Algorithms and Data Structures (II)

Gabriel Istrate

April 21, 2020

- Wanted: data structure to support dynamic sets
- Simplest form: (vectors), linked lists, stacks, queues.
- Hash tables: $O(1)$ performance under certain conditions.
- Tree-like data structures: wanted INSERT, DELETE, SEARCH in $O(\log n)$ time.
- AVL trees, red-black trees: met this bound.

Why did we want dynamic sets to start with ?

- To improve algorithms.
- Second part of today: brief respite from data structures.
- Computational geometry
- We'll see some algorithms that use stacks, red-black trees.

First part: Augmenting Data Structures (e.g. Red-Black Trees)

- What happens if a data structure doesn't support all the operations you want ?
- Augment it: modify it to support the new operations.
- Might need to add additional fields. These need to be maintained.

- What if no existing data structure fits your needs ?
- Invent a new one, or ...
- More realistic (in practice): slightly modify a "standard" data structure to support more operations.
- Done by storing extra information in it
- Not always straightforward: new information **must be updated and maintained** by D.S. operations.

Example: two data structures obtained by modifying red-black trees

- First data structure: supports order statistics queries on a dynamic set.
  - $\boxed{\text{Find } i\text{'th number in a set or the rank of an element}}$.
- Second data structure: maintain a set of intervals (e.g. time intervals).
- Plus: a general result about augmenting Data Structures.

- Order statistic tree: red-black tree with one extra field per node: size of the subtree rooted at that node.
- Thus fields: $key, color, p, left, right, size$.
- $size[nil[T]] = 0$.
- $size[x] = size[left[x]] + size[right[x]] + 1$.
- Supports $OS - SELECT(x, i)$: return i'th smallest element in the tree rooted at x. $O(\log n)$ time.
- Supports $OS - RANK(T, x)$: return the rank of x in the tree T. $O(\log n)$ time.

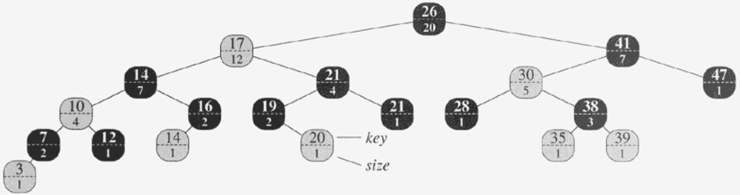


Figure 14.1 An order-statistic tree, which is an augmented red-black tree. Shaded nodes are red, and darkened nodes are black. In addition to its usual fields, each node x has a field $size[x]$, which is the number of nodes in the subtree rooted at x.

- If $i = size[left(x)] + 1$ then (by *BST* property) node x is the i'th element. Return x.
- If $i \leq size[left(x)]$ then node is in $left[x]$. i'th element. Call procedure recursively.
- If $i > size[left(x)] + 1$ then node is in $right[x]$. $i - size[left(x)]$'th element. Call procedure recursively.
- Running time: proportional to the height of the tree: $O(\log n)$.

Selecting i'th element

```
OS-SELECT(x, i)
r ← size[left[x]]+1
if i = r
    then return x
elseif i < r
    then return OS-SELECT(left[x], i)
else return OS-SELECT(right[x], i − r)
```

Rank of an element

```
OS-Rank(T, x)
r ← size[left[x]] + 1
y ← x
while y ≠ root[T]
      do if y = right[p[y]]
            then r ← r + size[left[p[y]]] + 1
         y ← p[y]
return r
```

- Perform inorder traversal.
- Return rank of node x in this traversal.
- Move pointer y from x up towards $root(T)$.
- Maintains the following invariant: at the start of each iteration of the while loop, r is the rank of $key[x]$ in the subtree rooted at y.
- If y is a right child, add the size of its left child to the count.
- Each iteration: $O(1)$ time. y goes up the tree, time complexity $O(\log n)$.

- During LEFT/RIGHT rotations.
- INSERTION. First phase: go from the root to the frontier, inserting the new node as the child of an existing node. new node gets size of 1. Each node from x to the path: size increases by 1. $O(\log n)$.
- Second phase: go up the tree, changing colors, and maintaining the red-black property by rotations.
- Second phase: changes via LEFT/RIGHT rotations.
- LEFT-ROTATE: add lines
  - $size[y] \leftarrow size[x]$.
  - $size[x] \leftarrow size[left[x]] + size[right[x]] + 1$.
- to rotation pseudocode.
- RIGHT-ROTATE: symmetric.

Maintaining *size* during rotations.

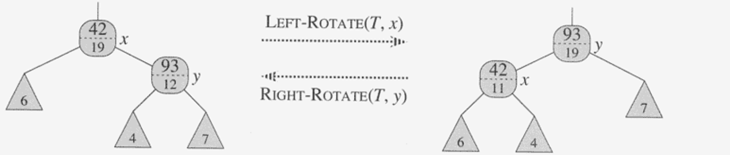


Figure 14.2 Updating subtree sizes during rotations. The link around which the rotation is performed is incident on the two nodes whose *size* fields need to be updated. The updates are local, requiring only the *size* information stored in x, y, and the roots of the subtrees shown as triangles.

Maintaining subtree sizes: Deletion.

- DELETION: two phases.
- First phase: delete node. Update tree size on the path from the node to the top. Decrement by 1 for each node.
- Rotations: as for insertion.

- Four steps:
- 1. Choose underlying data structure.
- 2. Determine additional information to be maintained.
- 3. Verify that additional information can be maintained in the D.S. operations.
- 4. develop new operations required by new fields.

1. Choose red-black trees. Clue: supports other dynamic set operations on total order: MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR.
2. We didn't need field size to implement OS-SELECT, OS-RANK, but then operations wouldn't run in $O(\log n)$ time. Additional information to be maintained: sometimes pointer rather than data.
3. Ideally only a few elements need to be updated to maintain D.S. E.g. if we simply stored in each node it rank in the tree then OS-SELECT and OS-UPDATE would be efficient but inserting a smallest node causes changes in the whole tree.
4. Developed OS-SELECT, OS-RANK. Occasionally, instead of new operations, speed-up old ones.

Theorem

Let f be a field that augments a RB tree of n nodes, and suppose the contents of f for node x can be computed in $O(1)$ using only information in node x, $left[x]$ and $right[x]$, including $f[left[x]]$ and $f[right[x]]$. Then we can maintain the values of f in all nodes in T during insertion and deletion without asymptotically affecting $O(\log n)$ performance.

Proof idea: change in field f at a node x propagates only to ancestors of x in the tree.

- closed interval: $[t_1, t_2]$. Also open, half-open intervals.
- $i = [t_1, t_2]$. $low[i] = t_1$, $high[i] = t_2$.
- i and i' overlap if $i \bigcap i' \neq \emptyset$. That is $low[i] \leq high[i']$ and $low[i'] \leq high[i]$.
- Want: Data structure representing a dynamic set of intervals.
- Must support the following operations:
- $INTERVAL - INSERT(T, x)$: adds element x, whose *int* field contains an interval.
- $INTERVAL - DELETE(T, x)$: removes element x from T.
- $INTERVAL - SEARCH(T, i)$: return pointer to an element x such that $int[x]$ overlaps i, or *nil* if no such element found.

- Any two intervals satisfy interval trichotomy: three alternatives:
  1. i and i' overlap.
  2. i is to the left of i' ($high[i] < low[i']$).
  3. i is to the right of i'.($low[i] > high[i']$).

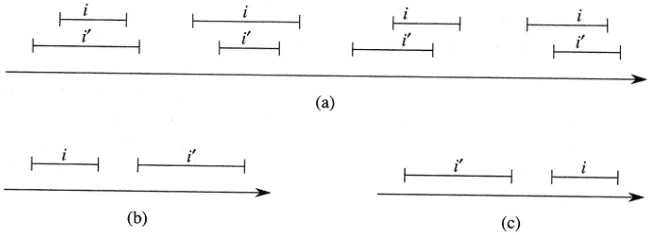


Figure 14.3 The interval trichotomy for two closed intervals i and i'. **(a)** If i and i' overlap, there are four situations; in each, $low[i] \leq high[i']$ and $low[i'] \leq high[i]$. **(b)** The intervals do not overlap, and $high[i] < low[i']$. **(c)** The intervals do not overlap, and $high[i'] < low[i]$.

1. Possible clue: intervals (partial) ordering. Might try to modify a total order. Then red-black tree. Each node x stores an interval $int[x]$.
   - $key[x] = low[int[x]]$.
2. Additional info: $max[x]$, the maximum value of any endpoint of an interval stored in the subtree rooted at x.
3. Maintain info: $max[x] = \max(high[int[x]], max[left[x]], max[right[x]])$.
4. By applying previous theorem: insertion/deletion $O(\log n)$ while maintaining $max[x]$.

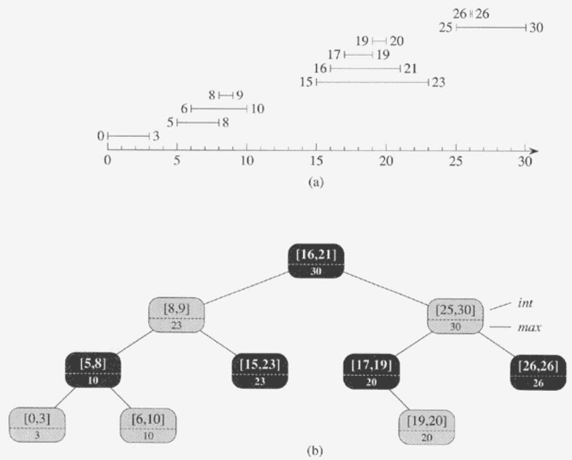


Figure 14.4 An interval tree. **(a)** A set of 10 intervals, shown sorted bottom to top by left endpoint. **(b)** The interval tree that represents them. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

- finds a node in tree T whose interval overlaps interval i, returns sentinel node $nil[T]$ if no overlapping interval found.
- Search starts at the root and proceeds downwards.
- Chooses *left* or *right* subtree based on the maximum element in the left subtree of x.
- If $max[left[x]]$ is $\geq low[i]$ (of course, $left[x] \neq nil[T]$) go left.
- otherwise go right.
- takes $O(\log n)$ time since each basic loop takes $O(1)$ time and the height of the RB tree is $O(\log n)$.

INTERVAL-SEARCH

```
INTERVAL-SEARCH(T, i)
x ← root[T]
while x ≠ nil[T] and i does not overlap int[x]
      do if left[x] ≠ nil[T] and max[left[x]] ≥ low[i]
            then x ← left[x]
            else x ← right[x]
return x
```

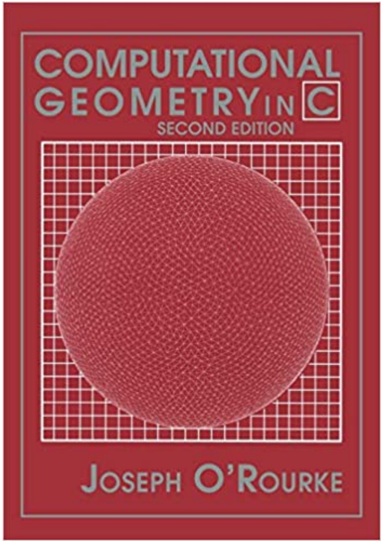
- Why is it enough to examine a single path ?
- Idea: search proceeds in a "safe direction".
- INVARIANT: If tree T contains an interval that overlaps i then there is such an interval in the subtree rooted at x.
- Initialization: clearly satisfied, $x = root[T]$.
- Either line 4 or line 5 executed.
- Line 5 executed: because $left[x] = nil[T]$ or $max[left[x]] < low[i]$. The subtree rooted at left[x] does not contain any interval that overlaps i.
- If such an interval is found in T, it must be in $right[x]$.

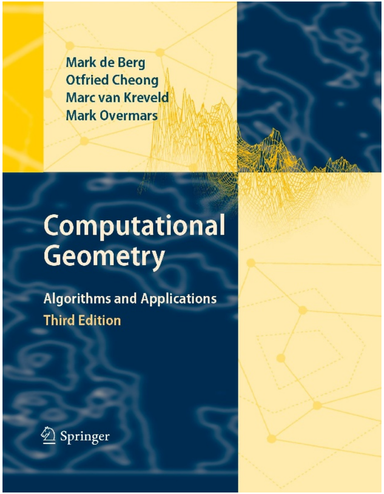
- Line 4 executed: contrapositive of loop invariant holds.
- If there is no such an interval in the subtree rooted at $left[x]$ then there is no such interval in tree T.
- Since line 4 executed $max[left[x]] \geq low[i]$. There exists i' with $high[i'] = max[left[x]] \geq low[i]$.
- i and i' do not overlap, by assumption. By trichotomy $high[i] < low[i']$.
- i'' interval in $right[x]$. Intervals keyed on the low endpoints.
- $high[i] < low[i'] \leq low[i'']$.
- Conclusion: no interval in $right[x]$ (and thus in T) overlaps i.

- Studies algorithms for geometric problems.
- Applications: computer graphics, robotics, VLSI, CAD.
- More applications: protein folding, molecular modeling, GIS.
- Huge area ! Only a sampler.
- Scientific conference: SOCG
- Software: CGAL.

Caution

- The biggest "enemy" to algorithms in computational geometry: degeneracy.
- Three points are collinear, three lines intersect at the same point, etc.
- Algorithms need patching to deal with degenerate situations.
- In the interest of teaching: Ignore it.


COMPUTATIONAL
GEOMETRY IN C
SECOND EDITION
JOSEPH O'ROURKE


Mark de Berg
Otfried Cheong
Marc van Kreveld
Mark Overmars
Computational
Geometry
Algorithms and Applications
Third Edition
Springer

- Input: set of points $\{p_i\}$, $p_i = (x_i, y_i)$. Example: polygon $P = (p_0, p_1, \ldots, p_n)$.
- Given $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, convex combination: any point $p_3 = (x_3, y_3)$ such that $x_3 = \lambda x_1 + (1 - \lambda)x_2, \lambda \in [0, 1]$, similarly $y_3 = \lambda y_1 + (1 - \lambda)y_2$.

1. Given two directed segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_0p_2}$, is $\overrightarrow{p_0p_1}$ clockwise from $\overrightarrow{p_0p_2}$ with respect to their common endpoint p_0?
2. Given two line segments $\overline{p_1p_2}$ and $\overline{p_2p_3}$, if we traverse $\overline{p_1p_2}$ and then $\overline{p_2p_3}$, do we make a left turn at point p_2?
3. Do line segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ intersect?

$$
\begin{aligned}
p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\
&= x_1 y_2 - x_2 y_1 \\
&= -p_2 \times p_1 \, .
\end{aligned}
$$

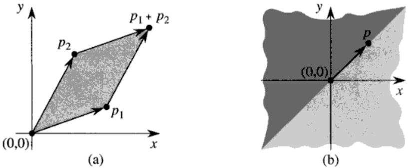


Figure 33.1 **(a)** The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. **(b)** The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.

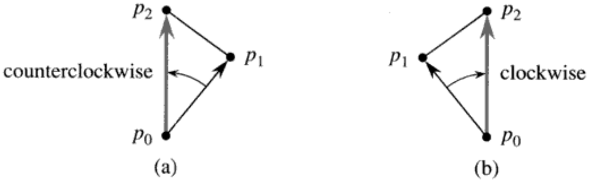


Figure 33.2 Using the cross product to determine how consecutive line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$ turn at point p_1. We check whether the directed segment $\overrightarrow{p_0 p_2}$ is clockwise or counterclockwise relative to the directed segment $\overrightarrow{p_0 p_1}$. **(a)** If counterclockwise, the points make a left turn. **(b)** If clockwise, they make a right turn.

Procedures DIRECTION and ON-SEGMENT

DIRECTION(p_i, p_j, p_k)

return $(p_k - p_i) \times (p_j - p_i)$

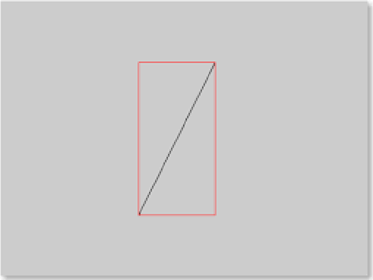
ON-SEGMENT(p_i, p_j, p_k)

if $\min(x_i, x_j) \le x_k \le \max(x_i, x_j)$ and $\min(y_i, y_j) \le y_k \le \max(y_i, y_j)$

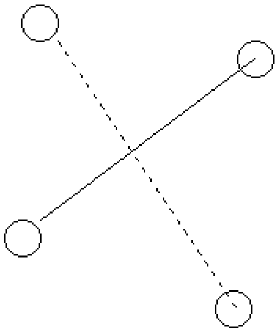
  then return TRUE

  else return FALSE

- QUICK REJECT: two segments cannot intersect if their BOUNDING BOXES don't.
- Smallest rectangle containing the segment with sides parallel to the xy axes.
- Bounding box of $\overline{p_1p_2}$, $p_i = (x_i, y_i)$ is rectangle with corners $(min(x_1, x_2), min(y_1, y_2)$, $(min(x_1, x_2), max(y_1, y_2)$ $(max(x_1, x_2), max(y_1, y_2)$ and $(max(x_1, x_2), min(y_1, y_2)$.



- Second stage: check whether each segment "straddles" the other.
- A segment $\overline{p_1 p_2}$ straddles a line if point p_1 lies on one side of the line and point p_2 lies on the other side. If p_1 or p_2 lies on the line, then we say that the segment straddles the line. Two line segments intersect if and only if they pass the quick rejection test and each segment straddles the line containing the other.



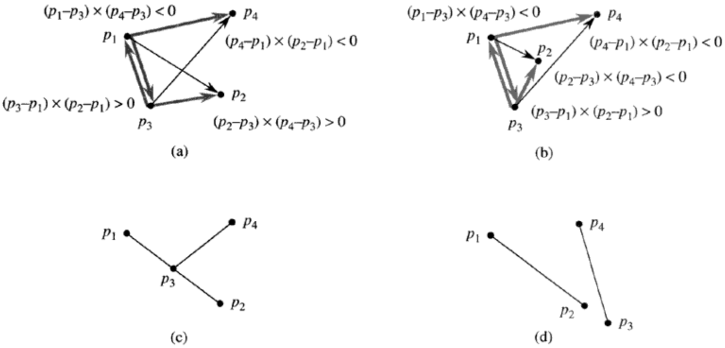


Figure 33.3 Cases in the procedure SEGMENTS-INTERSECT. **(a)** The segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ straddle each other's lines. Because $\overline{p_3p_4}$ straddles the line containing $\overline{p_1p_2}$, the signs of the cross products $(p_3-p_1)\times(p_2-p_1)$ and $(p_4-p_1)\times(p_2-p_1)$ differ. Because $\overline{p_1p_2}$ straddles the line containing $\overline{p_3p_4}$, the signs of the cross products $(p_1-p_3)\times(p_4-p_3)$ and $(p_2-p_3)\times(p_4-p_3)$ differ. **(b)** Segment $\overline{p_3p_4}$ straddles the line containing $\overline{p_1p_2}$, but $\overline{p_1p_2}$ does not straddle the line containing $\overline{p_3p_4}$. The signs of the cross products $(p_1-p_3)\times(p_4-p_3)$ and $(p_2-p_3)\times(p_4-p_3)$ are the same. **(c)** Point p_3 is collinear with $\overline{p_1p_2}$ and is between p_1 and p_2. **(d)** Point p_3 is collinear with $\overline{p_1p_2}$, but it is not between p_1 and p_2. The segments do not intersect.

Testing whether two segments intersect

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
d1 ← DIRECTION(p3, p4, p1)
d2 ← DIRECTION(p3, p4, p2)
d3 ← DIRECTION(p1, p2, p3)
d4 ← DIRECTION(p1, p2, p4)
if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
        ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
   then return TRUE
elseif d1 = 0 and ON-SEGMENT(p3, p4, p1)
   then return TRUE
elseif d2 = 0 and ON-SEGMENT(p3, p4, p2)
   then return TRUE
elseif d3 = 0 and ON-SEGMENT(p1, p2, p3)
   then return TRUE
elseif d4 = 0 and ON-SEGMENT(p1, p2, p4)
   then return TRUE
else return FALSE
```

Testing whether any two segments intersect

- Given: n segments $v_1, \ldots, v_n$.
- To test: do any two segments intersect ?
- Uses technique called sweeping.
- Running time: $O(n \log n)$. Naive algorithm $O(n^2)$.
- SWEEPING: an imaginary vertical sweep line passes through the given set of geometric objects, usually from left to right. The spatial dimension that the sweep line moves across, in this case the x-dimension, is treated as a dimension of time.
- Provides method for ordering geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of relationships among them.
- line-segment-intersection algorithm: considers all line-segment endpoints in left-to-right order and checks for an intersection each time it encounters an endpoint.

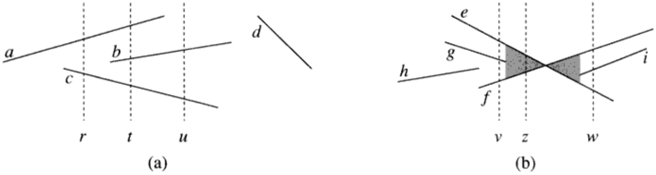


Figure 33.4 The ordering among line segments at various vertical sweep lines. **(a)** We have $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$, and $b >_u c$. Segment d is comparable with no other segment shown. **(b)** When segments e and f intersect, their orders are reversed: we have $e >_v f$ but $f >_w e$. Any sweep line (such as z) that passes through the shaded region has e and f consecutive in its total order.

- Sweeping algorithms: maintain two sets of data.
- sweep-line status: gives the relationships among objects intersected by the sweep line.
- event-point schedule: sequence of x-coordinates, ordered from left to right, that defines the halting positions of the sweep line.
- Call each such halting position an event point. Changes to the sweep-line status occur only at event points.
- Sweep-line status: total order T.
- $INSERT(T,s)$, $DELETE(T,s)$.
- $ABOVE(T,s)$: return segment above s in T.
- $BELOW(T,s)$: return segment below s in T.
- We can perform each of the above operations in $O(\log n)$ time using red-black trees.

Algorithm

```
ANY-SEGMENTS-INTERSECT(S)
T ← Ø
sort the endpoints of the segments in S from left to right,
        breaking ties by putting left endpoints before right endpoints
        and breaking further ties by putting points with lower
        y-coordinates first
for each point p in the sorted list of endpoints
    do if p is the left endpoint of a segment s
         then INSERT(T, s)
              if (ABOVE(T, s) exists and intersects s)
                    or (BELOW(T, s) exists and intersects s)
                then return TRUE
       if p is the right endpoint of a segment s
         then if both ABOVE(T, s) and BELOW(T, s) exist
                    and ABOVE(T, s) intersects BELOW(T, s)
                then return TRUE
              DELETE(T, s)
return FALSE
```

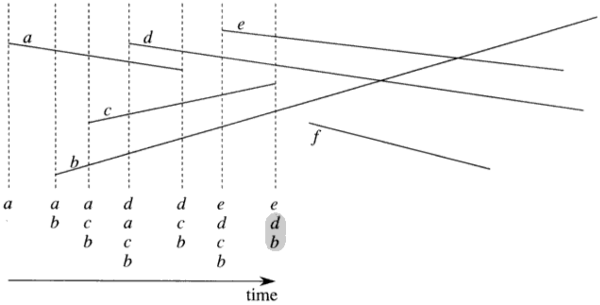


Figure 33.5 The execution of ANY-SEGMENTS-INTERSECT. Each dashed line is the sweep line at an event point, and the ordering of segment names below each sweep line is the total order T at the end of the **for** loop in which the corresponding event point is processed. The intersection of segments d and b is found when segment c is deleted.

- Can only fail by not reporting intersecting segments.
- p = leftmost intersection point, breaking ties by choosing the one with the lowest y-coordinate. a and b = the segments that intersect at p.
- No intersections occur to the left of $p \Rightarrow$ the order given by T is correct at all points to the left of p.
- no three segments intersect at the same point $\Rightarrow$ there exists a sweep line z at which a and b become consecutive in the total order.
- z is to the left of p or goes through p.
- There exists segment endpoint q on z that is the event point at which a and b become consecutive.
- If p is on z, then $q = p$. If p is not on z, then q is to the left of p. In either case, the order given by T is correct just before q is processed.

Algorithm: correctness/performance

- Either a or b is inserted into T, and the other segment is above or below it in the total order. Lines 4-7 detect this case.
- Segments a and b are already in T, and a segment between them in the total order is deleted, making a and b become consecutive. Lines 8-11.
- In either case, the intersection p is found.
- $2n$ insert/delete/tests. Taking $O(\log n)$ time.

- Convex hull of a set of points: smallest convex polygon that contains the set of points.
- place elastic rubber band around set of points and let it shrink.
- Two algorithms: Graham's Scan $O(n \log n)$.
- Jarvis's March $O(n \cdot h)$, h the number of points on the convex hull.
- Other algorithms:
- Incremental: points sorted from left to right forming sequence $p_1, \ldots, p_n$. At stage i add p_i to convex hull $CH(p_1, \ldots, p_{i-1})$, forming $CH(p_1, \ldots, p_i)$.
- Divide-and-conquer: divide into leftmost n/2 points and rightmost n/2 points. Compute convex hulls and combine them.
- Prune-and-search method.

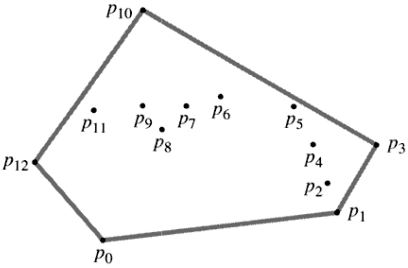


Figure 33.6 A set of points $Q = \{p_0, p_1, \ldots, p_{12}\}$ with its convex hull $\mathrm{CH}(Q)$ in gray.

- Maintains a stack S of candidate points.
- Each point of Q is pushed onto the stack.
- Points not in $CH(Q)$ eventually popped from the stack.
- $TOP(S)$, $NEXT-TO-TOP(S)$: stack functions, do not change its contents.
- Stack returned by the algorithm: points of $CH(Q)$ in counterclockwise order.

Convex hull algorithm

```
GRAHAM-SCAN(Q)
let p0 be the point in Q with the minimum y-coordinate,
        or the leftmost such point in case of a tie
let ⟨p1, p2, ..., pm⟩ be the remaining points in Q,
        sorted by polar angle in counterclockwise order around p0
        (if more than one point has the same angle, remove all but
        the one that is farthest from p0)
PUSH(p0, S)
PUSH(p1, S)
PUSH(p2, S)
for i ← 3 to m
    do while the angle formed by points NEXT-TO-TOP(S), TOP(S),
                and pi makes a nonleft turn
            do POP(S)
        PUSH(pi, S)
return S
```

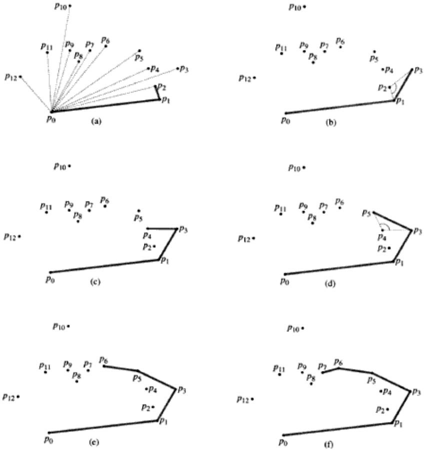
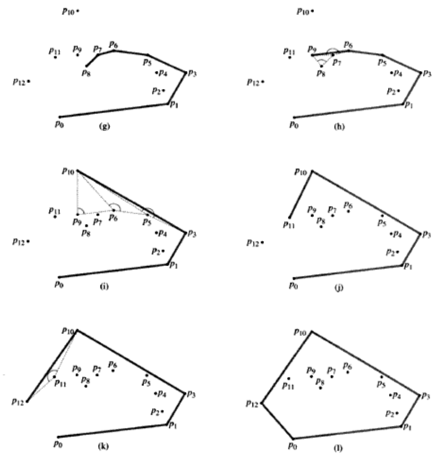


Figure 33.7 The execution of GRAHAM-SCAN on the set Q of Figure 33.6. The current convex hull contained in stack S is shown in gray at each step. **(a)** The sequence $\langle p_1, p_2, \ldots, p_{12} \rangle$ of points numbered in order of increasing polar angle relative to p_0, and the initial stack S containing p_0, p_1, and p_2. **(b)–(k)** Stack S after each iteration of the **for** loop of lines 6–9. Dashed lines show nonleft turns, which cause points to be popped from the stack. In part (h), for example, the right turn at angle $\angle p_7 p_8 p_9$ causes p_8 to be popped, and then the right turn at angle $\angle p_6 p_7 p_9$ causes p_7 to be popped. **(l)** The convex hull returned by the procedure, which matches that of Figure 33.6.

Graham's Scan:Example



Graham's Scan: Correctness and Performance

- Invariant: at the beginning of each iteration of the for loop stack S contains (from bottom to top) exactly the vertices of $CH(Q_{i-1})$ in counterclockwise order.
- Line 1: $\theta(n)$ time.
- Sorting $\theta(n \log n)$ time.
- Testing for left/right turn: vector product $\theta(1)$ time.
- The rest of the algorithm $O(n)$ time.

Graham's Scan: Correctness

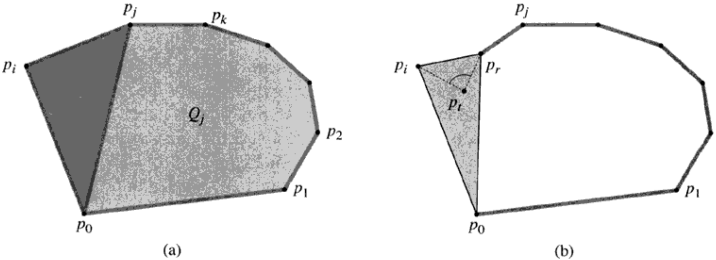


Figure 33.8 The proof of correctness of GRAHAM-SCAN. **(a)** Because p_i's polar angle relative to p_0 is greater than p_j's polar angle, and because the angle $\angle p_k p_j p_i$ makes a left turn, adding p_i to $\mathrm{CH}(Q_j)$ gives exactly the vertices of $\mathrm{CH}(Q_j \cup \{p_i\})$. **(b)** If the angle $\angle p_r p_t p_i$ makes a nonleft turn, then p_t is either in the interior of the triangle formed by p_0, p_r, and p_i or on a side of the triangle, and it cannot be a vertex of $\mathrm{CH}(Q_i)$.

- uses a technique known as gift wrapping.
- Simulates wrapping a piece of paper around set Q.
- Start at the same point p_0 as in Graham's scan.
- Pull the paper to the right, then higher until it touches a point. This point is a vertex in the convex hull. Continue this way until we come back to p_0.
- Formally: start at p_0. Choose p_1 as the point with the smallest polar angle from p_0. Choose p_2 as the point with the smallest polar angle from $p_1 \ldots$
- $\ldots$ until we reached the highest point p_k.
- We have constructed the right chain.
- Construct the left chain by starting from p_k and measuring polar angles with respect to the negative x-axis.

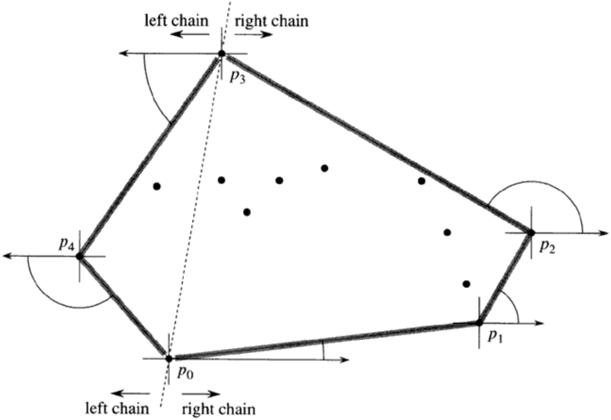


Figure 33.9 The operation of Jarvis's march. The first vertex chosen is the lowest point p_0. The next vertex, p_1, has the smallest polar angle of any point with respect to p_0. Then, p_2 has the smallest polar angle with respect to p_1. The right chain goes as high as the highest point p_3. Then, the left chain is constructed by finding smallest polar angles with respect to the negative x-axis.

- W.r.t. euclidean distance.
- Brute force: $\theta(n^2)$.
- Divide and conquer: $O(n \log n)$.
- Each iteration: subset $P \subseteq Q$, arrays X and Y.
- Points in X are sorted in increasing order of their x coordinates.
- Points in Y are sorted in increasing order of their y coordinates.
- To maintain upper bound cannot afford to sort in each iteration.
- $|P| \leq 3$: brute force. Otherwise recursive divide-and-conquer.
- **Divide:** Find a vertical line l that bisects set P into two sets P_L and P_R such that $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, all points of P_L to the left, all points of P_R to the right.
- X_L: subarray that contains point of P_L, X_R: subarray that contains point of P_R.
- Similarly for Y.

- **Conquer**. Recursive calls: P_L, X_L, Y_L and P_R, X_R, Y_R. Returns smallest distances δ_L and δ_R.
- **Combine.** $\delta = \min\{\delta_L, \delta_R\}$.
- Have to test whether some point in P_L is at distance $< \delta$ from some point in P_R.
- Both such points, if they exist, are within the 2δ-wide strip around l.
- Create an array Y' which is Y with all points not in the 2δ-wide strip around l removed, sorted by y-coordinate.
- For each point p in Y' try to find points in Y' at distance less than δ.
- Only the 7 points that follow p need to be considered.
- Compute smallest such distance δ'. If $\delta' < \delta$ we found a better pair. Otherwise δ is the smallest distance.
- Correctness, implementation nontrivial.

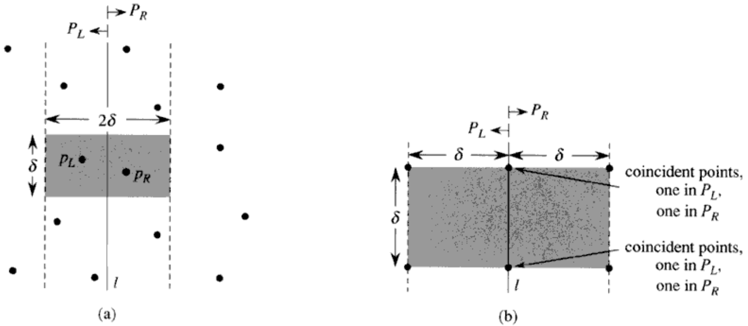


Figure 33.11 Key concepts in the proof that the closest-pair algorithm needs to check only 7 points following each point in the array Y'. **(a)** If $p_L \in P_L$ and $p_R \in P_R$ are less than δ units apart, they must reside within a $\delta \times 2\delta$ rectangle centered at line l. **(b)** How 4 points that are pairwise at least δ units apart can all reside within a $\delta \times \delta$ square. On the left are 4 points in P_L, and on the right are 4 points in P_R. There can be 8 points in the $\delta \times 2\delta$ rectangle if the points shown on line l are actually pairs of coincident points with one point in P_L and one in P_R.

- For each point: Consider the $\delta \times 2\delta$ rectangle centered at line l.
- At most 8 points within this rectangle.
- Assuming δ_L lower than δ_R, it follows that δ_R among the next 7 points following δ_L.
- $O(n \log n)$ bound from recurrence $T(n) = 2T(n/2) + O(n)$.
- Main difficulty: making sure that X_L, X_R, Y_L, Y_R, Y' sorted by appropriate coordinate.
- Key observation: in each call we wish to form a sorted subset of a sorted array.
- Splitting the array into two halves.
- Can be viewed as the inverse of the operation *MERGE* in *MERGESORT*.
- How to get sorted arrays in the first place ? presort. $\theta(n \log n)$.

$length[Y_L] = length[Y_R] = 0$;
```
for i = 1 to length[Y]
  if (Y[i] ∈ P_L)
  {
   length[Y_L]++;
   Y_L[length[Y_L]] = Y[i];
  }
  else
  {
   length[Y_R]++;
   Y_R[length[Y_R]] = Y[i];
  }
}
```