Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Johannes Lengler, David Steurer
Kasper Lindberg, Lucas Slot, Hongjie Chen, Manuel Wiedmer

4 November 2024

Algorithms & Data Structures — Exercise sheet 7 — HS 24

The solutions for this sheet are submitted on Moodle until 10 November 2024, 23:59.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

You can use results from previous parts without solving those parts.

The solutions are intended to help you understand how to solve the exercises and are thus more detailed than what would be expected at the exam. All parts that contain explanation that you would not need to include in an exam are in grey.

Exercise 7.1 *Subset sums with duplicates* **(1 point)**.

Let $A[1 \dots n]$ be an array containing n positive integers and let $b \in \mathbb{N}$. We want to know if we can write b as a subset sum of A where each element of A is allowed to be used once, twice or not at all. If this is possible, we say b is a subset sum of A with duplicates.

For example, consider $A = [3, 4, 2, 7]$ and $b = 22$. Then b is a subset sum of A with duplicates, as we can use 3 not at all, use 4 once, and use 2 and 7 twice. In other words, we can write b as $0 \cdot A[1] + 1 \cdot A[2] + 2 \cdot A[3] + 2 \cdot A[4]$.

Describe a DP algorithm that, given an array $A[1 \dots n]$ of positive integers and a positive integer b, returns True if and only if b is a subset sum of A with duplicates. Your algorithm should have asymptotic runtime complexity at most $O(b \cdot n)$.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps? Describe the calculation order in pseudocode.
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Solution:

1. *Dimensions of the DP table*: $DP[0 \dots n][0 \dots b]$
2. *Subproblems*: $DP[a][s]$ is True if, and only if, s can be written as a sum $\sum_{i \in I} c_i \cdot A[i]$ where $I \subseteq \{i : 1 \leq i \leq a\}$, and $c_i \in \{1, 2\}$ for each $i \in I$.

3. *Recursion*: DP can be computed recursively as follows:

$$DP[0][s] = \texttt{False} \qquad 1 \leq s \leq b \tag{1}$$
$$DP[a][0] = \texttt{True} \qquad 0 \leq a \leq n \tag{2}$$
$$DP[a][s] = DP[a-1][s] \texttt{ or } \qquad 1 \leq a \leq n, \tag{3}$$
$$DP[a-1][s-1 \cdot A[a]] \texttt{ or } DP[a-1][s-2 \cdot A[a]] \qquad 1 \leq s \leq b. \tag{4}$$

   Note that in equation (3), the entries '$DP[a-1][s-c \cdot A[a]]$' for $c \in \{1, 2\}$ might fall outside the range of the table, in which case we treat them as `False`.

   Equation (1) expresses that positive values cannot be equal to an empty sum (which equals 0). Equation (2) says 0 can always be written as an (empty) subset sum of $A[1], \ldots, A[a]$. Equation (3) provides the recurrence relation. Namely, it expresses that an integer s can be written as a subset sum with duplicates of $A[1], \ldots, A[a]$ if and only if $s - c \cdot A[a]$ can be written as a subset sum with duplicates of $A[1], \ldots, A[a-1]$ where $c \in \{0, 1, 2\}$.

4. *Calculation order*: Following the recurrence relations above, we compute the entries as follows:

   for $a = 0 \ldots n$ **do**
      for $s = 0 \ldots b$ **do**
         Compute $DP[a][s]$.

   Alternatively, we can (for example) also use the following calculation order:

   for $s = 0 \ldots b$ **do**
      for $a = 0 \ldots n$ **do**
         Compute $DP[a][s]$.

5. *Extracting the solution*: The solution can be found in $DP[n][b]$, by part 2.

6. *Running time*: The running time of the solution is $O(nb)$ as there are $(n+1) \cdot (b+1)$ entries in the table, each entry requires $O(1)$ time to compute, and we extract the solution in $O(1)$ time.

An alternative solution is to run the subset sum algorithm (you saw in the lecture notes) on the array $A'[1 \ldots 2n]$ where $A'[i] = A'[i+n] = A[i]$ for $i \in [n]$. That is, A' is obtained from A by duplicating each element of A once.

Guidelines for correction:

This exercise is very close to regular subset sum, and so the emphasis should be on correctly defining the meaning of a table entry and its computation (part 2 and part 3). Award 1 point if these are both correct (with proper justification for the recurrence), and 1/2 if the definition/recursion are correct, but the justification is not. Do not subtract points for missing edge-cases (where the index is out of range) in part 3.

Exercise 7.2 *Road trip.*

You are planning a road trip for your summer holidays. You want to start from city C_0, and follow the only road that goes to city C_n from there. On this road from C_0 to C_n, there are $n-1$ other cities $C_1, \ldots, C_{n-1}$ that you would be interested in visiting (all cities $C_1, \ldots, C_{n-1}$ are on the road from C_0 to C_n). For each $0 \leq i \leq n$, the city C_i is at kilometer k_i of the road for some given $0 = k_0 < k_1 < \ldots < k_{n-1} < k_n$.

You want to decide in which cities among $C_1, \ldots, C_{n-1}$ you will make an additional stop (you will stop in C_0 and C_n anyway). However, you do not want to drive more than d kilometers without making a stop in some city, for some given value $d > 0$ (we assume that $k_i < k_{i-1} + d$ for all $i \in [n]$ so that this is satisfiable), and you also don't want to travel backwards (so from some city C_i you can only go forward to cities C_j with $j > i$).

(a) Provide a *dynamic programming* algorithm that computes the number of possible routes from C_0 to C_n that satisfy these conditions, i.e., the number of allowed subsets of stop-cities. Your algorithm should have $O(n^2)$ runtime.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps? Describe the calculation order in pseudocode.
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Solution:

To solve such tasks, start by defining the subproblem and try to come up with possible recursions. Sketching and solving examples by hand can be helpful.

In this case, a table entry $T[i]$ contains the number of different routes to C_i and city C_i can be reached from C_j (for $0 \leq j < i$) if $k_i \leq k_j + d$. As appending C_i to these previous routes creates new unique routes, the recursion is described by (5).

1. *Dimensions of the DP table*: The DP table is linear, and its size is $n + 1$.
2. *Subproblems*: $DP[i]$ is the number of possible routes from C_0 to C_i (which stop at C_i).
3. *Recursion*: Initialize $DP[0] = 1$.

   For every $i > 0$, we can compute $DP[i]$ using the formula

$$DP[i] = \sum_{\substack{0 \leq j < i \\ k_i \leq k_j + d}} DP[j]. \tag{5}$$

   This recursion is correct since city C_i can be reached from C_j (for $0 \leq j < i$) if and only if $k_i \leq k_j + d$. Summing up the number of routes from C_0 to these C_j, we get the number of routes from C_0 to C_i.

4. *Calculation order*: We can calculate the entries of DP as follows:

   for $i = 0 \ldots n$ **do**
   Compute $DP[i]$.

5. *Extracting the solution*: All we have to do is read the value at $DP[n]$.

6. *Running time*: For $i = 0$, $DP[0]$ is computed in $O(1)$ time. For $i \geq 1$, the entry $DP[i]$ is computed in $O(i)$ time (as we potentially need to take the sum of i entries). Therefore, the total runtime is $O(1) + \sum_{i=1}^{n} O(i) = O(n^2)$.

   We can actually compute each entry of the DP table in $O(1)$ amortized time. Thus the running time of the above DP algorithm can be improved from $O(n^2)$ to $O(n)$, which means we don't need the additional assumption in the following part (b) to argue the above DP has $O(n)$ running time.

   Now we sketch the proof. For each $i \in [n]$, let

$$\ell(i) = \min\{j : 0 \leq j < i \text{ and } k_i \leq k_j + d\}\,.$$

   That is, $\ell(i)$ is the smallest index j satisfying $k_i \leq k_j + d$. Then the recursion formula (5) can be equivalently written as

$$DP[i] = \sum_{\ell(i) \leq j < i} DP[j]\,.$$

   Assume for now we already know the values of $\ell(1), \ell(2), \ldots, \ell(n)$. We can store an array $T[0..n]$ of prefix sums of $DP[0..n]$, i.e. $T[i] = \sum_{0 \leq j \leq i} DP[j]$ for each i. Then $DP[i]$ can be computed as follows,

$$DP[i] = T[i-1] - T[\ell(i) - 1]\,.$$

   (Note we need to be more careful with boarder cases such as $\ell(i) = 0$, but this should be rather standard, so we leave it as an exercise for you.)

   To show $\ell(1), \ell(2), \ldots, \ell(n)$ can be computed in $O(n)$ time, it suffices to observe that $\ell(i)$ is increasing as i increases. Thus each $\ell(i)$ can be computed in $O(1)$ amortized time.

(b) If you know that $k_i > k_{i-1} + d/10$ for every $i \in [n]$, how can you turn the above algorithm into a linear time algorithm (i.e., an algorithm that has $O(n)$ runtime) ?

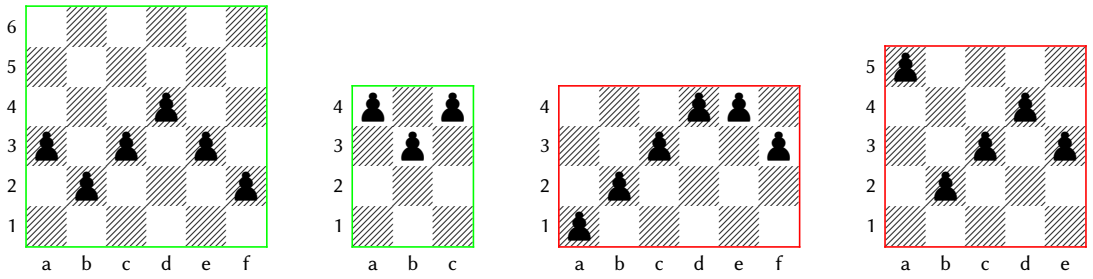
Solution:

Assuming that $k_i > k_{i-1} + d/10$ for all i, we know that $k_i > k_{i-10} + d$, and hence $k_i > k_j + d$ for all $j \leq i - 10$. Therefore, the sum in formula (5) contains at most 10 terms $DP[j]$ (and for each of them we can check in constant time whether we should include it or not, i.e., whether $k_i \leq k_j + d$). So in this case the computation of the entry $DP[i]$ takes time $O(1)$ for all $0 \leq i \leq n$, and hence the total runtime is $O(n)$.

Exercise 7.3 *Safe pawn lines.*

On an $N \times M$ chessboard (N being the number of rows and M the number of columns), a *safe pawn line* is a set of M pawns with exactly one pawn per column of the chessboard, and such that every two pawns from adjacent columns are located diagonally to each other. When a pawn line is not safe, it is called *unsafe*.

The first two chessboards below show safe pawn lines, the latter two unsafe ones. The line on the third chessboard is unsafe because pawns d4 and e4 are located on the same row (rather than diagonally); the line on the fourth chessboard is unsafe because pawn a5 has no diagonal neighbor at all.



Describe a DP algorithm that, given $N, M > 0$, counts the number of safe pawn lines on an $N \times M$ chessboard. Your solution should have complexity at most $O(NM)$.

In your solution, address the following aspects:

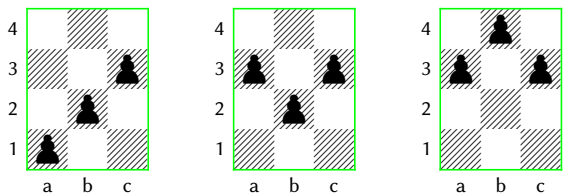
1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps? Describe the calculation order in pseudocode.
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Solution:

To come up with the solution, start by imagining an unbounded chessboard. Think about the relation between a specific field and its predecessors to the left, then formulate a recursion. When you are sure that it works for the unbounded board, restrict it and handle the base cases, i.e. the fields on any border.

1. *Dimensions of the DP table*: $DP[1 \dots N][1 \dots M]$
2. *Subproblems*: $DP[i][j]$ counts the number of distinct safe pawn lines on an $N \times j$ chessboard with the pawn in the last column located in row i.

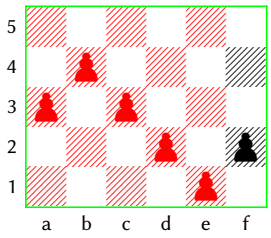
   For example, for $N = 4$, we have $DP[3][3] = 3$, since 3 safe pawn lines on a 4×3 chessboard have their last pawn in row 3, namely:



3. *Recursion*: DP can be computed recursively as follows:

$$DP[i][1] = 1 \qquad 1 \leq i \leq N \tag{6}$$
$$DP[1][j] = DP[2][j-1] \qquad 1 < j \leq M \tag{7}$$
$$DP[N][j] = DP[N-1][j-1] \qquad 1 < j \leq M \tag{8}$$
$$DP[i][j] = DP[i-1][j-1] + DP[i+1][j-1] \qquad 1 < i < N, 1 < j \leq M \tag{9}$$

Equation (6) solves the base case where the chessboard has only one column. In that case, there exists exactly one safe pawn line with the pawn in the last column located in row i. Equation (9) provides the general recurrence formula. The rationale behind this formula it is as follows: a pawn line on an $N \times j$ chessboard with its last pawn in row i is obtained by adding a single pawn located at (j, i) (the black pawn on the board below) to a pawn line on a $N \times (j-1)$ chessboard (the red pawns on first board below). Clearly, the last pawn of the smaller line must be on row $i+1$ or $i-1$. Hence, we have $DP[i][j] = DP[i-1][j-1] + DP[i+1][j-1]$. However, this is not true when we have the edge cases $i = 1$ or $i = N$. In these cases, only one position is available for the last pawn of the smaller line, yielding formulae (7) and (8).



4. *Calculation order*: We compute the table as follows:

   for $j = 1 \ldots M$ **do**
   for any order of $i \in \{1, \ldots, N\}$ **do**
   Compute $DP[i][j]$.

   For example, we can compute the second for-loop in increasing order.

5. *Extracting the solution*: The solution is $\sum_{i=1}^{N} DP[i][M]$.

6. *Running time*: The running time of the solution is $O(MN)$, as there are NM entries in the table which are processed in $O(1)$ time, and extracting the solution takes $O(N) \leq O(MN)$ time.

Exercise 7.4 *Weight and volume knapsack* **(1 point)**.

Consider a knapsack problem with n items with profits being positive integers in the array $P[1, \ldots, n]$, and weights being positive integer in the array $W[1, \ldots, n]$. The knapsack has a weight limit $W_{max} \in \mathbb{N}$. Furthermore, each item has a volume of 1 and the knapsack has a volume limit V_{max}.

Describe a DP algorithm that, given the arrays $P[1, \ldots, n]$, $W[1, \ldots, n]$, of positive integers and positive integers W_{max}, V_{max}, returns the total profit of the largest subset of items such that the items respect both the weight limit and volume limit of W_{max} and V_{max}. Your algorithm should have asymptotic runtime complexity at most $O(n \cdot W_{max} \cdot V_{max})$.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?

2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps? Describe the calculation order in pseudocode.
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Solution:

1. *Dimensions of the DP table*: $DP[0 \dots n][0 \dots W_{max}][0 \dots V_{max}]$
2. *Subproblems*:

   $DP[a][s][t] =$ Maximum profit of a subset of items of $1 \dots a$ which has at most weight s and volume t

3. *Recursion*: DP can be computed recursively as follows:

$$DP[0][s][t] = 0 \qquad 0 \le s \le W_{max}, \ 0 \le t \le V_{max} \tag{10}$$

$$DP[a][s][t] = \max\{DP[a-1][s][t], DP[a-1][s-W[a]][t-1] + P[a]\} \quad 1 \le a \le n, \ 1 \le s \le W_{max}, \ 1 \le t \le V_{max}. \tag{11}$$

   Note that in equation (11), the entries '$DP[a-1][s-W[a]][t-1]$' might fall outside the range of the table, in which case we treat them as $-\infty$.

   Equation (10) says that no items to select means 0 profit. Equation (11) provides the recurrence relation. The idea behind this formula is that the max profit from a subset of $1 \dots a$ which respects both a weight and volume limit of s and t respectively must either include not include the element or not.

   If it does not include the element a then it must be a subset of $1 \dots a-1$ which respects the weight and volume limits, namely $DP[a-1][s][t]$.

   Otherwise, we include the element a which means we have to handle the new added weight $W[a]$ and volume of 1. This can only be done if $s - W[a] \ge 0$ and $t - 1 \ge 0$ and thus we add the relevant profit $P[a]$. These conditions are enforced by the fact that the array is equal to $-\infty$ so that we always have $-\infty + P[a] \le 0$ and thus cannot include a.

4. *Calculation order*: Following the recurrence relations above, we compute the entries as follows (note that there are also many other valid calculation orders):

   for $a = 0 \dots n$ **do**
   for $s = 0 \dots W_{max}$ **do**
   for $t = 0 \dots V_{max}$ **do**
   Compute $DP[a][s][t]$.

5. *Extracting the solution*: The solution can be found in $DP[n][W_{max}][V_{max}]$ by part 2.

6. *Running time*: The running time of the solution is $O(n \cdot W_{max} \cdot V_{max})$ as there are $(n+1) \cdot (W_{max} + 1) \cdot (V_{max} + 1)$ entries in the table, each entry requires $O(1)$ time to compute, and we extract the solution in $O(1)$ time.

Guidelines for correction:

This exercise is close to regular knapsack, and so the emphasis should be on correctly defining the meaning of a table entry and its computation (part 2 and part 3). Award 1 point if these are both correct (with proper justification for the recurrence), and 1/2 if the definition/recursion are correct, but the justification is not. Do not subtract points for missing edge-cases (where the index is out of range) in part 3.

Exercise 7.5 *Zebra arrays* **(1 point)**.

A square two-dimensional array $Z[1 \dots k][1 \dots k]$ with entries in $\{0, 1\}$ is called a *zebra array* if no two adjacent entries of Z are equal. We say two distinct entries $Z[i_1][j_1]$ and $Z[i_2][j_2]$ in Z are adjacent if

- $i_1 = i_2$ and $|j_1 - j_2| \leq 1$; or
- $|i_1 - i_2| \leq 1$ and $j_1 = j_2$.

Describe a DP algorithm that, given a two-dimensional array $A[1 \dots n][1 \dots m]$ with entries in $\{0, 1\}$, outputs the size of a largest zebra array contained in A. That is, the largest k such that, for some $1 \leq i \leq n - k + 1, 1 \leq j \leq m - k + 1$, the array $A[i \dots i + k - 1][j \dots j + k - 1]$ is a zebra array. Your algorithm should have asymptotic runtime complexity at most $O(nm)$.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps? Describe the calculation order in pseudocode.
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Hint: *U*se a DP table $B[1 \dots n][1 \dots m]$. The meaning of entry $B[i][j]$ is

$$B[i][j] = \text{size of the largest zebra array in } A \text{ whose bottom-right entry is } A[i][j].$$

Hint: *Y*our recursion to compute $B[i][j]$ should involve the entries $B[i][j-1]$, $B[i-1][j]$, $B[i-1][j-1]$ and also the entries $A[i][j]$, $A[i][j-1]$, $A[i-1][j]$, $A[i-1][j-1]$.

Solution:

1. *Dimensions of the DP table*: We use a table $B[1 \dots n][1 \dots m]$ of size $n \times m$.
2. *Subproblems*: The meaning of entry $B[i][j]$ is

$$B[i][j] = \text{size of the largest zebra array in } A \text{ whose bottom-right entry is } A[i][j].$$

3. *Recursion*: As base cases, we set $B[i][j] = 1$ for all (i, j) with $i = 1$ or $j = 1$. Then, for $n \geq i > 1$, $m \geq j > 1$, we compute $B[i][j]$ recursively via:

   - If $A[i][j] = A[i][j-1]$, or $A[i][j] = A[i-1][j]$, or $A[i][j] \neq A[i-1][j-1]$, then set $B[i][j] = 1$.
   - Otherwise, set

   $$B[i][j] = 1 + \min\{B[i][j-1],\ B[i-1][j],\ B[i-1][j-1]\}.$$

   We show correctness as follows.

   First, note that all the base cases are certainly correct, because for an entry $A[i][j]$ at the boundary of the array, there is only one square sub array of A whose bottom right entry is $A[i][j]$, namely $A[i \ldots i][j \ldots j]$, which is of size 1 and thus automatically a zebra array.

   For the cases $i > 1, j > 1$, note first that, if the if-statement is true, it is indeed the case that any square subarray of A of size at least 2 whose bottom right entry is $A[i, j]$ cannot be a zebra array. So $B[i][j]$ is correctly set to 1 in this case.

   Finally, we consider the case where the if-statement does not hold, and $B[i][j]$ is set via the recursion. Suppose that there is a zebra array in A of size $k \geq 2$ whose bottom-right entry is $A[i][j]$. Then, there must be a zebra array of size $k-1$ whose bottom right entry is $A[i][j-1]$, and also one whose bottom right entry is $A[i-1][j]$, and also one whose bottom right entry is $A[i-1][j-1]$. Indeed, any square subarray of a zebra array is itself a zebra array. This shows that

   $$B[i][j] \leq 1 + \min\{B[i][j-1],\ B[i-1][j],\ B[i-1][j-1]\}.$$

   The reason is $B[i][j-1] \geq B[i][j]-1$, $B[i-1][j] \geq B[i][j]-1$, and $B[i-1][j-1] \geq B[i][j]-1$. For the other direction, assume there is a zebra array of size $k-1$ whose bottom right entry is $A[i][j-1]$, and also one whose bottom right entry is $A[i-1][j]$, and also one whose bottom right entry is $A[i-1][j-1]$. Then, we claim the array $A[i-k+1 \ldots i][j-k+1 \ldots j]$ is a zebra array (of size k, with bottom right entry $A[i][j]$). Note that the only entries of $A[i-k+1 \ldots i][j-k+1 \ldots j]$ for which the 'zebra-condition' could potentially not hold are $A[i][j],\ A[i][j-1],\ A[i-1][j],\ A[i-1][j-1]$. But, since the if-statement does not hold, we know that these entries do in fact satisfy the condition. Therefore,

   $$B[i][j] \geq 1 + \min\{B[i][j-1],\ B[i-1][j],\ B[i-1][j-1]\}.$$

   The reason is we can set $k-1 = \min\{B[i][j-1],\ B[i-1][j],\ B[i-1][j-1]\}$ in the above argument.

4. *Calculation order*: It is enough to make sure that $B[i][j]$ is computed only when all other entries $B[i'][j']$ with $i' \leq i$ and $j' \leq j$ have already been computed. This can be achieved, for instance, by the following calculation order:

   for $i = 1 \ldots n$ **do**
   for $j = 1 \ldots m$ **do**
   Compute $B[i][j]$.

5. *Extracting the solution*: We extract the solution by taking the maximum over all entries in B.

6. *Running time*: Our DP table is of size nm. Filling each entry takes $O(1)$ time. Extracting the maximum at the end takes $O(nm)$ time. So, in total, we use at most $O(nm)$ time.

Guidelines for correction:

The important elements of this exercise are:

- base cases
- recursion (if-statement)
- recursion (formula)
- justification that recursion is correct ($\geq$)
- justification that recursion is correct ($\leq$)
- calculation order
- extraction

Award 1 point if at least 6 elements are correct, award 1/2 point if at least 4 elements are correct.