CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2014

CS 5114: Theory of Algorithms

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, Virginia

Spring 2014

Copyright © 2014 by Clifford A. Shaffer

Title page

Students should be familiar with inductive proofs, recursion, data structures, and programming at the CS3114 level.

Geometric Algorithms

Potentially **large** set of objects to manipulate.

- Possibly millions of points, lines, squares, circles.
- Efficiency is crucial.

Computational Geometry

- Will concentrate on discrete algorithms – 2D

Practical considerations

- Special cases
- Numeric stability

Geometric Algorithms

Potentially large set of objects to manipulate.
- Possibly millions of points, lines, squares, circles.
- Efficiency is crucial.

Computational Geometry
- Will concentrate on discrete algorithms – 2D

Practical considerations
- Special cases
- Numeric stability

Same principles often apply to 3D, but it may be more complicated.
We will avoid continuous problems such as polygon intersection.

Special cases: Geometric programming is much like other programming in this sense. But there are a LOT of special cases! Co-point, co-linear, co-planar, horizontal, vertical, etc.

Numeric stability: Each intersection point in a cascade of intersections might require increasing precision to represent the computed intersection, even when the point coordinates start as integers. Floating point causes problems!

Definitions

- A **point** is represented by a pair of coordinates (x, y).
- A **line** is represented by distinct points p and q.
  - Manber's notation: $-p-q-$.
- A **line segment** is also represented by a pair of distinct points: the endpoints.
  - Notation: $p - q$.
- A **path** P is a sequence of points $p_1, p_2, \cdots, p_n$ and the line segments $p_1 - p_2, p_2 - p_3, \cdots, p_{n-1} - p_n$ connecting them.
- A **closed path** has $p_1 = p_n$. This is also called a **polygon**.
  - Points $\equiv$ vertices.
  - A polygon is a **sequence** of points, not a **set**.

Definitions

- A point is represented by a pair of coordinates (x, y).
- A line is represented by distinct points p and q.
  - Manber's notation: $-p-q-$.
- A line segment is also represented by a pair of distinct points: the endpoints.
  - Notation: $p - q$.
- A path P is a sequence of points $p_1, p_2, \cdots, p_n$ and the line segments $p_1 - p_2, p_2 - p_3, \cdots, p_{n-1} - p_n$ connecting them.
- A closed path has $p_1 = p_n$. This is also called a polygon.
  - Points ≡ vertices.
  - A polygon is a sequence of points, not a set.

Line alternate representation: slope and intercept.
For polygons, order matters. A left-handed and right-handed triangle are not the same even if they occupy the same space.

Definitions (cont)

- **Simple Polygon**: The corresponding path does not intersect itself.
  - A simple polygon encloses a region of the plane INSIDE the polygon.
- Basic operations, assumed to be computed in constant time:
  - Determine intersection point of two line segments.
  - Determine which side of a line that a point lies on.
  - Determine the distance between two points.

Definitions (cont)

- Simple Polygon: The corresponding path does not intersect itself.
  - A simple polygon encloses a region of the plane INSIDE the polygon.
- Basic operations, assumed to be computed in constant time:
  - Determine intersection point of two line segments.
  - Determine which side of a line that a point lies on.
  - Determine the distance between two points.

no notes

Point in Polygon

Problem: Given a simple polygon P and a point q, determine whether q is inside or outside P.

Basic approach:

- Cast a ray from q to outside P. Call this L.
- Count the number of intersections between L and the edges of P.
- If count is even, then q is outside. Else, q is inside.

Problems:

- How to find intersections?
- Accuracy of calculations.
- Special cases.

Point in Polygon
Problem: Given a simple polygon P and a point q, determine whether q is inside or outside P.
Basic approach:
Cast a ray from q to outside P. Call this L.
Count the number of intersections between L and the edges of P.
If count is even, then q is outside. Else, q is inside.
Problems:
How to find intersections?
Accuracy of calculations.
Special cases.

Special cases:

- Line intersects polygon at a vertex, goes in to out.
- Line intersects poly. at inflection point (stays in or stays out).
- Line intersects polygon through a line.

Simplify calculations by making line horizontal.

Accuracy of calculations is not a problem with integer coordinates for points and a horizontal line. But think about representing the intersection point for two arbitrary line segements (from a polygon intersection operation). Cascading intersections can lead to ever-increasing demand for precision in coordinate representation.

Point in Polygon Analysis (1)

Time complexity:

- Compare the ray to each edge.
- Each intersection takes constant time.
- Running time is $O(n)$.

Improving efficiency:

- $O(n)$ is best possible for problem as stated.
- Many lines are "obviously" not intersected.

Point in Polygon Analysis (1)
Time complexity:
Compare the ray to each edge.
Each intersection takes constant time.
Running time is O(n).
Improving efficiency:
O(n) is best possible for problem as stated.
Many lines are "obviously" not intersected.

no notes

Point in Polygon Analysis (2)

Two general principles for geometrical and graphical algorithms:

1. Operational (constant time) improvements:
   - Only do full calculation for 'good' candidates
   - Perform 'fast checks' to eliminate edges.
   - Ex: If $p_1.y > q.y$ and $p_2.y > q.y$ then don't bother to do full intersection calculation.
2. When doing many point-in-polygon operations, preprocessing may be worthwhile.
   - Ex: Sort edges by min and max y values. Only check for edges covering y value of point q.

Point in Polygon Analysis (2)
Two general principles for geometrical and graphical algorithms:
Operational (constant time) improvements:
Only do full calculation for 'good' candidates
Perform 'fast checks' to eliminate edges.
Ex: If p1.y > q.y and p2.y > q.y then don't bother to do full intersection calculation.
When doing many point-in-polygon operations, preprocessing may be worthwhile.
Ex: Sort edges by min and max y values. Only check for edges covering y value of point q.

Spatial data structures can help.

"Fast checks" take time. When they "win" (they rule something out), they save time. When they "lose" (they fail to rule something out) they add extra time. So they have to "win" often enough so that the time savings outweighs the cost of the check.

Constructing Simple Polygons

Problem: Given a set of points, connect them with a simple closed path.

Approaches:

1. Randomly select points.
2. Use a scan line:
   - Sort points by y value.
   - Connect in sorted order.
3. Sort points, but instead of by y value, sort by angle with respect to the vertical line passing through some point.
   - Simplifying assumption: The scan line hits one point at a time.
   - Do a rotating scan through points, connecting as you go.

Constructing Simple Polygons
Problem: Given a set of points, connect them with a simple closed path.
Approaches:
Randomly select points.
Use a scan line:
Sort points by y value.
Connect in sorted order.
Sort points, but instead of by y value, sort by angle with respect to the vertical line passing through some point.
Simplifying assumption: The scan line hits one point at a time.
Do a rotating scan through points, connecting as you go.

(1) Could easily yield an intersection.

(2) The problem is connecting point p_n back to p_1. This could yield an intersection.

Simplifying assumption is that the points are not colinear w.r.t. the scan line.

See Manber Figure 8.6.

Validation

Theorem: Connecting points in the order in which they are encountered by the rotating scan line creates a simple polygon.

Proof:

- Denote the points $p_1, \cdots, p_n$ by the order in which they are encountered by the scan line.
- For all i, $1 \leq i < n$, edge $p_i - p_{i+1}$ is in a distinct slice of the circle formed by a rotation of the scan line.
- Thus, edge $p_i - p_{i+1}$ does not intersect any other edge.
- Exception: If the angle between points p_i and p_{i+1} is greater than 180$\circ$.

Validation

Theorem: Connecting points in the order in which they are encountered by the rotating scan line creates a simple polygon.

Proof:
- Denote the points $p_1, \cdots, p_n$ by the order in which they are encountered by the scan line.
- For all i, $1 \leq i < n$, edge $p_i - p_{i+1}$ is in a distinct slice of the circle formed by a rotation of the scan line.
- Thus, edge $p_i - p_{i+1}$ does not intersect any other edge.
- Exception: If the angle between points p_i and p_{i+1} is greater than 180$\circ$.

So, the key is to pick a point for the center of the rotating scan that guarentees that the angle never reachese 180$\circ$.

Implementation

How do we find the point for the scanline center?

Actually, we don't care about angle – slope will do.

Select z;
for ($i = 2$ to n)
  compute the slope of line $z - p_i$.
Sort points p_i by slope;
label points in sorted order;

Time complexity: Dominated by sort.

Implementation

How do we find the point for the scanline center?

Actually, we don't care about angle – slope will do.

Select z;
for ($i = 2$ to n)
  compute the slope of line $z - p_i$.
Sort points p_i by slope;
label points in sorted order;

Time complexity: Dominated by sort.

Pick as z the point with greatest x value (and least y value if there is a tie). See Manber Figure 8.7.

The next point is the next largest angle between $z - p_i$ and the vertical line through z. It is important to use the slope, because then our computation is a constant-time operation with no transendental functions.

z is the point with greatest x value (minimum y in case of tie)

So, time is $\Theta(n \log n)$

Convex Hull

- A **convex hull** is a polygon such that any line segment connecting two points inside the polygon is itself entirely inside the polygon.
- A **convex path** is a path of points $p_1, p_2, \cdots, p_n$ such that connecting p_1 and p_n results in a convex polygon.
- The convex hull for a set of points is the smallest convex polygon enclosing all the points.
  - imagine placing a tight rubberband around the points.
- The point **belongs** to the hull if it is a vertex of the hull.
- **Problem**: Compute the convex hull of n points.

Convex Hull

- A convex hull is a polygon such that any line segment connecting two points inside the polygon is itself entirely inside the polygon.
- A convex path is a path of points $p_1, p_2, \cdots, p_n$ such that connecting p_1 and p_n results in a convex polygon.
- The convex hull for a set of points is the smallest convex polygon enclosing all the points.
  - imagine placing a tight rubberband around the points.
- The point belongs to the hull if it is a vertex of the hull.
- **Problem**: Compute the convex hull of n points.

no notes

Simple Convex Hull Algorithm

IH: Assume that we can compute the convex hull for $< n$ points, and try to add the nth point.

1. nth point is inside the hull.
   - No change.
2. nth point is outside the convex hull
   - "Stretch" hull to include the point (dropping other points).

Simple Convex Hull Algorithm

IH: Assume that we can compute the convex hull for $< n$ points, and try to add the nth point.

1. nth point is inside the hull.
   - No change.
2. nth point is outside the convex hull
   - "Stretch" hull to include the point (dropping other points).

See Manber Figure 8.9.

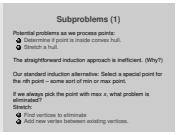
Subproblems (1)

Potential problems as we process points:
1. Determine if point is inside convex hull.
2. Stretch a hull.

The straightforward induction approach is inefficient. (Why?)

Our standard induction alternative: Select a special point for the nth point – some sort of min or max point.

If we always pick the point with max x, what problem is eliminated?
Stretch:
1. Find vertices to eliminate
2. Add new vertex between existing vertices.



Why? Lots of points don't affect the hull, and stretching is expensive.

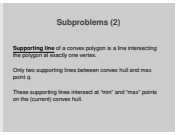
Subproblem 1 can be eliminated: the max is always outside the polygon.

Subproblems (2)

Supporting line of a convex polygon is a line intersecting the polygon at exactly one vertex.

Only two supporting lines between convex hull and max point q.

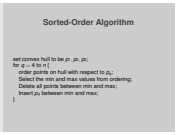
These supporting lines intersect at "min" and "max" points on the (current) convex hull.



"Min" and "max" with respect to the angle formed by the supporting lines.

Sorted-Order Algorithm

set convex hull to be p_1, p_2, p_3;
for $q = 4$ to n {
  order points on hull with respect to p_q;
  Select the min and max values from ordering;
  Delete all points between min and max;
  Insert p_q between min and max;
}



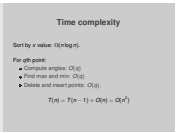
Sort by x value.

Time complexity

Sort by x value: $\mathrm{O}(n \log n)$.

For qth point:
- Compute angles: $O(q)$
- Find max and min: $O(q)$
- Delete and insert points: $O(q)$.

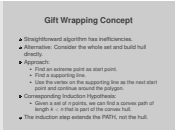
$$T(n) = T(n-1) + O(n) = O(n^2)$$



no notes

Gift Wrapping Concept

- Straightforward algorithm has inefficiencies.
- Alternative: Consider the whole set and build hull directly.
- Approach:
  - Find an extreme point as start point.
  - Find a supporting line.
  - Use the vertex on the supporting line as the next start point and continue around the polygon.
- Corresponding Induction Hypothesis:
  - Given a set of n points, we can find a convex path of length $k < n$ that is part of the convex hull.
- The induction step extends the PATH, not the hull.

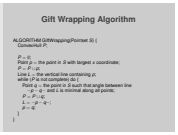


Straightforward algorithm spends time to build convex hull with points interior to final convex hull.

Gift Wrapping Algorithm

```
ALGORITHM GiftWrapping(Pointset S) {
  ConvexHull P;

  P = ∅;
  Point p = the point in S with largest x coordinate;
  P = P ∪ p;
  Line L = the vertical line containing p;
  while (P is not complete) do {
    Point q = the point in S such that angle between line
        −p − q− and L is minimal along all points;
    P = P ∪ q;
    L = −p − q−;
    p = q;
  }
}
```

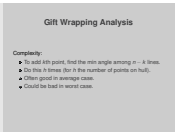



no notes

Gift Wrapping Analysis

Complexity:

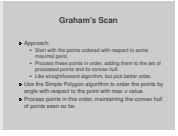
- To add kth point, find the min angle among $n - k$ lines.
- Do this h times (for h the number of points on hull).
- Often good in average case.
- Could be bad in worst case.



$O(n^2)$. Actually, $O(hn)$ where h is the number of edges to hull.

Graham's Scan

- Approach:
  - Start with the points ordered with respect to some maximal point.
  - Process these points in order, adding them to the set of processed points and its convex hull.
  - Like straightforward algorithm, but pick better order.
- Use the Simple Polygon algorithm to order the points by angle with respect to the point with max x value.
- Process points in this order, maintaining the convex hull of points seen so far.



See Manber Figure 8.11.

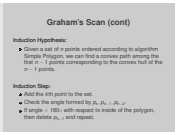
Graham's Scan (cont)

Induction Hypothesis:

- Given a set of n points ordered according to algorithm Simple Polygon, we can find a convex path among the first $n-1$ points corresponding to the convex hull of the $n-1$ points.

Induction Step:

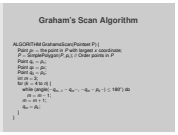
- Add the kth point to the set.
- Check the angle formed by p_k, p_{k-1}, p_{k-2}.
- If angle $< 180\circ$ with respect to inside of the polygon, then delete p_{k-1} and repeat.



no notes

Graham's Scan Algorithm

```
ALGORITHM GrahamsScan(Pointset P) {
  Point p_1 = the point in P with largest x coordinate;
  P = SimplePolygon(P, p_1); // Order points in P
  Point q_1 = p_1;
  Point q_2 = p_2;
  Point q_3 = p_3;
  int m = 3;
  for (k = 4 to n) {
    while (angle(-q_{m-1} - q_m-, -q_m - p_k-) ≤ 180°) do
      m = m - 1;
    m = m + 1;
    q_m = p_k;
  }
}
```

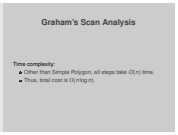



no notes

Graham's Scan Analysis

Time complexity:

- Other than Simple Polygon, all steps take $O(n)$ time.
- Thus, total cost is $\mathrm{O}(n \log n)$.



no notes

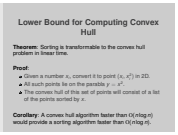
Lower Bound for Computing Convex Hull

Theorem: Sorting is transformable to the convex hull problem in linear time.

Proof:

- Given a number x_i, convert it to point (x_i, x_i^2) in 2D.
- All such points lie on the parabla $y = x^2$.
- The convex hull of this set of points will consist of a list of the points sorted by x.

Corollary: A convex hull algorithm faster than $\mathrm{O}(n \log n)$ would provide a sorting algorithm faster than $\mathrm{O}(n \log n)$.



WARNING: These are the most important two slides of the semester!

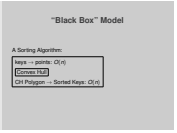
“Black Box” Model

A Sorting Algorithm:

keys → points: $O(n)$

Convex Hull

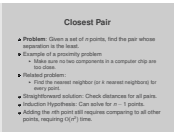
CH Polygon → Sorted Keys: $O(n)$



This is the fundamental concept of a reduction. We will use this constantly for the rest of the semester.

Closest Pair

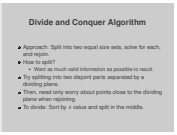
- **Problem**: Given a set of n points, find the pair whose separation is the least.
- Example of a proximity problem
  - Make sure no two components in a computer chip are too close.
- Related problem:
  - Find the nearest neighbor (or k nearest neighbors) for every point.
- Straightforward solution: Check distances for all pairs.
- Induction Hypothesis: Can solve for $n-1$ points.
- Adding the nth point still requires comparing to all other points, requiring $\mathrm{O}(n^2)$ time.



Next try: Ordering the points by x value still doesn't help.

Divide and Conquer Algorithm

- Approach: Split into two equal size sets, solve for each, and rejoin.
- How to split?
  - Want as much valid information as possible to result.
- Try splitting into two disjoint parts separated by a dividing plane.
- Then, need only worry about points close to the dividing plane when rejoining.
- To divide: Sort by x value and split in the middle.



Assume $n = 2^k$ points.

Note: We will actually compute smallest distance, not pair of points with smallest distance.

Closest Pair Algorithm

Induction Hypothesis:

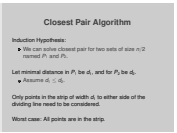
- We can solve closest pair for two sets of size $n/2$ named P_1 and P_2.

Let minimal distance in P_1 be d_1, and for P_2 be d_2.

- Assume $d_1 \leq d_2$.

Only points in the strip of width d_1 to either side of the dividing line need to be considered.

Worst case: All points are in the strip.



See Manber Figure 8.13

Closest Pair Algorithm (cont)

Observation:

- A single point can be close to only a limited number of points from the other set.

Reason: Points in the other set are at least d_1 distance apart.

Sorting by y value limits the search required.

Closest Pair Algorithm (cont)

Observation:
- A single point can be close to only a limited number of points from the other set.

Reason: Points in the other set are at least d_1 distance apart.

Sorting by y value limits the search required.

See Manber Figure 8.14

Closest Pair Algorithm Cost

$O(n \log n)$ to sort by x coordinates.

Eliminate points outside strip: $O(n)$.

Sort according to y coordinate: $O(n \log n)$.

Scan points in strip, comparing against the other strip: $O(n)$.

$T(n) = 2T(n/2) + O(n \log n)$.

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

Closest Pair Algorithm Cost

$O(n \log n)$ to sort by x coordinates.

Eliminate points outside strip: $O(n)$.

Sort according to y coordinate: $O(n \log n)$.

Scan points in strip, comparing against the other strip: $O(n)$.

$T(n) = 2T(n/2) + O(n \log n)$.

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

no notes

A Faster Algorithm

The bottleneck was sorting by y coordinate.

If solving the subproblem gave us a sorted set, this would be avoided.

Strengthen the induction hypothesis:

- Given a set of $< n$ points, we know how to find the closest distance and how to output the set ordered by the points' y coordinates.

All we need do is merge the two sorted sets – an $O(n)$ step.

$T(n) = 2T(n/2) + O(n)$.
$T(n) = O(n \log n)$.

A Faster Algorithm

The bottleneck was sorting by y coordinate.

If solving the subproblem gave us a sorted set, this would be avoided.

Strengthen the induction hypothesis:
- Given a set of $< n$ points, we know how to find the closest distance and how to output the set ordered by the points' y coordinates.

All we need do is merge the two sorted sets – an $O(n)$ step.

$T(n) = 2T(n/2) + O(n)$.
$T(n) = O(n \log n)$.

no notes

Horizontal and Vertical Segments

- Intersection Problems:
  - Detect if any intersections ...
  - Report any intersections ...

  ... of a set of <line segments>.
- We can simplify the problem by restricting to vertical and horizontal line segments.
- Example applications:
  - Determine if wires or components of a VLSI design cross.
  - Determine if they are too close.
    - Solution: Expand by 1/2 the tolerance distance and check for intersection.
  - Hidden line/hidden surface elimination for Computer Graphics.

Horizontal and Vertical Segments

- Intersection Problems:
  - Detect if any intersections ...
  - Report any intersections ...

  ... of a set of <line segments>.
- We can simplify the problem by restricting to vertical and horizontal line segments.
- Example applications:
  - Determine if wires or components of a VLSI design cross.
  - Determine if they are too close.
    - Solution: Expand by 1/2 the tolerance distance and check for intersection.
  - Hidden line/hidden surface elimination for Computer Graphics.

no notes

Sweep Line Algorithms (1)

Problem: Given a set of n horizontal and m vertical line segments, find all intersections between them.

- Assume no intersections between 2 vertical or 2 horizontal lines.

Straightforward algorithm: Make all $n \times m$ comparisons.

If there are $n \times m$ intersections, this cannot be avoided.

However, we would like to do better when there are fewer intersections.

Solution: Special order of induction will be imposed by a **sweep line**.

Sweep Line Algorithms (1)
Problem: Given a set of n horizontal and m vertical line segments, find all intersections between them.
▸ Assume no intersections between 2 vertical or 2 horizontal lines.
Straightforward algorithm: Make all n × m comparisons.
If there are n × m intersections, this cannot be avoided.
However, we would like to do better when there are fewer intersections.
Solution: Special order of induction will be imposed by a sweep line.

This is a "classic" computational geometry problem/algorithm

Sweep Line Algorithms (2)

Plane sweep or **sweep line** algorithms pass an imaginary line through the set of objects.

As objects are encountered, they are stored in a data structure.

When the sweep passes, they are removed.

Preprocessing Step:

- Sort all line segments by x coordinate.

Sweep Line Algorithms (2)
Plane sweep or sweep line algorithms pass an imaginary line through the set of objects.
As objects are encountered, they are stored in a data structure.
When the sweep passes, they are removed.
Preprocessing Step:
▸ Sort all line segments by x coordinate.

The induction here is to add a special nth element.

Sweep Line Algorithms (3)

Inductive approach:

- We have already processed the first $k-1$ end points when we encounter endpoint k.
- Furthermore, we store necessary information about the previous line segments to efficiently calculate intersections with the line for point k.

Possible approaches:

1. Store vertical lines, calculate intersection for horizontal lines.
2. Store horizontal lines, calculate intersection for vertical lines.

Sweep Line Algorithms (3)
Inductive approach:
▸ We have already processed the first k − 1 end points when we encounter endpoint k.
▸ Furthermore, we store necessary information about the previous line segments to efficiently calculate intersections with the line for point k.
Possible approaches:
1. Store vertical lines, calculate intersection for horizontal lines.
2. Store horizontal lines, calculate intersection for vertical lines.

Since we processed by x coordinate (i.e., sweeping horizontally) do (2). When we process a vertical line, it is clear which horizontal lines would be relevent (the ones that cross that include the x coordinate of the vertical line), and so could hope to find them in a data structure. If we stored vertical lines, when we process the next horizontal line, it is not so obvious how to find all vertical lines in the horizontal range.

Organizing Sweep Info

What do we need when encountering line L?

- NOT horizontal lines whose right endpoint is to the left of L.
- Maintain **active** line segments.

What do we check for intersection?

Induction Hypothesis:

- Given a list of k sorted coordinates, we know how to report all intersections among the corresponding lines that occur to the left of $k.x$, and to eliminate horizontal lines to the left of k.

Organizing Sweep Info
What do we need when encountering line L?
▸ NOT horizontal lines whose right endpoint is to the left of L.
▸ Maintain active line segments.
What do we check for intersection?
Induction Hypothesis:
▸ Given a list of k sorted coordinates, we know how to report all intersections among the corresponding lines that occur to the left of k.x, and to eliminate horizontal lines to the left of k.

See Figure 8.17 in Manber.

y coordinates of the active horizontal lines.

Sweep Line Tasks

Things to do:

1. $(k+1)$th endpoint is right endpoint of horizontal line.
   - Delete horizontal line.
2. $(k+1)$th endpoint is left endpoint of horizontal line.
   - Insert horizontal line.
3. $(k+1)$th endpoint is vertical line.
   - Find intersections with stored horizontal lines.

Sweep Line Tasks

Things to do:
(k + 1)th endpoint is right endpoint of horizontal line.
Delete horizontal line.
(k + 1)th endpoint is left endpoint of horizontal line.
Insert horizontal line.
(k + 1)th endpoint is vertical line.
Find intersections with stored horizontal lines.

Deleting horizontal line is $\mathrm{O}(\log n)$.

Inserting horizontal line is $\mathrm{O}(\log n)$.

Finding intersections is $O(\log n + r)$ for r intersections.

Data Structure Requirements (1)

To have an efficient algorithm, we need efficient

- Intersection
- Deletion
- 1 dimensional range query

Example solution: Balanced search tree

- Insert, delete, locate in $\log n$ time.
- Each additional intersection calculation is of constant cost beyond first (traversal of tree).

Data Structure Requirements (1)

To have an efficient algorithm, we need efficient
Intersection
Deletion
1 dimensional range query

Example solution: Balanced search tree
Insert, delete, locate in log n time.
Each additional intersection calculation is of constant cost beyond first (traversal of tree).

no notes

Data Structure Requirements (2)

Time complexity:

- Sort by x: $O((m+n)\log(m+n))$.
- Each insert/delete: $\mathrm{O}(\log n)$.
- Total cost is $\mathrm{O}(n \log n)$ for horizontal lines.

Processing vertical lines includes one-dimensional range query:

- $O(\log n + r)$ where r is the number of intersections for this line.

Thus, total time is $O((m+n)\log(m+n) + R)$, where R is the total number of intersections.

Data Structure Requirements (2)

Time complexity:
Sort by x: O((m + n) log(m + n)).
Each insert/delete: O(log n).
Total cost is O(n log n) for horizontal lines.

Processing vertical lines includes one-dimensional range query:
O(log n + r) where r is the number of intersections for this line.

Thus, total time is O((m + n) log(m + n) + R), where R is the total number of intersections.

no notes

Reductions

A **reduction** is a transformation of one problem to another

Purpose: To compare the relative difficulty of two problems

Example:
Sorting reals **reduces to** (in linear time) the problem of finding a convex hull in two dimensions

- Use CH as a way to solve sorting

We argued that there is a lower bound of $\Omega(n \log n)$ on finding the convex hull since there is a lower bound of $\Omega(n \log n)$ on sorting

Reductions

A reduction is a transformation of one problem to another

Purpose: To compare the relative difficulty of two problems

Example:
Sorting reals reduces to (in linear time) the problem of finding a convex hull in two dimensions
Use CH as a way to solve sorting

We argued that there is a lower bound of Ω(n log n) on finding the convex hull since there is a lower bound of Ω(n log n) on sorting

This example we have already seen.

NOT reduce CH to sorting – that just means that we can make CH as hard as sorting! Using sorting isn't necessarily the only way to solve the CH problem, perhaps there is a better way. So just knowing that sorting is ONE WAY to solve CH doesn't tell us anything about the cost of CH. On the other hand, by showing that we can use CH as a tool to solve sorting, we know that CH cannot be faster than sorting.

Reduction Notation

- We denote names of problems with all capital letters.
  - Ex: SORTING, CONVEX HULL
- What is a problem?
  - A relation consisting of ordered pairs (**I**, **SLN**).
  - **I** comes from the set of **instances** (allowed inputs).
  - **SLN** is the solution to the problem for instance **I**.
- Example: SORTING $=$ (**I**, **SLN**).
  I is a finite subset of $\mathcal{R}$.
  - Prototypical instance: $\{x_1, x_2, ..., x_n\}$.
- **SLN** is the sequence of reals from **I** in sorted order.

Reduction Notation
- We denote names of problems with all capital letters.
  - Ex: SORTING, CONVEX HULL
- What is a problem?
  - A relation consisting of ordered pairs (**I**, **SLN**).
  - **I** comes from the set of **instances** (allowed inputs).
  - **SLN** is the solution to the problem for instance **I**.
- Example: SORTING $=$ (**I**, **SLN**).
  I is a finite subset of $\mathcal{R}$.
  - Prototypical instance: $\{x_1, x_2, ..., x_n\}$.
- **SLN** is the sequence of reals from **I** in sorted order.

no notes

Black Box Reduction (1)

The job of an algorithm is to take an instance **I** and return a solution **SLN**, or to report that there is no solution.

A **reduction** from problem **A**(**I**, **SLN**) to problem **B**(**I**', **SLN**') requires two transformations (functions) T, T'.
T: $\mathbf{I} \Rightarrow \mathbf{I}'$
- Maps instances of the first problem to instances of the second.

T': $\mathbf{SLN}' \Rightarrow \mathbf{SLN}$
- Maps solutions of the second problem to solutions of the first.

Black Box Reduction (1)
The job of an algorithm is to take an instance **I** and return a solution **SLN**, or to report that there is no solution.
A **reduction** from problem **A**(**I**, **SLN**) to problem **B**(**I**', **SLN**') requires two transformations (functions) T, T'.
T: $\mathbf{I} \Rightarrow \mathbf{I}'$
- Maps instances of the first problem to instances of the second.

T': $\mathbf{SLN}' \Rightarrow \mathbf{SLN}$
- Maps solutions of the second problem to solutions of the first.

no notes

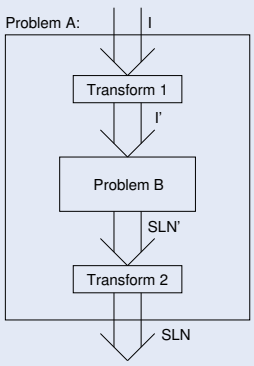
Black Box Reduction (2)

Black box idea:
1. Start with an instance **I** of problem **A**.
2. Transform to an instance **I**' $=$ T(**I**), an instance of problem **B**.
3. Use a "black box" algorithm for **B** as a subroutine to find a solution **SLN**' for **B**.
4. Transform to a solution **SLN** $=$ T'(**SLN**'), a solution to the original instance **I** for problem **A**.

Black Box Reduction (2)
Black box idea:
1. Start with an instance **I** of problem **A**.
2. Transform to an instance **I**' $=$ T(**I**), an instance of problem **B**.
3. Use a "black box" algorithm for **B** as a subroutine to find a solution **SLN**' for **B**.
4. Transform to a solution **SLN** $=$ T'(**SLN**'), a solution to the original instance **I** for problem **A**.

no notes

Black Box Diagram



Black Box Diagram


no notes

More Notation

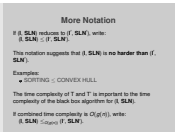
If (**I**, **SLN**) reduces to (**I**′, **SLN**′), write:
(**I**, **SLN**) $\leq$ (**I**′, **SLN**′).

This notation suggests that (**I**, **SLN**) is **no harder than** (**I**′, **SLN**′).

Examples:
- SORTING $\leq$ CONVEX HULL

The time complexity of T and T' is important to the time complexity of the black box algorithm for (**I**, **SLN**).

If combined time complexity is $O(g(n))$, write:
(**I**, **SLN**) $\leq_{O(g(n))}$ (**I**′, **SLN**′).



Sorting is no harder than Convex Hull. Conversely, Convex Hull is *at least as hard as* Sorting.

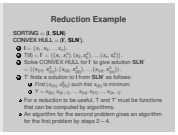
If T or T' is expensive, then we have proved nothing about the relative bounds.

Reduction Example

SORTING = (**I**, **SLN**)
CONVEX HULL = (**I**', **SLN**').

1. **I** $= \{x_1, x_2, ..., x_n\}$.
2. T(**I**) = **I**' $= \{(x_1, x_1^2), (x_2, x_2^2), ..., (x_n, x_n^2)\}$.
3. Solve CONVEX HULL for **I**' to give solution **SLN**' $= \{(x_{i[1]}, x_{i[1]}^2), (x_{i[2]}, x_{i[2]}^2), ..., (x_{i[n]}, x_{i[n]}^2)\}$.
4. T' finds a solution to **I** from **SLN**' as follows:
   1. Find $(x_{i[k]}, x_{i[k]}^2)$ such that $x_{i[k]}$ is minimum.
   2. Y $= x_{i[k]}, x_{i[k+1]}, ..., x_{i[n]}, x_{i[1]}, ..., x_{i[k-1]}$.

- For a reduction to be useful, T and T' must be functions that can be computed by algorithms.
- An algorithm for the second problem gives an algorithm for the first problem by steps 2 – 4.



no notes

Notation Warning

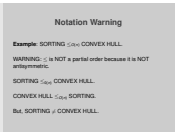
Example: SORTING $\leq_{O(n)}$ CONVEX HULL.

WARNING: $\leq$ is NOT a partial order because it is NOT antisymmetric.

SORTING $\leq_{0(n)}$ CONVEX HULL.

CONVEX HULL $\leq_{O(n)}$ SORTING.

But, SORTING $\neq$ CONVEX HULL.



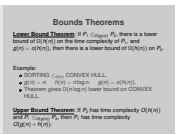
no notes

Bounds Theorems

Lower Bound Theorem: If $P_1 \leq_{O(g(n))} P_2$, there is a lower bound of $\Omega(h(n))$ on the time complexity of P_1, and $g(n) = o(h(n))$, then there is a lower bound of $\Omega(h(n))$ on P_2.

Example:
- SORTING $\leq_{O(n)}$ CONVEX HULL.
- $g(n) = n$. $h(n) = n \log n$. $g(n) = o(h(n))$.
- Theorem gives $\Omega(n \log n)$ lower bound on CONVEX HULL.

Upper Bound Theorem: If P_2 has time complexity $O(h(n))$ and $P_1 \leq_{O(g(n))} P_2$, then P_1 has time complexity $O(g(n) + h(n))$.



Notice o, not O.So, given good transformations, both problems take at least $\Omega(P_1)$ and at most $O(P_2)$.