# Chapter 10

# Orthogonal Range Searching

By Sariel Har-Peled, February 9, 2023[1] Version: 0.3

Finally: It was stated at the outset, that this system would not be here, and at once, perfected. You cannot but plainly see that I have kept my word. But I now leave my cetological System standing thus unfinished, even as the great Cathedral of Cologne was left, with the crane still standing upon the top of the uncompleted tower. For small erections may be finished by their first architects; grand ones, true ones, ever leave the copestone to posterity. God keep me from ever completing anything. This whole book is but a draft - nay, but the draft of a draft. Oh, Time, Strength, Cash, and Patience
– – Herman Melville, Moby Dick.

Here, we study the problem of answering range searching queries on a set of points $P$, where the queries are orthogonal. For example, consider quickly reporting the number of points in $\mathcal{r} \cap P$, where $\mathcal{r}$ is some axis-parallel rectangle. This problem is a good playground for several nice techniques, in particular recursion over dimensions in data-structures, secondary data-structures, etc.

## 10.1. One dimensional case

In one dimension, a set of $n$ points $P$ is just a set of $n$ real numbers. A "rectangle" is an ***interval*** of the form $[x, y]$. If we store $P$ in a balanced binary tree of height $O(\log n)$ that supports insertions and deletions in $O(\log n)$ time. It is not hard to modify the data-structure such that it supports the following operations in $O(\log n)$ per operation:

(I) ***emptiness query***: Given an interval $\mathcal{i}$, is $\mathcal{i} \cap P$ empty?

(II) ***counting query***: Given an interval $\mathcal{i}$, what is the value of $|\mathcal{i} \cap P|$?

(III) ***reporting query***: Given an interval $\mathcal{i}$, report all the points of $\mathcal{i} \cap P$. If $k = |\mathcal{i} \cap P|$ then this takes $O(\log n + k)$ time.

It is somewhat instrumental to consider the static case (no insertions/deletions), and build the tree directly using median selection, placing it in the root, and recursing. Here, the left subtree contains all the elements smaller than $m$, the root stores the value $m = \mathrm{median}(P)$, and the right subtree contains all the numbers in $P$ larger than $m$. (Whether we store the values in the nodes, or only the leafs of the tree does not matter in this specific case.)

**Canonical sets.** For such a tree $\mathsf{T}$, and a vertex $v \in \mathsf{V}(\mathsf{T})$, let $P(v)$ be the point-set stored in the subtree of $v$. We will refer to the set $P(v)$ as a ***canonical set***. Assume that for silly reason, we store at each node $v$ in $\mathsf{T}$, a linked list $L(v)$ with all the points of $P(v)$.

**Lemma 10.1.1.** *For a set $P$ of $n$ numbers, the total size of the canonical sets in the tree $\mathsf{T} = \mathsf{T}(P)$ constructed above is $O(n \log n)$. This holds if $\mathsf{T}$ is any balanced binary search tree of depth $O(\log n)$.*

[1]This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

*Proof:* Every number of $P$ appears in a single list in each level of $\mathsf{T}$. Thus, the total size of the lists (i.e., canonical sets) is $O(n \log n)$. ■

This algorithm has two natural extensions to higher dimensions, leading to $kd$-trees, and range-trees.

## 10.2. $kd$-trees

(**Completely sketchy, sorry**.)

## 10.3. Range trees

**Theorem 10.3.1.** *Given a set $P$ of $n$ points in $\mathbb{R}^d$, one can preprocess it in $O(n \log^{d-1} n)$ time, such that given an axis parallel box $\mathsf{r} = \prod_{i=1}^{d} \mathcal{i}_i$, one do counting queries in $O(\log^d n)$ time, or reporting queries in $O(\log^d n + k)$ time.*

### 10.3.1. Fractional cascading – how to save a log?

(**Somewhat sketchy, sorry**.)

We can improve the query time in the above to $O(\log^{d-1} n)$ by using a clever technique called ***fractional cascading***. To this end, consider building a range tree for the two dimensional case, but now we store the points only in the leafs). Thus, a node $v$ in the top tree corresponds to an interval in the $x$-axis, and all the points of $P(v)$ are in this vertical slab. A canonical set of a node of this top tree can be viewed as merging (or if you want, splitting) the two Canonical sets of its children.

For explanation purposes, we store the canonical sets as sorted arrays of the points in the $y$-axis. Let $v$ be the parent, and $\alpha, \beta$ be its two children in the top level $x$-ordered tree. Let $C_v$ be the sorted array (in the $y$-order) storing the canonical set of $v$ (define $C_x$ and $C_y$ similarly). Observe, that $C_v$ is the result of merge sorting the two arrays. In particular, store for each value of $C_v[i]$, the two largest indices $i'$ and $i''$ (you can just store these two values in arrays $A_\alpha[\ldots]$ and $A_\beta[\cdots]$) such that $C_\alpha[i'] \leq_y C_v[i]$ and $C_\beta[i''] \leq_y C_v[i]$ (one of these would be with equality).

The idea is now that given a query rectangles $\mathsf{r}$, we query the top array $C_{\mathrm{root}}$ to figure out the $y$-range of the rectangle – namely, in the $y$-interval of the rectangle contains all the points of $C_{\mathrm{root}}[i, j]$, and we can compute $i, j$ in $O(\log n)$ time. We now use the "cascading" to update the information to the children of the root. Every time the query algorithm visits a note, we update its $y$-range from its parent. This takes constant time as we go down a level. This implies that now, we no longer needs to perform a secondary search on the $y$-range, since we maintain the $y$-range of the query as the query descends. We thus the following.

**Lemma 10.3.2.** *Given a set $P$ of $n$ points in $\mathbb{R}^2$, one can preprocess it in $O(n \log n)$ time, such that given an axis parallel box $\mathsf{r} = \prod_{i=1}^{d} \mathcal{i}_i$, one do counting queries in $O(\log^{d-1} n)$ time, or reporting queries in $O(\log^{d-1} n + k)$ time.*

## 10.4. Bibliographical notes