Project 2 - Convex Hull

Austin McKamey
CS312
Sep 26, 2022

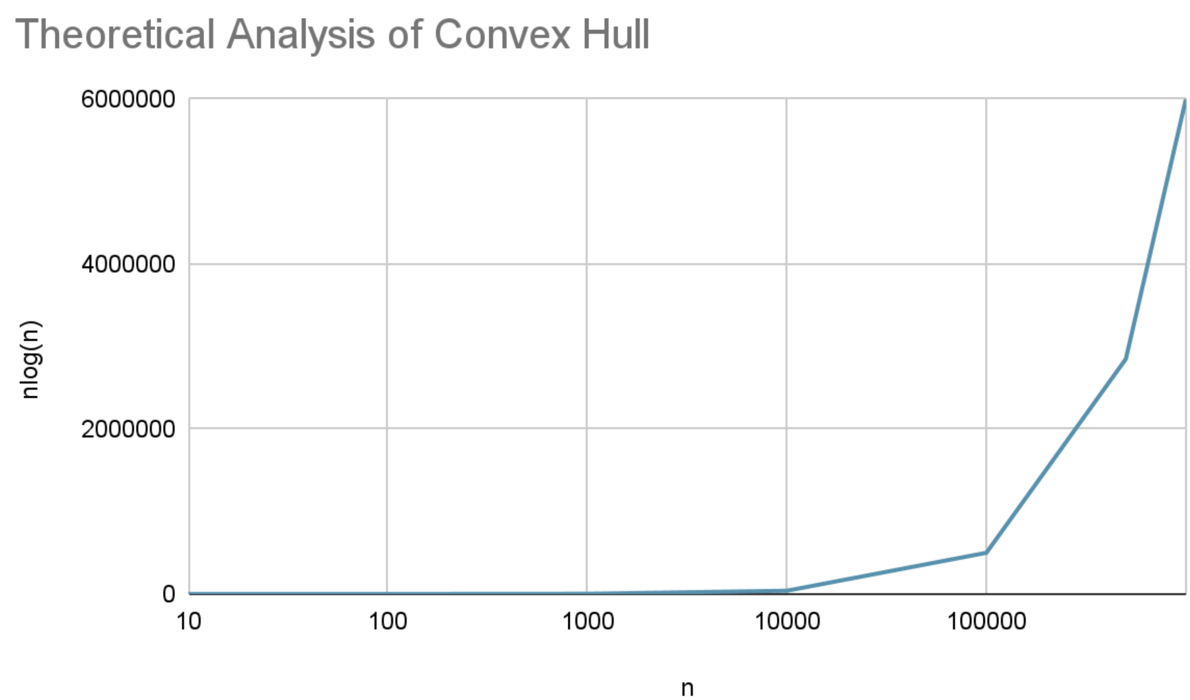
1.

```
# 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):
  self.pause = pause
  self.view = view
  assert( type(points) == list and type(points[0]) == QPointF )


  t1 = time.time()
  sorted_points = sorted(points, key= lambda pt: pt.x())
  t2 = time.time()

  t3 = time.time()
  hull = self.recurse_hull(sorted_points, pause, view)
  polygon = [QLineF(hull[i], hull[(i + 1) % len(hull)]) for i in range(len(hull))]
  t4 = time.time()

  self.showHull(polygon,RED)
  self.showText('Time Elapsed (Convex Hull): {:3.3f} sec'.format(t4-t3))

def recurse_hull( self, points, pause, view):
  l = len(points)
  h = l // 2
  if(l == 1):
     return points
  left = self.recurse_hull(points[:h], pause, view)
  right = self.recurse_hull(points[h:], pause, view)
  hull = self.combine_hull(left, right, pause)
  if pause:
     self.showHull([QLineF(hull[i], hull[(i + 1) % len(hull)]) for i in
range(len(hull))], RED)
     self.view.clearLines([QLineF(left[i], left[(i + 1) % len(left)]) for i in
range(len(left))])
     self.view.clearLines([QLineF(right[i], right[(i + 1) % len(right)]) for i in
range(len(right))])
  return hull

def combine_hull( self, left, right, pause):

  # clockwise is increasing index, counterclockwise is decreasing
  p = max(left, key=lambda pt: pt.x())
```

```
  q = min(right, key=lambda pt: pt.x())

  cp_p = p
  cp_q = q

  prev_p = None
  prev_q = None
  while (True):
     prev_p = p
     prev_q = q
     if pause:
        self.blinkTangent([QLineF(p, q)], BLUE)
     while self.turn_direction(q, p, self.counter_clockwise_point(p, left)) > 0:
        p = self.counter_clockwise_point(p, left)
        if pause:
           self.blinkTangent([QLineF(p, q)], BLUE)
     while self.turn_direction(p, q, self.clockwise_point(q, right)) < 0:
        q = self.clockwise_point(q, right)
        if pause:
           self.blinkTangent([QLineF(p, q)], BLUE)
     if p == prev_p and q == prev_q:
        break
  if pause:
     self.showTangent([QLineF(p, q)], GREEN)
  upper_tan_p = p
  upper_tan_q = q

  prev_cp_p = None
  prev_cp_q = None
  while (True):
     prev_cp_p = cp_p
     prev_cp_q = cp_q
     if pause:
        self.blinkTangent([QLineF(cp_p, cp_q)], BLUE)
     while self.turn_direction(cp_q, cp_p, self.clockwise_point(cp_p, left)) < 0:
        cp_p = self.clockwise_point(cp_p, left)
        if pause:
           self.blinkTangent([QLineF(cp_p, cp_q)], BLUE)
     while self.turn_direction(cp_p, cp_q, self.counter_clockwise_point(cp_q,
right)) > 0:
        cp_q = self.counter_clockwise_point(cp_q, right)
        if pause:
           self.blinkTangent([QLineF(cp_p, cp_q)], BLUE)
     if cp_p == prev_cp_p and cp_q == prev_cp_q:
        break
  if pause:
     self.showTangent([QLineF(cp_p, cp_q)], GREEN)
```

```
  lower_tan_p = cp_p
  lower_tan_q = cp_q

  result = []

  pt = lower_tan_p
  result.append(lower_tan_p)
  while(True):
     if lower_tan_p == upper_tan_p:
        break
     pt = self.clockwise_point(pt, left)
     if not pt == lower_tan_p:
        result.append(pt)
     if pt == upper_tan_p:
        break

  pt = upper_tan_q
  result.append(upper_tan_q)
  while(True):
     if upper_tan_q == lower_tan_q:
        break
     pt = self.clockwise_point(pt, right)
     if not pt == upper_tan_q:
        result.append(pt)
     if pt == lower_tan_q:
        break

  if pause:
     self.eraseTangent([QLineF(p, q)])
     self.eraseTangent([QLineF(cp_p, cp_q)])
  return result


# returns 0 if the next point does not turn the line in relation to the origin
# returns > 0 if the next point turns right
# returns < 0 if the next point turns left
def turn_direction(self, pt1, pt2, pt2_next):
  temp1 = QPointF(pt2_next.x() - pt1.x(), pt2_next.y() - pt1.y())
  temp2 = QPointF(pt2.x() - pt1.x(), pt2.y() - pt1.y())
  return temp1.x() * temp2.y() - temp2.x() * temp1.y()

def clockwise_point(self, pt, points):
  return points[(points.index(pt) + 1) % len(points)]

def counter_clockwise_point(self, pt, points):
  return points[(points.index(pt) - 1) % len(points)]
```

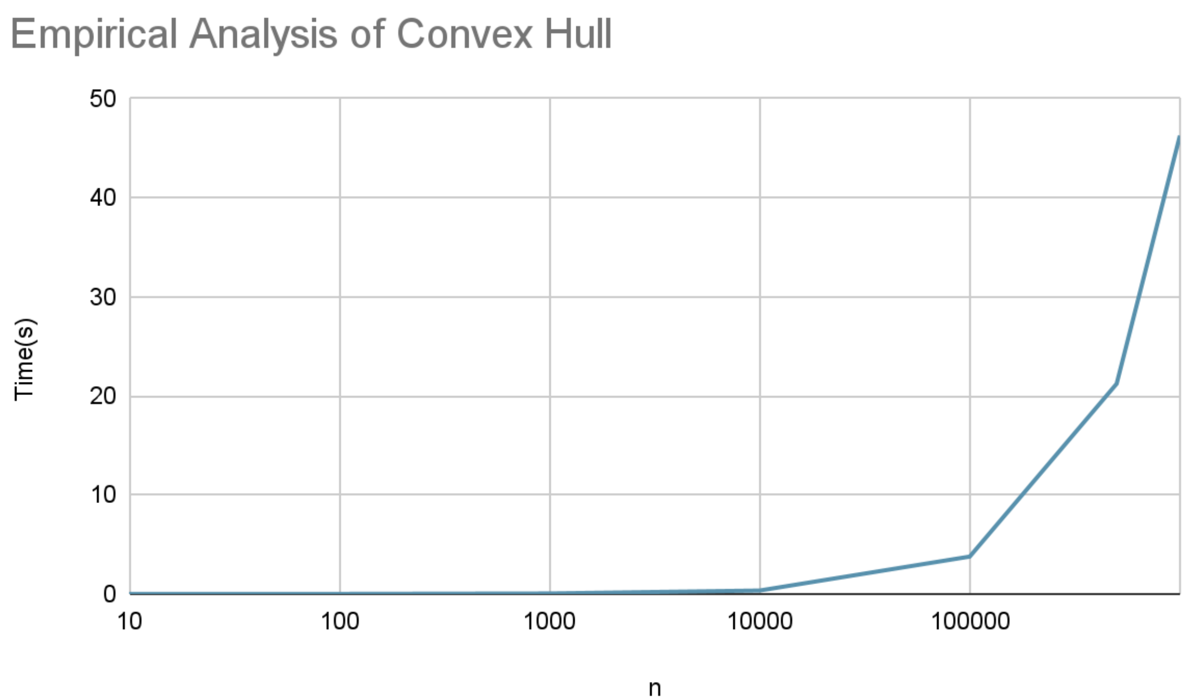
2. The time and space complexity can both be found using the Master Theorem because the algorithm is a divide-and-conquer algorithm. We are breaking the problem into 2 subproblems of size n/2 with each recursive call, and then combining these answers in O(n). In each case, the values a, b, and d are 2, 2, and 1, respectively. This gives us the time and space complexity of O(nlog(n)) because $d = \log_b a$.
   Theoretical Analysis:




3. Experimental outcomes:

| Input Size | Outcome 1 | Outcome 2 | Outcome 3 | Outcome 4 | Outcome 5 |
|---|---|---|---|---|---|
| 10 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 |
| 100 | 0.007 | 0.007 | 0.007 | 0.007 | 0.007 |
| 1000 | 0.052 | 0.053 | 0.053 | 0.053 | 0.043 |
| 10000 | 0.342 | 0.348 | 0.345 | 0.344 | 0.351 |
| 100000 | 3.501 | 4.101 | 3.641 | 3.756 | 3.866 |
| 500000 | 19.320 | 21.725 | 21.116 | 22.231 | 21.693 |
| 1000000 | 46.130 | 47.833 | 43.577 | 47.617 | 46.146 |

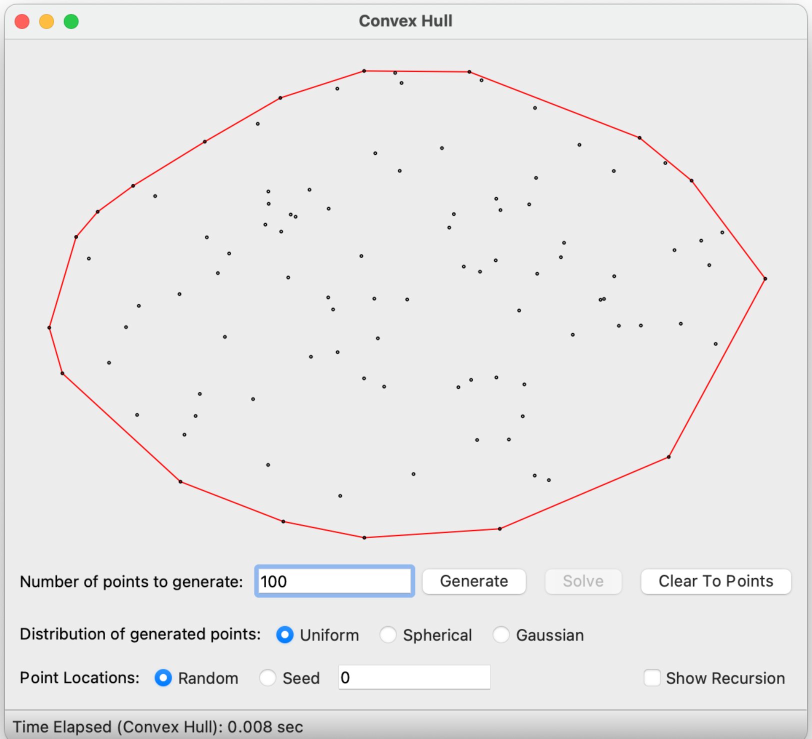
Experimental Analysis:

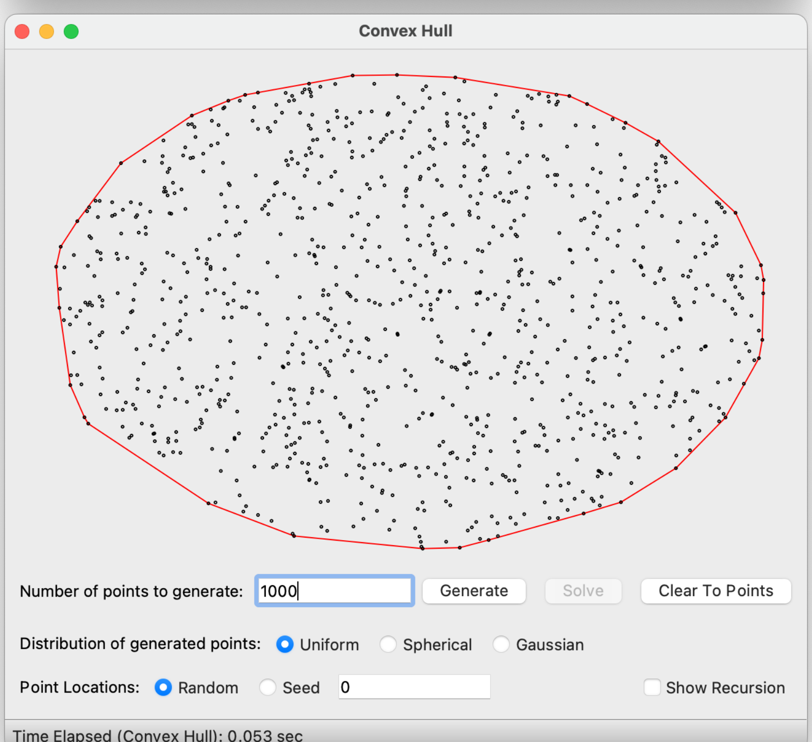



By observation, we can see that the order of growth nlog(n) fits the experimental data almost exactly. My estimate for the constant of proportionality is 0.0001.

4. The theoretical analysis helped me to see why the time complexity of nlog(n) is not nearly as efficient as those below it. Before, I had not seen graphs extensive enough to show the almost exponential growth over time. This also explained why it took so long to comput 500000, 1000000, or even 100000 points. The recursion seems trivial at first, until you increase your input, and suddenly the run time is increasing rapidly. In conclusion, this is an effective method for computing small inputs, but can be easily overwhelmed if you increase the input enough.

5.


Convex Hull
Number of points to generate:
100
Generate
Solve
Clear To Points
Distribution of generated points:
Uniform
Spherical
Gaussian
Point Locations:
Random
Seed
0
Show Recursion
Time Elapsed (Convex Hull): 0.008 sec


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): 0.053 sec