CSCI 3104 Algorithms Final Project David Skrenta

1) 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 new-lines!) 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 back-tracing 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.

(A, B): 2017
(A, C): 1985
(B, C): 2434

B and C are the most similar meaning they should be labeled as human since we are expecting 2 similar sequences and 1 relatively unsimilar sequence.

Sequence	Species
A	Soybean (Glycine Max)
B	Human (Homo Sapiens)
C	Human (Homo Sapiens)

b) Submit your separate python (.py) file along with your PDF submission.

```
def lcs(x, y):
 len_x = len(x) # m
 len_y = len(y) # n

 l = [[None] * (len_y + 1) for i in range(len_x + 1)]

 for i in range(len_x + 1):
   for j in range(len_y + 1):
```

```
      if i == 0 or j == 0:
        l[i][j] = 0
      elif x[i - 1] == y[j - 1]:
        l[i][j] = l[i - 1][j - 1] + 1
      else:
        l[i][j] = max(l[i - 1][j], l[i][j - 1])
 return l[len_x][len_y]

def problem1():
 a_file = open('sequence_A.fa', 'r')
 b_file = open('sequence_B.fa', 'r')
 c_file = open('sequence_C.fa', 'r')

 A = a_file.read()
 B = b_file.read()
 C = c_file.read()

 a_file.close()
 b_file.close()
 c_file.close()

 A = ''.join(A.split()).replace('>', '')
 B = ''.join(B.split()).replace('>', '')
 C = ''.join(C.split()).replace('>', '')

 print('(A, B):', str(lcs(A, B))) # 2017
 print('(A, C):', str(lcs(A, C))) # 1985
 print('(B, C):', str(lcs(B, C))) # 2434

 # B and C are human (Homo Sapiens), A is soybean (Glycine Max) because b
and c have the longest common substring

problem1()
```

2) 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 < · · · < v r for r coins types, where each coin's value v i is a positive P integer. His goal is to obtain a set of counts {d i }, one for each coin type, such that ri=1 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.

- No modification of d takes place which will always result in an empty anwer being returned
    - Without modifying d, an empty result will be returned as the solution
- The greedy algorithm starts looking at the smallest coin denominations instead of the largest first
    - The solution will always be composed of large multiples of the smallest denomination which satisfy the amount given. For example if 10 were given as n and 1 was the smallest denomination the answer given would be 1x10.

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.)

Set of available US denominations = [1, 10, 25]

Sample non-optimal solution:
N = 30
Solution: 25, 5, 1, (d = 3) optimal solution: 3x10 (d = 1)

Optimal substructure: The optimal substructure property states an optimal solution to the problem contains an optimal solution to subproblems. The optimal subsubstructure does hold true for this problem because the optimal solution can be seen as the combination of several

optimal sub solutions to sub problems. The algorithm produces a non optimal solution because it must greedily choose the largest denominations first without considering groups of smaller denominations.

Greedy-choice property: The greedy-choice property states a global optimum can be arrived at by selecting a local optimum. In this case the greedy-choice property is not satisfied because the local optimum will lead to an incorrect non optimal answer due to the set of provided denominations.

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 , . . . , c ` for some integers c > 1 and ` ≥ 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.

2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 16
10^0 = 1, 10^1 = 10, 10^2 = 100, 10^3 = 1000, 10000

The wizard's algorithm will always yield an optimal solution in these cases because the lower denominations are exact multiples of the higher ones which never results in a no solution or an non optimal solution. If a certain amount of current cannot fit into a denomination it will always be able to be covered by the next one because the multiples exist for all base values of c. There will never be a case where the algorithm passes an optimal solution because of the exponential nature the denominations are organized.

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) 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.

```
greedy(cards) {
        left = cards[0]
        right = cards[cards.length - 1]

        if (left >= right) {
```

```
                return cards.pop(0)
        }
        else {
                return cards.pop(0)
        }
}

player1() {
        left = cards[0]
        right = cards[cards.length - 1]
        nextRight = cards[cards.length - 2]
        nextLeft = cards[1]

        if (left >= right) {
                if (right + nextLeft > left + nextRight) {
                        return cards.pop()
                }
                else {
                        return cards.pop(0)
                }
        }
        else {
                if (right + nextLeft < left + nextRight) {
                        return cards.pop(0)
                }
                else {
                        return cards.pop()
                }
        }

}

main() {
        cards = [4, 2, 10, 5]
        player1_sum = 0
        player2_sum = 0

        while(cards.length > 0) {
                player1_sum += player1(cards)
                player2_sum = greedy(cards)
        }

        print player1_sum, player2_sum
```

}

main()

Player 1: 14
Player 2: 7

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

My strategy will do no worse than the greedy strategy because if it is unable to identify a move two moves ahead it defaults back to the largest between the cards available which is what the greedy strategy does. My strategy will always consider what the greedy strategy would do given the current cards before making a move and or planning the next.

c) 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.

```
import random

def problem3():
 def greedy(cards):
   left = cards[0]
   right = cards[len(cards) - 1]

   if left >= right:
     return cards.pop(0)
   else:
     return cards.pop()

 def player1(cards):
   left = cards[0]
   right = cards[len(cards) - 1]
   nextLeft = 0
   nextRight = 0

   try:
     nextLeft = cards[1]
   except:
```

```
    None

  try:
    nextRight = cards[len(cards) - 2]
  except:
    None

  if left >= right:
    # player 2 will pick left
    if (right + nextLeft) > (left + nextRight):
      return cards.pop()
    else:
      return cards.pop(0)
  else:
    # player 2 will pick right
    if (right + nextLeft) < (left + nextRight):
      return cards.pop(0)
    else:
      return cards.pop()

def main():
  cards = []
  for x in range(0, 10):
    cards.append(random.randint(1, 25))
  initialCards = cards.copy()
  player1_sum = 0
  player2_sum = 0

  while len(cards) > 0:
    player1_sum += player1(cards)
    player2_sum += greedy(cards)

  print('Cards', initialCards)
  print('Player 1:', str(player1_sum))
  print('Player 2:', str(player2_sum))
 main()
```

```
problem3()
```

Simulated games:

Cards [17, 18, 14, 15, 6, 14, 4, 13, 25, 1]
Player 1: 66
Player 2: 61

Cards [10, 9, 18, 16, 1, 1, 7, 22, 23, 5]
Player 1: 59
Player 2: 53

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) Design a divide and conquer algorithm for constructing R in O(3n/2) comparisons.

- Build a convex hull for the given points
    - Use the monotone chain convex hull algorithm
- Draw the rectangle parallel to coordinate axis
    - Take the max_x, max_y, min_x, and min_y from the convex hull and build the four bounding box coordinates
    - Coords = [(min_x, max_y), (min_x, min_y), (max_x, max_y), (max_x, min_y)]

b) 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.

```
import random

def convex_hull(points):
 """Computes the convex hull of a set of 2D points.

 Input: an iterable sequence of (x, y) pairs representing the points.
 Output: a list of vertices of the convex hull in counter-clockwise order,
```

```
   starting from the vertex with the lexicographically smallest
coordinates.
 Implements Andrew's monotone chain algorithm. O(n log n) complexity.
 """

 # Sort the points lexicographically (tuples are compared
lexicographically).
 # Remove duplicates to detect the case we have just one unique point.
 points = sorted(set(points))

 # Boring case: no points or a single point, possibly repeated multiple
times.
 if len(points) <= 1:
   return points

 # 2D cross product of OA and OB vectors, i.e. z-component of their 3D
cross product.
 # Returns a positive value, if OAB makes a counter-clockwise turn,
 # negative for clockwise turn, and zero if the points are collinear.
 def cross(o, a, b):
   return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

 # Build lower hull
 lower = []
 for p in points:
   while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
       lower.pop()
   lower.append(p)

 # Build upper hull
 upper = []
 for p in reversed(points):
   while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
     upper.pop()
   upper.append(p)

 # Concatenation of the lower and upper hulls gives the convex hull.
```

```
 # Last point of each list is omitted because it is repeated at the
beginning of the other list.
 return lower[:-1] + upper[:-1]

def randCoords(n):
 def newPoint():
   return (random.uniform(0, 100), random.uniform(0, 100))

 points = []

 for x in range(1, n):
   points.append(newPoint())

 return points

def problem4():
 points = randCoords(50)
 convex_hull_points = convex_hull(points)

 max_x = max(convex_hull_points, key=lambda x: x[0])[0]
 max_y = max(convex_hull_points, key=lambda x: x[1])[1]
 min_x = min(convex_hull_points, key=lambda x: x[0])[0]
 min_y = min(convex_hull_points, key=lambda x: x[1])[1]

 r_coords = [
   (min_x, max_y),
   (min_x, min_y),
   (max_x, max_y),
   (max_x, min_y)
 ]

 print(points)
 print(r_coords)

problem4()
```

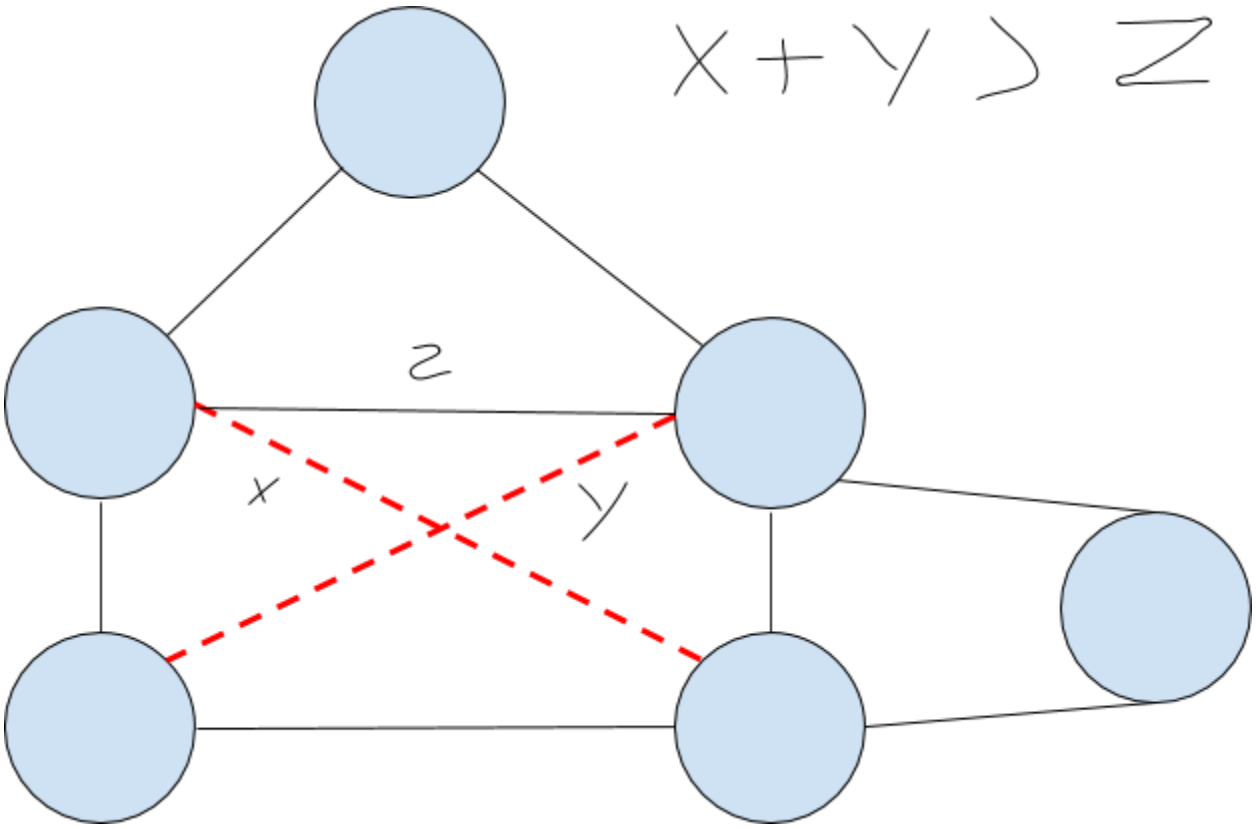
Explanation of results: Given the convex hull, we can easily identify the max and min values which will allow us to form a bounding box with sides parallel to the x and y axis. The monotone chain algorithm builds a convex hull by first sorting all of the points in lexicographical order and then builds the upper and lower portions of the hull allowing a final shape to be returned.

5) 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?

The professor has shown that P = NP because the given algorithm is able to find a the largest clique of the graph in O(n^k) which is polynomial time. Any other problem setup to find another clique of a smaller size than the max is a less complex problem because less vertices are visited. The professor has proved that P = NP because of the time complexity given for his clique finding algorithm.

6) 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.

Consider a setup where the path crosses itself:



Since two edges have crossed we know that a polygon with at least 4 vertices exists. The polygon will have a smaller perimeter due to the triangle inequality formed by this problem. An optimal tour cannot have crossing edges because it would be using the vertices in a way that produced a less efficient path than just connecting the vertices into a polygon. The triangle inequality states that given the sum of any two sides it must be greater or equal to the length of the remaining side. Given the polygon is the shorter path between these points and the based on the triangle inequality, an optimal tour of p will not have a pair of crossing edges.