Informatics II, Spring 2024, Solution Exercise 11

Publication of exercise: May 13, 2024
Publication of solution: May 20, 2024
Exercise classes: May 21 - 24, 2024

Learning Goal

- Learn how to find a recursion relation for DP to reconstruct the solution.
- Gain familiarity with DPs operating on arrays and matrices.
- Learn how to define the subproblems and understand the order of subproblems in DP.

Task 1 [Medium/Hard]

You are given an array of size n. Your task is to find the length of its *longest increasing subsequence*.

A *subsequence* of an array a is a sequence $a[i_1], a[i_2], ..., a[i_k] (k \geq 1)$ where the indexes satisfy the property $1 \leq i_1 < i_2 < ... < i_k \leq n$. You can consider a sequence is derived from the array by deleting some elements without changing the order of the remaining elements. For example, consider the following array a:

1	2	3	4	5	6	7	8
5	10	7	4	8	9	2	10

Figure 1: Array a

For example, valid subsequences are $[5, 7, 10]$, $[10, 7, 2, 10]$, $[9]$, while $[2, 4, 8]$ and $[5, 7, 7, 10]$ are not valid.

An *increasing* subsequence is a subsequence where all the elements are in increasing order, i.e. $a[i_1] \leq a[i_2] \leq ... \leq a[i_k]$. A *longest* increasing subsequence of an array a is an increasing subsequence of the largest length (it has the largest k).

For the array given above, the longest increasing subsequence is $[5, 7, 8, 9, 10]$.

We can solve this problem using dynamic programming:

$$dp_i = \text{the largest length of an increasing subsequence ending in index } i$$

The array dp for the example array above is:

1	2	3	4	5	6	7	8
1	2	2	1	3	4	1	5

Figure 2: Helper array *dp* for array *a*.

a. Given the array *b*

1	2	3	4	5	6	7	8
7	10	4	9	7	10	8	12

Figure 3: Array *b*

fill in its helper array *dp*:

1	2	3	4	5	6	7	8

Figure 4: Helper array *dp* for array *b*.

Hints: Try to find a pattern in how you approach this problem step by step. It often helps to establish an order or "direction" in which you solve the subproblems.

1	2	3	4	5	6	7	8
1	2	1	2	2	3	3	4

Figure 5: Helper array *dp* for array *b*.

b. Determine the recursive relation of *dp*.

Notice that we can split a subsequence ending at index i into the elements that come before i and the element at index i. The elements that come before i have indexes $< i$; let j be the index of the element that comes immediately before i.

1	2	$\cdots$	j	$\cdots$	i	$\cdots$	$n-1$	n
a_1	a_2	$\cdots$	a_j	$\cdots$	a_i	$\cdots$	a_{n-1}	a_n

a_i can be appended to the subsequence ending at index j only if $a_j \leq a_i$. We can look at all possible indexes j and select the j which brings us the maximum length:

$$dp_i = \begin{cases} 1 + \max(dp_j) & \text{for } 1 \leq j < i \text{ and } a_j \leq a_i \\ 1 & \text{if no such } j \text{ exists.} \end{cases} \tag{1}$$

The final answer to our problem is $\max(dp_i)$ for all $1 \leq i \leq n$ (look at the longest subsequence that ends in each index i).

c. Use a bottom-up approach to write the function `int longest_seq(int a[], int n)` which returns the length of the longest increasing subsequence of an array `a` of length `n`.

Solution: See code.

d. We can also calculate the longest increasing subsequence *itself* (the numbers which make up the subsequence) with the help of a *parent* array p. p_i will store the index of the subproblem of dp_i which gives the optimal solution for dp_i.

For the example of the array a from above

1	2	3	4	5	6	7	8
5	10	7	4	8	9	2	10

Figure 6: Array a

and it's helper array dp

1	2	3	4	5	6	7	8
1	2	2	1	3	4	1	5

Figure 7: Helper array dp for array a.

we have the following parent array p:

1	2	3	4	5	6	7	8
	1	1		3	5		6

Figure 8: Parent array p for array a.

Extend the function `int longest_sequence(int a[], int n)` to also print the elements of the longest increasing subsequence. If there are multiple answers, you can print any.

We store in p_i the index j from the recursive definition that brought us the maximum value of dp_i.

For example, for the array a:

1	2	3	4	5	6	7	8
5	10	7	4	8	9	2	10

Figure 9: Array a

and it's helper array dp

1	2	3	4	5	6	7	8
1	2	2	1	3	4	1	5

Figure 10: Helper array dp for array a.

we have the following parent array p:

1	2	3	4	5	6	7	8
	1	1		3	5		6

Figure 11: Parent array p for array a.

If the index of our final answer is k, then we can reconstruct the corresponding subsequence in this way: start from index k and print a_k. Then, go to the parent of k then print the value stored there, then go to the parent of the parent and so on. This process can be expressed in this way: for each transition to a parent, assign $k = p_k$.

Notice that we go through the subsequence starting from the last element and going until the first, so the numbers will actually be printed in reverse order. We can use a stack to store the elements and print them in the desired order.

Task 2 [Medium/Hard]

A path in a matrix M of dimensions $x \times y$ is defined as a sequence of cells $(i_1, j_1), (i_2, j_2), ..., (i_n, j_n)$ in the matrix M $(1 \leq i_k \leq x, 1 \leq j_k \leq y, \forall k \in \{1, 2, ..., n\})$ where each cell (i_{k+1}, j_{k+1}) must be directly below or to the right of the cell (i_k, j_k), but not above or to the left.

An *increasing* path in the matrix M is a path where the values of the cells are in strictly increasing order: $M[i_k][j_k] < M[i_{k+1}][j_{k+1}]$ for all k. The longest increasing path is an increasing path with the maximum number of cells (maximum n).

An example of a matrix M and an increasing path is:

$$\begin{pmatrix} 1 & 7 & 3 & 10 & 6 & 18 \\ 3 & 2 & 5 & 6 & 9 & 16 \\ 6 & 3 & 2 & 10 & 12 & 13 \\ 8 & 7 & 5 & 4 & 8 & 15 \end{pmatrix}$$

Figure 12: Example of the longest increasing path problem with the solution of length 7

We can solve this problem using dynamic programming:

$$dp_{i,j} = \text{length of the longest increasing path that ends in cell } (i,j)$$

The dp of the matrix M from above is:

$$\begin{pmatrix} 1 & 2 & 1 & 2 & 1 & 2 \\ 2 & 1 & 2 & 3 & 4 & 5 \\ 3 & 2 & 1 & 4 & 5 & 6 \\ 4 & 3 & 2 & 1 & 2 & 7 \end{pmatrix}$$

Figure 13: Matrix dp of matrix M

a. Write the recursive definition of $dp_{i,j}$.

Let the given matrix be M. We can arrive into the cell (i,j) only from the cell above it — $(i-1,j)$ — or from the cell to the left of it — $(i,j-1)$. A neighbouring cell can only contribute to $dp_{i,j}$ if the value of the neighbour is smaller than the value of the cell (i,j).

$$dp_{i,j} = \begin{cases} \max(1, 1+dp_{i-1,j}, 1+dp_{i,j-1}) & \text{if } i,j > 1, M[i][j] > M[i-1][j], M[i][j] > M[i][j-1] \\ 1+dp_{i-1,j} & \text{else if } i > 1, M[i][j] > M[i-1][j] \\ 1+dp_{i,j-1} & \text{else if } j > 1, M[i][j] > M[i][j-1] \\ 1 & \text{else} \end{cases} \tag{2}$$

b. Write the function `int longest_path(int x, int y, int M[x][y])` which returns the length of the longest increasing path in an $x \times y$ matrix M.

Solution: see code.

c. We extend the definition of a path to allow neighbours from all four directions: from a cell (i_k, j_k) we can now go directly below, above, to the right or to the left. For the matrix above, the longest increasing path now is:

$$\begin{pmatrix} 1 & 7 & 3 & 10 & 6 & 18 \\ 3 & 2 & 5 & 6 & 9 & 16 \\ 6 & 3 & 2 & 10 & 12 & 13 \\ 8 & 7 & 5 & 4 & 8 & 15 \end{pmatrix}$$

Figure 14: New longest path

This new problem can be solved with the same *dp* formulation and a similar recursive definition from above, but the cells have to be visited in a different order.

$$\begin{pmatrix} 1 & 2 & 1 & 4 & 1 & 8 \\ 2 & 1 & 2 & 3 & 4 & 7 \\ 3 & 2 & 1 & 4 & 5 & 6 \\ 4 & 3 & 2 & 1 & 2 & 7 \end{pmatrix}$$

Figure 15: Matrix *dp* showing the new longest path

i. For the given matrix N:

$$\begin{pmatrix} 6 & 5 & 2 & 3 \\ 7 & 14 & 15 & 4 \\ 3 & 12 & 10 & 8 \\ 2 & 7 & 9 & 1 \end{pmatrix}$$

Figure 16: Matrix N

fill in its corresponding *dp* matrix. As a hint, some entries have already been filled in.

$$\begin{pmatrix} 3 & 2 & 1 & 2 \\ & & & 3 \\ 2 & & & \\ 1 & & & 1 \end{pmatrix}$$

Figure 17: *dp* of matrix N

The trick is to iterate through the cells of the matrix in increasing order of the values stored in the cells.

$$\begin{pmatrix} 3 & 2 & 1 & 2 \\ 4 & 7 & 8 & 3 \\ 2 & 6 & 5 & 4 \\ 1 & 2 & 3 & 1 \end{pmatrix}$$

ii. Determine the new recursive definition of $dp_{i,j}$.

$dp_{i,j} = \max(dp_{x,y})$), where (x, y) is a neighbouring cell of (i, j) from the set $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, $(i, j+1)$ and $M[x][y] < M[i][j]$; or $dp_{i,j} = 1$ if no such neighbours exist.

iii. Write the function `int longest_path_all_directions(int x, int y, int M[x][y])` which returns the length of the new longest increasing path in an $x \times y$ matrix M.

Visit the cells of matrix M in ascending order by $M[i][j]$. This can be done the following way: create `structs` of the form `{cell_value, i_coordinate, j_coordinate}` for each cell in matrix M and sort this using a sorting function from the lecture or `qsort()` from the C standard library.

Task 3 [Medium/Hard]

You are given a set of n items, each with a weight and a value, and a fixed number W. We can represent these weights and values as arrays w and v, where w_i and v_i are the weight and the value of the i^{th} item, respectively.

The task is to determine which items to select such that their total weight is $\leq W$ and the total value is as big as possible.

For example, for $W = 20$ and the following $n = 5$ items:

item	w	v
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

the largest total value is 26 (choose items 1, 2, 3 and 5).

This problem can be solved using dynamic programming.

a. Define the dynamic programming matrix $dp_{i,j}$ in terms of:

- i - the number of considered items (how many items have we taken into account so far?)
- j - the total weight of the items chosen from those considered items (what is the weight of the items that we chose, out of the items we considered so far?).

What is the purpose of $dp_{i,j}$? What are we trying to maximize?

$dp_{i,j}$ = the maximum value that we can get if we select a subset of the first i items and the selected items have a total weight of exactly j.

Notice that this is a $n \times W$ matrix (there is no reason to store weights $> W$ because they are never a valid answer).

b. If we know all the entries of $dp_{i,j}$, how do we calculate the final answer for our problem?

The final answer is $\max(dp_{n,j})$ where $j \leq W$. It is not enough to look only at $dp_{n,W}$ because we may have an optimal combination with total weight $< W$ like $W - 1$.

c. Write a recursive formulation for $dp_{i,j}$.

If we look at the first i items with weight j, we have two options for item i:

- We take item i, and we get the value of $v_i + dp_{i-1,j-w_i}$.
- We do not take item i; the value is the same as if we ignore i: $dp_{i-1,j}$.

$$dp_{i,j} = \begin{cases} 0 & \text{for } i = 0, j = 0 \\ dp_{i-1,j} & \text{for } i \geq 1, j < w_i \\ \max(dp_{i-1,j}, v_i + dp_{i-1,j-w_i}) & \text{for } i \geq 1, j \geq w_i \\ -\infty & \text{for all other cases.} \end{cases} \tag{3}$$

We assign $-\infty$ to the places in the dp matrix where $dp_{i,j}$ indicates an invalid subproblem (like looking only at the first 2 objects but having an impossibly large total weight).

d. Write a function `int max_value(int n, int w[], int v[], int W)` which returns the answer for the problem (maximum total value of chosen items).

Solution: see code.

e. The current solution uses $\Theta(nW)$ memory. Without changing the recursive definition, reduce the space requirements of dp to $\Theta(W)$. Look at the recursive definition to see which subproblems are no longer needed and can be "forgotten".

Adapt the function `int max_value(int n, int w[], int v[], int W)` to use this memory optimization.

Notice in the dp relation that for row i we only use row $i-1$. In the answer, we only look at the last calculated row: row n. This means that we can only store the current row i and the previous row $i-1$ when calculating the DP matrix.

Note: This problem is also known as "0/1 Knapsack".

Task 4 [Medium/Hard]

You are given an sequence a of size n: $a_1, a_2, ..., a_n$. We do on this array n erasing operations. An erasing operation remove one of the numbers from the beginning or the end of the sequence. So, for the first operation, we erase either a_1 or a_n. If we are at the i^{th} operation and erase the element a_k, then the cost of that operation is $i \cdot a_k$.

Determine the maximum cost of doing n operations on a sequence a.

Example - for the sequence a below:

1	2	3	4
3	2	4	1

we have the maximum cost of 29, obtained this way:

1. In the first step erase 1 (with cost $1 \cdot 1 = 1$)
2. In the second step erase 3 (cost $2 \cdot 3 = 6$)
3. In the third step erase 2 (cost $3 \cdot 2 = 6$)
4. In the fourth step erase 4 (cost $4 \cdot 4 = 16$)
5. The total cost is $1 + 6 + 6 + 16 = 29$.

a. Because we are always erasing from the beginning or the end of a, at the end of every operation we are left with a contiguous sequence (subarray) of a. In other words, we are left with some subarray $a_i, a_{i+1}, ..., a_j$.

Define a dynamic programming approach with respect to the subarray $a_i, a_{i+1}, ..., a_j$ that remains after some operations (What defines this subsequence? What are we trying to maximize?).

Define the DP matrix according to the subarray $a_i, a_{i+1}, ..., a_j$ that remains after a number of erase operations:

$dp_{i,j}$ = the maximum cost that we can get out of operations, if the still undeleted numbers are the subarray $a_i, a_{i+1}, ..., a_j$.

b. Write the recursive relation of the DP approach. In what order do we have to solve the subproblems? What is the final answer to our problem?

Hint: The order is similar to the one used for the matrix multiplication problem.

We start from the initial array $(a_1, a_2, ..., a_n)$ and end up after $n-1$ operations with a single value: a_i. For each operation we decrease the length of the remaining array by 1, so it would make sense to iterate through the DP subproblems starting from the largest length (length $= n$ for the initial arraw) and going down to the smallest length (length $= 1$ for a_i).

So we can look at $dp_{i,j}$ the following way: to obtain $a_i, a_{i+1}, ..., a_j$, we can either erase a_{i-1} from $a_{i-1}, a_i, a_{i+1}, ..., a_j$ or erase a_{j+1} from $a_i, a_{i+1}, ..., a_j, a_{j+1}$. Thus the DP relation is:

$$dp_{i,j} = \begin{cases} \max(\text{cost of erasing } a_{i-1} + dp_{i-1,j}, \text{cost of erasing } a_{j+1} + dp_{i,j+1}) & \text{for } i > 1, j < n \\ \text{cost of erasing } a_{i-1} + dp_{i-1,j} & \text{for } i > 1, j = n \\ \text{cost of erasing } a_{j+1} + dp_{i,j+1} & \text{for } i = 1, j < n \\ 0 & \text{for } i = 1, j = n \end{cases} \tag{4}$$

How to calculate the cost of erasing? Notice that, to obtain $a_i, a_{i+1}, ..., a_j$, we must erase $i-1$ numbers from the left and $n-j$ numbers from the right. That means erasing $n-j+i-1$ numbers. Since the last operation out of these is erasing a number adjacent to $a_i, a_{i+1}, ..., a_j$, the costs of erasing are:

$$dp_{i,j} = \begin{cases} \max((n-j+i-1)a_{i-1} + dp_{i-1,j}, (n-j+i-1)a_{j+1} + dp_{i,j+1}) & \text{for } i > 1, j < n \\ (n-j+i-1)a_{i-1} + dp_{i-1,j} & \text{for } i > 1, j = n \\ (n-j+i-1)a_{j+1} + dp_{i,j+1} & \text{for } i = 1, j < n \\ 0 & \text{for } i = 1, j = n \end{cases} \tag{5}$$

For the final answer, look at each entry $dp_{i,i}$ where we are only left with one element a_i: this is the cost for the first $n-1$ operations. For each a_i, simulate the n^{th} erase.

c. Implement a function `max_cost(int n, int a[])` that receives an array a of length n and returns the maximum total cost after doing all n operations.

Solution: see code.