E0 208: Computational Geometry

Naman Mishra

August 2024

Contents

Lectures

The course

Course webpage

MS Teams: 9ng333s

Lecture 1.
Wednesday
August 7

Grading

Tentatively

(30%) 4 assignments

(30%) 1 midterm

(40%) Final / project

Overview

Applications to

- Robotics
- VLSI
- Databases (spatial)
- Machine learning

Examples.

- **Robotics:** Path planning
- **Databases:** Given a set of restaurants with ratings, find the top k restaurants within a distance r from a given location.
  - Processing a single query takes $O(n)$,
  - and preprocessing leads to space issues.

It seems we are fed.

- **Nearest-neighbour query:** Given a set of n points and a query point q, return the point closest to q. (Voronoi diagram)

The first half focuses on these data structure based problems. The second half focuses on geometric optimization queries.

Examples.

- **Travelling salesman** in the Euclidean setting.
- **Set cover:** Given a set of points and a set of disks, find the smallest set of disks that covers all points.
- **Convex hull/Skyline points/Arrangement of lines**

Conferences

- **SoCG:** Symposium on Computational Geometry
- **CCCG:** Canadian Conference on Computational Geometry

Chapter I

Skyline points

Definition I.1 (Domination and skyline)**.** A point $p = (p_1, p_2, \ldots, p_d)$ *dominates* another point $q = (q_1, q_2, \ldots, q_d)$ if $p_i > q_i$ for all $i \in [d]$.

A point p in a set S is a *skyline point* if no other point in S dominates it.

Question I.2. *Given a set of points in $\mathbb{R}^2$, find the set of skyline points.*

Solution. Sort the points by decreasing x-coordinate. Iterate through the list keeping track of the maximum y-coordinate seen so far (sweep left). If the current point has a y-coordinate less than the maximum, it is a skyline point.

```
SKYLINE(P):
    sort P by decreasing x-coordinate
    S ← ∅
    y_max ← −∞
    for p ∈ P
        if p_2 > y_max
            S ← S ∪ {p}
            y_max ← p_2
```

■

Question I.3. *Given a set of points in $\mathbb{R}^3$, find the set of skyline points.*

Solution. Sort P by decreasing z-coordinate. Whether a point is a skyline point depends only on points before it. Sweep through P, while maintaining the solution set S, and the 2D skyline for the projection of each point in S onto the xy-plane.

This 2D skyline is to be stored in a (balanced) binary search tree according to the x-coordinate. Each time a new point p_i is seen, we query the tree for the successor a_j of p_i. Then p_i is a skyline point iff $(p_i)_y > (a_j)_y$.

If p_i is a skyline point, insert it into the tree, and delete all points dominated by p_i. $O((1+k)\log n)$ time for deleting k points.

Time complexity is

$$\sum_{i=1}^{n} O(\log n) + k_i \cdot O(\log n) = O\left(\sum_{i=1}^{n} \log n\right) + \left(\sum_{i=1}^{n} k_i\right) O(\log n)$$
$$= O(n \log n + n \log n) = O(n \log n)$$

```
3DSkyline(P):
  P ← P ∪ {(−∞, +∞, −∞), (+∞, −∞, −∞)}
  sort P by decreasing z-coordinate
  S ← ∅, T ← create(P)
  for p ∈ P
      if succ(T, p)_y < p_y
          S ← S ∪ {p}
          add(T, p)
      while (q := pred(T, p))_y < p_y
          remove(T, q)
  return S
```

■

Lecture 2. Monday August 12

If the number of skyline points is much smaller than the number of points, we can do better than $O(n \log n)$.

In this lecture, we will figure out an $O(n \log k)$ algorithm, where k is the number of skyline points in the input.

Warm-up: If $k = 1$, is there a better algorithm than $O(n \log n)$?
Answer: Yes. Simply pick the maximum x-coordinate.

I.1 A slow algorithm

Question I.4. *Can we find an $O(nk)$ algorithm for the skyline problem? (assuming d to be constant)*

Solution. Pick the largest x-coordinate, and remove all points that are dominated by it. Repeat until the set is exhausted.

The next point chosen is not dominated by the previous one, hence it is not dominated by any other deleted point. Since it has the largest x-coordinate of the remaining points, it cannot be dominated by those either. ■

We will try to make this faster later.

I.2 Chan's algorithm

Question I.5. *Can we find an $O(n \log k)$ algorithm for the skyline problem in $\mathbb{R}^2$?*

We will first solve the following decision problem:

Question I.6. *Given a set of points P having k skyline points and an integer $\widehat{k}$, decide*

- *if $\widehat{k} \geq k$, output the skyline,*
- *if $\widehat{k} < k$, output **failure**.*

Solution. Partition P into roughly $\widehat{k}$ slabs of roughly equal size using the median of medians algorithm.

(1) Compute the median and partition the elements in $O(n)$ time.

(2) Repeat recursively on the two halves, upto a depth of $\log k$.

In total, this takes $O(n \log k)$ time. (One remarked that it's like quicksort, but stopping once we have exhausted our time budget.)

Visit each slab in order of decreasing x-coordinate.

```
CHAN-DECISION-2D(P, k̂):
  P_1, ..., P_k̂ ← PARTITION(P)
  S ← ∅
  y(p*) ← −∞
  for i = k̂ downto 1
      P'_i ← {p ∈ P_i | y(p) > y(p*)}
      S(P'_i) ← SKYLINE(P'_i)   ▷ using the slow algorithm
      report S(P'_i)
      S ← S ∪ S(P'_i)
      if |S| > k̂
          return ⊥
```

Failure is not because of our inability to output the correct answer, but because the runtime would be too large. ■

Analysis. Constructing the slabs took $O(n \log \widehat{k})$ time.

Running the slow algorithm on P_i takes $\frac{n}{\widehat{k}} \cdot k_i$ time, where k_i is the number of skyline points in P_i. In total, this step takes

$$\sum_{i=1}^{\widehat{k}} O\left(\frac{n}{\widehat{k}} \cdot k_i\right) = O\left(\frac{n}{\widehat{k}} \cdot \sum_{i=1}^{\widehat{k}} k_i\right) = O(n)$$

time. Thus the total runtime is $O(n \log \widehat{k})$. ■

We are now prepared to solve the original problem.

Solution. Guess $\widehat{k} = 1, 2, 4, \ldots, 2^{\lceil \log k \rceil}$. This will take

$$\sum_{i=0}^{\lceil \log k \rceil} O(n \log 2^i) = \sum_{i=0}^{\lceil \log k \rceil} O(ni) = O(n(\log k)^2)$$

time. We can be even more aggressive and guess $\widehat{k}_i = 2^{2^i}$ (square the previous guess). This takes

$$\sum_{i=0}^{\lceil \log \log k \rceil} O(n \log 2^{2^i}) = \sum_{i=0}^{\lceil \log \log k \rceil} O(n2^i) = O\big(n2^{\lceil \log \log k \rceil}\big) = O(n \log k)$$

time.

```
CHAN-2D(P):
  k̂ ← 2
  for ever
      S ← CHAN-DECISION-2D(P, k̂)
      if S ≠ ⊥
          return S
      square k̂
```

■

Question I.7. *Can we generalize this to* $\mathbb{R}^3$*?*

It suffices to generalize the decision problem.

Solution. We construct $\widehat{k}$ slabs based on the x-coordinate. The invariants are the following:

- S_i will store the skyline of the first i slabs.
- S_i^{yz} will store the 2D skyline of the projection of S_i onto the yz-plane.
- Each S_i and S_i^{yz} is sorted according to the y-coordinate.

The deletion step now takes $|P_i| \log |S_i^{yz}|$ time, (as opposed to $|P_i|$ earlier) which is still fine.

We now run the slow algorithm and report the elements added to S_i. This takes $O(n)$ time, and they are naturally reported in decreasing order of the y-coordinates.

```
CHAN-DECISION(P, k̂):
  P_1, ..., P_k̂ ← PARTITION(P)
  S ← ∅, S^yz ← ∅
  for i = k̂ downto 1
      P'_i ← FILTER(P_i, S^yz)
      S(P'_i) ← SKYLINE(P'_i)   ▷ using the slow algorithm
      report S(P'_i)
      S ← S ∪ S(P'_i)
      if |S| > k̂
          return ⊥
```

■

Chapter II

Orthogonal range searching

Lecture 3.
Wednesday
August 14

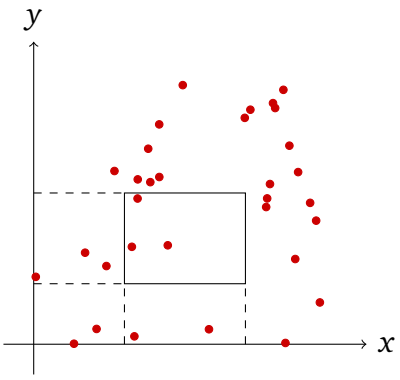


Figure II.1: Orthogonal range searching in $\mathbb{R}^2$.

Question II.1. *Given a set of points in $\mathbb{R}^2$ and a rectangle $[x_1, x_2] \times [y_1, y_2]$, report all points in the rectangle.*

Without preprocessing, we have

- $O(n)$ space,
- $O(n)$ time.

In the worst case, we can have $O(n)$ points in the rectangle, so we cannot do better than $O(n)$ time.

Let k be the number of points in the rectangle. Usually, $k \ll n$. Thus we will design a data structure that allows us to report the k points in $O(f(n)+k)$ time, where $f(n)$ is as small as possible. Of course, this requires preprocessing.

Figure II.2: Otherogonal range searching in $\mathbb{R}$.

II.1 The 1-dimensional case

Question II.2. *Given a set of points in $\mathbb{R}$, process queries of the form $[x_1, x_2]$, reporting all points in the interval.*

Solution. Sort the points in $O(n \log n)$ time using only $O(n)$ space. For each query, binary search for x_1 and keep going until x_2. This takes $O(\log n + k)$ time. ■

II.2 The 2-dimensional unbounded case

Question II.3. *Given a set of points in $\mathbb{R}^2$, process queries of the form $[x_1, x_2] \times [y_1, \infty)$, reporting all points in the rectangle.*

Naive. Project onto the x-axis and solve the 1-dimensional case. Discard all points with y-coordinate less than y_1.

This takes $O(n \log n)$ preprocessing time. For each query, it takes $O(\log n + \#[x_1, x_2])$ time, where $\#[x_1, x_2]$ is the number of points with x-coordinate in $[x_1, x_2]$. ■

This can be much larger than k, in fact, as large as n.

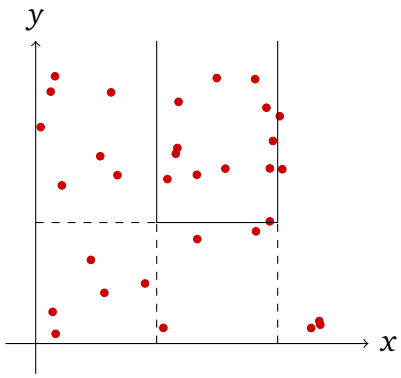


Figure II.3: Unbounded 3-sided range searching in $\mathbb{R}^2$.

II.2.1 Priority search trees

A priority search tree is a binary search tree where each node has an associated priority.

Solution. Create a *priority search tree* (PST) in $O(n \log n)$ time using $O(n)$ space, and sorted by x-coordinate with y-coordinate as priority.

Each query can be answered as follows.

- Locate the predecessor of x_1 and successor of x_2. To ensure they exist, add sentinels $-\infty$ and ∞ during preprocessing.
- Query each subtree in between with a DFS, searching down until the y-coordinate is less than y_1. ■

Analysis. Querying the subtree $\mathscr{T}$ takes $O(1 + k_{\mathscr{T}})$ time, where $k_{\mathscr{T}}$ is the number of rectangled points in the subtree. This is because each rectangled point is visited exactly once, and any non-rectangled point that is checked is preceded by a rectangled point.

Thus the total time is

$$\begin{aligned}\sum_{\mathscr{T}} O(1 + k_{\mathscr{T}}) &= O\left(\sum_{\mathscr{T}} 1\right) + O(k) \\ &= O(\log n + k).\end{aligned}$$

This is because the each subtree arises from a point on the search path for the predecessor or successor, which has length at most $\log n$.

```
PST-UNBOUND(P):
    𝒫 ← PST(P ∪ {(−∞, ⊥), (∞, ⊥)})
    u ← root(𝒫), v ← root(𝒫)
    while u = v
        TODO
```

■

Question II.4. *Given a set of colors $\mathscr{C}$ and a set of points in $\mathbb{R}$ each labeled with a color, process queries of the form $[x_1, x_2]$, reporting all colors of points in the interval.*

Lecture 4. Monday August 26

Let t be the number of points in $[x_1, x_2]$, and k be the number of distinct colors of these t points.

Naive. Go through each point in $[x_1, x_2]$ and store the colors in a set. Complexity: $O(\log n + t)$. ■

This is bad if $t \gg k$. Our goal is to achieve $O(\log n + k)$ time. We will try to check only one point per color in $[x_1, x_2]$. We will do this by only getting the first point.

Solution. For each point x with color c, define $pred(p)$ to be the immediate predecessor of x with color c. The map

$$x \mapsto (x, pred(x))$$

reduces this to the 2-D unbounded range searching problem. All we need to do is report every point in $[x_1, x_2] \times (-\infty, x_1)$.

This takes $O(\log n + k)$ query time, $O(n)$ space, and requires $O(n \log n)$ time for preprocessing.

- Partitioning by colors takes $O(n)$ time.
- Sorting and computing predecessors takes $\sum_{c \in \mathscr{C}} O(n_c \log n_c) = O(n \log n)$ time.
- Constructing the priority search tree takes $O(n \log n)$ time.

■

II.3 Top k reporting

AAAAAA

Midterm 1: September 9th.
Homework 1 will be posted by Wednesday.

Lecture 5. Monday, August 26
Lecture 6. Monday, August 26

Question II.5. *Let $L_1, L_2, \ldots, L_t$ be t sorted lists, with $\sum_{i=1}^{t} |L_i| = n$. Given a query q, report the successor of q in each list.*

The naive binary search approach is $O(t \log n)$, but with some preprocessing we can better this to $O(\log n + t)$.

Definition II.6. For any list L, we define Even(L) to be the list of all even-indexed elements of L.

Solution. For each i we define the list L'_i as follows.

$$L'_t = L_t$$
$$L'_i = \text{merged}(L_i, \text{Even}(L'_{i+1}))$$

For example, if

$$L_1 = \begin{bmatrix} 2 & 3 & 7 & 16 & 19 & 32 & 36 & 37 & 48 & 51 \end{bmatrix}$$
$$L_2 = \begin{bmatrix} 15 & 25 & 30 & 35 & 40 & 45 \end{bmatrix}$$
$$L_3 = \begin{bmatrix} 5 & 10 & 17 & 20 & 27 & 50 \end{bmatrix}$$

then

$$L_3' = \begin{bmatrix} 5 & 10 & 17 & 20 & 27 & 50 \end{bmatrix}$$
$$L_2' = \begin{bmatrix} 10 & 15 & 20 & 25 & 27 & 30 & 35 & 40 & 45 \end{bmatrix}$$
$$L_1' = \begin{bmatrix} 2 & 3 & 7 & 15 & 16 & 19 & 25 & 30 & 32 \\ 36 & 37 & 40 & 48 & 51 & & & & \end{bmatrix}$$

Let us formally show that the sum of the new lengths is still $O(n)$.

$$\begin{aligned} |L_t'| &= |L_t| \\ |L_{t-1}'| &\le |L_{t-1}| + \frac{1}{2}|L_t| \\ |L_{t-2}'| &\le |L_{t-2}| + \frac{1}{2}|L_{t-1}'| \\ &= |L_{t-2}| + \frac{1}{2}|L_{t-1}| + \frac{1}{4}|L_t| \\ |L_{t-i}'| &\le |L_{t-i}| + \frac{1}{2}|L_{t-i+1}'| \\ &= |L_{t-i}| + \frac{1}{2}|L_{t-i+1}| + \frac{1}{4}|L_{t-i+2}| + \cdots + \frac{1}{2^i}|L_t| \end{aligned}$$

Thus the sum of lengths is

$$\sum_i |L_i'| < 2 \sum_i |L_i|$$

Call the elements that have cascaded up (Even(L_{i+1}')) the *cascaders*.

While merging L_i with Even(L_{i+1}'), store a pointer, which we will call the *cascader pointer*, from each element $x \in L_i'$ to the largest element $\le x$ in Even(L_{i+1}'). In our example, we store the pointers

2 3 7 15 16 19 25 30 32 36 37 40 48 51

10 15 20 25 27 30 35 40 45

5 10 17 20 27 50

■

II.4 Back to range trees

2-D range trees store a sorted list or balanced binary search tree at each node. We can adapt fractional cascading to 2-D range trees. Since all internal lists are to be queried by the same y-coordinates y_1 and y_2, this allows us to shave off one log factor.

For d-dimensional range trees, which ultimately contain 2-D range trees at some level, this still reduces one log factor. Thus the total time complexity is $O(\log^{d-1} n + k)$.

Lecture 7. Wednesday August 28

Only one midterm, in the middle of September. Retcon: Midterm 1 is on September 30th.

In the pointer machine model, memory is organised in a graph. Chazelle (1990) showed that in a pointer machine model, any algorithm for range reporting that achieves $O(\log^c n + k)$ time must use $\Omega\left(n \frac{\log n}{\log \log n}\right)$ space.

We will achieve this bound in today's lecture.

Question II.7. *Given a set of points in $\mathbb{R}^2$ and an $x_0 \in \mathbb{R}$, process queries of the form $[x_1, x_2] \times [y_1, y_2]$ where $x_1 < x_0 < x_2$.*

Solution. Partition the set of points P into $P_L = P \cap ([-\infty, x_0) \times \mathbb{R})$ and $P_R = P \cap ([x_0, \infty) \times \mathbb{R})$. Use the 3-sided range reporting algorithm to report the points in $[x_1, \infty) \times [y_1, y_2]$ from P_L and $(-\infty, x_2] \times [y_1, y_2]$ from P_R.

This takes $O(\log n + k)$ time and consumes $O(n)$ space. ■

We can generalize this to work for any query, by using a balanced binary search tree on the x-coordinates of the points.

Solution. Let x_m be the median x-coordinate in P.

Partition P into P_L and P_R as before. Recurse on P_L and P_R. For each node, construct the data structure described above: two priority search trees containing all descendants of the node. The space requirement is given by the recursion

$$S(n) \leq O(n) + 2S(n/2)$$

which has solution $S(n) = O(n \log n)$. This can be thought of as $O(n)$ space per level of the tree. The construction time is given by the recursion

$$T(n) \leq O(n \log n) + 2T(n/2)$$

which has solution $T(n) = O(n \log^2 n)$.

For each query, locate the highest node in the tree that intersects the query rectangle. This takes $O(\log n)$ time. Use the data structure at that node to report the entire rectangle in $O(\log n + k)$ time. ■

The key idea is to use a search tree with larger fanout to reduce the height. A tree with fanout f has height $O(\log_f n)$. For a fanout of $\log n$, this is $O\left(\frac{\log n}{\log\log n}\right)$.

Let us look at the special case question II.7 first.

Question II.8. *Given a set of n points in $\mathbb{R}^2$ and $\ell_1, \ldots, \ell_{F-1} \in \mathbb{R}$, process queries of the form $[x_1, x_2] \times [y_1, y_2]$ where (x_1, x_2) contains at least one ℓ_i.*

Solution (naive). Split the points into F slabs $L_1, \ldots, L_F$ by the lines $x = \ell_1, \ldots, \ell_{F-1}$.

For each slab, build a priority search tree. Any query $[x_1, x_2] \times [y_1, y_2]$ will overlap with $L_i, \ldots, L_j$ for some $i < j$. Run the 3-sided range reporting algorithm on each of these slabs. This takes $O(n)$ space and $O(F \log n + k)$ query time. Oops! ■

(diagram) The range is three-sided only in the slabs L_i and L_j. The middle slabs are essentially 1-dimensional queries. Thus we can do better.

Solution. For each slab, build a PST. In addition, maintain a fractional cascading structure on the slabs, where each point is projected onto the y-axis. The total space occupied is still $O(n)$.

Any query $[x_1, x_2] \times [y_1, y_2]$ will overlap with $L_i, \ldots, L_j$ for some $i < j$. The intersection will be 3-sided only with L_i and L_j. Solve these using the PSTs in $O(\log n + k)$ time.

Use fractional cascading to obtain the successor of y_1 in each slab in the middle in $O(\log n + F)$ time. Keep going until y_2, in $O(k)$ time. The total time is $O(\log n + F + k)$. ■

If $F = O(\log n)$, we have $O(\log n + k)$ time using $O(n)$ space.

Finally, we can generalize this to any query.

Solution. Let $F = \lceil \log n \rceil$. Partition P into equal $P_1, \ldots, P_F$ by x-coordinate. Recurse on each P_i, storing F PSTs and a fractional cascade of F slabs at each node.

This consumes $O(n)$ space at each level, for a total of $O\left(\frac{n \log n}{\log\log n}\right)$.

At query time, we have two cases:

(Case 1) The query rectangle intersects at least one slab boundary. Solve the special case in $O(\log n + k)$ time.

(Case 2) The query rectangle is contained within a slab. Recurse into the slab.

Figuring out which case we are in takes $O(\log F)$ time. The tree has height $O\left(\frac{\log n}{\log\log n}\right)$, so getting to case 1 takes at most $O\left(\log F \cdot \frac{\log n}{\log\log n}\right) = O(\log n)$ time. Once we are in case 1, reporting requires $O(\log n + k)$ time. Thus we still have $O(\log n + k)$ time using the lower bound space. ■

Lecture 8. Monday September 2

To recap, we have solved 2D orthogonal range searching using

- **Range trees:** $O(n \log n)$ space and $O(\log^2 n + k)$ time, which we later improved to $O(\log n + k)$ using fractional cascading.
- **The crazy optimal:** $O\left(n \cdot \frac{\log n}{\log\log n}\right)$ space and $O(\log n + k)$ time.

What if we are limited to $O(n)$ space?

Question II.9. *Given a set of points in $\mathbb{R}^2$ and $O(n)$ space, process queries of the form $[x_1, x_2] \times [y_1, y_2]$.*

II.5 Kd-tree

Given a set of points $P \subseteq \mathbb{R}^2$, split them evenly by a vertical cut. Next, split the two halves by a horizontal cut. Repeat this process recursively. Create a tree with the cuts as nodes and the points as leaves.

The construction time is $O(n \log n)$.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The space used is $O(n)$.

$$S(n) = 2S\left(\frac{n}{2}\right) + O(1)$$

We can do this for any number of dimensions.

```
KD-TREE(d, P, a):
  if |P| = 1
      return LEAF(P)
  m ← MEDIAN({p.a | p ∈ P})
  P_ℓ ← {p ∈ P | p.a ≤ m}
  P_r ← {p ∈ P | p.a > m}
  T_ℓ ← KD-TREE(d, P_ℓ, a + 1 mod d)
  T_r ← KD-TREE(d, P_r, a + 1 mod d)
  return NODE(a, m, T_ℓ, T_r)
```

Solution. Store the points in a Kd-tree. To each internal node v, we have an associated rectangle $R(v)$ containing all points in the subtree rooted at that node.

For a query rectangle $Q = [x_1, x_2] \times [y_1, y_2]$, we traverse the tree from the root, while keeping track of the rectangles $R(v)$.

- At a leaf node, determine if the point is in the rectangle.
- At an internal node v

  (1) If $Q \cap R(v) = \varnothing$, ignore the subtree.

  (2) If $Q \supseteq R(v)$, report all points in the subtree.

  (3) If only part of $R(v)$ intersects Q, recurse on both children. Equivalently, the boundary of Q intersects $R(v)$.

```
KD-QUERY(v, Q, R = [x1, x2] × [y1, y2]):
  if v is a leaf
      report v if v ∈ Q and return
  if Q ∩ R = ∅
      return
  if v splits by x-coordinate
      R_ℓ ← [x1, v_x] × [y1, y2]
      R_r ← [v_x, x2] × [y1, y2]
  else
      R_ℓ ← [x1, x2] × [y1, v_y]
      R_r ← [x1, x2] × [v_y, y2]
  KD-QUERY(v_ℓ, Q, R_ℓ)
  KD-QUERY(v_r, Q, R_r)
```

■

Remark. Cases (2) and (3) are identical from the algorithm's perspective. Both involve recursing on both children. They are distinguished here for the purpose of analysis.

Analysis. We take $O(k)$ time across all nodes of type (2).

The parent of every node visited (including leaves) is of type (3), except the subtrees in case (2) already accounted for above. Thus the total number of nodes visited can be bounded by 2 times the number of type (3) nodes. We will bound this number.

How many nodes can intersect a line $x = L$? Fix a node v. If v is split by x-coordinate, at most one child intersects $x = L$. If v is split by y-coordinate, both children may intersect. Thus the number of intersecting nodes at most doubles every 2 levels. This gives a total of $O(2^{\frac{\log n}{2}}) = O(\sqrt{n})$ nodes. More formally,

$$I(n) \leq 3 + 2I\left(\frac{n}{4}\right),$$

which has solution $I(n) = O(\sqrt{n})$. The 3 is for the root and possibly both children, but only two of the grandchildren can intersect.

Thus there are at most $O(\sqrt{n})$ nodes of type (3). This gives total time $O(\sqrt{n}+k)$. ■

In the 3D case, the number of nodes doubles twice every 3 levels, giving $O(n^{2/3})$ nodes intersecting an axis-perpendicular plane, and a time complexity of $O(n^{2/3}+k)$. In d dimensions, we have $O(n^{1-1/d}+k)$ time.

$$I(n) \le (2^d - 1) + 2^{d-1} I(n/2^d) \implies I(n) = O(n^{\frac{d-1}{d}})$$

This is because whenever a cut is parallel to the place, there is no increase in the number of intersecting nodes.

II.6 Bootstrapping

Lecture 9. Wednesday September 4

Start with a basic data structure and keep obtaining an improved data structure. Our goal is to obtain, for each $\varepsilon > 0$, a data structure for 2D orthogonal range reporting that consumes $O(n)$ space and processes queries in $O(n^\varepsilon + k)$ time.

Lemma II.10. *Suppose there is a structure $\mathcal{T}$ which solves 2D orthogonal range reporting with space $S(n)$ and query time $Q(n)+O(k)$. Then, for any integer $\lambda \in [2, \frac{n}{2}]$, there exists a structure which uses*

$$\text{space} \le \lambda S\left(\left\lceil \frac{n}{\lambda} \right\rceil\right) + O(n)$$
$$\text{query time} \le 2Q\left(\left\lceil \frac{n}{\lambda} \right\rceil\right) + \lambda O\left(\log \frac{n}{\lambda}\right) + O(k)$$

Proof. Divide the points P into λ vertical slabs $P_1, P_2, \ldots, P_\lambda$ of size $\le \lceil \frac{n}{\lambda} \rceil$. Build a structure $\mathcal{T}_i$ for each slab P_i. Furthermore, sort the points in each slab by their y-coordinates.

To query, we locate the overlapping slabs $P_i, P_{i+1}, \ldots, P_{i'}$. We query P_i and $P_{i'}$ using $\mathcal{T}_i$ and $\mathcal{T}_{i'}$, and for each P_j in between, we query using the sorted list. This takes

$$O(\log \lambda) + 2Q\left(\left\lceil \frac{n}{\lambda} \right\rceil\right) + \lambda O\left(\log \frac{n}{\lambda}\right) + O(k)$$

time. ■

Question II.11. *Given a set of n points in the plane and an $\varepsilon > 0$, process queries of the form $[x_1, x_2] \times [y_1, y_2]$ using $O(n)$ space and $O(n^\varepsilon + k)$ query time.*

Solution. Use lemma II.10. We know that Kd-trees solve the problem with $O(n)$ space and $O(\sqrt{n}+k)$ query time. Choosing $\lambda = n^{1/3}$ gives the existence of a structure D_1 with

$$\begin{aligned}\text{query time} &\leq 2O\left(\left(\frac{n}{\lambda}\right)^{\frac{1}{2}}\right) + \lambda O(\log\frac{n}{\lambda}) + O(k)\\ &= O(n^{\frac{1}{3}} + n^{\frac{1}{3}}\log n^{\frac{1}{3}} + k)\\ &= \widetilde{O}(n^{\frac{1}{3}} + k)\end{aligned}$$

where $\widetilde{O}$ hides logarithmic factors.

We can now choose $\lambda = n^{1/4}$ to obtain a structure D_2 with

$$\text{query time} \leq \widetilde{O}(n^{\frac{1}{4}} + k).$$

Iterating this process $1 + \lceil \frac{1}{\varepsilon} \rceil$ times gives the desired structure. If $Q(n) = \widetilde{O}(n^{\frac{1}{\mu}})$ for some $\mu \in \mathbb{N}^+$, then choosing $\lambda = n^{\frac{1}{\mu+1}}$ gives a structure with $Q(n) = \widetilde{O}(n^{\frac{1}{\mu+1}})$. At each iteration, space occupied stays $O(n)$. ■

We can analyze the space in terms of both n and ε.

Lemma II.12. *The space occupied by D_t is $O(nt)$.*

Proof. The space occupied by D_1 is $O(n)$. If $S_t(n) \leq Cnt$, then

$$\begin{aligned}S_{t+1}(n) &\leq \sum_{i=1}^{\lambda} S_t(n_i) + C'n\\ &\leq Cnt + C'n\\ &= Cn\left(t + \frac{C'}{C}\right).\end{aligned}$$

Choosing $C > C'$ gives the claim by induction. ■

In our case, we get space $O(n/\varepsilon)$.

Exercise II.13. *Generalise this to d dimensions.*

Lemma II.14 (Bootstrapping)**.** *Suppose there is a structure $\mathcal{T}$ which solves d-dimensional orthogonal range reporting with space $S(n)$ and query time $Q(n)+O(k)$, and a structure $\mathcal{T}'$ which solves $(d-1)$-dimensional orthogonal range reporting with space $S'(n)$ and query time $Q'(n)+O(k)$. Then for each $\lambda \in [2, \frac{n}{2}]$, there exists an algorithm which solves d-dimensional range reporting with*

$$\begin{aligned}\textit{space} &\leq \lambda S\left(\left\lceil \frac{n}{\lambda} \right\rceil\right) + \lambda S'\left(\left\lceil \frac{n}{\lambda} \right\rceil\right)\\ \textit{query time} &\leq 2Q\left(\left\lceil \frac{n}{\lambda} \right\rceil\right) + \lambda Q'\left(\left\lceil \frac{n}{\lambda} \right\rceil\right) + O(k)\end{aligned}$$

Proof. Divide the points P into λ vertical slabs $P_1, P_2, \ldots, P_\lambda$ of size $\leq \lceil \frac{n}{\lambda} \rceil$. Build structures $\mathscr{T}_i$ and $\mathscr{T}'_i$ for each slab P_i.

To query, we locate the overlapping slabs $P_i, P_{i+1}, \ldots, P_{i'}$. We query P_i and $P_{i'}$ using $\mathscr{T}_i$ and $\mathscr{T}_{i'}$, and for each P_j in between, we query using $\mathscr{T}'_j$. This takes up

$$\begin{aligned} \text{space} &\leq \lambda(S + S')\left(\left\lceil \frac{n}{\lambda} \right\rceil\right) \\ \text{query time} &\leq 2Q\left(\left\lceil \frac{n}{\lambda} \right\rceil\right) + (\lambda - 2)Q'\left(\left\lceil \frac{n}{\lambda} \right\rceil\right) + O(k) \end{aligned}$$

■

Note that lemma II.10 is a special case of this for $d = 2$, $\mathscr{T}'$ being a sorted list, $S'(n) = O(n)$, and $Q'(n) = O(\log n)$.

Solution to exercise II.13. Suppose that for each $\varepsilon > 0$ we have a structure $\mathscr{T}'$ which solves $(d-1)$-dimensional orthogonal range reporting with space $O(n)$ and query time $O(n^\varepsilon + k)$.

We know that Kd-trees solve the problem with $O(n)$ space and $O(n^{\frac{d-1}{d}} + k)$ query time, for d dimensions. Call this structure D_0.

Fix an $\varepsilon > 0$. Using bootstrapping with $\lambda_i = n^{\frac{d-1}{d+(d-1)i}}$, we obtain structures $D_1, D_2, \ldots$ with

$$\begin{aligned} S_n(D_i) &= O(n) \\ Q_n(D_i) &= O(n^{\frac{d-1}{d+(d-1)i} + \frac{\varepsilon}{2}}) \end{aligned}$$

The proof is by induction. This clearly holds for D_0. Suppose it holds for D_{i-1}, $i \geq 1$. Then,

$$\begin{aligned} S_i(n) &\leq \lambda_i S_{i-1}\left(\left\lceil \frac{n}{\lambda_i} \right\rceil\right) + \lambda_i O(n) \\ &= O(n) \\ Q_i(n) &\leq 2Q_{i-1}\left(\left\lceil \frac{n}{\lambda_i} \right\rceil\right) + \lambda_i O(n^{\frac{\varepsilon}{2}}) + O(k) \\ &= O\left(\left(\frac{n}{\lambda_i}\right)^{\frac{d-1}{d+(d-1)(i-1)} + \frac{\varepsilon}{2}}\right) + \lambda_i O(n^{\frac{\varepsilon}{2}}) \\ &= O\left(\left(n^{\frac{d+(d-1)(i-1)}{d+(d-1)i}}\right)^{\frac{d-1}{d+(d-1)(i-1)} + \frac{\varepsilon}{2}}\right) + O(n^{\frac{d-1}{d+(d-1)i} + \frac{\varepsilon}{2}}) \\ &= O\left(n^{\frac{d-1}{d+(d-1)i} + \frac{\varepsilon}{2}}\right) \end{aligned}$$

For large enough i, $Q_i(n) = O(n^\varepsilon)$, giving total query time $O(n^\varepsilon + k)$. The space remains $O(n)$.

By induction on d, we can obtain such a structure for any dimension d. ■

Chapter III

Point location

III.1 Interval stabbing

Lecture 10.
Monday
September 9

Question III.1. *Let $\mathscr{I}$ be a set of n intervals in $\mathbb{R}$. Process queries of the form $x \in \mathbb{R}$, reporting all intervals which contain x.*

Look at a special case first.

Question III.2. *Let $\mathscr{I}$ be a set of n intervals in $\mathbb{R}$ with $\bigcap \mathscr{I} \neq \varnothing$. Process queries of the form $x \in \mathbb{R}$, reporting all intervals which contain x.*

Solution. Let $p \in \bigcap \mathscr{I}$, and define

$$\mathscr{I}_\ell = \{I \cap (-\infty, p] \mid I \in \mathscr{I}\},$$
$$\mathscr{I}_r = \{I \cap [p, +\infty) \mid I \in \mathscr{I}\}.$$

Note that $\subseteq$ is a total order on each of $\mathscr{I}_\ell$ and $\mathscr{I}_r$. Sort both of them in descending order under $\subseteq$. WLOG let a query x be to the right of p. Report from $\mathscr{I}_r$ until the first interval which does not contain x.

This takes $O(n)$ space and only $O(1+k)$ query time. ∎

Generalize this via a binary search tree.

Solution. Let P be the set of endpoints of the intervals. Let x_m be the median x-coordinate in P.

Partition $\mathscr{I}$ into $\mathscr{I}_\cap$, $\mathscr{I}_L$, and $\mathscr{I}_R$ as follows:

(1) $\mathscr{I}_\cap = \{I \in \mathscr{I} \mid I \ni x_m\}$

(2) $\mathscr{I}_L = \{I \in \mathscr{I} \mid I \subseteq (-\infty, x_m)\}$

(3) $\mathscr{I}_R = \{I \in \mathscr{I} \mid I \subseteq (x_m, \infty)\}$

Use the special case for $\mathscr{I}_\cap$ (this requires storing sorted $\mathscr{I}_\cap^\ell$ and $\mathscr{I}_\cap^r$). Recurse on $\mathscr{I}_L$ and $\mathscr{I}_R$.

The height of the tree is $O(\log n)$, since $\mathscr{I}_L$ and $\mathscr{I}_R$ both contain at most n endpoints, and hence $n/2$ intervals.

$$H(n) \leq 1 + H(n/2) \leq \log n + 1$$

The space requirement is $O(n)$, and the query time is $O(\log n + k)$.

$$S(n) \leq O(k_v) + 2S(n/2)$$
$$T(n) \leq T(n/2) + O(1 + k_v)$$

where $\sum_v k_v = n$. The preprocessing time is $O(n \log n)$ (exercise). ■

We move on to 2 dimensions.

Question III.3. *Let $\mathscr{R}$ be a set of n **disjoint** rectangles in $\mathbb{R}^2$. Process queries of the form $x \in \mathbb{R}$, reporting the rectangle containing x, if any.*

Again consider the special case. This time, there is no simple total order.

Question III.4. *Let $\mathscr{R}$ be a set of n **disjoint** rectangles in $\mathbb{R}^2$ such that $\bigcap \pi_x(\mathscr{R}) \neq \varnothing$. Process queries of the form $x \in \mathbb{R}$, reporting the rectangle containing x, if any.*

Solution. Let L be a vertical line passing through each rectangle. Define

$$\mathscr{I} = \{\pi_y(R) \mid R \in \mathscr{R}\}$$

Any two intervals in $\mathscr{I}$ are disjoint, since $\pi_y(R_1) \cap \pi_y(R_2) \cong (R_1 \cap L) \cap (R_2 \cap L) = \varnothing$ Thus $\mathscr{I}$ can be totally ordered by their lower endpoints. Sort it.

For any query $q = (x, y)$, binary search through $\mathscr{I}$ to find any interval $I = \pi_y(R) \ni y$. At most one such interval exists, and the corresponding rectangle R is the only possible candidate.

This takes $O(n)$ space and $O(\log n)$ query time. ■

Generalize.

Solution. Let P be the set of x-endpoints of the rectangles. Choose the median. Partition $\mathscr{R}$ into $\mathscr{R}_\cap$, $\mathscr{R}_L$, and $\mathscr{R}_R$ as before. Use the special case for $\mathscr{R}_\cap$. Recurse on $\mathscr{R}_L$ and $\mathscr{R}_R$.

The height of the tree is $O(\log n)$. The space requirement is $O(n)$. The query time is $O(\log n)$ at each level, for a total of $O(\log^2 n)$. However, the $O(\log n)$ is only a binary search, with the same query at each node. We can apply fractional cascading yet again! This gives $O(\log n)$ query time.

In general, we reduce d-dimensional orthogonal point location to $d-1$ dimensions. The space requirement is $O(n)$ per level, and the query time is $O(\log^{d-1} n)$. ■

Lecture 11. Wednesday September 11

In the next few lectures, we will study segment trees and cover 2D rectangle stabbing, 2D general point location, and maximum enclosing rectangle (hotspot detection).

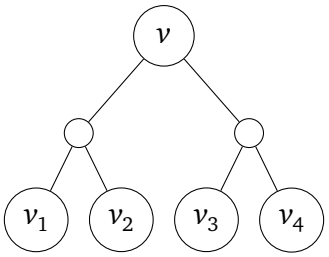
III.2 Segment tree

Definition III.5 (elementary intervals)**.** Let $\mathscr{I}$ be a set of n intervals. Let $p_1 < p_2 < \cdots < p_m$ be the distinct endpoints of $\mathscr{I}$ ($m \leq 2n$). They partition the real line into $2m+1$ *elementary intervals*,

$$(-\infty, p_1), \{p_1\}, (p_1, p_2), \{p_2\}, \ldots, \{p_m\}, (p_m, \infty).$$

Given a set of intervals $\mathscr{I}$, we can associate each elementary interval with the set of intervals that contain it. This takes $O(n^2)$ space, but $O(\log n + k)$ query time. Why would a tree be useful here?

Store the elementary intervals at the leaves of a balanced binary tree. What do we store in the internal nodes? We will store a list of some intervals.

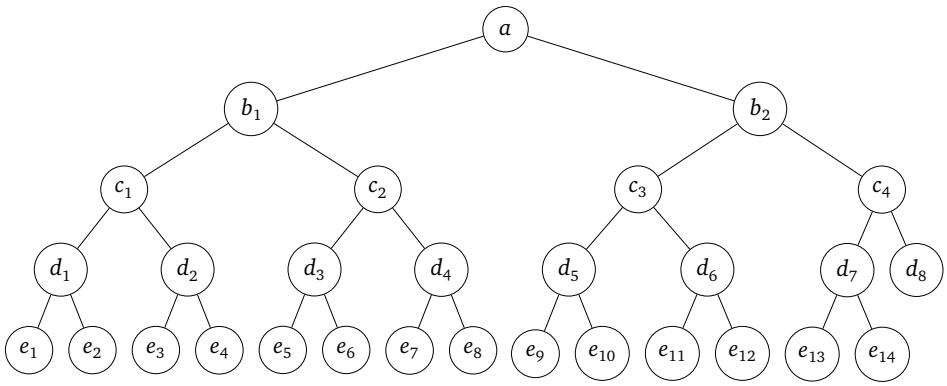


We define the *range* of a node v as follows:

- If v is a leaf, range(v) is the elementary interval stored at v.
- If v is an internal node, range(v) is the union of the ranges of its children.

For each interval $I \in \mathscr{I}$, we store I at v iff

- range(v) $\subseteq I$, and
- range($\pi(v)$) $\not\subseteq I$.



Lemma III.6. *An interval gets stored at most $O(\log n)$ times.*

Proof. We will show that each level of the tree stores an interval at most twice. Suppose $v_1 < v_2 < v_3$ store the same interval I at the same depth. WLOG suppose v_2 is a right child. $v_1 \leq$ sibling of v_2. Since I covers both v_1 and v_2, I covers the sibling of v_2. But then I covers $\pi(v_2)$, a contradiction. ■

III.2.1 Construction

Sort the elementary intervals in $O(n \log n)$ time. Build the skeleton of the tree in $O(n)$ time.

Pick any interval I. Start at the root.

- If I covers the range of the current node v, store I and stop.
- If $I \cap \text{range}(v_\ell) \neq \emptyset$, recurse into v_ℓ.
- If $I \cap \text{range}(v_r) \neq \emptyset$, recurse into v_r.

If v_1 and v_2 are nodes on the same level that intersect I partially, each node in between them will be covered by I. Thus only the children of these may be considered for recursion. This proves that only 4 nodes may be visited at each level. The construction time is therefore $O(n \log n)$.

Exercise III.7. *Consider any path Π from the root to a leaf. Prove that an interval cannot be stored at more than one node in Π.*

Solution. Fix a path $\Pi = (v_1, v_2, \ldots, v_m)$ and an interval I. Suppose I is stored at nodes v_i and v_j with $i < j$. Then I covers range(v_i). Thus I covers range(v_{i+1}). By induction, I covers range(v_{j-1}). Thus v_j fails the condition that range$(\pi(v_j)) \not\subseteq I$. ■

III.2.2 Querying

Given a $q \in \mathbb{R}$, search for q in the tree. Report all intervals stored along that path. This takes $O(\log n + k)$ time, by the exercise.

III.3 Rectangle stabbing

Lecture 12. Wednesday September 18

Question III.8. *Given a set $\mathscr{R}$ of n axis-aligned cells in $\mathbb{R}^d$, process queries of the form $q \in \mathbb{R}^d$, reporting all $R \in \mathscr{R}$ that contain q.*

We will write $\{R \in \mathscr{R} \mid q \in R\}$ as $q \odot \mathscr{R}$ Consider the 2D case first.

Question III.9. *Given a set $\mathscr{R}$ of n axis-aligned rectangles in $\mathbb{R}^2$, process queries of the form $q \in \mathbb{R}^2$, reporting all $R \in \mathscr{R}$ that contain q.*

Solution. ■

III.4 Vertical ray shooting

Question III.10. *Given a set $\mathscr{S}$ of n interior-disjoint (might have common endpoints) segments in $\mathbb{R}^2$, process queries of the form $q \in \mathbb{R}^2$, reporting the first segment hit by a vertical ray emanating upwards from q.*

Solution. Define $\mathscr{S}_x = \{\pi_x(S) \mid S \in \mathscr{S}\}$. Build a segment tree T on $\mathscr{S}_x$. For any node $v \in T$, let $\mathscr{S}(v)$ denote the segments stored at v.

For any query $q = (x, y)$, query T with x and let Π be the search path. We can solve the problem at each node $v \in \Pi$ in $O(\log n)$ time by binary searching through $\mathscr{S}(v)$. Since they span the range of v, the segments in $\mathscr{S}(v)$ are totally ordered by their y-coordinates.

Doing this for each node in Π and picking the lowest of these gives an $O(\log^2 n)$ query time. ■

Lecture 13. Wednesday September 25

III.5 Planar point location

We are given a rectangle $R = [x_1, x_2] \times [y_1, y_2]$ and a set of n non-intersecting (except at the endpoints) line segments in R such that they partition R into $O(n)$ cells. We are given a query point $q \in R$ and need to report the cell containing q.

If the line segments were all horizontal, we could perform binary search on the y-coordinate of q to find the cell containing q. Less specifically, even if the line segments are not horizontal but they span all of $[x_1, x_2]$, we can still perform binary search, since the line segments are still totally ordered by their y-coordinates.

For the general setting, we can partition R into $O(n)$ slabs at each of the vertices. We perform binary search on the x-coordinate to locate the slab, and binary search on the y-coordinate to locate the cell within the slab.

However, storing sorted arrays for each slab requires $O(n^2)$ space.

III.6 Persistent balancing search trees

(Sarnak and Tarjan, 1986) We wish to make modifications to a balanced search tree while maintaining its history. A naive approach would be to store a copy of the tree after each modification (for planar point location, this would correspond to storing the sorted arrays for each slab). For more complicated techniques, we'll only consider insertion.

One better technique is *path copying*.

- When inserting a node, create a copy of the path from the root to the new node.

This requires $O(\log n)$ additional space per insertion, and so takes $O(n+t \log n)$ space for t insertions.

Another technique is using *fat nodes*.

- For each node, store its children as vectors instead of pointers.
- When inserting a node v at time t, append (v, t) to the appropriate vector of $\pi(v)$.

This only requires $O(1)$ additional space per insertion, so only $O(n + t)$ space overall. However, searching is broken. In the case that each node along the path to a node v has been modified $t/\log n$ times, the search time is $O(\log n \log t)$.

We can combine the two techniques. We will use a slightly fat node to store modifications efficiently, but we will resort to copying the path when the node becomes too fat so that the search time remains $O(\log n)$.

III.6.1 Aggregate analysis for dynamic arrays

Suppose we start with an empty array of size 1 and double its size whenever there isn't space for an insertion. The first append operation takes $O(1)$ time. In general, the operation A_i takes $O(1)$ time when i is not a power of 2, and A_{2^k} takes $O(2^k)$ time. Then the total time for n appends is

$$\sum_{i=0}^{n} 1 + \sum_{k=0}^{\log n} 2^k = O(n+n) = O(n).$$

Thus the amortized append time is $O(n)/n = O(1)$.

III.6.2 Amortized space analysis

Define the potential function Φ to be the number of live nodes that have been modified, "live" meaning they are reachable from the current root.

Chapter IV

Convex hulls

Lecture 15. Monday October 7

Definition IV.1 (convex hull)**.** Given a set P of points in $\mathbb{R}^2$, the *convex hull* of P is the smallest convex set containing all of P. We denote it by $\mathrm{CH}(P)$.

Theorem IV.2. *The convex hull of a finite set is a convex polygon.*

We will represent a convex hull as a counter-clockwise ordering of vertices, starting from the bottom-most vertex, and ignoring any degeneracies.

Assumptions:

- No three points are collinear.
- All x- and y-coordinates are distinct.

Question IV.3. *Given a set P of n points in $\mathbb{R}^2$, compute* $\mathrm{CH}(P)$.

Naivest. An edge of the convex hull can be characterized by the fact that all other points lie on one side of it. Check all line segments between pairs

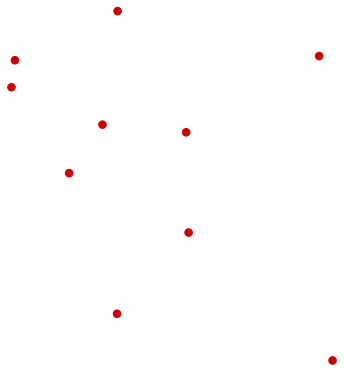


Figure IV.1: Convex hull of a set of points

of points. This takes $O(n^3)$ time. Post-processing to order them in counter-clockwise order takes $O(n)$ time. ■

For ease of use, we define the primitive operation $CCW(p, q, r)$, which returns $\top$ if $p \to q \to r$ is a counter-clockwise turn, and $\bot$ otherwise.

Naive – Jarvis' march. The bottom-most point p_0 is on the convex hull. The next vertex can be characterized as the point which makes the smallest angle with the horizontal drawn through p_0.

Once p_{i-1} and p_i have been computed, the next vertex p_{i+1} is the vertex p_j such that $\angle p_{i-1}p_ip_j$ is maximized. Continue this until $p_i = p_0$.

This takes $O(n^2)$ time. In terms of the size of the hull, the time taken is $O(nh)$.

```
HULL-ANGLE(P):
  p_0 ← bottom-most point in P
  p ← NEXT-POINT(p_0)
  report p_0, p
  while p ≠ p_0
      p ← NEXT-POINT(P \ {p}, p)
      report p
```

■

Exercise IV.4. *Write the* NEXT-POINT *function using only the* CCW *primitive (do not compute angles/inner products).*

Solution.

```
NEXT-POINT(P', p):
  q ← any point from P'
  for q' ∈ P'
      if not CCW(p, q, q')
          q ← q'
  return q
```

■

Graham's scan. Pick an arbitrary point p_0. Sort the points by angle with p_0. Traverse the sorted list, adding points to the hull if they make a counter-clockwise turn with the last two points on the hull. Otherwise, remove points from the hull until the turn is counter-clockwise.

The sorting step takes $O(n \log n)$ time. The scan step takes $O(n)$ time. ■

Lecture 16. Wednesday October 9

Graham's variant. Sort the points by x-coordinate. Perform a Graham's scan from left to right. This gives the lower hull. Similarly compute the upper hull (either use $\neg CCW()$ in place of $CCW()$, or scan from right to left). Concatenate the two hulls to get the convex hull.

Again, the time is $O(n \log n)$ for sorting and $O(n)$ for scanning. ■

Proof of correctness – sketch. Sorting the points as $p_1, p_2, \ldots, p_n$, we start with the initial curve joining each p_i to p_{i+1}. The invariant is that each point lies on or above this curve. ■

Divide and conquer. Partition P into P_ℓ and P_r by x-coordinate. Compute the convex hull of each half recursively. Merge them by computing the upper and lower common tengents. Exercise ■

Incremental. Sort the points by x-coordinate. Compute the convex hull for $p_1, \ldots, p_i$. When encountering the point p_{i+1}, compute the upper and lower tangents in $O(\log n)$ time (exercise) and remove vertices in between. ■

IV.1 Lower time bound

Theorem IV.5. *The general convex hull problem cannot be solved in time* $o(n \log n)$.

See *algebraic decision tree*.

Proof. Given a set $X \subseteq \mathbb{R}$ of n points, map it under the squaring map to place them on a convex curve. Compute the hull of these points. The points must be in (a cyclic permutation of) their sorted x values. We can recover the sorted sequence in $O(n)$ time. Since sorting requires $O(n \log n)$ time, computing the convex hull cannot be done in less. ■

IV.2 Chan's $O(n \log h)$ algorithm

Guess $\hat{h} = 2^{2^k}$ Divide P into $\frac{n}{\hat{h}}$ groups $S_1, S_2, \ldots, S_{n/\hat{h}}$ with $\hat{h}$ points each, completely arbitrarily. Compute $\mathrm{CH}(S_i)$ for each i using an $O(n \log n)$ algorithm. This takes $\frac{n}{\hat{h}} \times \hat{h} \log \hat{h} = n \log \hat{h}$ time.

Start with the bottom-most point. Compute the next point using the incremental algorithm idea:

- the next point from a point in $\mathrm{CH}(S_i)$ is either the next point in $\mathrm{CH}(S_i)$,
- or from some $\mathrm{CH}(S_j)$. We can compute the only candidate from each S_j in $\log \hat{h}$ time per S_j, for total time $\frac{n}{\hat{h}} \log \hat{h}$.

We determine the best of these candidates in $O(\hat{h})$ time.

Doing this for $\hat{h}$ points gives $\hat{h} \cdot \frac{n}{\hat{h}} \log \hat{h} = n \log \hat{h}$ time. If we return to the starting point, we report the hull. Otherwise, continue with the guess $\hat{h} = 2^{2^{k+1}}$.

The total time taken is

$$\sum_{k=1}^{\log \log h} O(n 2^k) = O(n \log h).$$

Theorem IV.6. *$O(n \log h)$ is the lower bound.*

Proof. Reduction to multiset problem (read up). ■

Chapter V

Delaunay triangulation

$$
\begin{aligned}
n - e + f &= 1 \\
e &= \frac{3f}{2} + \frac{h}{2} \\
\implies n - \frac{f}{2} - \frac{h}{2} &= 1 \\
\implies f &= 2n - 2 - h \\
\implies e &= 3n - 3 - h
\end{aligned}
$$

Definition V.1 (Delaunay triangulation)**.** Given a finite point set $P \subseteq \mathbb{R}^2$, a Delaunay triangulation of P is such a triangulation such that the circumcircle of any triangle contains no point from P in its interior.

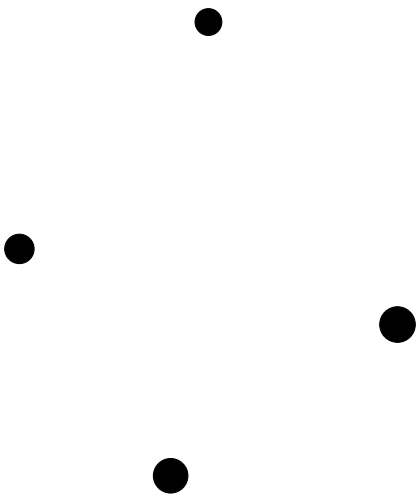
We will assume that

- no three points are collinear;
- no four points are cocircular.

Question V.2. *Given a set P on n points in $\mathbb{R}^2$, compute its Delaunay triangulation.*

Solution. Start with any triangulation. Pick any triangle abc, and check if any point lies in the interior. If no, be happy. If yes, call it d. Split the quadrilateral $abcd$ into 2 triangles via the FLIP operation—WLOG let ac be the diagonal of $abcd$. Flip this to be bd. $abcd$ is now partitioned into two triangles abd and cbd. The circumcircles of either of these don't contain the fourth point. ■

Lecture 17.
Monday
October 14



Definition V.3. Let T be a triangulation of a set of points P. We say that an edge $e = t_1 \cap t_2$ is *locally Delaunay* if $C(t_1)^\circ \cap t_2 \cap P = \varnothing$. We say that T is *locally Delaunay* if every edge in T is locally Delaunay.

```
DELAUNAY(P):
    T ← any triangulation of P
    while some edge e is not locally Delaunay
        flip e
```

Proposition V.4 (edge characterization)**.** *A triangulation T is a Delaunay triangulation iff for every $p, q \in P$, there is some circle C passing through p and q such that $C^\circ \cap P = \varnothing$.*

Definition V.5 (angle vector)**.** The angle vector of a triangulation T is the vector of all angles of each triangle in T, listed in increasing order. We denote this by $\alpha(T)$.

Proposition V.6. *Let $T_1 \xrightarrow{\text{FLIP}} T_2$. Then $\alpha(T_1) < \alpha(T_2)$ under the lexicographic order.*

Corollary V.7. *The algorithm* DELAUNAY *terminates.*

Proof. There are finitely many triangulations and the angle vector is strictly increasing. ∎

Fact V.8. *An edge pq once flipped by the* DELAUNAY *algorithm will never be flipped again.*

This is not easy to prove.

Theorem V.9. *A locally Delaunay triangulation is a Delaunay triangulation.*

Proof. Let T be a locally Delaunay triangulation that is *not* Delaunay. That is, there is some $t \in T$ such that $C(t)^\circ \cap P$ contains some point p.

Iteratively construct triangles $t_1, t_2, \ldots$ approaching p such that $p \in C(t_i)^\circ$ for all i. Each iteration, visits one new point, so eventually this is p. This violates local Delaunity. ■

Read proof from David Mount.

Corollary V.10. *The* DELAUNAY *algorithm computes the Delaunay triangulation.*

Corollary V.11. *The Delaunay triangulation is the unique triangulation which maximizes the angle vector.*

Proof. Start the DELAUNAY algorithm with any tiangulation. Then by proposition V.6, the Delaunay triangulation has a larger angle vector. ■

V.0.1 Incremental algorithm

Maintain a Delaunay triangulation. For any new point p_i inside the convex hull of the current set of points, Let abc be the triangle containing p_i. Draw new edges p_ia, p_ib, p_ic. The local Delaunay property can only be false for the edges ab, bc and ca.

```
DELAUNAY-INCREMENTAL(P):
  shuffle P
  T ← ∅
  for i ← 1 to n
      find △^abc ∈ T containing p_i
      add the edges p_ia, p_ib, p_ic
      for e ∈ {ab, bc, ca}
          LEGALIZE-EDGE(p_i, ab)
  return T
```

```
LEGALIZE-EDGE(p, xy):
  z ← vertex opposite to p
  if xy isn't locally Delaunay
      flip xy to pz
      LEGALIZE-EDGE(p, xz)
      LEGALIZE-EDGE(p, yz)
```

Each flip in LEGALIZE-EDGE connects a new point to p. Thus LEGALIZE-EDGE runs in $O(n)$ time. The total running time is thus $O(n^2)$.

Fact V.12. *The expected degree of any vertex in the Delaunay triangulation is constant.*

Thus the expected running time of the incremental algorithm is $O(n \log n)$, where the $\log n$ is required for searching $\triangle^{abc}$.

To ensure that points are only added inside the convex hull, add far away dummy points before the incremental algorithm. Show that the edges from the dummy points in the Delaunay triangulation of this only connect to the convex hull.

Lecture 19.
Wednesday
October 23

V.1 Point-line duality

Let $\mathscr{L}$ be the set of lines in $\mathbb{R}^2$ that are not parallel to the vertical axis.

Then $\varphi : (a, b) \mapsto \{y = ax - b\}$ is a bijection from $\mathbb{R}^2$ to $\mathscr{L}$. Let φ(any point) be its dual line, and φ^{-1}(any line) be its dual point.

For any two non-parallel lines $\ell_1, \ell_2 \in \mathscr{L}$, write $\ell_1 \wedge \ell_2$ for their intersection point. For any two points $p_1, p_2 \in \mathbb{R}^2$, write $p_1 \vee p_2$ for the line passing through p_1 and p_2.

Proposition V.13. *Let (p_1, ℓ_1), (p_2, ℓ_2), ... be primal-dual pairs.*

(1) (incidence preserving) $p_1 \in \ell_2 \implies p_2 \in \ell_1$.

(2) (order reversal) p_1 lies above ℓ_2 iff p_2 lies above ℓ_1.

(3) If ℓ_1 and ℓ_2 are not parallel, then $\ell_1 \wedge \ell_2$ is the dual of $p_1 \vee p_2$.

(4) If p_1, p_2, p_3 lie on ℓ_4, then ℓ_1, ℓ_2, ℓ_3 intersect at p_4.

Proof.

(1) Let $p_1 = (a_1, b_1)$ and $p_2 = (a_2, b_2)$. Then $p_1 \in \ell_2$ iff $b_1 = a_2 a_1 - b_2$. This is symmetric in the indices.

(2) Let $p_i = (a_i, b_i)$. Then p_1 lies above ℓ_2 iff $b_1 > a_2 a_1 - b_2$. This is symmetric in the indices.

(3) Let $p_1 = (a_1, b_1)$ and $p_2 = (a_2, b_2)$. Then $\ell_1 : y = a_1 x - b_1$ and $\ell_2 : y = a_2 x - b_2$. These intersect at $\left(\frac{b_1 - b_2}{a_1 - a_2}, a_1 \frac{b_1 - b_2}{a_1 - a_2} - b_1\right)$. The line joining p_1 and p_2 has slope $\frac{b_2 - b_1}{a_2 - a_1}$, and y-intercept $b_1 - a_1 \frac{b_1 - b_2}{a_1 - a_2}$.

Better: directly from incidence preservation. Since $\ell_1 \wedge \ell_2$ lies on both ℓ_1 and ℓ_2, its dual point passes through both p_1 and p_2.

(4) Direct from incidence preservation.

■

Definition V.14 (envelope)**.** Let $L = \{\ell_1, \ldots, \ell_n\} \subseteq \mathscr{L}$, and let $H^- = \{h_1^-, \ldots, h_n^-\} \subseteq 2^{\mathbb{R}^2}$ be the set of lower half-spaces bounded by the lines in L. Then the *lower envelope* of L is the set $\bigcap H^-$. The upper envelope is the intersection of all upper half-spaces.

Lemma V.15. *Given a finite set of points in $\mathbb{R}^2$, the lower envelope of the duals is given by the duals of the upper convex hull.*

Proof. Relabel the points so that $p_0, p_2, \ldots, p_m$ are the vertices on the upper hull, listed right to left. Let ℓ_i be the line joining p_{i-1} and p_i, for $i \in [m]$. Since every point lies below ℓ_i, every dual line passes above the dual of ℓ_i, say ℓ_i^*. Thus ℓ_i^* is a vertex in the lower envelope. Since the lower envelope is convex, the edges between these are also part of the lower envelope. This gives the complete envelope.

The lines $\varphi(p_1)$ and $\varphi(p_m)$ are left as edge cases. ■