CS 577: Introduction to Algorithms Fall 2019

Homework 4 Solutions

Instructor: Dieter van Melkebeek TA: Andrew Morgan

Problem 1

Design an algorithm that takes as input two binary sequences a and b of lengths m and n respectively, and outputs the smallest length of a sequence c such that both a and b are subsequences of c. Your algorithm should run in time $O(m \cdot n)$.

We need to find the length of a shortest common supersequence of a and b. A trivial common supersequence is the concatenation of a and b. We may obtain a shorter common supersequence c by using a given position in c to cover both a position in a as well as a position in b. We develop a dynamic program to figure out an optimal way.

Consider the last position in c. We can use it to either cover the last position in a, the last position in b, both, or neither. Covering neither is obviously not optimal. Covering both is only possible when the last symbols in a and b agree. In each case, what remains is to find the length of a shortest common supersequence for the unmatched prefixes of a and b, and to take the minimum of the three possibilities. This is a smaller instance of the same problem.

Applying this reduction recursively leads to subproblems of the following form: for $0 \leq i \leq n \doteq |a|$ and $0 \leq j \leq m \doteq |b|$, we let $\mathrm{OPT}(i,j)$ denote the length of a shortest common supersequence of $a[1,\ldots,i]$ and $b[1,\ldots,j]$. Per the above discussion, $\mathrm{OPT}(i,j)$ can be computed recursively as follows, for $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$\mathrm{OPT}(i,j) = \min \left\{ \begin{array}{ll} 1 + \mathrm{OPT}(i-1,j) & \\ 1 + \mathrm{OPT}(i,j-1) & \\ 1 + \mathrm{OPT}(i-1,j-1) & \text{if } a_i = b_j \end{array} \right\}$$

and where $\mathrm{OPT}(i,0) = i$, $\mathrm{OPT}(0,j) = j$, and $\mathrm{OPT}(0,0) = 0$. (In fact, if $a_i = b_j$ one can argue that $\mathrm{OPT}(i,j) = \mathrm{OPT}(i-1,j-1)+1$. We leave this observation as an exercise as we do not need it for the solution of the problem.)

This recurrence can be computed by a recursive algorithm. As the number of possibilities for i and j are small (n and m, respectively), memoization will make this implementation efficient.

We can also compute the recurrence iteratively. Computation of $\mathrm{OPT}(i,j)$ depends only on subproblems where i and/or j is one smaller. There are many ways to iterate through these. One is to iterate over choices of $i = 0, 1, \ldots$, and for each i, iterate through choices of $j = 0, 1, \ldots$. Done this way, one need only remember the output of subproblems for the most recent value of i (and all j). Pseudocode implementing this is given in Algorithm 1. Another option is to iterate over choices of j first, and for each of those, to iterate over choices of i. Such an implementation is symmetric to Algorithm 1. The most space efficient solution is to pick the first option when $n \geq m$, and the second when $n < m$.

Correctness Correctness essentially follows from the above discussion. Formally, we argue that the recurrence for $\mathrm{OPT}(i,j)$ correctly computes the definition of $\mathrm{OPT}(i,j)$. We do this by induction on i and j, starting with the cases where $i = 0$ or $j = 0$ and working up.

Algorithm 1 Shortest Supersequence

Input: sequences $a[1,\ldots,n]$, $b[1,\ldots,m]$
Output: length of shortest supersequence of a and b

1: **procedure** ShortestSupersequence(a, b)
2: $\quad$ OPT$[0,\ldots,m] \leftarrow$ fresh array of integers
3: $\quad$ OPT$[0] \leftarrow 0$
4: $\quad$ **for** $j \leftarrow 1$ to m **do** OPT$[j] \leftarrow j$
5: $\quad$ **for** $i \leftarrow 1$ to n **do**
6: $\quad\quad$ NewOPT $\leftarrow$ fresh array of integers
7: $\quad\quad$ NewOPT$[0] \leftarrow i$
8: $\quad\quad$ **for** $j \leftarrow 1$ to m **do**
9: $\quad\quad\quad$ $\text{NewOPT}[j] \leftarrow \min \left\{ \begin{array}{ll} 1 + \text{OPT}[j] & \\ 1 + \text{NewOPT}[j-1] & \\ 1 + \text{OPT}[j-1] & \text{if } a_i = b_j \end{array} \right\}$
10: $\quad\quad$ OPT $\leftarrow$ NewOPT
11: $\quad$ **return** OPT$[m]$

Base case ($i = j = 0$): When $i = j = 0$, both $a[1,\ldots,0]$ and $b[1,\ldots,0]$ are both empty sequences, so the shortest supersequence is likewise empty (length 0).

Base case ($i = 0$,$j > 0$): When $i = 0$, $a[1,\ldots,0]$ is empty, so the shortest supersequence of $b[1,\ldots,j]$ is just $b[1,\ldots,j]$ itself (length j).

Base case ($i > 0$, $j = 0$): This case is symmetric with the previous.

Inductive step ($i > 0, j > 0$): If $c[1,\ldots,\ell]$ is a shortest supersequence of both $a[1,\ldots,i]$ and $b[1,\ldots,j]$, then $c[\ell]$ agrees with one of $a[i]$ or $b[j]$ (or both). Thus we need only consider the following cases:

- $c[1,\ldots,\ell-1]$ is a supersequence of both $a[1,\ldots,i-1]$ and $b[1,\ldots,j]$ and $c[\ell] = a[i]$
- $c[1,\ldots,\ell-1]$ is a supersequence of both $a[1,\ldots,i]$ and $b[1,\ldots,j-1]$ and $c[\ell] = b[j]$
- $c[1,\ldots,\ell-1]$ is a supersequence of both $a[1,\ldots,i-1]$ and $b[1,\ldots,j-1]$ and $c[\ell] = a[i] = b[j]$

By the inductive hypothesis, $1+\text{OPT}(i-1,j)$ computes the smallest length of supersequences of the first type, $1+\text{OPT}(i,j-1)$ computes the smallest length of supersequences of the second type, and $1 + \text{OPT}(i-1,j-1)$ computes the smallest length of supersequences of the last type. Taking the minimum of these (only including the last when $a[i] = b[j]$) computes the shortest length of a supersequence of $a[1,\ldots,i]$ and $b[1,\ldots,j]$.

Correctness of the recurrence for OPT now follows by induction. This also proves correctness of a recursive implementation of OPT. Correctness of the iterative version follows as well, as after line 4, $\text{OPT}[j] = \text{OPT}(0,j)$ for all j, and after the i-th iteration of the outer for loop, $\text{OPT}[j] = \text{OPT}(i,j)$ for all j.

Time and space analysis There are mn subproblems and each update takes $O(1)$ time, so a recursive implementation takes $O(mn)$ time. It likewise uses $O(mn)$ space.

As for the iterative solution, there are mn iterations of the loops in lines 5 and 8. Each iteration takes $O(1)$ time. So the overall time complexity is $O(mn)$. The maximum size of the array OPT is m; the rest of the space used is $O(1)$. As mentioned above, one can exchange a and b, leading to OPT having size n. Picking the better of the two leads to an overall space complexity of $O(\min(m, n))$.

Alternate solution The above solution looks very similar to the algorithm from class for sequence alignment. In fact, we can efficiently reduce the given problem to sequence alignment using the following penalty scheme: deleting a symbol has a cost of 1, as does matching positions that agree; the cost of matching positions that do not agree is $n + m + 1$. The latter ensures that an optimal solution will never match positions that disagree.

We claim that a shortest sequence that contains a and b as subsequences is also an *optimal alignment* between a and b under our penalty scheme. Given an alignment with cost bounded by $n + m$, one can produce a common supersequence of length equal to the cost of the alignment. Conversely, given a common supersequence c of length ℓ we obtain an alignment of a and b of cost ℓ by identifying a, b in c, and producing a '-' symbol for unmatched positions. For example, for $a = 011$, $b = 101$ and $c = 0101$ we have

c	0	1	0	1
a	**0**	**1**	-	**1**
b	-	**1**	**0**	**1**

Therefore, we can invoke the dynamic programming algorithm for optimal alignment for this problem, and return the result. The time complexity is again $O(mn)$ since the sequence alignment problem from class runs in that amount of time.

Problem 2

> You want to know, given a string of n symbols $\texttt{a}, \texttt{b}, \texttt{c}$, with $n \geq 1$, whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is $\texttt{a}$.
>
> Design an algorithm to solve this problem in time polynomial in n.

Our plan is to write a dynamic program that recursively determines the last multiplication to perform in order to obtain the value $\texttt{a}$, if one exists at all. If the second operand is $\texttt{a}$, then the first operand must be $\texttt{c}$. We do not have to consider $\texttt{b}$ as the second operand, because there is no way to right multiply by $\texttt{b}$ and obtain $\texttt{a}$. If the second operand is $\texttt{c}$, then there are two possibilities for the first operand: either $\texttt{a}$ or $\texttt{b}$. We need to consider all possibilities for where the last multiplication is performed and for its two operands. In each case, the resulting subproblem looks nearly identical to our given problem, except that we may be trying to get a different output letter than $\texttt{a}$. We can handle other letters following the same principle, just with different possibilities for the operands.

The subproblems that arise are parametrized by a contiguous portion of the input expression and a target letter among $\{\texttt{a}, \texttt{b}, \texttt{c}\}$. Let the input string be $S[1, \ldots, n]$. We define $\text{CanMult}(i, k, \ell)$ for $1 \leq i \leq k \leq n$ to indicate whether $S[i, \ldots, k]$ can be parenthesized to compute ℓ. We wish to know $\text{CanMult}(1, n, \texttt{a})$. Following the above discussion, we can compute it with the following recurrence:

$$\text{CanMult}(i, k, \ell) = \begin{cases} \texttt{True} & \text{if } i = k \text{ and } S[i] = \ell \\ \texttt{False} & \text{if } i = k \text{ and } S[i] \neq \ell \\ \displaystyle\bigvee_{\substack{i \leq j < k \\ \ell_1, \ell_2 : \ell_1 \ell_2 = \ell}} \text{CanMult}(i, j, \ell_1) \wedge \text{CanMult}(j+1, k, \ell_2) & \text{if } i < k \end{cases}$$

Here, ℓ_1 and ℓ_2 range over all choices of letters such that $\ell_1 \ell_2$ multiply to ℓ. The $\vee$ represents a Boolean OR (`||` in Java), and $\wedge$ represents a Boolean AND (`&&` in Java).

We can compute this recurrence with a recursive algorithm. As the number of possibilities for i, k, ℓ is small, memoization will make this implementation efficient.

We can also compute the recurrence iteratively. Computation of $\text{CanMult}(i, k, \ell)$ depends on its values where $k - i$ is strictly smaller. So as long as we compute the subproblems in order of increasing value of $k - i$, the subproblems required for $\text{CanMult}(i, k, \ell)$ will have been computed by the time we need them. Pseudocode is given in Algorithm 2. For each i and k, the recurrence for $\text{CanMult}(i, k, \ell)$ is evaluated for all ℓ simultaneously, as this makes the code more concise.

Correctness Correctness of the recurrence for CanMult follows from the above discussion. We leave a detailed proof by induction on $k - i$ as an exercise. Correctness of a recursive computation of CanMult follows immediately, as does correctness of the above iterative implementation.

Time and space analysis There are $O(n^2)$ possibilities for $1 \leq i \leq k \leq n$ and ℓ, and we compute $\text{CanMult}(i, k, \ell)$ for each one. For a fixed subproblem, we need to consider up to $O(n)$ values for j, ℓ_1, ℓ_2; it takes constant work for each. So the local work is $O(n)$. Adding all the local work together, the total running time of recursive computation with memoization is $O(n^3)$. Its space usage is dominated by the memoization table, so is $O(n^2)$.

As for Algorithm 2, direct inspection reveals that it runs in $O(n^3)$ time and $O(n^2)$ space.

Algorithm 2 Multiplication Problem

Input: $S[1, \ldots, n]$ a string of letters from $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

Output: Yes/No, whether S can be parenthesized such that the result of multiplication is $\mathtt{a}$.

procedure MULTIPLYTOA($S[1, \ldots, n]$)
    CanMult$[1, \ldots, n][1, \ldots, n][\mathtt{a}, \mathtt{b}, \mathtt{c}] \leftarrow$ fresh $n \times n \times 3$ array
    for $i = 1$ to n **do**
        $$\text{CanMult}[i][i][\ell] \leftarrow \begin{cases} \mathtt{True} & \text{if } S[i] = \ell \\ \mathtt{False} & \text{otherwise} \end{cases}$$
    for $s = 1$ to $n - 1$ **do** (s is $k - i$)
        for $i = 1$ to $n - s$ **do**
            $k \leftarrow i + s$
            CanMult$[i][k][\ell] \leftarrow \mathtt{False}$ for each $\ell \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
            for $j = i$ to $k - 1$ **do**
                for $\ell_1, \ell_2 \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ **do**
                    $\ell \leftarrow \ell_1 \ell_2$ (using multiplication table)
                    if CanMult$[i][j][\ell_1] \wedge$ CanMult$[j + 1][k][\ell_2]$ **then**
                        CanMult$[i][k][\ell] \leftarrow \mathtt{True}$
    return CanMult$[1][n][\mathtt{a}]$

Problem 3

You are given a number k of bottles, a number n of substances, and nonnegative numbers e_{ij} for every pair of substances. We need to determine integers $0 \leq t_1 \leq t_2 \leq \ldots \leq t_{k-1} \leq n$ (indicating the last substances put in bottles 1 through $k-1$) such that

$$\sum_{i=1}^{k} \sum_{t_{i-1} < \ell < m \leq t_i} e_{\ell m}$$

is minimized, where $t_0 = 0$ and $t_k = n$.

(a) Design an $O(n^2)$ algorithm that outputs the (minimum) energies for the subinstances consisting of substances i through j for *all* $1 \leq i \leq j \leq n$ and $k = 1$.

(b) Design an $O(kn^2)$ algorithm to solve the problem for a given instance with n substances and a given k.

Part (a)

We show how to efficiently calculate all the possible sums $\sum_{i \leq \ell < m \leq j} e_{\ell m}$ for any i, j. We think of the energies e_{ij} as being organized in an (upper-triangular) 2-dimensional array. In the example array below, for six substances, the energy produced between 3 and 5 is the sum $e_{34} + e_{35} + e_{45}$, so we need to add the entries of the highlighted triangle in Table 1 shown below.

	1	2	3	4	5	6
1		e_{12}	e_{13}	e_{14}	e_{15}	e_{16}
2			e_{23}	e_{24}	e_{25}	e_{26}
3				e_{34}	e_{35}	e_{36}
4					e_{45}	e_{46}
5						e_{56}
6						

Table 1: Example sum of energies

In general, for every two substances i, j we need the sum of the values inside such a triangle. In order to do so, we first compute an auxiliary array A of the same shape such that each cell $A[i][j]$ stores the sum of all the energies in row i up to and including e_{ij}; i.e., $A[i][j] = \sum_{i < m \leq j} e_{im}$. We compute the elements of A using the recurrence $A[i][j] = A[i][j-1] + e_{ij}$, and use $A[i][i] = 0$ as the base case. Care should be taken to fill in the array in order of increasing j. This is illustrated in Table 2.

	1	2	3	4	5	6
1	0	$(\leftarrow) + e_{12}$	$(\leftarrow) + e_{13}$	$(\leftarrow) + e_{14}$	$(\leftarrow) + e_{15}$	$(\leftarrow) + e_{16}$
2		0	$(\leftarrow) + e_{23}$	$(\leftarrow) + e_{24}$	$(\leftarrow) + e_{25}$	$(\leftarrow) + e_{26}$
3			0	$(\leftarrow) + e_{34}$	$(\leftarrow) + e_{35}$	$(\leftarrow) + e_{36}$
4				0	$(\leftarrow) + e_{45}$	$(\leftarrow) + e_{46}$
5					0	$(\leftarrow) + e_{56}$
6						0

Table 2: The array A storing prefix sums for each row. The $\leftarrow$ means to use the value computed in the cell to the immediate left.

Finally, we compute and store the sums we need in a third array TotalEnergy of the same shape. That is, the value of TotalEnergy$[i][j]$ is $\sum_{i \leq \ell < m \leq j} e_{\ell m}$. We compute it using the recurrence

$$\text{TotalEnergy}[i][j] = \begin{cases} 0 & \text{if } i = j \\ \text{TotalEnergy}[i][j] + A[i-1][j] & \text{if } i < j \end{cases}.$$

Care should be taken to fill in the array in order of increasing i. Table 3 illustrates the computation:

	1	2	3	4	5	6
1	0	$(\downarrow) + A[1][2]$	$(\downarrow) + A[1][3]$	$(\downarrow) + A[1][4]$	$(\downarrow) + A[1][5]$	$(\downarrow) + A[1][6]$
2		0	$(\downarrow) + A[2][3]$	$(\downarrow) + A[2][4]$	$(\downarrow) + A[2][5]$	$(\downarrow) + A[2][6]$
3			0	$(\downarrow) + A[3][4]$	$(\downarrow) + A[3][5]$	$(\downarrow) + A[3][6]$
4				0	$(\downarrow) + A[4][5]$	$(\downarrow) + A[4][6]$
5					0	$(\downarrow) + A[5][6]$
6						0

Table 3: Calculation of sums of energies (array TotalEnergy). The $\downarrow$ means to use the value computed in the cell directly below.

Correctness Correctness follows by arguing that the recurrence for A and for TotalEnergy correctly compute their definitions. We leave a formal proof by induction as an exercise.

Time and space analysis Both A and TotalEnergy have $O(n^2)$ entries to fill in. Each entry takes constant time. So the algorithm uses an overall $O(n^2)$ time and $O(n^2)$ space.

Part (b)

We are given n chemical substances and k bottles and we need to divide the chemicals into the k bottles. Since we are given fewer bottles than substances ($k < n$) we essentially need to decide which substances will be put in the same bottle. Remember that we cannot change the order of the substances given. What we need to decide is in which $k-1$ indices we will place a "separator" to minimize the amount of energy produced, where a "separator" is the point at which we stop using the i'th bottle and start putting chemicals in the next one. Essentially determine integers $0 \leq t_1 \leq t_2 \leq \ldots \leq t_{k-1} \leq n$ indicating the last substances put in bottles 1 through $k-1$.

Subproblems The problem is trivial for 1 bottle, as there is no choice to be made and the minimum energy is the sum of all pairwise energies.

Let's say we have 2 bottles. In that case we need to find an index t such that all substances from 1 to t are in the first bottle, and all substances from $t+1$ until n are in the second one. And we want the sum $\sum_{1\le \ell<m\le t} e_{\ell m} + \sum_{t+1\le \ell<m\le n} e_{\ell m}$ to be the minimum possible. We try all possibilities for $t \in \{0,\ldots,n\}$ and select the one that minimizes the above sum.

If we have $k > 2$ bottles, we consider the last decision that we need to make, i.e., the choice of $t_{k-1} \in \{0,\ldots,n\}$. We do not know which choice is optimal, so we try all possibilities. Given a choice for t_{k-1}, what remains is to find an optimal break-up of the substances 1 through t_{k-1} into $k-1$ bottles, i.e., to solve the problem for the instance defined by the substances 1 through t_{k-1} with the given energies $e_{\ell m}$ for $1 \le \ell < m \le t_{k-1}$, and $k-1$ bottles. We recursively solve those instances, and then select the value of t_{k-1} that minimizes the sum of the minimum energy for the subinstance and the energy in the last bottle.

The recursive calls that arise during this recursion all have the following form: Solve the problem for the subinstance consisting of substances 1 through i using j bottles, where $i \in \{0,\ldots,n\}$ and $j \in \{1,\ldots,k\}$. We denote the minimum energy achievable for that subinstance by $\mathrm{OPT}(i,j)$.

Recurrence and correctness Generalizing the above approach, we obtain that the following recurrence correct computes $\mathrm{OPT}(i,j)$ for $i \in \{0,\ldots,n\}$ and $j \in \{2,\ldots,k\}$:

$$\mathrm{OPT}(i,j) = \min_{0\le t\le i} \left\{ \mathrm{OPT}(t,j-1) + \sum_{t+1\le \ell<m\le i} e_{\ell m} \right\},$$

where the sum is the total amount of energy produced in the j-th bottle, which contains substances from $t+1$ to j. This expression says that for bottle j we find the substance t to stop the previous bottle with such that the total energy is minimized, where the total energy is the minimum possible energy when storing substances 1 though t in $j-1$ bottles plus the energy when storing substances $t+1$ to i in bottle j. Correctness formally follows from a proof by induction; we leave the details as an exercise.

The base cases correspond to $j = 1$, i.e., all the first i substances go into a single bottle. In that case we have $\mathrm{OPT}(i,1) = \sum_{1\le \ell<m\le i} e_{\ell m}$. We apply the recurrence column by column, i.e., first for $j = 2$ and all i, then for $j = 3$ and all i, etc. Our final answer is $\mathrm{OPT}(n,k)$ as we want the minimum energy when we have n substances and k bottles in total.

Algorithm and analysis The algorithm starts by precomputing TotalEnergy from part (a). Next we build a table for OPT. It table has $O(n \cdot k)$ entries (i.e. substances $\times$ bottles). The amount of work involved in applying the recurrence for a given cell is $O(n)$, given that we have the sum already computed in TotalEnergy. Thus, the total amount of work is $O(kn^2)$ to fill in OPT. This plus the $O(n^2)$ time to compute TotalEnergy gives an overall running time of $O(kn^2)$.

As we only need to access the OPT values of the previous column when computing the next one, we only need keep $O(n)$ cells of OPT at a time to run this part of the process, although $\Omega(n^2)$ space is still required to store TotalEnergy.

Problem 4

Suppose we have a set of precincts $P_1, P_2, \ldots, P_n$, each containing m registered voters. We're supposed to group these precincts into two districts, each consisting of $n/2$ of the precincts. For each precinct, we have information on how many voters are registered to each of two political parties. We say that the set of precincts is susceptible to gerrymandering if it is possible to perform the division in such a way that the same party holds a majority in both districts.

Design an algorithm to determine whether a given set of precincts is susceptible to gerrymandering. The running time of your algorithm should be polynomial in n and m.

Before we start implementing an algorithm to find whether a given set of precincts is susceptible to gerrymandering, let us make the following observation. If our input is susceptible to gerrymandering, we should be able to divide the set of precincts into two districts each composed of $n/2$ precincts such that some party has more than $mn/2$ votes in both districts (note that $mn/2$ is always an integer). That party has to have the overall majority of the mn votes.

We can calculate which party has the largest number of voters by looping over all precincts and counting the number of votes. W.l.o.g. we assume that A has the largest number of voters, say $mn/2 + d$. The question now becomes whether we can equipartition the precincts in such a way that the 'extra' d votes are distributed over the two districts. If we can find such a division then we know that district one has $\lfloor mn/4 \rfloor + d_1$ votes and district 2 has $\lfloor mn/4 \rfloor + d_2$ votes where $d = d_1 + d_2$ for some $d_1, d_2 > 0$.

Let us denote by a_i the number of voters for party A in precinct i. Since we know all the a_i's, we can look at the problem as follows: Does there exist a subset of $n/2$ of the numbers a_i such that their sum is at least $\lfloor mn/4 \rfloor + 1$ and at most $\lfloor mn/4 \rfloor + d - 1$.

Subproblems This sounds a lot like the knapsack problem we discussed in class but there is an extra requirement which demands that the knapsack must contain exactly $n/2$ items. To handle this issue, we extend the subproblems we considered in our knapsack algorithm with one extra variable that keeps track of the number of items in the knapsack. We define $OPT(i, j, k)$ as the maximal sum less than or equal to k of the form $\sum_{\ell \in I} a_\ell$ where $I \subseteq \{1, 2, \ldots, i\}$ and $|I| = j$. The value we are interested in is $OPT(n, n/2, \lfloor mn/4 \rfloor + d - 1)$. The input is susceptible to gerrymandering iff this value is at least $\lfloor mn/4 \rfloor + 1$.

Recurrence and correctness What does the recurrence for $\mathrm{OPT}(i, j, k)$ look like? Consider an optimal solution $\mathcal{O}$ for $\mathrm{OPT}(i, j, k)$. For $i > 0$ and $j > 0$, we have the following:

- If $a_i \notin \mathcal{O}$ then $\mathrm{OPT}(i, j, k) = \mathrm{OPT}(i - 1, j, k)$.
- If $a_i \in \mathcal{O}$ then $\mathrm{OPT}(i, j, k) = a_i + \mathrm{OPT}(i - 1, j - 1, k - a_i)$.

Note that the latter case can only happen if $k \geq a_i$. This analysis shows that the following recurrence will correctly compute $\mathrm{OPT}(i, j, k)$:

- If $a_i \leq k$ then $\mathrm{OPT}(i, j, k) = \max(\mathrm{OPT}(i - 1, j, k), a_i + \mathrm{OPT}(i - 1, j - 1, k - a_i))$.
- Otherwise, $\mathrm{OPT}(i, j, k) = \mathrm{OPT}(i - 1, j, k)$.

The base cases are those where $i = 0$ or $j = 0$ (and $k \geq 0$), for which the value of $\mathrm{OPT}(i, j, k)$ is zero.

Running time analysis The resulting algorithm computes the values $\mathrm{OPT}(i, j, k)$ for $0 \leq i \leq n$, $0 \leq j \leq n/2$ and $0 \leq k \leq \lfloor mn/4 \rfloor + d - 1$ in the order of increasing values of $i + j$. Since one evaluation of the recurrence only involves a constant amount of work and our 3D array contains $O(n \cdot n \cdot mn)$ cells, our algorithm runs in $O(mn^3)$ time.

Problem 5

Design an algorithm that on input $A, B, a_1, \ldots, a_n, b_1, \ldots, b_n$, computes the maximum number of sculptures that can be made from the starting sheet of paper, as well as the minimum number of frayed edges that must be visible when Hiro creates this maximum number of sculptures. Your algorithm should run in time polynomial in A, B, and n.

In this problem, Hiro has to choose a sequence of horizontal or vertical cuts to make in the starting rectangular paper in order to maximize the number of sculptures (and to minimize the number of frayed edges present in those sculptures) he is left with at the end. For ease of exposition, let us ignore the frayed edges requirement for now.

Subproblems and recurrence Since any strategy is a sequence of cuts, let us consider what happens to the paper after a single cut—regardless of whether the paper is cut horizontally or vertically, Hiro is left with two rectangular sheets of paper afterwards. Each of these new rectangles can be independently sliced into its own set of sculptures, so maximizing the total number of sculptures for the original paper means maximizing the total number of sculptures for each of these new ones.

This observation suggests the following specification for our subproblems.

$$\text{OPT}(i, j) = \text{ max number of sculptures that can be created from starting paper of size } i \times j,$$

where $1 \leq i \leq A$ and $1 \leq j \leq B$. Given any paper of size $i \times j$, we can then determine the maximum number of sculptures it can produce by using the solutions to subproblems corresponding to smaller paper sizes to evaluate every reasonable possibility for the first cut (including no cut). Since the starting paper and all sculptures have integral dimensions, it suffices to try only cuts that produce rectangles with integral dimensions.

The recurrence for OPT is then as follows, where $\text{sculpt}(i, j)$ is a function that returns 1 if $i = a_x, j = b_x$ or $i = b_x, j = a_x$ for some $x \in [n]$, and 0 otherwise:

$$\begin{aligned}\text{OPT}(i, j) &= \max\{\text{max for no cut, max for horizontal cut, max for vertical cut}\} \\ &= \max\{\text{sculpt}(i, j), \max_{k \in [i-1]} [\text{OPT}(k, j) + \text{OPT}(i - k, j)], \max_{\ell \in [j-1]} [\text{OPT}(i, \ell) + \text{OPT}(i, j - \ell)]\},\end{aligned}$$

where the maximum over an empty set is defined here as 0. An example horizontal cut in a $i \times j$ sheet of paper is depicted in Figure 1.

We are now ready to consider frayed edges. The key insight here is that our first-cut-based schema for OPT still works when we extend it to include frayed edges. After the first cut, we can independently solve each of the resulting papers, where solving a paper entails computing the maximum number of sculptures creatable from that paper and the minimum number of frayed edges present in those sculptures. Then, the corresponding solution for the original paper (given the first cut) is the sum of the maximum sculpture counts and minimum frayed edge counts for the two resulting papers.

The only issue is that for all but the starting paper, any combination of the edges may be frayed, and each possible combination could yield different minimum frayed edge counts. This suggests modifying OPT to include an additional four variables f_t, f_r, f_b, f_l that keep track of precisely which edges are frayed, where f_t is 1 if the top edge is frayed and 0 otherwise, and f_r, f_b, f_l are similarly

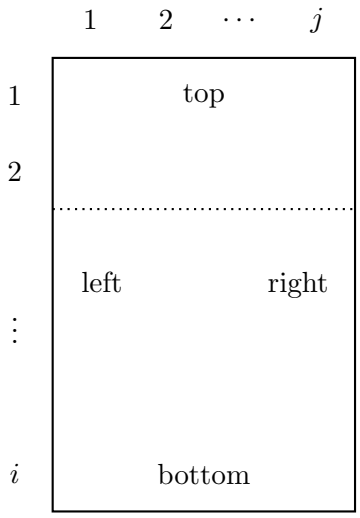

Figure 1: Horizontal cut in a $i \times j$ sheet of paper, with edge labels.

defined for the right, bottom, and left edges, respectively (and the edge labels are as depicted in Figure 1). The specification for our new table OPT' is then

$$\begin{aligned}\mathrm{OPT}'(i, j, [f_t, f_r, f_b, f_l]) = &(\text{max sculptures}, \text{min frayed edges given max sculptures}) \\ &\text{for } i \times j \text{ starting paper frayed according to } [f_t, f_r, f_b, f_l].\end{aligned}$$

In order to write the recurrence for OPT' neatly, it will be necessary to define an ordering on (sculpture count, frayed edge count) pairs. Given two such pairs (s_1, e_1) and (s_2, e_2), define $\leq$ so that

$$(s_1, e_1) \leq (s_2, e_2) \iff s_1 < s_2 \vee [s_1 = s_2 \wedge e_1 \geq e_2].$$

In other words, $(s_1, e_1) \leq (s_2, e_2)$ if the first pair has fewer sculptures, or if they have the same number of sculptures but the first pair has at least as many frayed edges. When we take a maximum over such pairs, we implicitly use this ordering. Additionally, define $(s_1, e_1)+(s_2, e_2)$ as (s_1+s_2, e_1+e_2). Then, the recurrence for OPT' is

$$\begin{aligned}\mathrm{OPT}'(i, j, [f_t, f_r, f_b, f_l]) = \max\{&(\mathrm{sculpt}'(i, j, [f_t, f_r, f_b, f_l]), \\ &\max_{k \in [i-1]} [\mathrm{OPT}'(k, j, [f_t, f_r, 1, f_l]) + \mathrm{OPT}'(i-k, j, [1, f_r, f_b, f_l])], \\ &\max_{\ell \in [j-1]} [\mathrm{OPT}'(i, \ell, [f_t, 1, f_b, f_l]) + \mathrm{OPT}'(i, j-\ell, [f_t, f_r, f_b, 1])]\},\end{aligned}$$

where $\mathrm{sculpt}'(i, j, [f_t, f_r, f_b, f_l]) = (1, f_t + f_r + f_b + f_l)$ if $i = a_x, j = b_x$ or $i = b_x, j = a_x$ for some $x \in [n]$, or $(0, 0)$ otherwise. By defining sculpt' in this way, we only count frayed edges when we use the current piece as a sculpture.

Since each entry in the table depends only on entries with a smaller sum of side lengths, we can fill out OPT' in increasing order of $i + j$ without any difficulties. At the end of the process, we should return $\mathrm{OPT}'(A, B, [0, 0, 0, 0])$.

Correctness and running time analysis Correctness follows from the discussion above. For run-time, note that sculpt$'$ takes $O(n)$ time to compute, while the remaining two maximums are over at most A elements and at most B elements, respectively. Hence, each entry of the table takes at most $O(n + A + B)$ time to compute, so with $16AB$ entries, this comes out to a total time of $O(AB(n + A + B))$, which is polynomial in n, A, and B.

As an aside, note that this time complexity can be improved to $O(n + AB(A + B))$. Currently, we are computing sculpt$'$ for each entry of OPT$'$, but we could instead perform a preprocessing step where for each of the n sculptures (a_i, b_i) we mark OPT$'(a_i, b_i, F)$ and OPT$'(b_i, a_i, F)$ for all 16 possible values of F with the correct return value of sculpt$'$. This takes $O(n)$ time, and we automatically know that sculpt$'$ will return $(0, 0)$ for any other entry, so we avoid having to compute sculpt$'$ every time and the total complexity becomes $O(n) + O(AB(A + B)) = O(n + AB(A + B))$.

Also note that though it does not change the asymptotic time complexity, we do not have to distinguish between every possible combination of frayed edges–the subproblems we have to consider are the same if, for example, either only the top edge is frayed or only the bottom edge is frayed. In fact, it suffices to keep track of only the number of horizontal frayed edges $f_t + f_b$ and the number of vertical frayed edges $f_l + f_r$ rather than each component individually, leaving us with a total of $9AB$ entries in OPT$'$ instead of $16AB$. We leave working out the recurrence in this case as an exercise for the reader.