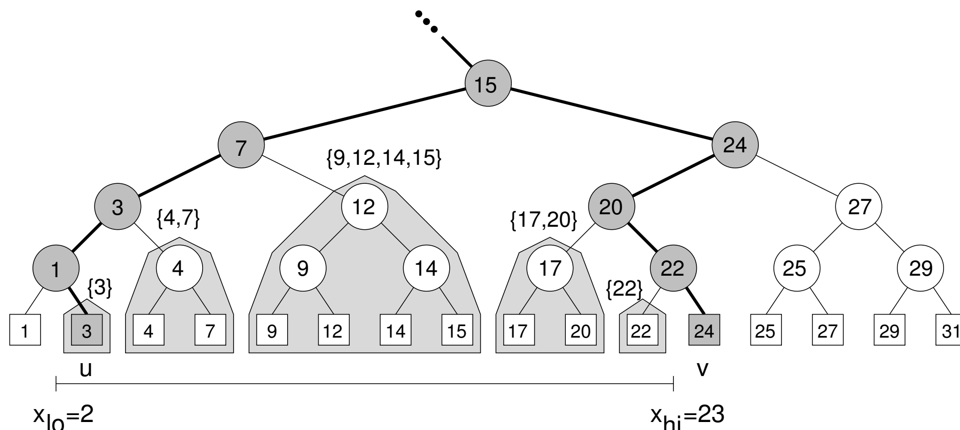

Figure 66: Canonical sets for interval queries.

We claim that the canonical subsets corresponding to any range can be identified in $O(\log n)$ time from this structure. Given any interval $[x_{lo}, x_{hi}]$, we search the tree to find the leftmost leaf u whose key is greater than or equal to x_{lo} and the rightmost leaf v whose key is less than or equal to x_{hi}. Clearly all the leaves between u and v, together possibly with u and v, constitute the points that lie within the range. If $key(u) = x_{lo}$ then we include u's canonical (single point) subset and if $key(v) = x_{hi}$ then we do the same for v. To form the remaining canonical subsets, we take the subsets of all the *maximal subtrees* lying between u and v.

Here is how to compute these subtrees. The search paths to u and v may generally share some common subpath, starting at the root of the tree. Once the paths diverge, as we follow the left path to u, whenever the path goes to the left child of some node, we add the canonical subset associated with its right child. Similarly, as we follow the right path to v, whenever the path goes to the right child, we add the canonical subset associated with its left child.

To answer a range reporting query we simply traverse these canonical subtrees, reporting the points of their leaves. Each tree can be traversed in time proportional to the number of leaves in each subtree. To answer a range counting query we store the total number of points in each subtree (as part of the preprocessing) and then sum all of these over all the canonical subtrees.

Since the search paths are of length $O(\log n)$, it follows that $O(\log n)$ canonical subsets suffice to represent the answer to any query. Thus range counting queries can be answered in $O(\log n)$ time. For reporting queries, since the leaves of each subtree can be listed in time that is proportional to the number of leaves in the tree (a basic fact about binary trees), it follows that the total time in the search is $O(\log n + k)$, where k is the number of points reported.

In summary, 1-dimensional range queries can be answered in $O(\log n)$ time, using $O(n)$ storage. This concept of finding maximal subtrees that are contained within the range is fundamental to all range search data structures. The only question is how to organize the tree and how to locate the desired sets. Let see next how can we extend this to higher dimensional range queries.

Kd-trees: The natural question is how to extend 1-dimensional range searching to higher dimensions. First we will consider kd-trees. This data structure is easy to implement and quite practical and useful for many different types of searching problems (nearest neighbor searching for example). However it is not the asymptotically most efficient solution for the orthogonal range searching, as we will see later.

Lecture 23: Orthogonal Range Trees

Reading: Chapter 5 in the 4M's.

Orthogonal Range Trees: Last time we saw that kd-trees could be used to answer orthogonal range queries in the plane in $O(\sqrt{n}+k)$ time. Today we consider a better data structure, called *orthogonal range trees*.

An orthogonal range tree is a data structure which, in all dimensions $d \geq 2$, uses $O(n \log^{(d-1)} n)$ space, and can answer orthogonal rectangular range queries in $O(\log^{(d-1)} n + k)$ time, where k is the number of points reported. Preprocessing time is the same as the space bound. Thus, in the plane, we can answer range queries in time $O(\log n)$ and space $O(n \log n)$. We will present the data structure in two parts, the first is a version that can answer queries in $O(\log^2 n)$ time in the plane, and then we will show how to improve this in order to strip off a factor of $\log n$ from the query time.

Multi-level Search Trees: The data structure is based on the concept of a *multi-level search tree*. In this method, a complex search is decomposed into a constant number of simpler range searches. We cascade a number of search structures for simple ranges together to answer the complex range query. In this case we will reduce a d-dimensional range search to a series of 1-dimensional range searches.

Suppose you have a query which can be stated as the intersection of a small number of simpler queries. For example, a rectangular range query in the plane can be stated as two range queries: Find all the points whose x-coordinates are in the range $[Q.x_{lo}, Q.x_{hi}]$ and all the points whose y-coordinates are in the range $[Q.y_{lo}, Q.y_{hi}]$. Let us consider how to do this for 2-dimensional range queries, and then consider how to generalize the process. First, we assume that we have preprocessed the data by building a range tree for the first range query, which in this case is just a 1-dimensional range tree for the x-coordinates. Recall that this is just a balanced binary tree on these points sorted by x-coordinates. Also recall that each node of this binary tree is implicitly associated with a *canonical subset* of points, that is, the points lying in the subtree rooted at this node.

Observe that the answer to any 1-dimensional range query can be represented as the disjoint union of a small collection of $m = O(\log n)$ canonical subsets, $\{S_1, S_2, \ldots, S_m\}$, where each subset corresponds to a node t in the range tree. This constitutes the first level of the search tree. To continue the preprocessing, for the second level, for each node t in this x-range tree, we build an *auxiliary tree*, t_{aux}, each of which is a y-coordinate range tree, which contains all the points in the canonical subset associated with t.

The final data structure consists of two levels: an x-range tree, such that each node in this tree points to auxiliary y-range tree. This notion of a tree of trees is basic to solving range queries by leveling. (For example, for d-dimensional range trees, we will have d-levels of trees.)

Query Processing: Given such a 2-level tree, let us consider how to answer a rectangular range query Q. First, we determine the nodes of the tree that satisfy the x portion of the range query. Each such node t is associated with a canonical set S_t, and the disjoint union of these sets $O(\log n)$ sets consistute all the points of the data set that lie within the x portion of the range. Thus to finish the query, we need to find out which points from each canonical subset lie within the range of y-coordinates. To do this, for each such node t, we access the auxiliary tree for this node, t_{aux} and perform a 1-dimensional range search on the y-range of the query. This process is illustrated in the following figure.

What is the query time for a range tree? Recall that it takes $O(\log n)$ time to locate the nodes representing the canonical subsets for the 1-dimensional range query. For each, we invoke a 1-dimensional range search. Thus there are $O(\log n)$ canonical sets, each invoking an $O(\log n)$ range search, for a total time of $O(\log^2 n)$. As before, listing the elements of these sets can be performed in additional k time by just traversing the trees. Counting queries can be answered by precomputing the subtree sizes, and then just adding them up.

Space: The space used by the data structure is $O(n \log n)$ in the plane (and $O(n \log^{(d-1)} n)$ in dimension d). The reason comes by summing the sizes of the two data structures. The tree for the x-coordinates requires only $O(n)$ storage. But we claim that the total storage in all the auxiliary trees is $O(n \log n)$. We want to count

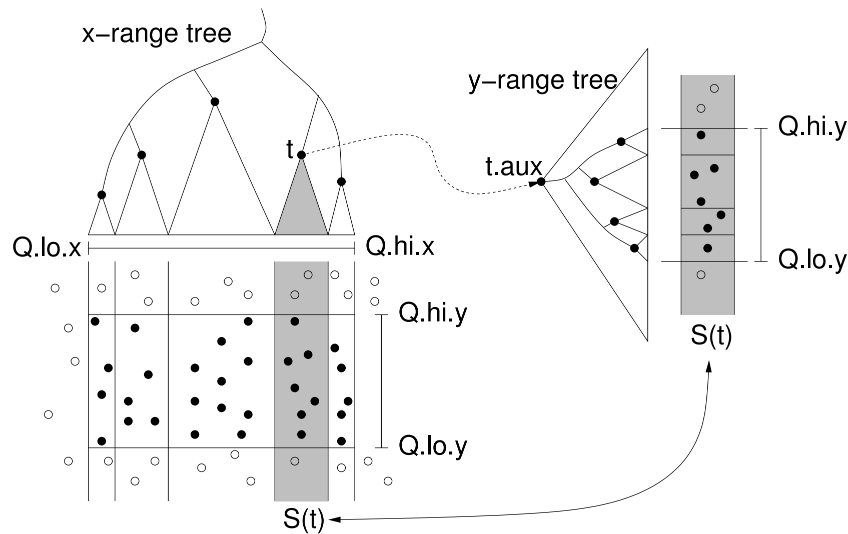

Figure 83: Orthogonal range tree search.

the total sizes of all these trees. The number of nodes in a tree is proportional to the number of leaves, and hence the number of points stored in this tree. Rather than count the number of points in each tree separately, instead let us count the number of trees in which each point appears. This will give the same total. Observe that a point appears in the auxiliary trees of each of its ancestors. Since the tree is balanced, each point has $O(\log n)$ ancestors, and hence each point appears in $O(\log n)$ auxiliary trees. It follows that the total size of all the auxiliary trees is $O(n \log n)$. By the way, observe that because the auxiliary trees are just 1-dimensional trees, we could just store them as a sorted array.

We claim that it is possible to construct a 2-dimensional range tree in $O(n \log n)$ time. Constructing the 1-dimensional range tree for the x-coordinates is easy to do in $O(n \log n)$ time. However, we need to be careful in constructing the auxiliary trees, because if we were to sort each list of y-coordinates separately, the running time would be $O(n \log^2 n)$. Instead, the trick is to construct the auxiliary trees in a bottom-up manner. The leaves, which contain a single point are trivially sorted. Then we simply merge the two sorted lists for each child to form the sorted list for the parent. Since sorted lists can be merged in linear time, the set of all auxiliary trees can be constructed in time that is linear in their total since, or $O(n \log n)$. Once the lists have been sorted, then building a tree from the sorted list can be done in linear time.

Multilevel Search and Decomposable Queries: Summarizing, here is the basic idea to this (and many other query problems based on leveled searches). Let (S, R) denote the range space, consisting of points S and range sets R. Suppose that each range of R can be expressed as an intersection of simpler ranges $R = R_1 \cap R_2 \cap \ldots \cap R_c$, and assume that we know how to build data structures for each of these simpler ranges.

The multilevel search structure is built by first building a range search tree for query R_1. In this tree, the answer to any query can be represented as the disjoint union of some collection $\{S_1, S_2, \ldots, S_m\}$ of canonical subsets, each a subset of S. Each canonical set corresponds to a node of the range tree for R_1.

For each node t of this range tree, we build an auxiliary range tree for the associated canonical subset S_t for ranges of class R_2. This forms the second level. We can continue in this manner, with each node of the range tree for the ranges R_i being associated with an auxiliary range tree for ranges R_{i+1}.

To answer a range query, we solve the first range query, resulting in a collection of canonical subsets whose disjoint union is the answer to this query. We then invoke the second range query problem on each of these canonical subsets, and so on. Finally we take the union of all the answers to all these queries.

Fractional Cascading: Can we improve on the $O(\log^2 n)$ query time? We would like to reduce the query time to $O(\log n)$. As we descend the search the x-interval tree, for each node we visit, we need to search the corresponding y-interval tree. It is this combination that leads to the squaring of the logarithms. If we could search each y-interval in $O(1)$ time, then we could eliminate this second log factor. The trick to doing this is used in a number of places in computational geometry, and is generally a nice idea to remember. We are repeatedly searching different lists, but always with the same key. The idea is to merge all the different lists into a single massive list, do one search in this list in $O(\log n)$ time, and then use the information about the location of the key to answer all the remaining queries in $O(1)$ time each. This is a simplification of a more general search technique called *fractional cascading*.

In our case, the massive list on which we will do one search is the entire of points, sorted by y-coordinate. In particular, rather than store these points in a balanced binary tree, let us assume that they are just stored as sorted arrays. (The idea works for either trees or arrays, but the arrays are a little easier to visualize.) Call these the *auxiliary lists*. We will do one (expensive) search on the auxiliary list for the root, which takes $O(\log n)$ time. However, after this, we claim that we can keep track of the position of the y-range in each auxiliary list in constant time as we descend the tree of x-coordinates.

Here is how we do this. Let v be an arbitrary internal node in the range tree of x-coordinates, and let v_L and v_R be its left and right children. Let A_v be the sorted auxiliary list for v and let A_L and A_R be the sorted auxiliary lists for its respective children. Observe that A_v is the disjoint union of A_L and A_R (assuming no duplicate y-coordinates). For each element in A_v, we store two pointers, one to the item of equal or larger value in A_L and the other to the item of equal or larger value in A_R. (If there is no larger item, the pointer is null.) Observe that once we know the position of an item in A_v, then we can determine its position in either A_L or A_R in $O(1)$ additional time.

Here is a quick illustration of the general idea. Let v denote a node of the x-tree, and let v_L and v_R denote its left and right children. Suppose that (in bottom to top order) the associated nodes within this range are: $\langle p_1, p_2, p_3, p_4, p_5, p_6 \rangle$, and suppose that in v_L we store the points $\langle p_2, p_3, p_5 \rangle$ and in v_R we store $\langle p_1, p_4, p_6 \rangle$. (This is shown in the figure below.) For each point in the auxiliary list for v, we store a pointer to the lists v_L and v_R, to the position this element would be inserted in the other list (assuming sorted by y-values). That is, we store a pointer to the largest element whose y-value is less than or equal to this point.

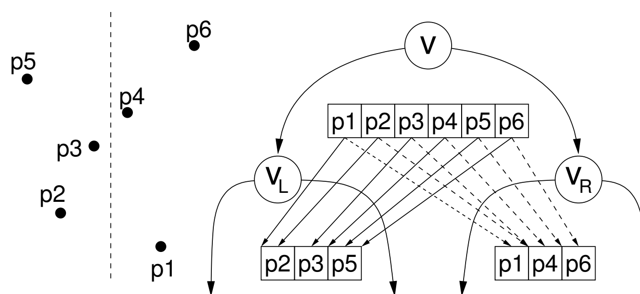

Figure 84: Cascaded search in range trees.

At the root of the tree, we need to perform a binary search against all the y-values to determine which points lie within this interval, for all subsequent levels, once we know where the y-interval falls with respect to the order points here, we can drop down to the next level in $O(1)$ time. Thus (as with fractional cascading) the running time is $O(2\log n)$, rather than $O(log^2 n)$. It turns out that this trick can only be applied to the last level of the search structure, because all other levels need the full tree search to compute canonical sets.

Theorem: Given a set of n points in R^d, orthogonal rectangular range queries can be answered in $O(\log^{(d-1)} n + k)$ time, from a data structure of size $O(n \log^{(d-1)} n)$ which can be constructed in $O(n \log^{(d-1)} n)$ time.