Lecture 4: Prefix/Interval Dynamic Programming

1 Philosophy

There are many problems which ask to understand some property of a sequence, or are about applying a sequence of operations to a sequence. These naturally lend themselves to dynamic programming based on subintervals. Sometimes we can simplify this structure further and only focus on prefixes of the sequence. In this lecture we will understand how to identify these structures and design dynamic programming algorithms for them.

Objectives of this lecture

In this lecture we will study dynamic programs on sequences, which natural introduces subproblems which are *prefixes* and *subintervals*. In these problems, it is often important to think carefully about exactly which subproblems are necessary in order to obtain an efficient algorithm. Example problems include:

- Longest increasing subsequence (LIS),
- Chain matrix multiplication,
- Parsing a context-free-grammar (CFG).

2 Example Problems

2.1 Longest Increasing Subsequence

Our first problem is the "longest increasing subsequence" (LIS) problem, which has an $O(n^2)$ solution.

Problem: Longest Increasing Subsequence

Given a sequence of comparable elements $a_1, a_2, \ldots, a_n$, an increasing subsequence is a subsequence $a_{i_1}, a_{i_2}, ..., a_{i_{k-1}}, a_{i_k}$ $(i_1 < i_2 < \ldots i_k)$ such that

$$a_{i_1} < a_{i_2} < \ldots < a_{i_{k-1}} < a_{i_k}.$$

A longest increasing subsequence is an increasing subsequence such that no other increasing subsequence is longer.

Find some optimal substructure Given a sequence $a_1, \ldots, a_n$ and its LIS $a_{i_1}, \ldots, a_{i_{k-1}}, a_{i_k}$, what can we say about $a_{i_1}, \ldots, a_{i_k-1}$? Since $a_{i_1}, \ldots, a_{i_k}$ is an LIS, it must be the case that $a_{i_1}, \ldots, a_{i_{k-1}}$ is an LIS of $a_1, \ldots a_{i_k}$ such that $a_{i_{k-1}} < a_{i_k}$. Alternatively, it is also an LIS that ends at (and contains) $a_{i_{k-1}}$. This suggests a set of subproblems.

Define our subproblems Let's define our subproblems to be

$$\text{LIS}[i] = \text{the length of a longest increasing subsequence of } a_1, \ldots a_i \text{ that contains } a_i$$

Note that the answer to the original problem **is not** necessarily LIS$[n]$ since the answer might not contain a_n, so the actual answer is

$$\text{answer} = \max_{1 \le i \le n} \text{LIS}[i]$$

Deriving a recurrence Since LIS$[i]$ ends a subsequence with element i, the previous element must be anything a_j before i such that $a_j < a_i$, so we can try all possibilities and take the best one

$$\text{LIS}[i] = \begin{cases} 0 & \text{if } i = 0, \\ 1 + \max\limits_{\substack{0 \le j < i \\ a_j < a_i}} LIS[j] & \text{otherwise.} \end{cases}$$

Analysis We have $O(n)$ subproblems and each one takes $O(n)$ time to evaluate, so we can evaluate this DP in $O(n^2)$ time.

Optimizing the runtime: better data structures. This runtime can actually be improved to $O(n \log n)$ using data structures. We will revisit this in a few lectures when we introduce *Segment Trees*.

2.2 Chain Matrix Multiplication

Our first example of interval DP is the "chain matrix multiplication" problem.

Problem: Chain Matrix Multiplication

We define an *operation* which can be performed to a sequence $(b_1, \ldots, b_m)$ of positive integers. Choose an index $i \in \{2, \ldots, m-1\}$ and delete b_i from the sequence, reducing its length to $m-1$, and incur cost $b_{i-1} b_i b_{i+1}$.

The input is a sequence of positive integers $(a_1, a_2, \ldots, a_n)$. What is the minimum cost way to apply $n-2$ of the above operations to the sequence (thus ending with the sequence (a_1, a_n))?

Example. Let $(a_1, a_2, a_3, a_4) = (10, 30, 5, 60)$. There are two possible ways to do two operations. The first way is to first delete a_2 and then a_3. The cost of this is:

$$10 \times 30 \times 5 + 10 \times 5 \times 60 = 4500.$$

The other way, first deleting a_3 and then a_2, has cost:

$$30 \times 5 \times 60 + 10 \times 30 \times 60 = 27000.$$

Thus the answer would be 4500.

Explaining the name. Consider a chain of matrices being multiplied: $A_1 A_2 \dots A_{n-1}$ where A_i has dimensions $a_i \times a_{i+1}$. Because matrix multiplication is associative we can do the multiplications in any order. The runtime cost of multiplying an $a \times b$ times $b \times c$ matrix is about $a\,b\,c$. So the chain matrix multiplication is asking for the cheapest way to multiply a sequence of matrices of given dimensions.

2.2.1 Brute-force-approach

The number of possible sequence of operations is $(n-2) \times (n-3) \times \cdots \times 1 = (n-2)!$. This is exponentially large. You might notice that if you for example do an operation to a_i and then to a_j where i and j are separated, then these operations commute (i.e., immediately have the same cost). It turns out that while this saves significantly runtime, the cost is still exponentially large (about 4^n, we will not discuss why in lecture).

2.2.2 $O(n^3)$ time algorithm using intervals

In this style of problem where you are collapsing a string using local operations, it is often useful to think about the very last operation performed in the sequence (the $(n-2)$-th operation). Starting with a sequence $(a_1, \dots, a_n)$, right before the last operation our sequence takes the form (a_1, a_i, a_n) for some index $i \in \{2, \dots, n-1\}$.

How did we reach this state? It must have been through applying operations to the subsequence $(a_1, \dots, a_i)$ and $(a_i, \dots, a_n)$ until they are down to length 2. Applying the same logic to these subproblems, you can see that subintervals are exactly the right structure to use as our dynamic programming states.

Here is the DP formula table where $\text{cost}[i, j]$ represents the minimum cost way of applying $j - i - 1$ operations to the subinterval $(a_i, \dots, a_j)$, reducing it to length 2. For $i < j$:

$$\text{cost}[i, j] = \begin{cases} 0 & \text{if } j = i + 1, \\ \min_{k \in \{i+1, \dots, j-1\}} \left(a_i a_k a_j + \text{cost}[i, k] + \text{cost}[k, j] \right) & \text{otherwise.} \end{cases}$$

The answer is $\text{cost}[1, n]$.

Extra Remarks. This problem actually admits an $O(n \log n)$ time algorithm by Chin-Hu-Shing (1981) which I have not personally tried to understand.

2.3 Parsing a context-free-grammar

Our final example is parsing a *context-free-grammar*. We first state the problem abstractly, and then explain how it arises. For notational clarity, we will try to make strings capital letter variable names, and characters as lowercase letter variable names.

Problem: Parsing a CFG

In this problems, all characters and strings consist of only lowercase English letters. The input consists of the following:

- An integer $g > 0$, character c, and a string S.
- For $i = 1, \ldots, g$, let $c_i \to S_i$ be *substitution rules* where c_i is a single character, and S_i is a string of length at least 2.

Decide whether applying the substitution rules starting with c can form string S.

Our goal will be to simply give a polynomial time algorithm for the problem. More precisely, the runtime will be $O(n^4 \cdot |\mathcal{G}|)$, where $n = |S|$ is the length of S, and $|\mathcal{G}|$ is defined as the total size of the strings in the substitution rules:

$$|\mathcal{G}| = \sum_{i=1}^{g} |S_i|.$$

Example. Let $g = 3$, $(c, S) = (\text{`}a', \text{``}a\,b\,a\,c'')$, and consider the substitution rules:

- $(c_1, S_1) = (\text{`}a', \text{``}b\,c'')$, and $(c_2, S_2) = (\text{`}b', \text{``}c\,a'')$, and $(c_3, S_3) = (\text{`}c', \text{``}a\,b'')$.
- In other words, we can substitute $\text{`}a' \to \text{``}b\,c''$ or $\text{`}b' \to \text{``}c\,a''$ or $\text{`}c' \to \text{``}a\,b''$.

The answer to this instance is YES, because the following sequence of substitutions works:

$$\text{`}a' \to \text{`}b\,c' \to \text{`}c\,a\,c' \to \text{`}a\,b\,a\,c'.$$

Explaining the name. Informally, when forming syntax/sentences, there are many natural substitutions one can use to create new valid sentences. For example, nouns can be replaced with adjective+noun, etc. In the problem statement we abstracted away single parts of speech as just a character.

2.3.1 Trying to design an algorithm forwards

Again, a natural *brute force* algorithm is to just start at c and try to generate all possible strings of length $|S|$ using the rules, finally checking if the input string is generated. Even in the above example, you can see that we can generate an exponentially large number of strings with length at most n. Again, we need a different approach.

2.3.2 Going backwards: intervals

Just like the example of chain matrix multiplication, it is instead useful to "think backwards". Instead of thinking of the input as substitution rules, think of them as collapsing rules. In other words, we can take the string S_i and replace it with a single character c_i. Our goal is to end with the single character c. Now, think of the final collapsing operation. It must have taken the form $S_i \to c_i = c$. So we must have found a way to collapse our initial string S to s_i somehow. Writing this more explicitly, let $S_i = z_1 z_2 \dots z_m$ where z_j are characters. Then there must be a way to partition s into substrings $S = T_1 T_2 \dots T_m$ (where T_i are *strings*) and then each T_i can be collapsed to z_i using some sequence of operations.

$$\begin{aligned} S &= T_1 T_2 \dots T_m \\ T_1 &\to z_1 \quad \text{using some operations.} \\ T_2 &\to z_2 \quad \text{using some operations.} \\ &\vdots \\ T_m &\to z_m \quad \text{using some operations.} \\ z_1 \dots z_m = S_i &\to c_i = c \quad \text{using a single operation.} \end{aligned}$$

This suggests a natural DP state: for the substring $S[l, r] \coloneqq S[l] \dots S[r]$, store the possible single characters it can be collapsed to after a sequence of operations, say as a lookup table. We now discuss how to fill in the DP table.

Base case. We start with the base case. If $l = r$, the set of possible characters is simply S_l itself.

The case $l < r$. We want to find what $S[l] \dots S[r]$ can be collapsed to. We iterate over all possible final operations, i.e., $i \in \{1, \dots, g\}$, and decide whether the final operation could be $S_i \to c_i$. If yes, add c_i to the set of possible single characters for the substring $S[l, r]$.

For this, we do a *prefix DP*. The DP states are: for a prefix of the substring $S[l] \dots S[r]$ and a prefix of S_i, is it possible to collapse one to the other? Each state can be computed in $O(n)$ time by adding a single letter (as the prefix of S_i). There are $O(n \times |S_i|)$ states in this prefix DP, and each can be processed in time $O(n)$, for a total runtime of $O(n^2 |S_i|)$.

Runtime analysis. Let $l < r$ and $i \in \{1, \dots, g\}$. The runtime cost of deciding whether the final operation could have been $S_i \to c_i$ is $O(n^2 |S_i|)$. Thus the total runtime cost is $O(n^2 \cdot \sum_{i=1}^{g} |S_i|) = O(n|\mathcal{G}|)$. There are $O(n^2)$ intervals, so the total runtime cost is $O(n^4 \cdot |\mathcal{G}|)$.

3 Extras (Not Required)

Problem: Returning the sequence in LCS

Give an algorithm that returns the elements in the longest increasing subsequence in $O(n^2)$ time.

Problem: Improving runtime of CFG Parsing

Improve the runtime of the CFG algorithm to $O(n^3 \cdot |\mathcal{G}|)$.

Hint: Your DP states should be an interval/substring $S[l, r]$ and a prefix of one of the input strings S_i. Thus, the total number of DP states is $O(n^2 \sum_{i=1}^{g} |S_i|) = O(n^2 \cdot |\mathcal{G}|)$.

Alternate approach: Here is an alternate approach which is more or less equivalent to the hint, but might be conceptually easier for some to think about. The first step is to just study the case where each S_i is length two: in this case, it it much easier to get the $O(n^3|\mathcal{G}|)$ runtime. The second step is to reduce to this case. Formally, given a CFG, design an equivalent CFG where all S_i are length two (possibly by introducing new letters), whose total size is within a constant factor of the original size. Building this equivalent CFG is called putting it into *Chomsky normal form* (essentially).