UNIT-4
DYNAMIC PROGRAMMING

Dynamic Programming: We begin discussion of an important algorithm design technique, called *dynamic programming* (or DP for short). The technique is among the most powerful for designing algorithms for optimization problems. (This is true for two reasons. Dynamic programming solutions are based on a few common elements. Dynamic programming problems are typically optimization problems (find the minimum or maximum cost solution, subject to various constraints). The technique is related to divide-and-conquer, in the sense that it breaks problems down into smaller problems that it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient. The basic elements that characterize a dynamic programming algorithm are:

Substructure: Decompose your problem into smaller (and hopefully simpler) subproblems. Express the solution of the original problem in terms of solutions for smaller problems.

Table-structure: Store the answers to the subproblems in a table. This is done because subproblem solutions are reused many times.

Bottom-up computation: Combine solutions on smaller subproblems to solve larger subproblems. (Our text also discusses a top-down alternative, called *memoization*.)

The most important question in designing a DP solution to a problem is how to set up the subproblem structure. This is called the *formulation* of the problem. Dynamic programming is not applicable to all optimization problems. There are two important elements that a problem must have in order for DP to be applicable.

Optimal substructure: (Sometimes called the *principle of optimality*.) It states that for the global problem to be solved optimally, each subproblem should be solved optimally. (Not all optimization problems satisfy this. Sometimes it is better to lose a little on one subproblem in order to make a big gain on another.)

Polynomially many subproblems: An important aspect to the efficiency of DP is that the total number of subproblems to be solved should be at most a polynomial number.

Strings: One important area of algorithm design is the study of algorithms for character strings. There are a number of important problems here. Among the most important has to do with efficiently searching for a substring or generally a pattern in large piece of text. (This is what text editors and programs like "grep" do when you perform a search.) In many instances you do not want to find a piece of text exactly, but rather something that is similar. This arises for example in genetics research and in document retrieval on the web. One common method of measuring the degree of similarity between two strings is to compute their longest common subsequence.

Longest Common Subsequence: Let us think of character strings as sequences of characters. Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Z = \langle z_1, z_2, \ldots, z_k \rangle$, we say that Z is a *subsequence* of X if there is a strictly increasing sequence of k indices $\langle i_1, i_2, \ldots, i_k \rangle$ ($1 \leq i_1 < i_2 < \ldots < i_k \leq n$) such that $Z = \langle X_{i_1}, X_{i_2}, \ldots, X_{i_k} \rangle$. For example, let $X = \langle ABRACADABRA \rangle$ and let $Z = \langle AADAA \rangle$, then Z is a subsequence of X.

Given two strings X and Y, the *longest common subsequence* of X and Y is a longest sequence Z that is a subsequence of both X and Y. For example, let $X = \langle ABRACADABRA \rangle$ and let $Y = \langle YABBADABBADOO \rangle$. Then the longest common subsequence is $Z = \langle ABADABA \rangle$. See Fig. 4
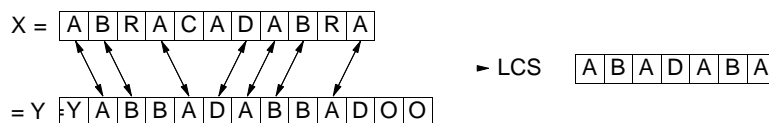

Fig. 4: An example of the LCS of two strings X and Y.

The *Longest Common Subsequence Problem* (LCS) is the following. Given two sequences $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ determine a longest common subsequence. Note that it is not always unique. For example the LCS of $\langle ABC \rangle$ and $\langle BAC \rangle$ is either $\langle AC \rangle$ or $\langle BC \rangle$.

DP Formulation for LCS: The simple brute-force solution to the problem would be to try all possible subsequences from one string, and search for matches in the other string, but this is hopelessly inefficient, since there are an exponential number of possible subsequences.

Instead, we will derive a dynamic programming solution. In typical DP fashion, we need to break the problem into smaller pieces. There are many ways to do this for strings, but it turns out for this problem that considering all pairs of *prefixes* will suffice for us. A *prefix* of a sequence is just an initial string of values, $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. X_0 is the empty sequence.

The idea will be to compute the longest common subsequence for every possible pair of prefixes. Let $c[i, j]$ denote the length of the longest common subsequence of X_i and Y_j. For example, in the above case we have $X_5 = \langle ABRAC \rangle$ and $Y_6 = \langle YABBAD \rangle$. Their longest common subsequence is $\langle ABA \rangle$. Thus, $c[5, 6] = 3$.

Which of the $c[i, j]$ values do we compute? Since we don't know which will lead to the final optimum, we compute all of them. Eventually we are interested in $c[m, n]$ since this will be the LCS of the two entire strings.

The idea is to compute $c[i, j]$ assuming that we already know the values of $c[i', j']$, for $i' \le i$ and $j' \le j$ (but not both equal). Here are the possible cases.

Basis: $c[i, 0] = c[j, 0] = 0$. If either sequence is empty, then the longest common subsequence is empty.

Last characters match: Suppose $x_i = y_j$. For example: Let $X_i = \langle ABCA \rangle$ and let $Y_j = \langle DACA \rangle$. Since both end in A, we claim that the LCS *must* also end in A. (We will leave the proof as an exercise.) Since the A is part of the LCS we may find the overall LCS by removing A from both sequences and taking the LCS of $X_{i-1} = \langle ABC \rangle$ and $Y_{j-1} = \langle DAC \rangle$ which is $\langle AC \rangle$ and then adding A to the end, giving $\langle ACA \rangle$ as the answer. (At first you might object: But how did you know that these two A's matched with each other. The answer is that we don't, but it will not make the LCS any smaller if we do.) This is illustrated at the top of Fig. 5.

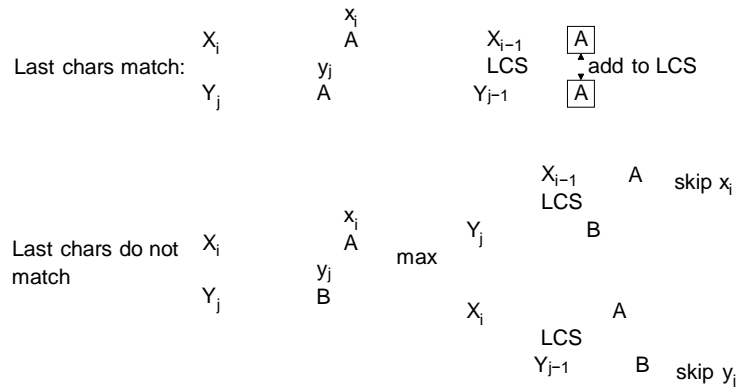$$\text{if } x_i = y_j \text{ then } c[i, j] = c[i - 1, j - 1] + 1$$

Last characters do not match: Suppose that $x_i = y_j$. In this case x_i and y_j cannot both be in the LCS (since they would have to be the last character of the LCS). Thus either x_i is *not* part of the LCS, or y_j is *not* part of the LCS (and possibly *both* are not part of the LCS).

At this point it may be tempting to try to make a "smart" choice. By analyzing the last few characters of X_i and Y_j, perhaps we can figure out which character is best to discard. However, this approach is doomed to failure (and you are strongly encouraged to think about this, since it is a common point of confusion.) Instead, our approach is to take advantage of the fact that we have already precomputed smaller subproblems, and use these results to guide us.

In the first case (x_i is not in the LCS) the LCS of X_i and Y_j is the LCS of X_{i-1} and Y_j, which is $c[i-1, j]$. In the second case (y_j is not in the LCS) the LCS is the LCS of X_i and Y_{j-1} which is $c[i, j-1]$. We do not know which is the case, so we try both and take the one that gives us the longer LCS. This is illustrated at the bottom half of Fig. 5.

$$\text{if } x_i = y_j \text{ then } c[i, j] = \max(c[i-1, j], c[i, j-1])$$

Combining these observations we have the following formulation:

$$c[i, j] = \begin{array}{ll} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i = y_j. \end{array}$$

Implementing the Formulation: The task now is to simply implement this formulation. We concentrate only on computing the maximum *length* of the LCS. Later we will see how to extract the actual sequence. We will store some helpful pointers in a parallel array, $b[0..m, 0..n]$. The code is shown below, and an example is illustrated in Fig. 6

Fig. 6: Longest common subsequence example for the sequences $X = \hbar BAC\,DB\mathrm{i}$ and $Y = \hbar BC\,DB\;\mathrm{i}$. The numeric table entries are the values of $c[i, j]$ and the arrow entries are used in the extraction of the sequence.

_____ Build LCS Table

```
LCS(x[1..m], y[1..n]) {              // compute LCS table
    int c[0..m, 0..n]
    for i = 0 to m                   // init column 0
        c[i,0] = 0; b[i,0] = SKIPX
```

```
    for j = 0 to n                              // init row 0
        c[0,j] = 0; b[0,j] = SKIPY
    for i = 1 to m                              // fill rest of table
        for j = 1 to n
            if (x[i] == y[j])                   // take X[i] (Y[j]) for LCS
                c[i,j] = c[i-1,j-1]+1;   b[i,j] = addXY
            else if (c[i-1,j] >= c[i,j-1])      // X[i] not in LCS
                c[i,j] = c[i-1,j];       b[i,j] = skipX
            else                                // Y[j] not in LCS
                c[i,j] = c[i,j-1];       b[i,j] = skipY
    return c[m,n]                               // return length of LCS
}
```

Extracting the LCS

```
getLCS(x[1..m], y[1..n], b[0..m,0..n]) {
    LCSstring = empty string
    i = m; j = n                                // start at lower right
    while(i != 0 && j != 0)                     // go until upper left
        switch b[i,j]
            case addXY:                         // add X[i] (=Y[j])
                add x[i] (or equivalently y[j]) to front of LCSstring
                i--;   j--;   break
            case skipX: i--;  break             // skip X[i]
            case skipY: j--;  break             // skip Y[j]
    return LCSstring
}
```

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 $b[i, j] = add_{XY}$ 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 ($-$). If $b[i, j] = skip_X$, 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 $b[i, j] = skip_Y$, 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 $1 \le i \le p$ and $1 \le j \le r$,

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

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, ..., 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 $1 \le i \le j \le n$, 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, $i \le k < j$, 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 $p_{i-1} p_k p_j$. This suggests the following recursive rule for computing $m[i, j]$.

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 $j \le n$, this means that $i + L - 1 \le n$, or in other words, $i \le n - L + 1$. 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

```
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 ⟨5, 4, 6, 2, 7⟩ 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 $((A_1 (A_2 A_3))A_4)$.

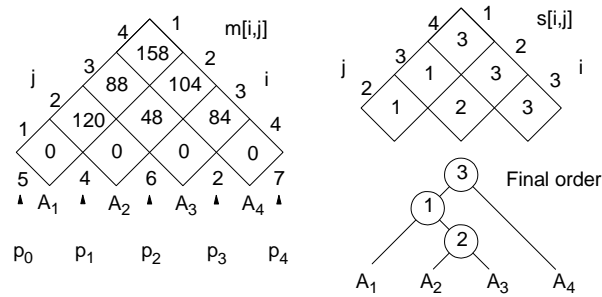# Lecture 6: Dynamic Programming: Minimum Weight Triangulation

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.

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 $\overline{v_i v_j}$ 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.)
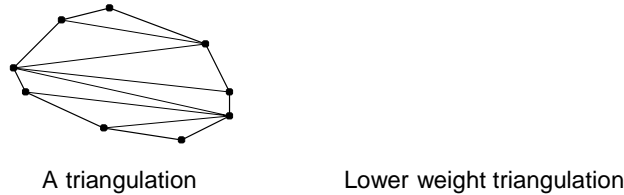

A triangulation   Lower weight triangulation

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 $v_i v_j$.

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 $\overrightarrow{v_i v_j}$, 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 $v_i v_j$. 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
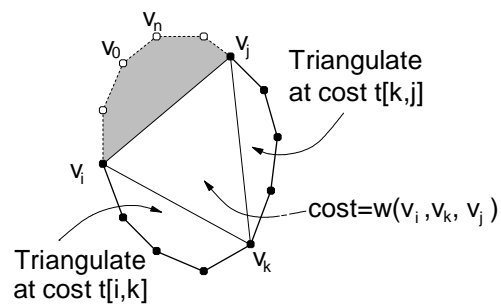
Fig. 12: Triangulations and tree structure.

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

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 triangulati
on.