Thomas Hart

CS 312

Convex Hull Project

1) All my code with comments for time and space complexity:

```
# This function implements a divide and conquer algorithm for finding a
convex hull of an array of sorted points
# Assume all Math functions are all constant time (division, subtraction,
etc)
def convexHullSolver(self, a):           # Whole thing O(nlogn)
    if len(a) < 4:                       # Space complexity also O(nlogn)
        return self.makeBaseHull(a)      # makeBaseHull function O(1)
    else:
        halfIndex = len(a) // 2          # Division is O(1)
        lowerArray = a[0:halfIndex]      # Each recursion takes O(n) space
        upperArray = a[halfIndex: len(a)]
        lowerArray = self.convexHullSolver(lowerArray)           # O(n)
        upperArray = self.convexHullSolver(upperArray)           # O(n)
        newArray = self.mergeHulls(lowerArray, upperArray)     # merge O(n)
    return newArray

# This takes an array of 3 or fewer points, returning it if it is 1 or two
points in size, and returning the
# clockwise order if it is 3 points in size. This is all in constant time.
def makeBaseHull(self, a):       # Whole thing O(1)
    if len(a) < 3:
        return a
    else:
        if self.findSlope(a[0], a[1]) > self.findSlope(a[0], a[2]):
            return a
        else:
            return [a[0], a[2], a[1]]

# Constant time function to find the slope of a line between two given
points
def findSlope(self, point1, point2):         # Whole thing O(1)
    x1, y1 = point1.x(), point1.y()
    x2, y2 = point2.x(), point2.y()
    return (y2-y1)/(x2-x1)

# This function merges two convex hulls together regardless of its size.
It takes the left-most point on the
# right hull and the right-most point on the left hull and compares slopes
incrementally to find the upper and
# lower tangents. A new array of clockwise points in formed representing
the new convex hull. The other points
# not included in the hull are forgotten, giving us a slight improvement
in time.
def mergeHulls(self, lt, rt):            # Whole thing O(n)
    lIndex = self.getRightPoint(lt)      # getRightPoint function is O(n)
    rIndex = 0
    slope = self.findSlope(lt[lIndex], rt[rIndex])
    # All of these while loops will be less than O(n) with constant work
    while (slope > self.findSlope(lt[lIndex - 1], rt[rIndex]) or slope <
           self.findSlope(lt[lIndex], rt[(rIndex + 1) % len(rt)])):
        while slope > self.findSlope(lt[lIndex - 1], rt[rIndex]):
            slope = self.findSlope(lt[lIndex - 1], rt[rIndex])
            lIndex = lIndex - 1
        while slope < self.findSlope(lt[lIndex],rt[(rIndex + 1)%len(rt)]):
            slope = self.findSlope(lt[lIndex], rt[(rIndex + 1) % len(rt)])
            rIndex = (rIndex + 1) % len(rt)

    luTangent = lIndex % len(lt)            # O(1) division
    ruTangent = rIndex % len(rt)            # O(1) division
    lIndex = self.getRightPoint(lt)         # O(n) getRightPoint
    rIndex = 0
    slope = self.findSlope(lt[lIndex], rt[rIndex])

    # All of these while loops will be less than O(n) with constant work
    while (slope > self.findSlope(lt[lIndex], rt[rIndex - 1]) or slope <
           self.findSlope(lt[(lIndex + 1) % len(lt)], rt[rIndex])):
        while slope > self.findSlope(lt[lIndex], rt[rIndex - 1]):
            slope = self.findSlope(lt[lIndex], rt[rIndex - 1])
            rIndex = rIndex - 1
        while slope < self.findSlope(lt[(lIndex + 1)%len(lt)],rt[rIndex]):
            slope = self.findSlope(lt[(lIndex + 1) % len(lt)], rt[rIndex])
            lIndex = (lIndex + 1) % len(lt)

    rIndex = rIndex % len(rt)
    newArray = []

    # No more than O(n) for these loops
    for i in range(luTangent + 1):
        newArray.append(lt[i])
    while ruTangent != rIndex:
        newArray.append(rt[ruTangent])
        ruTangent = (ruTangent + 1) % len(rt)
    newArray.append(rt[rIndex])
    while lIndex != 0:
        newArray.append(lt[lIndex])
        lIndex = (lIndex + 1) % len(lt)

    return newArray

# This finds the point with the right most x-coordinate in an array and
returns its index
def getRightPoint(self, a):             # O(n) time
    index = 0
    for i in range(len(a)):             # Loop makes this O(n)
        if a[i].x() > a[index].x():
            index = i
    return index

# The given function for the project. Whole thing O(nlogn) time AND space
complexity after sort and solve
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()
    points.sort(key=lambda point: point.x())     # O(nlogn) sort
    t2 = time.time()
    t3 = time.time()

    points = self.convexHullSolver(points)  # O(nlogn) time AND space
    polygon = [QLineF(points[i], points[(i + 1) % len(points)]) for i in
range(len(points))]
    t4 = time.time()

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

2) Time and Space Complexity Analysis:

The convexHullSolver function is the initial recursive function that the array of sorted points is passed to. For my algorithm, I added no edges until the whole convex hull was returned as an array of points beginning at the left-most point of the clockwise hull and moving clockwise until the final point.

```
Convex Hull Solver(a):          # This function recursively called logn times will be O(nlogn)
    If length of a is < 4: makeBaseHull    # Making the base hull is constant time and space
    Split Array into 2 upper and lower arrays     # Constant time but O(n) space
    Recurse two new arrays into this function     # Whole function is O(n)
    Merge the two new arrays                # Merge is roughly O(n)
    Return the new array
```

The makeBaseHull function takes an array that been broken down to at most 3 elements. If the array is 1 or 2 elements in size, it just returns that array because it is already in clockwise order. If there is a 3rd element, it checks to see which point from the left-most point (index 0) will create the highest slope. That point will be index 1 and the 3rd point index 2 to make it all clockwise.

```
Make Base Hull(a):    # Whole thing is constant time and space
    If array is 1 or 2 elements: return array
    If findSlope(a[0], a[1]) > findSlope(a[0], a[2]: return array  # findSlope is O(1)
    Else: return [a[0], a[2], a[1]]
```

The findSlope function simply finds the slope of the two points passed to it.

Find Slope(1, 2):

```
x1, y1 = 1.x(), 1.y()                # Takes a couple spaces but ultimately O(1) time and space
x2, y2 = 2.x(), 2.y()
return (y2 – y1)/(x2 – x1)
```

The mergeHulls function merges the hulls in an efficient way as discussed in class. It finds the upper and lower tangents and connects the two hulls across those and disregards the points inside the new hull

```
Merge Hulls(a1, a2):    # Highest time complexity here O(n) and <= O(n) space for new array
    Get left and right indexes for appropriate start points    # O(n) time to get right most point
    Get slope of those two points                              # Constant time and space
    Run two while loops inside other while loop for upper tangent    # Roughly O(n)
    Record upper tangent points between a1 and a2. Then reset indexes and slope # Constant
    Run next while loop set for lower tangent points           # Roughly O(n) as well
    Run for loops to make new array representing clockwise merged hull    # time <= O(n)
```

The getRightPoint function is used to get the right-most point for the lower array. This is necessary to find the upper and lower tangent points of the two arrays for merging.

```
getRightPoint(array)          # This is O(n) as it searches every element in the array passed to it
    Check each element in array and compare to find point with right-most x value
    Return index of right most point
```

I'm not sure if the function to sort the points by x-value is necessary to mention, but I was told the python sort function was O(nlogn) time, so that is what I used. It worked great.

The recurrence relation using the Master Theorem would look like this:

$aT(n/b) + O(n^d)$

$2T(n/2) + O(n^1)$

a = 2 because there are 2 child nodes at each non-leaf node in this function call

b = 2 because the size of each of the subtasks is cut in half each level you go down (n/2)

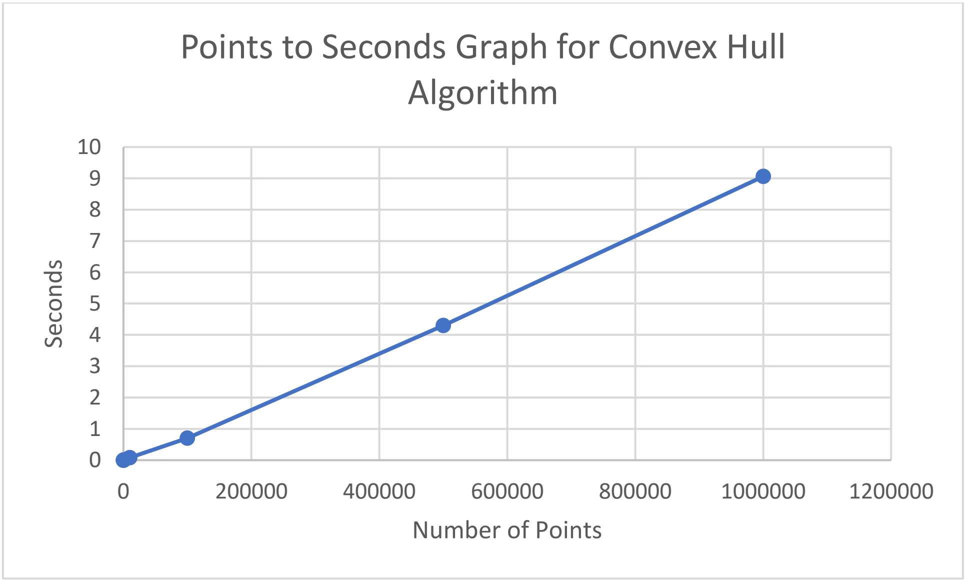
d = 1 because the highest complexity work done in those subtasks is in O(n) time

We then use $a/(b^d) = 2/(2^1) = 1$. We then use the middle selection on the Master Theorem:

$O(n^{(d)}\log n)$ which is just O(nlogn). This is congruent to the O(nlogn) time complexity we theorized for this algorithm.

3) Raw and mean experimental outcomes, plot, and discussion of the pattern in your plot:

- 10 points
  - Raw data – 0.002 s, 0.001 s, 0.000 s, 0.000 s, 0.000 s
  - Mean – 0.0006 s
- 100 points
  - Raw data – 0.001 s, 0.000 s, 0.000 s, 0.001 s, 0.003 s
  - Mean – 0.001 s
- 1000 points
  - Raw data – 0.008 s, 0.010 s, 0.008 s, 0.008 s, 0.009 s
  - Mean – 0.0086 s
- 10,000 points
  - Raw data – 0.074 s, 0.076 s, 0.075 s, 0.074s, 0.076 s
  - Mean – 0.075 s
- 100,000 points
  - Raw data – 0.693 s, 0.698 s, 0.703 s, 0.705 s, 0.701 s
  - Mean – 0.7 s
- 500,000 points
  - Raw data – 4.167 s, 4.174 s, 4.743 s, 4.224 s, 4.202 s
  - Mean – 4.302 s
- 1,000,000 points
  - Raw data – 9.640 s, 9.320 s, 8.509 s, 9.303 s, 8.522 s
  - Mean – 9.0588 s



Constant of proportionality:

I'm assuming this fits O(nlogn) because it is not a straight line on the graph. It has a slight upward curve as we would expect with an nlogn function. The accuracy of 10 and 100 points was only to 3 decimal places, so I'll start with 1000 points to avoid any inaccuracy with the smaller numbers.

To get the constant of proportionality, I'll compare two points on the plot. First, I'll assume the first point is correctly nlogn.

1000 points: x1000log1000 = 0.0086, x = 1.23398e-6

The x value represents the fraction of a second each n value corresponds to theoretically, so using that, the expected value of 10,000 points should be x10,000log10,000 = 0.113654

The actual value (in seconds) for 10,000 points, however, was 0.0086. Dividing the actual by the expected will give the constant of proportionality: **k = 0.0086/.113654 = .07566**. We'll do this for the rest of the points.

10,000 points: x10,000log(10,000) = 0.075 seconds so x = 8.14302e-7

x100,000log(100,000) = .937500. Actual = 0.7. **k = 0.7/.9375 = .74667**

100,000 points: x100,000log(100,000) = 0.7 seconds so x = 6.08012e-7

x500,000log(500,000) = 3.98928. Actual = 4.302. **k = 4.302/3.98928 = 1.0783**

500,000 points: k500,000log(500,000) = 4.302 seconds so k = 6.55675e-7

x1,000,000log(1,000,000) = 9.05848. Actual = 9.0588. **k = 9.0588/9.05848 = 1**

The first few comparisons had a constant of proportionality that showed the algorithm was performing faster than nlogn. As we increased the amount of points, however, the constant of proportionality became almost exactly 1, meaning the curve on the plot fit nlogn perfectly.

4) This will be a mixture of the plot discussion and observations of the theoretical and empirical analysis.

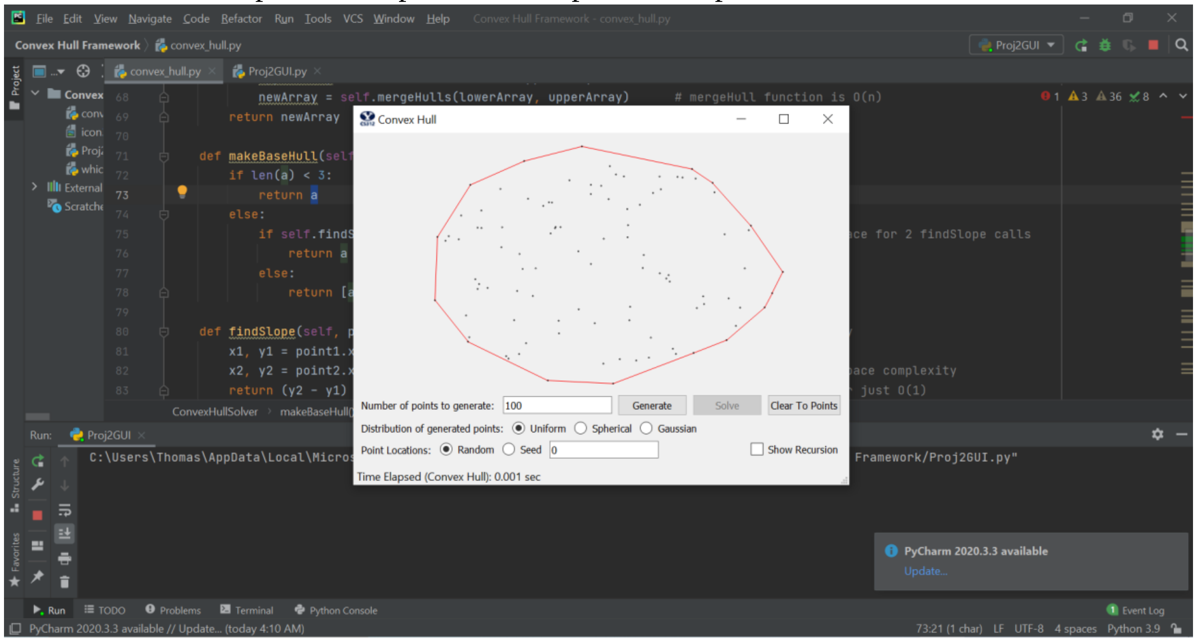
The plot made from the empirical analysis appears to be O(nlogn) as predicted. If it was linear, we would not see the slight upward curve of the line as we see with this graph. The calculations of the constant of proportionality shows that when the amount of points is low, the speed of the algorithm is a bit faster by a constant factor. I believe this speed increase happened for a couple of reasons.

A few of the functions I considered O(n) are a bit lower than that by a constant factor. For example, finding the upper and lower tangent lines likely only iterates through half or even fewer of all the points for the merge function. The O(n) function for getting the right most function would also only include half of the points from the original array because you only loop through the lower array to find this point.

The second reason is as we discussed in class. Getting rid of the points not included in the new convex hull would give us a bit of a speed up each time we did a merge. This gives us much fewer points to worry about each time we go up a level of the divide and conquer algorithm.

For these reasons, I think the empirical analysis corresponds very well to the theoretical O(nlogn) prediction, and when there is a small amount of points, it is a bit faster by a constant factor. Those speed ups seem to become insignificant as the points grow in number.

5) Screenshots of 100 point example and 1000 point example:



```
        newArray = self.mergeHulls(lowerArray, upperArray)      # mergeHull function is O(n)
    return newArray

def makeBaseHull(sel
    if len(a) < 3:
        return a
    else:
        if self.findS
            return a
        else:
            return [a

def findSlope(self, p
    x1, y1 = point1.x
    x2, y2 = point2.x
    return (y2 - y1)
```

```
C:\Users\Thomas\AppData\Local\Micros                    Framework/Proj2GUI.py"
```

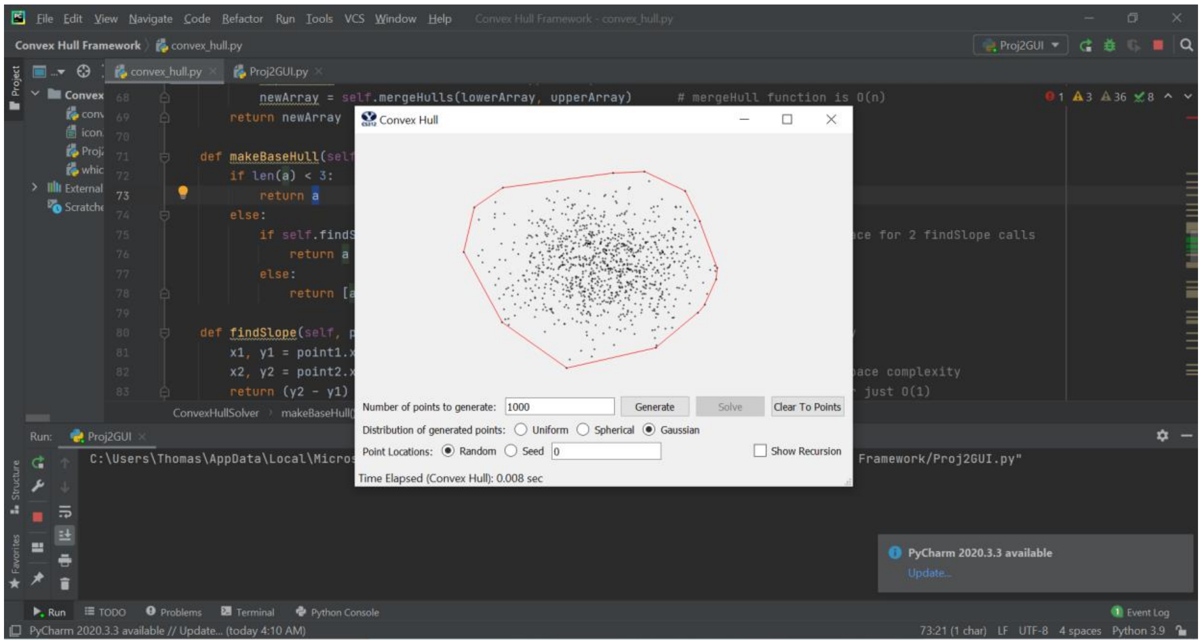
Number of points to generate: 100

Distribution of generated points: [✓ Uniform] Spherical Gaussian

Point Locations: [✓ Random] Seed 0 — Show Recursion

Time Elapsed (Convex Hull): 0.001 sec

File Edit View Navigate Code Refactor Run Tools VCS Window Help
Convex Hull Framework - convex_hull.py
Convex Hull Framework
convex_hull.py
Proj2GUI.py
Proj2GUI
Convex Hull
Generate
Solve
Clear To Points
ace for 2 findSlope calls
ace complexity
just O(1)
ConvexHullSolver
makeBaseHull(
Run:
PyCharm 2020.3.3 available
Update...
Run
TODO
Problems
Terminal
Python Console
Event Log
PyCharm 2020.3.3 available // Update... (today 4:10 AM)
73:21 (1 char) LF UTF-8 4 spaces Python 3.9



```
        newArray = self.mergeHulls(lowerArray, upperArray)      # mergeHull function is O(n)
    return newArray

def makeBaseHull(sel
    if len(a) < 3:
        return a
    else:
        if self.findS
            return a
        else:
            return [a

def findSlope(self, p
    x1, y1 = point1.x
    x2, y2 = point2.x
    return (y2 - y1)
```

```
C:\Users\Thomas\AppData\Local\Micros                    Framework/Proj2GUI.py"
```

Number of points to generate: 1000

Distribution of generated points: Uniform Spherical [✓ Gaussian]

Point Locations: [✓ Random] Seed 0 — Show Recursion

Time Elapsed (Convex Hull): 0.008 sec

File Edit View Navigate Code Refactor Run Tools VCS Window Help
Convex Hull Framework - convex_hull.py
Convex Hull Framework
convex_hull.py
Proj2GUI.py
Proj2GUI
Convex Hull
Generate
Solve
Clear To Points
ace for 2 findSlope calls
ace complexity
just O(1)
ConvexHullSolver
makeBaseHull(
Run:
PyCharm 2020.3.3 available
Update...
Run
TODO
Problems
Terminal
Python Console
Event Log
PyCharm 2020.3.3 available // Update... (today 4:10 AM)
73:21 (1 char) LF UTF-8 4 spaces Python 3.9