Mark Watson
CS 4720
Homework #3
11/09/2011

Problem 1 Closest-Pair Problem

For the closest pair problem I used a simple divide and conquer algorithm that took $\theta(nlogn)$ time complexity. The algorithm is recursive, and works in two stages. In the first stage it splits up the points into two parts and recurses on each half. In the second stage the code merges the two halves to combine the points back into a single solution.

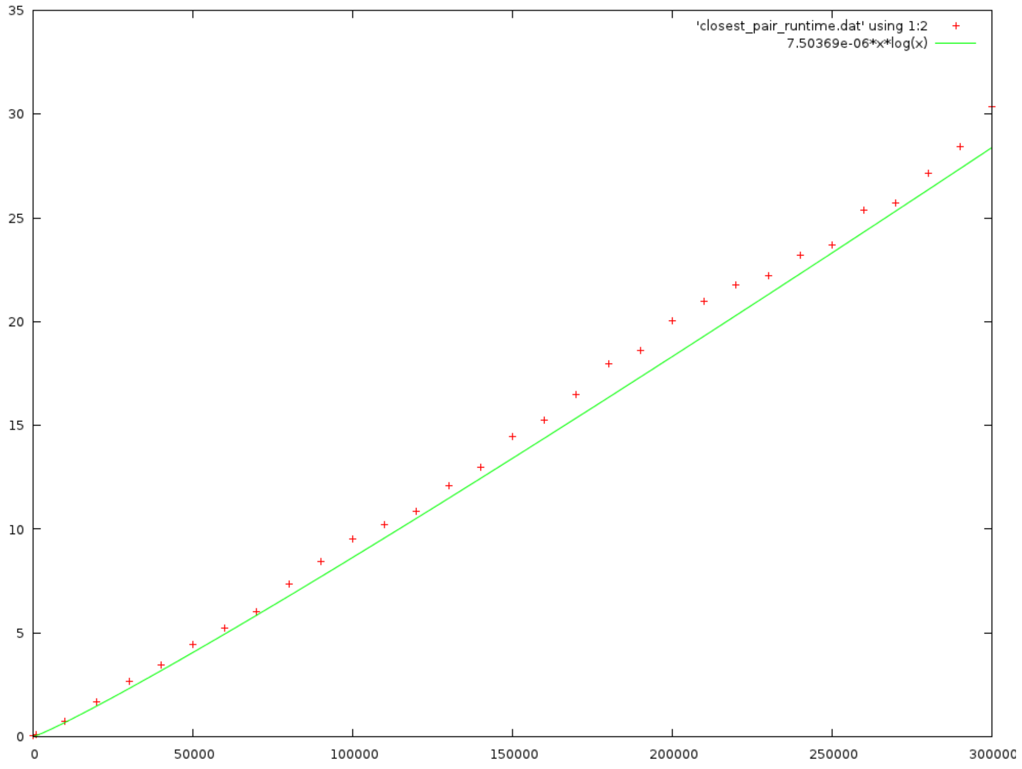
The pseudo-code is as follows. "by_x" and "by_y" are the list of points sorted by the x-axis and y-axis, respectively. The time to sort take $\theta(nlogn)$ time, so the final algorithm won't be faster than that.

```
closest_pair(by_x, by_y):
    if by_x.length == 1:
        return null
    else if by_x.length == 2:
        return new pair(by_x[0], by_x[1])
    set left_by_x and left_by_y to the first n/2 elements
        of by_x and by_y respectively
    set right_by_x and right_by_y to the last n/2 elements
        of by_x and by_y respectively
    l_pair = closest_pair(left_by_x, left_by_y)
    r_pair = closest_pair(right_by_x, right_by_y)
    l_dist = euclidean_distance(l_pair)
    r_dist = euclidean_distance(r_pair)
    if l_dist < r_dist:
        dist = l_dist
        closest = l_pair
    else:
        dist = r_dist
        closest = r_pair
    create y_strip containing all the values in by_y that
        are within 'dist' distance
    for i = 0 to y_strip.length:
            for j = i to i + 15:
                if distance(y_strip[i], y_strip[j]) < dist:
                    dist = distance(y_strip[i], y_strip[j])
                    closest = new pair(y_strip[i], y_strip[j])
        return closest
```

The time complexity of the algorithm is $\theta(nlogn)$. The process for obtaining this involves several pieces. The first piece is sorting the points by x and y, which takes $2\theta(nlogn)$ using an efficient sorting algorithm. The next part is the time to run the function, which can be found using a recurrence.

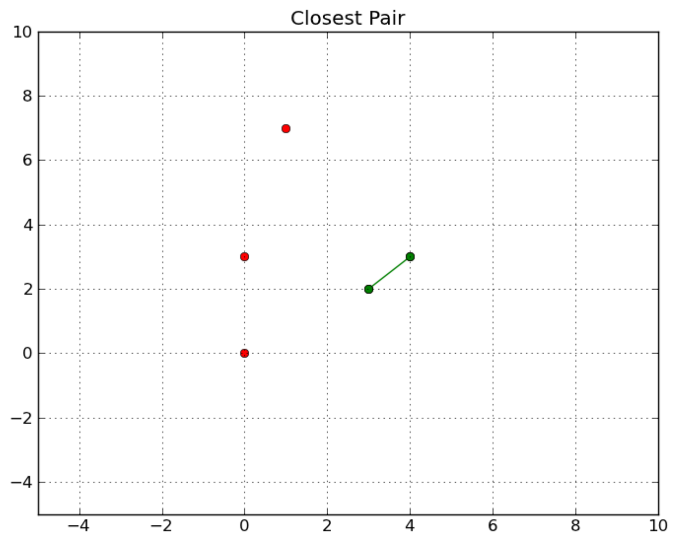
$$\begin{aligned} T(N) &= 2T(\frac{N}{2}) + 6N \\ &\in \theta(nlog(n)) \end{aligned}$$

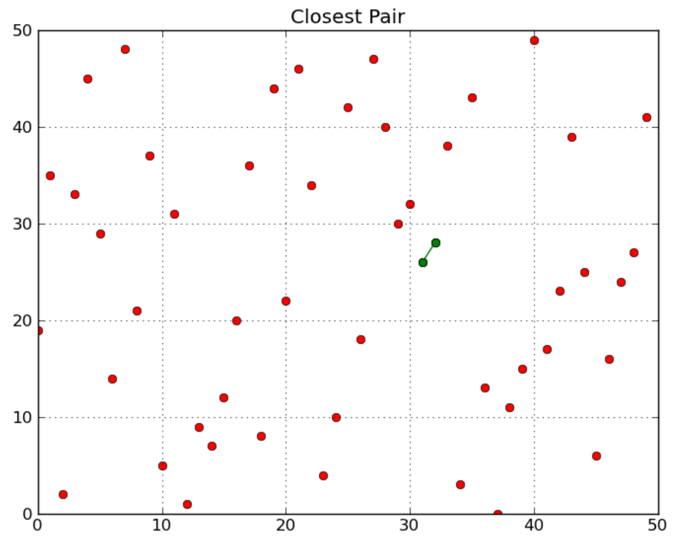
Above, the recurrence is solved using the master theorem. The numbers have to be iterated over "6" times for each step, which is where the 6N comes from. Since a linear function is added each time, we can get a time complexity on par with sorting, so the overall time complexity is $\theta(nlog(n))$.



The graph shows the algorithm being run for inputs up to 300,000. The function $nlog(n)$ is fitted to the graph to show the growth of the function. In the smaller instances of n, the function's growth rate almost seems more linear. Since $nlog(n)$ grows slightly faster than linear this makes sense.

The following two plots show the closest pair for 5 points and 50 points.





■

Problem 2 Counting Inversion Problem

(a) A brute force solution to the counting inversion problem simply visits each element in order, and then iterates over all the succeeding elements and checks which ones to swap by comparing them. The total number of swaps is added up and returned in the process.

To measure the time complexity we can use summations over some constant, C, representing the time to compare and swap two numbers:

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{n}\sum_{j=i}^{n} C \\
&= \sum_{i=0}^{n}(n - i + C) \\
&= C + n^2 - \sum_{i=0}^{n} i \\
&= C + n^2 - \frac{1}{2}(n+1)n \\
&= n^2 - \frac{1}{2}n^2 + \frac{1}{2}n \\
&= \frac{1}{2}n^2 + \frac{1}{2}n \\
&\in \theta(n^2)
\end{aligned}
$$

This makes sense since it is the same time complexity as the brute force sorting algorithms like bubble sort and selection sort. It's doing the same thing as a brute force sorting algorithm by looping through all the elements for each element in the list.

(b) For a divide and conquer algorithm I used a merge-and-count algorithm which was essentially the same as the merge-sort algorithm for sorting, but instead counts swaps that need to be made. Like most divide and conquer algorithms it consisted of two parts. The first part splits the list up into two parts recursively, and the second part merges the list back together while counting the number of swaps required.

The pseudo code for the algorithm is as follows:

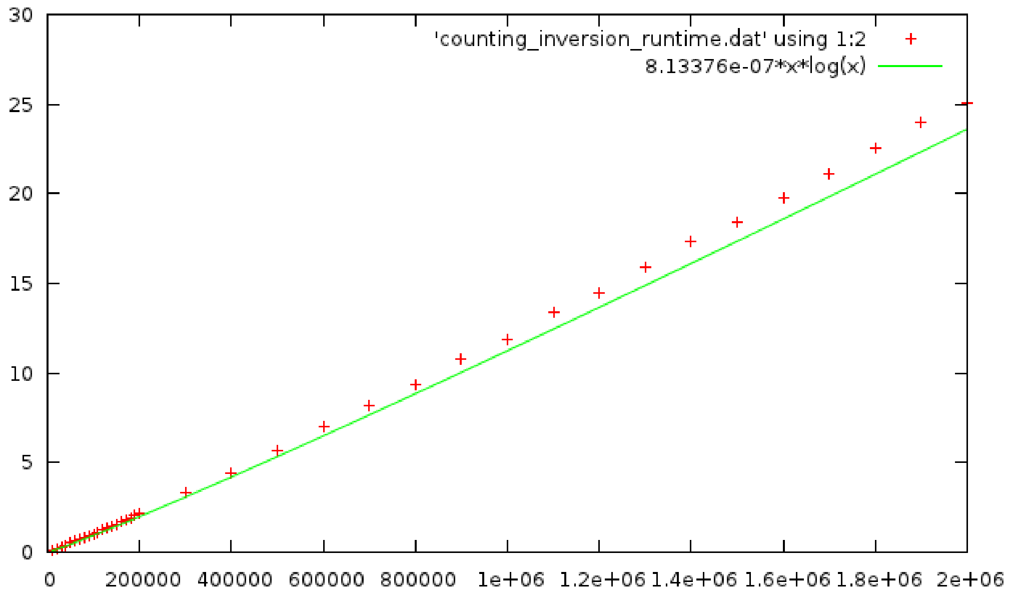
```
function sort_and_count(L): // l is the list to work with
    if L's length is 1:
        return 0, L
    else:
        // divide L into a and b
        a = first half of L
        b = second half of L
        count_a, a = sort_and_count(a)
        count_b, b = sort_and_count(b)
        count, L = merge_and_count(a, b)
        // return the count and the list.
        return count_a + count_b + count, L
function merge_and_count(a, b): // a and b are two lists
    initialize i and j to 0
    while i < length of a and j < length of b:
        append the minimum of a[i] and b[j] onto a new list, "c"
        if b[j] < a[i]:
            count += length of a - i
            j++
        else:
            i++
    add anything left at the end of a or b onto c
    return count, c
```

This code can be analyzed using a recurrence relation to find out the time complexity. We can use the master theorem to find out the time complexity.

$$\begin{aligned} T(n) = & \ 2T(n/2) + n \\ \in & \ \theta(nlog(n)) \end{aligned}$$

The $2T(n/2)$ part is for the two recursive calls that split the elements in half. The "n" part is how long the "merge_and_count" function takes, assuming the constant time for each iteration to be one, for simplicity.

The algorithm run time compares to the theoretical results. Below is a graph of the run time for increasing values of n fitted to an $nlog(n)$ function.



The runtime fits to the $nlog(n)$ function fairly well, and shows the algorithm performing quickly.

A few trace runs of the algorithm are shown below. The python implementation here finds the actual swaps required using brute force, and then verifies the number found using brute force by also using the divide and conquer algorithm to show the number of swaps expected. Using two different algorithms helps to ensure they're implemented correctly, and is needed for verification since the merge and count algorithm doesn't show which swaps are needed.

```
$ python counting_inversion.py counting_inversion_tests/simple
Sequence: <2, 4, 1, 3, 5>
Swaps (found with brute force):
Swap: (2, 1)
Swap: (4, 2)
Swap: (4, 3)
# Swaps (Found with divide and conquer): 3
$ python counting_inversion.py counting_inversion_tests/ordered
Sequence: <1, 2, 3, 4>
Swaps (found with brute force):
# Swaps (Found with divide and conquer): 0
$ python counting_inversion.py counting_inversion_tests/inverted
```

```
Sequence: <4, 3, 2, 1>
Swaps (found with brute force):
Swap: (4, 3)
Swap: (3, 2)
Swap: (2, 1)
Swap: (4, 3)
Swap: (3, 2)
Swap: (4, 3)
# Swaps (Found with divide and conquer): 6
```

■

Problem 3 Convex Hull Problem

The first algorithm for the convex hull problem I implemented was the *gift-wrapping* algorithm, aka the Jarvis march algorithm. It has a theoretical time complexity of $\theta(nh)$, where n is the number of total points, and h is the number of points on the hull. In the worst case situation all the points are on the hull, so the worst case time complexity is $O(n^2)$.

The pseudo code I used is as follows:

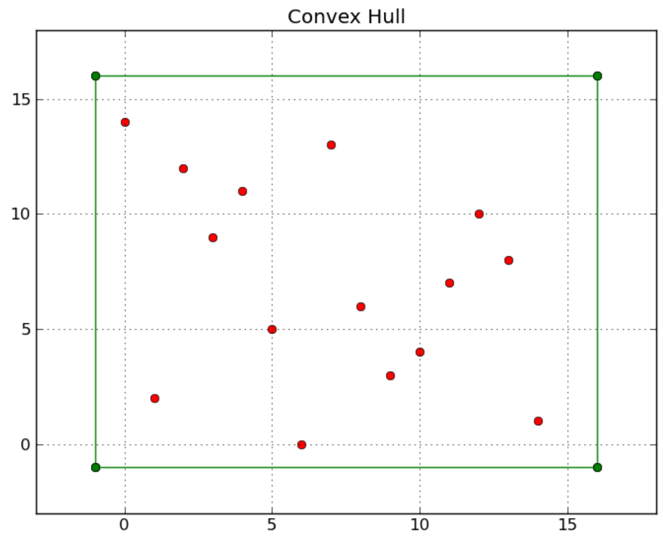
```
function gift_wrap(P):
    H = new empty list
    n = length of P

    // find the leftmost point
    point_on_hull = leftmost point in P
    do:
        append point_on_hull to list H
        endpoint = P[0]
        for j=1 to n-1: // skip the first element
            if endpoint == point_on_hull
               or IsLeftOf(points[j], point_on_hull, endpoint) then
                endpoint = points[j]
            endif
        point_on_hull = endpoint
    while endpoint != hull[0]
    return hull
```

The algorithm requires a function that can check if a point is to the left of the line through two other points. It's implementation is simple and just uses the determinant to find out if the area of a triangle made by the three points is negative or positive. The time complexity of the pseudo code above can be calculated as follows:

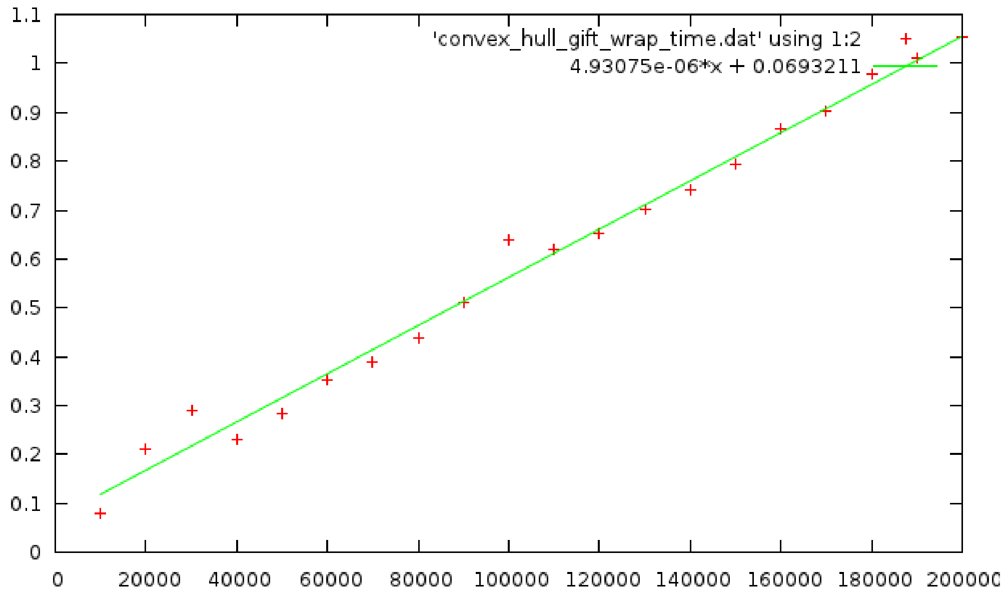
$$\begin{aligned} T(n) &= \textstyle\sum_{i=0}^{n-1} C + \sum_{i=0}^{h} \sum_{j=1}^{n-1} D \\ &= (n-1)C + \textstyle\sum_{i=0}^{h} CDn \\ &= Cn - C + Dnh \\ &\in \theta(nh) \end{aligned}$$

The first summation is for the time to find the leftmost point, and then the last two summations are the time to loop over all the points for each point on the hull.

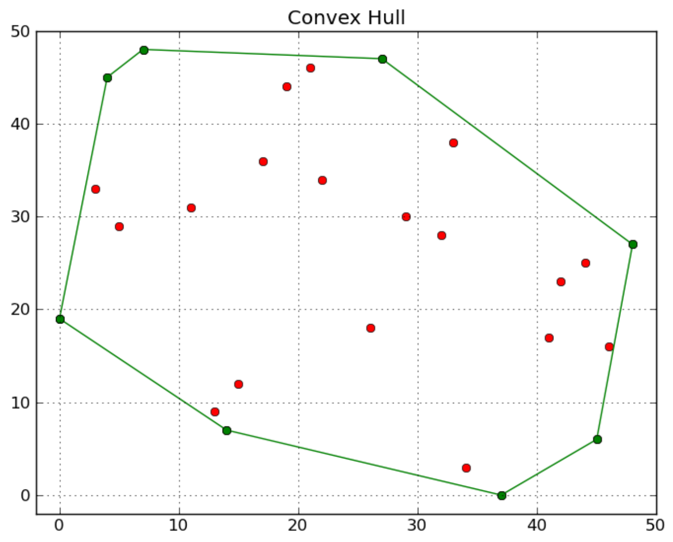
To compare the theoretical to the actual run time we can hold "h" constant and then measure the time taken to run for increasing values of n. When "h" is constant the growth rate of "n" should be linear, since $\theta(nh) = h\theta(n) = \theta(n)$ if h is constant. To set "h" constant we'll simply select some points that form a hull and then randomly generate points within the hull. An example of such a situation is below.

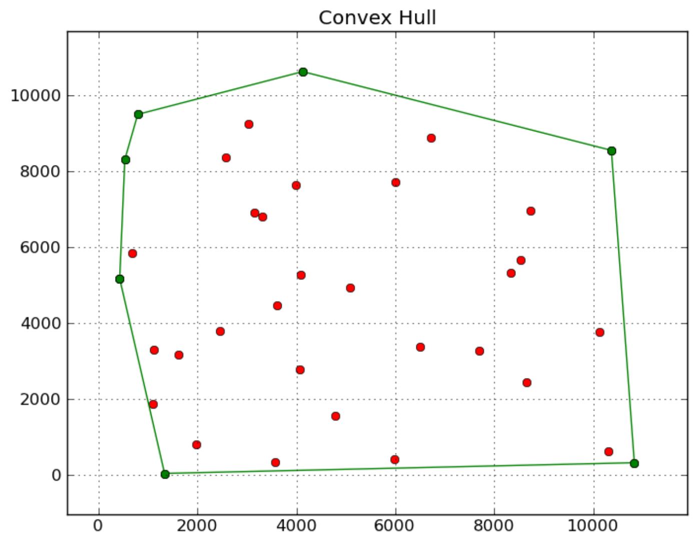


We then keep putting more and more randomly generated points in the hull and graph the time to find the closest pair over time. The time taken to run nicely fits a linear function, showing that the theoretical time analysis is correct.



A couple other interesting convex hulls are plotted below. They were all found with the gift wrapping algorithm.





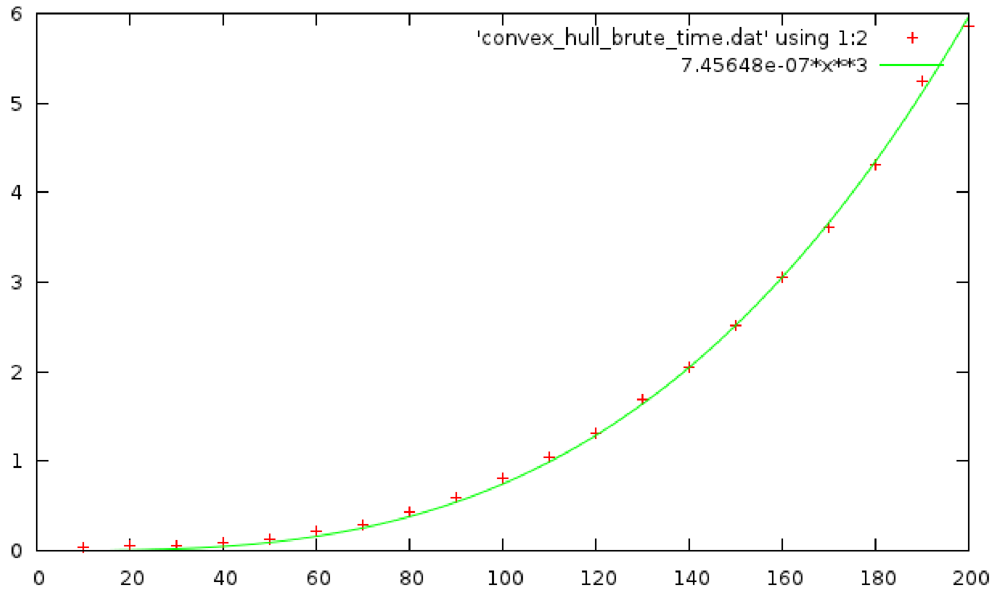
Another algorithm I implemented for the convex hull problem was a brute force algorithm. This proved to be useful for verifying my other algorithms, although the time complexity was too large to run on very big inputs. The algorithm involves finding each line between a pair of points and finding out if all the other points are on one side of the line. If all the points are on one side of the line then it's in the convex hull. The pseudo code for the algorithm is as follows:

```
function convex_hull(P):
    set n to length of P
    hull = new empty list
    for i=0 to n-1:
        for j=0 to n-1:
            right = false
            left = false
            colinear = false
            for k=0 to n-1:
                if P[k] != P[i] and P[k] != P[j]:
                    if P[k] is to the left of P[i] and P[j]:
                        left = true
                    if P[k] is to the right of P[i] and P[j]:
                        right = true
                    if P[k] is colinear to P[i] and P[j]:
                        colinear = true
        if (right and not left and not colinear)
            or (left and not right and not colinear)
            or (colinear and not right and not left) then
                append (points[i], points[j]) to hull
        endif
return hull
```

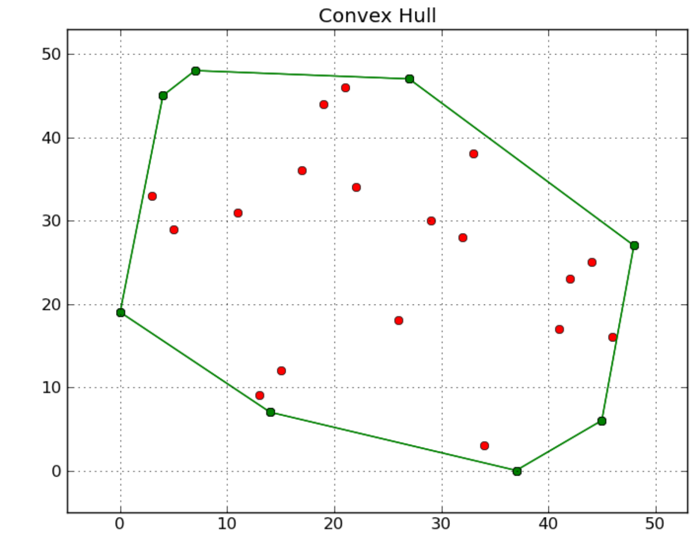
The time analysis for this algorithm is:

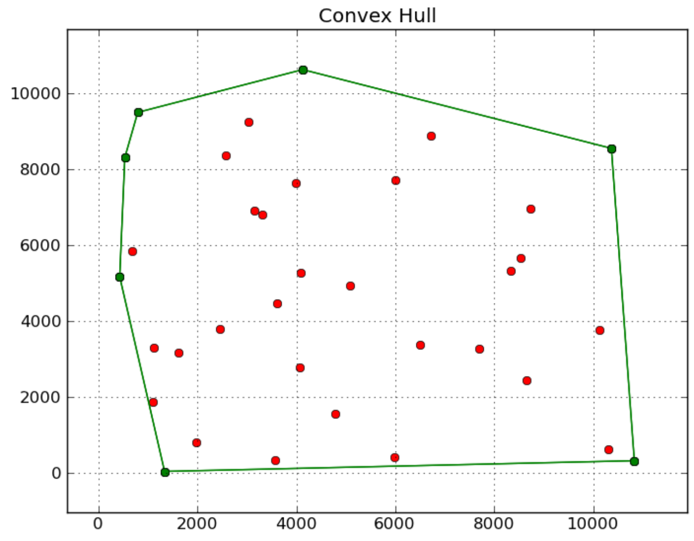
$$
\begin{aligned}
T(n) &= \textstyle\sum_{i=01}^{n}\sum_{j=1}^{n}(C + \sum_{k=1}^{n} D) \\
&= \textstyle\sum_{i=01}^{n}\sum_{j=1}^{n}(C + Dn) \\
&= \textstyle\sum_{i=01}^{n}(Cn + Dn^2) \\
&= (Cn^2 + Dn^3) \\
&\in \theta(n^3)
\end{aligned}
$$

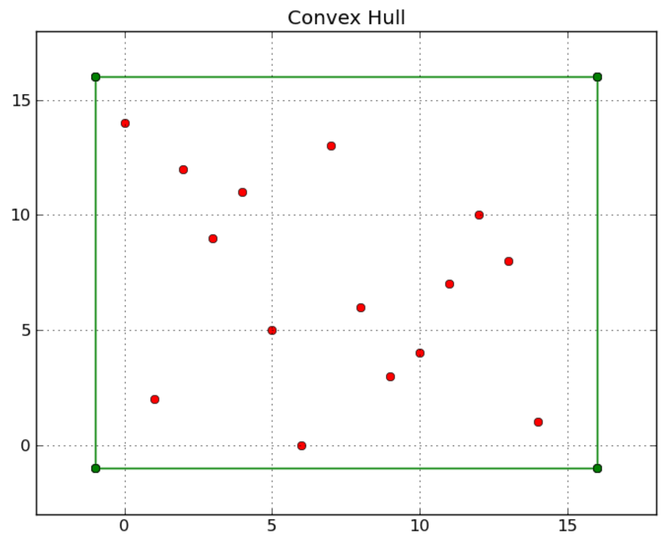
This matches with the actual time the algorithm took to run:



Here are some plots showing the brute force algorithm got the same results as the gift-wrapping method:


Convex Hull
50
40
30
20
10
0
0
10
20
30
40
50


Convex Hull
10000
8000
6000
4000
2000
0
0
2000
4000
6000
8000
10000


Convex Hull
15
10
5
0
0
5
10
15

■