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.

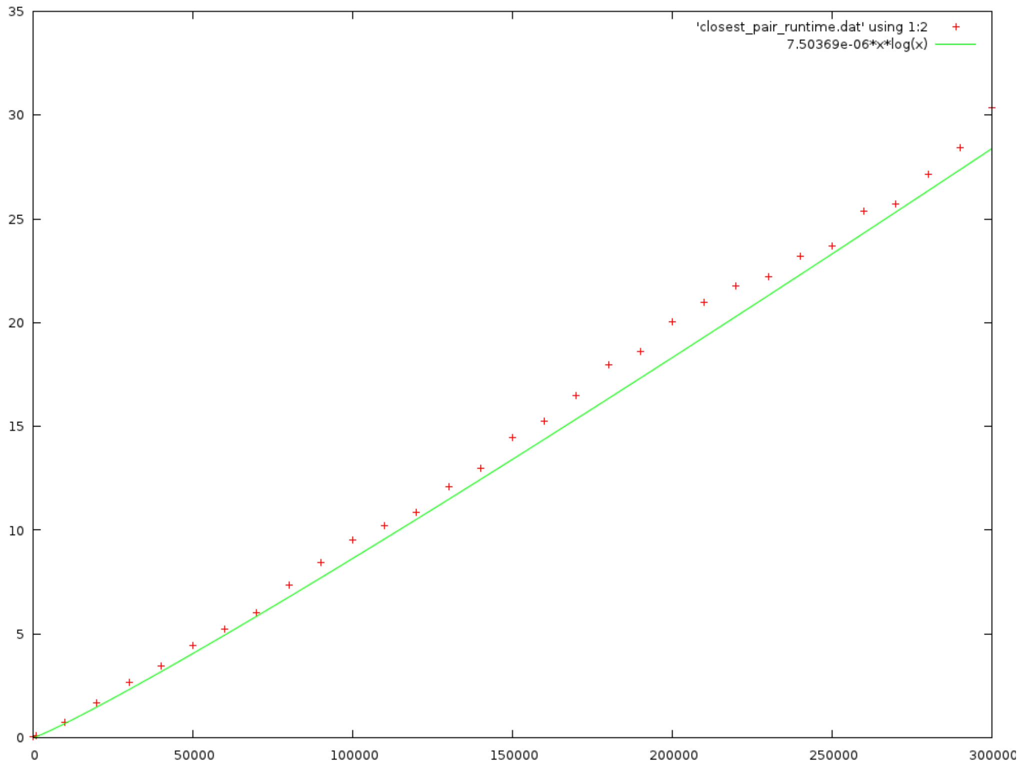
The pseudo-code is as follows. "by_x" and "by_y" are the list of points sorted by the x-axis and y-axis, respectively. The time to sort take $\theta(nlogn)$ time, so the final algorithm won't be faster than that.

```
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.

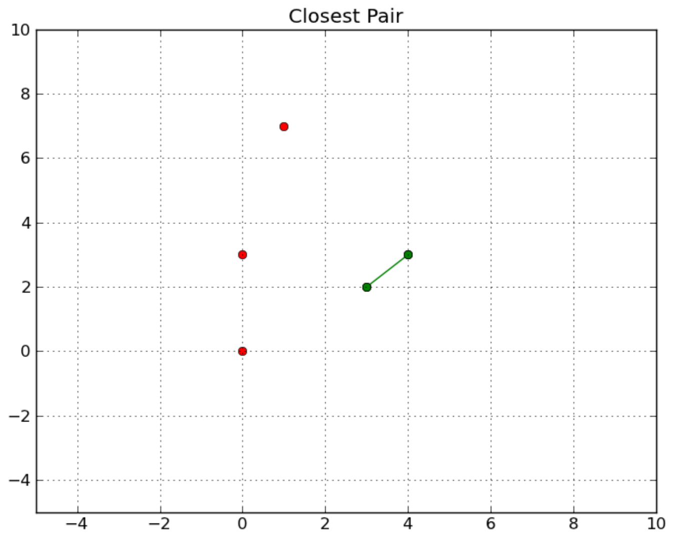
$$\begin{aligned} T(N) &= 2T(\frac{N}{2}) + 6N \\ &\in \theta(nlog(n)) \end{aligned}$$

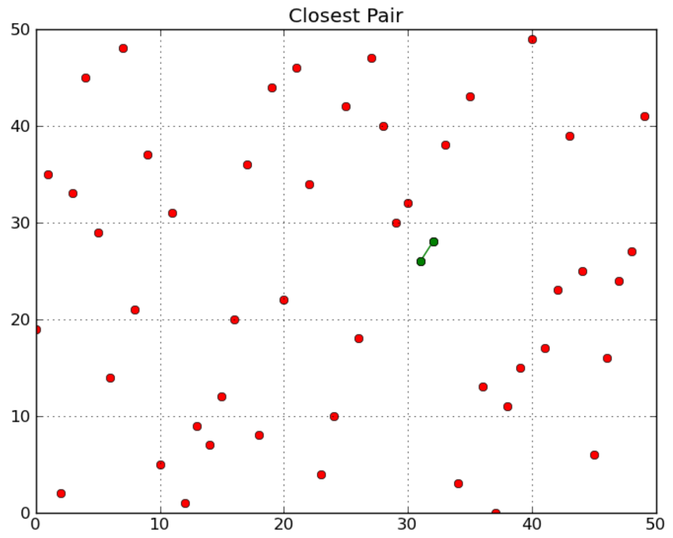
Above, the recurrence is solved using the master theorem. The numbers have to be iterated over "6" times for each step, which is where the 6N comes from. Since a linear function is added each time, we can get a time complexity on par with sorting, so the overall time complexity is $\theta(nlog(n))$.



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:

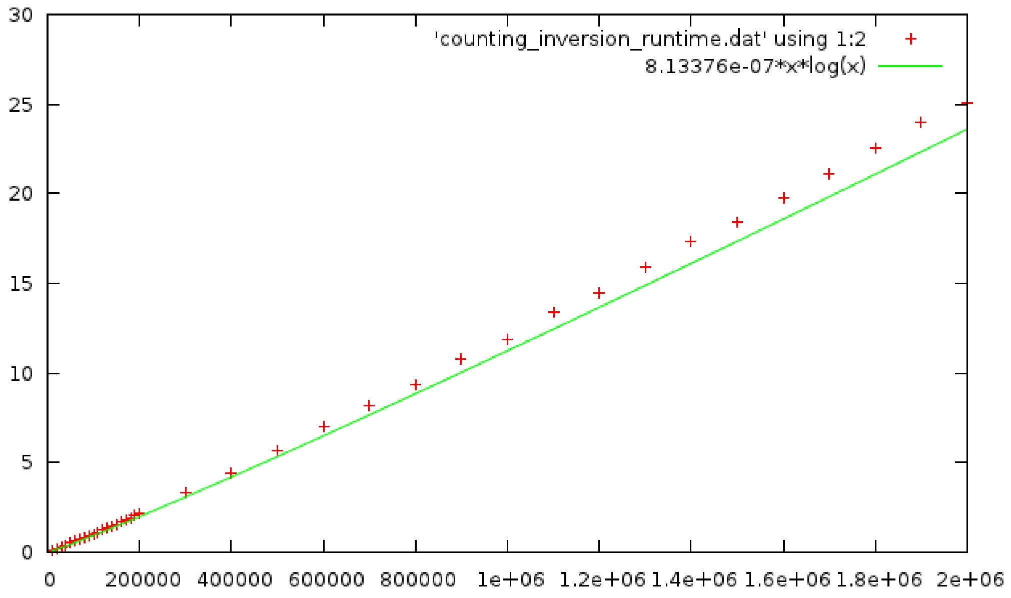
```
function sort_and_count(L): // l is the list to work with
    if L's length is 1:
        return 0, L
        else:
            // divide L into a and b
            a = first half of L
            b = second half of L
            count_a, a = sort_and_count(a)
            count_b, b = sort_and_count(b)
            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:

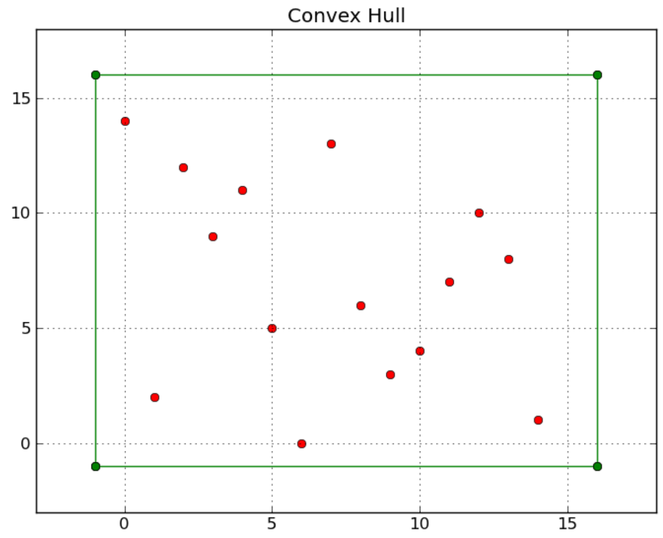
```
function gift_wrap(P):
    H = new empty list
    n = length of P

    // 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:

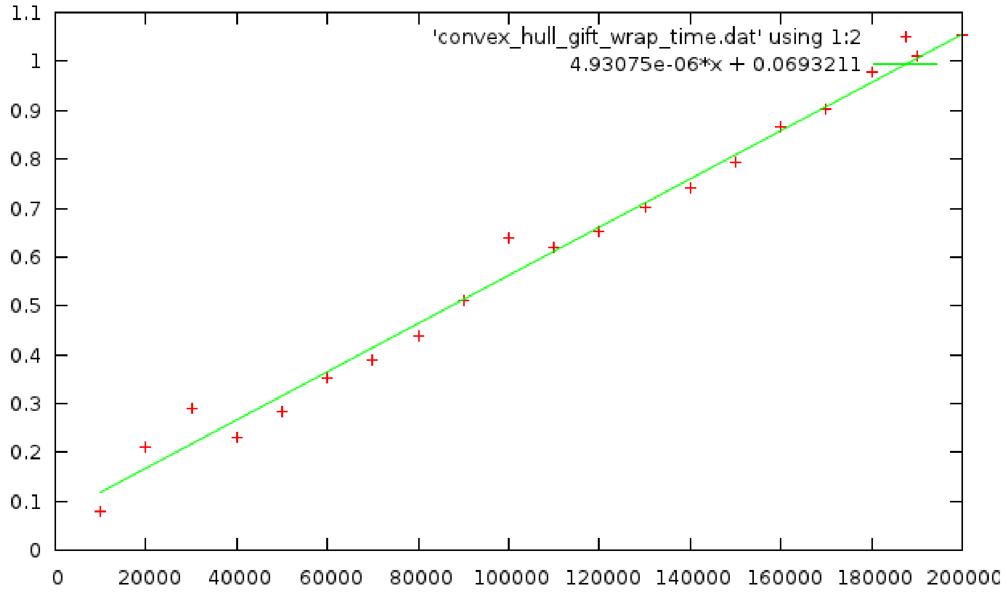
$$\begin{aligned} T(n) &= \textstyle\sum_{i=0}^{n-1} C + \sum_{i=0}^{h} \sum_{j=1}^{n-1} D \\ &= (n-1)C + \textstyle\sum_{i=0}^{h} CDn \\ &= Cn - C + Dnh \\ &\in \theta(nh) \end{aligned}$$

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.

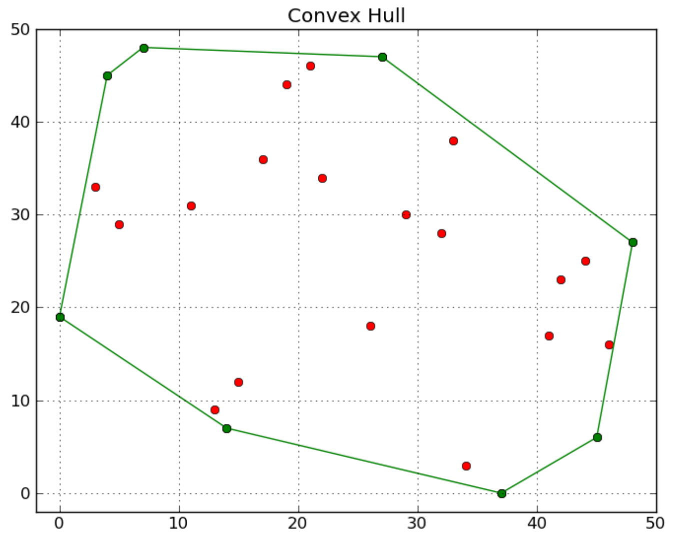
To compare the theoretical to the actual run time we can hold "h" constant and then measure the time taken to run for increasing values of n. When "h" is constant the growth rate of "n" should be linear, since $\theta(nh) = h\theta(n) = \theta(n)$ if h is constant. To set "h" constant we'll simply select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

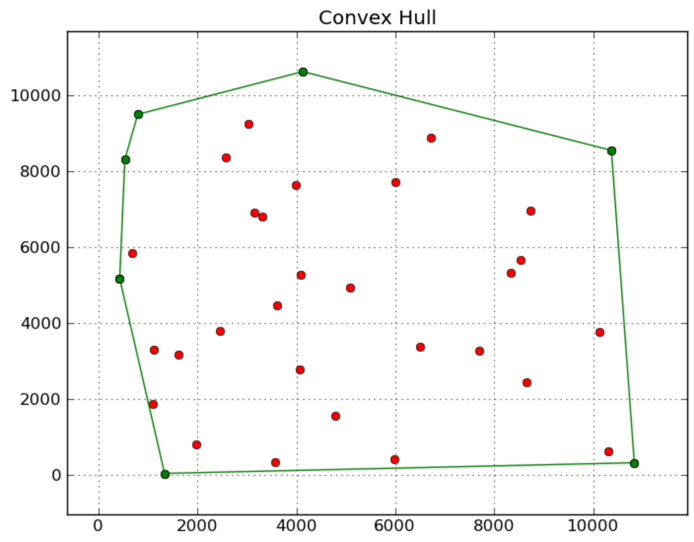


We then keep putting more and more randomly generated points in the hull and graph the time to find the closest pair over time. The time taken to run nicely fits a linear function, showing that the theoretical time analysis is correct.



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





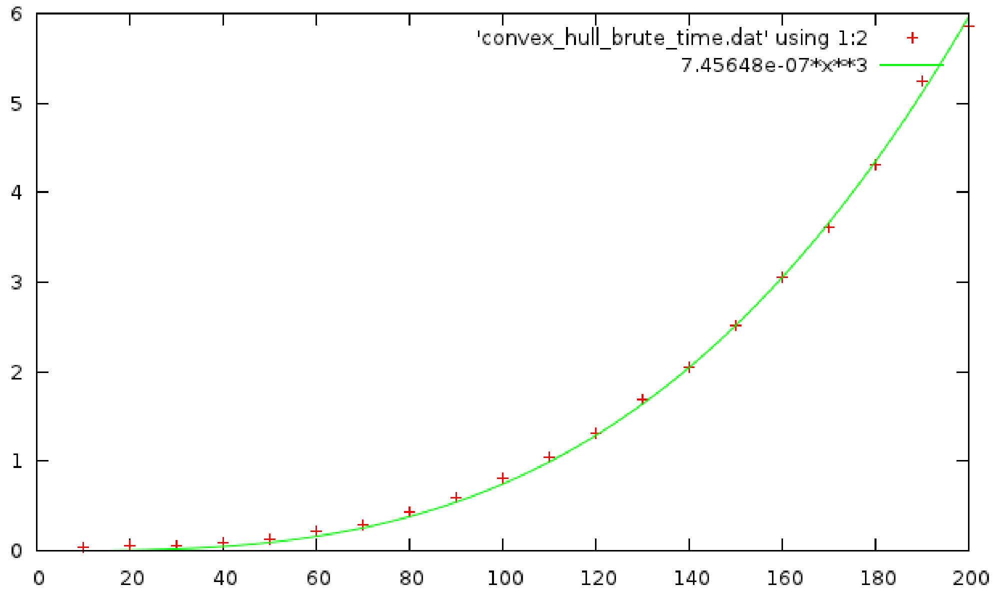
Another algorithm I implemented for the convex hull problem was a brute force algorithm. This proved to be useful for verifying my other algorithms, although the time complexity was too large to run on very big inputs. The algorithm involves finding each line between a pair of points and finding out 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
```

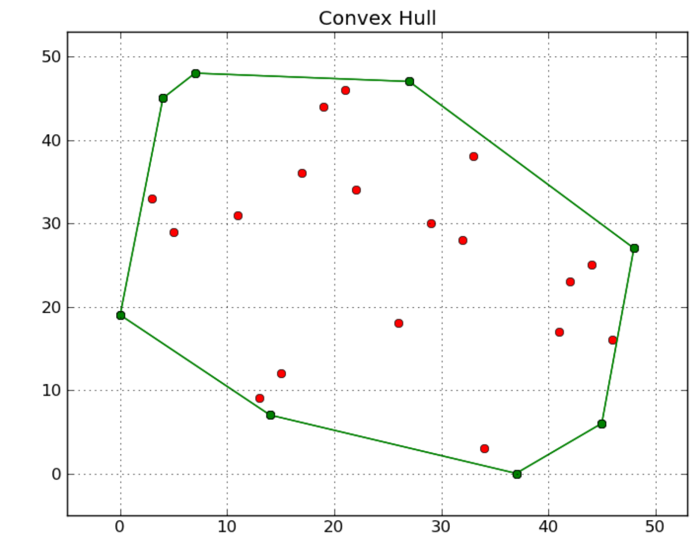
The time analysis for this algorithm is:

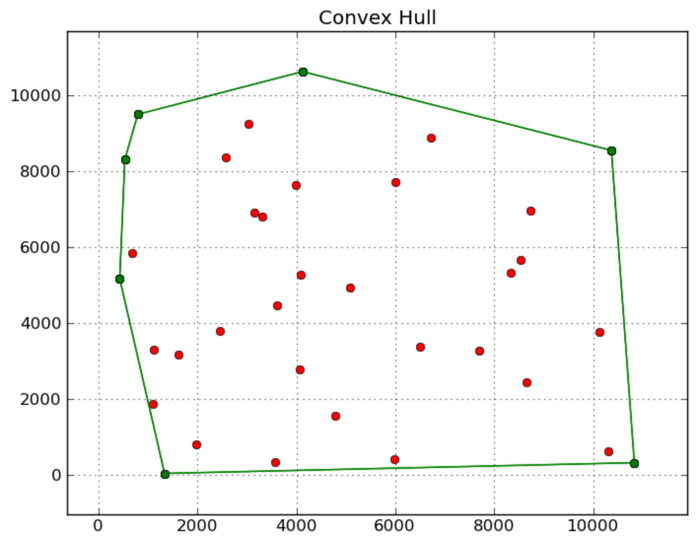
$$\begin{aligned} T(n) &= \sum_{i=01}^{n} \sum_{j=1}^{n} (C + \sum_{k=1}^{n} D) \\ &= \sum_{i=01}^{n} \sum_{j=1}^{n} (C + Dn) \\ &= \sum_{i=01}^{n} (Cn + Dn^2) \\ &= (Cn^2 + Dn^3) \\ &\in \theta(n^3) \end{aligned}$$

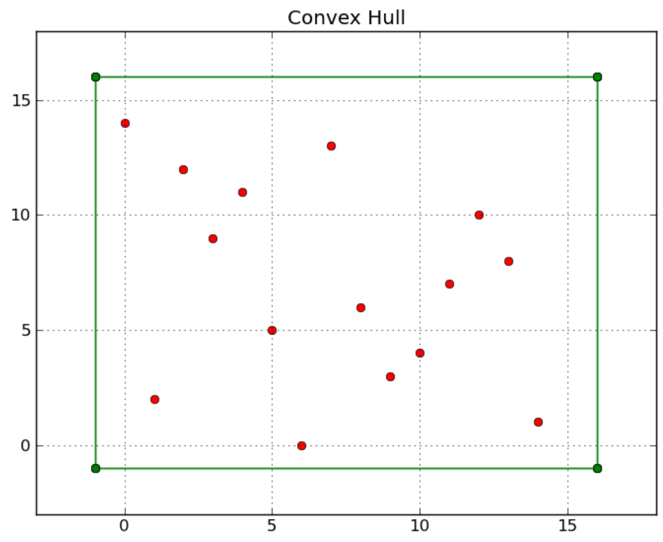
This matches with the actual time the algorithm took to run:



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

■

```
# Closest Pair

import sys
import os
import math
import argparse

# find the distance between two points
def fake_euclid(x, y):
    """
    Find a value given two points that's a multiple of the distance between the
    two points. Good for comparing distances where the actual value doesn't
    matter.
    """
    return (x[0] - y[0])**2 + (x[1] - y[1])**2

def closest_pair(points):
    """
    Find the closest pair of points given a list of points.
    In: [(x,y), (x,y), ...], where (x,y) are the points.
    Out: (i,j), where i and j are the index of the points.
    """
    x_ind = 0
    y_ind = 1

    # a recursive function to do the work
    def closest_pair_recurse(by_x, by_y):
        """
        Points sorted by x and y, and the span of the points on the x-axis)
        """
        # end cases
        if len(by_x) == 1:
            return None
        elif len(by_x) == 2:
            return (by_x[0], by_x[1])

        # divide
            # find a midpoint by looking at the middle x value
        mid = int(len(by_x) / 2)
        mid_point = by_x[mid]

            # find all the sorted point indexes for each side
        left_by_x = by_x[:mid]
        left_by_y = filter(lambda i: points[i][x_ind] < points[mid_point][x_ind], by_y)

        right_by_x = by_x[mid:]
        right_by_y = filter(lambda i: points[i][x_ind] >= points[mid_point][x_ind], by_y)

        # conquer
        l_pair = closest_pair_recurse(left_by_x, left_by_y)
        r_pair = closest_pair_recurse(right_by_x, right_by_y)

        # combine
            # find which side has the smaller distance pair
        try:
            l_dist = fake_euclid(points[l_pair[0]], points[l_pair[1]])
        except TypeError:
            l_dist = float("inf") # if one point, then infinite distance
        try:
            r_dist = fake_euclid(points[r_pair[0]], points[r_pair[1]])
        except TypeError:
            r_dist = float("inf")

        if l_dist < r_dist:
            dist = l_dist
            closest_pair = l_pair
        else:
            dist = r_dist
            closest_pair = r_pair

            # find the strip in the middle within the distance
        y_strip = filter(lambda i: abs(points[left_by_x[-1]][x_ind] - points[i][x_ind])
                         < dist, by_y)

            # Loop through all the points in the strip and compare
        for key, val in enumerate(y_strip):
            # loop through the next 15 elements
            for i in xrange(key+1, key+1+15):
                try:
                    d = fake_euclid(points[val], points[y_strip[i]])
                    if d < dist:
                        dist = d
                        closest_pair = (val, y_strip[i])
                except IndexError:
                    pass

        return closest_pair

    # sort by x and y, but only store the indices
    by_x = range(len(points))
    by_x.sort(key=lambda x: points[x][x_ind])
    by_y = range(len(points))
    by_y.sort(key=lambda x: points[x][y_ind])

    # return the correct values
    c = closest_pair_recurse(by_x, by_y)

    # map back to the point x,y values
    return tuple(points[i] for i in c)

def slow_closest_pair(points):
    """
    Finds the closest pair using a brute force algorithm. Slower than the above
    algorithm, but it's simpler so it's handy for testing the above algorithm.
    """
    dist = float('inf')
    closest_pair = None
    for x in points:
        for y in points:
            if x != y:
                d = fake_euclid(x, y)
                if d < dist:
                    dist = d
                    closest_pair =(x, y)
    return closest_pair

def parse_points_file(file_name):
    f = open(file_name)
    if f:
        points = []
        for l in f:
            try:
                points.append(tuple(map(int, l.split(','))))
            except:
                pass
        return points
    else:
        return []
```

```python
def plot(points, closest=None):
    import matplotlib.pyplot as plt

    # setup the figure
    fig = plt.figure()
    ax = fig.add_subplot(111)

    # plot the points
    x, y = zip(*points)
    ax.plot(x, y, 'ro')

    # plot the closest pair
    if closest is not None:
        x, y = zip(*closest)
        ax.plot(x, y, 'go-')

    # set additional settings and show plot
    ax.grid()
    ax.set_xlim(-5,10)
    ax.set_ylim(-5,10)
    ax.set_title('Closest Pair')
    plt.show()

def main():
    # parse command line arguments
    parser = argparse.ArgumentParser(
        description="Find the closest pair of points given many points.")
    parser.add_argument('--graph', dest='graph', # nargs='?',
                        const=True, default=False, action='store_const',
                        help='Graph the points and closest pair.')
    parser.add_argument('filename', metavar='FILE', type=str,
                        help='The file to parse the points out of. Expects a '+\
                            'file of lines with comma seperated x/y values.')
    args = parser.parse_args()

    # open the file and run the algorithm
    points = parse_points_file(args.filename)

    if points:
        #print "Working on points:  " + repr(points)
        closest_points = closest_pair(points)

        # really_closest_points = slow_closest_pair(points)
        # if fake_euclid(*closest_points) != fake_euclid(*really_closest_points):
        #     raise Exception("There is a problem with the closest pair algorithm")

        print "Closest:  " + repr(closest_points)

        if args.graph:
            plot(points, closest_points)
    else:
        print "Can't parse points file."

if __name__ == "__main__":
    main()
```

```python
# Counting Inversion

import sys
import argparse

def counting_inversion_brute(nums):
    swaps = []
    for i in xrange(len(nums)):
        for j in xrange(i, len(nums)):
            if nums[i] > nums[j]:
                swaps.append((nums[i], nums[j]))
                nums[j], nums[i] = nums[i], nums[j]
    return swaps

def counting_inversion_divide_and_conq(nums):
    def merge_and_count(a, b):
        i = j = 0
        c = []
        count = 0

        while i < len(a) and j < len(b):
            c.append(min(a[i], b[j]))
            if b[j] < a[i]:
                count += len(a) - i
                j += 1
            else:
                i += 1

        # append the list of b onto a
        if (i >= len(a)) and j < len(b):
            c += b[j:]
        elif (j >= len(b)) and i < len(a):
            c += a[i:]
        return count, c

    def sort_and_count(l):
        if len(l) == 1:
            return 0, l
        else:
            # divide l into a and b
            mid = len(l)/2
            a = l[:mid]
            b = l[mid:]

            r_a, a = sort_and_count(a)
            r_b, b = sort_and_count(b)
            r, l = merge_and_count(a, b)

            return r_a + r_b + r, l

    return sort_and_count(nums)[0]

def parse_numbers_file(filename):
    nums = []
    with open(filename) as f:
        for l in f:
            try:
                nums.append(int(l))
            except ValueError:
                pass
        return nums

def main():
    # parse command line arguments
    parser = argparse.ArgumentParser(
        description="The number of inversions to get a sorted sequence.")
    parser.add_argument('filename', metavar='FILE', type=str,
                        help='The file to parse the numbers out of. Expects a '+\
                             'file of on number on each line.')
    args = parser.parse_args()

    # open the file and run the algorithm
    nums = parse_numbers_file(args.filename)

    # swaps = counting_inversion_brute(nums[:])
    # # print "Brute force #:  ", len(swaps)
    # print "Sequence:  <%s>" % (', '.join(map(str, nums)),)
    # print "Swaps (found with brute force):"
    # for swap in swaps:
    #     print "Swap:  %s" % (repr(tuple(swap)),)
    # print

    num_swaps = counting_inversion_divide_and_conq(nums[:])
    print "# Swaps (Found with divide and conquer):", num_swaps

if __name__ == "__main__":
    main()
```

```python
# Convex-Hull

import sys
import argparse
import math

# point tuple indexes
p_x = 0
p_y = 1

def area2(a, b, c):
    """ The area of a triangle with three points. """
    return ((b[p_x] - a[p_x])*(c[p_y] - a[p_y]) -
           (b[p_y] - a[p_y])*(c[p_x] - a[p_x]))

def is_left_of(a, b, c):
    """ Checks if the point a is to the left of the line from b to c. """
    return area2(a, b, c) > 0

def is_colinear(a, b, c):
    """ Checks if the point a is collinear with the line from a to b. """
    return area2(a, b, c) == 0

def is_right_of(a, b, c):
    """ Checks if the point a is to the right of the line from b to c. """
    return area2(a, b, c) < 0

def brute(points):
    """
    Find the convex hull of a number of points using a brute for method.
    """
    hull_pairs = []
    # for every two points
    for i in xrange(len(points)):
        for j in xrange(len(points)):
            if i != j:
                # Go through all the points and check if they're all on one side
                on_right = False # if we have points on the right
                on_left = False # if we have points on the left
                colinear = False # if we have colinear points

                # for all points that are not part of the line we're considering
                for p in points:
                    if p != points[i] and p != points[j]:
                        a = area2(p, points[i], points[j])
                        if a > 0:
                            on_left = True
                        elif a < 0:
                            on_right = True
                        else:
                            colinear = True

                # check if we have situations where all points are on one side
                # or colinear (all points are in a line so the convex hull is
                # the line)
                if (on_right and not on_left and not colinear) \
                   or (on_left and not on_right and not colinear) \
                   or (colinear and not on_right and not on_left):
                    hull_pairs.append((points[i], points[j]))

    return hull_pairs


def vector(p1, p2):
    """ Returns the vector of two points. """
    return (p2[p_x] - p1[p_x], p2[p_y] - p1[p_y])

def flip(x, y):
    return y, x

def graham_scan(points):
    n = len(points)

    # find the anchor point (O(n))
    anchor = None
    for i, p in enumerate(points):
        if anchor is None or p[p_y] < points[anchor][p_y] or \
           (p[p_y] == points[anchor][p_y] and p[p_x] < points[anchor][p_x]):
           # go ahead and set the anchor
           anchor = i

    # sort the remaining points by creating a new list of indecies that are
    # sorted (O(nlogn))
    p_srtd = [x for x in xrange(n) if x != anchor]
    p_srtd.sort(
        key=lambda x: math.atan2(*flip(*vector(points[anchor], points[x])))
    )

    # insert the anchor point
    print "anchor", anchor, points
    points.insert(0, points[anchor])

    # insert a sentinal point
    points.insert(0, points[0])
    print len(points), points

    points = [points[i] for i in p_srtd if isinstance(i, int)]

    # push the first three points onto the stack
    m = 2
    i = 3
    while i < n:
        while area2(points[m-1], points[m], points[i]) <= 0:
            print 'i', i
            print 'm', m
            if m == 2:
                points[m], points[i] = points[i], points[m]
                i += 1
            else:
                m -= 1
        m += 1
        points[m], points[i] = points[i], points[m]

        i += 1

    return points

def gift_wrap(points):
    """
    Find the convex hull for a number of points using the gift
    wrapping method, also known as the Jarvis march method.
    """
    # degenerate cases
    if len(points) == 1 or len(points) == 2:
        return points

    hull = []
    # find the leftmost point
```

```python
    point_on_hull = (float('inf'), float('inf'))
    for p in points:
        if (p[p_x] < point_on_hull[p_x]) or \
           (p[p_x] == point_on_hull[p_x] and p[p_y] < point_on_hull[p_y]):
            point_on_hull = p

    i = 0 # TODO: maybe take out
    while True:
        hull.append(point_on_hull)

        # initial endpoint candidate
        endpoint = points[0]
        for j in xrange(1,len(points)):
            if endpoint == point_on_hull or \
               is_left_of(points[j], point_on_hull, endpoint):
                endpoint = points[j]
        i = i + 1
        point_on_hull = endpoint

        # end case
        if endpoint == hull[0]: # we got to the beginning of the hull
            break

    return hull

def parse_points_file(file_name):
    f = open(file_name)
    if f:
        points = []
        for l in f:
            try:
                points.append(tuple(map(int, l.split(','))))
            except:
                pass
        return points
    else:
        return []

def plot(points, hull):
    import matplotlib.pyplot as plt

    # setup the figure
    fig = plt.figure()
    ax = fig.add_subplot(111)

    # plot the points
    x, y = zip(*points)
    ax.plot(x, y, 'ro')

    # check if we're plotting hull pairs or an ordered point list
    print hull
    if isinstance(hull[0][0], tuple):
        # we have hull pairs
        # loop through the connections
        for h in hull:
            # plot the hull
            x, y = zip(*h)
            ax.plot(x, y, 'go-')
    else:
        # we have a list of hull points
        # close the hull
        hull.append(hull[0])

        # plot the hull
        x, y = zip(*hull)
        ax.plot(x, y, 'go-')

    # set additional settings and show plot
    x, y = zip(*points)
    # the fudge factor represents the spacing around the plot so it looks nicer
    fudge_factor = math.ceil((max(x) + max(y))/2 * 0.1)
    ax.grid()
    ax.set_xlim(min(x)-fudge_factor,max(x)+fudge_factor)
    ax.set_ylim(min(y)-fudge_factor,max(y)+fudge_factor)
    ax.set_title('Convex Hull')
    plt.show()

def main():
    # A dict of the supported methods
    methods = {
        'gift_wrap': gift_wrap,
        'brute': brute,
        'graham_scan': graham_scan,
    }

    # parse command line arguments
    parser = argparse.ArgumentParser(
        description="Find the convex hull of points using varius methods.")
    parser.add_argument('--graph', dest='graph',# nargs='?',
                        const=True, default=False, action='store_const',
                        help='Produce a graph.')
    parser.add_argument('method', metavar='METHOD', type=str,
                         help='The type of method to use. Supported methods are: '+\
                              ', '.join(methods))
    parser.add_argument('filename', metavar='FILE', type=str,
                        help='The file to parse the points out of. Expects a '+\
                             'file of lines with comma seperated x/y values.')
    args = parser.parse_args()

    # open the file and parse out the points
    points = parse_points_file(args.filename)

    # if we have points let's git-r-done
    if points:
        try:
            result = methods[args.method](points)
        except KeyError:
            sys.stderr.write("Invalid method selected\n")
            sys.exit(1)

        print "Convex hull: " + repr(result)

        if args.graph:
            plot(points, result)
    else:
        print "Can't parse points file."

if __name__ == "__main__":
    main()
```