

Homework 4 Solutions

Instructor: Dieter van Melkebeek

TA: Nicollas Mocelin Sdroievski

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 $\text{OPT}(i, j)$ denote the length of a shortest common supersequence of $a[1, \dots, i]$ and $b[1, \dots, j]$. Per the above discussion, $\text{OPT}(i, j)$ can be computed recursively as follows, for $1 \leq i \leq n$ and $1 \leq j \leq m$:

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

and where $\text{OPT}(i, 0) = i$, $\text{OPT}(0, j) = j$, and $\text{OPT}(0, 0) = 0$. (In fact, if $a_i = b_j$ one can argue that $\text{OPT}(i, j) = \text{OPT}(i-1, j-1) + 1$. 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 $\text{OPT}(i, j)$ 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 $i = 0, 1, \dots$, and for each i , iterate through choices of $j = 0, 1, \dots$. 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 $\text{OPT}(i, j)$ correctly computes the definition of $\text{OPT}(i, j)$. 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 $a[1, \dots, n]$, $b[1, \dots, m]$

Output: length of shortest supersequence of a and b

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

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

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

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

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

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

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 $O(\min(m, n))$. If we care about finding the optimal sequence c , the same Divide and Conquer approach from class to obtain the solution for sequence alignment also works here, and we can find c in time $O(mn)$ and space $O(m + n)$.

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 $\mathbf{a}, \mathbf{b}, \mathbf{c}$, 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 \mathbf{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 \mathbf{a} , if one exists at all. If the second operand is \mathbf{a} , then the first operand must be \mathbf{c} . We do not have to consider \mathbf{b} as the second operand, because there is no way to right multiply by \mathbf{b} and obtain \mathbf{a} . If the second operand is \mathbf{c} , then there are two possibilities for the first operand: either \mathbf{a} or \mathbf{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 \mathbf{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 $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$. Let the input string be $S[1, \dots, n]$. We define $\text{CanMult}(i, k, \ell)$ for $1 \leq i \leq k \leq n$ to indicate whether $S[i, \dots, k]$ can be parenthesized to compute ℓ . We wish to know $\text{CanMult}(1, n, \mathbf{a})$. Following the above discussion, we can compute it with the following recurrence:

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

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: $S[1, \dots, n]$ a string of letters from $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$.

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

```
1: procedure MULTIPLYTOA( $S[1, \dots, n]$ )
2:   CanMult[1, ..., n][1, ..., n][ $\mathbf{a}, \mathbf{b}, \mathbf{c}$ ]  $\leftarrow$  fresh  $n \times n \times 3$  array
3:   for  $i = 1$  to  $n$  do
4:     CanMult[ $i$ ][ $i$ ][ $\ell$ ]  $\leftarrow \begin{cases} \mathbf{True} & \text{if } S[i] = \ell \\ \mathbf{False} & \text{otherwise} \end{cases}$ 
5:   for  $s = 1$  to  $n - 1$  do ( $s$  is  $k - i$ )
6:     for  $i = 1$  to  $n - s$  do
7:        $k \leftarrow i + s$ 
8:       CanMult[ $i$ ][ $k$ ][ $\ell$ ]  $\leftarrow \mathbf{False}$  for each  $\ell \in \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ 
9:       for  $j = i$  to  $k - 1$  do
10:        for  $\ell_1, \ell_2 \in \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$  do
11:           $\ell \leftarrow \ell_1 \ell_2$  (using multiplication table)
12:          if CanMult[ $i$ ][ $j$ ][ $\ell_1$ ]  $\wedge$  CanMult[ $j + 1$ ][ $k$ ][ $\ell_2$ ] then
13:            CanMult[ $i$ ][ $k$ ][ $\ell$ ]  $\leftarrow \mathbf{True}$ 
14:   return CanMult[1][ $n$ ][ $\mathbf{a}$ ]
```

Problem 3

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

$$\sum_{i=1}^k \left(\max_{t_{i-1} < \ell < m \leq t_i} \{s_{\ell m}\} \right)$$

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

- (a) Design an $O(n^2)$ algorithm that outputs the maximum interference values for the subinstances consisting of antennas 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 antennas and a given k . Your algorithm should output the minimum total interference that can be realized under the constraints.

Part (a)

We show how to efficiently compute $\max_{i \leq \ell < m \leq j} \{s_{\ell m}\}$ for any i, j . We think of the interference values s_{ij} as being organized in an (upper-triangular) 2-dimensional array. In the example array below, for six antennas, the interference value for antennas between 2 and 5 is $\max\{s_{23}, s_{24}, s_{25}, s_{34}, s_{35}, s_{45}\}$, so we need to consider the entries of the highlighted triangle in Table 1 shown below.

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

Table 1: Example max interference

In general, for every two antennas i, j we need the max of the values inside such a triangle. Let MaxInterference be an $i \times j$ array such that

$$\text{MaxInterference}[i][j] = \max_{i \leq \ell < m \leq j} \{s_{\ell m}\}$$

Since this array has $O(n^2)$ entries, we need to be able to compute each entry in constant time. Let us look back at our example. We want to compute $\text{MaxInterference}[2][5]$, so let's think about other entries that could help us compute this value. $\text{MaxInterference}[2][4]$ includes many of the values we need to consider, but it is missing the pairs that end with 5, as shown in Table 2. If we also consider $\text{MaxInterference}[3][5]$, then we will only be missing s_{25} , as shown in Table 3. Note that there is no problem in considering s_{34} twice since we are taking the max in the end. Thus, we can compute $\text{MaxInterference}[2][5] = \max\{\text{MaxInterference}[2][4], \text{MaxInterference}[3][5], s_{25}\}$.

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

Table 2: Values considered by MaxInterference[2][4]

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

Table 3: Values considered by MaxInterference[3][5]

This idea can be generalized to compute $\text{MaxInterference}[i][j]$ for any $1 \leq i < j \leq n$. In this case, our final recurrence is

$$\text{MaxInterference}[i][j] = \begin{cases} 0 & \text{if } i = j \\ \max\{\text{MaxInterference}[i][j-1], \text{MaxInterference}[i+1][j], s_{ij}\} & \text{if } i < j \end{cases}.$$

Correctness Correctness follows by arguing that the recurrence for MaxInterference correctly computes its definition. We leave a formal proof by induction as an exercise.

Time and space analysis MaxInterference has $O(n^2)$ 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 antennas and k locations and we need to divide the antennas into the k locations. Since we are given fewer locations than antennas ($k < n$) we essentially need to decide which antennas will be placed in the same location. Remember that we cannot change the order of the antennas. What we need to decide is in which $k-1$ indices we will place a "separator" to minimize the amount of total interference, where a "separator" is the point at which we stop placing antennas in the i 'th location and start placing them in the next one. Essentially determine integers $0 \leq t_1 \leq t_2 \leq \dots \leq t_{k-1} \leq n$ indicating the last antennas placed in locations 1 through $k-1$.

Subproblems The problem is trivial for 1 location, as there is no choice to be made and the minimum total interference is just the maximum out of all pairwise interference values.

Let's say we have 2 locations. In that case we need to find an index t such that all antennas from 1 to t are in the first location, and all antennas from $t + 1$ until n are in the second one. And we want the sum $(\max_{1 \leq \ell < m \leq t} \{s_{\ell m}\}) + (\max_{t+1 \leq \ell < m \leq n} \{s_{\ell m}\})$ to be the minimum possible. We try all possibilities for $t \in \{0, \dots, n\}$ and select the one that minimizes the above sum.

If we have $k > 2$ locations, we consider the last decision that we need to make, i.e., the choice of $t_{k-1} \in \{0, \dots, n\}$. 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 antennas 1 through t_{k-1} into $k - 1$ locations, i.e., to solve the problem for the instance defined by antennas 1 through t_{k-1} with the given interference values $s_{\ell m}$ for $1 \leq \ell < m \leq t_{k-1}$, and $k - 1$ locations. We recursively solve those instances, and then select the value of t_{k-1} that minimizes the sum of the minimum total interference for the subinstance and the interference in the last location.

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

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

$$\text{OPT}(i, j) = \min_{0 \leq t \leq i} \left\{ \text{OPT}(t, j-1) + \max_{t+1 \leq \ell < m \leq i} \{s_{\ell m}\} \right\},$$

where the max is the interference in the j -th location, which contains antennas from $t + 1$ to i . This expression says that for location j we find the antenna t to stop the previous location with such that the total interference is minimized, where the total interference is the minimum possible interference when placing antennas 1 through t in $j - 1$ locations plus the interference when placing antennas $t + 1$ to i in location 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 antennas are placed into a single location. In that case we have $\text{OPT}(i, 1) = \max_{1 \leq \ell < m \leq i} \{s_{\ell m}\}$. 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 $\text{OPT}(n, k)$ as we want the minimum total interference when we have n antennas and k locations in total.

Algorithm and analysis The algorithm starts by precomputing MaxInterference from part (a). Next we build a table for OPT . This table has $O(n \cdot k)$ entries (i.e. antennas \times locations). 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 MaxInterference . Thus, the total amount of work is $O(kn^2)$ to fill in OPT . This plus the $O(n^2)$ time to compute MaxInterference 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 to 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 MaxInterference .

Problem 4

Design an algorithm that on input $A, B, a_1, \dots, a_n, b_1, \dots, b_n$, 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.

$\text{OPT}(i, j) =$ 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:

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

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.

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 have a table for sculpt and perform a preprocessing step where for each of the n sculptures (a_i, b_i) we set $\text{sculpt}(a_i, b_i)$ to 1 and the remaining positions to 0. Setting up the table takes $O(AB)$ time, and updating the values for all sculptures takes $O(n)$ time, resulting in a total time of $O(n + AB)$ for this preprocessing step. We can then keep the recurrence as it is, but now with constant cost for each access to sculpt . The total complexity then becomes $O(n + AB) + O(AB(A + B)) = O(n + AB(A + B))$. Note that this doubles the space requirement, but this does not make a difference since we don't care about constant factors. We could also merge the two tables into one, but this complicates the recurrence relation.

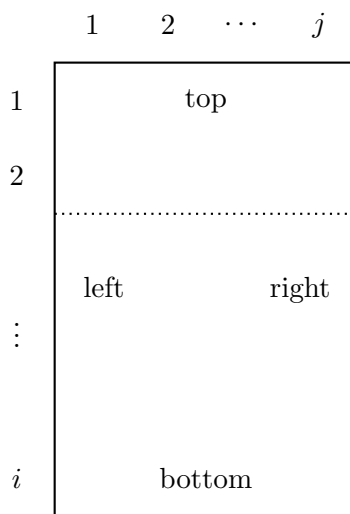


Figure 1: Horizontal cut in a $i \times j$ sheet of paper, with edge labels.

Problem 5

Suppose we have a set of precincts P_1, P_2, \dots, 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 $I \subseteq \{1, 2, \dots, i\}$ 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 $OPT(i, j, k)$ look like? Consider an optimal solution \mathcal{O} for $OPT(i, j, k)$. For $i > 0$ and $j > 0$, we have the following:

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

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

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

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

Running time analysis The resulting algorithm computes the values $\text{OPT}(i, j, k)$ 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.