Variants on Convex Hull Problem

Lecture #16: 3 March 1992
Scribe: Michael Goldwasser

1 The Standard Problem

The standard convex hull problem is the following: Given n points in E^d, the d-dimension Euclidean space, compute the convex hull of the points. The output is usually required to be in one of two forms, either return the entire structure of the hull as a $(d-1)$-dimensional polytope, or more flexibly return the set of points lying on the hull in no particular order. From earlier classes, we recall the following two results.

Theorem 1. *The computational complexity of computing the convex hull of n points in E^2, the Euclidian plane, is $\Theta(n \log n)$. This is true for both computing the topology of the hull, and for computing the set of points lying on the hull.*

Theorem 2. *The topological complexity of the convex hull of n points in E^d, the d-dimension Euclidean space, is $\Theta(n^{\lfloor \frac{d}{2} \rfloor})$.*

2 Convex Hull of Simple Polygon

Another version of the problem is given a simple polygon, compute the convex hull of the points. Clearly this problem is at least as easy the original problem and in fact the extra structure of the polygon edges allows us to solve this problem in linear time.

2.1 Graham-Yao algorithm

A well known algorithm for this problem was given by Graham and Yao. The details of it are not given here, but the general idea is to walk around the boundary of the polygon maintaining a stack of vertices, and removing vertices from the stack that cause a concave angle in hull begin built. The algorithm is optimal in using both linear time and space.

2.2 Melkman's algorithm

A lesser known algorithm was created by Melkman. This algorithm also is linear in both time and space. The key to this algorithm is the use of a data structure called a *dequeue*, a double-ended queue. This structure maintains a list of items, and allows arbitrary insertions or deletions from either end of the list.

The algorithm proceeds by starting with at an arbitrary vertex of the polygon, and incrementally walking around the polygon in one direction processing new vertices. The dequeue is maintained to simulate a circular queue, and so after each step, the two end elements are the same. The algorithm maintains the invariant that the dequeue contains the convex hull of all the points processed at that time.

As we process each vertex around the polygon, there are two possibilities. In constant time, we can check the segment connecting the new vertex to the previous vertex, and determine whether or not the new vertex is inside the working hull. If the new vertex is inside the hull, we throw it away and continue around the polygon. If the new vertex is outside the hull, then we prepare to add it to both ends of the queue. However, before doing this we must remove vertices from each end which are eliminated from the hull. So for each end of the dequeue, we consider the top vertex, and the angle created by the new vertex, the top vertex, and the following vertex, and if that angle is concave, then we pop the top vertex off the dequeue, and continue the process until a convex angle is found. This process is known as a *Graham Scan*. Figure 1 demonstrates a sample step in the algorithm. Notice that after the final vertex is processed, then the invariant is that the dequeue represents the convex hull of all the points. Also, since each vertex can be added or deleted from the dequeue at most two times, then both the time and space are clearly linear.

3 Dynamic Algorithms for Convex Hull

Another problem is to efficiently maintain the convex hull of a set of points, while processing some sequence of insertions and deletions of the point set. The obvious goal is to get a worst case bound of $O(\log n)$ on any insertion or deletion, while using only linear space. However whether or not this goal can be reached is an open problem.

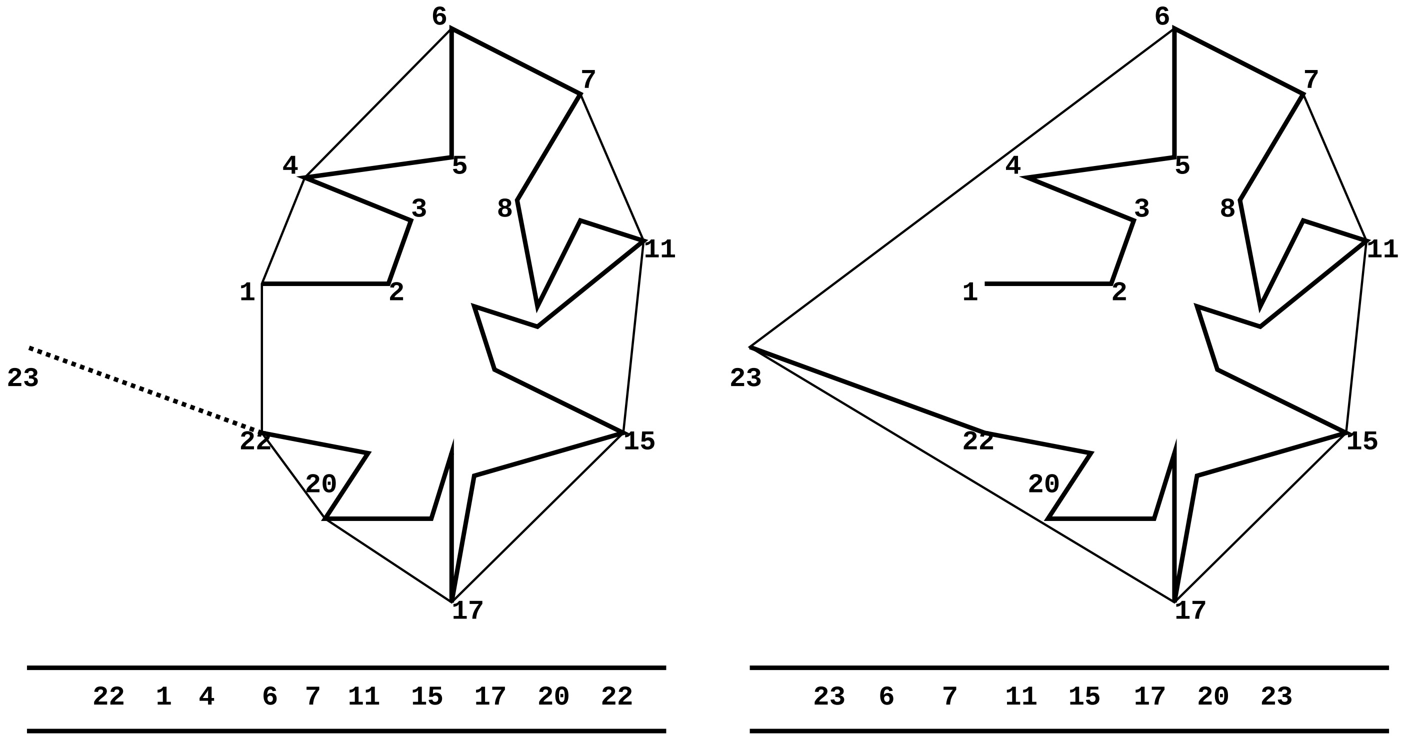

Figure 1: Melkman's Algorithm

3.1 Insertions

Handling only insertions of points turns out to be rather simple. In fact this can be done in $O(\log n)$ time for each insertion, storing nothing except the current convex hull as a polygon. (For both insertions and deletions, however, we need to store the points in a *concatenable queue*, which allows searching, splitting, and joining in logarithmic time.)

Lemma 3. *Given the convex hull, determining whether a query point lies inside the current hull can be done in $O(\log n)$ time.*

Proof: There are various methods. Given a query point, one such method is to consider the vertical line through that point. Then by considering the upper and lower hull separately, with a binary search on the x-coordinates, it is possible to locate where that vertical line cuts the two hulls, and then the y-coordinate of the query point can be checked against the intersections.

A second method is to pick some arbitrary point of the hull, and consider the rays sent out from that point to all the other points on the hull. Considering the

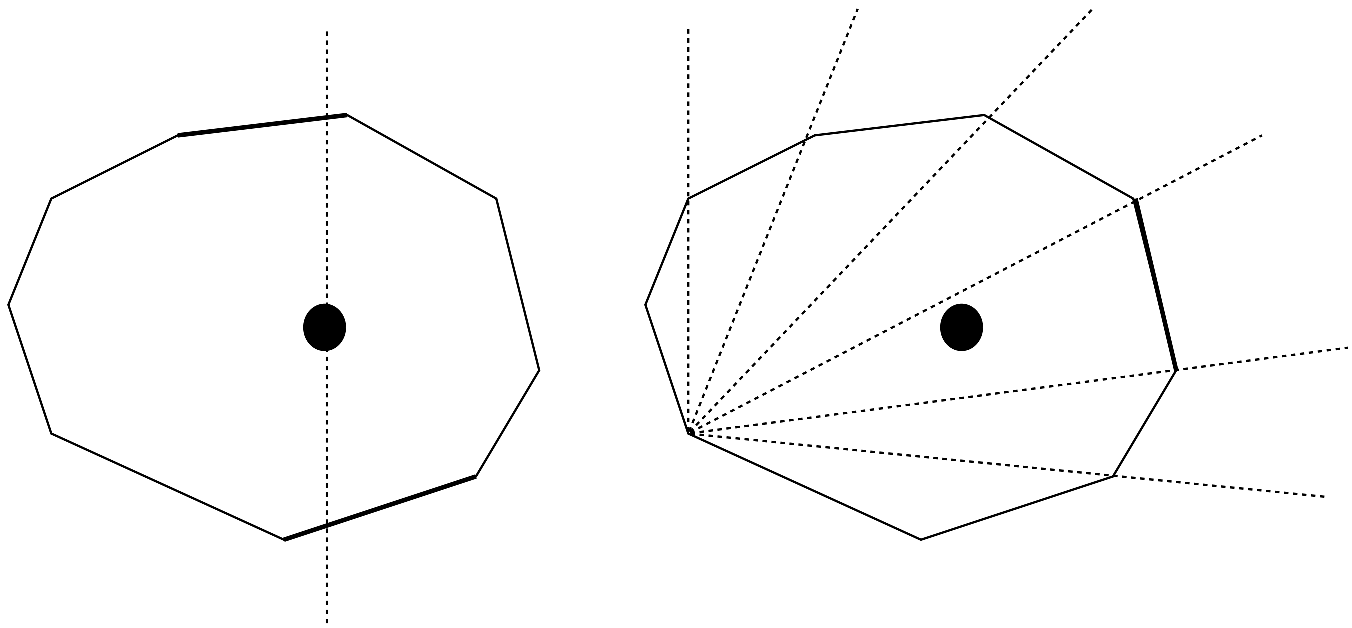

Figure 2: Point Location for Convex Hull

angles made between that point and some reference, this divides up the plane into angular intervals, and then given a query point, the proper interval can be located using a binary search on the angle. Once that slice is found, in constant time the query point can be compared to the hull edge falling in that interval. Figure 2 demonstrates these techniques. □

Lemma 4. *Given the convex hull and a query point outside of the hull, determining the two supporting tangents to the hull from the query point can be done in* $O(\log n)$ *time.*

Proof: For each tangent, a binary search strategy can be used to locate the supporting vertex on the hull. The key is in defining the relationships "less than", "equal", and "greater than", used in the binary search strategy to decide which direction to move. Given a query point and some key point on the hull, consider the line segment connecting the two. We will define the comparison relation by looking at this line and the two polygon edges leaving the key vertex. If both edges are on the same side of the line, then this is a supporting tangent, and we call this case "equal". If the two edges are on opposite sides of our line, then if the angle between them containing the query point is convex, we call this "less than", if that angle is concave, we call this "greater than." These three cases are

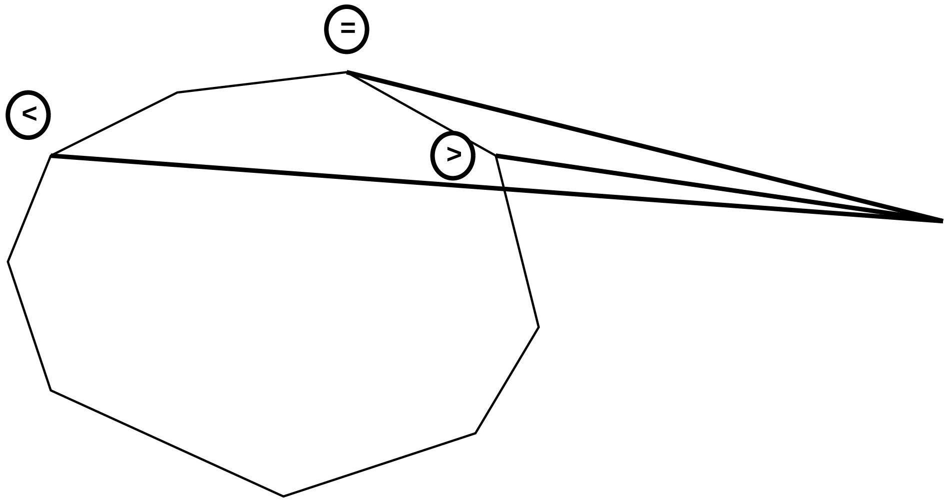

Figure 3: Locating Supporting Tangent

displayed in Figure 3. Under this comparison strategy, the sequence of angles around the polygon is monotone, and thus a binary search strategy will converge successfully. □

Now the strategy for insertions is simple. The current convex hull is maintained in a concatenable queue, such that insertion/deletion of chains can be added to any place in the queue in constant time, but so that searching can be done in logarithmic time. For each insertion, given the hull prior to that insertion, use the method in lemma 3 to see whether the point is inside the current hull. If it is, throw it away and continue. If it is outside the current hull, use the method in lemma 4 to find the two supporting vertices, then delete the chain between them and add in the two supporting tangents in its place.

3.2 Deletions

Whereas insertions could be done quickly and with no complicated data structures, deletions are not so simple. In fact the method we present will create a complicated (but linear) data structure just to achieve a worst case bound of $O(\log^2 n)$ time.

The first obvious fact to notice is that if we store nothing but the current convex hull, we cannot do better than $O(n \log n)$ time for deletions. A point set with a very simple hull, can become complicated when one of the hull points is removed. Figure 4 shows such a bad example. With the point set shown, the current hull is simple a rectangle, and if the only structure stored is that

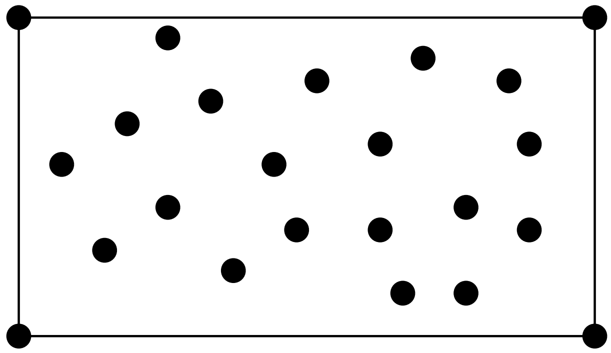

Figure 4: Lower Bound for Deletions

rectangle, then removing the four points forces us to compute the convex hull of the interior points from scratch, requiring $O(n \log n)$ time.

We present a strategy for insertions and deletions that uses linear storage and guarantees $O(\log^2 n)$ time. We will develop this strategy by successively refining several key intuitions.

For starters, we will break up the problem into maintaining the upper convex hull and the lower convex hull separately. Clearly with both of these available, we can provide the complete convex hull. We will only consider the problem of maintaining the upper convex hull. (the lower hull can be maintained symmetrically.)

The first key idea is to maintain the hull in a hierarchical manner. We maintain the points in a balanced binary tree representing a concatenable queue based on the x-coordinates. At each node, we would like to maintain the upper convex hull of all the points in the subtree rooted there. In this manner, we will always ensure that the upper convex hull of the entire point set is at the root of the tree.

However, this method may require $O(n \log n)$ space. If all of the points actually lie on the upper convex hull of the entire point set, then every node will contain a chain whose size is exactly the size of its subtree, and the tree will require $O(n \log n)$ storage. So the second idea is to utilize the relationship between the chains stored at a node and at its children. If a node had the upper hulls of both children, then its own hull is made up of the upper common tangent of the two children, together with a single subchain from each of the childrens' hulls. And so by only storing just the common tangent, and a pointer to the two points where that tangent connects to the childrens' hulls, each node can recursively calculate its own upper hull, and the total storage used is linear.

There are two problems with the above scheme. The original dynamic problem requires that after each step, we have the upper convex hull as a single polygonal chain, however in this manner, specifying the root's hull requires a series of pointers through the tree. Also, the tree balancing will require rotations, and maintaining these hulls through rotations does not appear to be simple. Therefore we make one last modification to arrive at the actual implementation. Rather than storing the two children hulls at the lower level and the common tangent at the parent, we will move all the information upward in the tree.

The root of the tree will store the entire upper convex hull in a single chain. Then from the child's point of view, the upper hull of its subtree consists of a subchain of the parent leading to the place with the common tangent jumped to the other half, along with a section of its upper hull that was not used by its parent. Therefore each child will store the part of its hull that was eliminated by the parent, and it will store a pointer to the place in the parent's chain where the common tangent broke off. In this manner, the root of the tree actually has the entire upper convex hull at its level, and also the tree rotations will be much easier to handle in this method.

Maintaining a balanced binary tree requires $O(\log n)$ rotations. In a standard binary tree, each rotation involves updating the pointers to the four subtrees, and hence takes constant time, allowing insertions and deletions to be done in $O(\log n)$ time. The rotations for the binary tree with our additional information is not quite so simple. The basic subroutine needed is to find the common tangent between two hulls of separate intervals. We will call this common tangent the *bridge*.

Lemma 5. *Given point set $S_1 : |S_1| = n_1$ and point set $S_2 : |S_2| = n_2$ and a vertical line separating all points from S_1 and S_2, the upper common tangent can be computed in $O(\log(n_1 + n_2))$ time.*

Proof: We will use a technique very similar to the binary search used in lemma 4. We start by taking the middle point of each upper hull. The current point from S_1 divides the left upper hull into two subchains which well will label LL and LR. Similarly the point chosen from the right chain will create the two subchains RL and RR. At each step we will throw out one of these four subchains, and then will recurse. Every time we throw out a section from the left side, the size of that set reduces by a half, and so we can have at most $O(\log n_1)$ such steps. Similarly we can have at most $O(\log n_2)$ steps throwing out a subchain from the right side, and so the total number of steps is $O(\log n_1 + \log n_2) = O(\log(n_1, n_2))$.

All that remains is to demonstrate that we may always throw out one of the four subchains. Given the two middle points, consider the line connecting them.

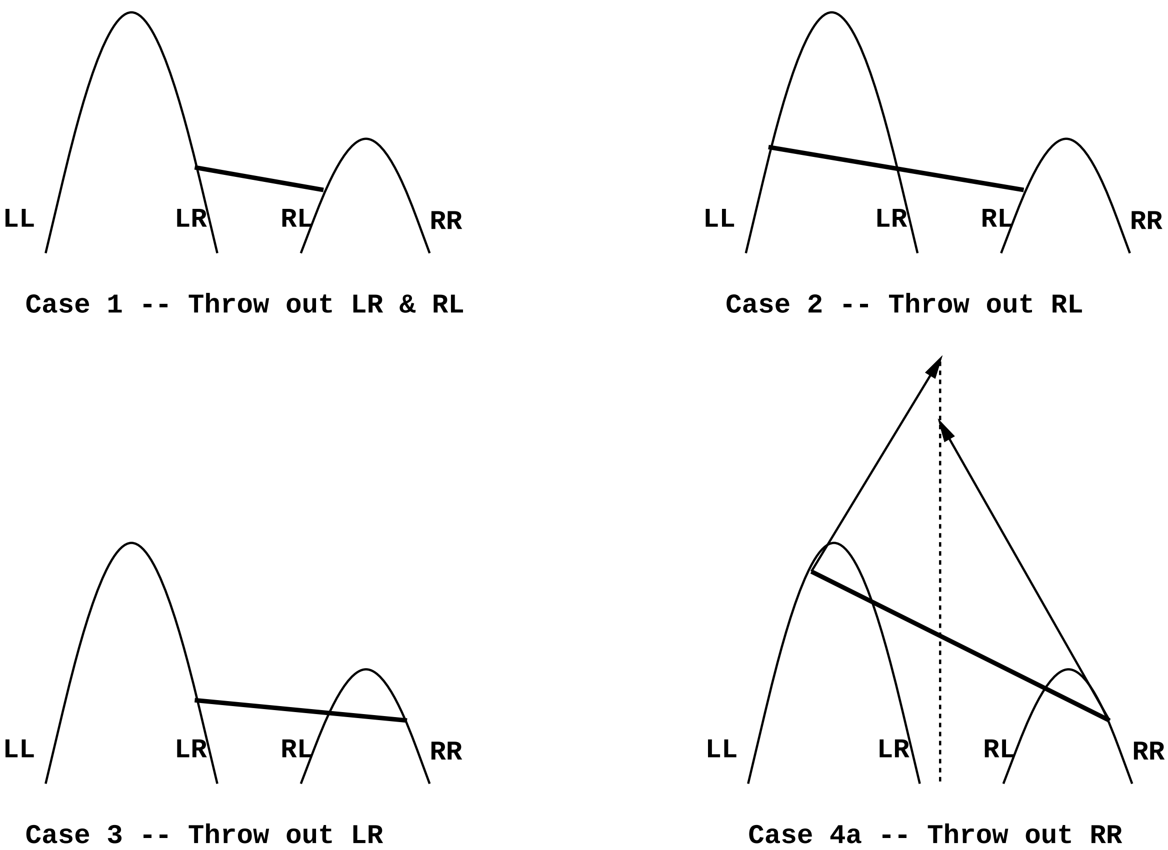

Figure 5: Locating The Bridge

We will use the same comparison scheme shown in Figure 3 to create four cases based on whether this line intersects each side in one or two places. The four cases are shown in Figure 5.

- Case 1 – The line intersects each side once. In this case we can actually throw out both LR and RL subchains since the bridge must be raised on both sides.
- Case 2 – The line intersects the left side twice, but the right side once. In this case we can throw out the subchain RL, because the right side of the bridge must be raised. Notice however that we cannot necessarily eliminate LL, because it is possible that the right side of the line will be raised so

much that the common tangent hits the left even farther over.

- Case 3 – The line intersects the left side once, but the right side twice. This case is symmetric to the second case, and thus subchain LR can be eliminated

- Case 4 – The line intersects each side twice. This case is slightly more complex. For each side, consider a line tangent to that side of the hull through the selected point, and now consider where those two tangent lines intersect the vertical line separating the two sides. Whichever of these two lines has the lower intersection point must shield the other line. If the left side's tangent line shields the right side's line, then it is safe to throw away the subchain LL. If the reverse is true, then subchain RR can be thrown away.

In each case, one of the four subchains is thrown out, and thus we can locate the bridge between the two upper hulls in $O(\log(n_1, n_2))$ time. $\square$

4 Output Sensitive Algorithms for Convex Hull

In the worst case, finding the hull of n points in the plane requires $\Omega(n \log n)$ time. However it is desirable to have an algorithm which is output sensitive. In other words, if the actual convex hull is very complex, then all we can hope for is to achieve the lower bound. However, if the hull is simple, we would hope our algorithm runs faster. The lower bound of $\Omega(n \log n)$ relies on an instance with many points lying on the convex hull, and so it might be possible to do better in cases with simpler hulls.

Many standard algorithms for computing the convex hull in the plane have the problem that they maintain many points that appear to be on the hull at some stage, but later are thrown away. For instance the standard divide and conquer construction breaks the point set into two halfs separated by a vertical line, and then builds the hull of each half. Then in merging the two subproblems, the upper and lower common tangents are located and added, and a subchain from each hull is removed. However much time may have already been spent in computing a very complex subchain that is thrown away for the final solution.

Below we present an output sensitive algorithm for computing the convex hull of n points in the plane due to Kirkpatrick and Seidel. If the final hull contains k points, then their algorithm will compute the convex hull in $O(n \log k)$ time.

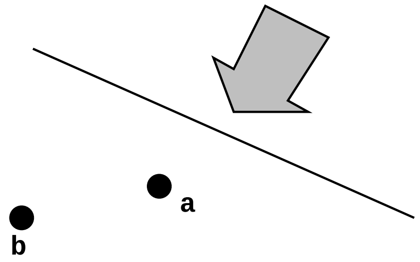

Figure 6: Eliminating Points using Decimation

4.1 Decimation

A key subroutine for the standard divide and conquer algorithm was to find the bridges between two separate point sets. If the convex hulls of both subproblems is available, then the bridges could be located in logarithmic time as discussed earlier. However, what if no structure is available other than the two sets of points.

A technique for solving this problem, called *decimation* was developed by Megiddo. The technique provides the following result.

Lemma 6. *Given point sets L and R and a vertical separator, the two bridges between the sets can be found in linear time.*

Proof: We will focus on finding just the upper bridge. (the lower bridge can be found symmetrically.) The output of the problem is a single point from L and a single point from R that define the bridge. However rather than looking for the needle in the haystack, we will start throwing away the hay until only the needle is left. If in linear time, we can throw away some fixed fraction of the points, then repeating this process creates a geometric series, and the total time to find the bridge will be linear.

The first key fact to notice is that if an oracle provided us with the slope of the bridge, then given any two points, we could throw one of the two out without affecting the final bridge. Figure 6 demonstrates this fact. The idea is that if we take a line with the predicted slope and start lowering the line towards those two points until it hits one, then the other point is not necessary for supporting the bridge. (If both points are hit at once, then either could be eliminated).

Unfortunately, we are not so lucky as to have such a kind and generous oracle. However we can do almost as well. Given the two sets of points, begin by pairing up all the points in any fashion. (Notice, a point in the left set may be paired with a point in the left or right set.) This clearly takes linear time. Now, each of these pairs has an associated slope, and using the linear time median finding

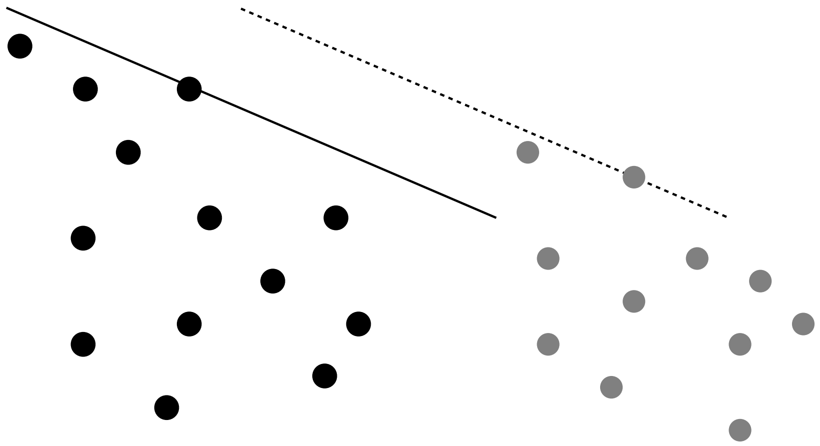

Figure 7: True Bridge must be more counterclockwise than median slope

algorithm, locate the median slope of these pairs. (In practice, picking a random pair will give the desired expected result). Now if we can determine whether the true bridge slope is greater than or less than this median slope, then we can eliminate a point from each pair whose slope is on the opposite side of the median. Since we chose the median slope, then we can eliminate one point from half the pairs, and thus in linear time we throw out one fourth of the points. By recursing, the running time is a geometric sum, and so we find the bridge in linear time.

To determine the relation between the median slope and the true bridge slope we do the following. Do a linear pass through just the points in the left set to find the "highest" point with respect to the median slope. Separately, find the highest point in the right set. If we are *very* lucky, and our median slope is the correct value, then the tangent lines through these two extreme points will actually be the same line. But if not, one of those tangent lines must be completely above the other set, and so the actual bridge slope will definitely be to that side of the median. Figure 7 demonstrates this idea.

Thus, with this last piece, we have satisfied the lemma. □

4.2 Kirkpatrick and Seidel's Algorithm

The general divided and conquer technique for convex hull breaks the point set into a left and right half, finds the hull of each half, and then uses those hulls to find the bridges in logarithmic time. The strategy used in the output-sensitive method is to break the points set into the save left and right halfs, but then first find the bridge using decimation, and then recursing on both halves.

One detail must be changed, rather than considering the problem of finding the convex hull of a point set, we want to be able to find a subchain of the convex hull from a starting and ending angle. This is important because after the bridge is found for the first subdivision, when solving the two halves, we do not want to find the complete hull, we only want to find the hull from one bridge around to the other. So at the top level, we will ask for the 360 degrees of the hull, but future recursive calls will only be interested in a subchain of the hull. Although the details are not given, the algorithm can easily be modified to restrict itself to the given range of the hull. The important fact is that all points reported by the recursive calls are actually points of the final hull.

The analysis shows that this algorithm has $O(n \log k)$ worst case running time. At each level of the recursion, the subroutine to find a bridge is linear in the number of points in the current set, however each call of the decimation routine results in one of the final edges of the convex hull, and so the recursion hs $O(\log k)$ depth. More formally, if $T(n, k)$ represents the running time on n points, where the final hull has k points, then we get the recurrence, $T(n, k) \leq T(\frac{n}{2}, k_L) + T(\frac{n}{2}, k_R) + O(n)$, where $k_L + k_R = k$. This recurrence solves to $T(n, k) = O(n \log k)$.