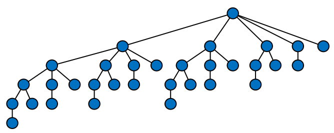
Lecture 14. Tail Bounds, Randomized Incremental Construction (Also Some Computational Geometry)

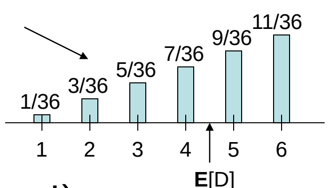
CpSc 8400: Algorithms and Data Structures
Brian C. Dean



School of Computing
Clemson University
Spring, 2016

Expected Value

- The expected value of a discrete random variable X, denoted **E**[X], is defined as
  $$\mathbf{E}[X] = \Sigma_{\text{values } v}\, v\, \mathbf{Pr}[X = v].$$
- Think of **E**[X] informally as the “center of mass” of X’s probability distribution.
- Example: Let D be the max of two dice rolls. Recall that D has this probability distribution.
  – Thus, **E**[D] = 1(1/36) + 2(3/36) + 3(5/36) + 4(7/36) + 5(9/36) + 6(11/36) = 161/36 = $4\ {}^{17}/_{36}$



- Careful: Don’t write **E**[A] if A is an event (another syntax error!)

Computing Expected Values

There are generally 4 different ways we will compute expected values in this class:

1. Directly using the definition **E**[X] = Σ_v v **Pr**[X = v].
2. The special case of an **indicator** random variable.
3. The special case of a **geometric** random variable.
4. Expressing a complicated random variable in terms of a sum of simpler r.v.'s and applying **linearity of expectation**.

3

Randomized Quicksort Revisited

- Let's think again about randomized quicksort.
- We've already shown an O(n log n) running time both w.h.p and also in expectation.
- There is another alternate expected running time proof that corresponds exactly to the w.h.p. proof:
- **W.h.p. proof:**

  Randomized reduction lemma → O(log n) / element w.h.p.

  Union bound → O(n log n) time for all elements w.h.p.
- **Expected running time proof:**

  Linearity of expectation → O(log n) expected time / element

  Linearity of expectation → O(n log n) total expected time.

4

Wald's Theorem

- Linearity of expectation tells us that for N identically distributed random variables $X_1 \ldots X_N$,

  $$E[X_1 + \ldots + X_N] = N\,E[X_1].$$

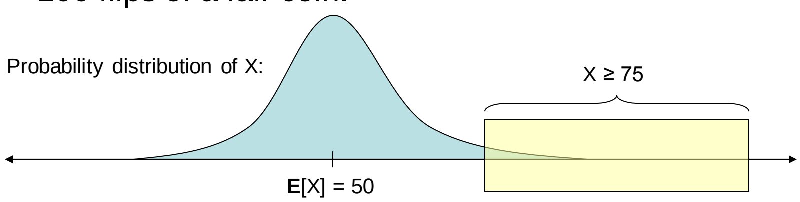
- What if N itself is a random variable though? (i.e., we have a random number of trials, each of which have identically-distributed random running times)
- **Wald's theorem** says that, as we might expect:

  $$E[X_1 + \ldots X_N] = E[N]\,E[X_1].$$

  (this requires a few technical conditions to hold; for example, N must be independent of the X_i's)

Tail Bounds

- We are often interested in the probability that a random variable X deviates significantly from its mean, **E**[X], when we randomly sample X from its distribution.
- Examples:
  - Show that **Pr**[$T = O(\log n)$] $\geq 1 - 1/n^c$ for arbitrary $c > 0$.
  - What is **Pr**[$X \geq 75$] if X denotes the number of heads in 100 flips of a fair coin.



Markov's Inequality

- X is any *nonnegative* random variable, then

  $\mathbf{Pr}[X \geq k\mathbf{E}[X]] \leq 1/k.$
- Sometimes written as $\mathbf{Pr}[X \geq a] \leq \mathbf{E}[X] / a$.
- Example: let X be the number of heads we see when flipping 100 coins.
  - $E[X] = 50$.
  - $Pr[X \geq 75] = Pr[X \geq (3/2)E[X]] \leq 2/3$.
- Due to its generality, Markov's inequality is a rather weak bound, although it's still quite useful.
- If expected running time = T, then probability our algorithm takes ≥ kT time is at most 1/k.

7

Chernoff Bounds

- Suppose $X = X_1 + X_2 + \ldots + X_n$ is a sum of independent indicator (0/1) random variables (a common scenario).
- Then the tails of X's distribution drop off very quickly:
  - $\mathbf{Pr}[X \geq \mathbf{E}[X] + t] \leq e^{-2t^2/n}$
  - $\mathbf{Pr}[X \leq \mathbf{E}[X] - t] \leq e^{-2t^2/n}$
  - $\mathbf{Pr}[X \leq (1 - \varepsilon)\mathbf{E}[X]] \leq e^{-\varepsilon^2 E[x]/2}$
  - $\mathbf{Pr}[X \geq (1 + \varepsilon)\mathbf{E}[X]] \leq e^{-\varepsilon^2 E[x]/4}$ (if $\varepsilon \leq 2e - 1$)
  - $\mathbf{Pr}[X \geq (1 + \varepsilon)\mathbf{E}[X]] \leq 2^{-(\varepsilon+1)E[x]}$ (if $\varepsilon > 2e - 1$)
- Example: If X denotes the number of heads we see when flipping 100 fair coins, then
  - $\mathbf{Pr}[X \geq 75] = \mathbf{Pr}[X \geq \mathbf{E}[X] + 25] \leq e^{-25^2/50} = e^{-12.5} \leq 0.000004$
- Much stronger than Markov's inequality, but only applies when X is a sum of independent indicator random variables.

8

Proving the Randomized Reduction Lemma

- **Theorem:** Suppose every iteration of our algorithm has probability $\geq p$ of reducing our problem size to at most $\leq q$ times its original size. Then given any constant $c > 0$, we can find another constant k such that **Pr**$[T \geq k \log n] \leq 1 / n^c$, where T is the total number of iterations of our algorithm.
- **Proof:**
  - Suppose we run our algorithm for $k \log n$ iterations.
  - Let X denote the # of "good" iterations among these (a good iteration is one where we reduce the problem size to $\leq q$ times original).
  - Now note that **Pr**$[T \geq k \log n] \leq$ **Pr**$[X \leq \log_{(1/q)} n]$, since the event that "$T \geq k \log n$" is a subset of the event that "$X \leq \log_{(1/q)} n$".
  - So now let's find a constant k such that **Pr**$[X \leq \log_{(1/q)} n] \leq 1/n^c$.
  - Continued…

9

Proving the Randomized Reduction Lemma

- **Proof Continued:**
  - We run our algorithm for only $L = k \log_2 n$ iterations.
  - X = # of good iterations among these.
  - We want to show that **Pr**$[X \leq \log_{(1/q)} n] \leq 1/n^c$.
  - Write $X = X_1 + X_2 + \ldots + X_L$, where X_j is an indicator r.v. whose value is 1 if the jth iteration is good. Note that **E**$[X_j] \geq p$.
  - **E**$[X]$ = **E**$[X_1] + \ldots +$ **E**$[X_L] \geq Lp = kp \log_2 n$.
  - So **Pr**$[X \leq \log_{(1/q)} n]$

    $\leq$ **Pr**$[X \leq (1 / kp \log_2(1/q))$ **E**[**X**]]

    $=$ **Pr**$[X \leq (r / k)$ **E**[**X**]], where $r = 1 / [p \log_2 (1/q)]$ is a constant.
  - Now we use the Chernoff bound with $(1 - \varepsilon) = r / k$:
    Pr$[X \leq (1 - \varepsilon)$**E**$[X]] \leq e^{-\varepsilon^2 E[x]/2} = e^{-(1-r/k)^2 E[x]/2} \leq e^{-E[x]/8}$, if we choose k sufficiently large so that $1 - r / k \geq ½$.
  - Thus, **Pr**$[X \leq \log_{(1/q)} n] \leq e^{-E[x]/8} \leq e^{-(kp/8)\log n} = e^{-(kp/8 \ln 2)\ln n} = 1 / n^{kp/8\ln 2}$.
  - So **Pr**$[X \leq \log_{(1/q)} n] \leq 1/n^c$ if we choose $k \geq (8 \ln 2)/p$.

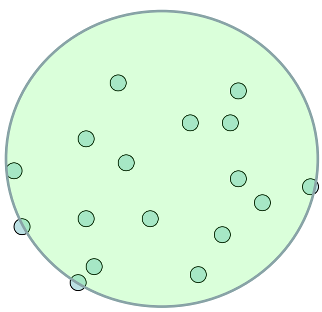
10

Randomized Incremental Construction

- When building a BST or a binary heap, inserting elements in certain orderings may lead to inefficient running times.
- However, inserting in a random order is (with high probability) quite efficient.
- These are examples of a general algorithm design technique called **randomized incremental construction**, where we build a solution by incorporating one element at a time, processing elements in random order.
- R.I.C. algorithms are especially common in computational geometry:
  - Examples: Convex hulls (2D and 3D), half-space intersections, Voronoi diagrams, Delaunay triangulations, smallest enclosing circle, low-dimensional linear programming, binary space partition trees, trapezoidal decompositions, closest pair, etc.

Example: Smallest Enclosing Circle

- Given n points in the plane, find the smallest circle enclosing them all.



Smallest Enclosing Circle

- Given n points in the plane, find the smallest circle enclosing them all.
- The optimal circle will be determined by at most 3 points, leading to an $O(n^4)$ "brute force" solution.
- With randomized incremental construction, however, we can solve this problem in only O(n) expected time!

13

3 Simple Steps…

- Suppose by magic that we already know 2 points on the boundary of an optimal circle…
  - Now it's easy to compute the answer in O(n) time:

    Let p_1 and p_2 be the points we know.

    Start with a circle C having p_1 and p_2 as endpoints of its diameter.

    Let the remaining points $p_3 \ldots p_n$ be arbitrarily ordered.

    Process $p_3 \ldots p_n$ in sequence, enlarging C when necessary so it remains an optimal circle for the set of points considered thus far.

14

3 Simple Steps…

- Suppose by magic that we already know 2 points on the boundary of an optimal circle…
  - Now it's easy to compute the answer in O(n) time.
- Now suppose (also by magic) that we already know only 1 point on the boundary of an optimal circle…
  - We can still compute the optimal circle in O(n) expected time…
  - Let p_1 be the point we know, and let $p_2 \ldots p_n$ be randomly ordered. Start with a zero-area circle C centered at p_1.
  - Process $p_2 \ldots p_n$ in sequence, updating C as we go.

15

3 Simple Steps…

- Suppose by magic that we already know 2 points on the boundary of an optimal circle…
  - Now it's easy to compute the answer in O(n) time.
- Now suppose (also by magic) that we already know only 1 point on the boundary of an optimal circle…
  - We can still compute the optimal circle in O(n) expected time…
- Finally, suppose we know none of the points on the boundary of an optimal circle…

16

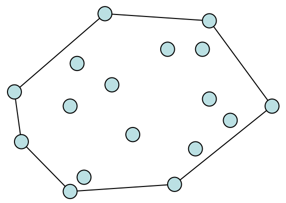
3 Simple Steps…

- Suppose by magic that we already know 2 points on the boundary of an optimal circle…
  - Now it's easy to compute the answer in O(n) time.
- Now suppose (also by magic) that we already know only 1 point on the boundary of an optimal circle…
  - We can still compute the optimal circle in O(n) expected time…
- Finally, suppose we know none of the points on the boundary of an optimal circle…
- Final running time: O(d! n), where d is the dimensionality of our space.

17

The Convex Hull Problem

- The **convex hull** of a set of n points is the smallest convex polygon containing all n points:



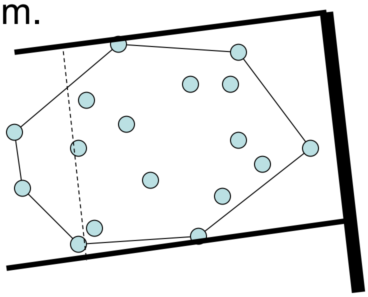
- The convex hull problem:
  - **Input:** A list of n points $(x_1, y_1) \ldots (x_n, y_n)$.
  - **Output:** An array or linked list specifying the points around the boundary of the hull in clockwise (or counterclockwise) order.

18

Farthest Pair of Points (2D)

- **Claim:** The farthest pair of points in a point set lies on the convex hull.
  - The two points will be on “opposite sides” of the hull.
- Once we’ve found the convex hull, we can therefore find the farthest pair of points in O(n) time using the “rotating calipers” algorithm.

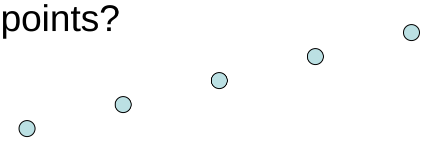
  (we could call this a “sort and scan” approach…)



19

Collinear Points and Other Special Cases…

- Do all of these points belong to the convex hull, or just the endpoints?

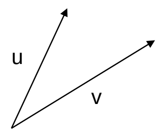


- In general, computational geometry problems tend to be plagued by special cases like this.
  - This plus the danger of round-off errors can make it somewhat tricky to implement computational geometry algorithms correctly in practice!
- Typical assumption: no 3 of our points are collinear.

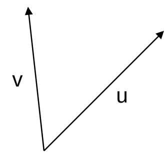
20

Primitive Operations

- Consider the following simple geometric questions:
  - Point P on line L?
  - Points P and Q on same side of line L?
  - Line segments S_1 and S_2 intersect?
  - Point P in angle formed by two rays R_1 and R_2?
  - Point P in convex polygon Q?
- A single “trick” makes all of these easy: look at the sign of the z component of the cross product!



$(u \times v)_z < 0$



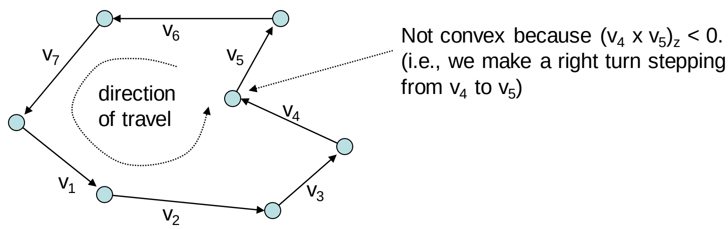
$(u \times v)_z > 0$

$$(u \times v)_z = \begin{vmatrix} u_x & u_y \\ v_x & v_y \end{vmatrix}$$

21

Checking Convexity

- Using the cross product test on consecutive pairs of vectors v, we can test a polygon for convexity:

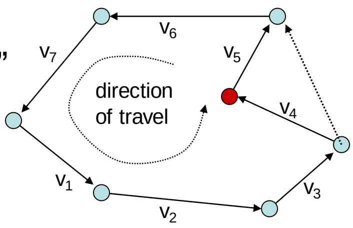


- As we walk around the polygon in a counter-clockwise direction, we should make only “left turns” (z component of cross product positive).

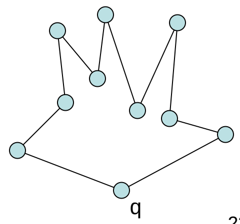
22

Convexifying a Polygon

- By splicing out points at which we turn right, we can "convexify" any polygon in O(n) time.

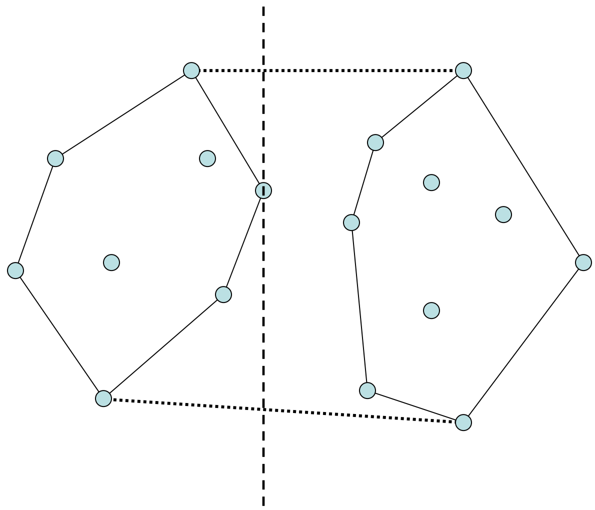


- This gives a simple O(n log n) algorithm (known as the "package wrapping" algorithm) for convex hulls:
  - Pick a "reference" point q known to be on the hull (e.g, with minimum y coordinate).
  - Sort remaining points by angle form q.
  - Use this sorted ordering to build a polygon and then convexify it.



23

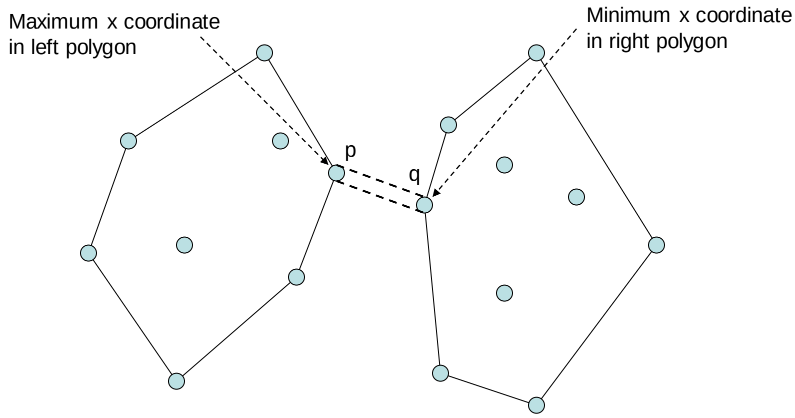
"QuickSort"



- How do we link two disjoint convex polygons in O(n) time?

24

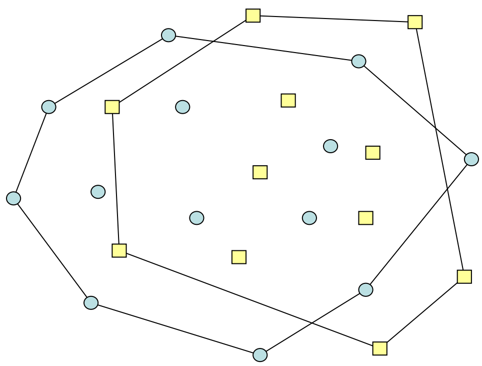
Linking Two Disjoint Polygons



- Splice polygons together into a non-convex polygon with a "doubled edge", then convexify!

25

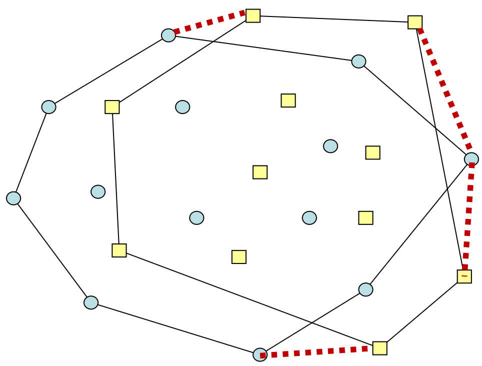
"Merge Sort"



- We can merge two (possibly overlapping) convex polygons in O(n) time by sweeping monotonically around each one in order of angle.
  - Just like merging two sorted sequences, only slightly more geometric details to deal with along the way (further details omitted…)

26

"Merge Sort"

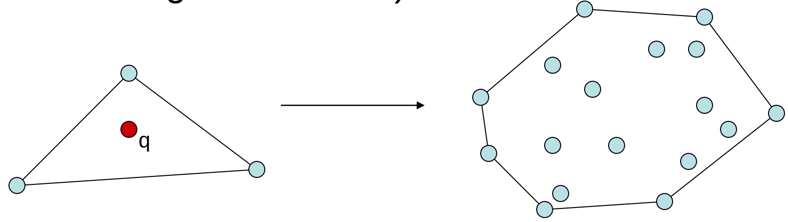


- We can merge two (possibly overlapping) convex polygons in O(n) time by sweeping monotonically around each one in order of angle.
  - Just like merging two sorted sequences, only slightly more geometric details to deal with along the way (further details omitted…)

27

A Randomized Incremental Construction Convex Hull Algorithm

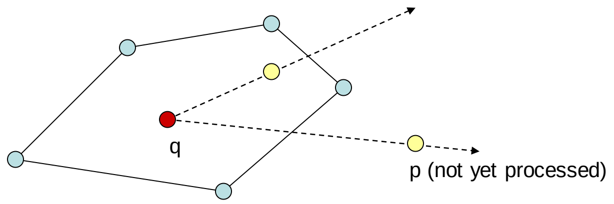
- We'll build up a convex hull by starting with 3 randomly-chosen points (a triangle) and adding the remaining points in random order, updating the hull when necessary.
- Maintain a point q inside the hull (the centroid of our initial triangle works fine).



28

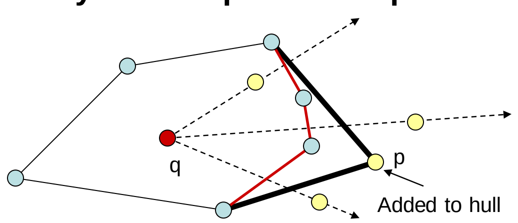
Randomized Incremental Construction of a Convex Hull

- Maintain an ordered linked list of all points (+edges) in clockwise order around the current hull.
- For each point p not yet processed, maintain bi-directional pointers between p and the edge in our hull crossed by the ray qp (with this extra information, we can quickly test if p is inside or outside the current hull).



Randomized Incremental Construction of a Convex Hull

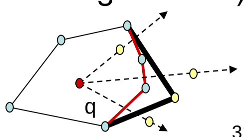
- When we add a new point p,
  - Check in O(1) time if p is inside or outside current hull.
  - If inside, nothing happens.
  - If outside, update the hull by removing necessary edges (shown in red below) and adding two new edges (the thick edges below); also **update the pointers of any affected yet-to-be-processed points**.



Running Time Analysis

- Add up running time per point p:
  - O(1) total work during initialization.
  - O(1) total work checking if p is inside or outside the current hull when p is finally selected.
  - O(1) total work inserting p into the hull (if at all).
  - O(1) total work later deleting p from the hull (if at all).
  - O(log n) expected total work for changing p's "current hull edge" pointer during the course of the algorithm

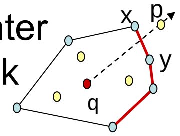
    (this is the only "expensive" part of the whole algorithm…)



31

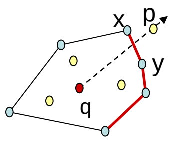
Probability of Pointer Change: Thinking Backwards Again…

- Consider the point in time at which we have added k points to our instance (some on the current hull, some inside the hull).
- p: not-yet-added point with pointer to hull edge xy.
- Only deletion of x or y causes p's pointer to change!
- We pick a point to delete at random from the set of all k outstanding points
- So w/ probability exactly 2/k, p's pointer changes when shrinking from a size-k point set to a size-(k-1) point set.



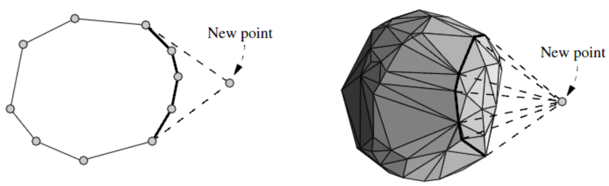
Linearity of Expectation

- With probability exactly 2/k, p's pointer changes when shrinking from a size-k point set to a size-(k-1) point set.
- X: Total number of changes to p's pointer
- X_k: indicator r.v. telling us whether p's pointer changes when adding the kth point.
- $E[X] = \sum_k E[X_k] = 2 \sum_k 1/k = O(\log n)$.
- And using randomized reduction, we can also argue that E[X] = O(log n).



Convex Hulls in Higher Dimensions

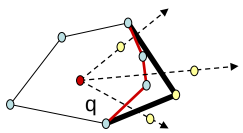
- Our randomized incremental construction algorithm extends readily to 3D, and keeps its O(n log n) expected running time!

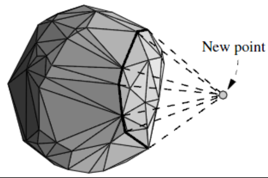


- In D>3 dimensions, the complexity of the hull can be as bad as $O(N^{\lfloor D/2 \rfloor})$. But we can compute convex hulls this quickly…

Running Time Analysis: 3D Convex Hull

- Add up running time per point p:
  - O(1) total work during initialization.
  - O(1) total work checking if p is inside or outside the current hull when p is finally selected.
  - ~~O(1) total work inserting p into the hull (if at all).~~
  - ~~O(1) total work later deleting p from the hull (if at all).~~
  - O(log n) expected total work for changing p's "current hull edge" pointer during the course of the algorithm

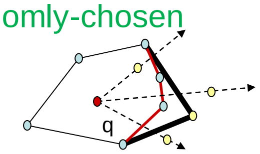


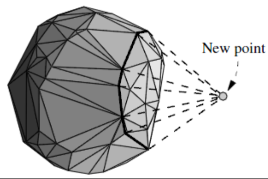


Running Time Analysis: 3D Convex Hull

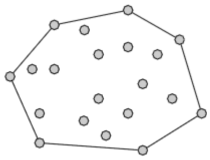
- Add up running time per point p:
  - O(1) total work during initialization.
  - O(1) total work checking if p is inside or outside the current hull when p is finally selected.
  - O(1) expected work inserting p into the hull (if at all).
  - O(1) expected work later deleting p from the hull (if at all).

(Euler's formula tells us that the average # of neighbors of a point in a polyhedron is O(1), so the expected # of neighbors of a randomly-chosen point is also O(1)).

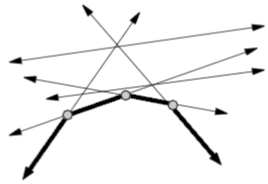




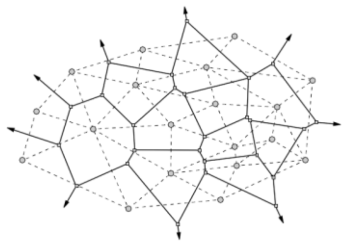
Some Prominent Problems in Computational Geometry…



Convex hull
Upper / lower hull



Lower / upper envelope



Voronoi diagram (solid)
Delaunay triangulation (dashed)

Geometric Duality

- In computational geometry, we often exploit geometric duality – exchanging the roles of points and lines.
  - Slope-intercept duality: point (a,b) ↔ line y = ax – b.
  - Polar duality: point (a,b) ↔ line ax + by = 1.
- The roles of points and lines reverse when we dualize:
  - Recall that two points determine a unique line, and likewise that two lines determine a unique point.
  - If L1 and L2 are two lines intersecting at point P, then dual(L1) and dual(L2) are two points that lie on the common line dual(P).
  - If P1 and P2 are two points determining line L, then dual(L) is the point of intersection between the lines dual(P1) and dual(P2).

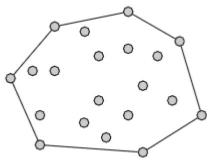
Duality – Example

- Given N points in the 2D plane, find the two that determine a line with maximum slope.

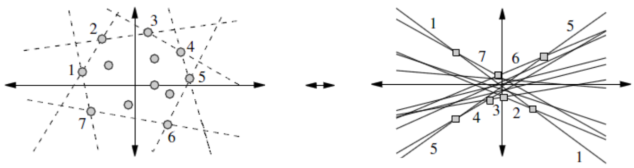
Duality – Example

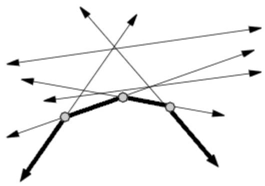
- Given N points in the 2D plane, find the two that determine a line with maximum slope.
- Apply slope-intercept duality: $(a,b) \leftrightarrow y = ax - b$
- Now we have a set of N lines, and we are looking for the two lines that determine a point with maximum coordinate!
- This point will occur between two lines with adjacent slopes (after sorting by slope); hence, in our original problem, the optimal line will occur between two points with adjacent x coordinates (after sorting by x coordinate).

Geometric Duality Between Upper/Lower Hulls and Lower/Upper Envelopes



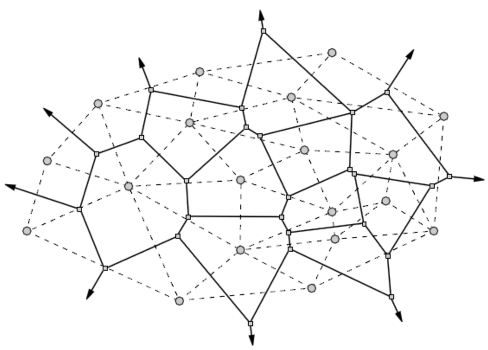
Convex hull
Upper / lower hull





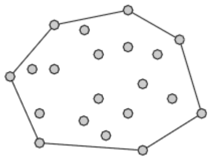
Lower / upper envelope

Planar Graph Duality Between Voronoi Diagrams and Delaunay Triangulations

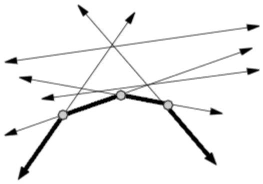


- There are many ways (most rather complicated!) to compute both objects in 2D in O(N log N) time (and there is a matching lower bound)
- In D>2 dimensions, the complexity of these objects can be as bad as $O(N^{\lfloor (D+1)/2 \rfloor})$.

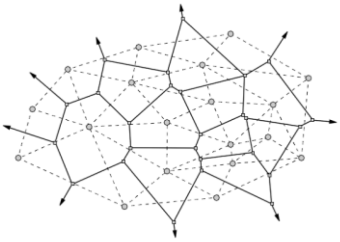
Back to Our High-Level List of Problems…



Convex hull
Upper / lower hull



Lower / upper envelope

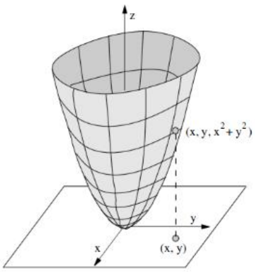


Voronoi diagram
Delaunay triangulation

We'll now discuss the *parabolic lifting* transformation, which allows us to transform between Voronoi/Delaunay problems in D dimensions and hull/envelope problems in D+1 dimensions!...

Parabolic Lifting

- Map each of our D-dimensional input points up to a (D+1)-dimensional point by lifting it onto a paraboloid ($z = x^2 + y^2$ in 2D):



- Then draw tangent planes to the paraboloid at each of the N lifted points…

Parabolic Lifting

- By projecting the upper envelope of our tangent planes back down onto D dimensions, we get the Voronoi diagram of our original point set!
- By projecting the lower hull of our lifted points back down to D dimensions, we get the Delaunay triangulation of the original point set!