CS 38 Introduction to Algorithms **Week 2 Recitation Notes**

Dynamic Programming

TA: Ethan Pronovost (epronovo@caltech.edu), Chinmay Nirkhe (chinmay@caltech.edu)

1 Key Ideas

Principal Properties There are two key characteristics of problems where dynamic programming can be applied.

1. Optimal substructure: we can (easily) relate the solution to a given problem to solutions of related subproblems. Here "subproblem" is defined over some ordering (which we must define as well).
2. Overlapping subproblems: in recursively solving a problem, we repeatedly face the same subproblem instances. By saving the solutions to these subproblems, we can save a lot of computing time.

Solution Format Typically, we store the solutions to subproblems in a (multi-)dimensional table. Certain base cases are stored upon initialization, and then we fill out the rest of the table iteratively. The several important steps in a dynamic programming problem are:

1. Define the problem in a way that lends itself to relatable subproblems.
2. Relate an optimal solution to optimal sub-solutions (i.e. the recurrence relation).
3. Cover base cases.
4. Use a systematic approach to iteratively fill out the table.

Proof Strategy There are two key parts to a proof of correctness for a dynamic programming problem. First, you must prove the base cases hold. This is often rather trivial (e.g. 1D clustering with only one cluster).

Second, you must show that the recurrence relation correctly relates an optimal solution to the solutions of subproblems. Rarely, this will be as straight forward as the Fibonacci case. More commonly, we will use an "optimality by exhaustion" approach: we consider every possible subproblem solution, and then take the optimal one. Algorithmically, this typically takes for form of a maximization or minimization over some subset of subproblems. For the proof, you must show that you correctly compute the result of extending a given subproblem, and that you consider all relevant subproblems.

Runtime The runtime analysis of dynamic programs is often rather straight forward. Generally, the algorithmic complexity takes for form

$$\text{Runtime} = (\text{Total number of subproblems}) \times \left(\begin{array}{c} \text{Time it takes to solve problems} \\ \text{given solutions to subproblems.} \end{array} \right)$$

Memoization and Space-Time Tradeoff You may remember from CS 2 the concept of *memoization*. In a normal *bottom-up* approach, we fill out the entire table, and then use that to compute the solution to the initial problem. This assumes that we must compute all subproblems. However, this is not always the case. If the recurrence relation is not as linear (e.g. a graph), we may not need to compute all the subproblems first.

Memoization essentially starts with the initial problem and works down to subproblems. For each subproblem, we check to see if we've already solved it. If we haven't, we recursively compute it, and store its value for later reuse. This adds the overhead of checking to see if the desired subproblem is in the dictionary of computed solutions. Generally, the complexity of both bottom-up and memorization are the same. Furthermore, the bottom-up approach is a lot easier to construct and reason about.

In general, dynamic programming is an example in the tradeoff between time and space complexity. Without storing the subproblem solutions, the recurrence relation would yield a potentially exponential number of subproblem. By sacrificing some space, we can drastically reduce this complexity to some polynomial time.

Answering the Question Dynamic programming problems typically ask for one of two things as a solution: the optimal value, or the optimal solution. For example, in the 1D k-clustering example, we could conceivably be asked to either compute the minimum clustering radius, or the actual cluster centers that yield this minimum. Generally, we are optimizing over the value, so will necessarily need to store this quantity. If, however, we need to return the full solution, then we must also store the way we've gotten to a given subproblem solution. In the 1D k-clustering case, this amounts to storing both the clustering radius and the set of points ($r[t, j]$ and $c[t, j]$ in the chapter notes). Be sure to return the quantity specified in the problem statement.

2 Box Stacking

Problem Given a collection of n 3D boxes of integer dimensions $\{b_1, \ldots, b_n\}$ where $b_i = (\ell_i, w_i, h_i)$, calculate the height of the tallest stack of boxes you can form under the following constraints: A box cannot be rotated[1] and a box b_i can only be stacked on b_j if the base of b_i fits completely within the base of b_j.

Algorithm Idea Sort all the boxes by base area ($\ell_i \times w_i$). Clearly a box of smaller base area cannot fit below a larger, so $j > i$ is a *necessary* condition for box j to be below box i in a valid stack.. Let $m(i)$ be the height of the tallest stack of boxes with box b_i at the top. Here is a recursive definition for $m(i)$:

$$m(i) = \max \left\{ h_i, \max_{\substack{j>i: \\ w_i \le w_j, \ell_i \le l_j}} \{m(j) + h_i\} \right\} \tag{1}$$

Why is this the definition? We consider all possible blocks b_j that could be directly below b_i in the stack and then maximize recursively to get the maximum stack height. Because of the sorting, no block b_j with $j < i$ can fit below block b_i.

Algorithm Description Sort the boxes by base area so that b_1 has the least base area. Create a table t of size n indexed by $1 \le i \le n$. Starting with $i = n$ to 1, define $t[i]$ by,

$$t[i] \leftarrow \max \left\{ h_i, \max_{\substack{j>i: \\ w_i \le w_j, \ell_i \le l_j}} \{t[j] + h_i\} \right\} \tag{2}$$

Return the following value:

$$\max \{t[1], \ldots, t[n]\} \tag{3}$$

[1] Additional exercise: solve this problem where rotation is allowed. You can still come up an algorithm of the same complexity.

Algorithm Correctness By the algorithm idea, $t[i] = m(i)$, the optimal height of a stack with box i on top. The tallest stack is clearly the one maximizied over all choice of top box.

Algorithm Complexity The space complexity is $O(n)$ as we only store table t. The algorithm complexity is $O(n \log n)$ for sorting followed by $O(n^2)$ computations as the calculation of each $t[i]$ is $O(n)$ given the previously solved subproblems and there are $O(n)$ subproblems. Total complexity: $O(n^2)$.

3 Longest Common Subsequence

Also available as CLRS Ex. 15-4.

Problem Given input strings $x = [x_1 \dots x_m]$ and $y = [y_1 \dots y_n]$, find the length of the longest common subsequence (LCS). A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous[2]. Ex. If $x = [abcdgh]$ and $y = [aedfhr]$ then the LCS is $[adh]$ of length 3.

Algorithm Idea Let $\ell(x, y)$ be the length of the LCS of x and y. Here is a recursive definition of $\ell(x, y)$ for $m, n > 1$:

$$\ell(x_1 \dots x_m, y_1 \dots y_n) = \begin{cases} 1 + \ell(x_1 \dots x_{m-1}, y_1 \dots y_{n-1}) & \text{if } x_m = y_n \\ \max\{\ell(x_1 \dots x_{m-1}, y_1 \dots y_n), \ell(x_1 \dots x_m, y_1 \dots y_{n-1})\} & \text{o.w.} \end{cases} \tag{4}$$

Why is this the definition? If the last characters of x and y match, then the are necessarily in the LCS, so recursively check the substring excluding them. If the last characters don't match then, then at least one of them isn't in the LCS, so check both cases recursively and choose the maximum.

Algorithm Description Generate a table t of size mn indexed by tuples (i, j) for $1 \le i \le m$ and $1 \le j \le n$. Set $t[1, 1] \leftarrow 1$ if $x_1 = y_1$ and $t[1, 1] \leftarrow 0$ otherwise. Systematically by increasing values of $i + j$, apply the following recursive definition to calculate all $t[i, j]$:

$$t[i][j] \leftarrow \begin{cases} 1 + t[i-1, j-1] & \text{if } x_i = y_j \\ \max\{t[i-1, j], t[i, j-1]\} & \text{o.w.} \end{cases} \tag{5}$$

Here we assume $t[i, j] = 0$ if $i < 1$ or $j < 1$. Return $t[m, n]$.

Algorithm Correctness As an exercise, convince yourself that the algorithm idea above is correct. Then, as $\ell(x_1, y_1) = t[1, 1]$ and the similarity of (4) and (5), it follows that $t[i, j] = \ell(x_1 \dots x_i, y_1 \dots y_j)$ for $1 \le i \le m$ and $1 \le j \le n$. Then returning $t[m, n]$ yields the desired result for the full strings.

Algorithm Complexity The space complexity is $O(mn)$ as we only store t. The time complexity is $O(mn)$ as well because the calculation of each $t[i, j]$ is $O(1)$ given the subproblems have been solved and there are $O(mn)$ subproblems.

Extension Given an $O(n^2)$ time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

Algorithm Description Sorting the sequence. Return the LCS of the sorted sequence and the original sequence.

[2]Additional exercise: consider the related problem of finding the longest common substring (here we require that the string be contiguous). You can build a related dynamic problem algorithm that is of the same complexity, but a smarter algorithm can be built using a generalized suffix tree to solve this in $O(n + m)$.

Algorithm Correctness The sorted sequence is by definition monotonically increasing so any subsequence has the same property. Therefore, the longest common subsequence with the original sequence is, by construction, (a) the longest and (b) monotonically increasing.

Algorithm Complexity Sorting is an $O(n \log n)$ algorithm and LCS as shown above is $O(n^2)$ as the strings have the same length. Total complexity: $O(n^2)$.

4 Possible Scores

Problem Consider a game where a player can score either $p_1, p_2, \ldots, p_m$ points in a move. Given a total score n, find number of ways to reach the total (order dependent).

Algorithm Description Generate a table t of size $n+1$ indexed $0 \ldots n$. For base case, set $t[0] \leftarrow 1$. Then for $i = 1, ..., n$, set

$$t[i] \leftarrow \sum_{j=1}^{m} t[i - p_j] \tag{6}$$

where we use the notation $t[k] = 0$ for $k < 0$. Return $t[n]$.

Algorithm Correctness We prove by induction. For $n = 0$ (base case), there is a unique way to reach the total: no scores were made. Assume for induction that correctness holds for all $n' < n$. Consider a score sequence $a_1, \ldots, a_k$ that sums to n. Then necessarily $a_1, \ldots, a_{k-1}$ sums to $n - a_k$. By the induction hypothesis, the number of ways to total $n - a_k$ is equal to $t[n - a_k]$. Therefore, by considering all possible a_k (i.e. the final score in the sequence), we count all ways to total n. The choices for a_k are precisely, $p_1, \ldots, p_m$, which the algorithm considers.

Algorithm Complexity The space complexity is $O(n)$ as we store $(n + 1)$ elements each requiring $O(1)$ space. The time complexity is $O(nm)$ as we calculate each $t[i]$ for $i = 0, \ldots, n$ and each calculating is the sum of m numbers. If m is small then the algorithm has complexity $O(n)$ which we will see next can be beat!

A Better Algorithm In the previous algorithm, we calculated each $t[i]$ from $i = 0, \ldots, n$. But this is tedious as there is a lot of repetition. Let's adjust this by using the repetition of squaring trick. Informally, this states that we can calculate x^n in time $O(\log n)$ by squaring until we reach the exponent. For more details, read the week 1 recitation notes.

For this part, we are **not** going to assume $p_1, \ldots, p_m, m$ or n are small numbers. Rather they can be potentially large, so our complexity will rely on this. However, we will assume that $p_1 \leq \ldots \leq p_m$. Recall the recurrence relation:

$$a_n = a_{n-p_1} + a_{n-p_2} + \ldots + a_{n-p_m} \tag{7}$$

This can be expressed in the equivalent matrix form:

$$\left(\begin{array}{c|c} 0 & \\ \vdots & \mathbb{I} \\ 0 & \\ \hline v & \end{array}\right) \cdot \begin{pmatrix} a_{n-p_m} \\ \vdots \\ a_{n-2} \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} a_{n-p_m+1} \\ \vdots \\ a_{n-1} \\ a_n \end{pmatrix} \tag{8}$$

where $\mathbb{I}$ is the identity matrix of size $p_m - 1$ and v is the $p_m \times 1$ vector that forms the linear equation in (7). Write

$$M = \left(\begin{array}{c|c} 0 & \\ \vdots & \mathbb{I} \\ 0 & \\ \hline v & \end{array}\right) \tag{9}$$

Then, by repeated application of (8),

$$M^{n-p_m} \cdot \begin{pmatrix} a_1 \\ \vdots \\ a_{p_m} \end{pmatrix} = \begin{pmatrix} a_{n-p_m+1} \\ \vdots \\ a_n \end{pmatrix} \tag{10}$$

By the repetition of squaring trick, we can calculate M^{n-p_m} in $O(\log n)$ matrix multiplications. We assume here that matrix multiplication can be performed in $O(k^\omega)$ time for a $k \times k$ matrix[3]. The result is $O({p_m}^\omega \log n)$ time algorithm for the matrix exponentiation. We also need to calculate the initial vector; apply the initial approach detailed above to get a $O(mp_m)$ runtime algorithm for calculating the initial vector. Lastly, to compute a_n we must calculate one row of this matrix multiplication, which is $O(p_m)$. Therefore the total complexity is

$$O\left(p_m + mp_m + {p_m}^\omega \log n\right) \tag{11}$$

If $p_1, \ldots, p_m$ and m are small, then the complexity of this algorithm is $O(\log n)$ which is a considerable improvement over the $O(n)$ algorithm previously presented!

5 Number of paths with $\leq \ell$ turns

Problem Given m, n, ℓ integers, count the number of paths of length exactly $m + n$ to reach the bottom right square from the top left square of a $m \times n$ matrix with at most ℓ turns.[4] One can only move horizontally or vertically and a turn is considered any rotation of motion by 90 degrees.

Algorithm Idea If the length of the path is exactly $m+n$ then you can only move right ($\mathcal{R}$) or down ($\mathcal{D}$).

Define $c(i, j, k, d)$ as the number of paths of paths to square (i, j) with k more turns makeable with the next movement in direction d (here $d = \mathcal{R}$ or $\mathcal{D}$). Here is a recursive definition for $c(i, j, k, d)$:

$$c(i,j,k,d) = \begin{cases} c(i-1,j,k,\mathcal{D}) + c(i,j-1,k-1,\mathcal{R}) & \text{if } d = \mathcal{D} \\ c(i-1,j,k-1,\mathcal{D}) + c(i,j-1,k,\mathcal{R}) & \text{if } d = \mathcal{R} \end{cases} \tag{12}$$

Why is this the definition? In order to get to square (i, j) you must have come from $(i - 1, j)$ moving $\mathcal{D}$ or $(i, j - 1)$ moving $\mathcal{R}$. Depending on the direction d that you end up after, it will cost you a turn if d disagrees with your direction of motion. Hence the $k - 1$, as you use a turn.

[3] The actual value of ω is a piece of great interest in theoretical computer science. There is a large class of problems that are shown to be as hard as matrix multiplication. A trivial bound for $\omega \leq 3$ using the standard matrix multiplication approach. Strassen's algorithm gives a bound for $\omega \leq \log_2 7 \approx 2.807$. The lower bound is known to be $\omega \geq 2$ because it is the minimal time required to access every element. It is an open problem if $\omega = 2$.

[4] Additional exercise: What happens when you remove this path length restriction? Can you still solve this problem using dynamic programming? If so come up with an algorithm of similar complexity.

Algorithm Description Generate a table t of size $2mn(\ell+1)$ indexed by tuples (i,j,k,d) where $1 \leq i \leq m, 1 \leq j \leq n, 0 \leq k \leq \ell$ and $d \in \{\mathcal{D}, \mathcal{R}\}$. For a base case, $t[1,1,\ell,d] = 1$ for either choice of d and $t[1,1,k,d] = 0$ for $k < \ell$. Systematically, apply the following definition to calculate all $t[i,j,k,d]$:

$$t[i,j,k,d] = \begin{cases} t[i-1,j,k,\mathcal{D}] + t[i,j-1,k-1,\mathcal{R}] & \text{if } d = \mathcal{D} \\ t[i-1,j,k-1,\mathcal{D}] + t[i,j-1,k,\mathcal{R}] & \text{if } d = \mathcal{R} \end{cases} \tag{13}$$

Here we assume $t[i,j,k,d] = 0$ if undefined. Then return the following sum

$$\sum_{\substack{0 \leq k \leq \ell \\ d \in \{\mathcal{D},\mathcal{R}\}}} t[m,n,k,d] \tag{14}$$

Algorithm Correctness As an exercise, convince yourself that the algorithm idea above is correct. Then, by the base case of the algorithm and the similarity of (12) and (13), its easy to see that $t[i,j,k,d] = c[i,j,k,d]$. Then, it is only a matter of returning the sum of any t element of either direction having any of $0,\ldots,\ell$ turns left to make for the cell (m,n).

Algorithm Complexity The space complexity is $O(mn\ell)$ as we only store t. The time complexity is $O(mn\ell)$ as well because the calculation of each element of t is $O(1)$ given the subproblems have been solved and there are $O(mn\ell)$ subproblems.[5]

[5] Additional exercise: Now that you are well acquainted with dynamic programming algorithms as well as the repetition of squaring trick, construct a $O(nm \log \ell)$ algorithm to solve this problem. Hint: The approach is very similar to that applied to the possible scores problem.