DUE WED FEB 23, 2022 AT 11.59P VIA GRADESCOPE

H4.SOLUTIONS

PROBLEM 1 *Sprungli bar*

You are given an $n \times m$ Sprungli chocolate bar. Your goal is to devise an algorithm A that takes as input (n, m) and returns the minimal number of cuts needed to divide the bar into perfect squares of either 1x1, 2x2, 3x3, ..., jxj. With each cut, you can split the bar either horizontally or vertically. For example, $A(2,3) = 2$ because 2x3 $\rightarrow$ (2x2, 2x1) $\rightarrow$ (2x2, 1x1, 1x1). The first cut is "V2" for vertical cut at index 2, and the second cut is "H1" for horizontal at index 1.

1. Notice that no matter the rectangle, it is always possible to make a perfect square in the first cut. Show that this strategy fails. Namely, show an input size for which the strategy of picking the cut which creates the largest box leads to extra cuts in total.

2. Devise a dynamic programming algorithm which determines the minimal number of cuts and outputs the list of cuts.

Solution:

1. The greedy strategy on input 6x7 takes 6 cuts. (6x7) $\rightarrow$ (6x6, 6x1) $\rightarrow$ (6x6, 1x1, 5x1) $\rightarrow$ ... $\rightarrow$ (6x6, 1x1, 1x1, 1x1, 1x1, 1x1, 1x1) Whereas the bar can be partitioned in 4 cuts. (6x7) $\rightarrow$ (6x3, 6x4) $\rightarrow$ (3x3, 3x3, 6x4) $\rightarrow$ (3x3, 3x3, 4x4, 2x4) $\rightarrow$ (3x3, 3x3, 4x4, 2x2, 2x2)

2. The key to this problem is to realize that the best way to cut the bar is to first either cut horizontally at row $i \in [1, n/2]$ or to cut vertically at column $i \in [1, m/2]$. Then the rest of the bar is cut using the best ways to split those smaller rectangles into perfect squares. Thus we define $C(n, m)$ as the "minimum number of cuts needed to divide a bar of size n by m into perfect squares." We set every $C(i, i) = 0$ and in general decide $C(n, m)$ as

$$C(n,m) = 1 + \min \begin{cases} C(i,m) + C(n-i,m) & \text{for } i \text{ from 1 to } n/2 \\ C(n,i) + C(n,m-i) & \text{for } i \text{ from 1 to } m/2 \end{cases} \quad (1)$$

Note that since each $C(i, j)$ takes $\theta(n + m)$ time to compute and C describes an nxm-dimensional array, the running time of this algorithm is $\theta(nm(n + m)) = \theta(n^2m + nm^2)$. The above approach naturally leads to the following pseudocode:

```
SPRUNGLI(n, m)
 1  Initialize all C(j,k) = ∞ for j ≠ k, and C(i,i) = 0 for i ≤ min(n,m)
 2  Initialize all L(j,k) = ()
 3  for j = 1 to n
 4          for k = 1 to m
 5                  for i = 1 to j/2
 6                          if 1 + C(i,k) + C(j − i,k) < C(j,k)
 7                                  C(j,k) = 1 + C(i,k) + C(j − i,k)
 8                                  L(j,k) = ("Hi, L(i,k), L(j − i,k))
 9                  for i = 1 to k/2
10                          if 1 + C(j,i) + C(j,k − i) < C(j,k)
11                                  C(j,k) = 1 + C(j,i) + C(j,k − i)
12                                  L(j,k) = ("Vi, L(j,i), L(j,k − i))
13  return C(n,m), L(n,m)
```

PROBLEM 2 *Age of War*

We want to play *roughly fair* game in cs5800. You are given an array that holds the weights of n people in the class $W = (w_1, w_2, \ldots, w_n)$. Your goal is to divide n people into two teams such that the total weight of the two teams is as close as possible to equal. Describe such an algorithm and give its running time. The total number of people on each team should differ by at most 1. Assume that M is the maximum weight of a person, i.e., $\forall i, w_i \leq M$. The running time should be $O(n^3 M)$. The output of the algorithm should be the list of people on each team and the difference in weight between the teams.

Solution:

One can use the same solution as the Gerrymander problem. Define the boolean variable $S_{j,k,x,y}$ to represent whether it is possible among the first j people to have k people assigned to team 1 such that team 1 has total weight x and team 2 has total weight y. After completing this table, one can scan the variables $S_{n,n/2\pm1,x,y}$ to find the one with the smallest $|x - y|$ difference. This solution has a higher running time then the following, which is an optimization:

We define a variable that is indexed by the difference in weight between the teams:

$C(w, i, j)$: true if it is possible to achieve a weight difference between team A and team B of w such that there are i players on team A and j players on team B.

Note here it is important that the difference is taken between team A and team B (and thus when it is negative, it represents the case when B weighs more than A).

For the base case, we can set $C(0, 0, 0)$ to be true and all other sub-problems $C(d, 0, 0)$ where $d \in [-M, M]$ to be false.

Consider the assignment of the $k = i + j^{\text{th}}$ person. That person is either assigned to team 1 or team 2. Thus the variable $C_{w,i,j}$ is true if either (a) we add that person to team A and $C_{w-w_k,i-1,j}$ is true, or (b) we add that person to team B and $C_{w+w_k,i,j-1}$ is true. This is because the

$$C_{w,i,j} \leftarrow C_{w-w_k,i-1,j} \vee C_{w+w_k,i,j-1}$$

Notice that when considering the k^{th} player, the weight differences of the sub-problems at that point must be in the range $[-(k-1)M, (k-1)M]$ where M is the maximum weight per person.

We can now consider adding students $1, \ldots, n$ and computing these variables. After solving all of these problems, we search for the smallest d such that $C(d, \lceil n/2 \rceil, \lfloor n/2 \rfloor)$ or $C(d, \lfloor n/2 \rfloor, \lceil n/2 \rceil)$ is true. We can store incrementally, or use a standard backtracking approach in order to output the actual teams.

```
WAR(w_1, ..., w_n)
1  Initialize all C(w,i,j) to FALSE. Set C(0,0,0) to TRUE.
2  for k = 1 to n
3          for w = -(k-1)M to (k-1)M
4                  for i,j ≥ 0 such that i + j = k
5                          C(w,i,j) = C(w - w_k, i-1, j) ∨ C(w + w_k, i, j-1)
                           ▷ Treat undefined sub-problems (e.g. C(w,-1,0)) as FALSE
6  for w = 0 to nM
7          for x ∈ {-w,w}, i ∈ {⌈n/2⌉, ⌊n/2⌋}, j = n - i
8                  if C(x,i,j) return BACKTRACK(x,i,j), x
```

```
BACKTRACK(w,i,j)
 1  Initialize A = {}, B = {}
 2  while k = i + j > 0
 3          if C(w - w_k, i-1, j)
 4                  A = A ∪ {k}
 5                  w = w - w_k
 6                  i = i - 1
 7             else
 8                  B = B ∪ {k}
 9                  w = w + w_k
10                  j = j - 1
11  return (A,B)
```

This simple version of the algorithm takes $O(n^3M)$ time. The first loop to calculate sub-cases takes $O(n \times (nM) \times n)$, and the latter loop to find the optimum $O(nM)$.

PROBLEM 3 *Mashup*

For simplicity, let x and y be *snippets of music* represented as strings over the alphabet of notes $\{A, B, C, \ldots, G\}$. For example, $x = ABC$ and $y = ADEF$. We say that z is a *loop* of x if z is a prefix of x^k for some integer $k > 0$. For example,

$z = ABCABCA$ is a loop of x since it is a prefix of $x^3 = ABCABCABC$.

A song s is a *mashup* of loops z and w if it is an interleaving of z and w. For example, one mashup of the loops $z = ABCABCA$ and $w = ADEFADE$ could be $ABCADEABFADECA$. Given song s and snippets (or hooks) of music x, y, devise an algorithm that determines if s is a mashup of loops of x and y.

Describe a dynamic programming solution to this problem as we have done in class by specifying a variable and then providing an equation that relates that variable to smaller instances of itself. Describe an algorithm based on your equation from above. The algorithm should simply output yes or no. Analyze the running time.

Solution: The input to the algorithm are the strings (z, x, y) and the output is a boolean value that is true if z is a mashup of x^k, y^k for some $k \in \mathbb{N}$. To solve MASHUP, we instead solve *a more general* problem of determining whether a string z consists of a partial interleaving of strings x', y'. (Notice that MASHUP is a special case of this problem in which x' and y' are strings of a special form.)

Observe that if z is an interleaving, then the last character z_n must belong to either x or y. However, we need to be careful and make sure this last character "comes from the correct spot" in x or y to ensure that z is still an interleaving. This suggests the following sub-problem definition $C_{i,j}$:

$$C_{i,j} : \text{ TRUE if string } z_1 \ldots z_{i+j} \text{ is an interleaving of } x_1 \ldots x_i \text{ and } y_1 \ldots y_j$$

In other words, $C_{i,j}$ is true if $z_1, \ldots, z_{i+j}$ can be created by interleaving the strings $x_1 \ldots x_i$ and $y_1 \ldots y_j$. We can compute this variable via dynamic programming as follows:

$$C_{i,j} = (C_{i-1,j} \wedge z_{i+j} = y_i) \vee (C_{i,j-1} \wedge z_{i+j} = x_j)$$

Notice, the first case corresponds to a situation in which $z_1 \ldots z_{i+j-1}$ is an interleaving of $y_1 \ldots y_{i-1}$ and $x_1 \ldots x_j$ and the character $z_{i+j} = y_i$. The second case is similar. An algorithm naturally follows: begin at $i, j = 0$, and then for $k = 1, \ldots, |z|$, work along the diagonal where $i + j = k$ and use the equation to determine $C_{i,j}$ from previous values. At the end, if there is any $C_{i,j} =$ TRUE for any i, j such that $i + j = |z|$, then output TRUE.

INTERLEAVES(z, x, y)

```
1  Let n = |z|
   ▷ Assume |x|, |y| ≥ n, or are padded with special ϵ symbols.
2  Initialize all C(i, j) to FALSE for i + j ≤ n and set C(0, 0) to TRUE.
3  for k = 1 to n
4          for i, j ≥ 0 such that i + j = k
5                  if i > 0
6                          C(i, j) = (C(i − 1, j) ∧ z_{i+j} = y_i)
7                  if j > 0
8                          C(i, j) = C(i, j) ∨ (C(i, j − 1) ∧ z_{i+j} = x_j)
9  for i, j ≥ 0 such that i + j = n
10         if C(i, j) return TRUE
11 return FALSE
```

The running time of the algorithm is $O(|z|^2)$ the first loop to check substrings runs in $O(n * n)$, and the second loop to check output runs in $O(n)$.

Application to MASHUP Now to solve MASHUP, simply construct $x' = x^k$ where $k = \lceil |z|/|x| \rceil$ and $y' = y^\ell$ where $\ell = \lceil |z|/|y| \rceil$ and then apply the interleaving algorithm described above.

MASHUP(z, x, y)

```
1  Let k = ⌈|z|/|x|⌉, ℓ = ⌈|z|/|y|⌉
2  return INTERLEAVES(z, x^k, y^ℓ)
```

Thus the MASHUP algorithm runs in the same time complexity as INTERLEAVES.

Example Here is how the algorithm would process the beginning of string $z = ABCADEABFADECA, x = ABCABCA, y = ADEFADE$ from the example above:

		x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	$\cdots$
	$C_{i,j}$	ϵ	A	B	C	A	B	C	A	B	$\cdots$
y_0	ϵ	T	T	T	T	T					
y_1	A	T			T						
y_2	D				T						
y_3	E				T	T	T				
y_4	F						T				
y_4	A						T				

Common mistakes: Here are some common mistakes that I noticed:

Greedy Approach: Make x and y into circularly linked lists, and maintain pointers p_x, p_y into both of them. Set $i = 0$. If z_i matches p_x, then increment i and p_x. Else if z_i matches p_y then increment i and p_y. If there is no match, then z cannot be a maskup, so stop and output 0. Upon reaching the end of z, stop and output 1.

The problem with this approach is when z_i matches *both* p_x and p_y. If the algorithm just greedily match p_x, then it fails on the following input:

$$x = ABC,\ y = BD,\ z = ABD$$

Notice in this case, the AB will match with x, and then the "D" matches neither p_x nor p_y. Some of you noticed this, and suggested to "fork" the state of the matching at this point. Unfortunately, doing so can lead to exponential running time if not done carefully. If $x = y$ and $z = xx$, then every character causes a "fork" of the current state. The table $C_{i,j}$ handles such "forking" implicitly in polynomial space.

Prefix/Suffix Approach: Some people were on the right track and noticed that if $z_1 \dots z_n$ is a mashup, then $z_1 \dots z_{n-1}$ must also be a mashup and z_n must belong to either x or y. This is true, but care must be taken to make sure that z_n extends the mashup of $z_1 \dots z_{n-1}$ in the proper way.