

☞ Homework 6 ☞

Due Tuesday, October 13, 2026 at 9pm Central Time

Please make sure that you read and understand the [standard dynamic programming rubric](#).

Dynamic programming solutions that do not include an English description of the recursive function being evaluated (or the *underlying* recursive function if the solution uses iterative pseudocode) will not receive any credit.

1. You're about to start a new job at a company with an unusual policy for taking time off: you may choose not to work any days you'd like, as long as you don't take off more than 2 days in *any* set of 28 consecutive work days. For example, you may choose to take off your 2nd workday and 20th workday, but then you cannot take off again until your 30th workday, because $30 - 2 = 28$. If you do take off that 30th workday, your next chance to take off will be your $20 + 28 = 48$ th workday.

As part of your onboarding process, you are given descriptions of what work you will be expected to do for the first n workdays (assuming you aren't taking off). You have to declare which of these n workdays you will take off *before* the first day begins. Fortunately, you have a good idea of how much you want to avoid each individual day's work and are able to assign a numeric score to each day representing how much joy you would take from taking off. You want to maximize the sum of scores for the days you choose to take off.

- (a) Describe an algorithm to compute the maximum amount of joy you can get from taking days off across your first n days of work. You may assume your algorithm is given an array $J[1..n]$ where $J[i]$ is the amount of joy received for taking off the i th workday. Scores could be positive, zero (you have nothing better to do those days), or even negative (the yearly Halloween party is supposed to be a blast!).
- (b) Your new company has a career development program where the number of days off and the length of the window limiting their use changes after each promotion. You want a strategy for choosing days to take off that will continue to work after promotion. Describe an algorithm to compute the maximum amount of joy you can get from taking days off across the first n days of work where you can take off at most ℓ days in any set of k consecutive work days. Give your running time using big-Oh notation in terms of n , ℓ , and k . You may assume you're given an array $J[1..n]$ as before along with the values ℓ and k . Assume ℓ is a **small integer constant**, but k is an integer variable such that $1 \leq k \leq n$.

2. Given an array $X[1..k]$ of characters from an arbitrary alphabet, we define a **run** as a maximal non-empty contiguous subarray (i.e., substring) whose characters are all equal. The **flatness** of X is sum of the squares of the lengths of runs in X . For example, $[Y, Y, D, D, P, Y, Y, Y, Y]$ has flatness $2^2 + 3^2 + 1^2 + 4^2 = 30$.

Suppose you are given an array $A[1..n]$ of characters over some fixed alphabet Σ . Describe an algorithm to compute the maximum flatness of any *subsequence* of A .

For example, given the array $[A, B, A, L, C, C, L, D, L, G, O, O, D, A, O]$, your algorithm should return 23 for the subsequence $[A, A, L, L, L, G, O, O, O]$ (this is not the only subsequence with this flatness.)

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); 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**AN**A**ANANAS **BAN**AN**A**ANANAS

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

PRODGYRNAMAMMIINCG **DYPRONGARMAMMICING**

- (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:

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

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 $O(mn)$ 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 m
    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[j] = 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. $-\frac{1}{2}$ 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.

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

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]  $\leftarrow$  1
  for  $j \leftarrow 1$  to  $n$ 
    #Shuf[0,j]  $\leftarrow$  0
    if ( $B[j] = C[j]$ )
      #Shuf[0,j]  $\leftarrow$  #Shuf[0,j-1]
  for  $i \leftarrow 1$  to  $m$ 
    #Shuf[i,0]  $\leftarrow$  0
    if ( $A[i] = B[i]$ )
      #Shuf[i,0]  $\leftarrow$  #Shuf[i-1,0]
    for  $j \leftarrow 1$  to  $n$ 
      #Shuf[i,j]  $\leftarrow$  0
      if  $A[i] = C[i+j]$ 
        #Shuf[i,j]  $\leftarrow$  #Shuf[i-1,j]
      if  $B[j] = C[i+j]$ 
        #Shuf[i,j]  $\leftarrow$  #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. $-\frac{1}{2}$ for reporting running time as $O(n^2)$. 3 points for a slower polynomial-time algorithm; scale partial credit accordingly.