Nikolai Alexander
Final Project

December 8, 2018
CSCI 3104, Algorithms

1. (15 pts) Doctor Jean needs your help! She's been relabeling her collection of chromosome sequences and has found three different sequences that were displaced from their original locations in the lab. She knows that two of the sequences represent two chromosomes from a human (Homo Sapiens) and the other sequence represents a chromosome from a soybean (Glycine Max).

   Help Doctor Jean by parsing in the following sequences into memory (be careful of newlines!) and applying the Longest Common Subsequence algorithm learned in recitation to each unique pair of sequences.

   Unzip the "sequence_data.zip" file located on Canvas to access the data. When backtracing through the computed two dimensional matrix to find the longest common subsequence, break ties uniformly at random. Compare the results from your implementation to determine which species the sequences pertain to (try to come up with a sensible metric that will compare the results).

   (a) Show a table that maps the sequence to the species.

Sequence	**Species**
Sequence_A	Soybean
Sequence_B	Human
Sequence_C	Human

		x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	...	x[3495]	x[3496]	x[3497]	x[3498]	x[3499]	x[3500]
		_	G	G	T	T	T	...	A	A	T	C	A	A
y[0]	_	0	0	0	0	0	0	...	0	0	0	0	0	0
y[1]	T	0	0	0	1	1	1	...	1	1	1	1	1	1
y[2]	A	0	0	0	1	1	1	...	2	2	2	2	2	2
y[3]	A	0	0	0	1	1	1	...	3	3	3	3	3	3
y[4]	C	0	0	0	1	1	1	...	4	4	4	4	4	4
y[5]	C	0	0	0	1	1	1	...	5	5	5	5	5	5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
y[3495]	C	0	1	2	3	4	5	...	2015	2015	2015	2016	2016	2017
y[3496]	T	0	1	2	3	4	5	...	2015	2015	2016	2016	2016	2017
y[3497]	G	0	1	2	3	4	5	...	2015	2015	2016	2016	2016	2017
y[3498]	C	0	1	2	3	4	5	...	2015	2015	2016	2017	2017	2017
y[3499]	C	0	1	2	3	4	5	...	2015	2015	2016	2017	2017	2017
y[3500]	T	0	1	2	3	4	5	...	2015	2015	2016	2017	2017	2017

Figure 1: Longest Common Subsequence table comparing Sequence_A and Sequence_B.

	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	...	x[3495]	x[3496]	x[3497]	x[3498]	x[3499]	x[3500]
	_	G	G	T	T	T	...	A	A	T	C	A	A
y[0]	_	0	0	0	0	0	0	...	0	0	0	0	0	0
y[1]	C	0	0	0	0	0	0	...	1	1	1	1	1	1
y[2]	G	0	1	1	1	1	1	...	2	2	2	2	2	2
y[3]	T	0	1	1	2	2	2	...	3	3	3	3	3	3
y[4]	A	0	1	1	2	2	2	...	4	4	4	4	4	4
y[5]	T	0	1	1	2	3	3	...	5	5	5	5	5	5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
y[3425]	T	0	1	2	3	4	5	...	1981	1982	1983	1983	1983	1983
y[3426]	A	0	1	2	3	4	5	...	1982	1982	1983	1983	1984	1984
y[3427]	C	0	1	2	3	4	5	...	1982	1982	1983	1984	1984	1984
y[3428]	A	0	1	2	3	4	5	...	1983	1983	1983	1984	1985	1985
y[3429]	T	0	1	2	3	4	5	...	1983	1983	1984	1984	1985	1985
y[3430]	G	0	1	2	3	4	5	...	1984	1984	1984	1984	1985	1985

Figure 2: Longest Common Subsequence table comparing Sequence_A and Sequence_C.

	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	...	x[3495]	x[3496]	x[3497]	x[3498]	x[3499]	x[3500]
	_	T	A	A	C	C	...	C	T	G	C	C	T
y[0]	_	0	0	0	0	0	0	...	0	0	0	0	0	0
y[1]	C	0	0	0	0	1	1	...	1	1	1	1	1	1
y[2]	G	0	0	0	0	1	1	...	2	2	2	2	2	2
y[3]	T	0	1	1	1	1	1	...	3	3	3	3	3	3
y[4]	A	0	1	2	2	2	2	...	4	4	4	4	4	4
y[5]	T	0	1	2	2	2	2	...	5	5	5	5	5	5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
y[3425]	T	0	1	2	3	4	5	...	2430	2431	2431	2431	2431	2432
y[3426]	A	0	1	2	3	4	5	...	2430	2431	2431	2431	2431	2432
y[3427]	C	0	1	2	3	4	5	...	2431	2431	2431	2432	2432	2432
y[3428]	A	0	1	2	3	4	5	...	2432	2432	2432	2432	2432	2432
y[3429]	T	0	1	2	3	4	5	...	2432	2433	2433	2433	2433	2433
y[3430]	G	0	1	2	3	4	5	...	2432	2433	2434	2434	2434	2434

Figure 3: Longest Common Subsequence table comparing Sequence_B and Sequence_C.

The sequences with the largest *Longest Common Subsequence* are `Sequence_B` and `Sequence_C`. Since we know two of the three sequences are human and the third is a soybean, we can determine that `Sequence_B` and `Sequence_C` represent chromosomes from a human and `Sequence_A` represents a chromosome from a soybean.

(b) Submit your separate python (.py) file along with your PDF submission.
Python code is in `FinalProject_Alexander.py` under *Problem 1* with outputs for the `Longest Common Sequence`e tables for each comparison, the subsequences produced by each comparison, and the species that the sequences are attached to.

2. (15 pts) Draco Malfoy is struggling with the problem of making change for n cents using the smallest number of coins. Malfoy has coin values of $v_1 < v_2 < \cdots < v_r$ for r coins types, where each coin's value v_i is a positive integer. His goal is to obtain a set of counts $\{d_i\}$, one for each coin type, such that $\sum_{i=1}^{r} d_i = k$ and where k is minimized.

   (a) A greedy algorithm for making change is the **wizard's algorithm**, which all young wizards learn. Malfoy writes the following pseudocode on the whiteboard to illustrate it, where n is the amount of money to make change for and v is a vector of the coin denominations:

```
wizardChange(n,v,r) :
   d[1 .. r] = 0       // initial histogram of coin types in solution
   while n > 0 {
      k = 1
      while ( k < r and v[k] > n ) { k++ }
      if k==r { return 'no solution' }
      else { n = n - v[k] }
   }
   return d
```

   Hermione snorts and says Malfoy's code has bugs. Identify the bugs and explain why each would cause the algorithm to fail.

   There is a of number bugs I found in the code:

   i. In the nested while loop, `v[k] > n` should be `v[k] <= n`. You can't make change with a coin that is of higher value than the change you need. Also, in the case that `v[0]` is less than n (which is highly likely), it would just never enter that while loop.

   ii. The coditional statement for no solution `if k = r` should be `if v[0] > n` for returning no solution. We can still technically find a solution by using the largest coin. However, if we cannot reduce `n` to 0 using the smallest valued coin `v[0]`, then we cannot find a solution for exact change.

   iii. Also for this same conditional statement conditional statement, `if v[0] > n` should be moved outside of the while loop. If the value of the smallest coin is greater than the value of `n`, then there is no solution, and there is no point in iterating through the larger coins in `v`.

   iv. (*Tenative*) Unless I'm misreading this line of pseudocode, `return 'no solution'` should be `return NULL`. `return 'no solution'` is returning a string, when we want to return no value if we cannot get the correct change. However, if this is an accepted way to write `NULL` in pseudocode, then this is not a bug.

v. In `else` of the conditional statement, `n = n - v[k]` should be `n = n - v[k-1]`. Since we go through the nested while loop until we reach a coin larger than `n`, we need to backtrack 1 index to the largest coin that is less than `n`. We also need to make sure we call the largest element in `v` if we reach the end of the list. So, the errors this bug would bring forth are returning the wrong change completely, or giving an overrun index error.

vi. The last bug I found was also in the else part of the conditional statement: Nothing is added to `d` in the function. Therefore, we need to add `d[k-1] += 1` either right before or right after `n = n - v[k-1]`.

So, with all of the bugs identified and fixed, the new algorithm should look like:

```
wizardChange(n,v,r) :
     d[0:r] = 0
     while n > 0 {
          k = 1
          if v[0] > n { return NULL }
          while ( k < r and v[k] <= n ) { k++ }
          d[k-1] += 1
          n = n - v[k-1]
     }
     return d
```

(b) Sometimes the goblins at Gringotts Wizarding Bank run out of coins, and make change using whatever is left on hand. Identify a set of U.S. coin denominations for which the greedy algorithm does not yield an optimal solution. Justify your answer in terms of optimal substructure and the greedy-choice property. (The set should include a penny so that there is a solution for every value of n.)

A *greedy algorithm* gives an optimal solution when the following properties are true:

i. *Greedy Choice Property*: there is always an optimal solution that starts with greedy choice.

ii. *Optimal Substructure Property*: the optimal solution to a problem contains within it optimal solutions to subproblems.

Consider a set of U.S. coin denominations `v = [0.01, 0.10, 0.25, 0.50, 1.00]`. We can prove that this will not always produce and optimal solution with a proof by contradiction:

Consider the case, where the `wizard's algorithm` is trying to return change for \$1.80. The first choice the greedy algorithm makes is \$1.00, reducing the change

to \$0.80. Its next choice is \$0.50, reducing the change to \$0.30. After that, it chooses \$0.25, since that is the closest coin to \$0.30. Finally it iterates through the loop five more times, choosing \$0.01. This gives us the set:

$$[1.00, 0.50, 0.25, 0.01, 0.01, 0.01, 0.01, 0.01] = 8 \text{ coins}$$

However, the optimal solution is actually:

$$[1.00, 0.50, 0.10, 0.10, 0.10] = 5 \text{ coins}$$

This violates the *Greedy Choice Property* as the greedy choice of \$0.25 to reduce \$0.30 is not the optimal choice to produce the change in the least amount of coins. This also violated the *Optimal Substructure Property* as the greedy choice for \$0.30 is not an optimal substructure of \$1.00. Therefore, for coin set [0.01, 0.10, 0.25, 0.50, 1.00], the greedy algorithm does not always yield an optimal solution.

(c) On the advice of computer scientists, Gringotts has announced that they will be changing all wizard coin denominations into a new set of coins denominated in powers of c, i.e., denominations of $c^0, c^1, \ldots, c^\ell$ for some integers $c > 1$ and $\ell \geq 1$. (This will be done by a spell that will magically transmute old coins into new coins, before your very eyes.) Prove that the wizard's algorithm will always yield an optimal solution in this case.

Hint: first consider the special case of $c = 2$.

We can prove that the wizard's algorithm will always yield an optimal solution for a set of coins denominated by base-c by Induction. In order so show that this algorithm works for this case, we need to show that it follow the *Greedy Choice Property* and the *Optimal Substructure Property*, which are again defined as follows:

i. *Greedy Choice Property*: there is always an optimal solution that starts with greedy choice.

ii. *Optimal Substructure Property*: the optimal solution to a problem contains within it optimal solutions to subproblems.

Proof by Induction

Base Case: $c = 2$

First, we need to prove that the *Optimal Substructure Property* holds true. Consider a set of coins $C = [1, 2, 4, 8, ..., 2^{\ell}]$ and an optimal set of coins $d[1, ..., k]$ such that:

$$\sum_{i=1}^{k} d_i = n, \text{ where } k \text{ is minimized.}$$

Now consider a subset $d'[1, ..., k']$ such that,

$$\sum_{i=1}^{k'} d'_i = n', \text{ where } k' \text{ is minimized,}$$

where $d' \subseteq d$. $k' < k$, and $n' < n$. Since all values of C follow the property,

$$C[m] = 2\,C[m-1], \text{ where } m = 2^{\ell},$$

for all values of C. Since all coins are factors of 2 larger than their predecessors, an optimal substructure will hold for all optimal structures of d. Therefore, the *Optimal Substructure Property* holds for a set of coins denominated by base-2

Now, we can need to prove the *Greedy Choice Property*. Consider the same set of coins $C = [1, 2, 4, 8, ..., 2^{\ell}]$ where $\ell \geq 1$. Suppose we choose the maximum value of C such that,

$$max(c) \leq n,$$

and continue to choose values in this fasion until we get a set,

$$\sum_{i=1}^{k} d_i = n, \text{ where } k \text{ is minimized.}$$

. Previously, we stated all values of C follow the property:

$$C[m] = 2\,C[m-1], \text{ where } m \geq 1,$$

So, every value contained in C can be broken down into combinations of smaller values contained in C. Since this is true, we can find the optimal solution by only picking $max(c) \leq n$, and the *Greedy Choice Property* holds true.

Since the *Optimal Substructure Property* and the *Greedy Choice Property* hold true for a set of coins $C = [1, 2, 4, 8, ..., 2^{\ell}]$, the *wizard's algorithm* by definition will always yeild an optimal solution in this case.

Induction Hypothesis:
For a set of coin denominations $C = [c^0, c^1, c^2, ..., c^\ell]$ where $c > 1$ and $\ell \geq 1$, the *Optimal Substructure Property* and *Greedy Choice Property* hold true for the *wizard's algorithm*. Thus, the *wizard's algorithm* will always yield an optimal solution.

Inductive Step: $c = n + 1$
Assuming the Induction Hypothesis holds true for n, consider a set of coins $C = [(n+1)^0, (n+1)^1, (n+1)^2, ..., (n+1)^\ell]$ and a number p for which we must find an optimal set of coins d contained in C such that,

$$\sum_{i=1}^{k} d_i = p, \text{ where } k \text{ is minimized.}$$

In order to prove that the *wizard's algorithm* always produces an optimal solution for set C, we must show that it follows the *Optimal Substructure Property* and the *Greedy Choice Property*.
To show the *Optimal Substructure Property* holds true, Consider the subset d' such that,

$$\sum_{i=1}^{k'} d'_i = p', \text{ where } k' \text{ is minimized,}$$

where $d' \subseteq d$. $k' < k$, and $p' < p$. The values of C hold a similar pattern to that of the base case,

$$C[m] = (n+1)C[m-1], \text{ where } m \geq 1,$$

This property allows us to find optimal solutions to the subproblem p' since all values of C are all factors of $(n+1)$ of each other. Thus, an optimal substructure will hold and the *Optimal Substructure Property* always hold true for the *wizard's algorithm*.
For the *Greedy Choice Property*, now consider we build a subset of C where we repeatedly choose the maximum value contained in C such that

$$max(c) \leq p,$$

until the sum of the elements in subset d is equal to p. Because each value in C is a factor $(n+1)$ apart from it's relative elements, each value in C can be built from combinations of smaller elements. As a result of this property, we will always get the optimal substructure when picking the maximum value of C less than p. Therefore, the *Greedy Choice Property* holds true.

Because the *wizard's algorithm* fulfills both the *Optimal Substructure Property* and the *Greedy Choice Property*, it will always yield an optimal solution for any set of coin denominations of a power of c.

3. In the two-player game "Pandas Peril", an even number of cards are laid out in a row, face up. On each card, is written a positive integer. Players take turns removing a card from either end of the row and placing the card in their pile. The player whose cards add up to the highest number wins the game. One strategy is to use a greedy approach and simply pick the card at the end that is the largest. However, this is not always optimal, as the following example shows: (The first player would win if she would first pick the 4 instead of the 5.)

   4 2 10 5

   (a) (10 pts) Write a dynamic programming algorithm for a strategy to play Pandas Peril. Player 1 will use this strategy and Player 2 will use a greedy strategy of choosing the largest card.

```
PandasPeril_Dynamic(A) :
     sum1 = A[0]
     sum2 = A.length - 1
     choice = NULL

     // Game 1 - if we choose the first card
     i1 = 1
     j1 = A.length-1
     // Player 2's Turn
     if A[i1] >= A[j1] {
          i1++
     }
     else{
          j1--
     }
     while i1 < j1 {
          // Player 1's Turn
          if A[i1] >= A[j1] {
               sum1 += A[i1]
               i1++
          }
          else {
               sum1 += A[j1]
               j1--
          }
```

```
        // Player 2's Turn
        if A[i1] >= A[j1] {
            i1++
        }
        else {
            j1--
        }
}

//Game 2 -  if we choose the last card
i2 = 0
j2 = A.length-2
//  Player 2's turn
if A[i2] >= A[j2] {
        i2++
}
else {
        j2--
}
// Continue the game until there are no cards left
while i2 < j2 {
        // Player 1's Turn
        if A[i2] >= A[j2] {
            sum2 += A[i2]
            i2++
        }
        else {
            sum2 += A[j2]
            j2--
        }

        // Player 2's Turn
        if A[i2] >= A[j2] {
            i2++
        }
        else {
            j2--
        }
}
```

```
// Choose card that produces highest sum
if sum1 >= sum2 {
     choice = A[0]
     delete A[0]
     return choice
}
else {
     choice = A[A.length-1]
     delete A[A.length-1]
     return choice
}
```

(b) (10 pts) Prove that your strategy will do no worse than the greedy strategy for maximizing the sum of each hand.

Proof by Induction

Base Case: $n = 2$

Consider a set of cards `A` of size 2. Our dynamic strategy `PandasPeril_Dynamic` picks 2 cards `A[0]` and `A[1]`. It then simulates two games, each starting with either `A[0]` or `A[1]` respectively. In each game, the first card is assigned to Player 1, and then Player 2 chooses a card. The two players then continue to greedily select and remove cards from `A` until there are no cards left in `A`. After both games complete, the sums of the hands for Player 1 in each game are compared. Since for a set of size 2, the highest sum would be of the larget card in `A`, the card chosen by `PandasPeril_Dynamic` would be,

`PandasPeril_Dynamic = max(A[0],A[2]).`

For the greedy strategy `PandasPeril_Greedy`, the card selected is always the card with the value such that,

`PandasPeril_Greedy = max(A[0],A[2]).`

Therefore,

`PandasPeril_Dynamic = PandasPeril_Greedy.`

Induction Hypothesis:

Given a set of cards `A` of size n, where $n \geq 2$ and is even, the dynamic strategy for maximizing the sum of a hand `PandasPeril_Dynamic` produces a sum greater than or equal to the sum of a hand produced by the greedy strategy `PandasPeril_Greedy`.

Inductive Step: $n+2$

Now consider a set of cards `A` of size $n+2$.Our dynamic strategy `PandasPeril_Dynamic` takes two cards `A[0]` and `A[n+1]` and simulates two games - one where `A[0]` is chosen first, and the other where `A[n+1]` is chosen first. In each game, Player 1 receives the first card, then Player 2 chooses the largest of the next two cards, and then each player takes turns selecting the largest card until there are no cards left in `A`. After both games have completed, the sums of each of Player 1's sets for each game are compared, and the card between `A[0]` and `A[n+1]` that produces the maximum sum is selected. This approach is continued until `A` is empty in the actual game.

The greedy strategy `PandasPeril_Greedy` just chooses each card such that,

$$\texttt{PandasPeril\_Greedy += max(A[0],A[A.length-1])}$$

until `A` is empty. Whereas `PandasPeril_Dynamic` makes it's choice based off the equation

$$\texttt{PandasPeril\_Dynamic} \mathrel{+}= \begin{cases} \texttt{A[0]} & iff \quad \texttt{sum1} \geq \texttt{sum2} \\ \texttt{A[n+1]} & iff \quad \texttt{sum2} > \texttt{sum1} \end{cases}$$

Where,

$$\begin{aligned} \texttt{sum1} &= \texttt{A[0]} + \textstyle\sum_{\texttt{i = len(A)-1}}^{0} \texttt{PandasPeril\_Greedy(A[0],A[i-1])} \\ \texttt{sum2} &= \texttt{A[n+1]} + \textstyle\sum_{\texttt{i = len(A)-1}}^{0} \texttt{PandasPeril\_Greedy(A[0],A[i-1])} \end{aligned}$$

Thus, if `PandasPeril_Greedy`'s choice produces the optimal sum, then,

$$\texttt{PandasPeril\_Dynamic = PandasPeril\_Greedy.}$$

Otherwise,

$$\texttt{PandasPeril\_Dynamic > PandasPeril\_Greedy,}$$

by definition. Therefore, `PandasPeril_Dynamic` produces a sum greater than or equal to the sum of a hand produced by `PandasPeril_Greedy`, and our Induction Hypothesis holds.

(c) (10 pts) Implement your strategy and the greedy strategy in Python and simulate a game, or two, of Pandas Peril. Your simulation should include a randomly generated collection of cards and show the sum of cards in each hand at the end of the game.

Python code is in `FinalProject_Alexander.py` under *Problem 3: Pandas Peril* with outputs for 2 games, the randomly generated sets for each game, the set chosen by each player, the total sums of each set, and the winner of each game.

4. A common problem in computer graphics is to approximate a complex shape with a bounding box. For a set, S, of n points in 2-dimensional space, the idea is to find the smallest rectangle, R. with sides parallel to the coordinate axes that contains all the points in S. Once S is approximated by such a bounding box, computations involving S can be sped up. But, the savings is wasted if considerable time is spent constructing R, therefore, having an efficient algorithm for constructing R is crucial.

   (a) (10 pts) Design a divide and conquer algorithm for constructing R in $O(\frac{3n}{2})$ comparisons.

```
// For a set of points S
Smallest_Rectangle(S) :
     // Separate the x coordinates and y coordinates and find their
     // minimum and maximum values in the set. Returns a set of two
     // numbers where the the first is the minimum value and the second
     // is the maximum value.
     x = MinMax_Merge(S[ ,0])
     y = MinMax_Merge(S[ ,1])

     // Assign coordinates of rectangle corners
     bottom_left = [x[0],y[0]]
     bottom_right = [x[0],y[1]]
     top_left = [x[1],y[0]]
     top_right = [x[1],y[1]]

     // Return a matrix of the coordinates of the four corners of the rectangle
     R = [bottom_left, bottom_right, top_left, top_right]
     return R


// Based off Merge-Sort method
MinMax_Merge(A) :
     if A.length > 1 {
          p = floor(A.length/2)

          // Divide
          A1 = MinMax_Merge(A[0:p])
          A2 = MinMax_Merge(A[p:A.length])

          // Conquer
```

```
            return MinMax_Compare(A1,A2)
        }

        return A

    MinMax_Compare(A1,A2) :
        // Array to store minimum and maximum values
        minmax_coords = [0,0]

        if A1.length = 1 and A2.length = 1 {
            minmax_coords[0] = min(A1[0], A2[0])
            minmax_coords[1] = max(A1[0], A2[0])
        }
        else if A1.length {
            minmax_coords[0] = min(A1[0], A2[0])
            minmax_coords[1] = max(A1[0], A2[1])
        }
        else if A2.length {
            minmax_coords[0] = min(A1[0], A2[0])
            minmax_coords[1] = max(A1[1], A2[0])
        }
        else {
            minmax_coords[0] = min(A1[0], A2[0])
            minmax_coords[1] = max(A1[1], A2[1])
        }

        return minmax_coords
```

This algorithm works similarly to Merge-Sort, however, instead of sorting an array, it is recursively comparing the minimum and maximum values of the list. It works by dividing the array by halves until there is only 1 element left in each array. It then begins comparing the minimum and maximum elements with the respective minimum and maximum elements from the array it previously split from. For the first comparison, makes 1 comparison for each of the elements, and stores the minimum of the two in the first element of an array of size 2, and stores the maximum of the two in the second element of the min-max array. It then returns the min-max array. The algorithm then works it's way back up the tree, recursively comparing the minimum values from two sets, and the maximum values from both sets respectively, and keeps the minimum and maximum values of those two set of comparisons. Since, the minimum values are only being compared

with other minimum values and maximum values are only being compared with other maximum values, the total possible number of comparisons is $\frac{3n}{2}$

(b) (10 pts) Implement your algorithm in Python. Generate 50 points randomly and show that your bounding box algorithm correctly bounds all points generated. Your code should output the list of points, as well as the coordinates of R. You should include an explanation of the results in your pdf file with your algorithm.

Python code is in `FinalProject_Alexander.py` under *Problem 4: Smallest Bounding Rectangle* with outputs for the set `S` of 50 points with coordinates x and y, and the coordinates of the smallest rectangle containing those points `R`.

Since the edges of the rectangle must be parallel to their respective coordinates, the dimensions of the rectangle are limited to the minimum and maximum x and y values in the set. Therefore, the y-coordinate of the top horizontal edge goes through the coordinate(s) with the maximum y-value while the bottom horizontal edge goes through the point with the minimum y-coordinate. The same applies to the horizontal edges.

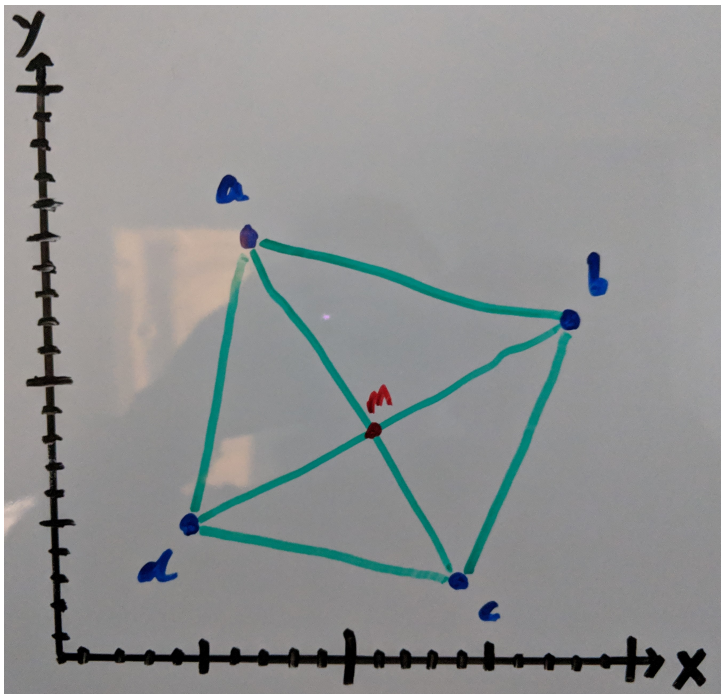
5. (10 pts) Professor Voldemort has designed an algorithm that can take any graph G with n vertices and determine in $O(n^k)$ time whether G contains a clique of size k. Has the Professor just shown that $P = NP$? Why or why not?

   Voldemort has **not** shown us that $P = NP$. A *non-deterministic polynomial algorithm* (NP) is a decision algorithm that produces a Boolean solution, in polynomial time, as to whether a solution to a problem can be found. However, a *polynomial algorithm* (P) is a search algorithm that produces an actual solution to a problem in polynomial time.

   Though Voldemort's NP algorithm is able to check to see if graph G contains a clique of size k, it does nothing to prove that the clique was actually found through a search algorithm in polynomial time. Additionally, as of right now there is no proof to show that P is theoretically equivalent to NP in all cases. Therefore, Voldemort's NP algorithm is not enough to demonstrate that the clique of size k was also found in polynomial time $O(n^k)$.

6. (10 points) Consider the special case of TSP where the vertices correspond to points in the plane, with the cost defined for an edge for every pair (p, q) being the usual Euclidean distance between p and q. Prove that an optimal tour will not have any pair of crossing edges.

   The *Traveling Salesman Problem* looks for an optimal path that crosses every point and returns back to the source point with the smallest possible weight. In order to prove that an optimal tour does not have any pair of crossing paths, we need to prove that the sum of the crossing paths are longer than the sum of any two outer paths. Now, in general, TSP does not require weights between two vertices to be symmetric in both directions ($W_{(p,q)} \neq W_{(q,p)}$), or for the weights to satisfy the *Triangle Inequality Theorem*. However, because in this case, the weights of the edges are the Euclidean distances between two points, this becomes a geometric problem and must satisfy both the above properties. We are going to use the *Triangle Inequality Theorem* to prove our hypothesis. For simplicity, consider the below quadrilateral:



For this quadrilateral , we need to find the optimal path through all four points. In order to use the *Triangle Inequality Theorem*, we can place a theoretical point m where (a,c) and (b,d) cross. There are three possible undirected paths we can take:

$$[(a,b),(b,c),(c,d),(d,a)],$$

$$[(a,b),(b,d),(d,c),(c,a)],$$

$$[(a,d),(d,b),(b,c),(c,a)].$$

We need to show that:

$$sum(W_{(a,b)}, W_{(b,c)}, W_{(c,d)}, W_{(d,a)}) < sum(W_{(a,b)}, W_{(b,d)}, W_{(d,c)}, W_{(c,a)}),$$

$$sum(W_{(a,b)}, W_{(b,c)}, W_{(c,d)}, W_{(d,a)}) < sum(W_{(a,d)}, W_{(d,b)}, W_{(b,c)}, W_{(c,a)}),$$

where $W_{(p,q)}$ is the distance between points p and q. So, the *Triangle Inequality Theorem* states for a triangle $\triangle ABC$,

$$W_{AB} + W_{AC} < W_{BC},$$

$$W_{AB} + W_{BC} < W_{AC},$$

$$\text{and } W_{AC} + W_{BC} < W_{AB}.$$

Now, we can use this to prove, $sum(W_{(a,b)}, W_{(b,c)}, W_{(c,d)}, W_{(d,a)}) < sum(W_{(a,b)}, W_{(b,d)}, W_{(d,c)}, W_{(c,a)})$. Using point m, where (a,c) and (b,d) cross, we can see that,

$$W_{(a,m)} + W_{(c,m)} = W_{(a,c)},$$

$$\text{and } W_{(b,m)} + W_{(d,m)} = W_{(b,d)}.$$

Also, since edges (a,b) and (c,d) are in both paths, we only need to show,

$$W_{(b,c)} + W_{(d,a)} < W_{(b,d)} + W_{(c,a)}.$$

So, we can see that,
$W_{(b,c)} < W_{(b,m)} + W_{(c,m)}$ for triangle $\triangle cbm$ and,
$W_{(a,d)} < W_{(a,m)} + W_{(d,m)}$ for triangle $\triangle adm$.
Adding these two relations we get,

$$W_{(b,c)} + W_{(a,d)} < W_{(b,m)} + W_{(c,m)} + W_{(a,m)} + W_{(d,m)}$$

$$W_{(b,c)} + W_{(a,d)} < (W_{(a,m)} + W_{(c,m)}) + (W_{(b,m)} + W_{(d,m)})$$

$$W_{(b,c)} + W_{(a,d)} < W_{(a,c)} + W_{(b,d)}$$

Thus,

$$sum(W_{(a,b)}, W_{(b,c)}, W_{(c,d)}, W_{(d,a)}) < sum(W_{(a,b)}, W_{(b,d)}, W_{(d,c)}, W_{(c,a)})$$

holds true. Now we can do the same for the other relation. Since edges (b,c) and (a,d) are in both paths, we need to show,

$$W_{(a,b)} + W_{(c,d)} < W_{(b,d)} + W_{(c,a)}.$$

So, we can see that,
$W_{(a,b)} < W_{(a,m)} + W_{(b,m)}$ for triangle $\triangle abm$ and,
$W_{(c,d)} < W_{(c,m)} + W_{(d,m)}$ for triangle $\triangle cdm$.
Adding these relations together,

$$W_{(a,b)} + W_{(c,d)} < W_{(a,m)} + W_{(b,m)} + W_{(c,m)} + W_{(d,m)}$$

$$W_{(a,b)} + W_{(c,d)} < (W_{(a,m)} + W_{(c,m)}) + (W_{(b,m)} + W_{(d,m)})$$

$$W_{(a,b)} + W_{(c,d)} < W_{(a,c)} + W_{(b,d)}$$

Therefore, both our above hypotheses hold true, and we have proved that an optimal tour will not have any pair of crossing edges in a TSP where the weights of the edges are the Euclidean distance between two points.