CS/ECE 374 B Fall 2019
Homework 6 Problem 1

Siyuan Chen (siyuanc2)
Megan Walsh (meganew2)
Jiashuo Tong (jtong8)

Solution:

(a) We design a backtracking algorithm for counting the smallest number of bills that is needed to produce a given amount of currency. It is represented as a python program as shown below. The given amount of money is `target` in the function definition, and `denominations` is expected to be a list containing the available denominations. There are two base cases. First, when `target` is 0, the function returns 0 meaning we do not need any bill to make 0 amount of money. Second, when `target` is less than 0, since no bills with the given denominations can make the money, the function will return `None`. Then, we use the backtracking method to reduce the original problem to a simpler problem: In order to compute the number of bills to make `target`, we only need to compute one plus the number of bills to make `target - denomination`. We use a for loop to go through all the available denominations, and add the result of a reduced problem only if it is not `None`. Finally, if `lst` is empty, we realize it is impossible to produce the money and return `None`. Otherwise, we return the smallest number in the list. Comments that explain the program structure are added to the code.

```python
def smallest_num_bills_bt(target,denominations):
   # Base Case 1: When target is 0, return 0
   if target == 0:
      return 0
   # Base Case 2: When target is 0, return None to opt it out
   elif target < 0:
      return None
   lst = [] # A list to include all possible combinations
   # Use a for loop to go through all denominations
   for d in denominations:
      # Check if the result for the reduced problem is None
      if smallest_num_bills_bt(target-denomination,denominations)
          != None:
         # Recursively call the function to compute the result of a
             reduced problem: number of bills needed to produce
             (target - denomination), and append the finit number to
             the list
         lst.append(smallest_num_bills_bt(target-denomination,denominations)+1)
   # If the list is empty, return None as "no solution" to the
      problem
   if lst == []:
      return None
   # Else, return the smallest number of bills in the list
   else:
      return min(lst)
```

As an example, we call `smallest_num_bills_bt(90, [1,4,7,13,28,52,91,365])` to compute the minimum number of bills needed to produce 90 with the denominations: 1 ,4 ,7, 13, 28, 52, 91, 365.

(b) We design a dynamic programming algorithm for finding the smallest number of bills. Variable names "target" and "denominations" have exactly the same meanings as in 1(a).

```python
def smallest_num_bills_dp(target,denominations):
   global records
   if records[target] != -1:
      return records[target]
   lst = []
   for j in denominations:
      if j <= target:
         if smallest_num_bills_dp(target-j,denominations) != None:
            lst.append(smallest_num_bills_dp(target-j,denominations)+1)
         else:
            pass
   if lst == []:
      return None
   else:
      records[target] = min(lst)
   return records[target]
```

(c) We define another function `smallest_num_bills_greedy(target, denominations)` which applies the greedy method to find the smallest number of bills. Function input variables `target` and `denominations` are exactly the same as in 1(a). The base case is when `target` is 0, the return value should certainly be 0. Then, the function creates an array of targets by substracting each denomination from `target` and removes the negative values from the array. Thus, the new target should be the smallest number in this array as it corresponds to the largest denomination that is smaller than the original target. It is possible that this `targets` is an empty array which implies the greedy algorithm fails to find a solution. (It is impossible to fail with the settings of this question since 1 is one of the denominations.) If that's the case, we return `None` as "no solutions found".

```python
def smallest_num_bills_greedy(target,denominations):
   # Base Case: When target is 0, should return 0.
   if target == 0:
      return 0
   # Whe
   targets = (target-denominations)[(target-denominations)>=0]
   if targets.size == 0:
      return None
   else:
      target = min(targets)
      return smallest_num_bills_greedy(target,denominations)+1
```

As an example, we call `smallest_num_bills_greedy(90, [1,4,7,13,28,52,91,365])` to greedily compute the number of bills needed to produce 90 with the denominations: 1 ,4 ,7, 13, 28, 52, 91, 365.

Finally, we run the following block of code to find the smallest amount of money where the greedy approach does not result in the smallest number of bills. The logic is simple: If the result from greedy algorithm is not equal to the result from dynamic programming, we say the greedy algorithm fails.

```
'''
Find the smallest number of bill that where the greedy approach gives a wrong answer
'''
max_t = 1000 # Set a maximum target to define the range for searching
d = np.array([1,4,7,13,28,52,91,365]) # Denominations
records = -np.ones(t+1) # Records stores the results computed in dynamic programming
records[0] = 0
for t in range(max_t):
   # If the dynamic programming result differs from the greedy result, it means greedy algorithm fails to find the smallest number
   if smallest_num_bills_dp(t,d) != smallest_num_bills_greedy(t,d):
      print(t)
```

The smallest amount is found to be 416.

■

CS/ECE 374 B Fall 2019

Homework 6 Problem 2

Siyuan Chen (siyuanc2)
Megan Walsh (meganew2)
Jiashuo Tong (jtong8)

Solution: For this problem we assume we are given a list `words` of length n, containing typeset lengths for the given words. Additionally we use 0-based index for this problem.
We define a recursive function `breakwords(i, L)`, which returns the minimized total slop generated by putting `words[i:]` (from i+1 to nth words inclusive) into a number of lines with maximum length L. This function is presented in code below:

```
import numpy as np
# Randomly initialize some test data
words = np.random.randint(1, 15, size=(10,), dtype=int)

def breakwords(i, L):
    """
    Put words [i, n-1] (inclusive) in lines of max length L, minimizing
       total slop.
    Returns minimized total slop. Indices are 0-based.
    """
    if i >= len(words):
        # Base case: If we have run out of words (i should be in [0,
           n-1]), return slop = 0
        return 0
    else:
        # For an arbitrary line break j = i to n-1, find the optimal
           breakpoint that minimizes the sum of slop of this line and
           slop of all following lines
        return min(slop(i, j, L) + breakwords(j+1, L) for j in range(i,
           len(words)))
```

We use a separate helper function `slop(i, j, L)` to calculate slop of a line containing words i through j (inclusive) in a line of max length L. If the words take too much space and can't fit in the line, `slop(i, j, L)` returns infinity.

```
def slop(i, j, L):
    """
    Calculate slop resulted by putting words [i, j] (inclusive) into a
       line of max length L.
    """
    if ((j-i) + np.sum(words[i:j+1])) <= L:
        if j == len(words) - 1:
       # If we are at the last line, slop is 0
            return 0
        else:
       # Calculate slop as defined in the problem
            return pow(L-(j-i)-np.sum(words[i:j+1]), 3)
    else:
   # Can't fit into a single line, return inf
        return float('inf')
```

To get the final answer, we need to call `breakwords(0, L)`.

We can memoize all function values into a 1-D array `bw[0 .. n]` where `bw[0 .. n-1]` stores results for `breakwords(0..n-1, L)` and `bw[n]=0` is a sentinel node. It represents the base case in the recursive implementation: since $i \in [0, n-1]$, when $i \geq n$ we don't have any words to break and the slop is automatically 0. Each array entry `bw[i]` depends only on the entries immediately to its right: `bw[j+1]` for j = i+1 to n-1. Thus we can fill the array in descending order starting from i=n-1 to 0. The dynamic programming algorithm is then written as follows:

```
def breakwords_dp(L):
   """
   DP version of breakwords(). bw[len(words)] = 0 is a sentinel element.
   """
   bw = np.zeros((len(words)+1,), dtype=int)
   for i in range(len(words)-1, -1, -1): # for i = n-1 to 0
      bw[i] = min(slop(i, j, L) + bw[j+1] for j in range(i, len(words)))
   return bw[0]
```

Assuming that each call to `slop(i, j, L)` is constant time, the algorithm makes at most n calls to `slop(i, j, L)` during its visit to each of the n entries in the memoizing array, we conclude that the algorithm runs in $O(n^2)$ time and takes $O(n)$ space. ■

CS/ECE 374 B Fall 2019
Homework 6 Problem 3

Siyuan Chen (siyuanc2)
Megan Walsh (meganew2)
Jiashuo Tong (jtong8)

Solution:

(a) The modified recursive algorithm to compute the bounded edit distance is presented below with key changes marked in red. Any edit distance that is higher than 5 will be returned as infinity. An additional input argument `curr_cost` has been added to the function tracking the edit cost before the current recursive call.
Given two strings S and T, we know that it takes *at least* `abs(len(S)-len(T))` edits to match them. Therefore, when `curr_cost + abs(len(S)-len(T))` is higher than the bound, we do not need to proceed to calculate the exact number of edits, and therefore this algorithm runs faster than the original algorithm at least in some cases.

```python
def edit_recursive_bounded(S, T, curr_cost=0):
    """ Computes the minimum cost to edit string S to obtain string
        T. The program returns inf if the minimum cost is larger than
        5. """
    if (abs(len(S)-len(T)) + curr_cost > 5):
        # If number of edits already made + length difference between
          S and T is already larger than 5, do not continue
        return float('inf')
    else:
        if len(S) == 0:
            # insert all characters in T
            return len(T)
        if len(T) == 0:
            # delete all characters in S
            return len(S)
        # cost to delete one char of S. Recurse with curr_cost+1
        del_cost = edit_recursive_bounded(S[:-1], T, curr_cost+1) + 1
        # cost to insert one char of T. Recurse with curr_cost+1
        ins_cost = edit_recursive_bounded(S, T[:-1], curr_cost+1) + 1
        if S[-1] == T[-1]:
            # zero cost to match chars. Recurse with curr_cost
            match_cost = edit_recursive_bounded(S[:-1], T[:-1],
                curr_cost)
        else:
            # cost to edit match chars. Recurse with curr_cost+1
            match_cost = edit_recursive_bounded(S[:-1], T[:-1],
                curr_cost+1) + 1
        return min(del_cost, ins_cost, match_cost)
```

(b) The following function is based on Esko Ukkonen's improved string matching algorithm presented in this paper [**?**] on page 105. Given strings S, T, and edit distance bound t, we can define a function `editdistance_ukkonen(S, T, t=5)` that returns minimized bounded edit distance needed to transfer S into T. The edit distance bound *t* is set to 5 as specified by the problem. The function follows the identical recurrence relation as defined in part (a). To get the final answer, we need to call `editdistance_ukkonen(S, T)`.

```
def editdistance_ukkonen(S, T, t=5):
    """
    Computes the bounded minimum cost to edit string S to obtain
        string T. The program returns inf if the minimum cost is
        larger than t.
    """
    if t < abs(len(S)-len(T)):
        # If abs(len(S)-len(T))>t then we know minimum cost can’t be
            smaller than t.
        return float(’inf’)
    # Store the longer string in strM and shorter string in strN. As
        this can be done by reference instead of a deep copy, we
        assume this process is constant time.
    if len(S) >= len(T):
        strM = T
        strN = S
    else:
        strM = S
        strN = T
    m = len(strM)
    n = len(strN)
    p = math.floor((t-abs(len(strM)-len(strN)))/2)
    # Initialize memoization data structure of ((m+1)x(n+1)) size
        with big values (sentinel values).
    ret = np.full((m+1, n+1), t+100, dtype=int)
    # Assume n >= m.
    for i in range(m+1):
        # Computing in row-major order, only visiting locations (i,
            j) satisfying max(0, i-p) <= j <= min(n, i+(n-m)+p)
        for j in range(max(0, i-p), min(n, i+(n-m)+p)+1):
            if i == 0:
                # Insert all chars in N
                ret[i, j] = j
            elif j == 0:
                # Insert all chars in M
                ret[i, j] = i
            else:
                # Cost to delete one char of M
                del_cost = ret[i-1, j] + 1
                # Cost to insert one char of N
                ins_cost = ret[i, j-1] + 1
                if strM[i-1] == strN[j-1]:
                    # Zero cost to match chars
                    match_cost = ret[i-1, j-1]
                else:
                    # Cost to edit one char
                    match_cost = ret[i-1, j-1] + 1
                ret[i, j] = min(del_cost, ins_cost, match_cost)
    if ret[m, n] > t:
        return float(’inf’)
    else:
        return ret[m, n]
```

For simplicity, we denote the shorter of the two strings as M with length m, and the longer string as N with length n.
As shown above, the most straightforward way to implement dynamic programming is to memoize values into a 2D array `ret[0...m+1, 0...n+1]`, where `ret[0...m+1, :]` and `ret[:, 0...n+1]` are base cases where one of the strings have run out of characters and the remaining number of edit becomes the remaining length of the other string.
Each array entry `ret[i, j]` depends only on the entries immediately above and to the left: `ret[i, j-1]`, `ret[i-1, j-1]`, and `ret[i-1, j]`. Thus we can fill the array in row-major order.
Ukkonen's algorithm makes further improvements by only visiting entries close enough to the diagonal, or `ret[i, j]` that satisfies $\max(0, i-p) \leq j \leq \min(n, i+(n-m)+p)$ where $p = \text{floor}(\frac{1}{2}(t-|m-n|))$. Any path from upper left corner to bottom right corder on the dependency graph that passes nodes outside that region must take more than t horizontal and vertical moves, which guarantees the edit cost will exceed the cost bound. Any nodes that does not satisfy the above conditions then becomes base cases and are initialized with a large value. In other words, only elements that belong to the centermost s diagonal lines, where:

$$s = i+(n-m)+p-(i-p) = n-m+2p = n-m+2\times\text{floor}(\frac{t-|m-n|}{2})$$

Although the above algorithm only visits $(m+1)\times(s+1) = O(mn-m^2)$ states in the memoizing array and does $O(1)$ work in each node, it involves initialization of $(m+1)\times(n+1)$ array and therefore cannot run in $o(mn)$ time. However, note that only $(m+1)\times(s+1) = O(mn-m^2)$ elements have been accessed and these entries can be safely mapped into a $(m+1)\times(s+1)$ array, resulting in reduced space and time requirements. Namely, we define a custom mapping that stores meaningful entries `ret[i, j]` that satisfies $\max(0, i-p) \leq j \leq \min(n, i+(n-m)+p)$ where $p = \text{floor}(\frac{1}{2}(t-|m-n|))$, and return a fixed large number if other entries (base cases) are accessed. The custom mapping and modified algorithm that takes advantage of said mapping is presented below.

```python
class mapdiagonals:
    """
    Custom class used to map ret[i, j] into the smaller array. All
        operations are constant time.
    """
    def __init__(self, M, N, t):
        self.m = len(M)
        self.n = len(N)
        self.p = math.floor((t-abs(self.m-self.n))/2)
        # Initialize array with size (m+1)*(n-m+2p+1) to store only
            the interesting entries.
        self.ret = np.full((self.m+1, self.n-self.m+2*self.p+1),
            t+100, dtype=int)
        self.bound = t

    def isvalid(self, i, j):
        # Returns True if (i, j) need to be stored in self.ret, False
            if (i, j) is a base case
        return (j >= max(0, i-self.p)) and (j <= min(self.n,
            i+(self.n-self.m)+self.p))

    def read(self, i, j):
        # Returns saved value for useful entries, upper bound for base cases (entries too far away from diagonal)
        if self.isvalid(i, j):
            return self.ret[i, j-min(self.n, i+(self.n-self.m)+self.p)-1]
        else:
            return self.bound

    def write(self, i, j, val):
        # Save value for useful entries
        if self.isvalid(i, j):
            self.ret[i, j-min(self.n, i+(self.n-self.m)+self.p)-1] = val

def ukkonen_faster(S, T, t=5):
    """
    Computes the bounded minimum cost to edit string S to obtain string T. The program returns inf if the minimum cost is larger than t.
    Takes advantage of the mapping mapdiagonals.
    """
    if t < abs(len(S)-len(T)):
        # If abs(len(S)-len(T))>t then we know minimum cost can’t be smaller than t.
        return False
    # Store the longer string in strM and shorter string in strN. As this can be done by reference instead of a deep copy, we assume this process is constant time.
    if len(S) >= len(T):
        strM = T
        strN = S
    else:
        strM = S
        strN = T
    m = len(strM)
    n = len(strN)
    p = math.floor((t-abs(len(strM)-len(strN)))/2)
    # Initialize memoization data structure of (mxn) size with big values (sentinel values).
    ret = mapdiagonals(strM, strN, t)
    if n >= m:
        for i in range(m+1):
            # Computing in row-major order, only visiting locations (i, j) satisfying max(0, i-p) <= j <= min(n, i+(n-m)+p)
            for j in range(max(0, i-p), min(n, i+(n-m)+p)+1):
                if i == 0:
                    # Insert all chars in N
                    ret.write(i, j, j)
                elif j == 0:
                    # Insert all chars in M
                    ret.write(i, j, i)
                else:
                    # Cost to delete one char of M
                    del_cost = ret.read(i-1, j) + 1
                    # Cost to insert one char of N
                    ins_cost = ret.read(i, j-1) + 1
                    if strM[i-1] == strN[j-1]:
                        # Zero cost to match chars
                        match_cost = ret.read(i-1, j-1)
                    else:
                        # Cost to edit one char
                        match_cost = ret.read(i-1, j-1) + 1
                    ret.write(i, j, min(del_cost, ins_cost, match_cost))
    if ret.read(m, n) > t:
        return float('inf')
    else:
        return ret.read(m, n)
```

To get the final answer, we need to call `ukkonen_faster(S, T)`.
The new algorithm runs in constant time when $|\text{len(S)} - \text{len(T)}| > t$. In other cases, it has a reduced space requirement:

$$(\min(\text{len(S)}, \text{len(T)}) + 1) \cdot (s + 1) = O(s \cdot \min(\text{len(S)}, \text{len(T)}))$$

where

$$s = |\text{len(S)} - \text{len(T)}| + 2 \times \text{floor}(\frac{t - |\text{len(S)} - \text{len(T)}|}{2}) = O(t) \text{ for } t > |\text{len(S)} - \text{len(T)}|$$

The algorithm visits each entry exactly once and does $O(1)$ work. Therefore, both runtime and space are bounded by $O(t \cdot \min(\text{len(S)}, \text{len(T)}))$. Then the algorithm runs in $O(t \cdot n)$ if the two input strings are of length n. Since t is $o(n)$, the runtime is $O(n)$ and we can conclude that both runtime and space taken is $o(n^2)$.

■

References

[1] E. Ukkonen, "Algorithms for approximate string matching," *Information and control*, vol. 64, no. 1-3, pp. 100–118, 1985.