Project 2 - Convex Hull

Part 1 - Source Code

```
# This is the method that gets called by the GUI and actually executes
# the finding of the hull
  def compute_hull(self, points, pause, view):                                        # Fn - T O(nlogn) | S(n)
    self.pause = pause
    self.view = view

    assert(type(points) == list and type(points[0]) == QPointF)

    # Sorts the points by increasing x-value
    points.sort(key= lambda k: k.x()) # Overwrites points with same but new order      # T O(nlogn) - Sorting optimized by Python

    t3 = time.time()                                                                  # S O(1)
    polygonPts = self.solve(points)                                                   # n*logn | S O(n) - See fn, 1 array
    polygon = self.toPolygon(polygonPts)                                              # T O(n) | S O(1) - See fn
    t4 = time.time()                                                                  # S O(1)

    self.showTangent(polygon, BLUE);

    # when passing ines to the display, pass a list of QLineF objects.  Each QLineF
    # object can be created with two QPointF objects corresponding to the endpoints
    self.showHull(polygon,RED)
    self.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4 - t3))

    polygon = []

  '''Divide-and-conquer convex hull solver (recursive)'''
  def solve(self, points):                                                                # T n*logn | S O(n) - Technically S 3n
    # Base case
    if len(points) <= 2:  # If 2 points, have array with 1 line connecting the 2 points
      return points

    # Recursive case
    midIdx = len(points) // 2                                                             # O(1) - Big O of n, not N
    leftHull = self.solve(points[:midIdx])  # First half                                 # T O(logn) | S O(n) - 1 array, recursve
    rightHull = self.solve(points[midIdx:]) # Second half                                # T O(logn) | S O(n) - 1 array, recursive
    return self.combine(leftHull, rightHull)                                              # T(n) | S O(n) - See fn, 1 array

  '''Converts array of QPointF points to an array of QLineF lines'''
  def toPolygon(self, points):                                                # Fn - T O(n) | S(1)
    polygon = []                                                              # S O(1)
    for i in range(len(points)):                                              # T O(n) - Number of points
      if i < len(points) - 1:
        polygon.append(QLineF(self.at(points, i), self.at(points, i+1)))      # T O(1) - Appending is constant | S O(n) - need extra space
      else:   # On last iteration connect last point to first
        polygon.append(QLineF(self.at(points, i), self.at(points, 0)))        # T O(1) - Appending is constant | S O(n) - need extra space
    return polygon

  '''
    NOTE
    Hulls are sorted in cw order
    The 1st element in a hull is its leftmost point
  '''

  '''Finds upper/lower tangent of each hull, then returns back indices for the left and right 2 upper/lower tangents.'''
  def findTangentIndices(self, leftHull, rightHull, findLower):                                  # Fn - T O(n) | S(1) - Technically T
    # Find rightmost point of left hullleftmost and rightmost points
    leftCurrIndex = self.getRightmostPtIdx(leftHull)                                             # T O(n) - See fn | S O(1) - 1 int
    rightCurrIdx = LEFTMOST_PT_IDX                                                               # T O(1) | S O(1) - 1 int

    fullyOptimized = False                                                                       # T O(1) | S O(1) - 1 boolean

    while not fullyOptimized:                                                                    # T O(n) - Worst case, looping throug
      fullyOptimized = True

      # Left Side - finding optimal point

      leftNextIdx = leftCurrIndex - 1 # Left hull's next counter-clockwise index
      if findLower:
        leftNextIdx = leftCurrIndex + 1 # Left hull's next clockwise index

      currSlope = self.findSlope(self.at(rightHull, rightCurrIdx), self.at(leftHull, leftCurrIndex))  # T O(1) - See fn | S O(1) - 1 float
      nextSlope = self.findSlope(self.at(rightHull, rightCurrIdx), self.at(leftHull, leftNextIdx))    # T O(1) - See fn | S O(1) - 1 float
```

```
    isBetter = nextSlope < currSlope                                                    # T O(1) | S O(1) - 1 boolean
    if findLower:
      isBetter = nextSlope > currSlope

    if isBetter:
      currSlope = nextSlope
      leftCurrIndex = leftNextIdx
      fullyOptimized = False

    # Right Side - finding optimal point

    rightNextIdx = rightCurrIdx + 1  # Right hull's next clockwise index
    if findLower:
      rightNextIdx = rightCurrIdx - 1 # Right hull's next counter-clockwise index

    currSlope = self.findSlope(self.at(leftHull, leftCurrIndex), self.at(rightHull, rightCurrIdx))  # T O(1) - See fn | S O(1) - 1 float
    nextSlope = self.findSlope(self.at(leftHull, leftCurrIndex), self.at(rightHull, rightNextIdx))  # T O(1) - See fn | S O(1) - 1 float

    isBetter = nextSlope > currSlope                                                    # T O(1)
    if findLower:
      isBetter = nextSlope < currSlope

    if isBetter:
      currSlope = nextSlope
      rightCurrIdx = rightNextIdx
      fullyOptimized = False

  return leftCurrIndex, rightCurrIdx

'''Combines 2 hulls together (returns a list of lines)'''
def combine(self, leftHull, rightHull):                                                 # Fn - T O(n) | O(n) - Technically T
  # Find upper tangent connecting the hulls
  leftUpperIdx, rightUpperIdx = self.findTangentIndices(leftHull, rightHull, False)     # T O(n) - See Fn | S(1) - 2 ints
  # Find lower tangent connecting the hulls
  leftLowerIdx, rightLowerIdx = self.findTangentIndices(leftHull, rightHull, True)      # T O(n) - See Fn | S(1) - 2 ints

  # Combine points together into new hull
  comboHull = []                                                                                                    # S(1)
  comboHull = self.addPoints(comboHull, leftHull, LEFTMOST_PT_IDX, leftUpperIdx) # Top part of left hull to keep         # T O(n), S O(n
  comboHull = self.addPoints(comboHull, rightHull, rightUpperIdx, rightLowerIdx) # Part of right hull to keep            # T O(n), S O(n
  comboHull = self.addLowerLeftPoints(comboHull, leftHull, leftLowerIdx) # Bottom part of left hull to keep             # T O(n), S O(n

  return comboHull

'''Finds slope of the line made by connecting the 2 points passed in'''
def findSlope(self, point1, point2):                                    # T O(1), S O(1)
  line = QLineF(point1, point2)                                         # S O(1) - 1 line variable
  return line.dy() / line.dx()                                          # T O(1) - Typically more expensive, but Python has optimized thi

'''Returns index of item in hull (handles logic so that hull can loop circularly).'''
def at(self, hull, index):                                              # T O(1)
  return hull[index % len(hull)]                                        # T O(1) - Access is constant, mod is constant

'''Adds points from an old hull to a new one'''
def addPoints(self, newHull, oldHull, currIdx, finalIdx):               # Fn - T O(n), S O(n)
  newHull.append(self.at(oldHull, currIdx))                             # T O(1) - Appending is constant | S O(1) - just one item added t
  while (currIdx % len(oldHull)) != (finalIdx % len(oldHull)):          # T O(n) - Size of old hull, mod is constant
      currIdx += 1
      newHull.append(self.at(oldHull, currIdx % len(oldHull)))          # T O(1) - Appending is constant, mod is constant | # S O(n) - ne

  return newHull

'''Handles edge case for adding points to the lower left of a hull'''
def addLowerLeftPoints(self, newHull, leftHull, leftLowerIdx):          # Fn - T O(n), S O(n)
  if len(leftHull) == 1:                                                # T O(1)
    return newHull

  while (leftLowerIdx % len(leftHull)) != LEFTMOST_PT_IDX:                          # T O(n) - Size of left hull
    newHull.append(self.at(leftHull, leftLowerIdx))                     # T O(1) - Appending is constant | S O(n) - need extra space for
    leftLowerIdx += 1
  return newHull

'''Returns the index of the rightmost point in hull'''
def getRightmostPtIdx(self, hull):                      # Fn - T O(n)
```

Part 2 - Time & Space Complexity

In code, `Fn` refers to function complexity

In code, comments with `T O(...)` are time complexities, then comments with `S O(...)` are space complexities.

If no time complexity is provided next to a line of code, it is `O(1)` constant time.

If no space complexity is provided next to a line of code, it doesn't cost any space.

Theoretical Analysis

Solving Complexity using Recurrence Relations / Master Theorem

$$T(n) = aT(\frac{n}{b}) + O(n^d)$$

n = n, a (branching factor) = 2, b (reduction factor) = 2, d = 1

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

The branching factor is 2 because with every iteration you go through, you increase the number of branches in the tree by 2 (more and more lines, factor of 2 every time). The reduction factor is 2 because with every iteration the left and right hulls are reduced by a factor of 2. The pre and post work complexity is n, as seen by the complexity for the `combine()` function detailed below.

$$r = \frac{a}{b^d}, r = \frac{2}{2^1} = \frac{2}{2} = 1$$

$$\frac{a}{b^d} = 1, \text{so } T(n) = O(n^d logn) \text{ , or } T(n) = O(nlogn)$$

Therefore, the algorithm's time complexity is

$$O(nlogn)$$

> 💡 For details line by line on complexity of functions, see src code

Individual Function Analysis

`compute_hull` overall complexity

- Time (`T`) - $O(nlogn)$
  - Calls the solve function which has $O(nlogn)$ complexity. The overall algorithm is $O(nlogn)$ because it gets halved every time (logn), then n because of costs to recombine the array back.
- Space (`S`) - $O(n)$
  - While several extra integers and variables are made/used in the program aside from parameters, those are negligible in Big O compared to storing arrays, which takes $O(n)$ space.

`solve()` complexity

- Time (`T`) - $O(nlogn)$
  - The function itself is technically n (for combine fn), since it gets called recursively, cutting it in half every time, it's $O(nlogn)$.
- Space (`S`) - $O(n)$

Requires 3 arrays (left hull, right hull, and combined hull) which take `n` space a piece, which comes out to $O(n)$ in Big O complexity.

`toPolygon()` complexity

Time (`T`) - $O(n)$

The function loops through all the points as it converts them to lines, so that looping through the entire array costs $O(n)$.

Space (`S`) - $O(n)$

One array is made that stores all the new lines, which comes out to $O(n)$ in Big O complexity.

`findTangentIndices` complexity

Time (`T`) - $O(n)$

The function calls `getRightMostPtIdx()` which is $O(n)$, and it also has to optimize slope for upper/lower tangent, which at worst will also be $O(n)$, since looping through each half of the left and right hulls would be n/2 for each. Therefore even though it's 2n, it comes out to $O(n)$ in Big O complexity.

Space (`S`) - $O(1)$

Just requires space for integers and booleans, so $O(1)$.

`combine()` complexity

Time (`T`) - $O(n)$

The function calls `findTangentIndices()` , `addPoints()` , and `addLeftLowerPoints()`, all which take $O(n)$. So even though it's 3n, comes out to $O(n)$ in Big O complexity.

Space (`S`) - $O(n)$

Just requires the space to store the new array, which is $O(n)$ in Big O complexity.

`findSlope()` complexity

Time (`T`) - $O(1)$

Just performs an assignment and division which is constant.

Space (`S`) - $O(1)$

Just stores a line, which is linear (since it's just a single line)

`at()` complexity

Time (`T`) - $O(1)$

Just performs an access and a mod, both which are constant.

Space (`S`) - $O(1)$

No extra space, just returns

`addPoints()` complexity

Time (`T`) - $O(n)$

The function does a while loop which at worst goes through the entire n size of the old hull, so its $O(n)$ in Big O complexity.

Space (`S`) - $O(n)$

Need extra space for every n insertion into the array

`addLowerLeftPoints()` complexity

Time (`T`) - $O(n)$

The function does a while loop which at worst goes through the entire n size of the left hull, so its $O(n)$ in Big O complexity.

Space (`S`) - $O(n)$

Need extra space for every n insertion into the array

`getRightmostPtIdx()` complexity

Time (T) - $O(n)$

The max function at worst will be $O(n)$ since it might have to loop through all items to find the one with the biggest value (Python's implementation may be optimized though).

Space (S) - $O(1)$

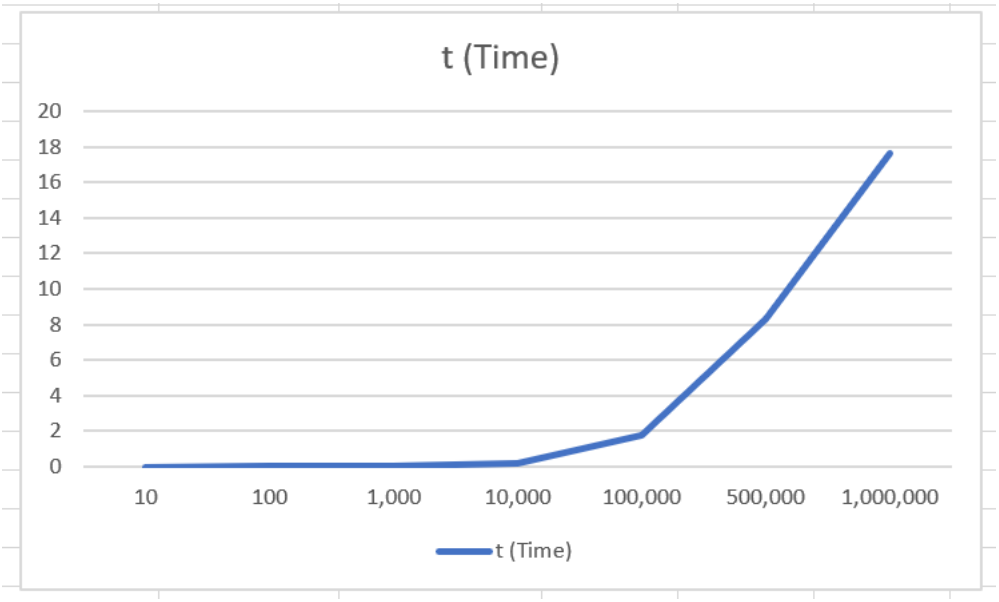
Just stores an int for the biggest x val, so it's linear.

Part 3 - Experimental Outcomes

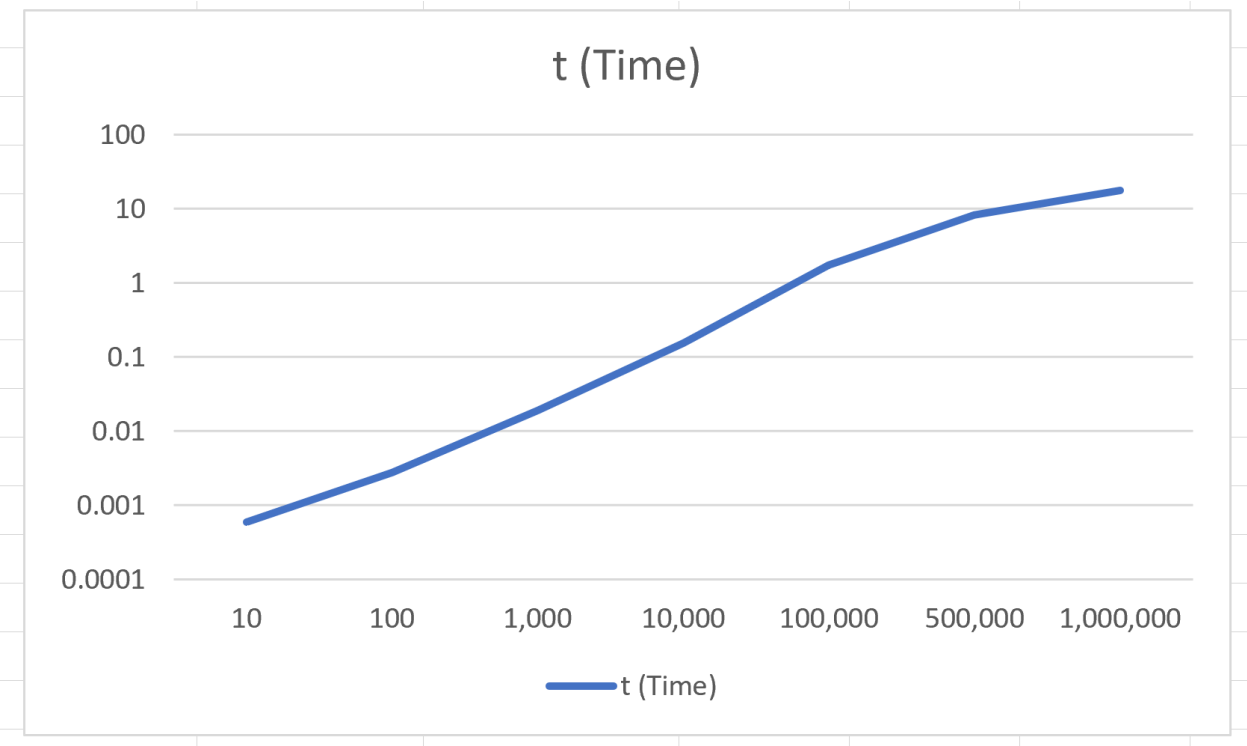
Raw & Experimental Outcomes

	A	B	C	D	E	F	G	H
1	n	10	100	1,000	10,000	100,000	500,000	1,000,000
2								
3	5 Tests	0.000	0.003	0.015	0.157	1.753	8.373	17.735
4	(Elapsed	0.001	0.002	0.015	0.154	1.728	8.348	17.623
5	Time)	0.001	0.003	0.026	0.156	1.726	8.279	17.886
6		0.000	0.003	0.024	0.151	1.731	8.351	17.831
7		0.001	0.003	0.016	0.155	1.804	8.162	17.238
8								
9	Avg	0.0006	0.0028	0.0192	0.1546	1.7484	8.3026	17.6626
10								

Normal Graph



Logarithmic Graph



With a logarithmic graph, each time you go up by 1 y value, you're going up by a factor of 10. Because the y-axis is labelled exponentially (10^n) instead of linearly, the shape of the graph will be altered. It will essentially stretch your function, thus flattening the curve. The differences here are very apparent in our graph because our function is $O(nlogn)$, so by having it graphed across an logarithmic scale, it basically negates the log and the graph more closely resembles a linear $O(n)$ function.

The logarithmic graph fits the function better than the linear graph for the reasons stated above (since the line it makes is straighter, it's fitting it better and giving a better approximation of values). Also like mentioned above, the $O(nlogn)$ order of growth fits it best, since every time the n values increase by a factor of 10, so do the y values (the slope is 1). In fact, the line would be even straighter if we threw out the data for n = 500,000, since it doesn't follow the same formula as the rest of the x values (10^x tick). $10 = 10^1, 100 = 10^2, ..., 100000 = 10^5, 1000000 = 10^6$, then $500000 = 10^5.69...$, which doesn't follow the trend.

As far as the **constant of proportionality** k

$k = y/x$ is the general formula, but for this function we'd find k by using the follwing equations:

General equation

$$k = \frac{t}{n}$$

Exponential equation

$$y = ka^x$$

For our case, a = 10 (since that's the base that our x and y ticks are exponentiated from), x=x, y=y, so then $k = 1.$ This is because by doing 10 to the power of x, we were already getting the according y value, so we don't need to multiply by anything else to make the 2 sides equivalent.

Here's some examples to solidfy our case:

$$10 = 1 * 10^1, 100 = 1 * 10^2, 1000 = 1 * 10^3, ...$$

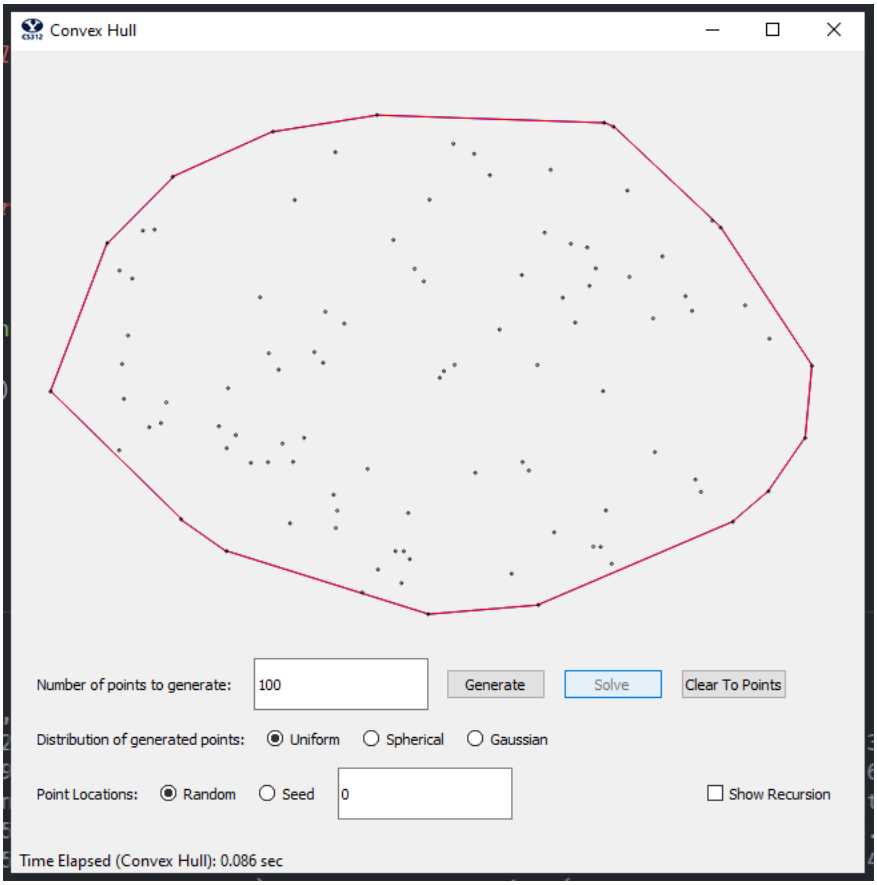
Part 4 - Theoretical & Empirical Analyses Observations

With the theoretical analysis (using recurrence relations and the master theorem) I came to the conclusion that the algorithm was $O(nlogn)$. Then with empirical analysis (through my experiments on the different n tests and the outputs), I also saw that it lined up well with that Big O complexity from the master theorem, since the slope was steeper than a standard logn function, yet not as steep as n^2. Especially when I plotted the logarithmic graph, my findings became clear that it was correct, because the nlogn function's logn part essentially became negated with a y axis that increased by a factor of 10 (since log is base 10, increasing by a factor of 10 countered it). I knew it was negated because the line flattened out and looked more like an n line (nlogn without logn).

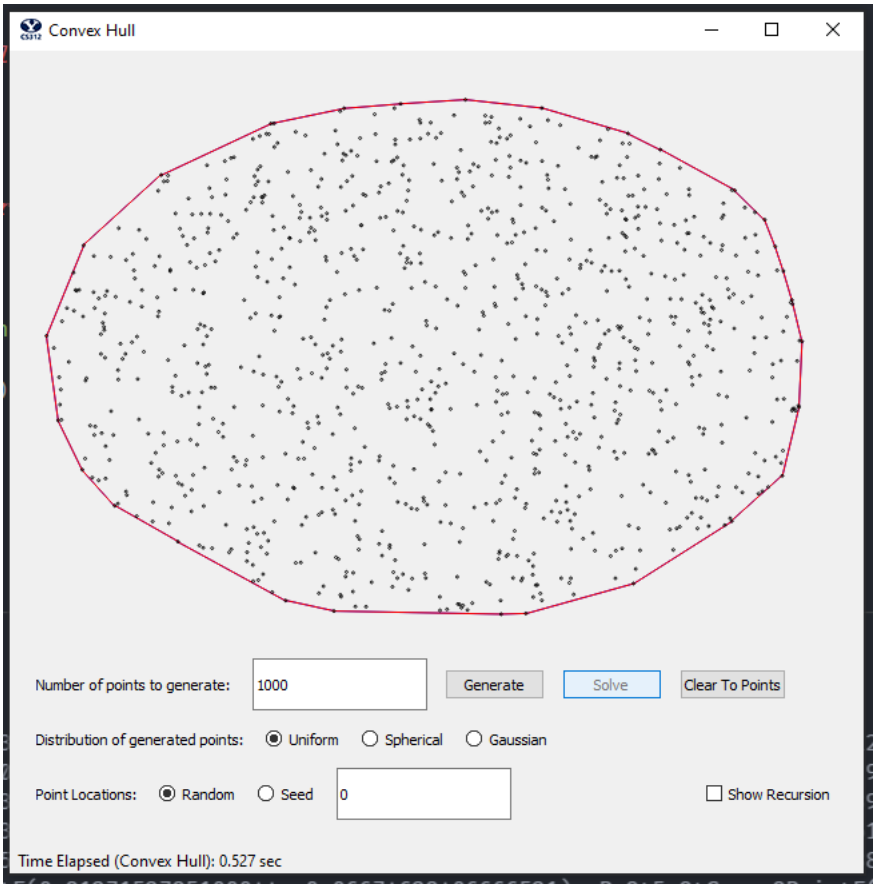
Obviously there were slight differences seen, as my test results didn't full follow a perfect nlogn function, but the results were close, at least in a Big O magnitude. The differences looked a bit more distinct on the graph because of the x tick with the value of 500,000 since that threw off the x ticks incrementing by a factor of 10 each time, making my logarithmic graph look less linear at the end over that x value. All in all though, the results were quite complimentary.

Part 5 - Screenshots

100 Points



1,000 Points


Convex Hull
Number of points to generate:
1000
Generate
Solve
Clear To Points
Distribution of generated points:
Uniform
Spherical
Gaussian
Point Locations:
Random
Seed
0
Show Recursion
Time Elapsed (Convex Hull): 0.527 sec