Computational Geometry

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2012 ; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. All rights reserved.

Computational Geometry deals with objects related to points of space. The objects that are defined in computational geometry are sets of points in Euclidean geometry. A coordinate system is assumed and a point is thus represented as a vector of cartesian coordinates of the appropriate dimension.

By E^d we denote the d-dimensional Euclidean space of the d-tuples $(x_1, \ldots, x_d)$ of real numbers with metric $(\sum_i x_i^2)^{1/2}$.

The principal objects we will be dealing with are :

Point. A d-tuple $(x_1, \ldots, x_d)$ denotes a point q of E^d.

Line. Given two distinct points q_1 and q_2 of E^d, the linear combination $aq_1 + (1-a)q_2$, where $a \in R$ is a line in E^d. More generally k points $q_1, \ldots, q_k$ define a linear variety of dimension $k-1$ defined by $a_1q_1 + \ldots + a_{k-1}q_{k-1} + (1 - a_1 - \ldots - a_{k-1})q_k$.

Line segment. Given two distinct points q_1 and q_2 of E^d, the linear combination $aq_1 + (1-a)q_2$, where $a \in R$ and $0 \leq a \leq 1$ is a line segment in E^d. The linear combination is known as the convex combination of q_1, and q_2. It defines the straight line segment connecting q_1 and q_2. The line seqgment is denoted by $\overline{q_1q_2}$.

Convex set. A domain D in E^d is convex if for any two points q_1, q_2 in D the segment $\overline{q_1q_2}$ is entirely contained in D. The intersection of convex domains is also convex.

Polygon. In E^2 a polygon is a finite set of segments such that every segment extreme is shared by exacly two edges and no subset of edges has the same property. The segments are the edges and their extremes are the vertices (nodes) of the polygon. An n-vertex polygon is an n-gon.

A polygon is simple if there is no pair of consecutive edges sharing a point. A simple polygon partitions the plane into an interior, an exterior and the polygon. A simple polygon is convex if its interior is convex. Sometimes a polygon is defined to be itself along with its interior.

Convex hull. The convex hull of a set of points S in E^d is the boundary of the smallest convex domain in E^d containing S, i.e. is the smallest convex polygon P for which each point in S is either on the boundary of P or in its inside.

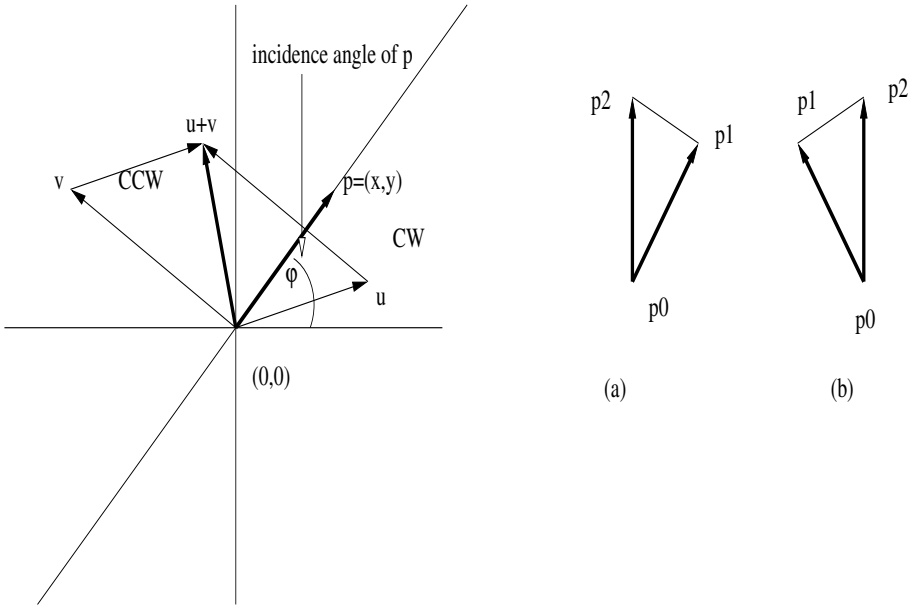


Figure 1: Points and Vectors in E^2.

In Figure 1, in E^2, a point $p = (x, y)$ is also represented by the vector joining the origin $(0, 0)$ and that point. So we can talk about point p or vector p as well (and we use p for the latter rather than $\vec{p}$).

For a given point p, the angle ϕ between the point (or its vector) and the horizontal axis/line is $\tan \phi = y/x$, where tan is the tangent function.

For vectors u and v the sum is denoted by $u + v$ and shown also in the figure as a bold-faced vector. For vector p the line running through p splits the plance into two areas (semi-planes). One area `CCW` contains vectors that are counterclockwise from p (such as (v and $u + v$) and the other area `CW` contains vectors that are clockwise from p.

For vectors u, v the cross product $u \times v$ is the area of the parallelogram defined by the end-points $(0, 0)$, u, v and $u + v$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Vectors and products

Consider two vectors $p_1 = (x_1, y_1)$, and $p_2 = (x_2, y_2)$.

Cross Product. The cross-product $p_1 \times p_2$ is defined as the determinant of a 2×2 matrix whose columns are p_1 and p_2 respectively. In other words $p_1 \times p_2 = x_1y_2 - x_2y_1 = -p_2 \times p_1$. If the cross product is 0, the two vectors are collinear (i.e. parallel to each other) and can point to the same or other direction.

Incidence Angle. The incidence angle ϕ of p_1 and p_2 with the x-axis has tangent equal to $(y_2 - y_1)/(x_2 - x_1)$.

Three point angle. If three points are given p_0, p_1, p_2 sometimes we need to determine whether a directed segment $\overline{p_0p_1}$ is closer counterclockwise or not to a vector $\overline{p_0p_2}$. To determine this we form $p_1 - p_0$ and $p_2 - p_0$ and take the cross product of $p_1 - p_0$ and $p_2 - p_0$. If it's positive then $\overline{p_0p_1}$ is clockwise from $\overline{p_0p_2}$. If Figure 1, in (a) $\overline{p_0p_1}$ is clockwise from $\overline{p_0p_2}$ and counterclockwise in (b) . For the textbook readers Figure 33.2 discusses the case of $\overline{p_0p_2}$ relative to $\overline{p_0p_1}$.

Dot Product. For two vectors p_1, p_2 the dot product between p_1 and p_2 is defined as $p_1 \cdot p_2 = x_1x_2 + y_1y_2$. The incidence angle θ between the two vectors is given by the formula $\theta = cos^{-1}((p_1 \cdot p_2)/\sqrt{(x_1^2 + y_1^2)(x_2^2 + y_2^2)})$. Depending on the sign of the dot product, θ is less than 90 degrees if the product is positive, it is equal to 90 degrees if the product is zero, and otherwise, greater than 90 degrees if it is negative.

(c) Copyright A. Gerbessiotis. All rights reserved.

The convex hull of a set of points $S = \{s_1, \ldots, s_n\}$ is the smallest convex polygon P in which every point in S is either on P or inside P. The polygon P has end-points that are in S; several points in S might not be in P.

Points in general position. In the remainder we assume that S contains at least three points (i.e. P is at least a triangle) and not three points are collinear. This is what we mean when we say that the points in S are "**in general position**".

One of the algorithms that will be presented, Jarvis-march (`Jarvis-M`) has worst-case running time $O(nh)$, where n is the number of points of S, and h is the number of points in S that are on the convex-hull. Since h can be $\Theta(n)$, this is an $O(n^2)$ algorithm in the worst-case. This is what we call an **output-sensitive** algorithm. Its running time depends on the output.

A better choice is Graham's-scan (`Graham-S`) algorithm which is $O(n \lg n)$. Two naive algorithms `Naive-CH-1` and `Naive-CH-2` are also given.

What points ARE in the convex hull? There are some points that are guaranteed to be in the convex hull. These are the extreme points of S, i.e. the points with the lowest/highest x and y coordinates. We can have at most four such points. In the algorithms that will be presented the starting point for the convex hull is one such extreme point (eg. the lowest y coordinate point).

Naive-CH-1. A (very) naive algorithm for the convex hull problem has worst-case performance $O(n^4)$. Consider an arbitrary triplet (p_1, p_2, p_3) of points. There are $\binom{n}{3}$ of them. If an arbitrary fourth point p_j is inside the triangle formed by the three points, then that fourth point p_j is discarded as a candidate for the polygon forming the convex hull. There are $O(n)$ choices for p_j, $j \neq 1, 2, 3$. The non-discarded points become part of the convex hull. The running time is obviously $O(n^4)$.

Naive-CH-2. Another naive algorithm considers arbitrary pairs of points. Let (p_1, p_2) be such a pair. If the remaining points are not on the same side of the line segment $\overline{p_1 p_2}$, this means that the segment $\overline{p_1 p_2}$ can not be part of the convex hull. There are $\binom{n}{2}$ pairs to consider with $O(n)$ points whose orientation need to be determined for each pair. A total time of $O(n^3)$ is needed to determine the segments of the convex hull (or disqualify all other segments).

Convex-Hull Algorithm Preprocessing. For each one of these two algorithms or also the ones to be presented later for determining the convex hull of S the following preprocessing steps might be added.

```
1. Identify the  points with the lowest/highest x and y coordinates.
2. If there are three or more, let them be a,b,c, or d
3. Discard all points of S in the interior of the polygon formed by a,b,c or d
```

(c) Copyright A. Gerbessiotis. All rights reserved.

```
        Jarvis-march(S,n)
0   i=0;
1.  Find point p with lowest y-coordinate point (Tiebreaker: rightmost one).
2.  i++; temp=p[i]= p ; i++;
3.  do
4.     find angle of all points relative to temp;
5.     find among them the point p2 with the smallest (positive) angle
6.     i++; p[i] = p2 ;
7.     segment (p[i-1],p[i]) is a segment of the convex hull polygon P.
8.     temp=q;
9.  while (temp != p[1])
```

Jarvis march is a greedy algorithm. Think of trying to gift-wrap the points in S with paper or better with a string/cord starting from a point that is in the hull. The way we imagine this, this is the way Jarvis march works. In other words, Jarvis march uses the idea of **Naive-CH-2** by finding successive point of S that are in the convex-hull. Any two successive points have the property used in **Naive-CH-2**, i.e. all other points of S lie on the same side of the segment determined by the two points.

The starting point of the algorithm is by locating the lowest y-coordinate point that is in the convex hull P. If there are many points with the same y coordinate, the right-most largest x-coordinate point is used. Let us call this first point p_1. We then form rays or segments starting with p_1 and ending at nother end-point s_j, where $s_j \neq p_1$. The next end-point p_2 to consider for the convex hull, is such that the line segment $\overline{p_1 p_2}$ leaves all the remaining points of S to the same side of the segment. The point p_2 to consider is the point that minimizes the incidence angle of $\overline{p_1 p_2}$ with the line that is parallel to the x-axis and runs through p_1. There are $O(n)$ points to consider, for each such point the tangent of the angle between p_1 and the point with the x-axis needs to be computed in time that is overall the candidate points equal to $O(n)$. Then the point with the lowest angle (point p_2 also in the pseudocode) can be selected in linear time and included in the convex hull. That point has the property that all other points of S all lie on the same side of that segment.

For each point that is to be in the hull $O(n)$ time is spent in angle formation and minimum-angle finding for a total of $O(nh)$. Although Jarvis march is not optimal, if $h = O(\lg n)$, i.e. not many points lie on the convex hull, its performance is close to optimal.

We next give an example to illustrate Jarvis march.

(c) Copyright A. Gerbessiotis. All rights reserved.

Example: Jarvis march

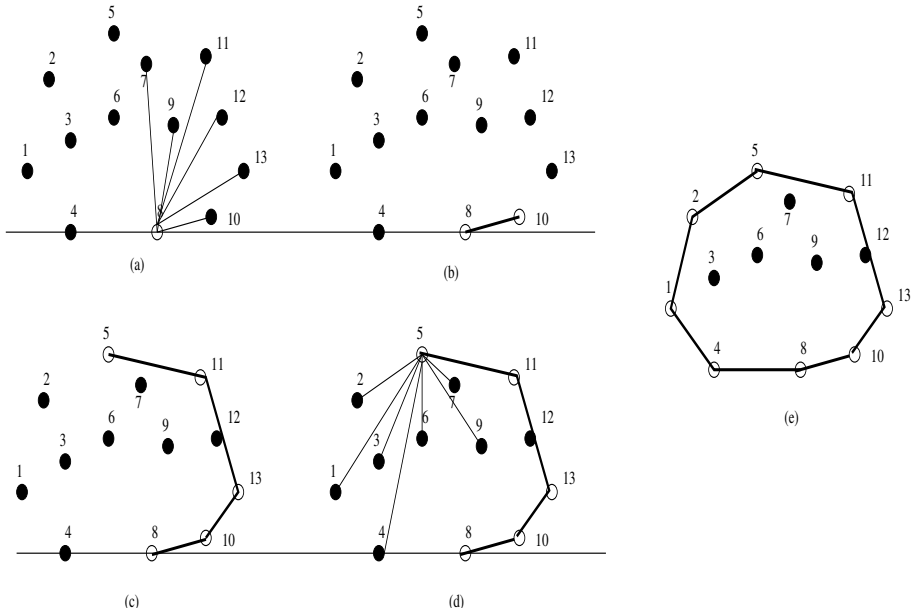


Figure 2: Jarvis march

In Figure 2, we show 13 points whose convex hull is to be determined. With reference to Figure 2, between points `4` and `8` the rightmost one with the lowest y-coordinate is point `8`. It is included in the hull and thus shown in the figure unfilled as opposed to the remaining points that are black-filled.
We then form segments (or half-lines) that start from `8` to every other point and determine the incidence angle of each one of them. Some (but not all) of them such as $\overline{8\,7}, \overline{8\,9}, \overline{8\,10}, \overline{8\,11}, \overline{8\,12}, \overline{8\,13}$ are depicted; this reduces the clutter in the figure. The segment with the lowest incidence angle is $\overline{8\,10}$, and thus `10` (or equivalently, segment $\overline{8\,10}$) is included in the convex hull as shown in Figure 2(b).
The points `13`, `11`, `5` are then similarly added to the hull (or the corresponding consecutive line segments $\overline{8\,10}, \overline{10\,13}, \overline{13\,11}, \overline{11\,5}$) as shown in Figure 2(c). In Figure 2(d), when the segments starting from `5` are formed the relevant angles are all greater than 180 degrees since `5` lies higher than all the other points and angles are formed counterclockwise. Finally in Figure 2(e) the resulting convex hull is shown.

(c) Copyright A. Gerbessiotis. All rights reserved.

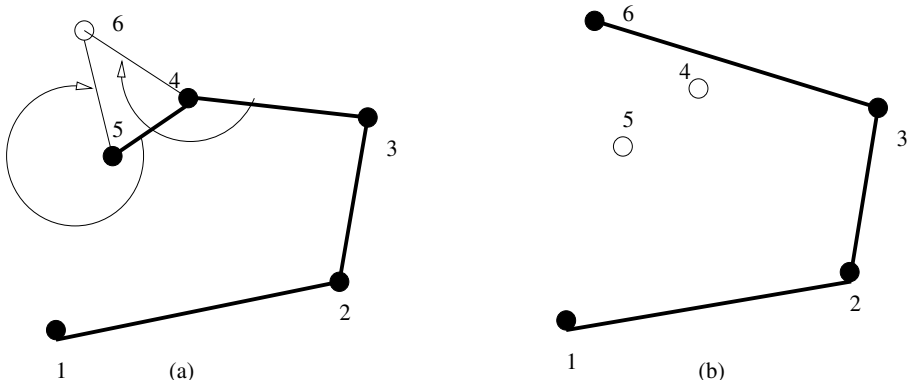


Figure 3: Graham scan

Graham scan is a worst-case running-time optimal solution to the convex hull problem. Graham's scan is an incremental algorithm. The convex hull is being built one-element at a time by using a stack. The contents of the stack at the very end form the points of the polygon P of the convex hull in counterclockwise fashion starting with the first point, which is one of the extreme points. Then points are inserted into the stack and potentially in the convex hull one by one in an incremental fashion. In order for the incremental algorithm to work, all points are sorted based on their incidence angle relative to the first point. Thus this step makes the algorithm look similar to the Jarvis march algorithm. The difference is that each step of Jarvis march reexamines each point multiple times. Graham scan discards points in the stack if a newly considered point provides a reason to disqualify stack points from being in the convex hull (they are determined to be internal to the convex hull and thus removed from the stack).

With reference to Figure 3(a), suppose that points `1,2,3,4,5` are in the stack in that order bottom-to-top. The next point that is being considered is point `6`. Point `6` causes the removal from the stack of points `5,4`. This is because point `6` lies on the other side of line-segment $\overline{5\,4}$ than `3,2,1`. Similarly for `6` and line-segment $\overline{4\,3}$. We detect this by noting that the angles formed by `4,5,6` in the former and `3,4,6` in the latter case are greater than 180 degrees, i.e. each triplet forms a non-left turn which is causing the angles to be more than 180 degrees. Thus for the shown example, Figure 3(b) shows what the deletion of points `4, 5` from the stack and the inclusion of point `6` does to the resulting implied convex hull. The stack now contains `1,2,3,6`.

(c) Copyright A. Gerbessiotis. All rights reserved.

```
        Graham-scan (S,n)
1. Let p1 be the lowest y-coordinate point; tiebreaker is left-most one
2. Let { s1, s2,  ... sn-1} be the remaining points after they are sorted
   in nondrecreasing order by polar (incidence) angle relative to p1.
3. Push(Stack,p1);
4. Push(Stack,s1);
5. Push(Stack,s2);
6. for(i=3;i< n;i++) {
7.   while angle<NEXT-to-TOP(Stack), TOP(Stack), s[i])> makes a non-left turn
8.      Pop(Stack);
9.   Push(Stack,s[i]);
10 }
11.return(Stack);
```

In Graham Scan the convex hull points appear in counterclockwise order starting with the one with the lowest y-coordinate. The sorting step in Line 2 is realized by MergeSort/HeapSort in $O(n \lg n)$. If there are more than one points with the same computed angle, all but the furthest one from p_1 are discarded from consideration. Thus the sorted list in Line 2 is of size $n-1$ or less depending on the number of discarded points. Because of Lines 3-5 there are always three points in the stack, and every point is Pushed exactly once and popped exactly once off the stack. Thus the total running time is indeed $O(n \lg n)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Closest-Pair Problem Definition. There are given n points in the plane. Find the pair q_1, q_2 such that their distance $d(q_1, q_2)$ is minimized. Distance d is (usually) defined to be the Euclidean distance.

Closest-Pair Problem in E^1. For the one-dimensional case (i.e. points in E^1) an algorithm that solves this problem is easy: sort the n points in $O(n \lg n)$ time (eg. use heapsort) and then find the consecutive pair of points that minimizes d.

For the two dimensional case we will devise a divide-and-conquer approach. Before we deal with it in two dimensions let us see how it translates into the one-dimensional space.

One-dimensional divide-and conquer approach. Partition the n points around some middle point m into two sets S_1, S_2 so that all point $p \in S_1$ are such that $p < q$, for any $q \in S_2$. Such a split can be realized by running a worst-case linear-time selection/median algorithm and each one of S_1, S_2 contains $n/2$ points (up to ± 1).

Let us assume that we have solved recursively, the closest-pair problem recursively on S_1 and S_2 and let (p_1, p_2) be the closest pair of S_1 and (q_1, q_2) the closest pair in S_2. Let $D = \min(|p_2 - p_1|, |q_2 - q_1|)$, where $|p_2 - p_1|$ is the distance of p_1, p_2.

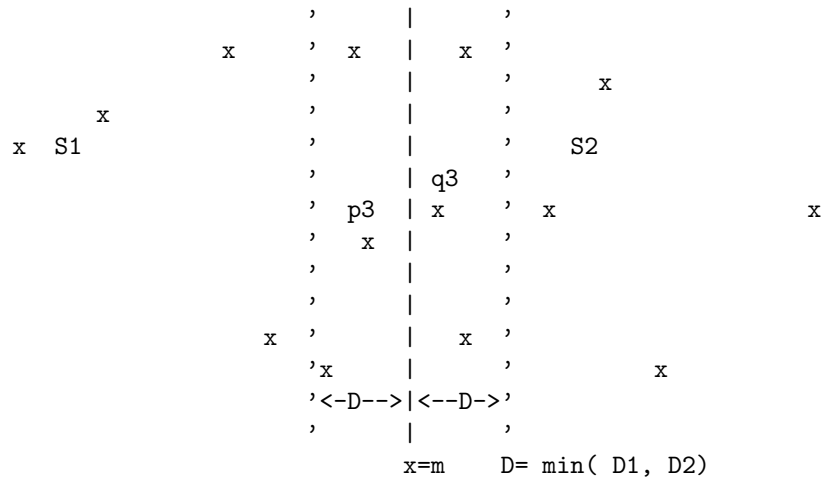
The closest pair in S can be one of (q_1, q_2) and (p_1, p_2) or of the form (p_3, q_3) where one point is in S_1 and the other in S_2. The two points p_3, q_3 however must be within D from the middle point m since otherwise the pair p_3, q_3 will be at distance more than D and can not improve upon the closest pairs already found in S_1 or S_2. For those p_3, q_3 within D from m, It suffices to find the rightmost point in S_1 and the leftmost in S_2, call them p_3, q_3 respectively and then determine whether they provide a pair whose distance is shorter than D. This completes the description of an alternative algorithm for the one-dimensional case whose running time is also given by $T(n) = 2T(n/2) + n$ i.e. it is $O(n \lg n)$. The bound is tight however.

Overview of the algorithm

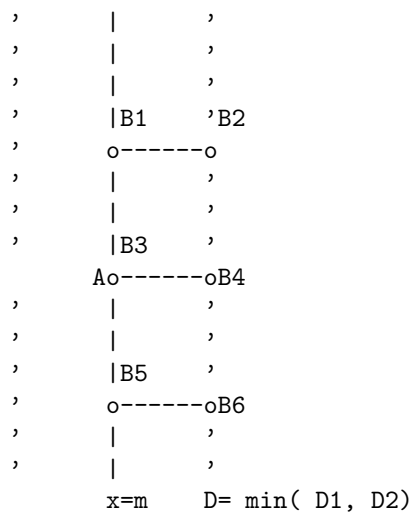
Divide-and-Conquer in E^2. In two dimensions divide-and-conquer works the same way to recursively determine (p_1, p_2) and (q_1, q_2) i.e. the clsoest-pair for each one of the two sets. The two sets S_1, S_2 are formed by using the x coordinates of the points in S and applying a selection/median to those points with key-value the x coordinate of the corresponding point. The lower half (small x coordinates) becomes S_1 and the upper (large x coordinates) becomes S_2. Let us call the dividing line $x = m$.

Having established the closest pair of sets S_1, S_2, the next step is to determine the closest pair of S. This is either the closest pair of S_1 or that of S_2 or a pair (p_3, q_3) where one point lies in S_1 and the other in S_2. Candidates for p_3 and q_3 must lie within D from the dividing line $x = m$, where $D = \min\{D_1 = d(p_1, p_2), D_2 = d(q_1, q_2)\}$. The figure below depicts the problem at this given stage.

Question 1. How many candidates do we have for p_3 and q_3?



Answer 1. I It might be the case that all points in S_1 and S_2 lie within D from $x = m$, thus comparing a point p_3 exhaustively and naively to q_3 might cost us in the worst case $O(n^2)$, which is too expensive.



Consider a point A of S_1 and let this point in the extreme fall slightly on the left of line $x = m$. There can only be at most six points of S_2 denoted by $B_1, \ldots, B_6$ in Figure 2, where B_1, B_3, B_5 are slightly on the right of $x = m$ that are at distance $D \leq D_2$ from A. Every other point is at distance more than D and can be ignored when A is considered as a p_3 candidate. For example there can be no point Z in S_2 within the rectangles $B_1B_2B_4B_3$ or $B_3B_4B_6B_5$ since then Z and of these points would at a distance less than D_2, which is impossible.

Therefore for a p_3 candidate only 6 points can be considered for a q_3 candidate. Thus every point of S_1 needs to be compared to at most 6 points of S_2 in the joining phase of the divide-and-conquer algorithm. There is only one question left to be answered.

Question 2. How do we find $B_1, \ldots, B_6$ for a given A?

Answer 2. For this we only need to project the S_2 points on the $x = m$ line, i.e. sort the S_2 points according to the y coordinates. Thus we need to have the points sorted according to their y coordinates. This is done once in $O(n \lg n)$ time at the beginning of the algorithm. The points are not resorted during divide-and-conquer. However during divide-and-conquer we select the relevant (of the sorted) points i.e. we separate the S_1 from the S_2 points in linear time into two sorted lists.

Therefore for a given point A of S_1 within a sorted sequence (based on the y coordinates of all S_1, S_2 points) we only need to determine those 6 S_2 points that are within D from A.

We wrap up this discussion in the following Algorithm.

(c) Copyright A. Gerbessiotis. All rights reserved.

```
  CLOSEST-PAIR(S)
1. Sort S according to the y coordinate.
2. Closest-Pair(S,n) //Overloaded function


  Closest-Pair(S,n)
1. Partition S into S1 , S2 around the vertical median line x=m. //Use Selection in linear time
2. D1= Closest-Pair(S1, (n+1)/2); // floor((n+1)/2) points on the left half
3. D2= Closest-Pair(S2, n-(n+1)/2); // the other on the right half
4. D= min(D1, D2);
5. P1 = points in S1 within D from x=m;
6. P2 = points in S2 within D from x=m;
7. Project P1 and P2 points on x=m //Use the sorting information of Step 1 of CLOSEST-PAIR
8. Let P1* and P2* be the sorted sequences of step 7 for P1 and P2.
9. For each point p3 in P1* scan the P2* for points q3 within distance D
   from p3. While a pointer advances in P1* the P2* pointer may oscillate
   at most 2D. Let d1 be the smallest distance of any pair thus examined.
10. d=MIN (D, d1);
11.return(d).
```

The running time of CLOSEST-PAIR is $O(n \lg n)$ for step 1 (sorting is realized with heapsort) plus the cost of Closest-Pair. The time of Closest-Pair is $T(n/2)$ for each one of S_1 and S_2 (Lines 2 and 3) plus $\Theta(n)$ for line 1 (worst-case linear time selection), $O(1)$ for line 4, and $O(n)$ for lines 4 through 10. Note that line 8 does not sort the points, it just extracts the relevant ones of the recursive call from the original sorting operation of line 1 of CLOSEST-PAIR (not of the recursive Closest-Pair). Thus the running time for Closest-Pair is $T(n) = 2T(n/2) + n$ giving $T(n) = O(n \lg n)$ and adding to this $O(n \lg n)$ for Step 1, we obtain also $O(n \lg n)$ for CLOSEST-PAIR.

(c) Copyright A. Gerbessiotis. All rights reserved.

Complexity Issues: Lower Bounds

Fundamental Lower Bound. The element distinctness problem requires $\Omega(n \lg n)$ time on an algebraic decision tree (NOTE: no floors/ceilings are allowed). Element distinctness is the problem of identifying whether n items have duplicates or not.

The Fundamental lower bound can be used to show that closest-pair requires $\Omega(n \lg n)$ time and thus our upper bound is tight: if we want to solve the element distinctness problem, run the closest pair problem and if the answer is 0 it means that there are two duplicate points, otherwise there are no duplicates.

(c) Copyright A. Gerbessiotis. All rights reserved.

An expected linear time randomized algorithm

We now present a randomized algorithm for the same problem. Its running time will be shown to be expected linear time. The expected time is due to the fact that we use hash functions to store points (hashing with chaining), and a preliminary randomization that redistributes the points randomly.

For the remainder we assume that the points have coordinates between 0 and 1; if not we can scale the coordinates and translate the points (change axes) maintain this property.

The idea behind the randomized algorithm is simple. We first randomize the points i.e. generate a random permutation of the n points. If $A_1, \ldots, A_n$ are the original points, after they are randomly permuted, they become $B_1, \ldots, B_n$ i.e. $B_i = A_{\pi(i)}$ where π is the random permutation used to "mix-up" the points. We then examine the points in turn starting from B_1 and maintain the best distance D and the pair realizing it: at the end the closest pair and its distance will be found.

The algorithm works in stages.

Stage 1. We set $D = d(B_1, B_2)$, i.e. D is the distance of the first two points (after the random permutation). The goal of a stage is to either confirm that D is the distance of the closest pair of points or to find B_i and B_j such that $d(B_i, B_j) < D$. During the course of a stage points are examined in the order $B_1, B_2, \ldots$. The number of stages depends on how lucky or unlucky we are. If the closest pair is indeed B_1, B_2 then a single stage suffices. We might in the worst case have as many as $n - 2$ stages i.e. the addition of a point always decreases the minimum distance.

Testing a Proposed Distance. Suppose the current best distance is D. We then split the defining square $[0, 1] \times [0, 1]$ in which all the points reside in N^2 smaller squares where $N = (2/D)$. Square S_{kl} has dimension $D/2 \times D/2$ and is such that

$$S_{kl} = \{(x, y) : kD/2 \leq x < (k + 1)D/2, lD/2 \leq y < (l + 1)D/2\}.$$

(c) Copyright A. Gerbessiotis. All rights reserved.

Observation 1. Two points p, q of the same square are at distance less than D.

Proof. Obvious from $D^2/4 + D^2/4 < D^2$ since for two antidiagonal points the distance is maximized.

Definition 1. Two squares S_{kl}, S_{tu} are close if $|k - t| \leq 2$ and $|u - l| \leq 2$.

Observation 2. If for two points p, q we have $d(p, q) < D$, then the squares of the two points are close.

Observation 3. For a point p belonging to square S_{kl} the squares that could contain a point q such that $d(p, q) < D$ are the 25 squares of the 5×5 square grid centered on S_{kl}.

Determining the square of a point p is easy; if $p = (x, y)$ then $k = x/(2/D)$ and $l = y/(2/D)$.

Claim 1. Finding S_{kl} to which a point p belongs takes $O(1)$ time.

Claim 2. The n points can belong to at most n squares; for a given D the number of squares is $N^2 = (2/D)^2$. A point belongs to a single square. The hash function that we will use hashes N^2 squares.

Observation 4. The hash table depends on D and N depends on D. Every time D gets smaller the hash table needs to get reorganized.

The Algorithm

```
 Closest-Pair (A1...An,n)
1. Randomly permute A1..An into B1..Bn
   // Let D be the distance of the current closest pair.
2. D= d(B1,B2);
3. CreateHashTable(D/2)
4. for(i=1;i<=n;i++) {
5.   Determine S(kl) containing Bi.
6.   Lookup 25 squares around Bi
7.   Computer distance from Bi to points of these squares
8.   If there is a point Bj with d= d(Bi,Bj) <D then
9.    DeleteHashTable(D/2)
10.   CreateHashTable(d/2)
11.   for each point Bm of B1, B2, ... , Bi {
12.      determine the d/2xd/2 square containing  Bm
13.      InsertintoHashTable(Bm,d/2)
14.   }
15.  else
16.   InsertintoHashTable(Bi,D/2)
17.}
```

Note 1. The n points can belong to at most n squares out of the potentially N^2 possible squares that subdivide $[0,1]\times[0,1]$.
Note 2. A square stores in a linked list the points that belong to it.
Note 3. Hashing with chaining can be replaced by Universal Hashing.

(c) Copyright A. Gerbessiotis. CS 667-101 : Spring 2012 . All rights reserved.

Claim 3. There are $O(n)$ distance computations, $O(n)$ Lookup operations and $O(n)$ MakeHashTable operations.

Fact 1. In the i-th iteration, if an update is performed (step 8) then $O(i)$ insertions are required.

Let X be a random variable specifying the number of Insert operations performed. We are interested in $E[X]$. Let X_i be 1 if the i-th point B_i causes the minimum distance to change and 0 otherwise. Each point is inserted once when it is first encountered (line 16) and i points need to be reinserted if the minimum distance changes at iteration i. Therefore summarizing we have

Claim 4. The total number of Insert operations performed is $n + \sum_i iX_i$.

We now bound $Pr(X_i = 1)$.

Claim 5. $Pr(X_i = 1) \leq 2/i$.

Proof of Claim 5. Consider $B_1, \ldots, B_i$. If the minimum distance among these points is due to p and q, B_i can force the minimum distance to decrease if p or q is B_i. The probability that p or q is last is at most $2/i$ (since multiple pairs might define the same distance). ■

Then we also have $E[X] = E[n + \sum_i iX_i] \leq n + 2n = 3n$.

(c) Copyright A. Gerbessiotis. All rights reserved.