Problem Set 6

All parts are due Friday, May 9 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Your Name: Eric Klinkhammer

Collaborators: George Cheng, Yang Dai

Part A

Problem 6-1. Problem 6-1

(a) Naive Recursive Approach
This problem asks to find the shortest path of length k between two nodes. We know that the shortest path of length k must have, as its final node, a node that connects with a single edge to the target node. We can then ask the question, if we know the shortest path of length k-1 to every node that connects to the target node, which of those nodes would be in the best path. Once the node is identified, we can repeat this reasoning with that node as the target node.

```
def DP_Biking(depth, source, target, reverse_adj_list):
  min = infinty
  if ( depth == 0 and source == target ):
    return 0
  else if ( depth == 0 ):
    return infinity
  for u in reverse_adj_list[target]:
    path_length = DP_Biking(depth - 1, source, u,
        reverse_adj_list) + weight(target, u)
    if ( path_length < min ):
      min = path_length
  return path_length
```

The reverse adjacency list is a list of the reverse of all the edges in a graph (tells what points to each node in the original graph). It can be constructed in $O(|E|)$. Depending on the connectedness of the graph, this approach can become exponentionally impossible, with a running time of $O(n^k)$ because each level of the recursion can create up to n new subproblems per problem.

(b) Dynamic Programming
We can make use of memoization to prevent recalculating values. Within the recursive tree, you can encounter a given node at precisely k unique situations. In other words, for a given node, you only need to know the score of a the shortest path to that node for all k paths (shortest length 1 path, shortest length 2 path $\cdots$ length k path).

```
create table that maps (u, depth) --> shortest path

def DP_Biking(depth, source, target, reverse_adj_list):
  min = infinty
  if ( depth == 0 and source == target ):
    return 0
  else if ( depth == 0 ):
    return infinity
  for u in reverse_adj_list[target]:
    if ( table[u,depth] == null ):
        path_length = DP_Biking(depth - 1,
        source, u, reverse_adj_list) + weight(target, u)
        table[u,depth] = path_length
    else:
        path_length = table[u,depth] + weight(target,u)
    if ( path_length < min ):
      min = path_length
  table[source, depth] = path_length
  return path_length
```

Because of the memoization, you can only make a recursive call if that value in the table has not been filled. However, ever call to the function fills at least one entry in the table. For any given call to the function at a depth k, it will do $O(|E|)$ work (looping through each value). It will then do $O(1)$ work (either with a table lookup or with a base case). This approach will be $O(kE)$, or $O(nk)$. It is psuedo-linear because of the dependence on the problem specifications.

(c) Dynamic Programming: Bottom Up Approach
Using a bottom up approach with dynamic programming, we can use an approach

similar to breath first search.

Starting from the source node, we explore each connected node. If the path through a given node for a given path length is shorter than the existing path length stored for that node, length pair, then that value is stored and that destination node is added to the queue of nodes to be explored (with its k value).

```
k_max is the k of the goal path

best_path_lengths = table that maps
(node, depth) ---> (shortest path, parent)
initizalize best_path_lengths to infinity

best_path_lengths[source, 0] = 0

Q = new Queue()
Q.append( (source, 0) )

while ( Q is not empty ):
  (u, depth) = Q.extract()
  if ( depth >= k_max ):
    continue
  for v in u.adjList():
    dist = weight(u,v) + best_path_lengths[u, depth]
    if ( dist < best_path_lengths[v, depth + 1 ] ):
      best_path_lengths[v, depth + 1] = (dist, u)
      Q.append( (v, depth + 1) )
```

This algorithm explores every possible node up to k times, for a running time of $O(nk)$. This can be verified by observing that there are a total of n*k entries in the memo table, and since the algorithm runs bottom up, each step is guarenteed to have all dependent steps filled, so filling each new table entry is $O(1)$.

(d) Recursive vs Bottom Up

The bottom-up approach explores in all possible directions from the source, while the recurisve version explores in reverse from the destination. While there are situations where both of those can be better or worse, for an example of when the memoized algorithm performs better, consider a graph with only a single path from the source to the destination, but a maze of interconnected nodes unreachable by the destination node. The bottom-up approach will consider all of the nodes, while the recursive approach will only have one choice.

More generally, both approaches suffer in a more fully connected graph, but if the recursive one does better with a connected region near the start node, and the bottom-up one does better with a connected region near the destination. This is because the depth bottoms out (reaches k) before exploring large amounts of nodes (either recursively or adding to queue).

(e) Best Path
To find the best path between two nodes that has a set number of edge lengths, we can simply use the bottom-up algorithm above as is. When it completes, follow the parent pointers back to the source node. Note, that while normally parent points are followed back linearly, this time the parent pointer points to k values, so the proper one must be selected each time. As an example, the parent pointer stored at (destination, k) has, itself, k different parents depending on the length of the path, so once the node in the path is determined, its parent pointer can be found at (node, k-1).

Following the pointers back takes linear time, so finding the best path is dominated by the $O(nk)$ time of the dynamic algorithm. In terms of V and E, this algorithm runs in $O(k(|V| + |E|))$ time.

Problem 6-2. Geometric Progression

(a) Simple Approach
The simplest way to find a geometric progression among a list of numbers is to sort the numbers first, and then, with every pair of numbers, see how many numbers would follow the pair in a geometric progression.

```
n = list of numbers
n = n.sort()
max = 0

for i in range(1,n):
  for j in range(i,n):
    if ( n[j] % n[i] != 0 ):
      continue
    temp_j = n[j]
    count = 2
    for k in range(j,n):
      if ( temp_j % n[k] == 0 and
                n[k] / temp_j == n[j]/n[i] ):
        count++
```

```
        temp_j = n[k]
    if ( count > max ):
      max = count
```

The running time of this algorithm is $O(n^3)$. Because of how inefficient it is, the choice of sorting algorithm isn't important, although you can assume quicksort is used, for a worst case $O(n^2)$ sorting time.

(b) Warm up - Geometric Trios
Given a number j, and a sorted list of elements (call the list N) ranging from 2 to n-1, we need to find all pairs (i,k) such that the triplet (i,j,k) is a geometric series.

A geometric trio of numbers is one in which $\frac{j}{i} = \frac{k}{j}$. Therefore, mapping those quotient values to a list of numbers will associate together the appropriate i and k per quotient entry in the hash table. I think the code will be clearer.

```
n is a list of sorted numbers
j is the middle value in the geometric progession

table maps quotients ---> lists

for x in n:
  if ( j % x == 0 ):
    table[j/x].append(x)
  if ( x % j == 0 ):
    table[x/j].append(x)

pairs = []
for k in table.keys():
  if len(table[k]) == 2:
    pairs.append(table[k])
```

This algorithm runs in $O(n)$ time because it is iterating through a list and doing constant work each time.

(c) Finding Geometric Progressions Efficiently
Now, we will find the size of the largest geometric progression more efficiently.

In part b of this question, we determined that it was possible to construct a table that linked, for a particular number j, the multiplicative factor of the geometric series with a pair of numbers i and k such i,j,k formed a geometric series in $O(n)$ time. This means that such a table could be constructed for all n values in $O(n^2)$ time. Crucially,

access to any element in this table can be done with two successive hashes (or clever indexing), so determining if an given j is in a geometric trio that progresses with a factor of f can be found in $O(1)$ time.

Now, using that table, we will determine the length of the largest sequence. For this step, we will use a table of size n^2 that represents every possible combination of factor and number (if every number could divide every other number, there would be n^2 combinations). The table maps the number-factor pair to a the length of the geometric sequence from that point onward in the sequence. At this point, the subproblem becomes clear. For a given number j, the length of geometric sequence from that point onward is 1 + the length of the geometric series from k.

```
N is a sorted list of numbers.

dict is hash table of hash tables
the first maps numbers in N --> hash table
        that table maps quotients ---> (i,k) pairs

for j in N:
  table_j = dict[j]
  for x in N:
    if ( j % x == 0 ):
      table_j[j/x].append(x)
    if ( x % j == 0 ):
      table_j[x/j].append(x)

len_geo_seq = {}
# maps a (number, factor) tuple to the lenght of the sequence

for j in N.reverse():
  for (i,k) in dict[j]:
    f = k / j
    if ( len_geo_seq[k,f] == null ):
      len_geo_seq(k,f) = 1
    len_geo_seq[j,f] = 1 + len_geo_seq[k,f]

# Need to offset by 1 whatever the max is (since never
#       look at i)
max = 0
for lengths in len_geo_seq:
  if ( lengths > max ):
    max = lengths
```

```
return max + 1
```

The running time of this algorithm is $O(n^2)$. As stated above, the construction of the table of tables of lists is done in $O(n^2)$ time (with $O(n)$ per number). The sorting and reversing of the lists doesn't contribute to the complexity of the algorithm, with run times of $O(n \log n)$ and $O(n)$ respectively. The interesting part of the algorithm, the construction of the table mapping each number and factor to a maximum length sequence runs in $O(n^2)$. There are two ways to think about this problem. The first is by considering the size of the table that has to be constructed - it will be on $O(n^2)$ because there are n numbers and potentially $O(n)$ factors per number. The other way is by looking at the loops. The outer loop loops n times, while the inner loop loops at most n times, for a total of the inner loop body running $O(n^2)$ times. The body runs in constant time because it is just 4 lines of code. Because the array is being traversed backwards, the maximum lengths are being built bottom up in only $O(1)$ operations.

The final step, the traversal of the table to determine the maximum, also takes $O(n^2)$ time since that is the size of the table.

Problem 6-3. Gene Alignment

(a) Highest Scoring Alignment
This problem is very similar to the most common subsequence problem described in lecture, and it is from that lecture that I got most of the ideas for this problem.

When considering two vectors x and y and the highest scoring alignment, one must consider, for each pair of letters, what operation will result in the highest score. The operation score, however, has a component that depends on the remaining vector suffix. This is where dynamic programming comes in. There are only m*n possible combinations of different suffixes, so what would involve many exponential recurisve calls can be done with memoization.

The subproblem can be defined as follows: the highest score for a given suffix of x and y. The solution to that problem can also be defined in terms of that same subproblem (with slightly smaller suffixes). Therefore, the algorithm will be as described below:

```
init table: will map (a,b) ---> score where a
  is the start of a suffix in x, b is the start of
  a suffix in y, and score is the alignment score of
  those suffixes.

x and y are DNA strings
```

```
d is the scoring matrix for a pair of letters, including gaps

def DP_Genes(a,b):
  if ( table(a+1,b) != null ):
    gap_score = d(x[a], '--') + table(a+1,b)
  else:
    gap_score = d(x[a], '--') + DP_Genes(a+1,b)
  if ( table(a+1, b+1) != null): #Can also be mismatch
    match_score = table(a+1, b+1) + d(x[a],y[b])
  else:
    match_score = d(x[a],y[b]) + DP_Genes(a+1,b+1)
  if ( table(a, b+1) != null ):
    other_gap = d('--', y[b] ) + table(a, b+1)
  else:
    other_gap = d('--', y[b] ) + DP_Genes(a, b+1)
  table[a,b] = max(gap_score, match_score, other_gap)
  return table[a,b]
```

THe running time of this algorithm is $O(mn)$. The number of subproblems is mn because there are only nm possible different suffix combinations (m positions in string x, n positions in string y). Each subproblem is $O(1)$ time because, assuming all the subproblems are solved, there are only 12 lines of cod.

(b) Highest Scoring Substrings
This problem is asking for the highest scoring substring of x and y. From the problem above, we have a memo table that records the scores of every suffix combination. From that, determine the maximum score for a given a and b by simply iterating through the table (since it is size mn, this will take $O(mn)$ time). Now, we can repeat the above step with just the reverse of the optimum suffixes to get the indecies that correspond to the end of the substring. This will take order $O((m-a)(n-b)i)$ time.

Starting with strings x and y of length m and n respectively, find the indices a and b that correspond to the maximum scoring suffixes. This means that $x[a:]$ and $y[b:]$ are the best scoring suffixes. Define x' and y' as $x[a:]$ and $y[b:]$. Reverse x' and y'. Repeating the process with x' and y' will get you two new indices, c and d, that correspond to the best length suffixes of x' and y' (so $x'[c:]$ and $y'[d:]$). Those suffixes are the optimum substrings of the original string.

```
table_x_and_y = new table()
# maps (a,b) ---> score

DP_Genes(0,0)
```

```
(a,b) = getMaxIndex(table_x_and_y)

x = x[a:]
y = y[b:]
x.reverse()
y.reverse()

reset table

DP_Genes(0,0)

(c,d) = getMaxIndex(table_x_and_y)

x = [c:]
y = [d:]
x.reverse()
y.reverse()
return x,y

def getMaxIndex(table):
  max = 0
  for (i,j) in table_x_and_y.keys():
    score = table_x_and_y[(a,b)]
    if ( score > max ):
      max = score
      (a,b) = (i,j)
  return (a,b)
```

This approach to the problem just solves the previous problem twice, so it is still $O(mn)$. Reversing strings takes $O(m)$ or $O(n)$. Substring takes $O(1)$ or $O(n)$ depending on whether you copy the string or not. Looking for the max takes $O(mn)$, so in total, this algorithm takes $O(mn)$.