COMPUTATIONAL GEOMETRY
Some Suggested Optional Projects

By **April 17, 2020**, you should notify me of which project you are doing and submit by email a brief (about 1-page) description of the project plan: the project/problem, what language you will use, etc. Projects should be done individually.

Project reports are due **May 7, 2020**; on Tuesday, **May 5, 2020**, there will be in-class presentations and demos, done via Zoom. In addition to submitting a report, you must submit the bundle of software. The report can be brief but should state clearly the problem, the approach, and give basic information about compiling/running the software, and report relevant experimental experience. Submission should be by email. If you do the project in Java, please also create an applet and webpage, so that particularly educational projects can be easily featured in future classes.

Note that some projects may lead to publishable research or to more extensive projects (e.g., masters projects (CSE 523/524)), depending on how successful the investigation is.

In general, software should be written in standard language (most common are Python, Java, C, C++,; some have used MATLAB or R). It should be easily compilable and executable under Unix (e.g., a PC running Linux) or Windows. Please document your code! Also, provide a README, and bundle all files in one directory. You may find that C or Java is the most convenient: you are allowed to use C code fragments provided in the O'Rourke textbook (and available on the web, both in C and in Java – follow pointers to O'Rourke's home page, from our course home page). You may be asked to give a short demonstration of your code to me on your own computer. Try to handle degenerate cases and to make your implementation as robust as you can.

You will find it most convenient to have a graphical display to output and "animate" your algorithm. You may also want to be able to mouse in data. Remember that you will be giving a demo to the class, so there should be visual appeal so that classmates can understand what you did.

If you are proficient in C++, you may find it useful to develop your project within CGAL (Computational Geometry Algorithms Library).

(1). As we know, Fisk's proof method gives us a means of placing guards at vertices of a simple polygon. The set of guards it produces need not be optimal, of course. Further, the set of guards produced depends on the particular triangulation used – some triangulations $\mathcal{T}$ of simple polygon P lead to a much smaller set of guards, using the Fisk method, than other triangulations. (We know there can be *lots* of different triangulations for some polygons.)

Implement a dynamic programming algorithm to find the best triangulation of P, yielding the fewest vertex guards possible using the Fisk method.

(2). *Crowd size estimation: The deer population problem.* Sensors are placed throughout the county in order to record sightings of deer. This results in a set of time-stamped points $S = \{(x_1, y_1, t_1), \ldots, (x_n, y_n, t_n)\}$, with (x_i, y_i, t_i) meaning that a deer was at position (x_i, y_i) at time t_i. We know that deer are fast, but their maximum speed is $v_{max} = 1$. Our goal is to estimate the number of deer actually present in the county, realizing that many of the sightings may correspond to the very same deer (running and moving around between locations). You are to compute estimates of the deer population. The minimum number, N_{min}, of deer is the minimum number of directed paths needed to cover the set S, in the *accessibility graph*, which links (x_i, y_i, t_i) to (x_j, y_j, t_j) if and only if it is possible for a deer (running at speed at most 1) to be present at both sightings. Devise and implement a strategy to compute N_{min} (or an estimate thereof) and experiment on data that you simulate (e.g., using k random trajectories in the plane). You are welcome to be creative

and devise heuristics, and potentially compare them (e.g., if more than one person is on the projoect team). Try to use CG concepts to process data efficiently, as the number $n = |S|$ could be quite huge in practice.

Possibly relevant references include:
`https://en.wikipedia.org/wiki/Path_cover`,
`http://www.austms.org.au/Publ/Jamsb/V44P2/1761.html`, and
a 2002 paper by D. S. Franzblau and A. Raychaudhuri, "Optimal Hamiltonian completions and path covers for trees, and a reduction to maximum flow".

(3). *Max/min area polygonalizations.* Given a set S of n points in the plane, compute a simple polygonalization of S (a simple polygon whose vertex set is precisely the set S) that has maximum or minimum area among all polygonalizations of S. (Every set S of n points in the plane has at least one polygonalization. The number of different polygonalizations is finite (though possibly exponentially large) for any finite point set S.) The optimization problems, both for maximization or for minimization, are known to be NP-hard; see S. P. Fekete, On Simple Polygonalizations with Optimal Area, Discrete and Computational Geometry 23:73-110 (2000). Thus, the goal of this project is to devise and implement heuristics encourages implementations of solutions that are based on heuristics, on methods of combinatorial optimization (e.g., branch and bound), guided search, etc.

There is an optimization challenge for this project, associated with the annual computational geometry meeting; see http://cgshop.ibr.cs.tu-bs.de/. There, we have posted data sets for the competition, etc.

(4). *Computing the Niceness of a Polygonal Shape.* There are various notions of quantifying how "nice" or how "fat" a simple polygon P is. A "nicest" polygon might be a regular n-gon, which most closely approximates a circular disk. This project seeks to implement some precise metrics for niceness, and compare them on simple polygons (possibly moused in by a user or read in from a file, etc).

To make it simple and discrete, I propose that you discretize the boundary of P with discrete points p_i (additional vertices), with a prescribed spacing, δ. (That is, for edges of P longer than δ, add vertices along the edge so that the new subsegments are of length at most δ, for a user-specified parameter δ.)

Then, consider discrete choices of radii $r = \rho, 2\rho, 3\rho, \ldots$ for disks, $B(p_i, r)$, centered at the discrete boundary points p_i, for each radius r. The (discrete) *fatness* of P is given by the smallest value of the ratio $area(B(p_i, r) \cap P)/area(B(p_i, r)$, over all choices of p_i and $r = \rho, 2\rho, 3\rho, \ldots$ such that P is not contained fully inside $B(p_i, r)$.

Implement and experiment with this fatness measure. It may be possible to use the algorithm to assist a project at Harvard on Gerrymandering, where the goal is to quantify how "compact" polygonal election districts are. (So I hope that at least a couple of people choose to do this project! We can discuss further.)

(Theoretically, we are interested in finding an algorithm to compute the "exact" fatness of P (which allows disks centered anywhere inside P, of any radius such that the disk does not contain all of P), without resorting to the simple discretization. Or, can we compute a provable approximation to the exact fatness?)

(5). "Tether TSP": Given a set $S = \{p_1, p_2, \ldots, p_n\}$ of n points in the plane, our goal is to move two "tethered" robots, R_1 and R_2, in the plane, so that every point is visited by at least one robot, and at all moments in time the two robots are at distance at most L from each other (there is a "tether" cord linking them). The goal is to complete the visitations as soon as possible, for a given maximum speed of each robot (possibly different speeds). Devise and experiment with a heuristic (e.g., based on using an MST, and local shortcutting, etc).

(6). *The Magnet and Bead.* Develop an applet or an app to explore the following problem. A set S of horizontal or vertical line segments forms a connected union of "road segments". Let G denote this graph/network. A metal "bead" moves along the network G. It is attracted by a magnet, M, that you, the user, control. Assume that M moves on the boundary of a rectangle R, which surrounds G (and is not in contact with it). The bead starts at a position s on the network G, and the magnet starts at position M_0. The bead is attracted to the magnet, sliding along edges of G to get as close as possible, until it is "stuck" (at a local min of Euclidean distance). Now, the user gets to move M around the boundary of R. As M is moved, B moves (possibly) to stay locally as close as possible to M. The goal is to move M in order to

get B to a target destination point t on the network, or to decide that it is not possible. Develop a (visual) program that lets the user move the magnet and thereby control the bead.

Can you devise and implement an algorithm to plan a movement of M to get B to t (or decide that it is impossible)? One can be based on establishing a "state graph", in which nodes are pairs (X, Y), where X is an open interval or a discrete point on the boundary of R, and Y is an open interval or a discrete point on a segment within the network G. The system is in state (X, Y) if the magnet's location is X and the bead's is Y. If n hor/vert segments make up the arrangement of road segments, there are only $O(n^2)$ choices for Y and only $O(n)$ for X. In the state graph, make an edge from (X, Y) to (X', Y') if if moving the magnet from X to X', when the bead is in Y, causes the bead to move to Y'. We can now view the planning problem of manipulating the magnet so that the bead gets to t as finding a path in the state graph, from the initial state to any state (X, t). This can be solved by simple depth-first search, e.g., in the state graph.

Another possible direction for a project involving this problem is to create a 2-player game, in which each player manipulates his magnet and tries to get the bead to go to his "goal" (each player has a goal). A simple model for attraction is this: the bead is attracted to only one magnet – the one that is currently closest to it.

Be creative!

(7). Let $S = \{p_1, \ldots, p_n\}$ be a set of n points within the unit square, $U = \{(x, y) : 0 \leq x \leq 1, 0 \leq y \leq 1\}$, in the plane. Assume that $(0, 0) \in S$ (i.e., the origin is one point of S). A rectangle R_i is *valid* if (a) p_i is the bottom left corner of R_i, and (b) no other point $p_j \in S$ lies interior to R_i. Our goal is to find a set of valid rectangles that forms a "packing" (which means that no two rectangles overlap – they can touch on their boundaries, but have no overlap of their interiors) of maximum possible area.

Devise a heuristic (or exact) algorithm to find a good set of rectangles, using concepts from CG. Try out your idea on sets of random points S in U. Compute the areas you get. (Are they always over 1/2? It is conjectured that one can always achieve area at least 1/2; can you prove it? Can you prove that one can always achieve area at least ϵ, for any positive ϵ? See the paper by Dumitrescu and Tóth, "Packing Anchored Rectangles", *Combinatorica* 35 (1) (2015) 39–61, as well as papers that cite it.)

(8). *Computing a "Central" Trajectory.* The input is a set of n (polygonal) trajectories, with each specified as a sequence of $k+1$ "points", (x, y, t), in space-time. For instance, the trajectory $\tau = ((x_0, y_0, t_0), (x_1, y_1, t_1), \ldots, (x_k, y_k, t_k))$ corresponds to a trajectory for a point that starts at (x_0, y_0) at time t_0, then travels at constant speed along a straight line segment to (x_1, y_1), where it arrives at time t_1 (the time difference, $t_1 - t_0$, and the distance from (x_0, y_0) to (x_1, y_1) determine the speed of motion). Assume that the time stamps t_0 and t_k are the same for all trajectories and that the starting point, (x_0, y_0), and destination point, (x_k, y_k), are also the same for all trajectories. Other intermediatre points/times along trajectories can be arbitrary (varying from trajectory to trajectory), except that time is monotone ($t_0 < t_1 < \cdots < t_k$) for each trajectory. Your goal is to define, compute, and compare notions of a "central trajectory" that represents the input set of n trajectories. The idea is that the trajectory data may be noisy, and while the trajectories are all "similar" in some sense (starting and ending at the same times/places), the motion varies: Can we compute a single trajectory, τ^*, that "optimally" represents the set of n trajectories, serving as a "central trajectory". Describe your proposed definitions and try them out on trajectory data that you randomly generate in a meaningful way. How efficient are your algorithms? (For some definitions, you may implement an algorithm that is not theoretically best (maybe your implementation is even brute-force), but you should state what the best method in theory is.)

(9). The k-box cover problem takes as input a set S of n points in the plane and an integer k. The output should be a set, $\{B_1, \ldots, B_k\}$, of k axis-aligned boxes (rectangles with sides that are vertical/horizontal) that cover S. The goal is that the sum of the "sizes" ("size" may be area or perimeter or diameter) of the boxes be minimized.

Design at least two distinct heuristics for k-box cover in 2D, and run comparisons between them, and compare to brute-force methods for small values of k ($k = 2, 3, 4$).

(10). Use a bounding volume hierarchy (e.g., of axis-aligned bounding boxes) to perform intersection queries among a set of n line segments in the plane. Play with different methods to build the hierarchy (i.e., to

partition the segments into two sets) and see what impact they have on the running time.

Try randomly generated line segments. How fast can you solve the REPORT problem to find all pairs of intersecting segments?

Another option is to do a careful experimental study of grid-based methods, quadtree-based methods, etc. Investigate the best grid sizes, considering the space-time tradeoffs.

(11). Implement an algorithm that runs in $O(n)$ time to triangulate a y-monotone simple polygon having n vertices. The input is assumed to be a text file with a clockwise listing of the (x, y) coordinates (floating point) of the vertices; each line of the text file consists of two floating point numbers, separated by a space. (Again, it will be useful to have the option of mouse-clicking input too.) You may assume that the polygon is simple and that it is y-monotone (you need not check to confirm this in your code), but you do not know which vertex comes first (e.g., it need not be the topmost or the bottommost). Again, there may be degeneracies (duplicated points, collinear points, etc).

The output should be a list of pairs of points that define the diagonals of the triangulation. (You need not build a DCEL or other data structure for the triangulation.)

Conduct experiments to determine the in-practice efficiency of the algorithm for large inputs.

(12). Implement the Hertel-Mehlhorn algorithm to compute a decomposition of a simple polygon P into a small number of convex polygonal pieces. You can use O'Rourke Triangulation code to produce the initial triangulation, but you will have to put it into a DCEL (or similar) in order to do the Hertel-Mehlhorn algorithm efficiently. (It is OK if the conversion to DCEL is done naively, but the H-M algorithm should be done efficiently (linear time), after you have a DCEL triangulation.

(13). Given a simple polygon P, compute an "optimal" triangulation using dynamic programming. Here, "optimal" may mean (a) minimum weight triangulation (minimize the sum of the edge lengths of diagonals), (b) minimize the longest edge, (c) maximize the shortest edge, etc. The input is assumed to be a text file whose first line gives the x and y coordinates of p (separated by a space), and whose remaining lines give a clockwise listing of the (x, y) coordinates (floating point) of the vertices (two floating point numbers, separated by a space). It will be useful to have the option of mouse-clicking (and saving) input too. You may assume that the polygon is definitely simple.

Experimentally compare the weight of the resulting triangulation with the weight of the triangulation obtained using ear clipping (algorithm Triangulate, from O'Rourke).

(14). Implement the algorithm VisibleVertices(p, S) (page 328 (3rd edition)) to determine the set of vertices of a given set of disjoint simple polygons that are visible from a point p. You may assume that each simple polygon in the set S is given by a list of its vertices in order (ccw) around its boundary, that they are known to be pairwise disjoint (and not touching), and that p is necessarily disjoint from S. Draw the connections (line segments) from p to each visible vertex.

(15). Let S be a set of n points ("sensors") in the plane. These sensors lie within a polygonal domain (polygon with holes ("obstacles")), P. We say that two points $p, q \in S$ *see* each other if the segment $pq \subset P$ (i.e., the points are line-of-sight visible to each other). The *visibility graph*, $VG(S, P)$, joins pairs of sensors that are line-of-sight visible; the nodes are the sensors S and the edges E join pairs of visible sensors.

Our goal is to find a longest (or nearly longest) edge of $VG(S, P)$, without building the entire visibility graph. Come up with some clever ideas to try, and do an experiment to compare your ideas with the naive method of just trying every pair of points from S.

What about finding a shortest (or nearly shortest) edge of $VG(S, P)$?

(16). Implement a method to place a "small" number of guards in a simple polygon, P. The goal is to design a reasonable heuristic to do this, making sure that the entire polygon is seen. The input is assumed to be a text file whose first line gives the x and y coordinates of p (separated by a space), and whose remaining lines give a clockwise listing of the (x, y) coordinates (floating point) of the vertices (two floating point numbers, separated by a space). It will be useful to have the option of mouse-clicking (and saving) input too. You may assume that the polygon is definitely simple.

The implementer of this algorithm may work in partnership with the implementer of the witness point algorithm below.

(17). Implement a method to place a "large" number of independent witness points in a simple polygon. The input is assumed to be a text file whose first line gives the x and y coordinates of p (separated by a space), and whose remaining lines give a clockwise listing of the (x, y) coordinates (floating point) of the vertices (two floating point numbers, separated by a space). It will be useful to have the option of mouse-clicking (and saving) input too. You may assume that the polygon is definitely simple.

(18). Implement an algorithm to test if a given simple polygon P is star-shaped (1-guardable). This can be done in linear time. The algorithm is described in: Lee, D. T.; Preparata, F. P. (July 1979), "An Optimal Algorithm for Finding the Kernel of a Polygon", Journal of the ACM 26 (3): 415–421, doi:10.1145/322139.322142. See also the wikipedia article on star-shaped polygons.

(19). *The Guarding Game.* Develop an applet or an app to let users play the following "Guarding Game". Given a set S of n points in the plane (moused in or randomly generated). There are 2 players: The "Guarder" and the "Polygonalizer". There are two modes of play: (a) single-round, and (b) multi-round. For the project, you can choose to do either one; if you partner with someone, then you should do both modes.

(a). In single-round play, the Guarder selects a subset of $k \leq n$ points, trying to pick them cleverly. The Guarder has no idea what the polygon P will be; he only knows that the vertices of P will be exactly the points S. The parameter k is selected prior to playing the game; by default, I suggest using $k = \lfloor n/3 \rfloor$ or $k = \lfloor n/4 \rfloor$. The Guarder makes his choice without the Polygonalizer seeing it (in an ideal world, this is an app that is played on 2 devices, but you can implement it with a split screen (2 windows) for simplicity). Separately, the Polygonalizer draws a simple polygon on the set S of vertices, e.g., using the mouse to click in the connections among the points. Your software should check that there are no self-intersections (that it is a valid polygonalization). Then, the two players compare their results: the Guarder wins if P is guarded by the set $G \subset S$ of vertices he selected; otherwise, the Polygonalizer wins. Your implementation should compute to check if G guards P. In general, this might be a bit tricky; to make it much easier, we will say that G guards P if the vertices G succeeed in seeing all of the *vertices* of P (i.e., the set S). (Note that G might see all vertices of P but fail to see some interior points of P; do you see an example?)

(b). In the multi-round mode of play, the Guarder and the Polygonalizer take turns, alternating. (You should allow for either player to go first.) Each time the Guarder takes a turn, he places a guard at one vertex (with the Polygonalizer watching). Each time the Polygonalizer takes a turn, he selects M edges, connecting pairs of vertices S, where M is a parameter that can be adjusted (make the default $M = 3$). Since his goal is to construct a polygonalization of S, his edges can share endpoints but cannot cross; at the end, they should form a cycle that defines a valid simple polygon P. (If n is not divisible by M, his final play may have fewer than M edges.) Again, the Guarder wins if P is guarded by the set, G, of guards selected by the Guarder; otherwise, the polygonalizer wins. (Again, it suffices to consider that the *vertices* of P be seen by the guards G.)

Play the game and have some fun! Who is at an advantage? What is the best strategy? (optional questions, to which I do not know the answers)

(20). Implement the incremental linear programming algorithm of Section 4.3 for the case of two dimensions ($d = 2$). The input is assumed to be a text file of floating numbers, with the first line being c_1 c_2 (separated by a space), then the next n lines being $a_{i,1}$ $a_{i,2}$ b_i (with spaces separating the numbers). The output should print the optimal objective function value (with the option to return "NOT FEASIBLE" or "UNBOUNDED"), along with the point (x, y) that achieves the optimal value. You should animate the algorithm, showing each constraint line as it is inserted, along with the currently optimal vertex, etc.

Apply the algorithm to the red-blue line separation problem: The input is a set R of "red" points and a set B of "blue" points, and the output should be a line ℓ that separates R and B, or the statement that no such line exists.

There is no need to randomize the order of the input; you may assume that the constraints arrive in random order already.

(21). Implement the randomized incremental algorithm of Chapter 4 for computing the minimum enclosing ball of a set of points in the plane.

(22). Implement kd-trees for points in the plane. The input should be a set of points, $S = \{p_1, p_2, \ldots, p_n\}$, given either by mouse clicks or by reading a text file, with each line consisting of floating point numbers x_i and then y_i, separated by a space. (The first line of the file should be the number, n, of points.) The program should then allow one to specify a rectangle, $[x, x'] \times [y, y']$, by text input from the user and/or mouse input from the user. The output, then, should be a listing of the points that are within the query box. (If you use graphical output, change the color of the points that lie within the query box.)

Perform the following experiment with your code. For various values of n (e.g., $n = 100, 200, 300, \ldots, 1000$, or other interesting range of values), do the following. Generate a set S of n random points in the unit square (each x_i and y_i is uniformly distributed between 0 and 1, using the random number generator of the language in which you wrote the code). Build the kd-tree for each S, and compute the average time, over, say, 20 samples, it takes to answer a query Q given by a box that is, say, 0.1-by-0.1. (A simple way to do this is to generate the query box as $[x, x + 0.1] \times [y, y + 0.1]$, where x and y are chosen uniformly at random from the interval (0,0.9).) Compare the speed of the kd-tree method to the "brute force" method of simply testing all n points, one by one, to see which ones are in the query box. It would be nice to see a plot of the two times, as a function of n. For what value of n is it worth using the more sophisticated data structure of the kd-tree?

Also use kd-trees on an input set of axis-aligned rectangles, so that those within a query box can be reported/counted.

(23). This project is just like the previous project, except that instead of using a kd-tree to do orthogonal range queries, use a 2d range tree. (You may do it without the use of fractional cascading, if you want.)

(24). Implement the point location data structure (DAG) of Chapter 6. The input consists of a list of segments $S = (s_1, s_2, \ldots, s_n)$, which are to be inserted in the order given. Show graphically the trapezoidal diagram as each one is inserted. (Basic animation of the algorithm using simple graphics will greatly ease the debugging.)

(25). Implement and experiment with the Lawson edge swap algorithm (LegalizeTriangulation) for Delaunay triangulations of points in the plane.

(26). Implement and experiment with the randomized incremental method for Delaunay triangulations of points in the plane.

(27). Implement a method to compute a minimum-area terrain triangulation of a set of points in 3D: Each point $p_i = (x_i, y_i, z_i)$ sits above the xy-plane by distance $z_i > 0$ and our goal is to triangulate the projections of the points onto the xy-plane so that the corresponding set of triangles in 3D has the smallest possible surface area. This problem arises in surface reconstruction from point cloud data. I have some ideas for how to do this with heuristic methods – see me, and I can give details. You are encouraged to come up with your own ideas too. I think local search methods may be good.

I am also very interested in theoretical results: Can you *approximate* optimal in polynomial time? Can you show that the exact solution is NP-hard?

(28). Implement the algorithm to construct an arrangement (in a DCEL) of n lines in the plane.

(29). *Hitting Lines.* Given a set of n lines in the plane, the Hitting Lines problem asks us to find a smallest set of points so that each line is "hit" by one of the points. It is closely related to some guarding problems in polygons. Of course, the hitting points chosen may as well always be at vertices of the arrangement of the n input lines. Devise and compare heuristic methods to find small hitting sets of points for a set of n lines. ("Greedy" methods are natural, but you may think of other options too.) Key to doing experiments on this problem is figuring out a way to generate input sets of n lines that are nontrivial: If you just generate n lines "at random" (e.g., picking slope and intersept uniformly at random), the probability that 3 or more lines pass through a common point is 0. (And, for a set of lines in general position (what we call a "simple arrangement" of lines), it is trivial to find an optimal hitting set, since any vertex where lines cross is a

point of crossing of just *two* lines, so we know we need exactly $\lceil n/2 \rceil$ points to hit all n lines.) Figure out a way to generate "interesting" instances, and try out your heuristic(s) on them. (Can you think of ways of computing a lower bound on the optimal number of points in a hitting set (like our notion of "witness points" in guarding problems)?) Play with this problem as much as you can.

(30). A set S of n line segments in the plane is said to have a Type 1 degeneracy if all segments lie on a common line. They are said to have a Type 2 degeneracy if some subset of 3 or more endpoints are collinear. It is easy to see that a Type 1 degeneracy can be detected in $O(n)$ time. Using duality and building a line arrangement, you can see that a Type 2 degeneracy can be detected in $O(n^2)$ time. (Do you see how?) Implement a degeneracy-tester that takes as input a set of line segments in the plane and reports if there is a Type 1 or Type 2 degeneracy.

(31). *Computing a compact summary of a massive trajectory database.* Trajectory data is everywhere these days, as each of us has associated GPS trails recorded in our phones, showing where we are all through the day. Every modern vehicle is also tracked. In this project, the goal is to compress a large set of trajectories in order to be able to perform basic queries on such data: How many trajectories left region A between 5:07pm and 6:34pm, and arrived in region B between 10:11pm and 10:44pm?

The input is a set T of n (polygonal) trajectories, with each specified as a sequence of $k+1$ "points", (x, y, t), in space-time. For instance, the trajectory $\tau = ((x_0, y_0, t_0), (x_1, y_1, t_1), \ldots, (x_k, y_k, t_k))$ corresponds to a trajectory for a point that starts at (x_0, y_0) at time t_0, then travels at constant speed along a straight line segment to (x_1, y_1), where it arrives at time t_1 (the time difference, $t_1 - t_0$, and the distance from (x_0, y_0) to (x_1, y_1) determine the speed of motion).

Your goal is to compute a "trajectory summary" for T and then use the summary to perform queries on the trajectory set.

(32). Implement and experiment with Timothy Chan's convex hull algorithm ($O(n \log h)$) in 2D. Ideally, show an animation of the algorithm to help make it clear how it works.

(33). Implement a gift-wrapping (Jarvis march) algorithm to compute the non-dominated points of an input point set $S = \{p_1, \ldots, p_n\}$ in the plane. Ideally, the program should allow input by mouse, read from a text file, and randomly generated.

Also implement a randomized incremental algorithm for the same problem. Compare.

The output should be shown graphically, and a list of the non-dominated points in ccw order. Try to animate the algorithms to make it educational. (You will need to be able to control the speed of animation.)

(34). Implement the Bentley-Ottmann sweep to DETECT if a polygonal chain $C = (p_1, p_2, \ldots, p_n)$ is *simple* (it is not simple if it properly crosses itself, at a point interior to two edges, or if it crosses itself at a vertex, in the degenerate case). Report a witness to the crossing if the chain is not simple.

The input is assumed to be either graphical (e.g., allowing the user to mouse-click enter points) or read from a simple text file, with each line consisting of two floating point numbers (the x- and y-coordinates of a vertex) separated by spaces. You should animate the algorithm to show each step during the sweep.

Conduct an experiment to determine the running time in practice for large values of n, for randomly generated inputs. Keep track also of the largest size of the SLS.

(35). (See Problem 2.10, Chapter 2, BKOS) Let S be a planar subdivision with n vertices, stored as a DCEL. Let P be a set of m points. Implement a plane-sweep algorithm to locate each point of P in a face of S. In order to be able to verify correctness (and to debug), you should use a graphical interface and animate the algorithm. Be creative! (It may be useful to have a mouse-based editor to input S and build a DCEL representation of it.)