CS 577: Introduction to Algorithms Fall 2022

Homework 4

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

This homework covers dynamic programming. **Problem 3 must be submitted for grading by 2:29pm on 10/11.** Please refer to the homework guidelines on Canvas for detailed instructions.

Warm-up problems

1. Consider the sequence alignment problem from class, but instead of maximizing the number of matches we minimize the number of non-matches, i.e., the total number of mismatches and gaps.

   For example, consider the sequences $A = aabcde$ and $B = bfghaa$. An alignment that maximizes the number of matches is:

   $$\begin{array}{rcccccccccc} A: & - & - & - & - & a & a & b & c & d & e \\ & & & & & | & | & & & & \\ B: & b & f & g & h & a & a & - & - & - & - \end{array}$$

   It has 2 matches and 8 non-matches (all 8 gaps).

   An alignment that minimizes the number of non-matches is:

   $$\begin{array}{rcccccccc} A: & a & a & b & c & d & e & - & - \\ & & & | & \not| & \not| & \not| & & \\ B: & - & - & b & f & g & h & a & a \end{array}$$

   It has 1 match and 7 non-matches (3 mismatches and 4 gaps).

   Design an algorithm that takes as input two sequences $A[1 \ldots n]$ and $B[1 \ldots m]$, and outputs the minimum number of non-matches in an alignment of A and B. Your algorithm should run in time $O(m \cdot n)$.

2. Consider the multiplication defined in Table 1. For example, $ab = b$ and $ba = c$. Note that this multiplication is neither associative nor commutative.

   Table 1: Multiplication Table

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

   You want to know, given a string of n symbols a, b, 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 a. For example, the string $bbbbac$ can be parenthesized as $((b(bb))(ba))c$, and that evaluates to a.

   Design an algorithm to solve this problem in time polynomial in n.

Regular problems

3. [Graded] A lab in the Chemistry department was running some experiments with some highly reactive liquid chemicals. Now that the experiments have ended (successfully), the lab assistant needs to place the n chemical substances into k bottles, which will then be shipped to a facility for safe destruction. The chemicals are numbered 1 through n, and need to be place in the k bottles in this order. We know that $k < n$ so we have to mix some of the chemicals together. The problem is that when some substances are mixed, chemical reactions between them produce energy. Specifically, when substances i and j are put in the same bottle, they produce e_{ij} units of energy. In order to reduce the risk of explosion during transportation, we want to minimize the total amount of energy produced.

   More formally, 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 $t_k = n$. The inner sum in this expression represents the energy produced in bottle i.

   For example, consider the instance with $k = 2$, $n = 3$, and energies $e_{12} = 10$, $e_{13} = 5$, and $e_{32} = 42$. Observe that there are 4 possible ways to split the 3 substances into the 2 bottles:

   - Split 1: $\{1, 2, 3\}\{\}$
   - Split 2: $\{1\}\{2, 3\}$
   - Split 3: $\{1, 2\}\{3\}$
   - Split 3: $\{\}\{1, 2, 3\}$

   The first and last ones are equivalent and yield a total energy of $42 + 5 + 10 = 57$. The second has energy 42, and the third has energy 10, so the answer is 10. Note that if we could change the order of the substances, the energy could be reduced even further (bottle 1 contains $\{1, 3\}$ and bottle 2 contains $\{2\}$, for a total energy of 5), but this is not allowed.

   (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. Your algorithm should output the minimum total energy.

4. In modern origami (the Japanese art of paper folding), one typically starts with a square sheet of paper and attempts to transform this square into a three-dimensional animal, geometric object, or any other sculpture one can think of, using nothing but a sequence of folds. In traditional 17th–18th century origami, however, the starting shape of the paper was less strictly prescribed.

   Hiro has stumbled across a book containing instructions for n origami sculptures from this early period, each of which starts from rectangular paper of size $a_i \times b_i$ where a_i and b_i are positive integers. He would like to make a diorama containing as many of these (not necessarily distinct) sculptures as possible, but he only has access to a single sheet of paper

of size $A \times B$ (where A and B are also positive integers) and no scissors. By folding the paper carefully and tearing along the crease, Hiro is confident that he can make perfect horizontal and vertical cuts across an entire sheet of paper, splitting the sheet into two.

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. Your algorithm should run in time polynomial in A, B, and n.

5. Gerrymandering is the practice of carving up electoral districts in very careful ways so as to lead to outcomes that favor a particular political party. Recent court challenges to the practice have argued that through this calculated redistricting, large numbers of voters are being effectively (and intentionally) disenfranchised.

   Computers, as it turns out, have been implicated as some of the main "villains" in much of the news coverage on this topic: it is only thanks to powerful software that gerrymandering grew from an activity carried out by a bunch of people with maps, pencil, and paper into the industrial-strength process it is today. Why is gerrymandering a computational problem? Partly it is the database issues involved in tracking voter demographics down to the level of individual streets and houses; and partly it is the algorithmic issues involved in grouping voters into districts. Let's think a bit about what these latter issues look like.

   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. Now, for each precinct, we have information on how many voters are registered to each of two political parties. (Suppose for simplicity that every voter is registered to one of these two.) 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.

   Example Suppose we have $n = 4$ precincts, and the following information on registered voters. Party A has 55, 43, 60, and 47 voters in districts P_1, P_2, P_3, P_4 respectively, and party B has 45, 57, 40, and 53. This set of precincts is susceptible, since if we grouped precincts P_1 and P_4 into one district, and precincts P_2 and P_3 into the other, then party A would have a majority in both districts. (Presumably, the "we" who are doing the grouping here are members of party A.) This example is a quick illustration of the basic unfairness in gerrymandering: although party A holds only a slim majority in the overall population (205 to 195), it ends up with a majority in not one but both districts.

Challenge problem

6. Design an algorithm for problem 3(b) that runs in time $O(kn \log n)$ when given access to the one-bottle energies computed in part (a).

Programming problem

7. SPOJ problem Square Brackets (problem code SQRBR).

CS 577: Introduction to Algorithms Fall 2022

Homework 4 Solutions to Warm-up Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 1

> Consider the sequence alignment problem from class, but instead of maximizing the number of matches we minimize the number of non-matches, i.e., the total number of mismatches and gaps. Design an algorithm that takes as input two sequences $A[1 \ldots n]$ and $B[1 \ldots, m]$, and outputs the minimum number of non-matches in an alignment of A and B. Your algorithm should run in time $O(m \cdot n)$.

The process of computing an alignment of A and B can be interpreted as a sequence of decisions indicating which pairs of symbols from A and B are (mis-)matched or whether we skip a symbol by introducing a gap. Consider the last such decision: we can choose to match $A[n]$ and $B[m]$, incurring a penalty if $A[n] \neq B[m]$, or we can choose to skip either $A[n]$ or $B[m]$, which always incurs a penalty. In each of these cases, what remains is to find an alignment between prefixes of the original sequences A and B. In doing so, we should choose the option that leads to the fewest penalties, or equivalently to the fewest non-matches.

Applying this reduction recursively leads to subproblems of the following form: for $0 \leq i \leq n$ and $0 \leq j \leq m$, we define $\mathrm{OPT}[i, j]$ to be the minimum number of non-matches in an alignment of $A[1, \ldots, i]$ and $B[1, \ldots, j]$. Per the discussion above, $\mathrm{OPT}[i, j]$ can be computed as follows for $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$\mathrm{OPT}[i, j] = \min \begin{cases} 1 - \delta_{A[i], B[j]} + \mathrm{OPT}[i-1, j-1] \\ 1 + \mathrm{OPT}[i-1, j] \\ 1 + \mathrm{OPT}[i, j-1], \end{cases}$$

where $\delta_{a,b} \doteq 1$ if $a = b$ and 0 otherwise. Moreover, we have $\mathrm{OPT}[i, 0] = i$, $\mathrm{OPT}[0, j] = j$ and $\mathrm{OPT}[0, 0] = 0$.

The final output for the algorithm is $\mathrm{OPT}[n, m]$, which can be computed by a recursive algorithm that implements the recurrence. 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 $j = 0, 1, \ldots$. Visually, this amounts to filling the OPT table row by row, from left to right and top to bottom. Done this way, one need only remember the output of subproblems for the most recent value of i (and all j), i.e., remember the last row in the OPT table to compute the current one. Another option is to iterate over choices of j first, and for each of those, to iterate over choices of i. This amounts to filling the OPT table column by column, from top to bottom and left to right. It is also possible to compute the values by diagonal-by-diagonal, more precisely for increasing value of $i + j$, and compute the entries in a given diagonal in any order; for this process it suffices to keep track of the previous two diagonals.

Correctness Correctness essentially follows from the above discussion. Formally, we argue that the recurrence for OPT$[i, j]$ correctly computes the definition of OPT$[i, j]$. We do this by induction on $i + j$, starting with the cases where $i = 0$ or $j = 0$ and working up from there.

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 minimum number of non-matches in an alignment of $A[1, \ldots, 0]$ and $B[1, \ldots, 0]$ is 0.

Base case ($i = 0, j > 0$): When $i = 0$, $A[1, \ldots, 0]$ is empty, so the only choice there is is to introduce j gaps in $B[j]$, incurring a cost of j.

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

Inductive step ($i > 0, j > 0$): An alignment of $A[1, \ldots, i]$ and $B[1, \ldots, j]$ that minimizes the number of non-matches either matches $A[i]$ with $B[j]$, skips $A[i]$ or skips $B[j]$. Thus, we need only consider the following cases:

- If the alignment matches $A[i]$ with $B[j]$, its number of non-matches is $1 - \delta_{A[i],B[j]}$ plus the number of non-matches in the alignment that minimizes the number of non-matches between $A[1, \ldots, i-1]$ and $B[1, \ldots, j-1]$.
- If the alignment skips $A[i]$, its number of non-matches is 1 plus the number of non-matches in the alignment that minimizes the number of non-matches between $A[1, \ldots, i-1]$ and $B[1, \ldots, j]$.
- If the alignment skips $B[j]$, its number of non-matches is 1 plus the number of non-matches in the alignment that minimizes the number of non-matches between $A[1, \ldots, i]$ and $B[1, \ldots, j-1]$.

By the inductive hypothesis, $1-\delta_{A[i],B[j]}+$OPT$[i-1, j-1]$, $1+$OPT$[i-1, j]$, and $1+$OPT$[i, j-1]$ compute the minimum number of non-matches in an alignment of the first, second, and third type, respectively. Taking the minimum of these computes the minimum number of non-matches in an alignment 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 an iterative version follows as well since it computes OPT using the recurrence and only computes a value OPT$[i, j]$ once it has already computed the values of OPT$[i-, j-1]$, OPT$[i-1, j]$, and OPT$[i, j-1]$.

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

An iterative implementation as discussed above also takes time $O(mn)$, since it has two nested for loops computing the recurrence, one iterating over $0 \le i \le n$ and another over $0 \le j \le m$. Depending on the choice of iterative implementation, we are able to obtain a space complexity of $O(n)$ (if we just remember the last row) or $O(m)$ (if we remember the last column). By choosing the better of the two options we obtain an overall space complexity of $O(\min(m, n))$, which is also the space complexity of the diagonal-by-diagonal computation order.

Problem 2

You want to know, given a string of n symbols a, b, c, with $n \ge 1$, whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is 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 a, if one exists at all. If the second operand is a, then the first operand must be c. We do not have to consider b as the second operand, because there is no way to right multiply by b and obtain a. If the second operand is c, then there are two possibilities for the first operand: either a or 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 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 {a, b, c}. Let the input string be $S[1, \ldots, n]$. We define CanMult(i, k, ℓ) for $1 \le i \le k \le n$ to indicate whether $S[i, \ldots, k]$ can be parenthesized to compute ℓ. We wish to know CanMult$(1, n,$ a$)$. Following the above discussion, we can compute it with the following recurrence:

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

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 CanMult(i, k, ℓ) 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 CanMult(i, k, ℓ) will have been computed by the time we need them. Pseudocode is given in Algorithm 1. For each i and k, the recurrence for CanMult(i, k, ℓ) 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 $1 \le i \le k \le n$ and ℓ, and we compute CanMult(i, k, ℓ) 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 1, direct inspection reveals that it runs in $O(n^3)$ time and $O(n^2)$ space.

Algorithm 1 Multiplication Problem

Input: $S[1, \ldots, n]$ a string of letters from {a, b, c}.
Output: Yes/No, whether S can be parenthesized such that the result of multiplication is a.

```
1:  procedure MultiplyToA(S[1,...,n])
2:      CanMult[1,...,n][1,...,n][a, b, c] ← fresh n × n × 3 array
3:      for i = 1 to n do
4:          CanMult[i][i][ℓ] ← { True  if S[i] = ℓ
                               { False otherwise
5:      for s = 1 to n − 1 do    (s is k − i)
6:          for i = 1 to n − s do
7:              k ← i + s
8:              CanMult[i][k][ℓ] ← False    for each ℓ ∈ {a, b, c}
9:              for j = i to k − 1 do
10:                 for ℓ1, ℓ2 ∈ {a, b, c} do
11:                     ℓ ← ℓ1ℓ2 (using multiplication table)
12:                     if CanMult[i][j][ℓ1] ∧ CanMult[j + 1][k][ℓ2] then
13:                         CanMult[i][k][ℓ] ← True
14:     return CanMult[1][n][a]
```


CS 577: Introduction to Algorithms Fall 2022

Homework 4 Solutions to Regular Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

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 $0 \le t_1 \le t_2 \le \ldots \le t_{k-1} \le n$ (indicating the last substances put in bottles 1 through $k-1$) such that

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

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* $1 \le i \le j \le n$ 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. Your algorithm should output the minimum total energy.

Part (a)

We show how to efficiently calculate all the possible sums $\sum_{i \le \ell < m \le j} e_{\ell m}$ 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 2 and 5 is the sum $e_{23} + e_{24} + e_{25} + e_{34} + e_{35} + e_{45}$, so we need to add the entries of the highlighted triangle in Figure 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						

Figure 1: Example sum of energies

In general, for every two substances i, j we need the sum of the values inside such a triangle. Let TotalEnergy be an $n \times n$ array such that

$$\text{TotalEnergy}[i,j] = \sum_{i \le \ell < m \le j} e_{\ell m}$$

As there are $O(n^2)$ entries, it suffices to compute each entry in time $O(1)$ in order to guarantee an overall running time of $O(n^2)$. Let us look back at our example. We want to com-

pute TotalEnergy[2, 5], so let us look at other entries that could help us compute this value. TotalEnergy[2, 4] contains many of the values we need to consider, but it is missing the pairs that end with 5, as shown in Figure 2. If we also consider TotalEnergy[3, 5], then we will only be missing e_{25}, as shown in Figure 3. Note, however, that we end up adding e_{34} twice, so we need to subtract it to get the sum we want. This way, we can compute TotalEnergy[2, 5] = e_{25} + TotalEnergy[2, 4] + TotalEnergy[3, 5] − TotalEnergy[3, 4]. We write TotalEnergy[3, 4] instead of just e_{34} because for larger intervals we may add more than one value twice.

	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						

Figure 2: Values whose sum equals TotalEnergy[2, 4]

	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						

Figure 3: Values whose sum equals TotalEnergy[3, 5]

This idea can be generalized to compute TotaEnergy[i, j] for any $1 \le i < j \le n$. In this case, our final recurrence is

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

Even though we need to compute TotalEnergy[i, j] for all $1 \le i \le j \le n$, a recursive implementation with memoization (where we make $O(n^2)$ calls to the recursive function) suffices for our purposes. If we wish to compute these values iteratively, we can do so by filling in the TotalEnergy table diagonal-by-diagonal, more precisely for increasing value of $j - i$, and compute the values in the diagonal in any order.

Correctness Correctness follows by arguing that the recurrence for TotalEnergy correctly computes its definition. The key step here is the equality

$$\sum_{i \le \ell < m \le j} e_{\ell m} = e_{ij} + \sum_{i \le \ell < m \le j-1} e_{\ell m} + \sum_{i+1 \le \ell < m \le j} e_{\ell m} - \sum_{i+1 \le \ell < m \le j-1} e_{\ell m},$$

where an inductive hypothesis on $j - i$ allows us to replace each of the three sums in the right-hand side by a position in the TotalEnergy table.

Time and space analysis TotalEnergy has $O(n^2)$ entries to fill in. Each entry takes constant time, so a recursive implementation with memoization uses an overall $O(n^2)$ time and $O(n^2)$ space. If we compute these values iteratively diagonal-by-diagonal, then we only need $O(n)$ space to keep track of the previous two diagonals. However, if we take the space of the output into account, we need $O(n^2)$ space (and that is the best we can hope for).

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 $0 \le t_1 \le t_2 \le \ldots \le t_{k-1} \le n$ 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 OPT(i, j).

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

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

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. Note that the latter is equal to TotalEnergy[t + 1, j], which we showed how to compute in part (a). Because of this, we can precompute the values of all entries in TotalEnergy and use those instead of the sum in the recurrence for OPT. With this change, the recurrence looks like

$$\text{OPT}(i,j) = \min_{0 \le t \le i} \{\text{OPT}(t, j-1) + \text{TotalEnergy}[t+1, i]\}.$$

The base cases correspond to $j = 1$, i.e., all the first i substances go into a single bottle. In that case we have 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 OPT(n, k) as we want the minimum energy when we have n substances and k bottles in total.

Correctness formally follows from a proof by induction on $i + j$. The key idea is that we try all possibilities for which substance to put first in the last bottle and choose the one that minimizes the overall sum. The inductive hypothesis allows us to replace the OPT calls in the recurrence by their specification, which suffices because we only need to consider optimal solutions for each subproblem.

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

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. 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 he is left with at the end.

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.

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

where $1 \le i \le A$ and $1 \le j \le B$. 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 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 4.

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 AB 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 set OPT(a_i, b_i) and OPT(b_i, a_i) to 1. This takes $O(n)$ time, and we automatically know that sculpt will return 0 for any other entry, so we avoid having to compute sculpt every time. Still, we may need to update the values for OPT(a_i, b_i) later in case there is a better way of solving that subproblem (cutting two smaller rectangles instead of a larger one, for example), but this is taken care of by the recurrence. The total complexity then becomes $O(n) + O(AB(A + B)) = O(n + AB(A + B))$.

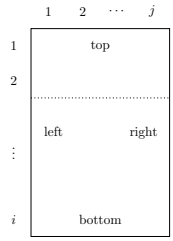

Figure 4: Horizontal cut in a $i \times j$ sheet of paper, with edge labels.

Problem 5

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 $mn/4$ votes in both districts. 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 $\mathrm{OPT}(i, j, k)$ 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 $|I| = j$, or $-\infty$ if there are no such subsets I. 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 $i \notin \mathcal{O}$ then $\mathrm{OPT}(i, j, k) = \mathrm{OPT}(i - 1, j, k)$.
- If $i \in \mathcal{O}$ 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 $i = 0$. For j also zero we have $\mathrm{OPT}(0, 0, k) = 0$ for every $k \geq 0$ as is witnessed by $I = \emptyset$. For $j > 0$ there is no way to pick exactly j elements from the empty set, so $\mathrm{OPT}(0, j, k) = -\infty$ for $j > 0$ and $k \geq 0$.

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.