Orthogonal range queries

- Given: a set of points $p_1 \ldots p_n$
- Goal: preprocess the points so that later, given a square $T = [x_l, x_r] \times [y_l, y_r]$, we can quickly check if there is $p_i \in T$

We ignore the preprocessing time, for simplicity.

Discretization

In general, coordinates of $p_i$'s and $T$'s are reals. But during last lecture, we showed it suffices to assume they are integers from $\{1 \ldots n\}$, by adding

- $O(n)$ space (to store a successor data structure)
- $O(\log n)$ (to "discretize" the rectangle $T$ by finding successors)

From now on, can assume coordinates are from $\{1 \ldots n\}$

Solution I

Idea: Since all four parameters of a query $T$ are numbers from $\{1 \ldots n\}$, there are at most $n^4$ queries. Therefore, we can precompute answers to all queries and store it in a lookup table. This gives:

- $O(1)$ query time (plus the $O(\log n)$ discretization time)
- $O(n^4)$ space (to store all answers)

Query time is good, but space is huge, especially when compared to, say, Binary Search Trees ($n$ space, $\log n$ time). Will try to get the space down.
Solution II

Ideas:
- Decompose the data set into smaller sets, each equipped with a data structure solving “simpler” queries
- Each query can be answered using $O(\log n)$ “simpler” queries
- The total space used by all data structures is small
Now we just need to implement it...

Data structure for simpler queries

A query $T$ is simple if it is of the form

$$\{ -\infty \ldots \infty \} \times \{ y_l \ldots y_r \}$$

I.e., does not specify any constraints on $x$-coordinates. But then, the problem is just 1-dimensional, since only the $y$-coordinates matter!

How to solve it:
- Take the $y$-coordinates $y_1 \ldots y_m$
- Build a successor-supporting data structure
- To check if there is any $y_i$ falling into the query interval $\{ y_l \ldots y_r \}$, check if the successor of $y_i$ is $\leq y_r$

Data structure for simpler queries, ctd.

The structure requires
- $O(\log m)$ query time (when Binary Search Trees are used)
- $O(m)$ space
Now we just need to “lift” the data structure to higher dimensions....

Dyadic intervals

Consider an interval $\{1 \ldots n\}$, assume $n = 2^k$. A dyadic interval is:
- the interval $\{1 \ldots n\}$
- intervals $I_l$ and $I_r$ obtained by splitting a dyadic interval into two equal parts. I.e., if $I = \{a \ldots b\}$, then

$$I_l = \{a \ldots (a + b - 1)/2\}$$

and

$$I_r = \{(a + b + 1)/2 \ldots b\}$$

The length of a dyadic interval is always a power of 2.

Example dyadic intervals:

__________

____   ___  ____  _____

et cetera
The power of dyadic intervals

Basic fact of life: any interval $I = \{a \ldots b\} \subset \{1 \ldots n\}$ can be decomposed into a union of $\leq 2 \log n$ dyadic intervals.

**Proof**: For simplicity, focus on intervals of the form \(1 \ldots y\), \(y < n\). Run the following algorithm (similar to binary search):

1. Set $L = 1$, $R = n$
2. While $L \neq R$
   - Compute $M = (l + r - 1)/2$
   - If $M \leq y$, report interval \(\{L \ldots M\}\) and set $L = M + 1$
   - Otherwise, set $R = M$

For general intervals algorithm is similar but a bit more complex.

The decomposition of point-set

- For any $i = 1 \ldots \log n$
  - For any dyadic interval $I$ of length $2^i$, construct the set of points $P_I = P \cap (I \times \{1 \ldots n\})$
  - Prepare a data structure solving simple queries for the set $P_I$

The decomposition of query rectangle

In order to answer query $T = I_x \times I_y$:

- Decompose the interval $I_x$ into $\leq 2 \log n$ dyadic intervals $I_1 \ldots I_l$
- For each such interval $I_i$, query the data structure designed for $I_i$ with a query $\{-\infty, \infty\} \times I_y$
- If any query returned YES, return YES. Otherwise, return NO

Complexity

- $2 \log n \cdot O(\log n)$ query time, since each simple query takes $O(\log n)$ time and we perform $\leq 2 \log n$ of them
- $O(n \log n)$ space

<table>
<thead>
<tr>
<th>Structure</th>
<th>Space</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do nothing</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Solution I</td>
<td>$O(n^4)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Solution II</td>
<td>$O(n \log n)$</td>
<td>$O(\log^2 n)$</td>
</tr>
</tbody>
</table>

Solution II is “the best of both worlds”.

The query time can be made $O(\log n)$, but it is difficult.