Math 416

Worksheet 19. Dynamic Programming II

Dynamic Programming principle. To solve problems, we solve subproblems. There is an ordering (implicit DAG) on the subproblems and a relation that can be used to solve a subproblem given solutions to previous (in the ordering) subproblems.

Longest increasing subsequences The input: a sequence of n numbers $a_1, a_2, \ldots, a_n$. Our goal is to find the longest increasing subsequence, i.e., the longest sequence $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$ for $1 \le i_1 < i_2 < \cdots < i_k \le n$.

Problem 1. Find the longest increasing subsequence in the following.

8 4 12 2 10 6 14 1 9

Problem 2. You might try picking the least term and then the least term to the right of that and so on. Or pick the greatest term and then the next-greatest to the left of that one and so on. Why do these ideas not work? (Give examples.) Why do they naturally lead to a dynamic-programming method?

Problem 3. Given a sequence $(a_1, \ldots, a_n)$, build a directed graph with vertices $1, 2, \ldots, n$. Say that (i, j) is an edge iff $i < j$ and $a_i < a_j$.

(a) Why is this graph acyclic?

(b) Paths in the graph correspond to [ ] subsequences.

(c) What is the graph corresponding to the sequence in Problem 1?

(d) Write $L[j]$ for the length of the longest increasing subsequence ending in a_j. How does $L[j]$ relate to $L[i]$ for other i?

Use this relation to write a dynamic-programming algorithm that solves this problem.

Problem 4. Run the algorithm to find the longest increasing subsequence of the sequence below.

5 2 8 6 3 6 9 7

Again, you can recover the subsequence itself using backpointing. The algorithm runs in $O(n^2)$ time in the worst case.

Problem 5. Why does this longest-increasing-subsequence algorithm run in $O(n^2)$ time, in the worst case?

Edit distance Given two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$, we want to measure how well they can be aligned (e.g. for a spellcheck).

The **edit distance** of two strings is the minimum number of **edits** — insertions, deletions, and substitutions of characters — needed to transform one string into the other.

Problem 6. An **alignment** of two strings is a way of writing the strings one on top of the other, possibly with the additional placeholder character '-' inserted, so that they have the same length. The **cost** of an alignment is the number of **mismatches**, i.e., columns in which the characters are different. (See the next example.)

Convince yourself that the edit distance of two strings equals the minimum cost of an alignment of the two strings.

Example. `ocurrance` versus `occurrence`:

`o-currance`	`o-curr-ance`
`occurrence`	`occurre-nce`

The comparison on the left has 1 gap and 1 mismatch, so its cost is $1 + 1 = 2$. The comparison on the right has 3 gaps, giving a cost of 3.

To solve this problem using dynamic programming, we need to choose subproblems and see how a subproblem can be solved by assembling solutions to smaller subproblems.

The right subproblems to consider for this problem are bi-initial segments (or *prefixes*):

$$x_1\, x_2 \cdots x_i \quad i \le m,$$
$$y_1\, y_2 \cdots y_j \quad j \le n.$$

Let $E(i, j)$ be the edit distance of the initial segments $x_1 \cdots x_i$ and $y_1 \cdots y_j$. We want $E(m, n)$.

Problem 7 (relation between subproblems)**.** When aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$, three things can happen at the right end:

$$\begin{matrix} x_i \\ \text{-} \end{matrix}, \quad \begin{matrix} \text{-} \\ y_j \end{matrix}, \quad \text{or} \quad \begin{matrix} x_i \\ y_j \end{matrix}.$$

Fill in the blanks.

if we have $\begin{matrix} x_i \\ \text{-} \end{matrix}$ **on the right then:** $E(i, j) =$ ☐.

if we have $\begin{matrix} \text{-} \\ y_j \end{matrix}$ **on the right then:** $E(i, j) =$ ☐.

if we have $\begin{matrix} x_i \\ y_j \end{matrix}$ **on the right then:** $E(i, j) =$ ☐, where $\Delta(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \ne b \end{cases}$.

So we have the crucial relation between subproblems.

$$E(i, j) = \min\left(\square, \square, \square\right).$$

The idea of the algorithm is this. We store $E(i, j)$ in an $(m + 1) \times (n + 1)$ table; before computing $E(i, j)$ we need to have computed $E(i - 1, j)$, $E(i, j - 1)$, and $E(i - 1, j - 1)$. The base cases are $E(i, 0) = i$ and $E(0, j) = j$.

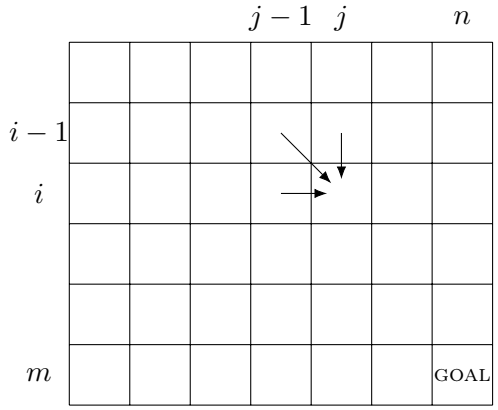

FIGURE 1. The table of subproblems $E(i, j)$. Entries $E(i - 1, j)$, $E(i, j - 1)$, and $E(i - 1, j - 1)$ are needed to fill in $E(i, j)$.

Problem 8. Use the edit-distance algorithm to compute the edit distances between the following pairs of strings (or at least, say, two of the pairs):

ALGORITHM ALGOREISM	MATH MOUTH	MISPPEEL MISSPELL

Problem 9. Write the algorithm in pseudocode and analyze its running time.

Problem 10. How should our algorithm be modified to give the optimal alignment, in addition to the edit distance?

Problem 11. Every dynamic-programming problem has an underlying (weighted) DAG structure on the subproblems. What DAG models the edit-distance problem? What does a solution correspond to in the DAG?

Problem 12. As we've presented the problem, the three possible operations —insertion, deletion, and substitution— each have the same cost in the computation of edit distance. Specific applications might call for different definitions of distance that e.g. assign insertions and deletions higher costs than substitutions. Briefly discuss how the algorithm should be modified to accommodate changes like this.

Knapsack problems We have n items to choose from; their values are $v_1, \ldots, v_n$ and their weights are $w_1, \ldots, w_n$. Our knapsack has a capacity of W. What's the most valuable combination of items we can fit in the knapsack, so that the weight doesn't exceed W?

We will study two variants of this question:

(a) with repetition (unlimited supply of each item)
(b) no repetition (there is only one of each item)

Example. Say $W = 6$ and we have three items:

$$\begin{array}{lll} w_1 = 2 & w_2 = 4 & w_3 = 3 \\ v_1 = 5 & v_2 = 10 & v_3 = 8 \end{array}$$

The optimal solution with repetition is 3 3 (i.e., select two of item 3).

The optimal solution without repetition is $\boxed{}$.

Knapsack with repetition

Problem 13. A first attempt at solving the knapsack problem might be to greedily add items that offer the most value per unit of weight; i.e., add items maximizing v_i/w_i as long as this is possible. Does this work? Why or why not?

We consider these subproblems: for $w \leq W$ let $K(w)$ be the max value of a knapsack of capacity w.

This is the crucial relation between subproblems:

$$K(w) = \boxed{}.$$

Problem 14. Write the algorithm in pseudocode.

Problem 15. Explain why this algorithm runs in $O(nW)$ time. Is this efficient?

Problem 16. What DAG models the knapsack problem? What does a solution correspond to in the DAG?

Knapsack without repetition Now we don't allow repetitions; each item can be used only once.

Problem 17. Explain why the previous approach won't work if repetitions aren't allowed.

So we need to divide into subproblems more carefully. For $w \leq W$ and $j \leq n$ let $K(w, j) =$ the maximum value achievable using a knapsack of capacity w and items $1, \ldots, j$. So we seek $K(W, n)$.

Problem 18. Either item j is needed, or it isn't. This gives the following crucial relation between subproblems.

$$K(w, j) = \boxed{}.$$

Problem 19. Now write the algorithm in pseudocode.

Problem 20. What is the worst-case running time of this algorithm?