Jeff Reimschussel

Section 2

Convex Hull

**Pseudo code**

**def compute\_hull(points):**

if points is only 2 or 3 points:

top = createTopHalf(points)

bottom = createBottomHalf(points)

return top, bottom

else:

left = solveConvexHull(1st ½ of points)

right = solveConvexHull(2nd ½ of points)

return merge(left, right)

Worst case time efficiency: log n calls of merge.

**def merge(left, right):**

pivoted = true

left\_top = -1

right\_top = 0

current\_slope = slope(left.top[-1] -> right.top[0])

while I have pivoted from at least 1 side:

pivoted = false

while slope(left.top[left\_top] -> right.top[right\_top + 1]) > current\_slope):

right\_top++

pivoted = true

while slope(left.top[left\_top - 1] -> right.top[right\_top]) < current\_slope):

left\_top—

pivoted = true

pivoted = true

left\_bottom = 0

right\_bottom = -1

current\_slope = slope(left.bottom[0] -> right.bottom[-1])

while I have pivoted from at least 1 side:

pivoted = false

while slope(left.bottom[left\_bottom] -> right.bottom[right\_bottom–1] < current\_slope):

right\_bottom—

pivoted = true

while slope(left.bottom[left\_bottom+1] -> right.bottom[right\_bottom] > current\_slope):

left\_bottom++

pivoted = true

merged\_top = left.top[0:left\_top] + right.top[right\_top:]

merged\_bottom = right.bottom[0:right\_bottom] + left.bottom[left\_bottom:]

return merged\_top, merged\_bottom

Worst case time efficiency: Having to switch between hulls at every pivot, meaning visiting each node in the left & right hulls’ respective top/bottom. O(n).

**Source Code**

def slope(a, b):

delta\_x = b.x() - a.x()

delta\_y = b.y() - a.y()

return delta\_y / delta\_x

class ConvexHull():

def \_\_init\_\_(self, top, bottom):

self.top = top

self.bottom = bottom

def \_\_str\_\_(self):

return 'top: %s\nbottom: %s' % (self.top, self.bottom)

def mergeHulls(left, right):

# Merge top

tl = -1

tr = 0

l\_min = 0 - len(left.top)

r\_max = len(right.top) - 1

did\_pivot = True

cur\_slope = slope(left.top[tl], right.top[tr])

while did\_pivot:

did\_pivot = False

while tr < r\_max and slope(left.top[tl], right.top[tr+1]) > cur\_slope:

tr += 1

cur\_slope = slope(left.top[tl], right.top[tr])

did\_pivot = True

while tl > l\_min and slope(left.top[tl-1], right.top[tr]) < cur\_slope:

tl -= 1

cur\_slope = slope(left.top[tl], right.top[tr])

did\_pivot = True

# Wrap around negative index within bounds

tl += len(left.top)

top = left.top[:tl+1] + right.top[tr:]

# Merge bottom: same as top except mirrored

bl = 0

br = -1

l\_max = len(left.bottom) - 1

r\_min = 0 - len(right.bottom)

did\_pivot = True

cur\_slope = slope(left.bottom[bl], right.bottom[br])

while did\_pivot:

did\_pivot = False

while br > r\_min and slope(left.bottom[bl], right.bottom[(br-1)]) < cur\_slope:

br -= 1

cur\_slope = slope(left.bottom[bl], right.bottom[br])

did\_pivot = True

while bl < l\_max and slope(left.bottom[bl+1], right.bottom[br]) > cur\_slope:

bl += 1

cur\_slope = slope(left.bottom[bl], right.bottom[br])

did\_pivot = True

# Wrap around negative index within bounds

br += len(right.bottom)

bottom = right.bottom[:br+1] + left.bottom[bl:]

result = ConvexHull(top, bottom)

return result

def solveConvexHull(points):

if len(points) <= 3:

top = []

top.append(points[0])

bottom = []

bottom.insert(0, points[0])

for p in points[1:-1]:

if p.y() > points[0].y():

top.append(p)

elif p.y() < points[0].y():

bottom.insert(0,p)

top.append(points[-1])

bottom.insert(0, points[-1])

return ConvexHull(top, bottom)

else:

half = len(points)//2

left = solveConvexHull(points[:half])

right = solveConvexHull(points[half:])

return mergeHulls(left, right)

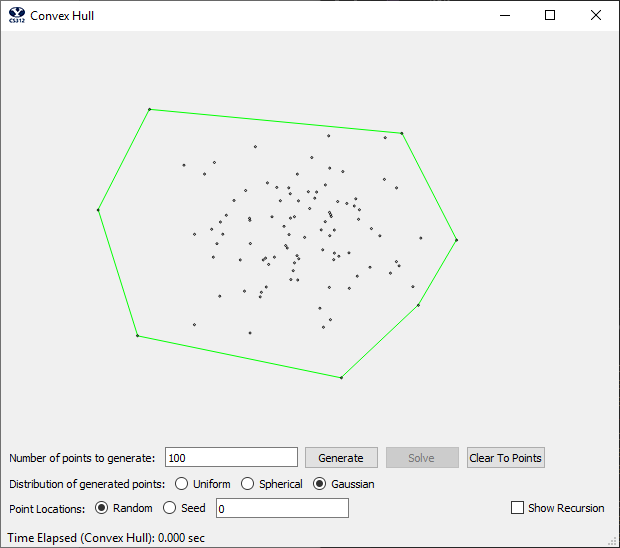
**Theoretical Analysis**

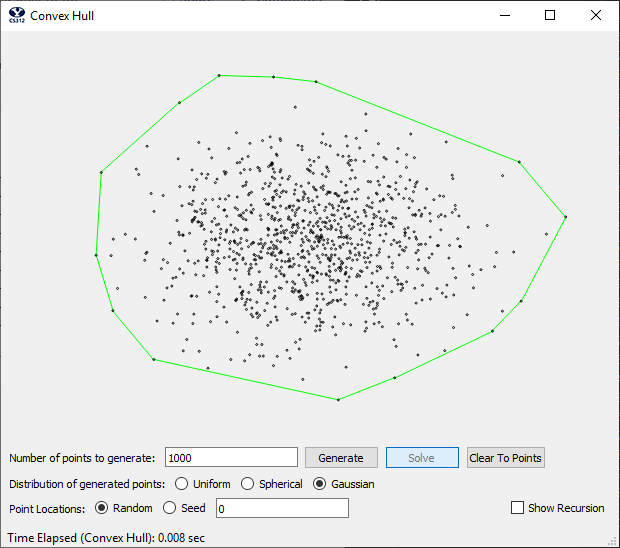
The merge function’s time complexity is at worst O(n). With the compute\_hull function calling it log n times, that would mean the time complexity should be O(n log n).

This supports the same pattern as the Master theorem. Compute\_hull solves *a* subproblems of size *n/b*, then combines the answers in O(). In this solution, both *a* and *b* would be 2 (solving 2 subproblems of half the size each), and d would be 1 (to result in recombining in O(n) time). Because d = , T(n) = O(log *n*) = O(n log n).

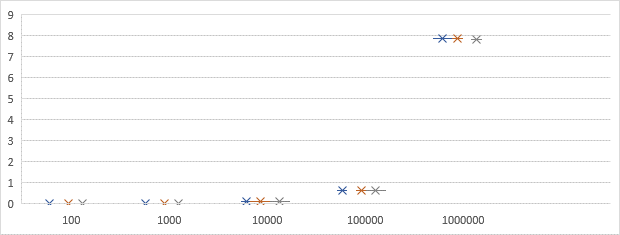
Merge’s require O(n) space for the left and right lists. Because the edge is redefined by moving indices into current arrays, there no new space being used in the bulk of the function. The worst is returning the newly merged top/bottom, which would require at worst O(n) space. Compute\_hull’s requires space for log n calls of merge, so O(n log n) space.

**Empirical Analysis**





|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Elapsed Times (seconds) | | | | | Mean |
| 10 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| 100 | 0.001 | 0.000 | 0.000 | 0.001 | 0.000 | 0.0004 |
| 1,000 | 0.008 | 0.008 | 0.008 | 0.008 | 0.008 | 0.008 |
| 10,000 | 0.071 | 0.073 | 0.072 | 0.072 | 0.072 | 0.072 |
| 100,000 | 0.633 | 0.632 | 0.631 | 0.630 | 0.634 | 0.632 |
| 500,000 | 3.839 | 3.915 | 3.929 | 3.916 | 3.848 | 3.889 |
| 1,000,000 | 7.840 | 7.878 | 7.830 | 7.740 | 7.925 | 7.843 |



\* In order to fit the data, a logarithmic graph has been used. This does scale down the expected width of the distribution by a factor of ten.

The shape that appears to fit the best for the distribution of points is too wide (taking into account the logarithmic scale of x) for it to be quadratic. But the distribution is not linear either—the distribution appears to be lower than a linear projection at the beginning, eventually surpassing a linear growth rate. When applying the same growth rate as surmised in the theoretical analysis, it appears to have a growth of O(n log n).

Using a growth rate of O(n log n), an estimate for a constant of proportionality can be made by solving k \* n log n for k. Assume a logarithmic base 2 because the equation divides the remaining points by half every time. Plugging in 100 for n, we get k 6.024E-7.