The running time of the algorithm is clearly $O(mn)$ since there are two nested loops with m and n iterations, respectively. The algorithm also uses $O(mn)$ space.

Extracting the Actual Sequence: Extracting the final LCS is done by using the back pointers stored in $b[0..m, 0..n]$. Intuitively $b[i, j] = \textit{add}_{XY}$ means that $X[i]$ and $Y[j]$ together form the last character of the LCS. So we take this common character, and continue with entry $b[i-1, j-1]$ to the northwest ($\nwarrow$). If $b[i, j] = \textit{skip}_X$, then we know that $X[i]$ is not in the LCS, and so we skip it and go to $b[i-1, j]$ above us ($\uparrow$). Similarly, if $b[i, j] = \textit{skip}_Y$, then we know that $Y[j]$ is not in the LCS, and so we skip it and go to $b[i, j-1]$ to the left ($\leftarrow$). Following these back pointers, and outputting a character with each diagonal move gives the final subsequence.

Lecture 5: Dynamic Programming: Chain Matrix Multiplication

Read: Chapter 15 of CLRS, and Section 15.2 in particular.

Chain Matrix Multiplication: This problem involves the question of determining the optimal sequence for performing a series of operations. This general class of problem is important in compiler design for code optimization and in databases for query optimization. We will study the problem in a very restricted instance, where the dynamic programming issues are easiest to see.

Suppose that we wish to multiply a series of matrices

$$A_1 A_2 \ldots A_n$$

Matrix multiplication is an associative but not a commutative operation. This means that we are free to parenthesize the above multiplication however we like, but we are not free to rearrange the order of the matrices. Also recall that when two (nonsquare) matrices are being multiplied, there are restrictions on the dimensions. A $p \times q$ matrix has p rows and q columns. You can multiply a $p \times q$ matrix A times a $q \times r$ matrix B, and the result will be a $p \times r$ matrix C. (The number of columns of A must equal the number of rows of B.) In particular for $1 \leq i \leq p$ and $1 \leq j \leq r$,

$$C[i, j] = \sum_{k=1}^{q} A[i, k] B[k, j].$$

This corresponds to the (hopefully familiar) rule that the $[i, j]$ entry of C is the dot product of the ith (horizontal) row of A and the jth (vertical) column of B. Observe that there are pr total entries in C and each takes $O(q)$ time to compute, thus the total time to multiply these two matrices is proportional to the product of the dimensions, pqr.

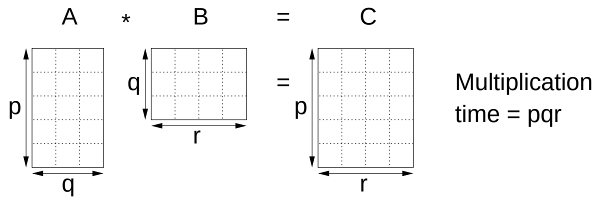

Fig. 7: Matrix Multiplication.

Note that although any legal parenthesization will lead to a valid result, not all involve the same number of operations. Consider the case of 3 matrices: A_1 be 5×4, A_2 be 4×6 and A_3 be 6×2.

$$\begin{aligned} \text{multCost}[((A_1 A_2) A_3)] &= (5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180, \\ \text{multCost}[(A_1 (A_2 A_3))] &= (4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) = 88. \end{aligned}$$

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.

Chain Matrix Multiplication Problem: Given a sequence of matrices $A_1, A_2, \ldots, A_n$ and dimensions $p_0, p_1, \ldots, p_n$ where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication (represented, say, as a binary tree) that minimizes the number of operations.

Important Note: This algorithm does not perform the multiplications, it just determines the best order in which to perform the multiplications.

Naive Algorithm: We could write a procedure which tries all possible parenthesizations. Unfortunately, the number of ways of parenthesizing an expression is very large. If you have just one or two matrices, then there is only one way to parenthesize. If you have n items, then there are $n-1$ places where you could break the list with the outermost pair of parentheses, namely just after the 1st item, just after the 2nd item, etc., and just after the $(n-1)$st item. When we split just after the kth item, we create two sublists to be parenthesized, one with k items, and the other with $n-k$ items. Then we could consider all the ways of parenthesizing these. Since these are independent choices, if there are L ways to parenthesize the left sublist and R ways to parenthesize the right sublist, then the total is $L \cdot R$. This suggests the following recurrence for $P(n)$, the number of different ways of parenthesizing n items:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

This is related to a famous function in combinatorics called the *Catalan numbers* (which in turn is related to the number of different binary trees on n nodes). In particular $P(n) = C(n-1)$, where $C(n)$ is the nth Catalan number:

$$C(n) = \frac{1}{n+1}\binom{2n}{n}.$$

Applying Stirling's formula (which is given in our text), we find that $C(n) \in \Omega(4^n/n^{3/2})$. Since 4^n is exponential and $n^{3/2}$ is just polynomial, the exponential will dominate, implying that function grows very fast. Thus, this will not be practical except for very small n. In summary, brute force is not an option.

Dynamic Programming Approach: This problem, like other dynamic programming problems involves determining a structure (in this case, a parenthesization). We want to break the problem into subproblems, whose solutions can be combined to solve the global problem. As is common to any DP solution, we need to find some way to break the problem into smaller subproblems, and we need to determine a recursive formulation, which represents the optimum solution to each problem in terms of solutions to the subproblems. Let us think of how we can do this.

Since matrices cannot be reordered, it makes sense to think about sequences of matrices. Let $A_{i..j}$ denote the result of multiplying matrices i through j. It is easy to see that $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix. (Think about this for a second to be sure you see why.) Now, in order to determine how to perform this multiplication optimally, we need to make many decisions. What we want to do is to break the problem into problems of a similar structure. In parenthesizing the expression, we can consider the highest level of parenthesization. At this level we are simply multiplying two matrices together. That is, for any k, $1 \leq k \leq n-1$,

$$A_{1..n} = A_{1..k} \cdot A_{k+1..n}.$$

Thus the problem of determining the optimal sequence of multiplications is broken up into two questions: how do we decide where to split the chain (what is k?) and how do we parenthesize the subchains $A_{1..k}$ and $A_{k+1..n}$? The subchain problems can be solved recursively, by applying the same scheme.

So, let us think about the problem of determining the best value of k. At this point, you may be tempted to consider some clever ideas. For example, since we want matrices with small dimensions, pick the value of k that minimizes p_k. Although this is not a bad idea, in principle. (After all it might work. It just turns out that it doesn't in this case. This takes a bit of thinking, which you should try.) Instead, as is true in almost all dynamic programming solutions, we will do the dumbest thing of simply considering *all possible* choices of k, and taking the best of them. Usually trying all possible choices is bad, since it quickly leads to an exponential number of total possibilities. What saves us here is that there are only $O(n^2)$ different sequences of matrices. (There are $\binom{n}{2} = n(n-1)/2$ ways of choosing i and j to form $A_{i..j}$ to be precise.) Thus, we do not encounter the exponential growth.

Notice that our chain matrix multiplication problem satisfies the principle of optimality, because once we decide to break the sequence into the product $A_{1..k} \cdot A_{k+1..n}$, we should compute each subsequence optimally. That is, for the global problem to be solved optimally, the subproblems must be solved optimally as well.

Dynamic Programming Formulation: We will store the solutions to the subproblems in a table, and build the table in a bottom-up manner. For $1 \leq i \leq j \leq n$, let $m[i,j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. The optimum cost can be described by the following recursive formulation.

Basis: Observe that if $i = j$ then the sequence contains only one matrix, and so the cost is 0. (There is nothing to multiply.) Thus, $m[i,i] = 0$.

Step: If $i < j$, then we are asking about the product $A_{i..j}$. This can be split by considering each k, $i \leq k < j$, as $A_{i..k}$ times $A_{k+1..j}$.

The optimum times to compute $A_{i..k}$ and $A_{k+1..j}$ are, by definition, $m[i,k]$ and $m[k+1,j]$, respectively. We may assume that these values have been computed previously and are already stored in our array. Since $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix, and $A_{k+1..j}$ is a $p_k \times p_j$ matrix, the time to multiply them is $p_{i-1}p_kp_j$. This suggests the following recursive rule for computing $m[i,j]$.

$$\begin{aligned} m[i,i] &= 0 \\ m[i,j] &= \min_{i \leq k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) \qquad \text{for } i < j. \end{aligned}$$

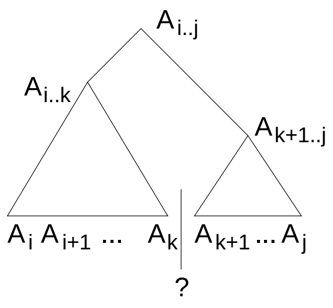

Fig. 8: Dynamic Programming Formulation.

It is not hard to convert this rule into a procedure, which is given below. The only tricky part is arranging the order in which to compute the values. In the process of computing $m[i,j]$ we need to access values $m[i,k]$ and $m[k+1,j]$ for k lying between i and j. This suggests that we should organize our computation according to the number of matrices in the subsequence. Let $L = j-i+1$ denote the length of the subchain being multiplied. The subchains of length 1 ($m[i,i]$) are trivial to compute. Then we build up by computing the subchains of lengths $2, 3, \ldots, n$. The final answer is $m[1,n]$. We need to be a little careful in setting up the loops. If a subchain of length L starts at position i, then $j = i + L - 1$. Since we want $j \leq n$, this means that $i + L - 1 \leq n$, or in other words, $i \leq n - L + 1$. So our loop for i runs from 1 to $n - L + 1$ (in order to keep j in bounds). The code is presented below.

The array $s[i,j]$ will be explained later. It is used to extract the actual sequence. The running time of the procedure is $\Theta(n^3)$. We'll leave this as an exercise in solving sums, but the key is that there are three nested loops, and each can iterate at most n times.

Extracting the final Sequence: Extracting the actual multiplication sequence is a fairly easy extension. The basic idea is to leave a *split marker* indicating what the best split is, that is, the value of k that leads to the minimum

Chain Matrix Multiplication

```
Matrix-Chain(array p[1..n]) {
    array s[1..n-1,2..n]
    for i = 1 to n do m[i,i] = 0;              // initialize
    for L = 2 to n do {                        // L = length of subchain
        for i = 1 to n-L+1 do {
            j = i + L - 1;
            m[i,j] = INFINITY;
            for k = i to j-1 do {              // check all splits
                q = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j]
                if (q < m[i, j]) {
                    m[i,j] = q;
                    s[i,j] = k;
                }
            }
        }
    }
    return m[1,n] (final cost) and s (splitting markers);
}
```

value of $m[i, j]$. We can maintain a parallel array $s[i, j]$ in which we will store the value of k providing the optimal split. For example, suppose that $s[i, j] = k$. This tells us that the best way to multiply the subchain $A_{i..j}$ is to first multiply the subchain $A_{i..k}$ and then multiply the subchain $A_{k+1..j}$, and finally multiply these together. Intuitively, $s[i, j]$ tells us what multiplication to perform *last*. Note that we only need to store $s[i, j]$ when we have at least two matrices, that is, if $j > i$.

The actual multiplication algorithm uses the $s[i, j]$ value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices $A[1..n]$, and that $s[i, j]$ is global to this recursive procedure. The recursive procedure Mult does this computation and below returns a matrix.

Extracting Optimum Sequence

```
Mult(i, j) {
    if (i == j)                            // basis case
        return A[i];
    else {
        k = s[i,j]
        X = Mult(i, k)                     // X = A[i]...A[k]
        Y = Mult(k+1, j)                   // Y = A[k+1]...A[j]
        return X*Y;                        // multiply matrices X and Y
    }
}
```

In the figure below we show an example. This algorithm is tricky, so it would be a good idea to trace through this example (and the one given in the text). The initial set of dimensions are $\langle 5, 4, 6, 2, 7\rangle$ meaning that we are multiplying A_1 (5×4) times A_2 (4×6) times A_3 (6×2) times A_4 (2×7). The optimal sequence is $((A_1(A_2A_3))A_4)$.

Lecture 6: Dynamic Programming: Minimum Weight Triangulation

Read: This is not covered in CLRS.

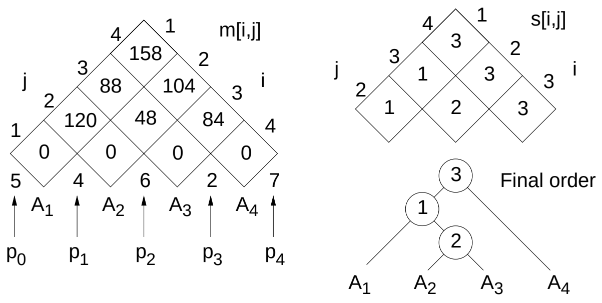

Fig. 9: Chain Matrix Multiplication Example.

Polygons and Triangulations: Let's consider a geometric problem that outwardly appears to be quite different from chain-matrix multiplication, but actually has remarkable similarities. We begin with a number of definitions. Define a *polygon* to be a piecewise linear closed curve in the plane. In other words, we form a cycle by joining line segments end to end. The line segments are called the *sides* of the polygon and the endpoints are called the *vertices*. A polygon is *simple* if it does not cross itself, that is, if the sides do not intersect one another except for two consecutive sides sharing a common vertex. A simple polygon subdivides the plane into its *interior*, its *boundary* and its *exterior*. A simple polygon is said to be *convex* if every interior angle is at most 180 degrees. Vertices with interior angle equal to 180 degrees are normally allowed, but for this problem we will assume that no such vertices exist.

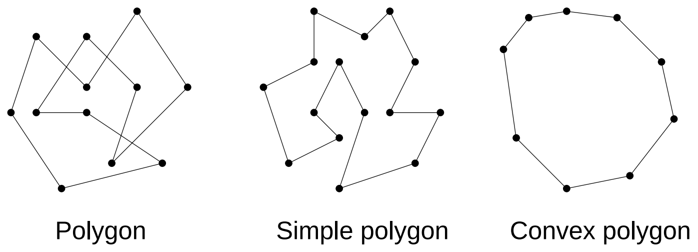

Fig. 10: Polygons.

Given a convex polygon, we assume that its vertices are labeled in counterclockwise order $P = \langle v_1, \ldots, v_n \rangle$. We will assume that indexing of vertices is done modulo n, so $v_0 = v_n$. This polygon has n sides, $\overline{v_{i-1}v_i}$.

Given two nonadjacent sides v_i and v_j, where $i < j-1$, the line segment $\overline{v_iv_j}$ is a *chord*. (If the polygon is simple but not convex, we include the additional requirement that the interior of the segment must lie entirely in the interior of P.) Any chord subdivides the polygon into two polygons: $\langle v_i, v_{i+1}, \ldots, v_j \rangle$, and $\langle v_j, v_{j+1}, \ldots, v_i \rangle$. A *triangulation* of a convex polygon P is a subdivision of the interior of P into a collection of triangles with disjoint interiors, whose vertices are drawn from the vertices of P. Equivalently, we can define a triangulation as a maximal set T of nonintersecting chords. (In other words, every chord that is not in T intersects the interior of some chord in T.) It is easy to see that such a set of chords subdivides the interior of the polygon into a collection of triangles with pairwise disjoint interiors (and hence the name *triangulation*). It is not hard to prove (by induction) that every triangulation of an n-sided polygon consists of $n - 3$ chords and $n - 2$ triangles. Triangulations are of interest for a number of reasons. Many geometric algorithm operate by first decomposing a complex polygonal shape into triangles.

In general, given a convex polygon, there are many possible triangulations. In fact, the number is exponential in n, the number of sides. Which triangulation is the "best"? There are many criteria that are used depending on the application. One criterion is to imagine that you must "pay" for the ink you use in drawing the triangulation, and you want to minimize the amount of ink you use. (This may sound fanciful, but minimizing wire length is an important condition in chip design. Further, this is one of many properties which we could choose to optimize.) This suggests the following optimization problem:

Minimum-weight convex polygon triangulation: Given a convex polygon determine the triangulation that minimizes the sum of the perimeters of its triangles. (See Fig. 11.)

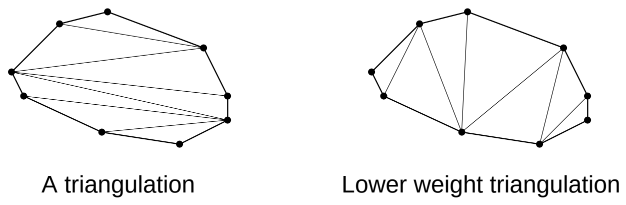

Fig. 11: Triangulations of convex polygons, and the minimum weight triangulation.

Given three distinct vertices v_i, v_j, v_k, we define the *weight* of the associated triangle by the weight function

$$w(v_i, v_j, v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|,$$

where $|v_i v_j|$ denotes the length of the line segment $\overline{v_i v_j}$.

Dynamic Programming Solution: Let us consider an $(n+1)$-sided polygon $P = \langle v_0, v_1, \ldots, v_n \rangle$. Let us assume that these vertices have been numbered in counterclockwise order. To derive a DP formulation we need to define a set of subproblems from which we can derive the optimum solution. For $0 \le i < j \le n$, define $t[i,j]$ to be the weight of the minimum weight triangulation for the subpolygon that lies to the right of directed chord $\overline{v_i v_j}$, that is, the polygon with the counterclockwise vertex sequence $\langle v_i, v_{i+1}, \ldots, v_j \rangle$. Observe that if we can compute this quantity for all such i and j, then the weight of the minimum weight triangulation of the entire polygon can be extracted as $t[0,n]$. (As usual, we only compute the minimum weight. But, it is easy to modify the procedure to extract the actual triangulation.)

As a basis case, we define the weight of the trivial "2-sided polygon" to be zero, implying that $t[i, i+1] = 0$. In general, to compute $t[i,j]$, consider the subpolygon $\langle v_i, v_{i+1}, \ldots, v_j \rangle$, where $j > i+1$. One of the chords of this polygon is the side $\overline{v_i v_j}$. We may split this subpolygon by introducing a triangle whose base is this chord, and whose third vertex is any vertex v_k, where $i < k < j$. This subdivides the polygon into the subpolygons $\langle v_i, v_{i+1}, \ldots v_k \rangle$ and $\langle v_k, v_{k+1}, \ldots v_j \rangle$ whose minimum weights are already known to us as $t[i,k]$ and $t[k,j]$. In addition we should consider the weight of the newly added triangle $\triangle v_i v_k v_j$. Thus, we have the following recursive rule:

$$t[i,j] = \begin{cases} 0 & \text{if } j = i+1 \\ \min_{i<k<j}(t[i,k] + t[k,j] + w(v_i v_k v_j)) & \text{if } j > i+1. \end{cases}$$

The final output is the overall minimum weight, which is, $t[0,n]$. This is illustrated in Fig. 12

Note that this has almost exactly the same structure as the recursive definition used in the chain matrix multiplication algorithm (except that some indices are different by 1.) The same $\Theta(n^3)$ algorithm can be applied with only minor changes.

Relationship to Binary Trees: One explanation behind the similarity of triangulations and the chain matrix multiplication algorithm is to observe that both are fundamentally related to binary trees. In the case of the chain matrix multiplication, the associated binary tree is the evaluation tree for the multiplication, where the leaves of the tree correspond to the matrices, and each node of the tree is associated with a product of a sequence of two or more matrices. To see that there is a similar correspondence here, consider an $(n+1)$-sided convex polygon $P = \langle v_0, v_1, \ldots, v_n \rangle$, and fix one side of the polygon (say $\overline{v_0 v_n}$). Now consider a rooted binary tree whose root node is the triangle containing side $\overline{v_0 v_n}$, whose internal nodes are the nodes of the dual tree, and whose leaves

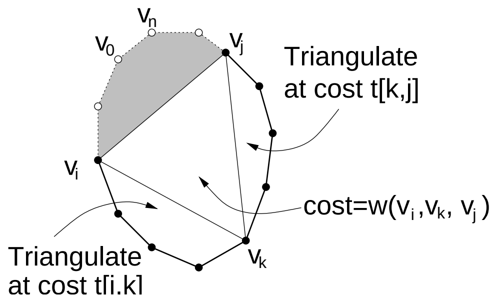
v_n
v_0
v_j
Triangulate at cost t[k,j]
v_i
cost=w(v_i,v_k, v_j)
Triangulate at cost t[i.k]
v_k

Fig. 12: Triangulations and tree structure.

correspond to the remaining sides of the tree. Observe that partitioning the polygon into triangles is equivalent to a binary tree with n leaves, and vice versa. This is illustrated in Fig. 13. Note that every triangle is associated with an internal node of the tree and every edge of the original polygon, except for the distinguished starting side $\overline{v_0 v_n}$, is associated with a leaf node of the tree.

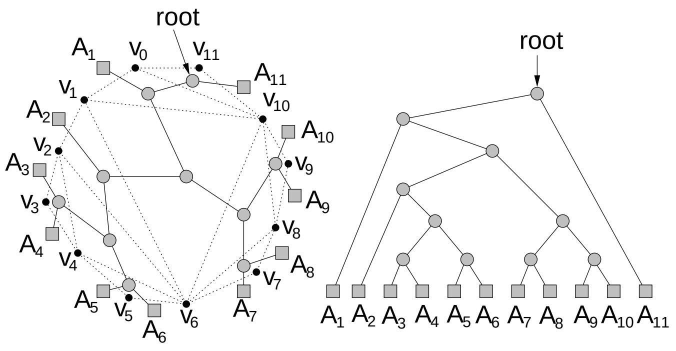
root
A_1 v_0 v_{11} A_{11} v_1 v_{10} A_2 A_{10} v_2 v_9 A_3 A_9 v_3 v_8 A_4 v_4 A_8 v_7 A_5 v_5 A_6 v_6 A_7
root
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11}

Fig. 13: Triangulations and tree structure.

Once you see this connection. Then the following two observations follow easily. Observe that the associated binary tree has n leaves, and hence (by standard results on binary trees) $n - 1$ internal nodes. Since each internal node other than the root has one edge entering it, there are $n-2$ edges between the internal nodes. Each internal node corresponds to one triangle, and each edge between internal nodes corresponds to one chord of the triangulation.

Lecture 7: Greedy Algorithms: Activity Selection and Fractional Knapack

Read: Sections 16.1 and 16.2 in CLRS.

Greedy Algorithms: In many optimization algorithms a series of selections need to be made. In dynamic programming we saw one way to make these selections. Namely, the optimal solution is described in a recursive manner, and then is computed "bottom-up". Dynamic programming is a powerful technique, but it often leads to algorithms with higher than desired running times. Today we will consider an alternative design technique, called *greedy algorithms*. This method typically leads to simpler and faster algorithms, but it is not as powerful or as widely applicable as dynamic programming. We will give some examples of problems that can be solved by greedy algorithms. (Later in the semester, we will see that this technique can be applied to a number of graph problems as well.) Even when greedy algorithms do not produce the optimal solution, they often provide fast heuristics (nonoptimal solution strategies), are often used in finding good approximations.