Divide and Conquer (Part 1)

Debarka Sengupta

Divide and Conquer

- Reduce problem to one or more sub-problems of the same type
- Typically, each sub-problem is at most a constant fraction of the size of the original problem
- Subproblems typically disjoint
- Often gives significant, usually polynomial, speedup
- Examples: Quick Sort, Mergesort, Binary Search, Strassen’s Algorithm, Quicksort. FFT, etc.

Steps involved

- Divide-and-conquer algorithm has three parts:

- Divide: the problem into a number of subproblems that are smaller instances of the same problem.

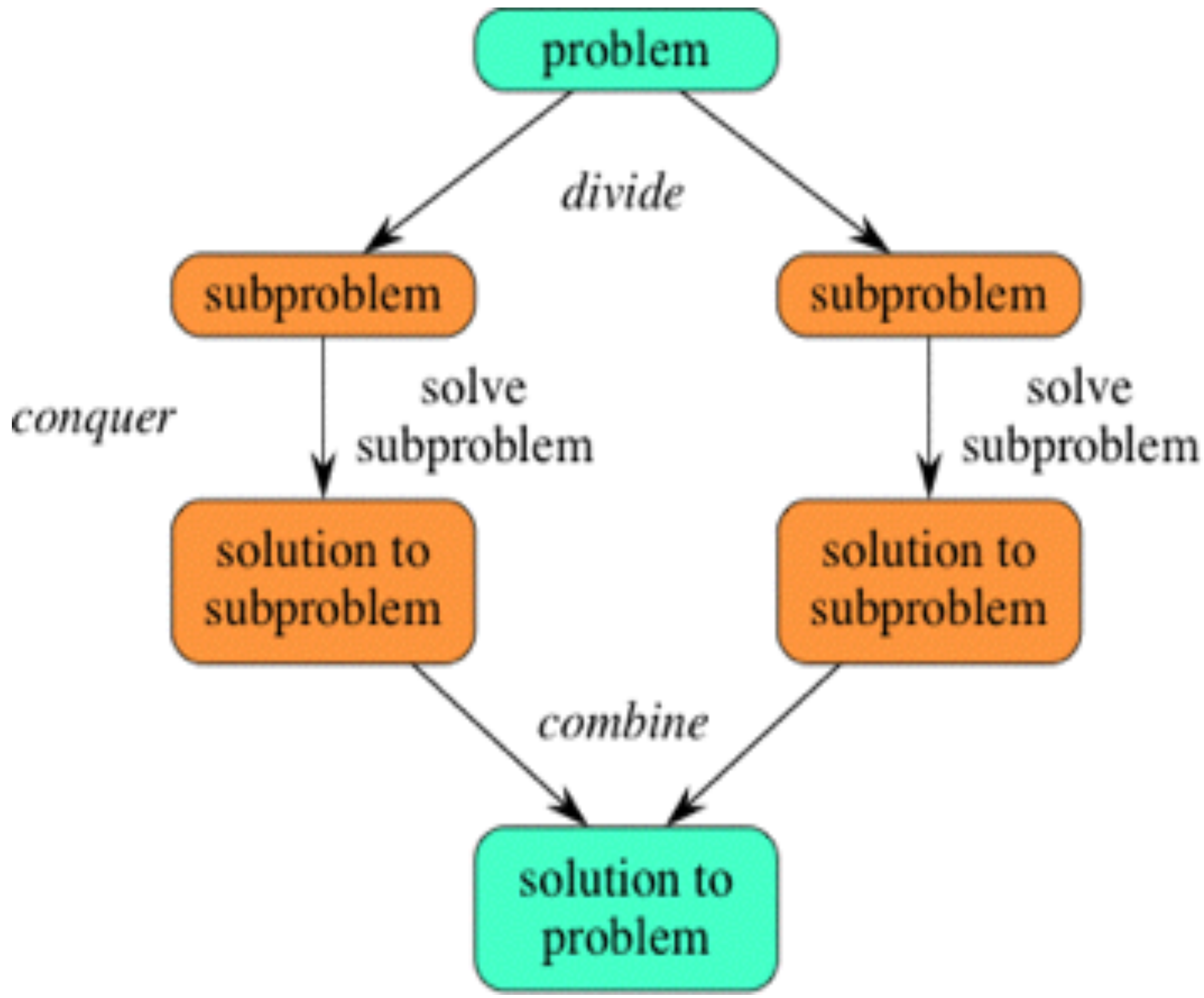
- Conquer: the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.

- Combine: the solutions to the subproblems into the solution for the original problem.Consequence:

Benefit

- Brute force / naïve solution: N^2 (typically)
- Divide-and-conquer: $N \log N$

Algorithm sketch - two subproblems



Algorithm sketch - more than two subproblems

Finding word in dictionary

Suppose you want to find **"janissary"** in a dictionary:

- open the book near the middle
- the heading on the top left page is **"kiwi"**,
- so move back a small number of pages
- here you find **"hypotenuse"**, so move forward
- find **"ichthyology"**, move forward again

The number of pages you move gets smaller (or at least adjusts in response to the words you find)

Common D&C algorithms

- **Mathematics**
    - Polynomial & Matrix Multiplication
    - Exponentiation
    - Large Integer Manipulation
    - FFT
- **Geometry**
    - Convex Hull
    - Closest Pair
- **Searching**
    - Binary Search
- **Sorting**
    - Merge Sort
    - Quick Sort

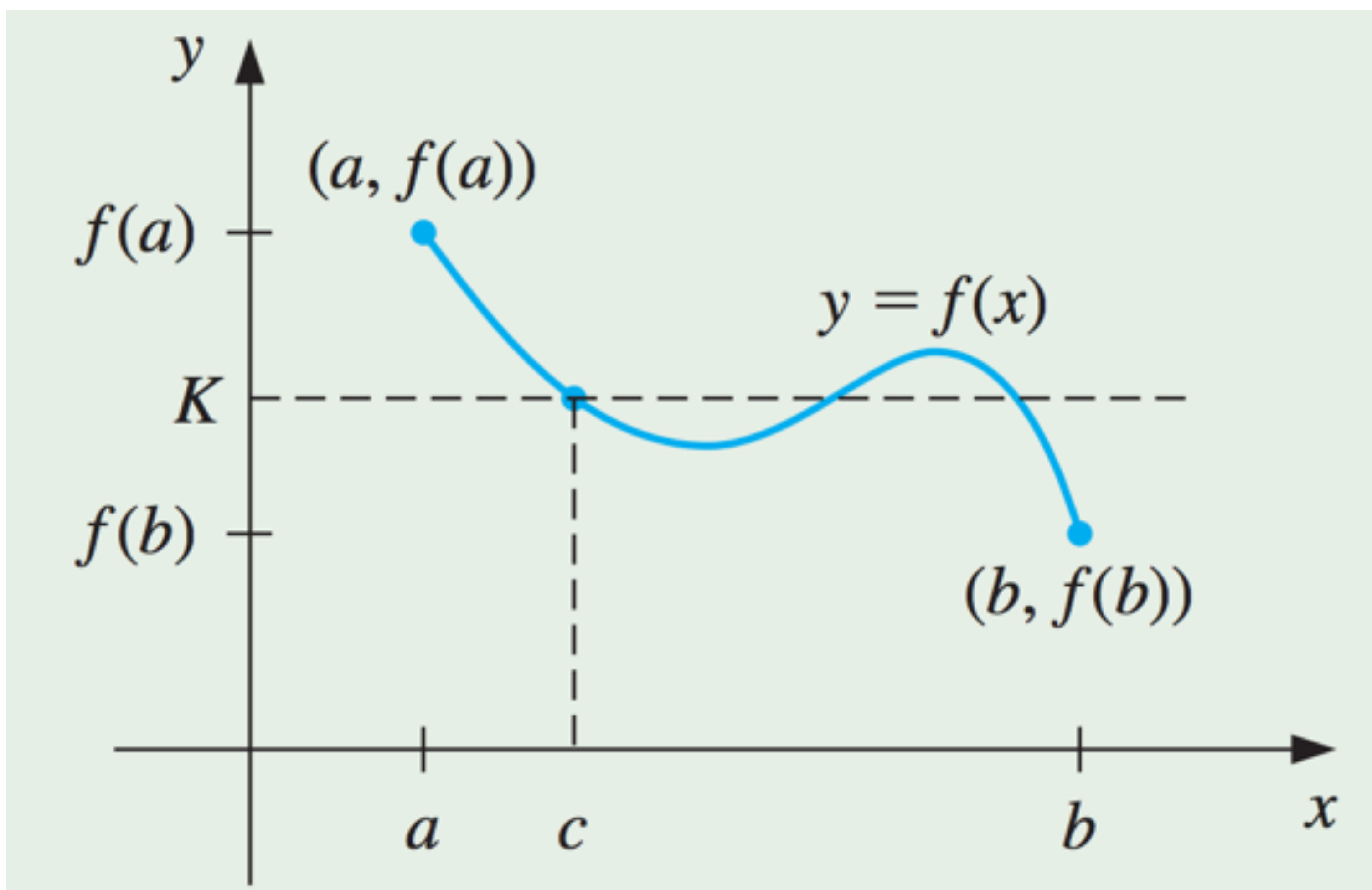
Root finding - bisection

Root finding problem

- Finding Zero of function f(x)
- This process involves finding a root, or solution, of an equation of the form $f(x) = 0$ for a given function f

Intermediate value theorem (IVT)

It simply states that for any value L between f(a) & f(b), there's a value c in [a,b] for which f(c)=L.



Example

- Show that $f(x) = x^3 + 4x^2 - 10 = 0$ has a root in [1, 2] and use the Bisection method to determine an approximation to the root that is accurate to at least within 10^{-4}

Solution sketch

- Because $f(1) = -5$ and $f(2) = 14$ the IVT ensures that this continuous function has a root in $[1, 2]$.

Solution sketch continued …

- For the first iteration of the Bisection method we use the fact that at the midpoint of [1, 2] we have $f(1.5) = 2.375 > 0$.
- This indicates that we should select the interval [1, 1.5] for our second iteration.
- Then we find that $f(1.25) = -1.796875$ so our new interval becomes [1.25, 1.5], whose midpoint is 1.375.
- Continuing in this manner gives the values shown in the following table

Iterations

Iter	a_n	b_n	p_n	$f(a_n)$	$f(p_n)$	RelErr
1	1.000000	2.000000	1.500000	-5.000	2.375	0.33333
2	1.000000	1.500000	1.250000	-5.000	-1.797	0.20000
3	1.250000	1.500000	1.375000	-1.797	0.162	0.09091
4	1.250000	1.375000	1.312500	-1.797	-0.848	0.04762
5	1.312500	1.375000	1.343750	-0.848	-0.351	0.02326
6	1.343750	1.375000	1.359375	-0.351	-0.096	0.01149
7	1.359375	1.375000	1.367188	-0.096	0.032	0.00571
8	1.359375	1.367188	1.363281	-0.096	-0.032	0.00287
9	1.363281	1.367188	1.365234	-0.032	0.000	0.00143
10	1.363281	1.365234	1.364258	-0.032	-0.016	0.00072
11	1.364258	1.365234	1.364746	-0.016	-0.008	0.00036
12	1.364746	1.365234	1.364990	-0.008	-0.004	0.00018
13	1.364990	1.365234	1.365112	-0.004	-0.002	0.00009

Precision

- The number of iterations depends on the precision being looked for

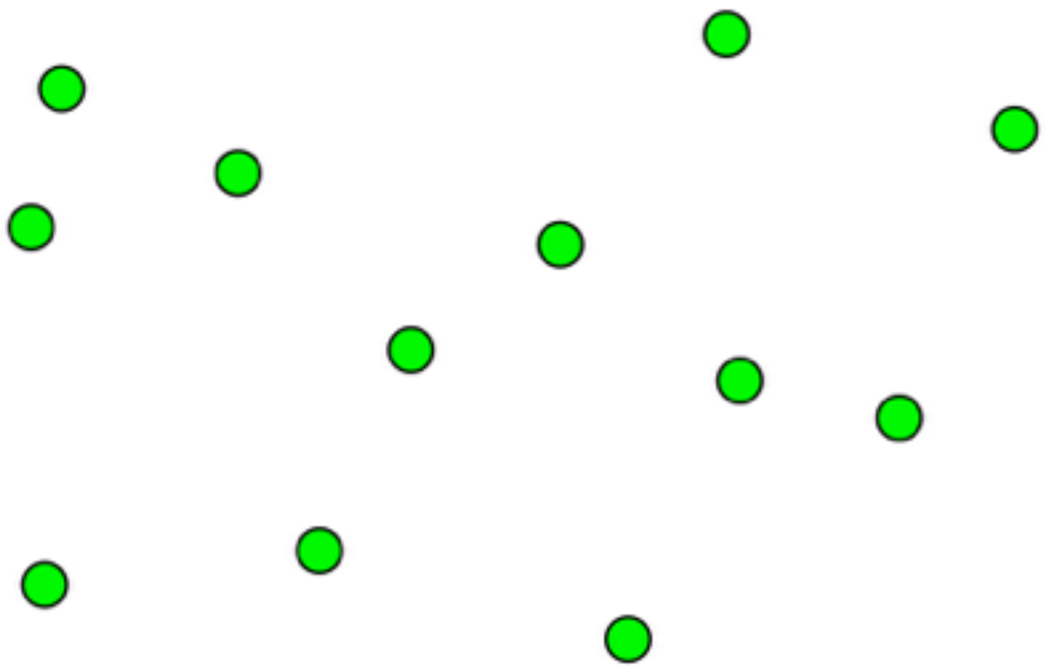
- if $c_1 = (a+b)/2$ is the midpoint of the initial interval, and c_n is the midpoint of the interval in the n^{th} step, then the difference between c_n and a solution c is bounded by $|c_n - c| <= |b - a|/2^n$

- One should be able to perform the back calculation

Closest Pair

Closest pair - problem statement

- **Given a set of points $\{p_1, \ldots, p_n\}$ find the pair of points $\{p_i, p_j\}$ that are closest**

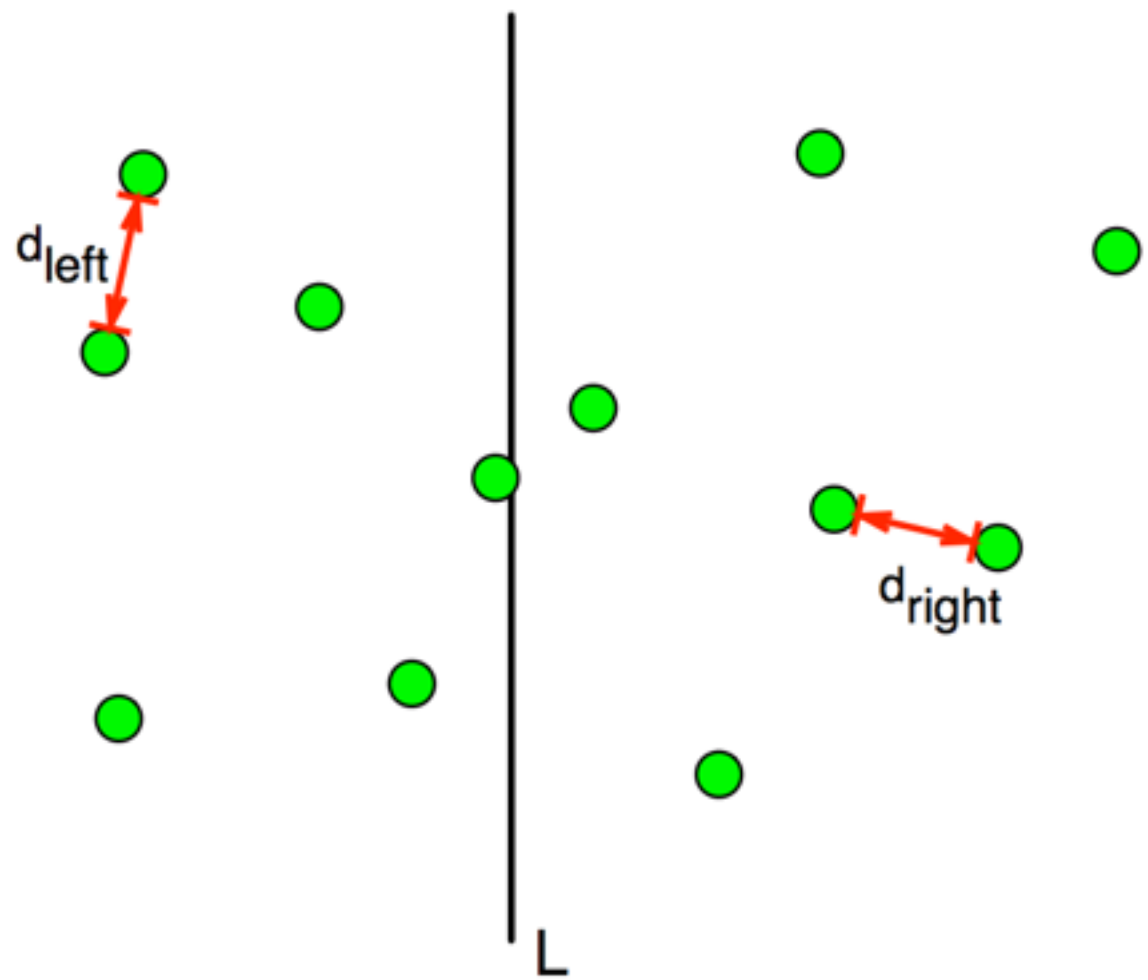


Brute force vs. divide and conquer

- Brute force gives an $O(n^2)$ algorithm to just check ever pair of points.
- Can we do it faster? Seems like difficult without checking every pair.
- In fact, we can find the closest pair in $O(n \log n)$ time.

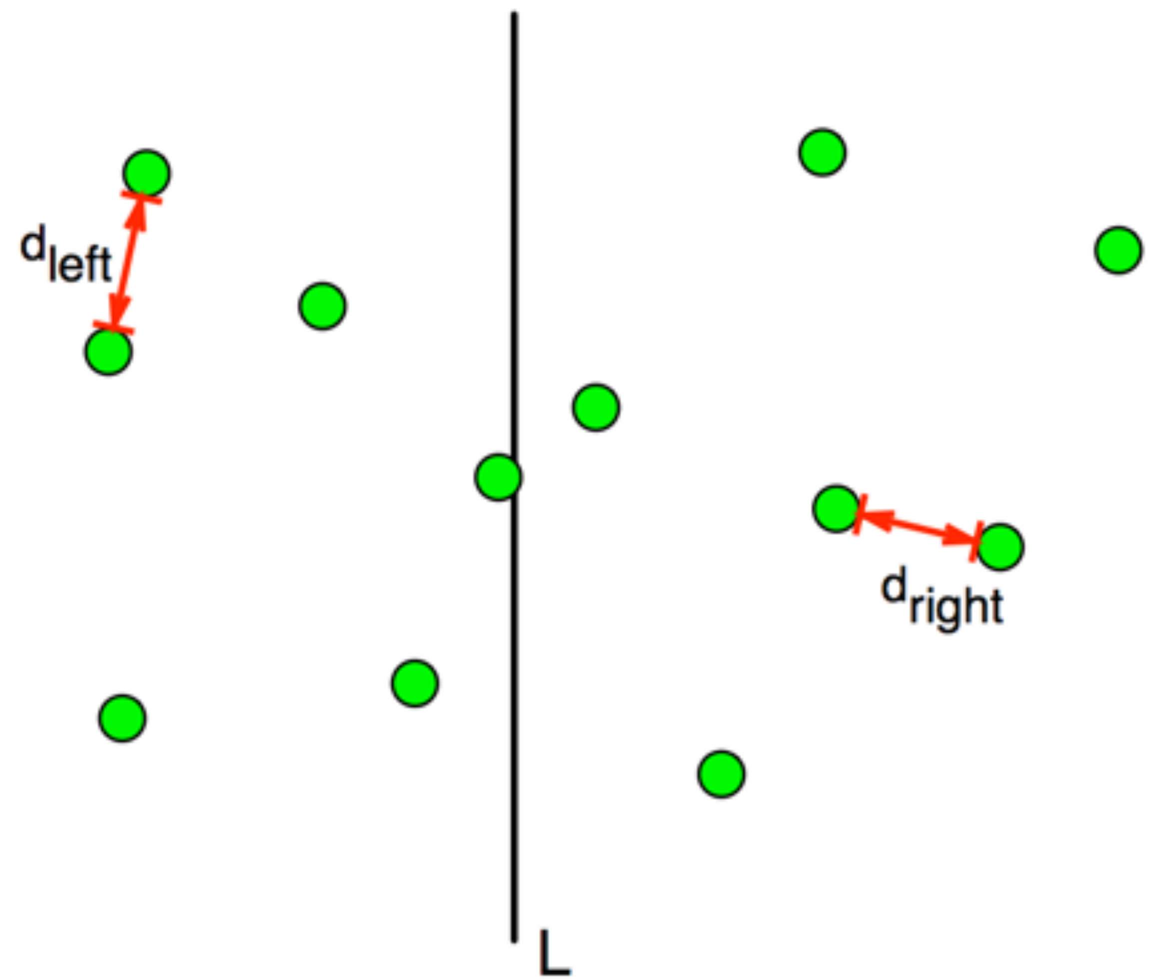
Divide

- Split the points with line L so that half the points are on each side.
- Recursively find the pair of points closest in each half.



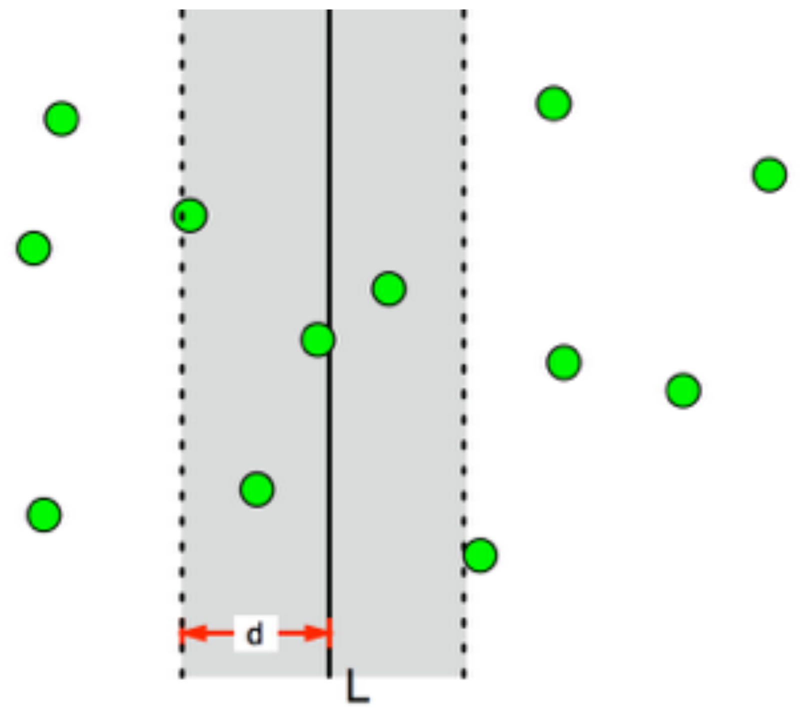
Merge

- Let $d = \min\{d_{left}, d_{right}\}$
- d would be the answer, except maybe L split a close pair!



Region near L

- **If the closest pair exists across the L, it should contained within d margins on both sides**



A life saver observation

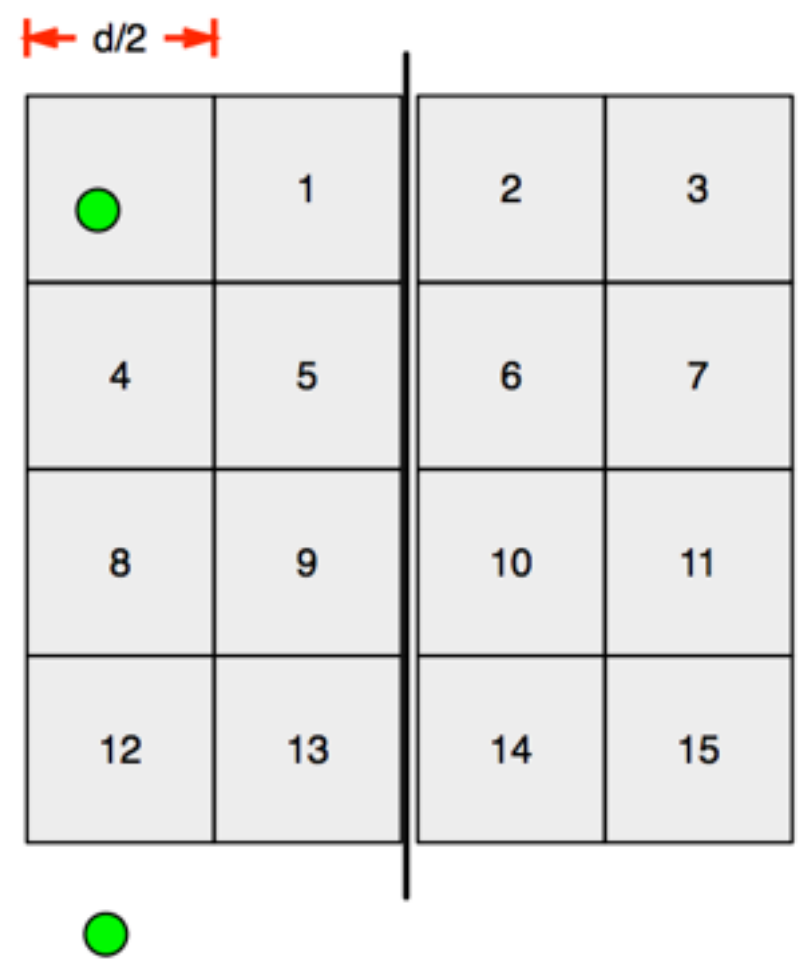
- Let S_y be an array of the points in that region, sorted by decreasing y-coordinate value.
- S_y might contain all the points, so we can't just check every pair inside it.

Theorem

Suppose $S_y = p_1, \ldots, p_m$. If $dist(p_i, p_j) < d$ then $j - i \leq 15$.

Constant number of checks for each point is enough

- We can split the entire margin area into d/2 sided square boxes
- Each box can have max 1 point
- Suppose 2 points are separated by > 15 indices.
  - Then, at least 3 full rows separate them (the packing shown is the smallest possible).
  - But the height of 3 rows is > 3d/2, which is > d.
  - So the two points are further than d apart.



Conclusion

- Starting from the bottom most point, for each point, it is enough to check next 15 data points up in the sequence of increasing value in S_y
- Constant (15) time for each of the n points

The algorithm

```
ClosestPair(Px, Py):
   if |Px| == 2: return dist(Px[1],Px[2])    // base

   d1 = ClosestPair(FirstHalf(Px,Py))      // divide
   d2 = ClosestPair(SecondHalf(Px,Py))
   d = min(d1,d2)

   Sy = points in Py within d of L     // merge
   For i = 1,...,|Sy|:
       For j = 1,...,15:
           d = min( dist(Sy[i], Sy[j]), d )
   Return d
```

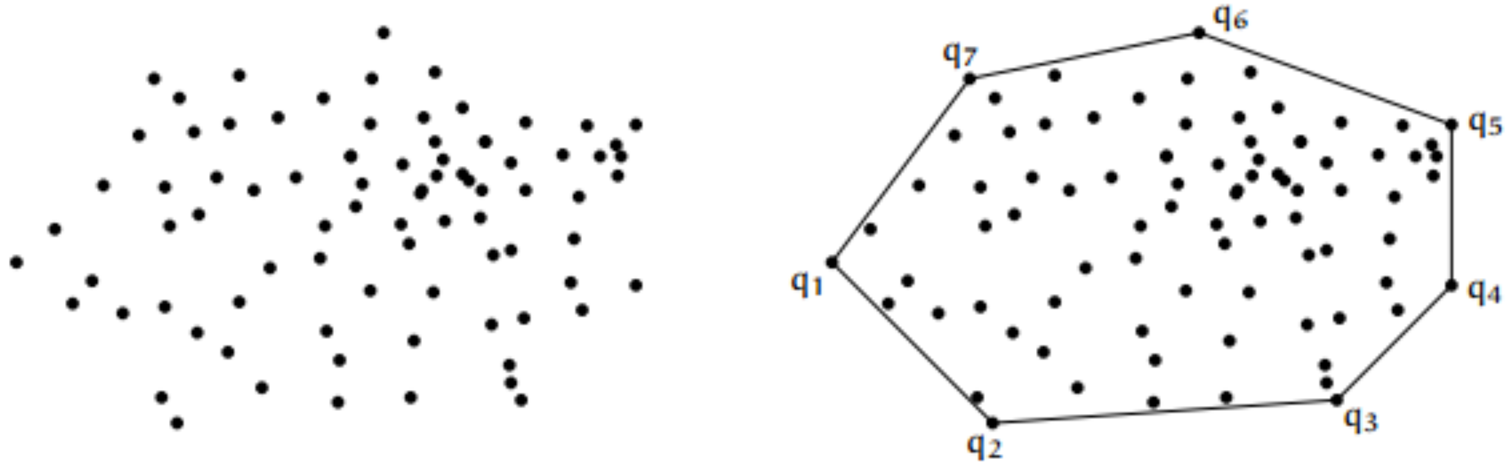
Running time

- Divide set of points in half each time: O(log n) depth recursion
- Merge takes O(n) time.
- Recurrence: $T(n) = 2T(n/2) + cn$
- Analysis is same as MergeSort ⇒ O(n log n) time

Convex Hull

Convex Hull

- Studied in the field of Computational Geometry.
- A convex hull, of a set of N points is the smallest perimeter fence enclosing all these points.
- Convex Hull Output - Sequence of vertices in counterclockwise/ clockwise order.



(a) Input. (b) Output.

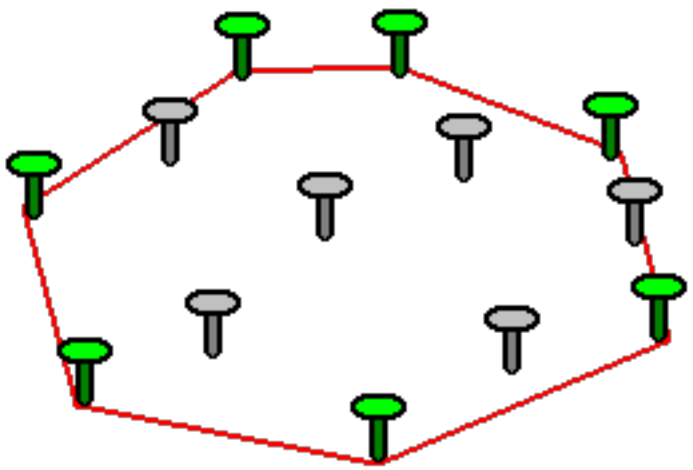
Motion planning

- Robot Motion Planning - Find the shortest path in the plane from a starting point s to an ending point t, that avoids a polygonal obstacle.

- Here, we can see that the shortest path is either the straight line from s to t, or one of the two polygon chains of the convex hull.

Mechanical algorithm

Hammer nails perpendicular to the plane and stretch elastic rubber band around these points.



Brute force

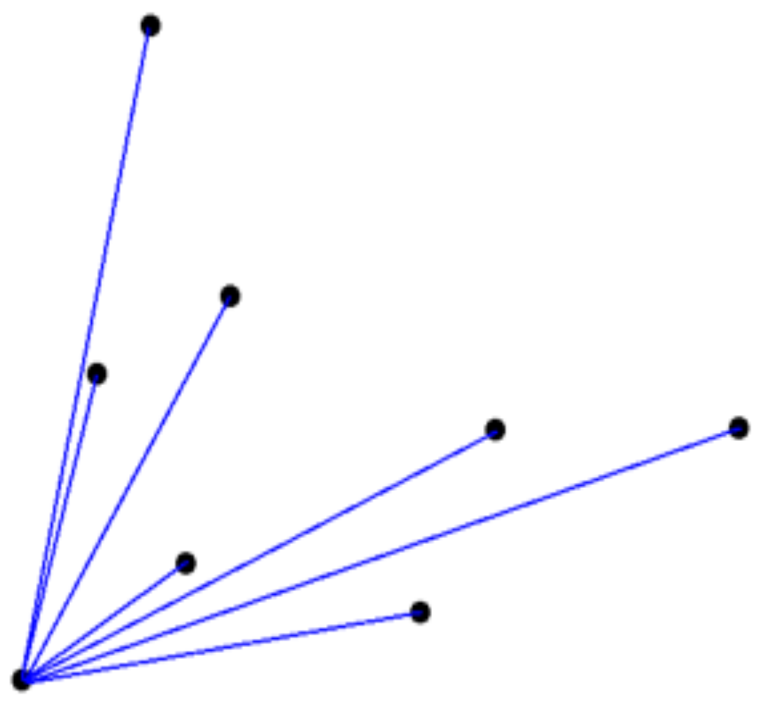
- Create a list of all possible line segments
  - N choose 2 line segments for N points
- For each line segment check if both sides have points
  - If that's not the case, include the segment in the solution set
- Quadratic complexity algorithm

Graham's scan

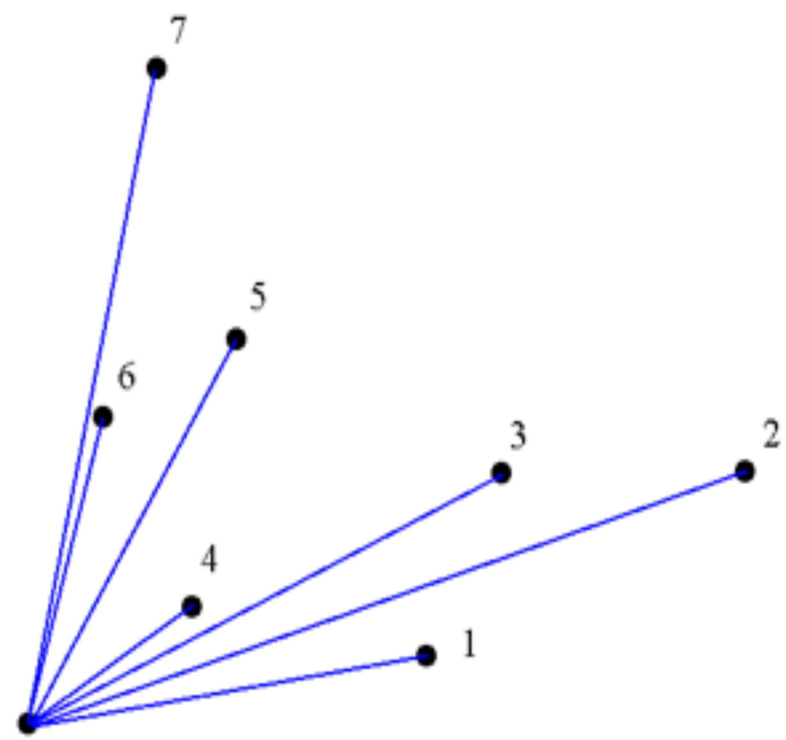
- Start at point guaranteed to be on the hull. (the point with the minimum y value)
- Sort remaining points by polar angles of vertices relative to the first point.
- Go through sorted points, keeping vertices of points that have left turns and dropping points that have right turns.

Graham's scan example

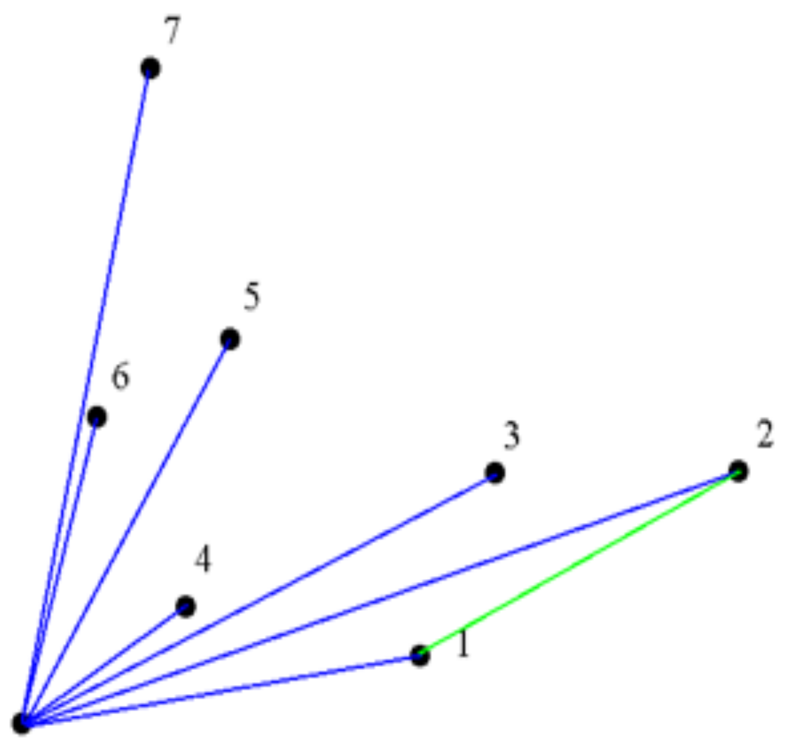
Graham's scan example



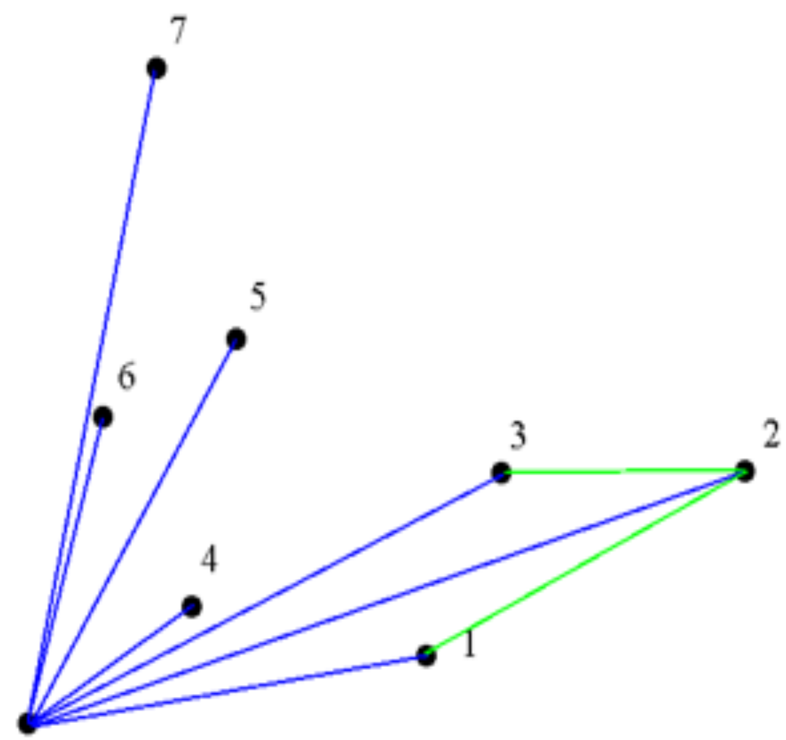
Graham's scan example



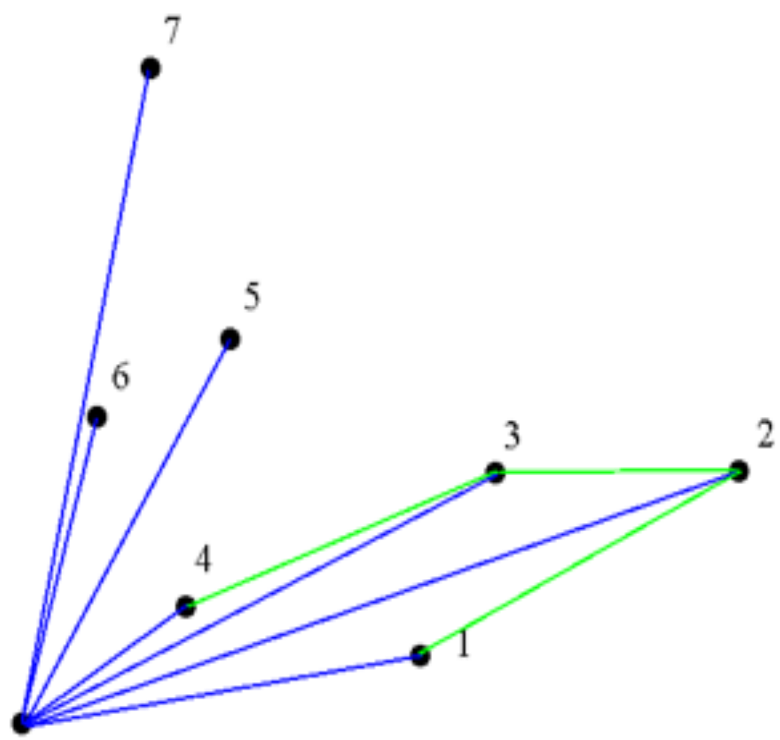
Graham’s scan example



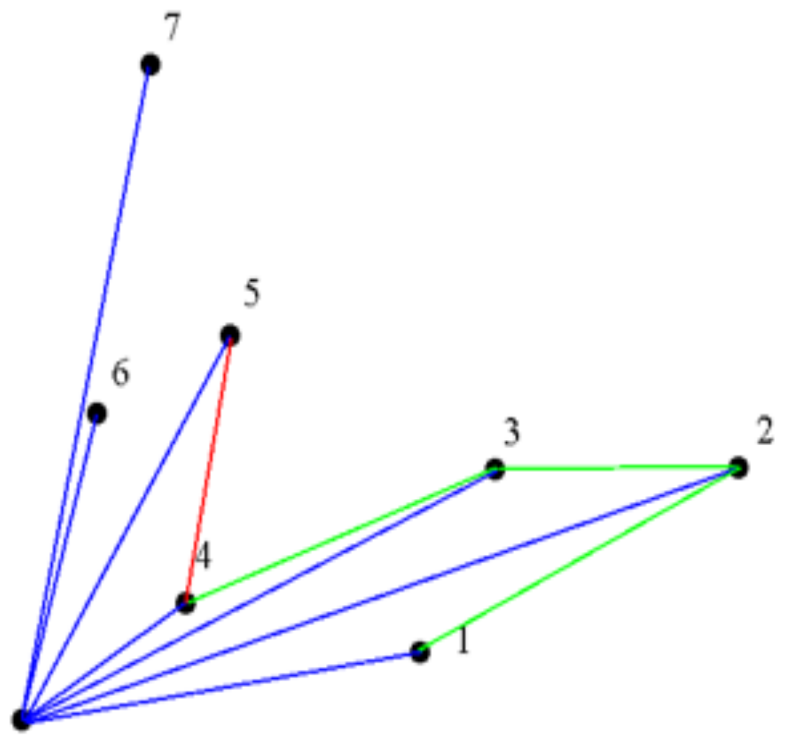
Graham’s scan example



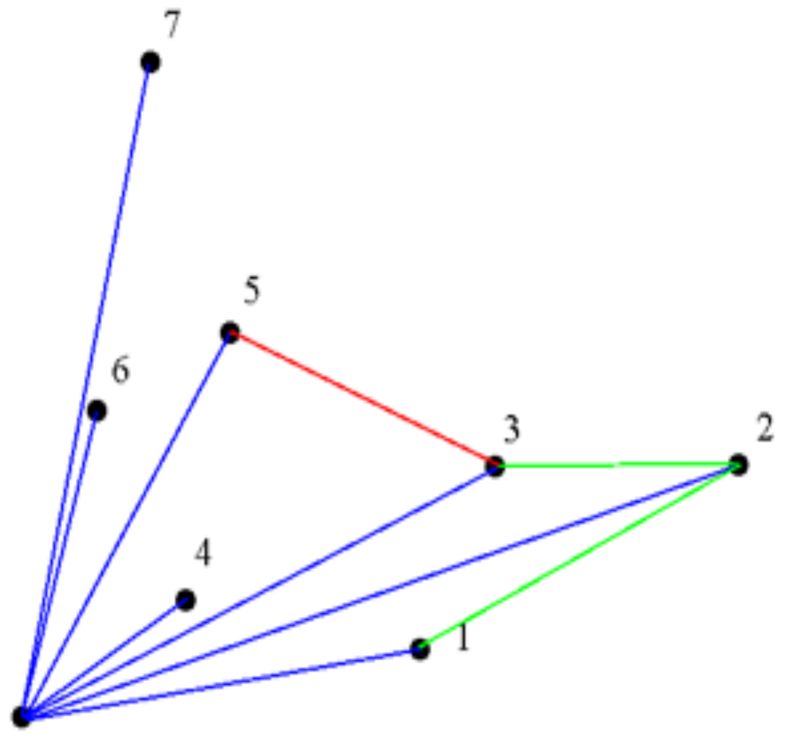
Graham's scan example



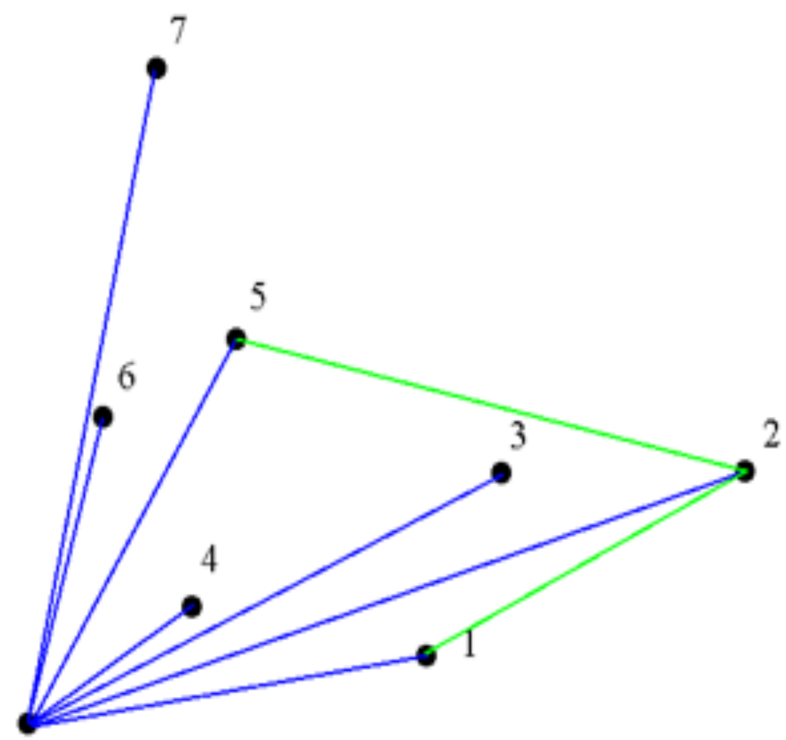
Graham’s scan example



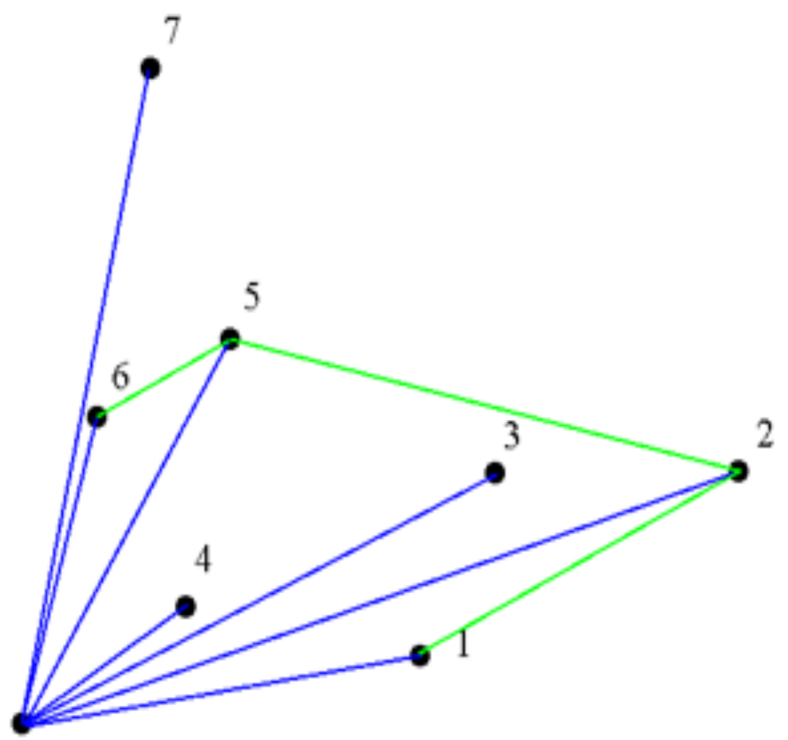
Graham’s scan example



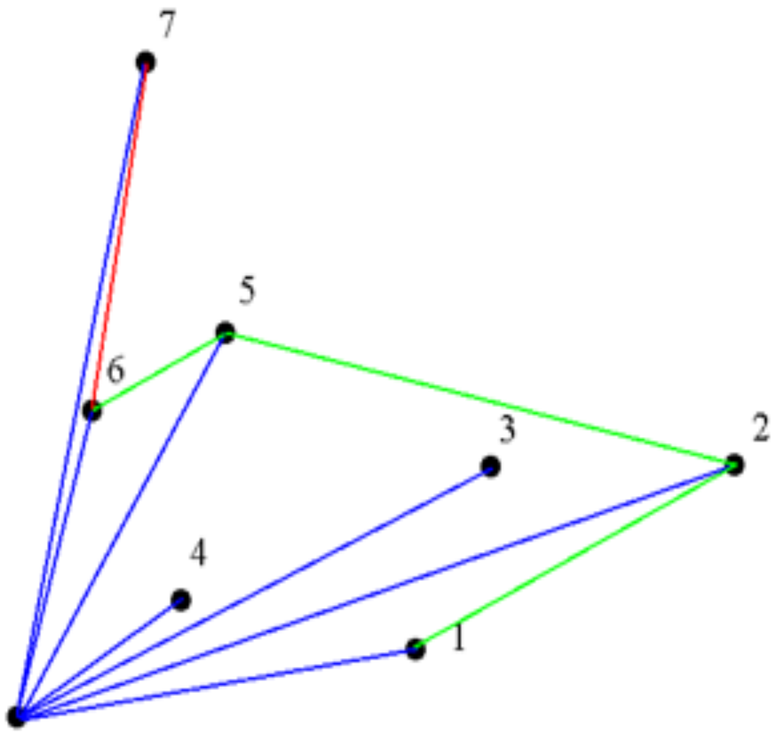
Graham’s scan example



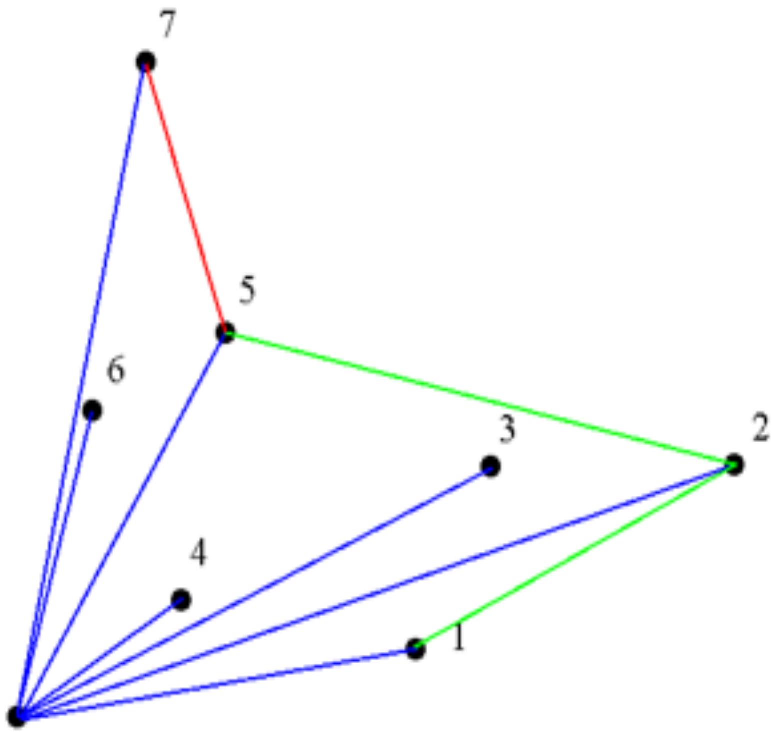
Graham's scan example



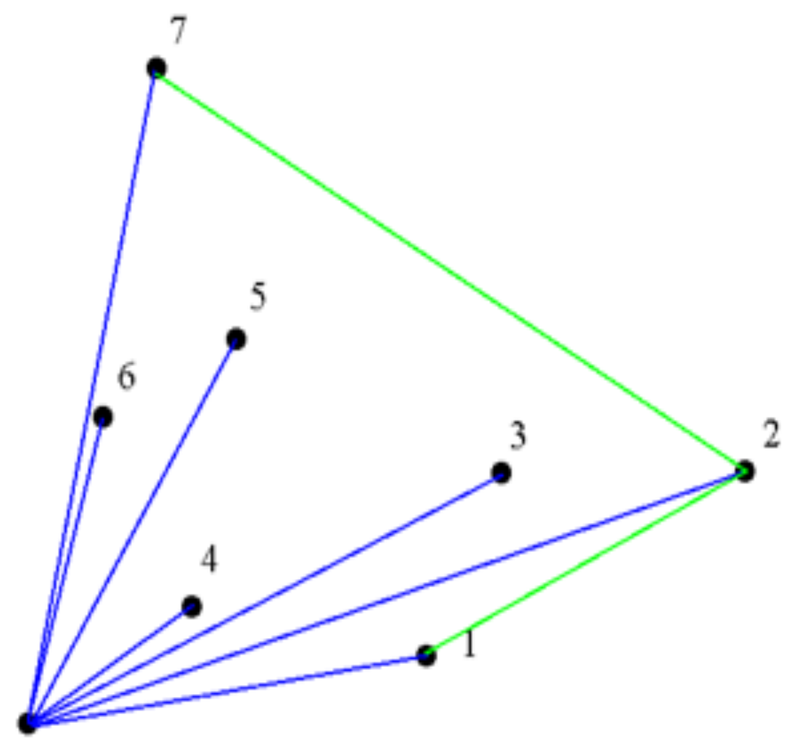
Graham’s scan example



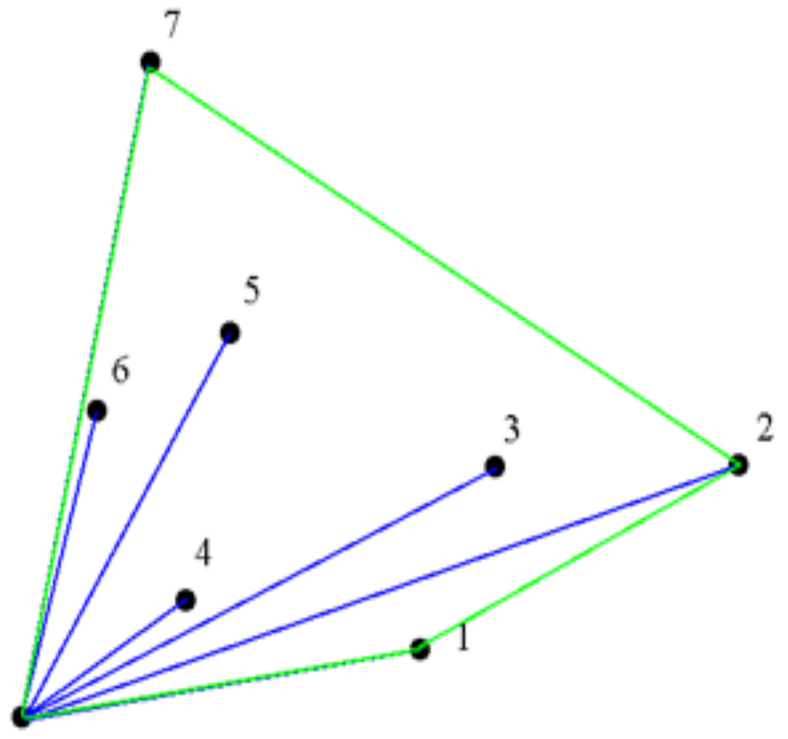
Graham's scan example



Graham's scan example



Graham's scan example

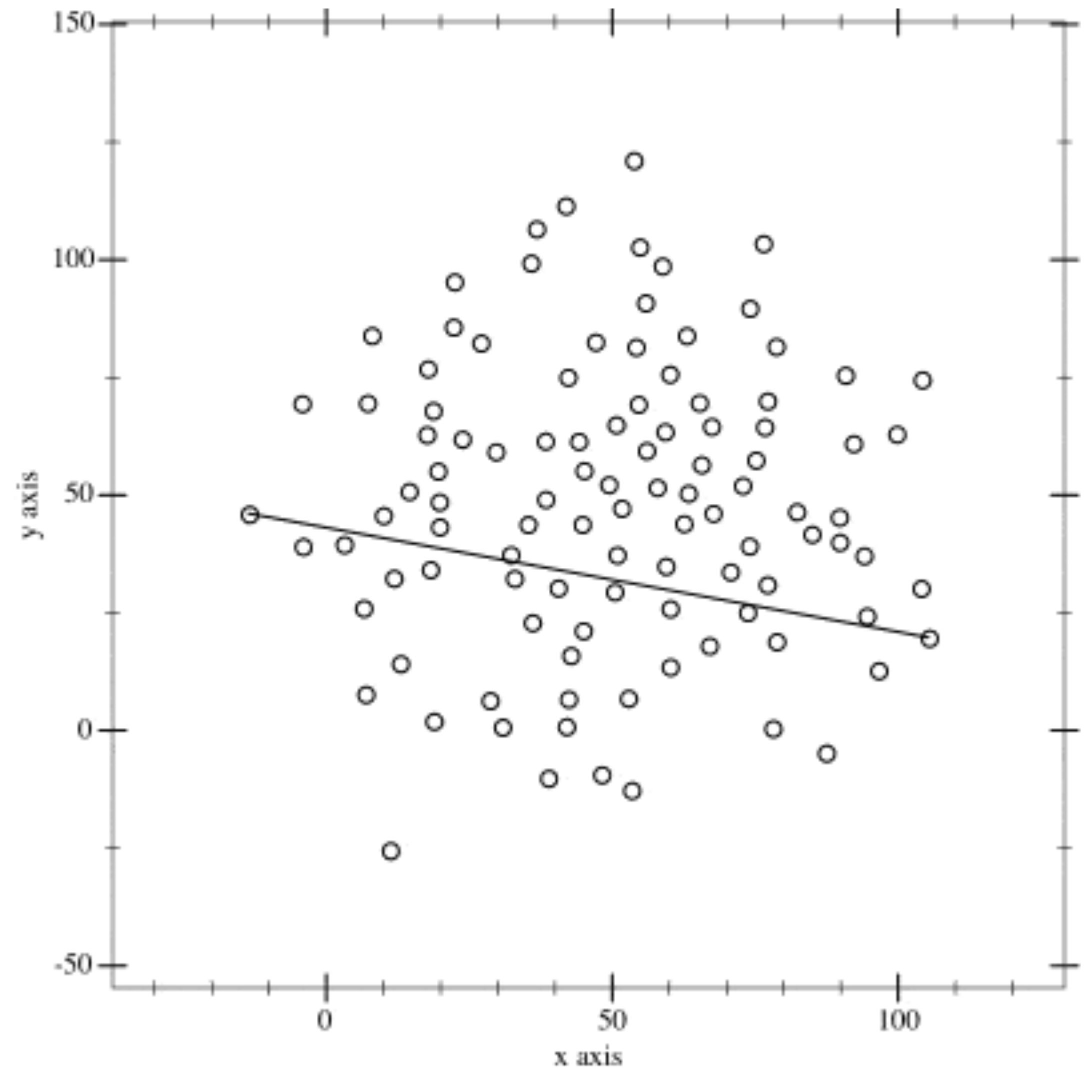


Runtime of Graham's can

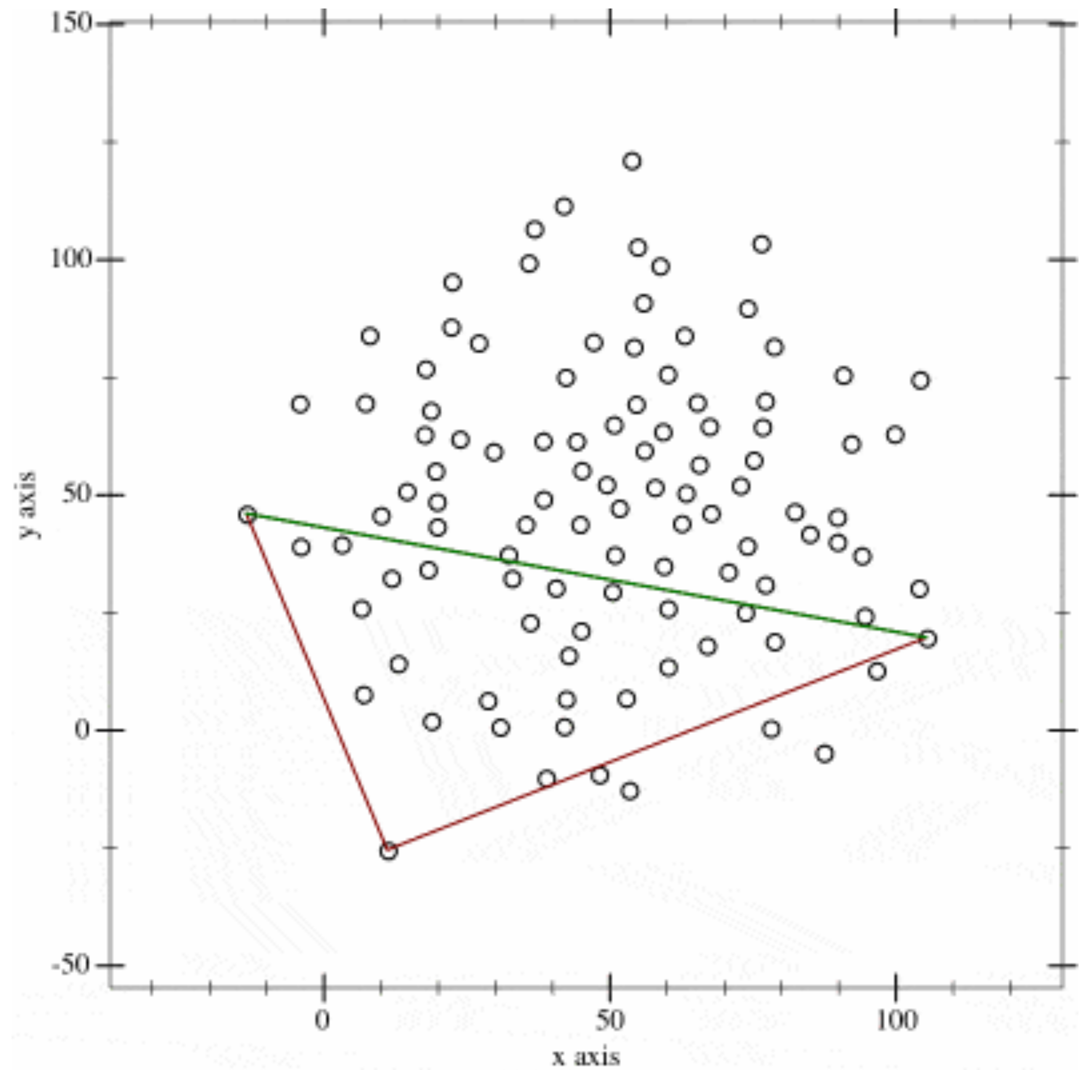
- Graham's scan is O(n log n) due to initial sort of angles.

Quick Hull Algorithm

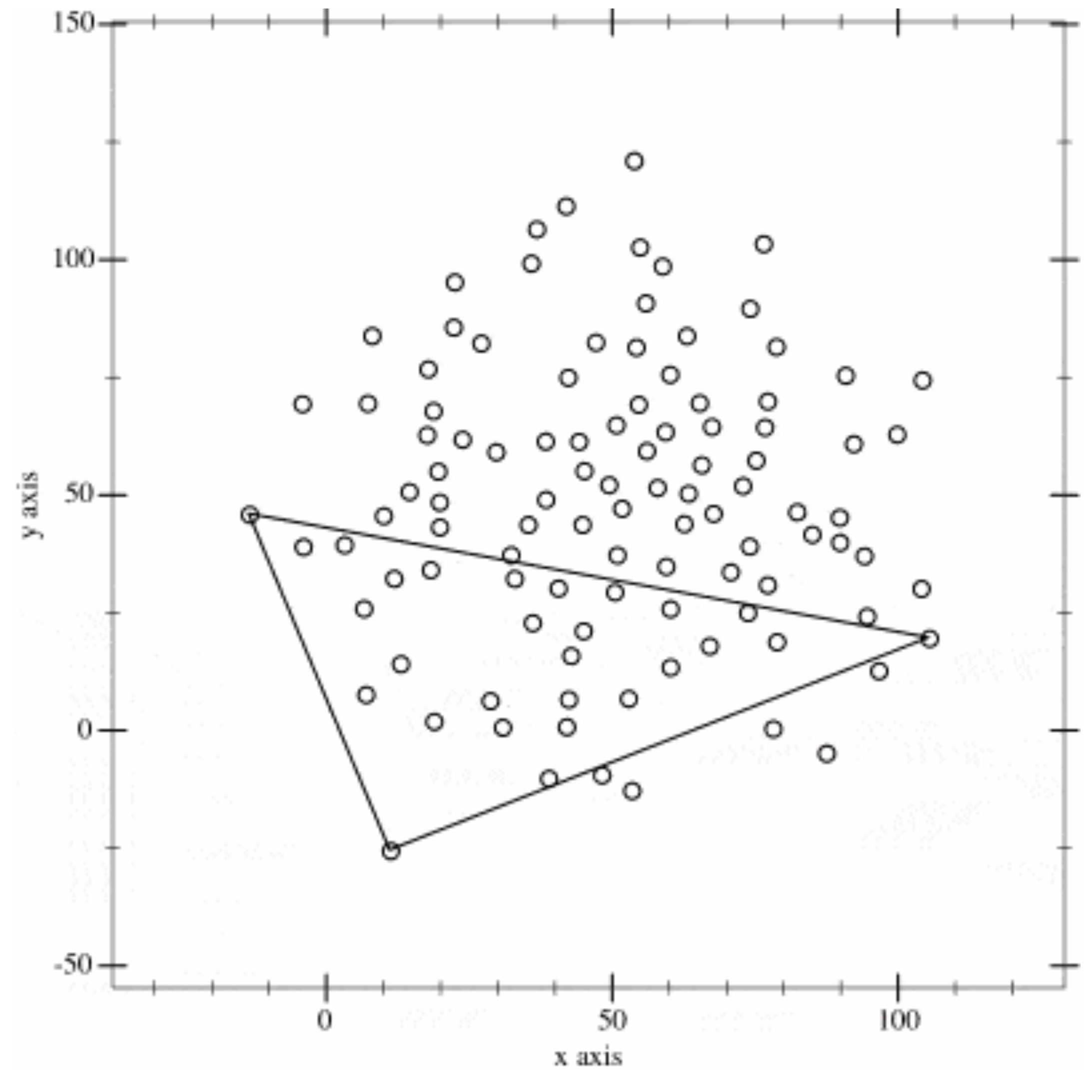
Quick hull example



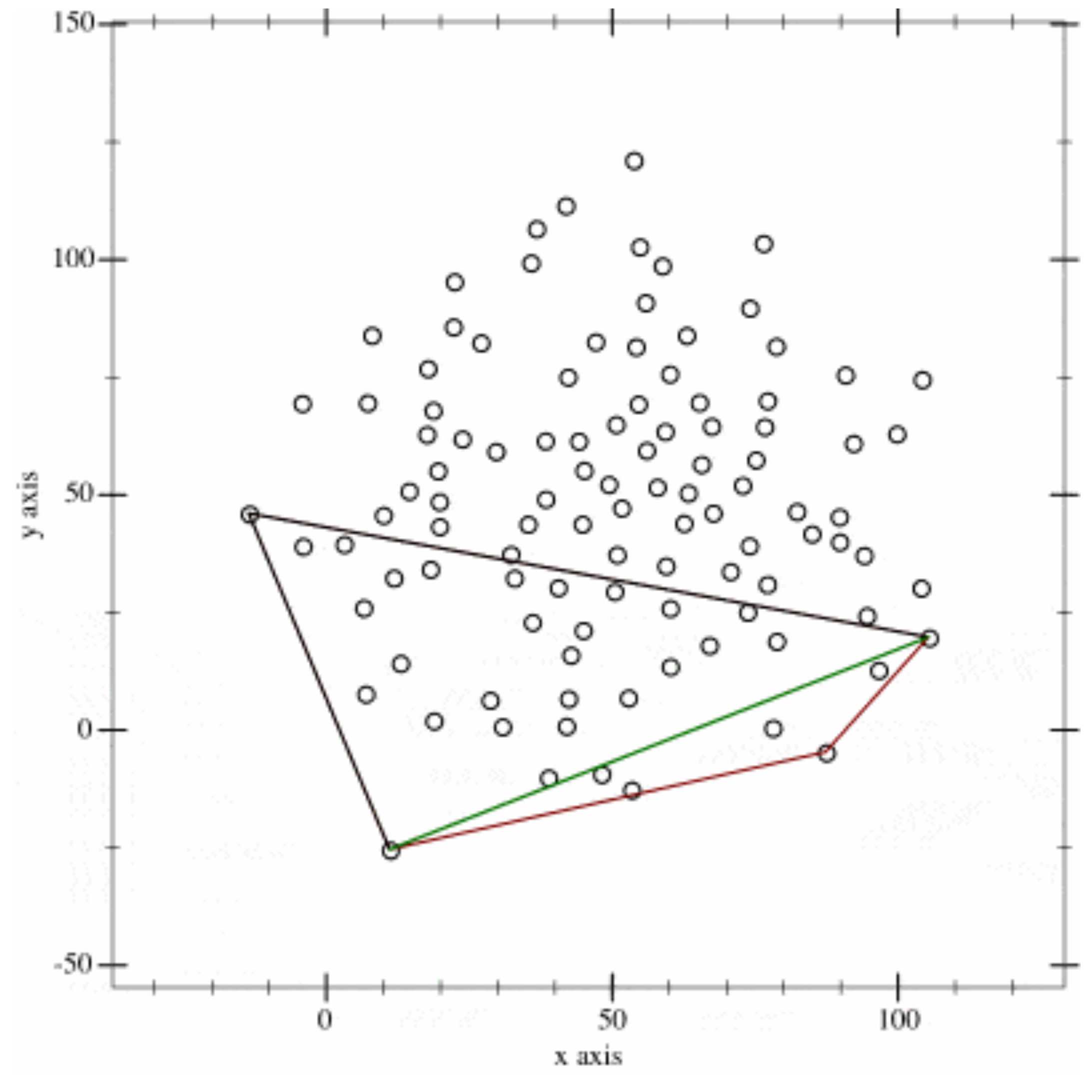
Quick hull example



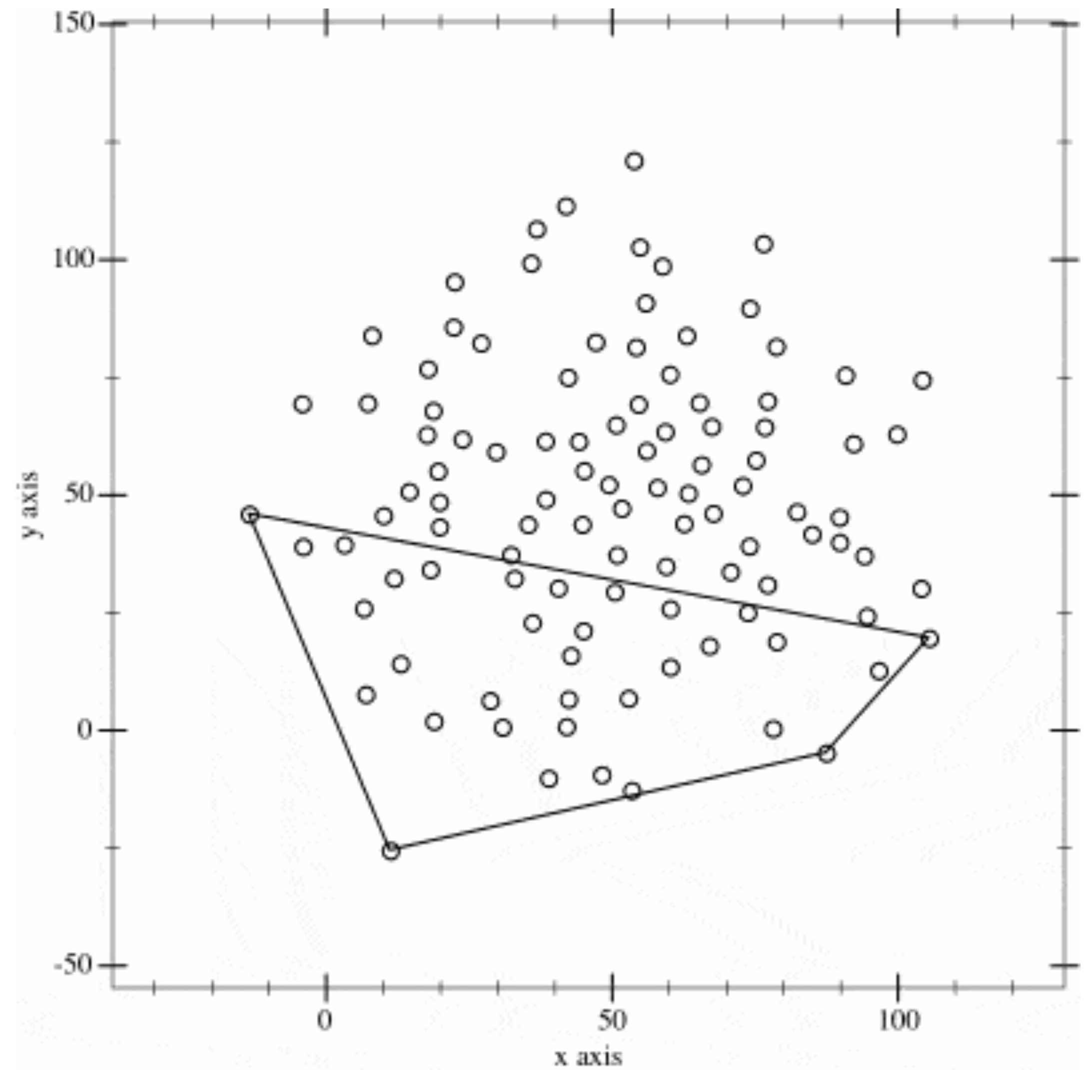
Quick hull example



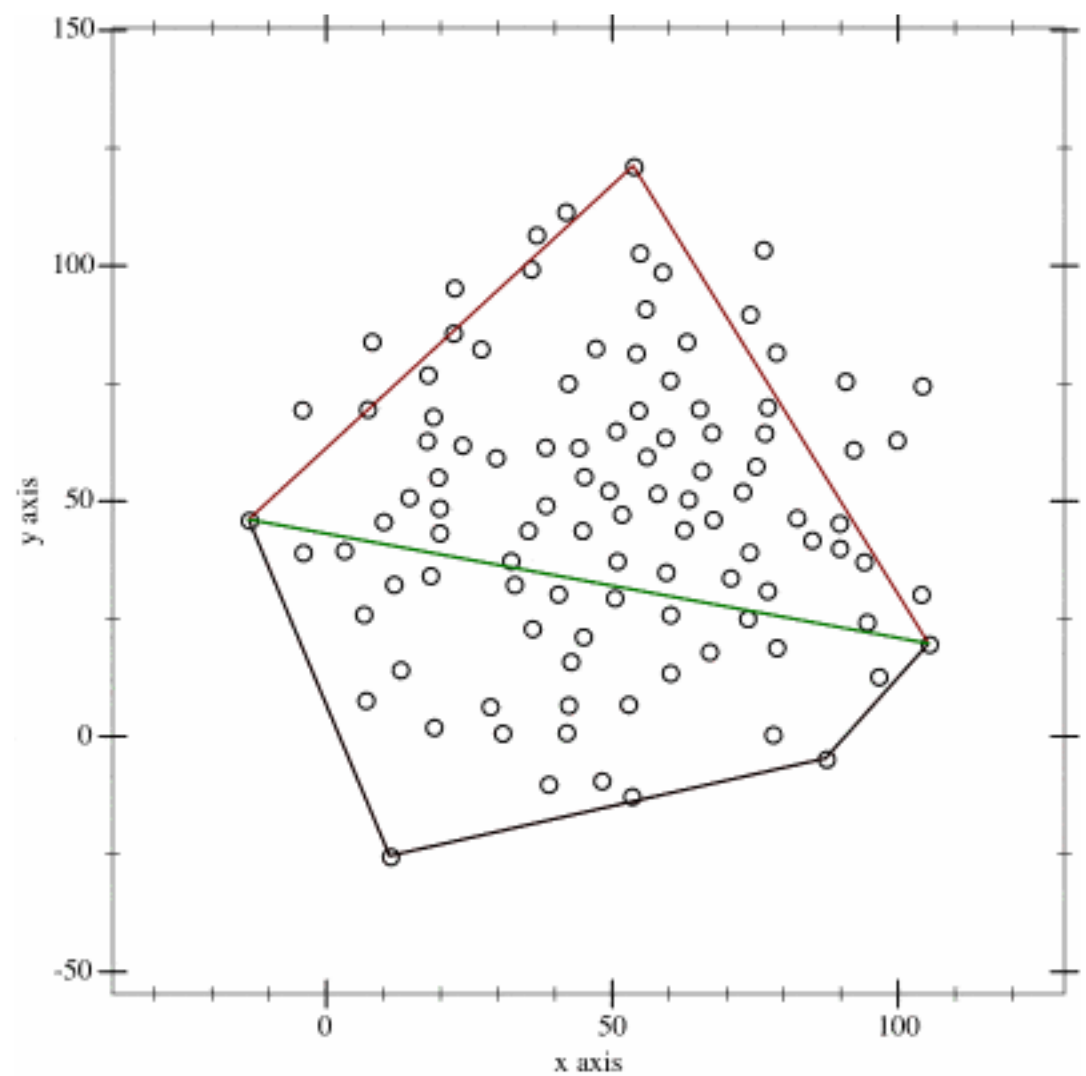
Quick hull example



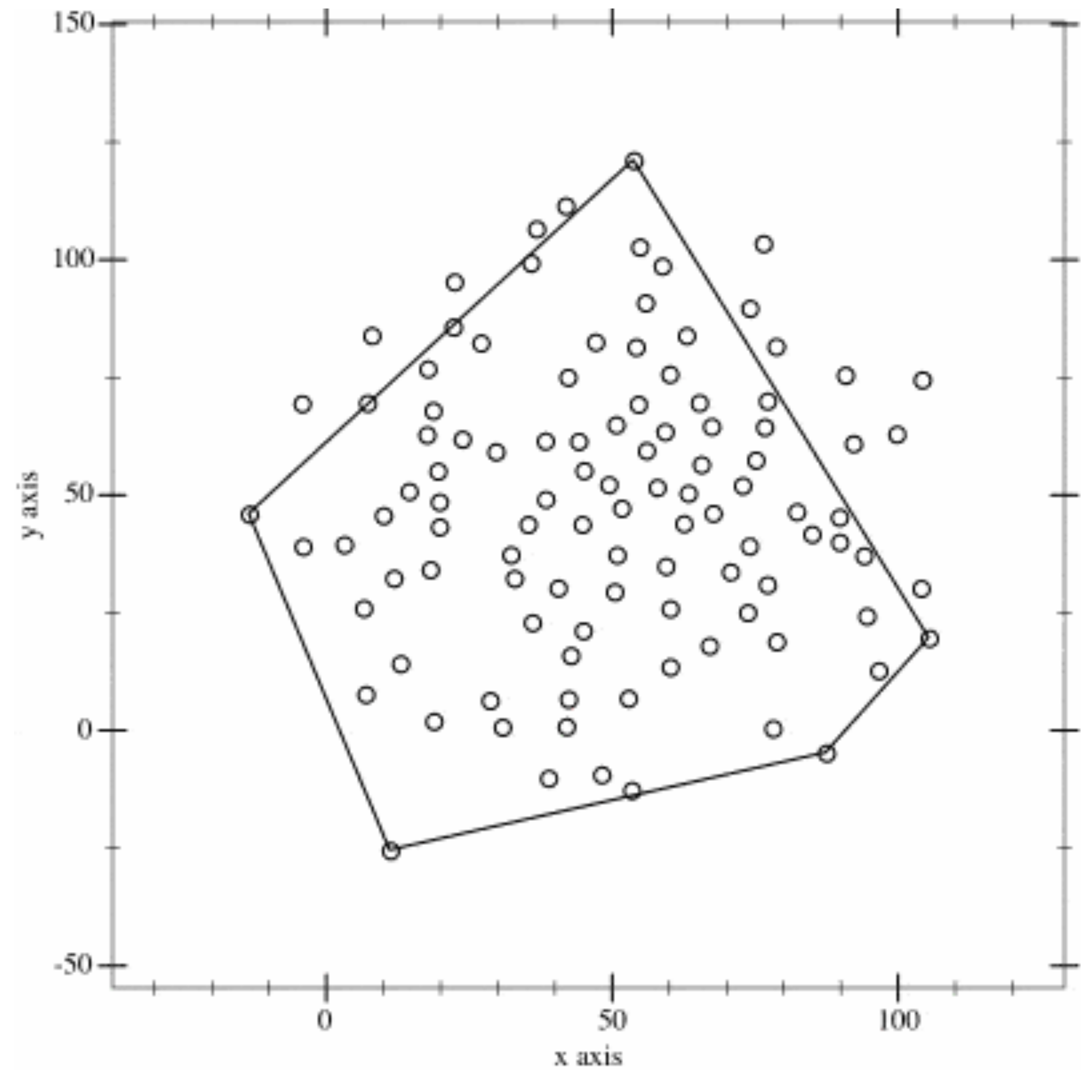
Quick hull example



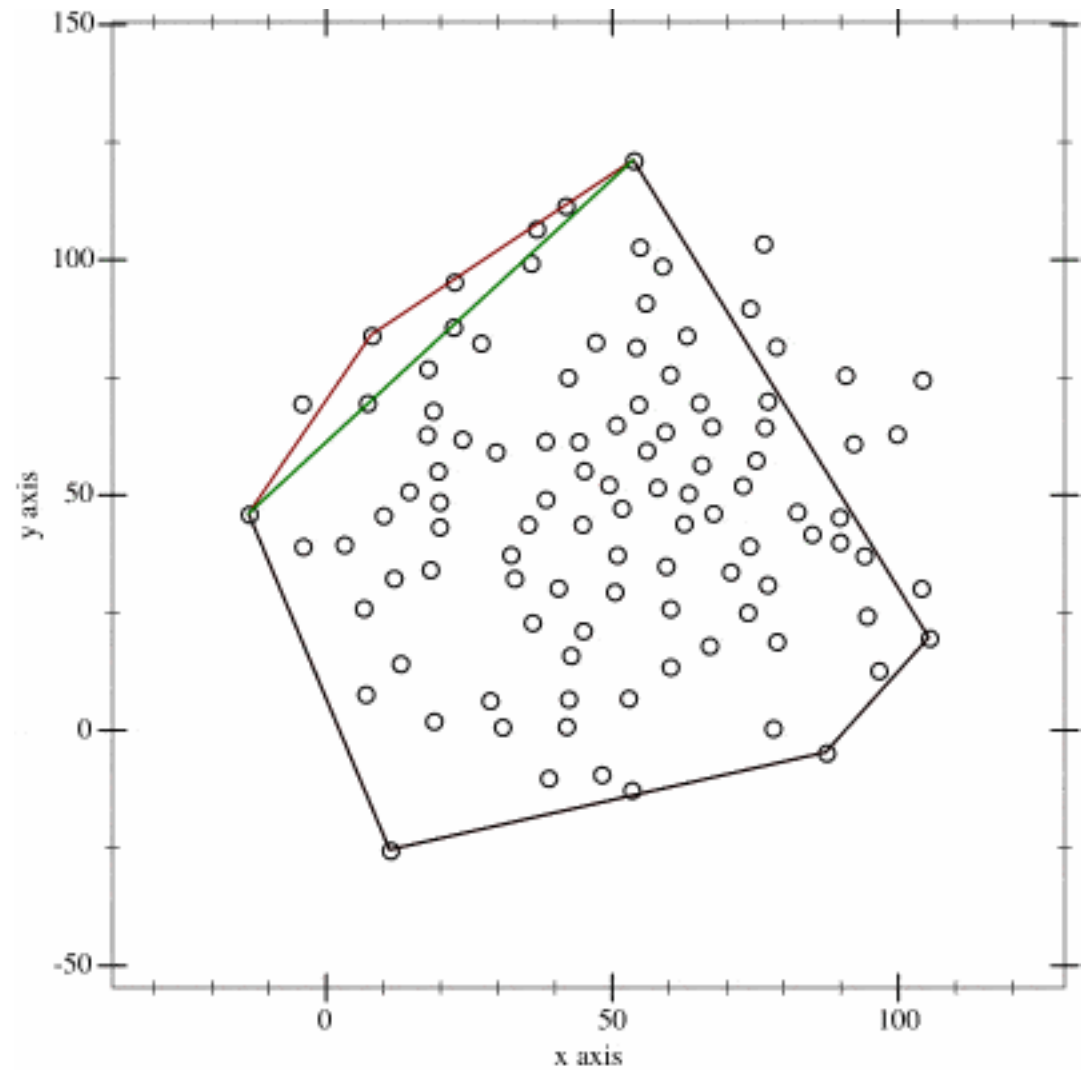
Quick hull example



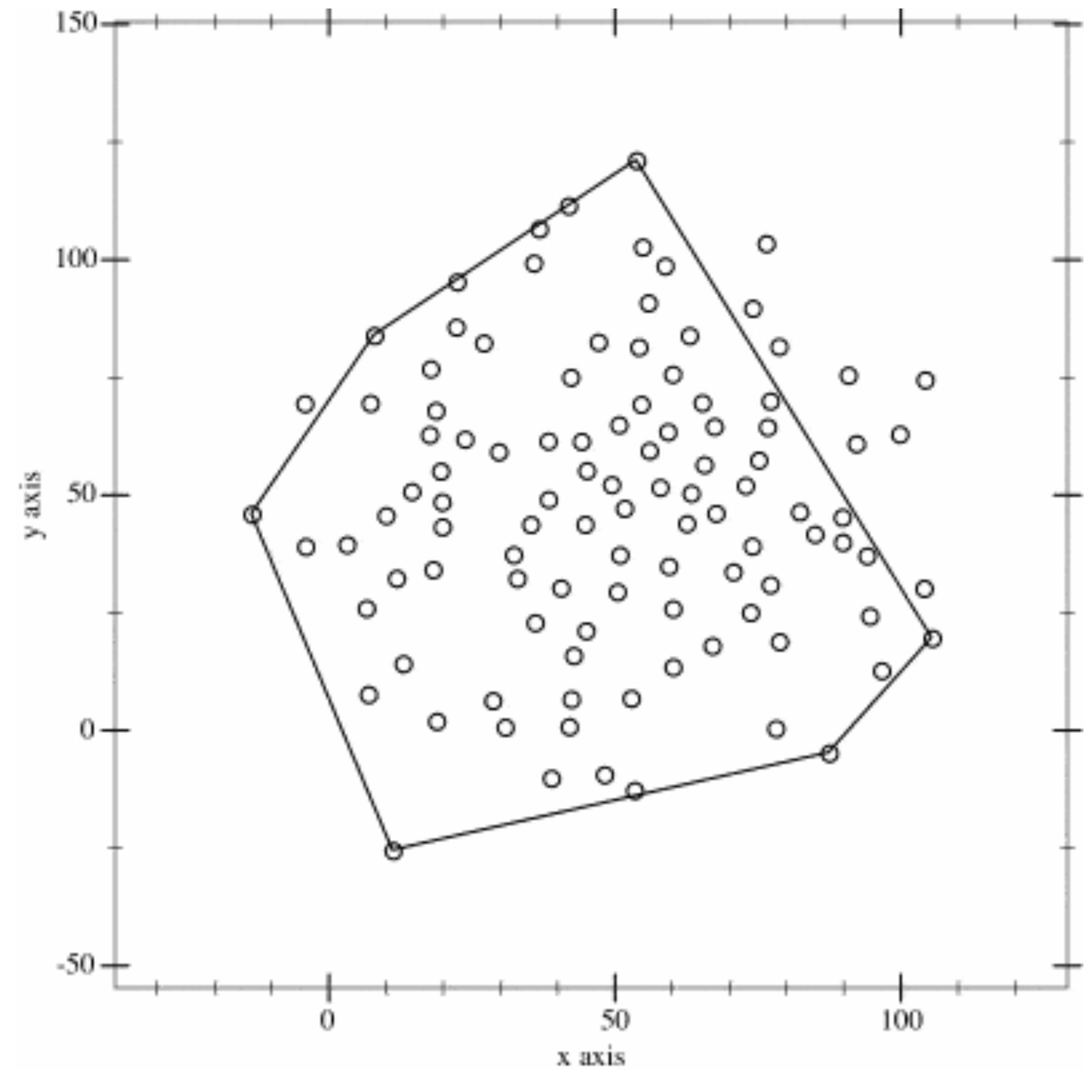
Quick hull example



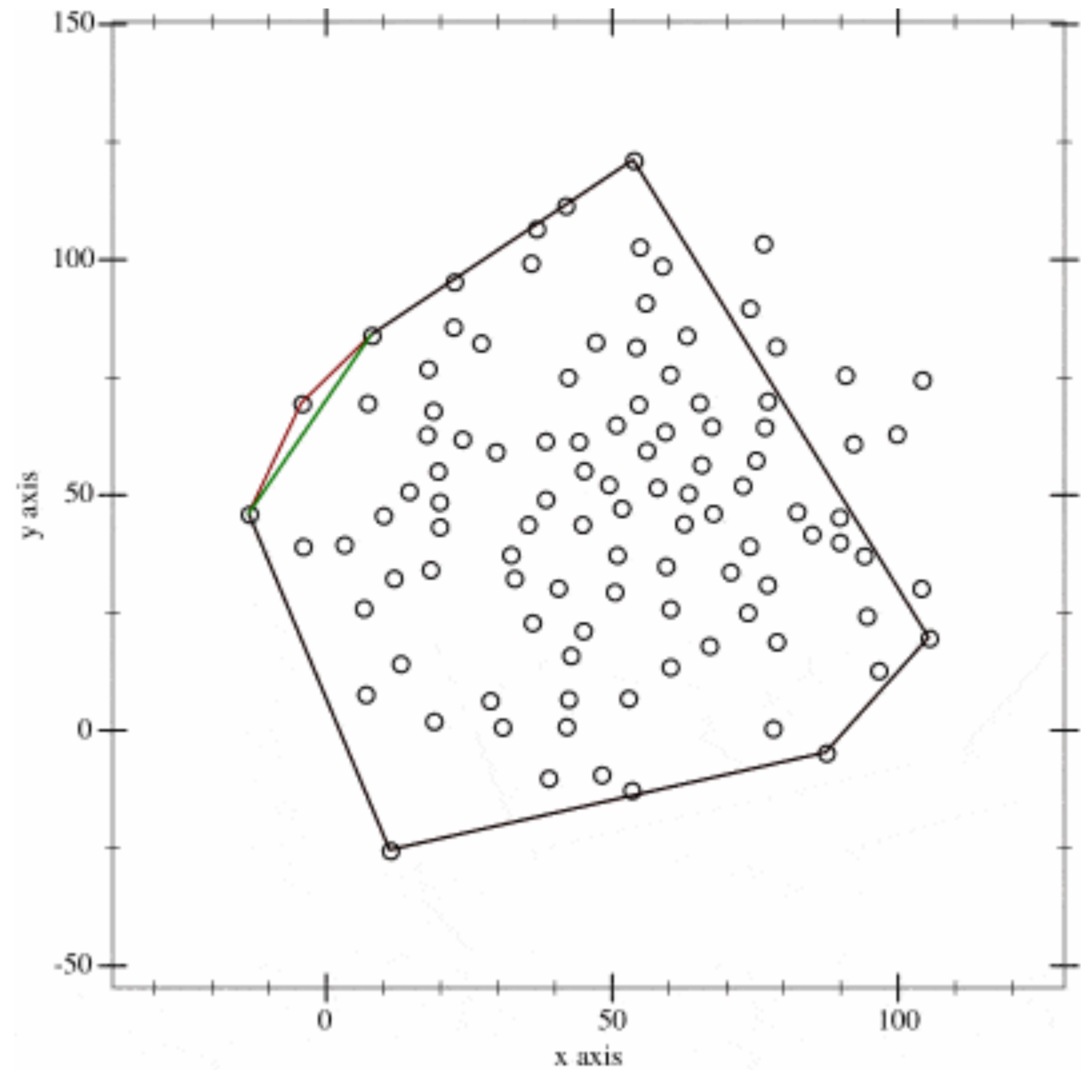
Quick hull example



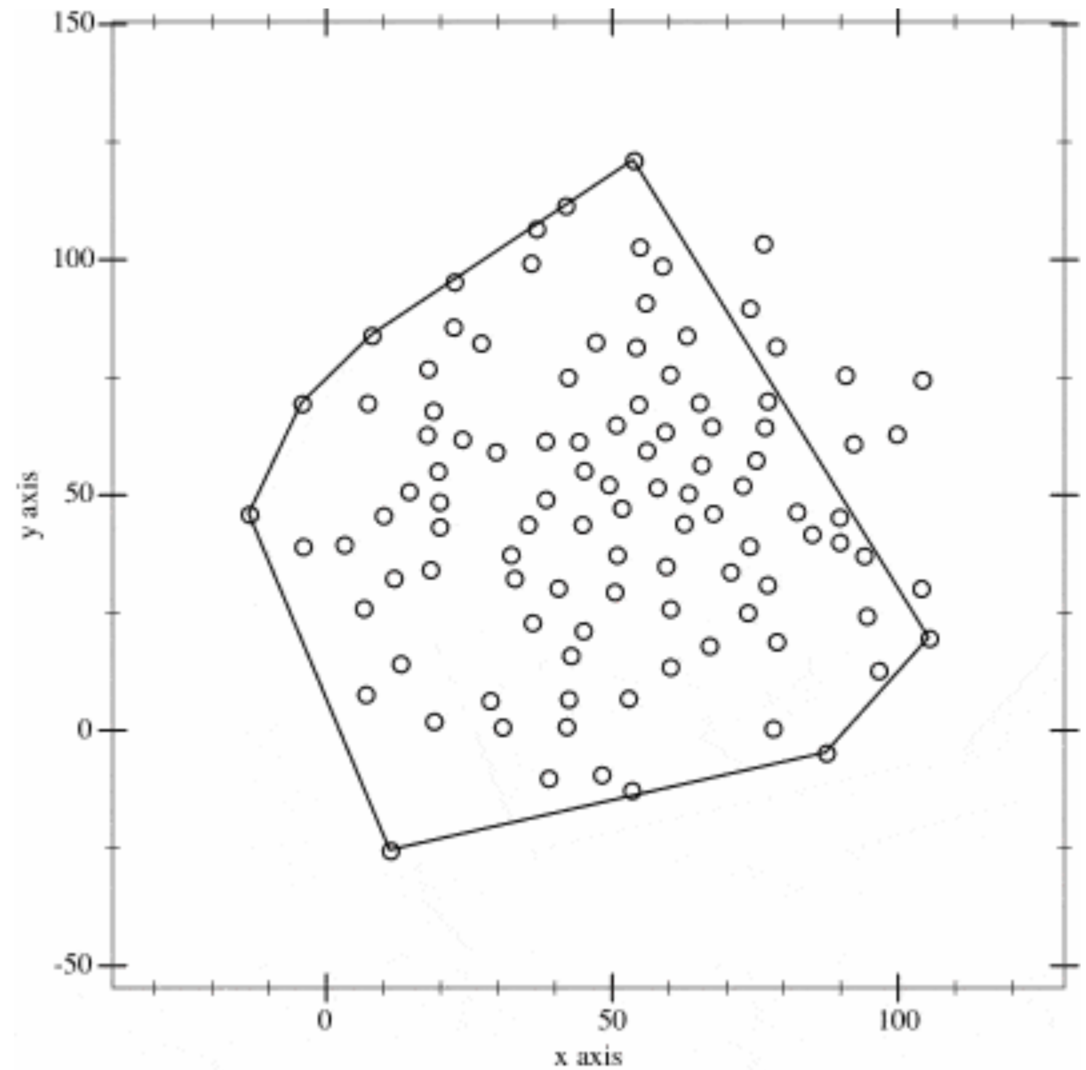
Quick hull example



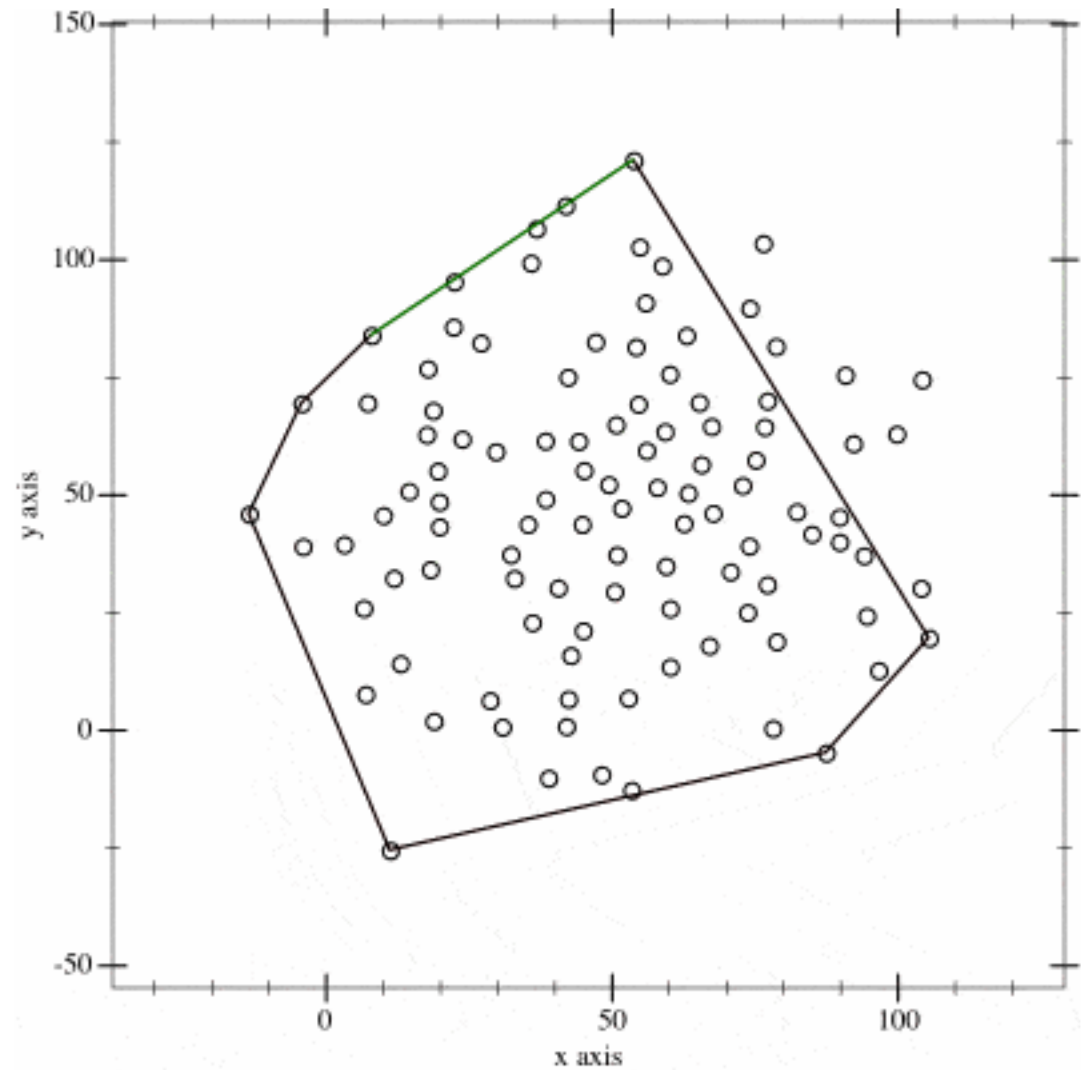
Quick hull example



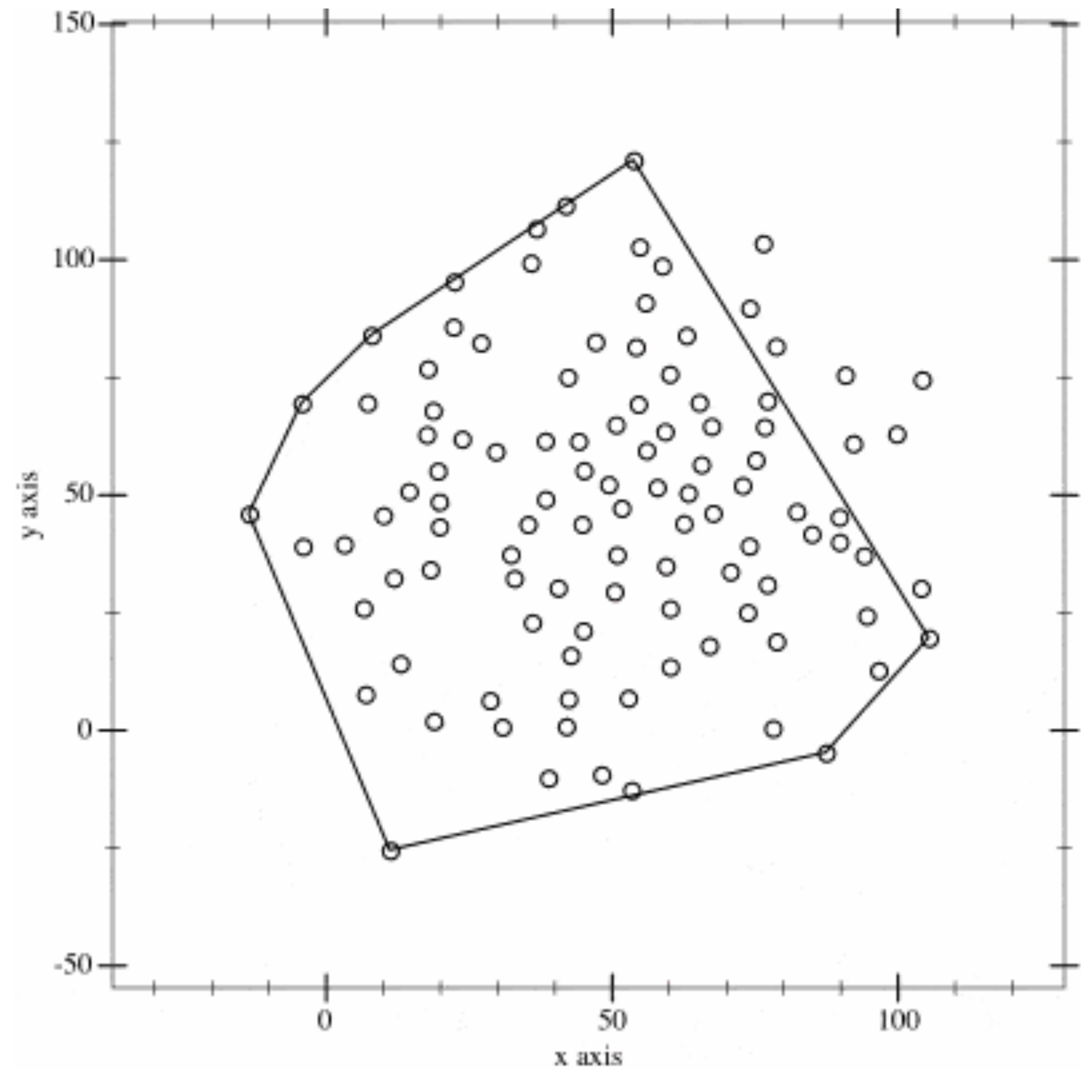
Quick hull example



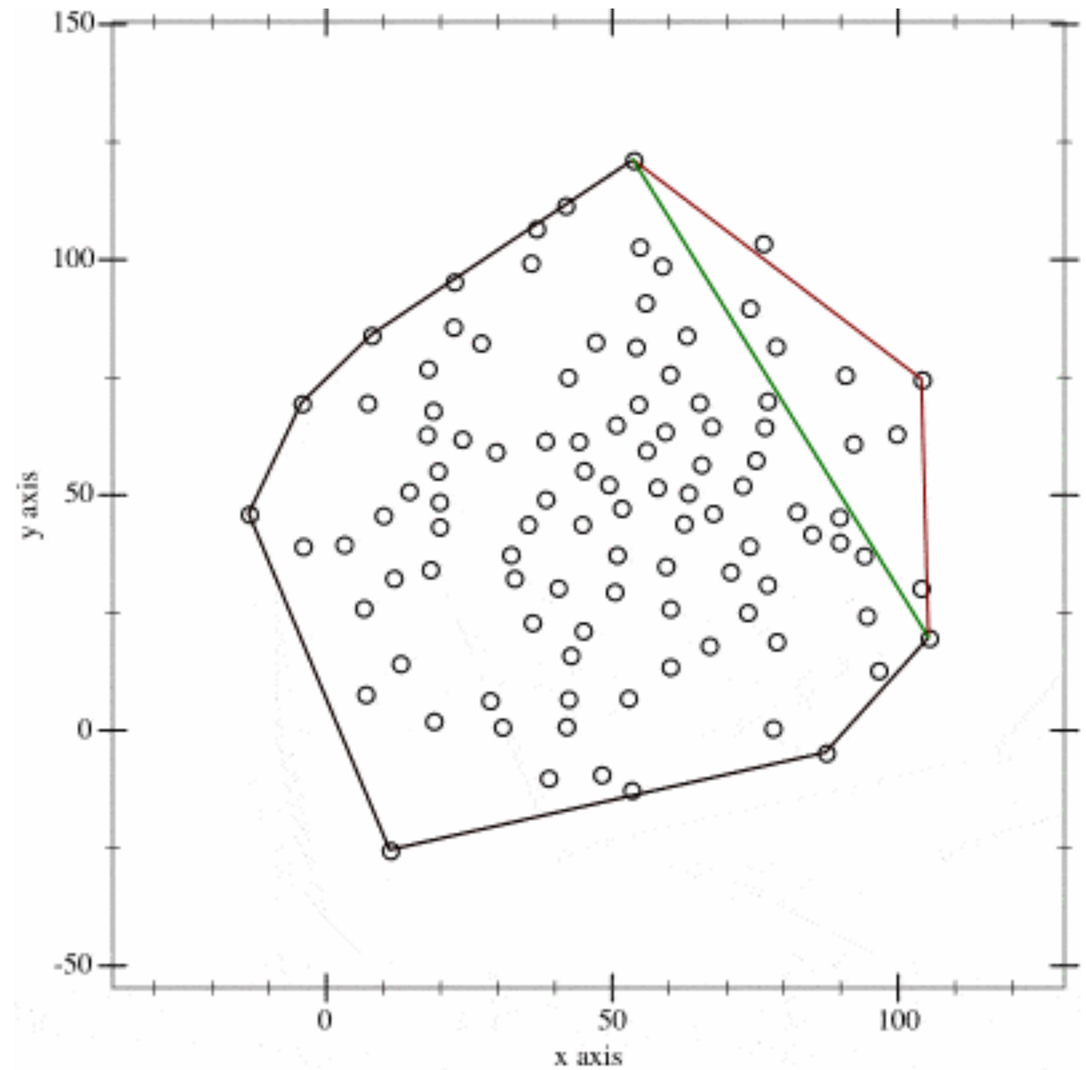
Quick hull example



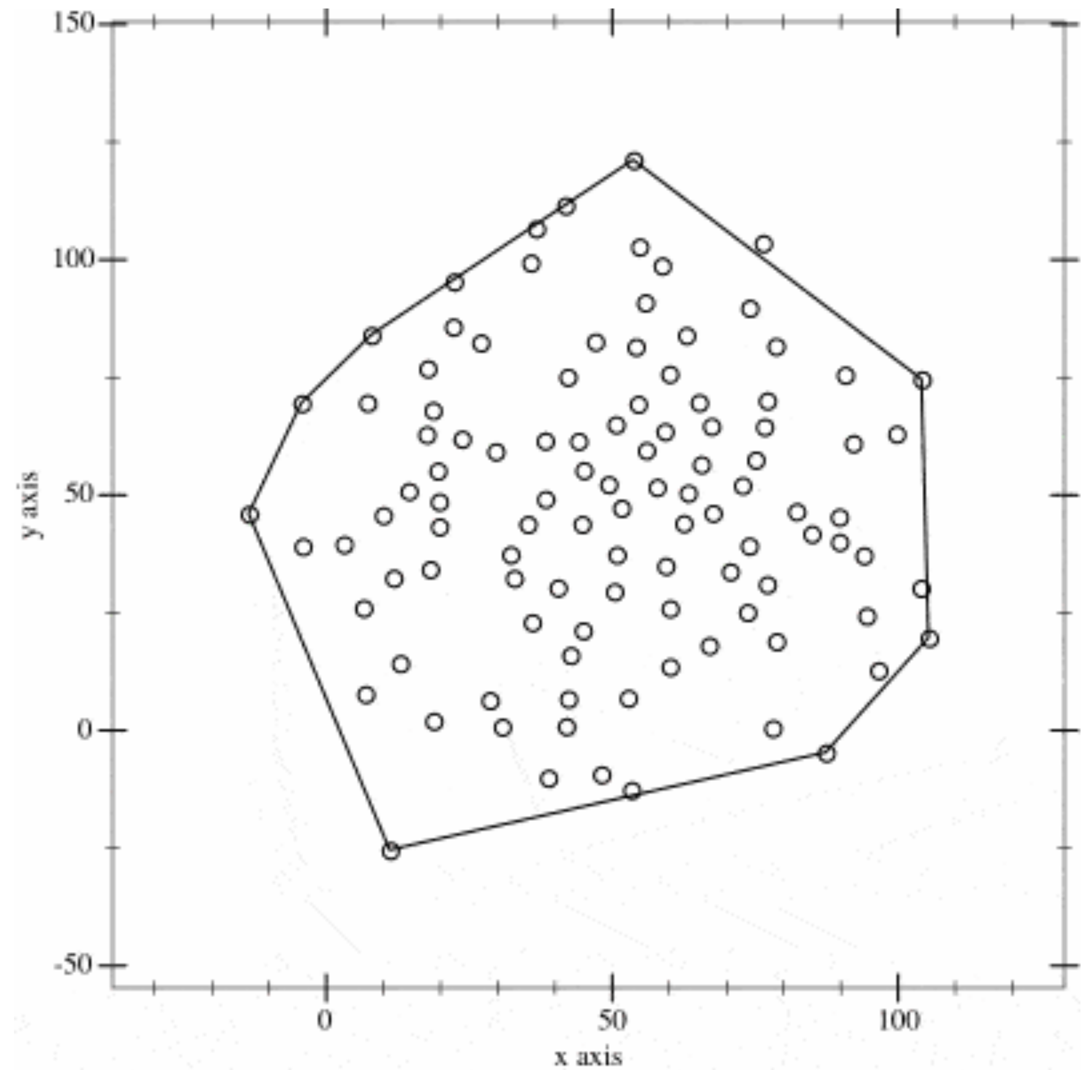
Quick hull example



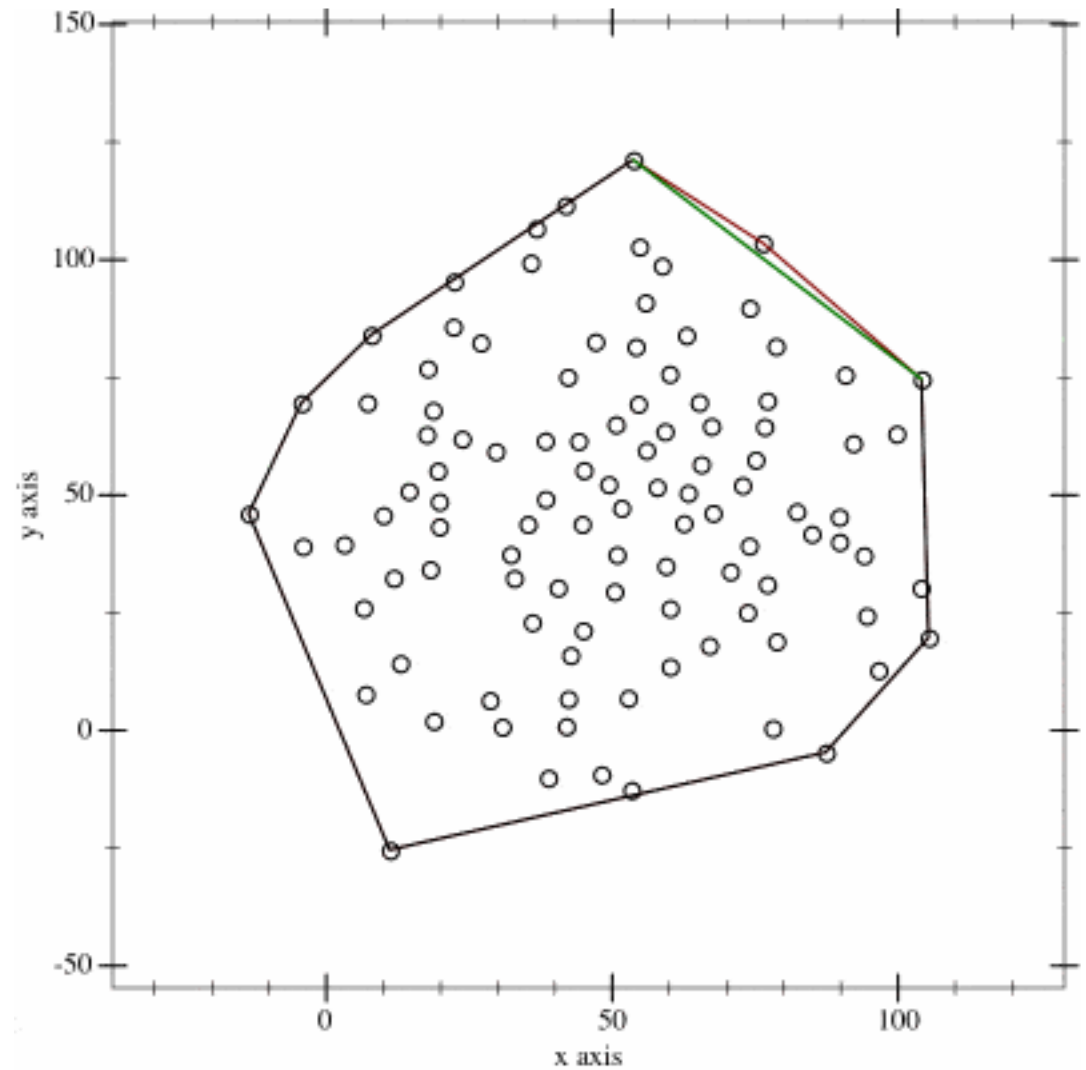
Quick hull example



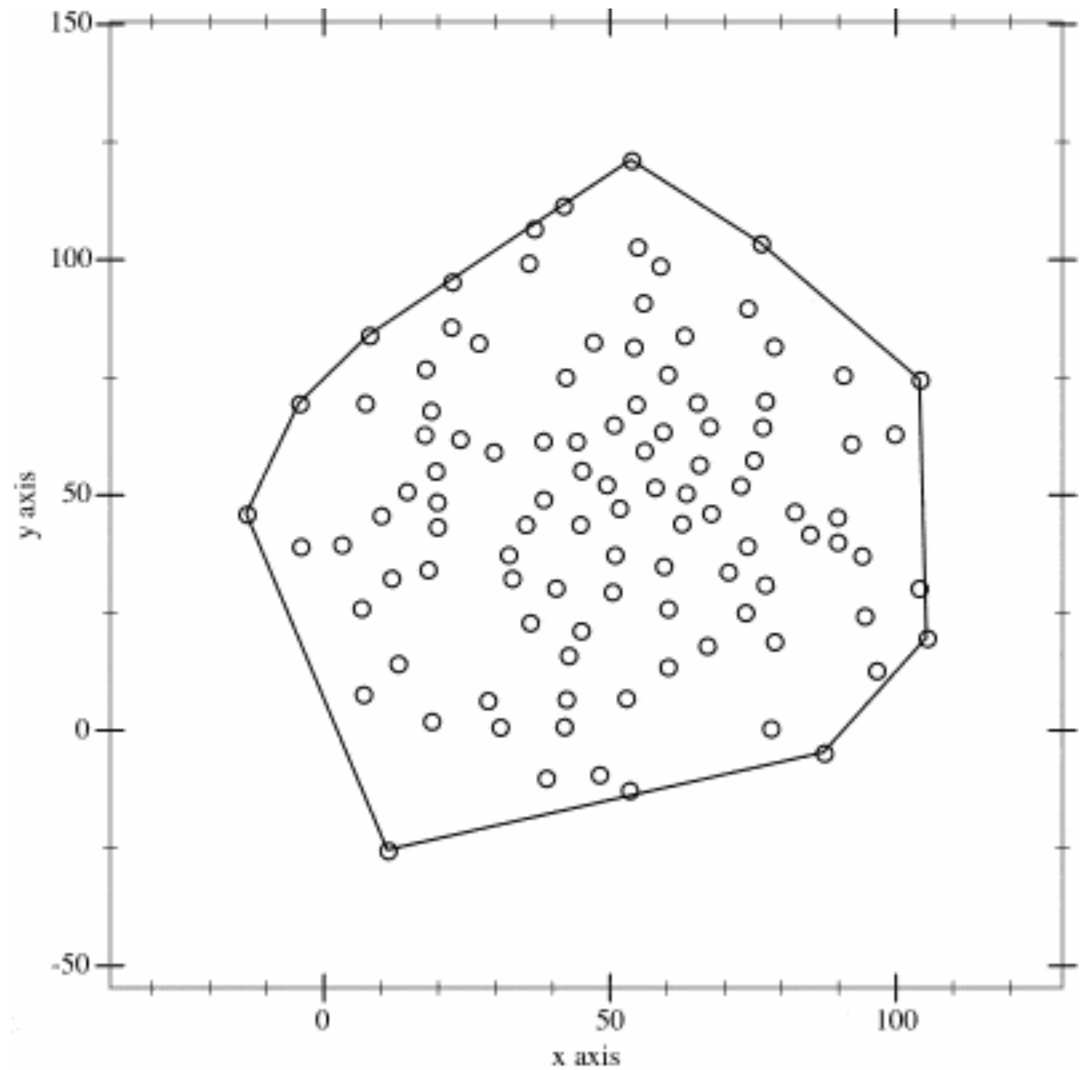
Quick hull example



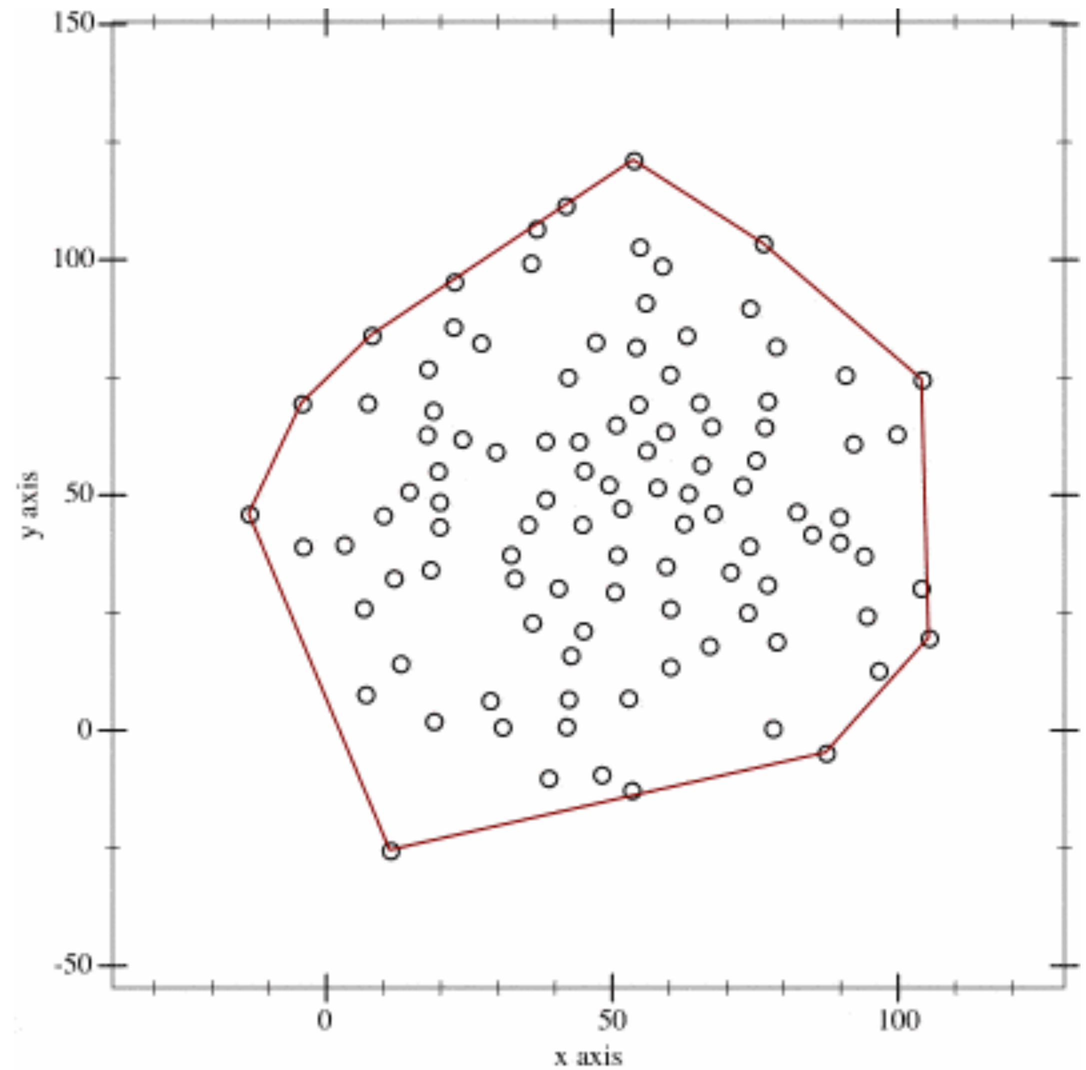
Quick hull example



Quick hull example



Quick hull example



Practice

- Write down the also pseudocode

Solution

- Wiki: https://en.wikipedia.org/wiki/Quickhull

Runtime analysis

- $T(n) = T(n_1) + T(n_2) + O(n)$
  - Worst case $n_1 = n - k$ or $n_2 = n - k$, where k is a small constant (e.g., k=1)
    - $T(n) = O(n^2)$
  - Average case, $n_1 = \alpha n$ and $n_2 = \beta n$, where $\alpha < 1$ and $\beta < 1$
    - $T(n) = O(n \log n)$