Programming Assignment 5: Dynamic Programming 1

Revision: April 21, 2019

Introduction

In this programming assignment, you will be practicing implementing dynamic programming solutions. As usual, in some code problems you just need to implement an algorithm covered in the lectures, while for some others your goal will be to first design an algorithm and then implement it.

Learning Outcomes

Upon completing this programming assignment you will be able to:

1. Apply the dynamic programming technique to solve various computational problems. This will usually require you to design an algorithm that solves a problem by solving a collection of overlapping subproblems (as opposed to the divide-and-conquer technique where subproblems are usually disjoint) and combining the results.

2. See examples of optimization problems where a natural greedy strategy produces a non-optimal result. You will see that a natural greedy move for these problems is not safe.

3. Design and implement an efficient algorithm for the following computational problems:

   (a) Implement an efficient algorithm to compute the difference between two files or strings. Such algorithms are widely used in spell checking programs and version control systems.

   (b) Design and implement a dynamic programming algorithm for a novel computational problem.

Passing Criteria: 3 out of 5

Passing this programming assignment requires passing at least 3 out of 5 programming challenges from this assignment. In turn, passing a programming challenge requires implementing a solution that passes all the tests for this problem in the grader and does so under the time and memory limits specified in the problem statement.

Contents

1 Money Change Again

As we already know, a natural greedy strategy for the change problem does not work correctly for any set of denominations. For example, if the available denominations are 1, 3, and 4, the greedy algorithm will change 6 cents using three coins $(4 + 1 + 1)$ while it can be changed using just two coins $(3 + 3)$. Your goal now is to apply dynamic programming for solving the Money Change Problem for denominations 1, 3, and 4.

Problem Description

Input Format. Integer *money*.

Output Format. The minimum number of coins with denominations 1, 3, 4 that changes *money*.

Constraints. $1 \leq money \leq 10^3$.

Sample 1.
    Input:
```
2
```
    Output:
```
2
```
$2 = 1 + 1$.

Sample 2.
    Input:
```
34
```
    Output:
```
9
```
$34 = 3 + 3 + 4 + 4 + 4 + 4 + 4 + 4 + 4$.

2   Primitive Calculator

Problem Introduction

You are given a primitive calculator that can perform the following three operations with the current number x: multiply x by 2, multiply x by 3, or add 1 to x. Your goal is given a positive integer n, find the minimum number of operations needed to obtain the number n starting from the number 1.
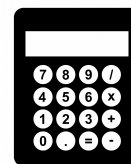
Problem Description

Task. Given an integer n, compute the minimum number of operations needed to obtain the number n starting from the number 1.

Input Format. The input consists of a single integer $1 \le n \le 10^6$.

Output Format. In the first line, output the minimum number k of operations needed to get n from 1. In the second line output a sequence of intermediate numbers. That is, the second line should contain positive integers $a_0, a_2, \ldots, a_{k-1}$ such that $a_0 = 1$, $a_{k-1} = n$ and for all $0 \le i < k - 1$, a_{i+1} is equal to either $a_i + 1$, $2a_i$, or $3a_i$. If there are many such sequences, output any one of them.

Sample 1.
Input:
```
1
```
Output:
```
0
1
```

Sample 2.
Input:
```
5
```
Output:
```
3
1 2 4 5
```
Here, we first multiply 1 by 2 two times and then add 1. Another possibility is to first multiply by 3 and then add 1 two times. Hence "1 3 4 5" is also a valid output in this case.

Sample 3.
Input:
```
96234
```
Output:
```
14
1 3 9 10 11 22 66 198 594 1782 5346 16038 16039 32078 96234
```
Again, another valid output in this case is "1 3 9 10 11 33 99 297 891 2673 8019 16038 16039 48117 96234".

Starter Files

Going from 1 to n is the same as going from n to 1, each time either dividing the current number by 2 or 3 or subtracting 1 from it. Since we would like to go from n to 1 as fast as possible it is natural to repeatedly reduce n as much as possible. That is, at each step we replace n by $\min\{n/3, n/2, n-1\}$ (the terms $n/3$ and $n/2$ are used only when n is divisible by 3 and 2, respectively). We do this until we reach 1. This gives rise to the following algorithm and it is implemented in the starter files:

```
GreedyCalculator(n):
```
$numOperations \leftarrow 0$
```
while n > 1:
```
$\quad numOperations \leftarrow numOperations + 1$
```
  if n mod 3 = 0:
```
$\quad\quad n \leftarrow n/3$
```
  else if n mod 2 = 0:
```
$\quad\quad n \leftarrow n/2$
```
  else:
```
$\quad\quad n \leftarrow n - 1$
```
return numOperations
```

This seemingly correct algorithm is in fact incorrect. You may want to submit one of the starter files to ensure this. Hence in this case moving from n to $\min\{n/3, n/2, n-1\}$ is not *safe*.

3 Computing the Edit Distance Between Two Strings

Problem Introduction

The edit distance between two strings is the minimum number of operations (insertions, deletions, and substitutions of symbols) to transform one string into another. It is a measure of similarity of two strings. Edit distance has applications, for example, in computational biology, natural language processing, and spell checking. Your goal in this problem is to compute the edit distance between two strings.

Problem Description

Task. The goal of this problem is to implement the algorithm for computing the edit distance between two strings.

Input Format. Each of the two lines of the input contains a string consisting of lower case latin letters.

Constraints. The length of both strings is at least 1 and at most 100.

Output Format. Output the edit distance between the given two strings.

Sample 1.
Input:
```
ab
ab
```
Output:
```
0
```

Sample 2.
Input:
```
short
ports
```
Output:
```
3
```
An alignment of total cost 3:

s	h	o	r	t	−
−	p	o	r	t	s

Sample 3.
Input:
```
editing
distance
```
Output:
```
5
```
An alignment of total cost 5:

e	d	i	−	t	i	n	g	−
−	d	i	s	t	a	n	c	e

4 Longest Common Subsequence of Two Sequences

Problem Introduction

Compute the length of a longest common subsequence of two sequences.

Problem Description

Task. Given two sequences $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$, find the length of their longest common subsequence, i.e., the largest non-negative integer p such that there exist indices $1 \leq i_1 < i_2 < \cdots < i_p \leq n$ and $1 \leq j_1 < j_2 < \cdots < j_p \leq m$, such that $a_{i_1} = b_{j_1}, \ldots, a_{i_p} = b_{j_p}$.

Input Format. First line: n. Second line: $a_1, a_2, \ldots, a_n$. Third line: m. Fourth line: $b_1, b_2, \ldots, b_m$.

Constraints. $1 \leq n, m \leq 100$; $-10^9 < a_i, b_i < 10^9$.

Output Format. Output p.

Sample 1.
Input:
```
3
2 7 5
2
2 5
```
Output:
```
2
```
A common subsequence of length 2 is $(2, 5)$.

Sample 2.
Input:
```
1
7
4
1 2 3 4
```
Output:
```
0
```
The two sequences do not share elements.

Sample 3.
Input:
```
4
2 7 8 3
4
5 2 8 7
```
Output:
```
2
```
One common subsequence is $(2, 7)$. Another one is $(2, 8)$.

5 Longest Common Subsequence of Three Sequences

Problem Introduction

Compute the length of a longest common subsequence of three sequences.

Problem Description

Task. Given three sequences $A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_m)$, and $C = (c_1, c_2, \ldots, c_l)$, find the length of their longest common subsequence, i.e., the largest non-negative integer p such that there exist indices $1 \leq i_1 < i_2 < \cdots < i_p \leq n$, $1 \leq j_1 < j_2 < \cdots < j_p \leq m$, $1 \leq k_1 < k_2 < \cdots < k_p \leq l$ such that $a_{i_1} = b_{j_1} = c_{k_1}, \ldots, a_{i_p} = b_{j_p} = c_{k_p}$

Input Format. First line: n. Second line: $a_1, a_2, \ldots, a_n$. Third line: m. Fourth line: $b_1, b_2, \ldots, b_m$. Fifth line: l. Sixth line: $c_1, c_2, \ldots, c_l$.

Constraints. $1 \leq n, m, l \leq 100$; $-10^9 < a_i, b_i, c_i < 10^9$.

Output Format. Output p.

Sample 1.
Input:
```
3
1 2 3
3
2 1 3
3
1 3 5
```
Output:
```
2
```
A common subsequence of length 2 is $(1, 3)$.

Sample 2.
Input:
```
5
8 3 2 1 7
7
8 2 1 3 8 10 7
6
6 8 3 1 4 7
```
Output:
```
3
```
One common subsequence of length 3 in this case is $(8, 3, 7)$. Another one is $(8, 1, 7)$.