Illinois Institute of Technology
Department of Computer Science

Lectures 10–11: October 1–6, 2014

CS 430 Introduction to Algorithms
Fall Semester, 2014

Dynamic Programming

We now introduce a useful approach to algorithm design: *dynamic programming* (Chapter 15 of CLRS). The word "programming" here is used as in the phrase "television programming" (that is, meaning to set a schedule), not in the sense of "computer programming," which originally had that sense also. Dynamic programming is typically used on optimization problems, particularly optimization problems that exhibit *optimal substructure*: an optimal solution is composed of optimal solutions to smaller subproblems. In dynamic programming, we store the solutions to these subproblems in a table so we can avoid computing them multiple times.

Matrix-chain multiplication

In the problem of matrix-chain multiplication (section 15.2 of CLRS) we are given a sequence (or chain) of n matrices that we would like to multiply:

$$A_1 A_2 A_3 \cdots A_n$$

CLRS describes an algorithm for multipling two matrices, but in this case we are multiplying together many matrices, and have to do these multiplications two matrices at a time. Note that matrix multiplication is associative: that is, $A_1(A_2A_3) = (A_1A_2)A_3$. Thus, if we wanted to compute $A_1A_2A_3$, we could either first multiply A_1 and A_2 and then multiply the result by A_3 or we could first multiply A_2 and A_3 and then multiply A_1 by the result.

If the results are the same, why do we care? Recall that the efficiency of the CLRS matrix multiplication algorithm depends on the dimensions of the matrices. Specifically, multiplying an $a \times b$ matrix by a $b \times c$ matrix requires roughly abc computation steps. In our example, we will let let A_1 be a $p_0 \times p_1$ matrix, A_2 be a $p_1 \times p_2$ matrix, and A_3 be a $p_2 \times p_3$ matrix.

Let us first consider $A_1(A_2A_3)$. Multiplying A_2 and A_3 using our simple algorithm will take $(p_1p_2p_3)$ steps and multiplying A_1 and A_2A_3 will take $(p_0p_1p_3)$ steps. (Recall that A_2A_3 is a $p_1 \times p_3$ matrix.) Thus under this parenthesization the total multiplication will take roughly $(p_0p_1p_3 + p_1p_2p_3)$ steps.

Now consider $(A_1A_2)A_3$. Multiplying A_1 and A_2 will take $(p_0p_1p_2)$ steps and multiplying A_1A_2 and A_3 will take $(p_0p_2p_3)$ steps. Thus under this parenthesization the total multiplication will take roughly $(p_0p_1p_2 + p_0p_2p_3)$ steps.

It is clear that for any given values of p_0, p_1, p_2, and p_3, one of the parenthesizations will likely yield a more efficient multiplication procedure. For example, if $p_0 = 1$, $p_1 = 2$, $p_2 = 3$, and $p_3 = 4$, the the first parenthesization will require roughly $(p_0p_1p_3 + p_1p_2p_3) = 8 + 24 = 32$ steps, though the second parenthesization will require only roughly $(p_0p_1p_2 + p_0p_2p_3) = 6 + 12 = 18$ steps.

Our goal will be to determine the most efficient way to carry out the multiplication (i.e. parenthesize the expression so that the amount of steps needed for multiplication are minimized). The naive way to do this is to try all possible parenthesizations, and pick the one that minimizes computation time.

For 4 matrices there are 5 different parenthesizations we would need to check: $(A_1A_2)(A_3A_4)$, $A_1((A_2A_3)A_4)$, $A_1(A_2(A_3A_4))$, $((A_1)A_2)A_3A_4$ and $(A_1(A_2A_3))A_4$.

For 5 matrices there are 15 different parenthesizations: 5 ways for $A_1(A_2A_3A_4A_5)$, 2 ways for $(A_1A_2)(A_3A_4A_5)$ 2 ways for $(A_1A_2A_3)(A_4A_5)$ and 5 ways for $(A_1A_2A_3A_4)A_5$.

The number of parenthesizations grows quite quickly, and we can categorize this growth with a recurrence relation. Specifically, suppose $P(n)$ is the event of parenthesizing the product of n matrices. We apply the rule of sum to $P(n)$ to split it into events E_i according to the break for the last multiplication.

For example, when multiplying 5 matrices $A_0 \ldots A_5$, E_2 will correspond to a last multiplication of $(A_1A_2)(A_3A_4A_5)$. In other words, when multiplying these matrices we first multiplied A_1A_2 somehow, and then multiplied $A_3A_4A_5$ somehow, and our last multiplication is the product of these two quantities.

Therefore, the number of events in E_i will consist of the number of ways of parenthesizing $A_1 \ldots A_i$ **and** parenthesizing $A_{i+1} \ldots A_n$. In other words, $E_i = P(i)P(n-i)$. Based on the rule of sum, we can therefore establish the following recurrence for $P(n)$:

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$

We can examine some values of this recurrence:

$$\begin{aligned}
P(1) &= 1 \\
P(2) &= 1 \\
P(3) &= P(1)P(2) + P(2)P(1) = 2 \\
P(4) &= P(1)P(3) + P(2)P(2) + P(3)P(1) = 5 \\
P(5) &= P(1)P(4) + P(2)P(3) + P(3)P(2) + P(4)P(1) = 14 \\
P(6) &= P(1)P(5) + P(2)P(4) + P(3)P(3) + P(4)P(2) + P(5)P(1) = 42
\end{aligned}$$

It turns out that this is an extremely messy recurrence to solve. Nevertheless, this is a very well-known sequence of numbers called the **Catalan Numbers**. In fact:

$$P(n) = \frac{1}{n}\binom{2n-2}{n-1}$$

which is just the n^{th} row center (from the top) of Pascal's Triangle divided by n. Pretty neat, no?

Notice that the Catalan Numbers grow quite fast. If we exhaustively search each and every possible parenthesization, we would be in school for a very long time. Thus, we have to come up with a better solution that is based on dynamic programming.

Specifically, suppose you had a magic oracle that could tell you the best place to perform the last multiplication of a chain-multiplication. Then you could simply split up the chain-multiplication according to the advice of the oracle, and then use the oracle again on the two sub-chains, etc.

The reason you can use this oracle is that if you have an optimal solution for the parenthesization, then every subchain of the solution must also be optimally parenthesized (or else you could replace the non-optimal parenthesization with an optimal one to get a better solution). This is a subtle point...think about it. This property of a problem is often called **optimal substructure**.

In reality, we do not have an oracle for the parenthesization, and we have to actually work to figure out the solution the oracle would give us, but we **can** store the optimal sub-chains as we compute them so that they do not need to be recomputed every time.

Specifically, let us define $C(i,j)$ as the "cost" (# of steps) of multiplying $A_i, A_{i+1}, \cdots, A_{j-1}, A_j$ in the optimal way. Clearly, $C(1,n)$ is the optimal way to multiply all of our n matrices, which is what we are trying to compute.

With careful examination, we can see that:

$$C(i,j) = \min_{i \le k < j} (C(i,k) + C(k+1,j) + p_{i-1}p_k p_j) \tag{1}$$

In other words, we look through every possibility for the last multiplication ($\min_{i \le k < j} \ldots$) and check the cost of the chain multiplication if we were to optimally multiply the sub-chains ($C(i,k)+ C(k+1,j)+ p_{i-1}p_k p_j$).

We know that $C(i,i) = 0$ so we may now recursively solve for $C(i,j)$. Moreover, each time we calculate $C(i,j)$, we can store it in a cell location (i,j) on an $n \times n$ table, so that we would only have to compute values of $C(i,j)$ once for each i and j. Let's draw this table more clearly:

	$j=1$	$j=2$	$j=3$	$\ldots$	$j=n$
$i=1$	$C(1,1)$	$C(1,2)$	$C(1,3)$	$\cdots$	$C(1,n)$
$i=2$	$C(2,1)$	$C(2,2)$	$C(2,3)$	$\cdots$	$C(2,n)$
$i=3$	$C(3,1)$	$C(3,2)$	$C(3,3)$	$\cdots$	$C(3,n)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$i=n$	$C(n,1)$	$C(n,2)$	$C(n,3)$	$\cdots$	$C(n,n)$

We may now put in some knowledge about the table. Specifically, it makes sense that $C(i,i) = 0$ and that $C(i,j)$ is non-existent when $i > j$.

	$j=1$	$j=2$	$j=3$	$\ldots$	$j=n$
$i=1$	0	$C(1,2)$	$C(1,3)$	$\cdots$	$C(1,n)$
$i=2$		0	$C(2,3)$	$\cdots$	$C(2,n)$
$i=3$			0	$\cdots$	$C(3,n)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$i=n$				$\cdots$	0

Now to fill (or compute) cell (i,j) of the table, we need to have computed all the cells below (i,j) and also all the cells to the left of (i,j), as we can see from our general minimization equation (1). Thus, one way to fill all the cells in the table is to zig-zag across the table, computing cells in the following diagrammed order:

0	fill	$C(1,3)$	$\cdots$	$C(1,n)$
	0	fill	$\cdots$	$C(2,n)$
		0	fill	$C(3,n)$
			$\ddots$	fill
				0

0	fill	fill	$\cdots$	$C(1,n)$
	0	fill	fill	$C(2,n)$
		0	fill	fill
			$\ddots$	fill
				0

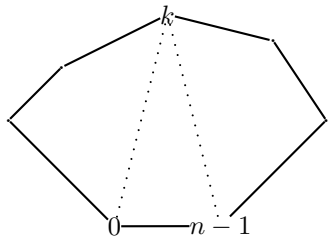
0	fill	fill	fill	$C(1,n)$
	0	fill	fill	fill
		0	fill	fill
			$\ddots$	fill
				0

When we finally get to $C(1,n)$, we have computed the solution to our problem. This algorithm is such that we will fill about $n^2/2$ cells of the table. Each value requires $O(n)$ work to fill (taking the minimum of all the appropriate cell sums). The entire algorithm turns out to require about n^3 steps.

Optimal Polygon Triangulation

In the problem of optimal polygon triangulation, we determine a triangulation of a polygon[1] such that the sum of the lengths of the sides of the triangles is minimized.

Consider a polygon with n vertices, numbered 0 through $n-1$:



Clearly the edge between vertex 0 and vertex $n-1$ must be on some triangle. If the third vertex on this triangle is vertex k, this triangle is formed by the addition of the dotted lines shown. This gives us a natural way to break the problem into smaller pieces: first draw the triangle shown and then triangulate the polygons on vertices 0 through k and k through $n-1$. Further, if we find optimal triangulations of these smaller polygons, then the overall triangulation will be optimal.

Unfortunately, the problem is not this simple. We have no way of quickly knowing which vertex forms the triangle with vertices 0 and $n-1$. Therefore we have to try all possibilities. Generalizing this process leads to the following recurrence for $C(i,j)$, the lowest cost to triangulate the polygon on vertices $i \ldots j$:

$$C(i,j) = \min_{i<k<j} [C(i,k) + C(k,j) + \overline{ik} + \overline{kj}]$$

$$C(i,(i+2) \bmod n) = 0, \qquad 0 \le i \le n-1$$

This recurrence can be translated into a straightforward recursive program to find the minimum triangulation. Unfortunately, the resulting program runs in exponential time. The problem is that the smaller triangulation problems are solved many times. To prevent this, the solutions to these subproblems should be stored in a two-dimensional array. In fact, rather than making the recursive calls suggested above, the triangulation problem can be solved by filling in the entries of this array, beginning with the base case elements ($C(i,i+2)$- the second row above the diagonal) and working upward to the corner, which will hold the desired solution. Since at most $O(n)$ work is required to fill in an entry and there are $O(n^2)$ entries, this algorithm takes $O(n^3)$ time.

Longest Common Subsequence

Next we examine the longest common subsequence problem. In this problem, we are given two sequences X and Y and are asked to find the subsequence of both of maximum length. Importantly, we are looking for a *subsequence* rather than a *substring*, i.e. the common elements do not need to be contiguous. This makes the problem much harder since, in a sequence of length n, there are $O(n^2)$ substrings, but 2^n subsequences (section 15.4 of CLRS).

To break the problem into smaller subproblems, we can compare the last elements. If these elements are equal, then there must be a longest common subsequence which contains them. In this case, we search for

[1] A triangulation of a polygon is a set of chords of the polygon that divide it into disjoint triangles

the longest common subsequence of the remaining elements and then append the final matching element. If the last elements are not equal, then the longest common subsequence must be either the longest common subsequence of the first sequence and the last sequence minus the final element or the longest common subsequence of the first sequence minus the final element and the second sequence.

Once again, the obvious recursive implementation of this property runs in exponental time, but the algorithm can be made efficient by storing the solution to smaller problems in a two-dimensional array. If L_{ij} is the length of the longest common subsequence of the first i elements of the first sequence and the first j elements of the second sequence, we get the following recurrence:

$$L_{ij} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L_{i-1,j-1} + 1 & \text{if } x_i = y_j \\ \max(L_{i,j-1}, & \\ \qquad L_{i-1,j}) & \text{otherwise} \end{cases}$$

This can be used to fill in the table of values of L, beginning with row 0 and column 0. The desired solution will occupy L_{nm}, where n is the length of the first sequence and m is the length of the second sequence. Since each entry can be computed in constant time, filling in the table takes $O(nm)$ time.

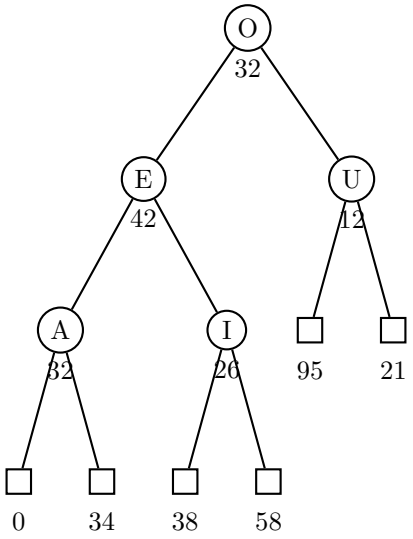
Optimal Binary Search Trees

As we have seen, one property of binary search trees is that elements near the root of the tree can be accessed relatively quickly, while elements far from the root take longer to access. In the problem of optimal binary search trees, we are given a set of elements $\{x_1, x_2, \ldots, x_n\}$, along with search frequencies $\{q_1, q_2, \ldots, q_n\}$ and search failure frequencies $\{p_0, p_1, p_2, \ldots, p_n\}$, where q_i is the frequency of x_i being accessed and p_i is the frequency of search failure occurring between x_i and x_{i+1}. We would like to build a binary search tree with the property that the cost of an average search is minimized. The application here is one in which a table is constructed once and accessed many times but is not modified often (see section 15.5 in CLRS).

For instance, consider the vowels A, E, I, O, and U, with their frequencies of occurrence in English text. We might have the following frequencies:

i	x_i	q_i	p_i
0			0
1	A	32	34
2	E	42	38
3	I	26	58
4	O	32	95
5	U	12	21

This table tells us, for example, that the letter I occurs 26 times. Letters between E and I occur 38 times and letters between I and O occur 58 times. One possible binary search tree representing these five letters is:



We can formalize the notion of average search cost by defining the *weighted path length*:

$$WPL(T) = \sum_{i=1}^{n} q_i(1 + depth(x_i)) + \sum_{i=0}^{n} p_i \cdot depth(y_i)$$

where $x_1, x_2, \ldots, x_n$ are internal nodes and $y_0, y_1, y_2, \ldots, y_n$ are external nodes. We can give an alternate (and equivalent) recursive definition:

$$\begin{aligned} WPL(\square) &= 0 \\ WPL(T) &= WPL(T_l) + WPL(T_r) + \sum p_i + \sum q_i \end{aligned}$$

It follows that the weighted path length divided by $\sum p_i + \sum q_i$ is the average search time.
For the above tree T,

$$\begin{aligned} WPL(T) &= \sum_{i=1}^{5} q_i(1 + depth(x_i)) + \sum_{i=0}^{5} p_i \cdot depth(y_i) \\ &= (32 \cdot 3 + 42 \cdot 2 + 26 \cdot 3 + 32 \cdot 1 + 12 \cdot 2) + (0 \cdot 3 + 34 \cdot 3 + 38 \cdot 3 + 58 \cdot 3 + 95 \cdot 2 + 21 \cdot 2) \\ &= 314 + 622 \\ &= 936 \end{aligned}$$

Dividing by $\sum p_i + \sum q_i$,

$$\begin{aligned} \frac{WPL(T)}{\sum p_i + \sum q_i} &= \frac{936}{390} \\ &= 2.4 \end{aligned}$$

One approach is to build all possible trees and then determine which is best. Unfortunately, the number of trees is exponential in n. However, if we observe that, for any root, its subtrees must be optimal, we can invoke dynamic programming.

A Dynamic Programming Solution

Observe that, if T is an optimal binary search tree on weights $p_0, q_1, p_1, \ldots, q_n, p_n$ with weight q_i at the root, the left subtree must be optimal on weights $p_0, q_1, p_1, \ldots, q_{i-1}, p_{i-1}$ and the right subtree must be optimal on weights $p_i, q_{i+1}, p_{i+1}, \ldots, q_n, p_n$. If this were not the case, then we could replace one of the subtrees with the optimal subtree, yielding a lower overall weighted path length, thereby yielding a contradiction. This is simply the property of optimal substructure, and it steers us towards dynamic programming as a (hopefully) efficient solution.

Let C_{ij}, where $0 \le i \le j \le n$, be the cost of an optimal tree over weights $p_i, q_{i+1}, p_{i+1}, \ldots, p_j$. By the optimal substructure property above, we can define this recursively:

$$\begin{aligned} C_{ii} &= 0 \\ C_{ij} &= \min_{i<k\le j}(C_{i,k-1} + C_{kj}) + \sum_{t=i}^{j} p_t + \sum_{t=i+1}^{j} q_t \end{aligned}$$

Defining

$$\begin{aligned} W_{ii} &= p_i \\ W_{ij} &= W_{i,j-1} + q_j + p_j, \text{ where } i < j \end{aligned} \tag{2}$$

so that $W_{ij} = p_i + q_{i+1} + p_{i+1} + \cdots + q_j + p_j$. We then have

$$\begin{aligned} C_{ii} &= 0 \\ C_{ij} &= W_{ij} + \min_{i<k\le j}(C_{i,k-1} + C_{kj}) \end{aligned} \tag{3}$$

Finally, we define

$$R_{ij} = \text{a value of } k \text{ that minimizes } C_{i,k-1} + C_{kj} \text{ in equation (3)} \tag{4}$$

As in the other dynamic programming problems, we could devise a recursive solution around equations (2) and (3), but that would lead to the exponential blowup we are trying to avoid. Instead, we build a table of values of R_{ij}, W_{ij}, and C_{ij} for $0 \le i \le j \le n$ based on equations (2), (3), and (4). From the completed table, we can easily construct the optimal tree.

Since the time required to compute one entry in the table is $O(n)$, the total time required for the algorithm is $O(n^3)$.

A Better Solution

We can improve on this solution by noticing that this algorithm does more work than is necessary. In particular, there is an optimal tree over $p_i, q_{i+1}, \ldots, q_j, p_j$ whose root R_{ij} satisfies $R_{i,j-1} \le R_{ij} \le R_{i+1,j}$. This reduces the total amount of work required to fill in the table to $O(n^2)$.

The Traveling Salesman Problem

Dynamic programming does not always yield polynomial-time algorithms (a fact that the text does not discuss), but even when it gives exponential-time algorithms, it can provide a useful framework for algorithm design. In this section we'll look at such an example, the Travelling Salesman Problem, or TSP for short.

In the TSP, we are given n cities and a distance matrix C in which C_{ij} is the distance from city i to city j. These distances do not have to correspond to physical distances; they are costs. Because they are costs, they do not need to satisfy constraints such as symmetry (the distance from i to j need not be the same as the distance from j to i) or the triangle inequality (it is not necessarily shorter to go from i to j directly—it mean be cheaper to go from i to k to j). We need to route a salesman from his home city, arbitrarily called city 1, through all other $n-1$ cities and then back home to city 1. Such a routing is called a *tour*. The TSP is to find the cheapest tour.

There are $(n-1)!$ possible tours (why?), so a brute-force examination of the cost of each tour would take time $(n-1) \times (n-1)!$ because it would take $n-1$ additions to compute the cost of each tour (this could be improved by choosing successive tours that differ only by the interchange of two cities). With a dynamic programming approach we can improve this time bound to $O(n^2 2^n)$. We define

$$T(i; j_1, j_2, \ldots, j_k) = \begin{cases} \text{cost of the optimal tour from city } i \text{ to city} \\ 1 \text{ that goes through each of the intermediate} \\ \text{cities } j_1, j_2, \ldots, j_k \text{ exactly once, in any order, and} \\ \text{through no other cities.} \end{cases}$$

The optimal substructure property tells us that

$$T(i; j_1, j_2, \ldots, j_k) = \min_{1 \le m \le k} \{C_{ij_m} + T(j_m; j_1, j_2, \ldots, j_{m-1}, j_{m+1}, \ldots j_k)\}.$$

Furthermore,

$$T(i; \emptyset) = C_{i1}.$$

The value we want is

$$T(1; 2, 3, \ldots, n).$$

Without memoization, direct evaluation of $T(i; j_1, j_2, \ldots, j_k)$ takes time $\Theta(k)$ time plus the time for the k recursive calls; if $t(k)$ is the order of the time needed for k intermediate cities,

$$t(k) = k + k \times t(k-1)$$

and $t(0)$ is constant. The overall time is then $t(n-1) > (n-1)!$.

With memoization, there are $n-1$ choices for city i and $\binom{n-2}{k}$ sets of k intermediate cites chosen from among all cities except cities 1 and i. The total number of memos [including the memo for $T(1; 2, 3, \ldots, n)$] is thus

$$1 + \sum_{k=1}^{n-2} (n-1) \binom{n-2}{k} = 1 + (n-1) \sum_{k=1}^{n-2} \binom{n-2}{k} = 1 + (n-1)(2^{n-2} - 1).$$

Evaluating a memo once all the needed memos are available is $O(n)$, so the overall cost of the memoized algorithm is $O(n^2 2^n)$.