One-dimensional range counting
From last week: Counting using an array

Given $n$ points on a line, query = count points in an interval

Binary search for start and end of interval in a sorted array of the points, and subtract their positions

Preprocessing time $O(n \log n)$ by sorting

Space $O(n)$ to store sorted array

Query time $O(\log n)$
Augmented binary search trees

Store aggregate information in each node, the size of its subtree

“x, y” means key value is x, subtree size is y

Can compute at each node as sum of child subtree sizes + 1
Ranking

Rank(x) = its position in sorted array

Call the following recursive search with node = tree root:

def rank(x, node):
    if node == None:
        return 0
    else if x <= node.key:
        return rank(x, node.left)
    else:
        return rank(x, node.right) + node.left.size + 1
This problem has an easy solution:

- Binary search to find the element at the start of the range
- Binary search to find the element at the end of the range
- Rank both elements
- Subtract the ranks and add one

Instead, we will find a solution without subtraction that can be generalized to many other range searching problems.

Avoiding subtraction allows it to work for operations like min that do not have inverses.
One-dimensional range searching
Decomposing ranges into subtrees

Main idea: Given interval \([L,R]\), decompose the set of keys \(k\) with \(L \leq k \leq R\) into \(O(\log n)\) subtrees and \(O(\log n)\) individual nodes.

Number of keys in range = sum of sizes of selected subtrees + 1 for each individual node
Finding the decomposition

As we recurse, replace range endpoints by flag values $-\infty$ and $+\infty$ in subtrees for which endpoints are no longer relevant.

To decompose subtree rooted at $x$, for range $[\text{low}, \text{high}]$:

- If $\text{low} = -\infty$ and $\text{high} = +\infty$, output $x$ as a whole subtree.
- If $\text{key} > \text{high}$, recursively decompose left child for same range.
- If $\text{key} < \text{low}$, recursively decompose right child for same range.
- Otherwise:
  - Recursively decompose left with $[\text{low}, +\infty]$
  - Output $x$ as an individual node
  - Recursively decompose right with $[-\infty, \text{high}]$
Using the decomposition

To count points in range:
Decompose range
Add size for each generated subtree, 1 for each individual node

To list all points in range:
Decompose range
Traverse each generated subtree, outputting all its nodes; also output each individual node

Same time and space as sorted array

Disadvantage: More complicated
Advantage: Easier to update
Decomposable range search problems

Same idea works whenever we have:

- We have a collection of key, value pairs with sorted keys
- An associative binary operation $\odot$ operates on the values
- We want to find the result of applying $\odot$ to the values whose keys are within a query range $[\text{low}, \text{high}]$

Range counting: value $= 1$, $\odot = \text{addition}$

Range reporting: value $= \text{[itself]}$, $\odot = \text{concatenation of lists}$

Prioritization: value $= \text{pair of priority and item}$, $\odot = \text{maximum}$
Multi-level data structures
Rectangular range counting revisited

Input: a set of 2d points

Query: a rectangle, defined by the coordinates of its left, right, top, and bottom sides

Output: how many points are in the rectangle?

We saw how to do this in $O(n)$ space and $O(\sqrt{n})$ query time by using kD-trees

With a little more space, we can get much faster queries!
Range counting as a 1d decomposable problem

The left and right sides of a query rectangle define a 1d interval.

We want to know: For the points whose $x$-coordinates are in that interval, how many also have $y$-coordinates between the top and bottom sides of the rectangle?

This is decomposable!

Value of a point $= 1$ if its $y$-coordinate is between top and bottom, $0$ otherwise

Aggregation operation $\oplus = \text{addition}$
Rectangular range counting query algorithm

Use a binary search tree of points, by their $x$-coordinates

Decompose the range $[\text{left}, \text{right}]$ into $O(\log n)$ subtrees and $O(\log n)$ individual points

For each individual point $x, y$, test if $\text{bottom} \leq y \leq \text{top}$ and if so add one to the total

For each subtree, count the points with $\text{bottom} \leq y \leq \text{top}$ and add the result to the total

How can we augment the nodes of the search tree by extra information to make this possible? We can’t just store an extra number on each binary search tree node, because different rectangles that use the same node will have different counts.
Augmenting for rectangular range counting

We need to perform computations like count the points with $\text{bottom} \leq y \leq \text{top}$ in a subtree of our binary search tree.

This is a one-dimensional range counting problem!

Augment each node of the binary search tree with a pointer to a range counting data structure for its subtree (this can just be a sorted array).

Then we can perform each computation as a range counting query in this 1d data structure.
Multi-level structure

Make a binary search tree of points, sorted by $x$.

Each node stores an array of all points in its subtree, sorted by $y$.
Using a multi-level structure

To count points in a query rectangle:
  ▶ Perform query on $x$-range of rectangle
  ▶ For each individual point $(x, y)$ found by query:
      Test whether $y$ is in range
  ▶ For each subtree identified by query:
      Use sorted array at subtree root to count descendants whose $y$ coordinate is in range
  ▶ Add the results and return the total
Multi-level analysis

Each point contributes to sorted arrays in $O(\log n)$ nodes (its ancestors in the $x$-tree)
$\Rightarrow$ total length of all sorted arrays is $O(n \log n)$

Rectangle query makes $O(\log n)$ binary searches in sorted arrays
$\Rightarrow$ rectangular range counting query time $O(\log^2 n)$
$\Rightarrow$ rectangular range reporting query time $O(\log^2 n + k)$
With fractional cascading

All the binary searches in the $y$-sorted 1d arrays use the same keys (the top and bottom sides of the query rectangle)

All of the items in the child lists already appear in the parent list, so there is nothing to merge from the children

For each point in each $y$-sorted list, store its successor in the left and right child lists

Binary search for top and bottom sides at the $x$-tree root, then in each step down the tree we can follow pointers to their positions in the child lists in constant time

⇒ rectangular range counting query time $O(\log n)$
⇒ rectangular range reporting query time $O(\log n + k)$