CS/ECE 374 B Fall 2019
Homework 6 Problem 1

Jin Yucheng (yucheng9)
Fan Wenyan (wenyanf2)
Lu Hanxiao (hl4)

Solution:

(a) The algorithm is defined as follows:

def SmallestNumBills(C, B):

if $C = 0$: # Base case where the amount of currency is 0
  return 0
else if $C < 0$: # Base case where the amount of currency is negative
  return ∞
else: # Try every possible bill
  return 1 + min{SmallestNumBills($C - B[1], B$), ..., SmallestNumBills($C - B[n], B$)}

The argument C means the amount of currency, B is the set of all denominations with cardinality n, in problem 1's case, $B[1]$ represents 1, $B[2]$ represents 4, ..., $B[8]$ represents 365.

If the bill we choose is larger than the amount of currency, we simply return ∞ to ignore the situation, if C is 0, there is no choice to draw a bill, so we return 0.

Otherwise, we try every possible bill to make a sequence of decisions, and return the smallest number of bills.

(b) The worst case of this problem is obviously bounded by $\Theta(C)$, for example, we only have two denominations, one is 1, and the other is $C + 1$. A dynamic programming utilizes a 1-d array, and is defined as follows:

def SmallestNumBills(C, B):

for $i \leftarrow 1$ to C:
  $N[i] \leftarrow \infty$
for $j \leftarrow 1$ to n:
  $N[B[j]] \leftarrow 1$
$N[0] \leftarrow 0$
for $i \leftarrow 1$ to C:
  for $j \leftarrow 1$ to n:
    if $C - B[j] < 0$:
      $a_j \leftarrow \infty$
    else:
      $a_j \leftarrow N[C - B[j]]$
  $N[i] \leftarrow$ 1+ min{$a_1, ..., a_n$}
return $N[C]$

N is a 1-d array, in which $N[i]$ records the smallest number of bills to produce an amount of i currency. So for every possible bill $B[j]$, $N[B[j]] = 1$, which means we only need 1 bill to produce $B[j]$ currency. $N[0] = 0$, because 0 indicates no bill. Local variables a_1 to a_n indicate for every possible bill, the corresponding smallest number of bills to produce an amount of $C - B[j]$ currency, if the amount becomes negative when the bill is larger than C, we assign ∞.

For initialization, the time complexity is $O(C + n + 1)$, for the main for-loop, the time complexity is $O(Cn)$, because the number of dominations is limited, so n is a constant, then the total time complexity is $O(C)$.

(c)

```
Bill = [1, 4, 7, 13, 28, 52, 91, 365]

def SmallestNumBills(n):
    N = []
    N.append(0)
    for i in range(1, 366):
        N.append(float("inf"))

    for j in range(len(Bill)):
        N[Bill[j]] = 1

    if n > 365:
        for i in range(n-365):
            N.append(float("inf"))

    A = []
    for i in range(len(Bill)):
        A.append(float("inf"))

    for i in range(1, n+1):
        for j in range(len(Bill)):
            if i - Bill[j] < 0:
                A[j] = float("inf")
            else:
                A[j] = N[i - Bill[j]]
        N[i] = 1 + min(A)

    return N[n]

def SmallestNumBillsGreedy(n):
    count = 0
    while n>0:
        for j in reversed(range(len(Bill))):
            if n - Bill[j] >= 0:
                n -= Bill[j]
                count +=1
                break
    return count
```

```
for i in range(1, 1000):
    if SmallestNumBillsGreedy(i) != SmallestNumBills(i):
        print(i)
```

```
416
```

Implement both algorithm in part (b) and greedy algorithm, 416 is the smallest number that the results of these two algorithms differ. Since the greedy algorithm will give us 7 bills ({365, 28, 14, 7, 1, 1, 1}) but the correct result is 5 {(91, 91, 91, 91, 52)}. ■

CS/ECE 374 B Fall 2019	Jin Yucheng (yucheng9)
Homework 6 Problem 2	Fan Wenyan (wenyanf2)
	Lu Hanxiao (hl4)

Solution:

Reference

[1] E. Demaine, L. W. Sun, and C. E. Leiserson. "Intro to Algorithms Problem Set 8 Solutions". *Massachusetts Institute of Technology*. November 14, 2001.

• Define $MinLength(i,j)$ for typesetting the i^{th} through j^{th} words as:

$$MinLength(i,j) = (j-i) + \Sigma_{k=i}^{j} l(k) \tag{1}$$

which indicates the minimum length required to fit the i^{th} through j^{th} words in a line.

• Define $LineCost(i,j)$ for typesetting the i^{th} through j^{th} words as:

$$LineCost(i,j) = \begin{cases} \infty & \text{if } MinLength(i,j) > L \\ 0 & \text{if } j = n\text{, which indicates the last line} \\ (L-(j-i)-\Sigma_{k=i}^{j} l(k))^3 & \text{otherwise} \end{cases} \tag{2}$$

which returns ∞ if the i^{th} through j^{th} words could not be fitted in a single line, or 0 for the last line, or the slop for any line excluding the last.

• Define the value of the optimal solution in which the total slop (the sum of line slops excluding the last) is minimized as:

$$C(j) = min\{C(i-1) + LineCost(i,j)\}, \text{ where } 1 \leq i \leq j \tag{3}$$

where $C(j)$ denotes the value of the optimal solution of typesetting the 1^{st} through j^{th} words, we need to consider every possible way of line setting to find the optimal solution. Define $C(0) = 0$ as a base case.

• The algorithm for solving this problem is defined as follows:

We construct a 1-d array to store the values from $C(0)$ to $C(n)$. We calculate the value of each element in C from bottom to up since for every $C(j)$, every $C(k)$ (where $0 \leq k < j$) is available by the time when $C(j)$ is being computed.

To record how to divide words into lines, construct another 1-d array, P, where $P(k)$ is the value that minimizes $C(k)$, or:

$$P(k) = argmin(C(k)) = argmin\{C(i-1) + LineCost(i,k)\}, \text{ where } 1 \leq i \leq k \tag{4}$$

so $P(k)$ is the i that minimizes $C(k)$.

• So after the arrays C and P are obtained, $C(n)$ is the optimal cost and the optimal solution is to print the $P(n)^{th}$ to n^{th} words in the last line, the $P(P(n)-1)^{th}$ to $P(n-1)^{th}$ words in the second last line, ..., and so forth.

Computing $LineCost(i,j)$ takes $O(j-i+1)$ time, since we have to add up the length of the i^{th} to j^{th} words. But it can be optimized in $O(1)$ time by constructing a 1-d array, L, where $L(i)$ stores the cumulative sum of lengths of the 1^{st} to i^{th} words.

The time complexity of our algorithm is $O(n^2)$, since the main for-loop for computing C takes $O(n^2)$ time, this is because we should consider each i to determine the minimum cost. The space complexity is $O(n)$ since we constructs three 1-d arrays, namely, C, P, L.

■

CS/ECE 374 B Fall 2019

Homework 6 Problem 3

Jin Yucheng (yucheng9)
Fan Wenyan (wenyanf2)
Lu Hanxiao (hl4)

Solution:

(a) The original EditRecursive(S, T) function takes two arguments, string $S[1...m]$ and string $T[1...n]$, and returns the edit distance of S and T.

def EditRecursive(S, T):

1 : **if** $Length(S) = 0$: # Base case where S is empty

2 : return $Length(T)$ (or n)

3 : **if** $Length(T) = 0$: # Base case where T is empty

4 : return $Length(S)$ (or m)

5 : DelCost ← EditRecursive($S[1...m-1], T[1...n]$) + 1 # Cost of deleting one char in S

6 : InsCost ← EditRecursive($S[1...m], T[1...n-1]$) + 1 # Cost of inserting one char in S

7 : **if** $S[m] = T[n]$: # Matched case where the last char in S matches the last char in T

8 : MatchCost ← EditRecursive($S[1...m-1], T[1...n-1]$)

9 : **else**: # Mismatched case where the last char in S mismatches the last char in T

10 : MatchCost ← EditRecursive($S[1...m-1], T[1...n-1]$) + 1

11 : return min{DelCost, InsCost, MatchCost}

In EditRecursive(S, T), line 1 to line 4 solve the problem for two base cases where S or T is empty, line 5 is the cost of deleting S's last character, line 6 is the cost of inserting a character to the end of S, line 10 is the cost of substituting $S[m]$ with $T[n]$ if $S[m]$ mismatches $T[n]$, and line 8 is the cost of DOING NOTHING if $S[m]$ matches $T[n]$.

Although the above code can handle all 5 cases correctly, I will modify it by considering if we can DO NOTHING first, because DOING NOTHING STRICTLY DOMINATES deletion, insertion, and substitution. This is easy to prove, since if $S[m]$ matches $T[n]$, the edit distance is 0 for substrings $S[m]$ and $T[n]$, we can just consider the edit distance of $S[m-1]$ and $T[n-1]$.

Therefore, I will move the statement "if $S[m] = T[n]$" above, such that we can reduce the number of calls by line 5 (deletion) and line 6 (insertion). The modified algorithm is as follows:

def EditRecursive(S, T):

1 : **if** $Length(S) = 0$:

2 : return $Length(T)$ (or n)

3 : **if** $Length(T) = 0$:

4 : return $Length(S)$ (or m)

5 : **if** $S[m] = T[n]$:

6 : MatchCost ← EditRecursive($S[1...m-1], T[1...n-1]$)

7 : return MatchCost

8 : **else**:

9 : DelCost ← EditRecursive($S[1...m-1], T[1...n]$) + 1

    InsCost ← EditRecursive($S[1...m], T[1...n-1]$) + 1
    MatchCost ← EditRecursive($S[1...m-1], T[1...n-1]$)+ 1
    return min{DelCost, InsCost, MatchCost}

Then we can implement the modified EditRecursive(S, T) function to compute the bounded edit distance, and write a function BoundedEditDist(S, T, B), that takes strings S, T, and a non-negative integer B (B is valid if and only if it is non-negative, 0 means S and T are exactly the same), and returns either ∞ if the edit distance of S and T is greater than B, or the value of edit distance if it is less than or equal to B. Note that after each deletion, insertion, or substitution, we should take $B-1$ as the third argument of our new recursive call.

def BoundedEditDist(S, T, B):

if $Length(S) = 0$:
    if $Length(T) > B$:
        return ∞
    else:
        return $Length(T)$
if $Length(T) = 0$:
    if $Length(S) > B$:
        return ∞
    else:
        return $Length(S)$
if $S[m] = T[n]$:
    MatchCost ← BoundedEditDist($S[1...m-1], T[1...n-1], B$)
    return MatchCost
else:
    DelCost ← BoundedEditDist($S[1...m-1], T[1...n], B-1$) + 1
    InsCost ← BoundedEditDist($S[1...m], T[1...n-1], B-1$) + 1
    MatchCost ← BoundedEditDist($S[1...m-1], T[1...n-1], B-1$)+ 1
    return min{DelCost, InsCost, MatchCost}

Reference

[1] J. Erickson. Algorithm, Chapter 3.7.

(b) I will answer this question for a more general case, such that S is of length m and T is of length n, and the bound is $O(mn)$.

We use a $(m+1) \times (n+1)$ 2-d array, $Edit$, to save the edit distances of substrings of S and T, each entry $Edit[i, j]$ represents the edit distance of $S[1...i]$ and $T[1...j]$. Entry $Edit[i, 0] = i$ which represents i deletions to convert $S[1...i]$ to an empty string. Entry $Edit[0, j] = j$ which represents j insertions to convert an empty string to $T[1...j]$.

The algorithm is as follows:

def EditDist(S, T):

for $j \leftarrow 0$ to n:

```
    Edit[0, j] ← j:
for i ← 1 to m
    Edit[i, 0] ← i:
    for j ← 1 to n:
        ins ← Edit[i, j − 1] + 1
        del ← Edit[i − 1, j] + 1
        if S[i] = T[j]:
            match ← Edit[i − 1, j − 1]
        else:
            match ← Edit[i − 1, j − 1] + 1
            Edit[i, j] ← min{ins, del, match}
return Edit[m, n]
```

The first for-loop takes $O(n)$ time, the second for-loop takes $O(mn)$ time, therefore, the algorithm is bounded by $O(mn)$ time, if S and T both have length n, then it is bounded by $O(n^2)$ time. ■