Ch. 5 Orthogonal Range Searching

E.g. Database

<table>
<thead>
<tr>
<th>Employee</th>
<th>Data base record</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>salary</td>
</tr>
<tr>
<td>salary</td>
<td>start date</td>
</tr>
<tr>
<td>city address</td>
<td></td>
</tr>
</tbody>
</table>

0) Suppose want to know all employees of company

1) that make salary $\in [40k, 50k]$  
   \[ \Rightarrow \text{Scan records + pick out those in range} \quad \ast \text{slow}\ast \]

2) that make salary $\in [40k, 50k]$ AND age $\in [25, 40]$  
   \[ \Rightarrow \text{Scan records + check each one} \quad \ast \text{slow}\ast \]

Alternative to scan: each employee is a point in space

\[ 4D \Rightarrow \begin{cases} 
\text{age range} & [15, 75] \\
\text{salary range} & [0, 500k] \\
\text{start date} & [1/1/1900 - today] \\
\text{city address} & \text{College Station, Bryan, Austin, ...} \end{cases} \]

\text{May just keep as data w/ records}
- could encode
- could use categories (of area)

Orthogonal Range Query (Rectangular)

\[ \Rightarrow \text{Want all pts in the orthogonal range} \]
\[ \Rightarrow \text{Fast, then linear scan if use good datastructure.} \]
\[ \Rightarrow \text{Want time } O(cn) + k \ast \# \text{pts} \to \text{sorted} \]
1-D Range Searching (Sect 5.1)

Data: \( p = p_1, p_2, \ldots, p_n \) 3

Query: Which points in 1D query rectangle (in interval \([x, x']\))

Data Structure 1: Sorted Array

\[ A = 3 \ 9 \ 27 \ 28 \ 29 \ 98 \ 141 \ 187 \ 200 \ 201 \ 202 \ 999 \]

* Query: Search for \( x + x' \) in \( A \) (binary search \( O(\log n) \) time)
  - Update: Hard to insert \# points (need to shift array \( O(n) \) time in \( WC \))

Data Structure 2: Balanced Binary Search Tree

leaves: Store points in \( P \) (in order left to right)

internal nodes: Splitting values \( x \) used to guide search

- Left subtree of \( v \) contains all values \( \leq x_v \)
- Right subtree of \( v \) contains all values \( > x_v \)

Query: \([x, x']\)

- locate \( x + x' \) in \( T \) (search ends at leaves \( u + u' \))
- Pts we want are located in leaves
  - In between \( u + u' \)
  - Possibly in \( u \) (if \( x = u_v \))
  - Possibly in \( u' \) (if \( x' = u'_v \))

leaves of subtrees rooted at nodes \( v \) s.t. \( u \) is on search path root to \( u \) (or root to \( u' \))

Query: \([18, 77]\)

Look for node \( v \) split when search paths for \( x + x' \) split

=> report all values in right subtree on search path for \( x \)

=> report all values in left subtree on search path for \( x' \)
Find Split Node \((T, x, x')\)

input: Balanced Binary Search Tree \(T\), + two values \(x \leq x'\)
output: \(v_{\text{split}}\) node whose paths split (or leave when both end)

1. \(v := \text{root}(T)\)
2. While \((v \neq \text{leaf})\) and \((x \geq x_v \text{ or } x' \leq x_v)\) \(\rightarrow\) go "same way"
3. if \(x' \leq x_v\)
   4. then \(v := \text{lc}(v)\)
   5. else \(v := \text{rc}(v)\)
6. endwhile
7. return \(v\)
8. End /* Find Split Node */

Algorithm: 1D-Range Query \((T, x, x')\)

input: range tree \(T\) + \(x \leq x'\)
output: all pts in range \([x, x']\)

1. \(v_{\text{split}} := \text{FindSplitNode}(T, x, x')\)
2. if \((v_{\text{split}} = \text{leaf})\)
   then report \(x_{v_{\text{split}}} = x\) if necessary
   else \(v := \text{lc}(v_{\text{split}})\) /* first check path to \(x\)*/
   While \((v \neq \text{leaf})\)
   if \((x < x_v)\)
     then \(\text{ReportSubtree}(\text{rc}(v)) \leftarrow\) report all nodes in subtree
     \(v := \text{lc}(v)\)
   else \(v := \text{rc}(v)\)
   endwhile
   report \(x_{v_{\text{split}}} = x'\) if necessary
   \(v := \text{rc}(v_{\text{split}})\) /* now check path to \(x'\)*/
   similar...
Correctness...

Lemma (5.1) Algorithm 1D-Range Query reports exactly those points in $[x, x']$

Proof:

claim: every pt $p$ reported lies in $[x, x']$
- if $p$ lies in $u$ or $u'$ (end leaf of search) it's checked explicitly
- otherwise $p$ is reported in call Report Subtree
  - wlog assume made on path to $x$ (other case is symmetrical)
  - let $v$ be node on path s.t. $p$ reported in Report Subtree($rc(v)$)
  - $v$ & $rc(v)$ lie in left subtree $V_{split}$
    $\Rightarrow$ $p < X_{V_{split}}$
    $\Rightarrow$ $X_{V_{split}} < x'$ (Since search path $x'$ goes right $V_{split}$)
    $\Rightarrow$ $p < x'$
      - $x < p$ (Since search $x$ goes left at $v$ & $p$ in right subtree $v$)
    $\Rightarrow$ $p \in [x, x']$

claim every pt $p$ that lies in range $[x, x']$ is reported.

Let $\mu$ be leaf where $p$ is stored
Let $p$ be stored at leaf $\mu$
$V$ be lowest ancestor of $\mu$ visited by query alg.

claim $\mu = v$ (which implies $p$ is reported)
- assume $v \neq \mu$.
  $\Rightarrow$ $v$ cannot be visited by call Report Subtree (otherwise visited & reported)

$V_{split}$ $\Rightarrow$ $v$ is on search path to $x$ or $x'$ (or both)
- if $\mu$ is in left subtree $V$
  $\Rightarrow$ search for $x$ goes right at $v$ (since $v$ lowest visited ancestor)
    $\Rightarrow$ $p < x$ $\Leftarrow$ assumption $p$ is in $[x, x']$
- if $\mu$ is in right subtree $V$
  $\Rightarrow$ search for $x'$ goes left at $v$ (since $v$ lowest ancestor on path)
    $\Rightarrow$ $p > x'$ $\Leftarrow$ assumption $p$ is in $[x, x']$

not visited by Report Subtree
Time Complexity of 1D-Range Query

Balanced Binary Search Tree
- $\Theta(n)$ Storage
- $\Theta(n \log n)$ construction time

Query Time
- Find Split Node - $O(\log n)$ time since $T$ balanced
- Time spent on traversing root to $\mu + \mu'$ paths - $O(\log n)$ time
- Time spent in ReportSubtree
  - worst case $\Theta(n)$
    Since may need to report all $n$ points if they fall in query range.
  - finer analysis gives total time in ReportSubtree is proportional to # nodes reported. $\Rightarrow O(k)$ if report $k$ node

Total Time - $O(k + \log n)$ where $k$ is # nodes reported

"Output Sensitive" Algorithm.
**Kd-Trees**

Higher dimensional generalization of 1D-Range Tree...

E.g. for 2-dimensions

Idea: first split on x-coord (even levels)

Next split on y-coord (odd levels)

Repeat

Leaves: Store pts

Internal nodes: Splitting lines (as opposed to values)

---

Algorithm: Build Kd-tree (P, depth)

Input: set of pts P + current depth

Output: root of Kd-tree storing P

1. If (|P| = 1)

2. Then return leaf storing P

3. Else if (depth is even)

4. Split P into 2 equal sets by vertical line L (P + P2)

5. Else

6. Split P into equal sized P1 + P2


7. Endif

8. V_left := Build Kd-tree (P1, depth + 1)

9. V_right := Build Kd-tree (P2, depth + 1)

10. (new) V S.t. lc(v) := V_left

11. rc(v) := V_right

12. Return V

End // Build Kd-tree

---

Conventions: pt on splitting line belong to left or bottom set (median = L\[\frac{1}{2}\])
**Construction Time**

- **Expensive op**: determining splitting line (median finding)
  - Can use linear time median finding alg \(\Rightarrow O(n \log n)\) time.
  
  \[ T(n) = O(n) + 2T(n/2) \]
  
  \[ = O(n \log n) \]

- But can obtain this time w/o fancy median finding
  - Present \(n\) pts by \(x\)-coord AND by \(y\)-coord \((O(n \log n))\)
    - Each time find median in \(O(1)\) and partition
      - lists & update \(x\) & \(y\) ordering by scan in \(O(n)\) time

  \[ T(n) = O(n) + 2T(n/2) \]
  
  \[ = O(n \log n) \]

**Storage**

- \# leaves = \(n\) (one per pt.)
- Still binary tree \(\Rightarrow O(n)\) storage total.

**Queries**

- Each node corresponds to a region in plane
- Need only search nodes whose region intersects query region
- Report all points in subtrees whose regions contained in query range
- When reach leaf, check if pt in query region.
Algorithm `Search KdTree (v, R)`

1. if (v is leaf)
2. then report v's pt if in R
3. else if (region(lc(v)) fully contained in R)
4. then `ReportSubtree(lc(v))`
5. else if (region(lc(v)) intersects R)
6. then `Search KdTree(lc(v), R)`
7. if (region(rc(v)) fully contained in R)
8. then `ReportSubtree(rc(v))`
9. else if (region(rc(v)) intersects R)
10. then `Search KdTree(rc(v), R)`
11. endif

`end /* Search KdTree */`

**Note:** Need to know region(v)
- can precompute & store
- compute during recursive calls, e.g.,

\[
\text{region(lc(v))} = \text{region(v)} \cap \ell(v)^{\text{left}}
\]

\[\ell(v)\] is v's splitting line \[\ell(v)^{\text{left}}\] is left halfplane of lc(v)
Lemma 5.4 A query w/ an axis parallel rectangle in a Kd-tree storing $n$ pts can be performed in $O(n + k)$ time when $k$ is the number of reported pts.

Proof

- Total time for reporting pts in ReportSubtree is $O(k)$.

So need to bound # nodes visited by query algorithm that are not in a traversed subtree.

- For each such node $v$, region($v$) intersects but is not contained in $R$.

  => to bound # such nodes we bound # regions intersected by vertical (gives bound on # regions intersected by top or bottom edge of $R$) line (bound on # intersected by left or right edge of $R$ is similar).

- Let $l$ be vertical line.

- Let $l(root(t))$ be root's splitting line.

  => $l$ intersects region to right or left of $l(root(t))$ but not both.

$Q(n) = \#$ intersected regions in n pt Kd-tree whose root contains vertical splitting line

  => got down 2 levels before counting.

  - $l$ intersects 2 of 4 regions at this level.
  - Each contains $\frac{n}{4}$ pts.

$Q(n) = 2 + 2Q(n/4)$
$Q(1) = O(n)$

$= O(\sqrt{n})$
Note: Analysis is probably pessimistic...

bounded the # of regions intersecting an edge of the query rectangle by the # regions intersecting line through its edges.

⇒ if range is small, so will be edges & won't intersect this many

Generalization to Higher Dimensions

Construction Similar: one level for each dimension

Storage $O(d \cdot n)$  time $O(d \cdot n \cdot \log n)$

Query time $O(n^{1-\frac{1}{d}} + k)$
Range Trees

2D-Kd-Tree
storage: \( O(n) \)
query time: \( O(\sqrt{n} + k) \)

2D-Range Tree
storage: \( O(n \log n) \)
query time: \( O(\log \frac{n}{k} + k) \)

Observation: A 2D-range query is composed of two 1D sub-queries.

- Kd-Tree uses this obs + splits point set alternately on \( x^- + y \)-coordinate...

- Range Tree uses it differently
  - build one binary search tree \( T_x \) on \( x \)-coordinates only (first-level tree)
    - for each internal node \( v \in T_x \), let \( P(v) = \) set of points stored in leaves of subtree
      \( \Rightarrow \) set \( P(v) \) is stored w/ \( v \) as another rooted at \( v \) balanced binary search tree \( T_y(v) \) (second level tree)
      on \( y \)-coordinate. (have ptr from \( v \) to \( T_y(v) \))
Build 2D-Range Tree (P)

input: a set P of points in the plane
output: root of a 2d range tree

1. Construct 2nd level tree $T_y$ for P
   (store entire pt at leaves)
2. if ($|P| = 1$)
3. then create leaf $v$ and make $T_y$ associate
4. else Split $P$ in $P_{left} + P_{right}$ of equal size by $x$ coord around $x_{mid}$
5. $V_{left} := \text{Build 2d Range Tree}(P_{left})$
6. $V_{right} := \text{Build 2d Range Tree}(P_{right})$
7. New $v$ s.t. $x_v := x_{mid}$
   \[ l(v) := V_{left} \]
   \[ r(v) := V_{right} \]
   \[ T_y(v) := T_y \]
8. return v

end /* Build 2d Range Tree */

Lemma 5.6 A 2d range tree w/ $n$ points uses $O(n \log n)$ storage

**Proof**
- Consider a point $p \in P$.
- $p$ is stored in $T_y(v)$ for every node $v \in T_x$
  s.t. $p$ is a leaf of the subtree of $T_x$ rooted at $v$
- There are $O(\log n)$ such subtrees $T_x$ \(\Rightarrow\) those rooted
  on $\text{root}(T_x)$ to $p$ path. ($T_x$ has height $O(\log n)$)
\(\Rightarrow\) each pt stored $O(\log n)$ times
\(\Rightarrow\) total $n$ pts $= O(n \log n)$ Storage total.
Construction Time for 2D Range Tree

- naive implementation of step 1 takes $O(n \log n)$ time (unsorted pts.)
- can instead pre-sort points on $y$-coordinate.
- construct maintain points sorted by $x$- and $y$-coord.
- but, if points already sorted by $y$-coordinate
  - can build binary search tree in $O(n)$ time (bottom up)
- pre-sort pts by $x$- and $y$-coordinates
- build trees bottom up & maintain sorted lists
- construction of trees $T_y$ takes $O(n')$ time ($n' = \#$ points in $T_y$)
- total construction time (= storage) = $O(n \log n)$

Queries in 2D Range Trees

- first determine $O(\log n)$ subtrees to search (those with $x$-coord in range)
- search each such subtree $T_y$ for points in $y$-coord range.
  - both above steps use 1D search algorithm.
- $\Rightarrow$ so alg identical to 1DRangeQuery except replace calls to ReportSubtree by a 1DRangeQuery on y-coords.

Lemma 25.3: Two query in 2014
Lemma 5.7 A query w/ axis-parallel rectangle in range tree for n points takes \(O(\log^2 n + K)\) time, where \(K = \#\) reported pt.

Proof

- Spend \(O(\log n)\) time searching 1st level tree \(T_x\) for each 1D range query in a second level tree
- Spend \(O(\log n + K_v)\) time searching 2nd level tree \(T_y(v)\)

\[ \# pt\ in \ T_y(v) \rightarrow \# reported\ points\ in \ T_y(v) \]

- total time is

\[ \sum_{v} O(\log n + K_v) \quad (\text{since } \log n_v = O(\log n)) \]

where summation is over all visited nodes \(v\).

- \[ \sum_{v} K_v = K \quad (\text{total \# nodes reported}) \]

- \[ \sum_{v} O(\log n) = O(\log^2 n) \quad \text{since \(O(\log n)\) nodes \(v\) visited when searching } T_x \]

\[ \Rightarrow \text{total} = O(O(\log^2 n + K) \checkmark) \]

\[ \square \]

Note: Can improve query time to \(O(\log n + k)\) using fractional cascading technique.
Higher Dimensional Range Trees

generalize 2D Range trees

- 1st level tree is balanced binary search tree on 1st coordinate
- 2nd level tree is (d-1)-dimensional range tree for P(v)
  - restricted to last (d-1)-coordinates of points
  - this tree constructed recursively
  - last tree is 1D balanced binary search tree on dth coordinate

Query

- generalize naturally
  - at first level select \( O(\log n) \) 2nd level trees to query
  - for each tree on 2nd level, select \( O(\log n) \) 3rd level trees to query
  - for each tree on (d-1)st level, select \( O(\log n) \) dth level trees to query

\[ \Rightarrow \text{in total, will query } O(\log^{d-1} n) \text{ trees to query} \]
\[ \Rightarrow \text{each tree } \text{BST query takes } O(\log n) \text{ time} \]
\[ \Rightarrow O(\log^d n) \text{ query time total } + O(k) \text{ to report pts.} \]

\[ Q_d(n) = O(\log n) + O(\log n) Q_{d-1}(n) \]

where \( Q_d(n) = O(\log^{2d} n) \). \( \Rightarrow Q_d(n) = O(\log^d n) \).

Total Query time \( O(\log^d n + k) \) (can be improved by log factor by best cascading)
Lemma 5.9. A range tree uses $O(n \log^{d-1} n)$ storage and can be constructed in $O(n \log^{d-1} n)$ time, $d \geq 2$.

Proof

$T_d(n) = \text{construction time for range tree on } n \text{ pts in } d\text{-dimensions}$.

- $T_2(n) = O(n \log n)$ (we've seen before)
- Construction involves $O(n \log n)$ time for 1st level tree on 1st coord plus time to construct assoc. $2$-trees (on remaining coords)

\[
T_d(n) = O(n \log n) + O(\log n) T_{d-1}(n)
\]

- time to const. level 1 tree
- # level 2 trees
- time to construct (d-1)-dim range trees
- time to build 2-tree range tree

= $O(n \log^{d-1} n)$ construction time

The Space is

\[
S_d(n) = O(n) + O(\log n) S_{d-1}(n)
\]

- Space for each pt
- Space for level 1 tree
- $S_d(n)$ appears in $O(\log n)$
- (d-1)-dimensional range trees

= $O(n \log^{d-1} n)$ storage
General Sets of Points

So far we have assumed all x-coords + y-coords are unique
that is

**Trick to ensure Phi is true**

\[ p = (p_x, p_y) \Rightarrow p' = (p_x'p_y, p_y'p_x) \]

if composite number space

+ use lexicographic order here

\[ (a, b) < (a', b') \iff (a < a') \text{ or } (a = a' \text{ and } b < b') \]

Note all \( p_x, p_y \) (and \( p_y, p_x \)) are distinct now since unique p'ts
transform query range too...

\[ R = [x, x'] \times [y, y'] \Rightarrow R' = [(x-\infty, x+\infty] \times [y-\infty, y+\infty] \]

**Lemma 5.10** \( p \in R \iff p' \in R' \)

**Proof:**

\( \Rightarrow \) Assume \( p \in R \).

Then \( x \leq p_x \leq x' \) and \( y \leq p_y \leq y' \)

\( \Rightarrow (x-\infty) \leq p_x'p_y \leq (x+\infty) \) \( \text{AND} \) \( (y-\infty) \leq p_y'p_x \leq (y+\infty) \)

\( \Leftarrow \) Assume \( p \notin R \)

So either \( p_x < x \) or \( x' < p_x \) or \( p_y < y \) or \( y' < p_y \).

If \( p_x < x \), then \( (p_x'p_y) < (x-\infty) \Rightarrow p' \notin R \)

Similarly for other 3 cases.
Fractional Cascading

- Query time for 2d-Range Tree is $O(\log^2 n + K)$
- Can reduce this to $O(\log n + K)$ using fractional cascading technique

Recall: query in 2d Range Tree $[x, x'] \times [y, y']$

1. Locate $O(\log n)$ nodes in primary binary search tree $[x, x']$
2. Search each subtree associated with each of the nodes $[y, y']$
   - These are binary search trees on $y$ coord.
   (Note: the values stored in each subtree are a subset of those stored in parent)

Search in 2 takes $O(\log n + K)$ time

⇒ good if we could do them in $O(1 + k_v)$ time
we'd do queries in $O(\log n + K)$ time total.

In general this not possible in general... but...

- Since we do many searches w/ same range we can use results to speed them up!

**Idea:** Let $S_2 \subseteq S$, and suppose stored in sorted order in $A_2 + A_2$.

```
A_1: 3 10 19 23 30 37 59 62 70 80 100 105
```

Add ptrs from $A_1$ to $A_2$:

* ptr from $A_1[i]$ to element of $A_2$ w/ smallest value larger than $A_1[i]$
  (ptr from $A_1[i]$ to $A_2[j] = \min j$ s.t. $A_2[j] \geq A_1[i]$)
Search for objects in \( S_1 + S_2 \) in range \([y, y']\):

- Binary search for \( y \) in \( A_1 \) at \( A_1[i] \).
- Walk thru \( A_1 \) till find key \( z \) range than \( y' \).
- Follow ptr from \( A_1[i] \) to \( A_2 \).
- Walk thru \( A_2 \) till find key larger \( y' \).

Key:
- \( \Rightarrow \) avoid 2nd binary search & spend \( O(1+k) \) time to report \( k \) values in \( S_2 \).

Range Trees (key: \( P(lo(v)) < P(v) \) and \( P(lo(v)) < P(v) \))

- Store \( P(v) \) in ysorted Array \( A_v \).
- Each element \( A_v[i] \) stores:
  - \( p \) to element in \( A_{v(lo)} \) larger than \( A_v[i] \).
  - \( p \) to smallest element in \( A_{v(lo)} \) larger than \( A_v[i] \).

Queries:
- As before, search 1st level tree for \( O(\log n) \) nodes whose
  - subtrees contain range \([x, x']\).
- At \( V_{split} \) (where paths for \( x + x' \) split), find entry
  - in \( A_{v_{split}} \) whose y-coord is smallest one \( z \geq y \)
  - (this takes \( O(\log n) \) time).
- As we search further for \( x + x' \), we continue
to follow ptrs from \( A_{v_{split}} \) into
  - \( A_{v_{split}} + A_{v_{split}} \) (\( O(1) \) time - avoid subsequent binary searches).
- To report pts, just walk in arrays from starting pt.
  - Takes \( O(1+k) \) time for each subtree.

Query time = \( O(\log n + k) \).
NOTE: Fractional Cascading also improves query time of higher-dimensional queries by logarithmic factor. (use on the 2-dimensional queries)

**Theorem 5.11** A layered range tree for n pts P in d-dimensions, d \geq 2, can be
- constructed in \(O(n \log^{d-1} n)\) time
- uses \(O(n \log^{d-1} n)\) space
- queries in \(O(\log^{d-1} n + k)\) time.