CS/ECE 374 A ✧ Fall 2025

Homework 6

Due Tuesday, October 14, 2025 at 9pm Central Time

Please make sure that you read and understand the standard dynamic programming rubric.

---

1. A *triumph* in a sequence of integers (from the Latin *tri-* meaning "three" and *-umph* meaning "bodacious") is a consecutive triple of sequence elements whose sum is a multiple of 3. For example, the sequence

$$\langle 3, \underline{1, 4,} \overline{\underline{1, 5,} 9}, 6, 2, 3, 5, \underline{8, 9, 7}, 9, 3, 2, 3, \underline{8,} \overline{\underline{4, 6,} 2}, 6 \rangle$$

   contains five triumphs (indicated by lines above and below).

   We say that one sequence A is *more triumphant* (or *less heinous*) than another sequence B if there are more triumphs in A than in B.

   Describe and analyze an algorithm to compute the number of triumphs in the most triumphant (or equivalently, least heinous) ***subsequence*** of a given array $A[1..n]$ of integers. For example:

   - Given the input array $[0, 1, 0, 1, 0]$, your algorithm should return the integer 1, which is the number of triumphs in the subsequence $[0, 0, 0]$. Notice that $0, 0, 0$ is a triumph *in the optimal subsequence* even though those three 0s are not consecutive *in the input array*.
   - Given the input array $[0, 1, 1, 2, 3, 5, 8, 13, 21]$, your algorithm should return the integer 4, which is the number of triumphs in the most triumphant subsequence $[0, 1, 2, 3, 8, 13, 21]$. Again, $0, 1, 2$ and $3, 8, 13$ are triumphs *in the optimal subsequence* even though they are not consecutive triples *in the input array*. Notice also that the optimal subsequence includes the consecutive triple $2, 3, 8$, even though that triple is not a triumph.

2. Several years after graduating from Sham-Poobanana University, you decide to open a one-day pop-up art gallery selling NFTs, using the following dynamic pricing strategy.

   All NFTs at your gallery have the same advertised price, which you set at the start of the day, but which you can *decrease* later. Customers visit your gallery one at a time. If a customer is willing to pay your current advertised price, they buy one NFT at that price. On the other hand, if your advertised price is too high, the customer will suggest a lower price that they are willing to pay. If you refuse to lower your advertised price, the customer will leave without buying anything. If you agree to lower your advertised price to match their offer, the customer will buy one NFT at the new lower price. Whenever you lower your advertised price, the new lower price stays in effect until you lower it again, or until the end of the day. *You can never increase your advertised price.*

You know your customers extremely well, so you can accurately predict both when each customer will come to the gallery, and how much each customer is willing to pay for one of your NFTs.

Describe and analyze an algorithm that computes the maximum amount of money you can earn using this dynamic pricing strategy. Your input consists of an array $Value[1..n]$, where $Value[i]$ is the amount that the ith customer (in chronological order) is willing to pay for one NFT.

For example, if the input array is $[5, 3, 1, 4, 2]$, your algorithm should output 13, because you can earn $5+3+0+3+2 = 13$ dollars using the prices $[5, 3, 3, 3, 2]$, and this is optimal.

[Hint: Do not assume that the input values $Value[i]$ are constants independent of n. Your prices might be in Hungarian pengő, and one of your customers might be Elon Musk.]

3. ***Practice only. Do not submit solutions.***

(a) Any string can be decomposed into a sequence of palindromes. For example, the string BUBBASEESABANANA ("Bubba sees a banana.") can be broken into palindromes in the following ways (and 65 others):

BUB • BASEESAB • ANANA

B • U • BB • ASEESA • B • ANANA

BUB • B • A • SEES • ABA • N • ANA

B • U • BB • A • S • EE • S • A • B • A • NAN • A

B • U • B • B • A • S • E • E • S • A • B • A • N • A • N • A

Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string BUBBASEESABANANA, your algorithm should return 3.

(b) A ***metapalindrome*** is a decomposition of a string into a sequence of palindromes, such that the sequence of palindrome lengths is itself a palindrome. For example, the string BOBSMAMASEESAUKULELE ("Bob's mama sees a ukulele") has the following metapalindromes (among others):

BOB • S • MAM • ASEESA • UKU • L • ELE

B • O • B • S • M • A • M • A • S • E • E • S • A • U • K • U • L • E • L • E

The length sequences of these metapalindromes are $(3, 1, 3, 6, 3, 1, 3)$ and $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$; notice that both of these sequences are themselves palindromes.

Describe and analyze an efficient algorithm to find the smallest number of palindromes in any metapalindrome for a given string. For example, given the input string BOBSMAMASEESAUKULELE, your algorithm should return 7.

Solved Problems

3. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings **BANANA** and ANANAS in several different ways.

   BANANAANANAS **BAN**ANA**ANA**NAS **B**AN**AN**A**A**NA**NA**S

   Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of the strings **DYNAMIC** and PROGRAMMING:

   PRO**D**G**Y**R**NAM**AMMI**I**N**C**G **DY**PRO**N**G**A**R**M**AMM**IC**ING

   (a) Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

   Solution (design): We define a boolean function $Shuf(i,j)$, which is TRUE if and only if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$. We need to compute $Shuf(m,n)$. The function $Shuf$ satisfies the following recurrence:

$$Shuf(i,j) = \begin{cases} \text{TRUE} & \text{if } i = j = 0 \\ Shuf(0,j-1) \wedge (B[j] = C[j]) & \text{if } i = 0 \text{ and } j > 0 \\ Shuf(i-1,0) \wedge (A[i] = C[i]) & \text{if } i > 0 \text{ and } j = 0 \\ \big(Shuf(i-1,j) \wedge (A[i] = C[i+j])\big) & \\ \qquad \vee \big(Shuf(i,j-1) \wedge (B[j] = C[i+j])\big) & \text{otherwise} \end{cases}$$

   We can memoize this function into a two-dimensional array $Shuf[0..m][0..n]$. Each array entry $Shuf[i,j]$ depends only on the entries immediately above and immediately to the left: $Shuf[i-1,j]$ and $Shuf[i,j-1]$. Thus, we can fill the array in standard row-major order in $\boldsymbol{O(mn)}$ ***time***. ■

Solution (pseudocode): The following algorithm runs in $O(mn)$ *time*.

```
IsShuffle?(A[1..m], B[1..n], C[1..m+n]):
  Shuf[0,0] ← True
  for j ← 1 to n
      Shuf[0,j] ← Shuf[0,j−1] ∧ (B[j] = C[j])
  for i ← 1 to n
      Shuf[i,0] ← Shuf[i−1,0] ∧ (A[i] = B[i])
  for j ← 1 to n
      Shuf[i,j] ← False
      if A[i] = C[i+j]
          Shuf[i,j] ← Shuf[i−1,j]
      if B[i] = C[i+j]
          Shuf[i,j] ← Shuf[i,j] ∨ Shuf[i,j−1]
  return Shuf[m,n]
```

At the end of the algorithm, for all indices i and j, we have $Shuf[i,j] = \text{True}$ if the prefix $C[1..i+j]$ is a shuffle of the prefixes $A[1..i]$ and $B[1..j]$, and $Shuf[i,j] = \text{False}$ otherwise. ■

Rubric: 5 points, standard dynamic programming rubric. **Each of these solutions is separately worth full credit.** These are not the only correct solutions. −½ for reporting running time as $O(n^2)$. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.

(b) Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine *the number of different ways* that A and B can be shuffled to obtain C.

Solution (design): Let $\#Shuf(i,j)$ denote the number of different ways that the prefixes $A[1..i]$ and $B[1..j]$ can be shuffled to obtain the prefix $C[1..i+j]$. We need to compute $\#Shuf(m,n)$.

The $\#Shuf$ function satisfies the following recurrence. Here I am using Iverson bracket notation to convert booleans to integers: For any proposition P, the expression $[P]$ is equal to 1 if P is true and 0 if P is false.

$$\#Shuf(i,j) = \begin{cases} 1 & \text{if } i = j = 0 \\ \#Shuf(0,j-1) \cdot \big[B[j] = C[j]\big] & \text{if } i = 0 \text{ and } j > 0 \\ \#Shuf(i-1,0) \cdot \big[A[i] = C[i]\big] & \text{if } i > 0 \text{ and } j = 0 \\ \big(\#Shuf(i-1,j) \cdot \big[A[i] = C[i]\big]\big) & \\ \qquad + \big(\#Shuf(i,j-1) \cdot \big[B[j] = C[j]\big]\big) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $\#Shuf[0..m][0..n]$. As in part (a), we can fill this array in standard row-major order in ***O(mn) time***. ■

Solution (pseudocode): The following algorithm runs in $O(mn)$ *time*:

```
NumShuffles(A[1..m], B[1..n], C[1..m+n]):
  #Shuf[0,0] ← 1
  for j ← 1 to n
      #Shuf[0,j] ← 0
      if (B[j] = C[j])
          #Shuf[0,j] ← #Shuf[0,j−1]
  for i ← 1 to n
      #Shuf[0,j] ← 0
      if (A[i] = B[i])
          #Shuf[0,j] ← #Shuf[i−1,0]
  for j ← 1 to n
      #Shuf[i,j] ← 0
      if A[i] = C[i+j]
          #Shuf[i,j] ← #Shuf[i−1,j]
      if B[i] = C[i+j]
          #Shuf[i,j] ← #Shuf[i,j] + #Shuf[i,j−1]
  return Shuf[m,n]
```

At the end of the algorithm, for all indices i and j, $\#Shuf[i,j]$ is the number of different ways that the prefixes $A[1..i]$ and $B[1..j]$ can be shuffled to obtain the prefix $C[1..i+j]$. ■

Rubric: 5 points, standard dynamic programming rubric. **Again, each of these solutions is separately worth full credit.** These are not the only correct solutions. −½ for reporting running time as $O(n^2)$. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.