Advanced Algorithms

(Feodor F. Dragan)
Department of Computer Science
Kent State University

What is a computer program exactly?

Input

Some mysterious processing

Output

Programs = Data Structures + **Algorithms**

CHAPTER 15

Dynamic Programming

• We begin discussion of an important algorithm design technique, called *dynamic programming* (or *DP* for short).

• It is not a particular algorithm, it is a metatechnique.

• Programming =“tableau method”, not writing a code.

• 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 typical optimization problems (find the minimum or maximum cost solution, subject to various constraints.

• The technique is related to divideandconquer, 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 divideandconquer 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. (Unlike divideandconquer problems, it is not usually sufficient to consider one decomposition, but many different ones.)

- **Tablestructure:** Store the answers to the subproblems in a table. This is done because (typically) subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem.

- **Bottomup computation:** Combine solutions on smaller subproblems to solve larger subproblems, and eventually to arrive at a solution to the complete problem. (Our text also discusses a topdown 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:** Optimal solution to the problem contains within it optimal solutions to subproblems. This is sometimes called the *principle of optimality*. It states that for the global problem to be solved optimally, each subproblem should be solved optimally.

• **Polynomially many subproblems:** An important aspect to the efficiency of DP is that the total number of distinct 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 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. Genetic codes are stored as long DNA molecules. The DNA strands consists of a string of molecules of four basic types: C, G, T, A. Exact matches rarely occur in biology because of small changes in DNA replication. For this reason, it is of interest to compute similarities between strings that do not match exactly.
- One common method of measuring the degree of similarity between two strings is to compute their longest common subsequence.

Longest Common Subsequence

- Given two sequences of characters $X = \langle x_1, x_2, \ldots, x_m \rangle$ 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 $\langle i_1, i_2, \ldots, i_k \rangle$ $(1 \le i_1 < i_2 < \ldots < i_k \le m)$ such that $Z = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$.
- In $X = \langle ABRACADABRA \rangle$, $Z = \langle AADAA \rangle$, $T = \langle ADAD \rangle$ *Z* is a subsequence of *X*, *T* is not.
- Given two strings *X* and *Y*, the longest common subsequence (LCS) of *X* and *Y* is a longest sequence *Z* which is both a subsequence of *X* and *Y*.
- The LCS of $X = \langle ABRACADABRA \rangle$ and $Y = \langle YABBADABBADOO \rangle$ is $Z = \langle ABADABA \rangle$.
- The ***longest common subsequence problem*** is the following. Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, determine a LCS of *X* and *Y*.
- It is not always unique. LCS of *(ABC)* and *(BAC)* is either *(AC)* or *(BC)*.
- **Brute-force algorithm**: (very inefficient)
  - for every subsequence of *X* check if it is in *Y*.
  - $O(n2^m)$ time (2^m subseqences of *X*, *O(n)* for scanning *Y* with *X*-subseq.)

Dynamic Programming Approach

- 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 (➔*polynomial* (how many?) number of subproblems).

- Let *c[i,j]* denote the length of the longest common subsequence of X_i and Y_j. Eventually we are interested in *c[m, n]* since this will be the LCS of the two entire strings (where is the *principle of optimality*?).

- 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).

- We begin with some observations.

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

Dynamic Programming Approach (cont.)

Last characters match: Suppose $x_i = y_j$. Since both end in x_i, we claim that the LCS must also end in x_i. (We will leave the proof as an exercise.) Since the x_i is part of the LCS we may find the overall LCS by removing x_i from both sequences and taking the LCS of X_{i-1} and Y_{j-1}, and then adding x_i to the end. Thus

$$if \;\; x_i = y_j \;\; then \;\; c[i,j] = c[i-1, j-1] + 1.$$

Last characters do not match: Suppose $x_i \neq y_j$. In this case x_i and y_j cannot both be in 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).

In the first case 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 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.

Thus, $if \;\; x_i \neq y_j \;\; then \;\; c[i,j] = \max(c[i-1, j], c[i, j-1]).$

Implementing the Rule

Thus,

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

The task now is to simply implement this rule. 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.

```
LCS (char x[1..m], char y[1..n] ) {                    // compute LCS table
    int c[0..m, 0..n]; char b[0..m,0..n];
    for i=0 to m do {c[i,0]=0; b[i,0]=SKIPX;}   // initialize column 0
    for j=0 to n do {c[0,j]=0; b[0,j]=SKIPY;}    // initialize row 0
    for i=1 to m do {                                  // fill rest of table
       for j=1 to n do {
          if (x[i]==y[j])   {c[i,j]=c[i-1,j-1]+1; b[i,j]=ADDXY; }  // take x[i] (=y[j]) for LCS
          else if (c[i-1,j]>=c[i,j-1]) {                                  // x[i] is not in LCS
                  c[i,j]=c[i-1,j]; b[i,j]=SKIPX;}
               else {c[i,j]=c[i,j-1]; b[i,j]=SKIPY;}  }  }          // y[j] is not in LCS
    return c[m,n];                  }                                    // return length of LCS
```

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] = ADDXY 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]=SKIPX, then we know that x[i] is not in the LCS, and we skip it and go to b[i-1,j] (↑). Similarly, if b[i, j] = SKIPY, 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 (←). Following these back pointers, and outputting a character with each diagonal move gives the final subsequence.

```
getLCS (char x[1..m], char y[1..n], char b[0..m,0..n]) {          // extract the LCS
    LCS=empty string; i=m; j=n;                    // start at lower right
    while (i !=0 && j !=0) {                       //  go untill upper left
        switch  b[i,j] {
            case ADDXY:
                add x[i] (or equivalently y[j]) to front of LCS
                i--; j--; break
            case SKIPX: i--; break                 //  skip x[i]
            case SKIPY: j--; break }  }            //  skip y[j]
    return LCS  }
```

Time and Space Bounds and an Example

• The running time of the algorithm LCS is clearly *O(mn)* since there are two nested loops with *m* and *n* iterations, respectively. The running time of the algorithm getLCS is *O(m+n)*. Both algorithms use *O(mn)* space.

		Y:	0	1	2	3	4 = n
				B	D	C	B
0			0	0	0	0	0
1	B		0	1	1	1	1
2	A		0	1	1	1	1
3	C		0	1	1	2	2
4	D		0	1	2	2	2
m= 5	B		0	1	2	2	3

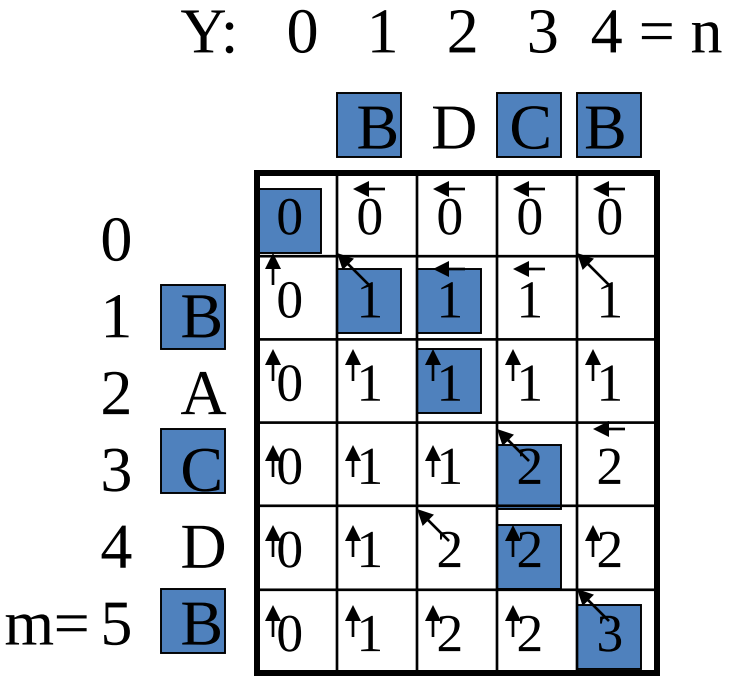
X: (row header)

LCS Length Table

X=BACDB

Y=BDCB

LCS=BCB



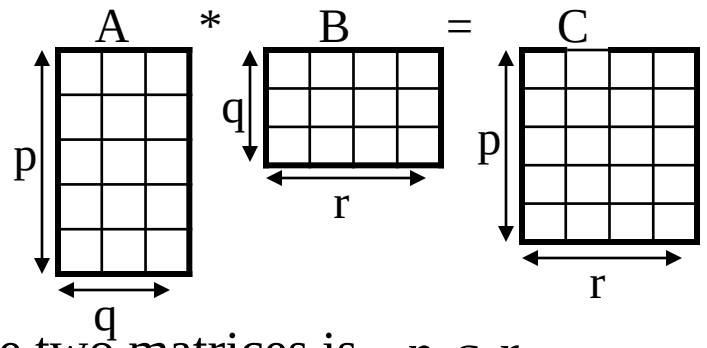
with back pointers included

Longest common subsequence example.

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 (non-square) 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 $1 \le j \le r$,

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



- There are $p \cdot r$ total entries in C and each takes $O(q)$ time to compute, thus the total time (e.g. number of multiplications) to multiply these two matrices is $p \cdot q \cdot r$.

Problem Formulation

• 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:

$$A_1 \text{ be } 5\times 4,\ A_2 \text{ be } 4\times 6,\ A_3 \text{ be } 6\times 2.$$

• Then

$$\text{multCost}[((A_1\ A_2)A_3)] = (5\cdot 4\cdot 6) + (5\cdot 6\cdot 2) = 180;$$
$$\text{multCost}[(A_1(\ A_2 A_3))] = (4\cdot 6\cdot 2) + (5\cdot 4\cdot 2) = 88.$$

• 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 $p_0, p_1, p_2, \ldots, p_n$ where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication that minimizes the number of operations.

• *Important note:* This algorithm does not perform the multiplications, it just figures out 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 item, 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 *first* item, just after the *second* item, etc., and just after the *(n-1)*st item.
- When we split just after the *k*th 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* times *R*. This suggests the following recurrence for *P(n)*, the number of different ways of parenthesizing *n* items:

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

- This is related to 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 *n*th Catalan number;

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

- Applying Stirling's formula, 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*.

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.
- For convenience we can write $A_{i..j}$ to be 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.
- 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 $k, 1 \le k \le n-1, \quad A_{1..n} = A_{1..k} \cdot A_{k+1..n}$.
- 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 by recursively 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, it turns out that it does not work in this case. Instead, (as in almost all dynamic programming solutions), we will do the dumb thing. We will consider all possible values of k, and take the best of them.
- Notice that this 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 bottomup 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 $i \neq j$, then we are asking about the product $A_{i..j}$. This can be split by considering each k, $1 \le k \le n-1$, as $A_{i..k} \cdot A_{k+1..j}$.

The optimum time to compute $A_{i..k}$ is m[i, k], and the optimum time to compute $A_{k+1..j}$ is m[k + 1, j]. We may assume that these values have been computed previously and 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 $p_{i-1} \cdot p_k \cdot p_j$.

- This suggests the following recursive rule for computing m[i, j].

$$m[i,i] = 0,$$
$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) \quad for \quad i < j.$$

Implementing the Rule

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 will 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 subchain. Let $L = j-i+1$ denote the length of the subchain being multiplied. The subchains of length 1 (m[i, i]) are trivial. Then we build up by computing the subchains of length 2, 3, …, *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 $n-L+1$ (to keep j in bounds).

```
Matrix-Chain(array p[1..n], int 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); }
```

- The running time of this procedure is $\Theta(n^3)$. (3 nested loops, each iterates $\le$n times.)

Extracting Optimum Sequence

The array s[i, j] can be used to extract the actual sequence. To extract it is a fairly easy extension. The basic idea is to keep in s[i,j] a split marker indicating what the best split is, that is, what value of *k* leads to the minimum value of m[i, j]. s[i, j] = *k* 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 $j > i$.

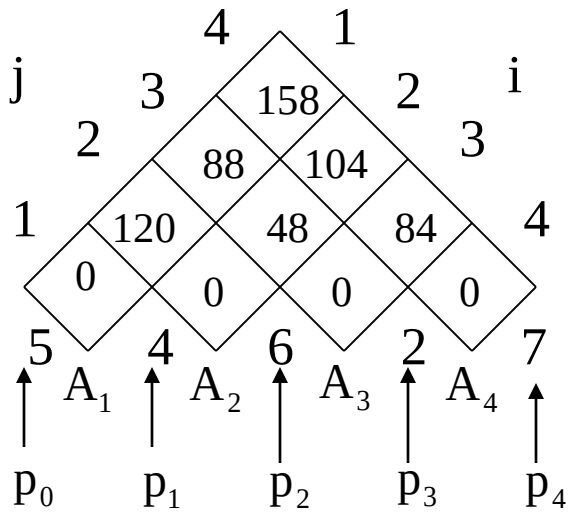
The actual multiplication algorithm uses the s[i, j] value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices A[1..n], and that s[i, j] is global to this recursive procedure. The procedure returns a matrix.

```
Mult(i,j) {
    if (i ==j)  return A[i];              // basic case
    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
        }
    }
```

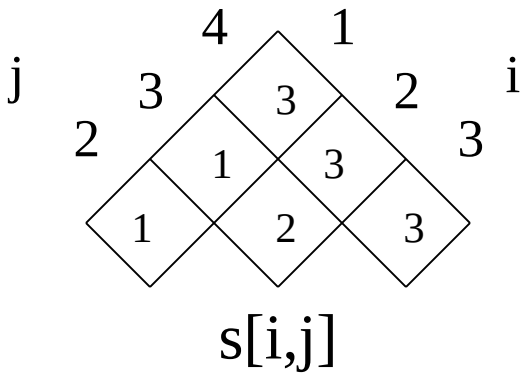
Chain Matrix Multiplication Example

• 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 $A_1(5\times4)$ times $A_2(4\times6)$ times $A_3(6\times2)$ times $A_4(2\times7)$.

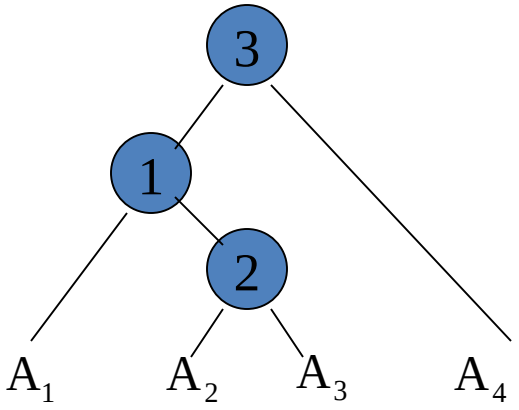
• The optimal sequence is $((A_1(A_2A_3))A_4)$.



m[i,j]



s[i,j]



Recursive Implementation

• We have described dynamic programming as a method that involves the ``bottom-up" computation of a table. However, the recursive formulations that we have derived have been set up in a ``topdown" manner. Must the computation proceed bottomup?

• Consider the following recursive implementation of the chainmatrix multiplication algorithm. The call *RecMatrixChain(p, i, j)* computes and returns the value of m[i, j]. The initial call is *RecMatrixChain(p, 1, n).* We only consider the cost here.

```
Rec-Matrix-Chain(array p, int i, int j) {
    if  (i==j) m[i,i]=0;                                  // basic case
    else {
        m[i,j]=infinity;                                  // initialize
        for k=i to j-1 do {                               // try all splits
            cost=Rec-Matrix-Chain(p,i,k)+
                 Rec-Matrix-Chain(p,k+1,j)+p[i-1]*p[k]*p[j];
            if (cost<m[i,j]) m[i,j]=cost;  } }            // update if better
    return m[i,j]; }                                      // return final cost
```

• (Note that the table m[1..n, 1..n] is not really needed. We show it just to make the connection with the earlier version clearer.) This version of the procedure certainly looks much simpler, and more closely resembles the recursive formulation that we gave previously for this problem. So, what is wrong with this?

Recursive Implementation (cont.)

- The answer is the running time is much higher than the $\Theta(n^3)$ algorithm that we gave before. In fact, we will see that its running time is exponential in n. This is unacceptably slow.
- Let $T(n)$ denote the running time of this algorithm on a sequence of matrices of length n. (That is, n = j –i+1.)
- If $i=j$, then we have a sequence of length 1, and the time is $\Theta(1)$. Otherwise, we do $\Theta(1)$ work and then consider all possible ways of splitting the sequence of length n into two sequences, one of length k and the other of length $n-k$, and invoke the procedure recursively on each one. So, we get the following recurrence, defined for $n \geq 1$. (We have replaced the $\Theta(1)$ ‘s with the constant 1.)

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

Claim: $T(n) \geq 2^{n-1}$

Proof: By induction on n. Clearly, this is true for $n=1$. The induction hypothesis is that $T(m) \geq 2^{m-1}$ for all $m<n$. Using this we have

$$T(n) = 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k)) \geq 1 + \sum_{k=1}^{n-1} T(k)$$

- ignore the term T(n-k),
- apply the induction, and

$$\geq 1 + \sum_{k=1}^{n-1} 2^{k-1} = 1 + \sum_{k=0}^{n-2} 2^k = 1 + (2^{n-1} - 1) = 2^{n-1}.$$

- apply the formula for the geometric series.

- Why is this so much worse than the dynamic programming version? If you ``unravel" the recursive calls on a reasonably long example, you will see that the procedure is called repeatedly with the same arguments. The bottomup version evaluates each entry exactly once.

Memoization

• Is it possible to retain the nice topdown structure of the recursive solution, while keeping the same $O(n^3)$ efficiency of the bottomup version? The answer is yes, through a technique called ***memoization.***

• Here is the idea. Let's reconsider the function *RecMatrixChain()* given above. It's job is to compute m[i, j], and return its value. As noted above, the main problem with the procedure is that it recomputes the same entries over and over. So, we will fix this by allowing the procedure to compute each entry exactly once. One way to do this is to initialize every entry to some special value (e.g. UNDEFINED). Once an entries value has been computed, it is never recomputed.

```
Mem-Matrix-Chain(array p, int i, int j) {
    if   (m[i,j] != UNDEFINED)   return m[i,j];          // already defined
    else if  (i==j) m[i,i]=0;                            // basic case
    else {
        m[i,j]=infinity;                                 // initialize
        for k=i to j-1 do {                              // try all splits
            cost=Mem-Matrix-Chain(p,i,k)+
                Mem-Matrix-Chain(p,k+1,j)+p[i-1]*p[k]*p[j];
            if (cost<m[i,j]) m[i,j]=cost;  } }            // update if better
    return m[i,j]; }                                     // return final cost
```

Memoization (cont.)

• This version runs in $O(n^3)$ time. Intuitively, this is because each of the $O(n^2)$ table entries is only computed once, and the work needed to compute one table entry (most of it in the forloop) is at most ***O(n).***

• • •

• Memoization is not usually used in practice, since it is generally slower than the bottomup method.

• However, in some DP problems, many of the table entries are simply not needed, and so bottomup computation may compute entries that are never needed.

• In these cases memoization may be a good idea.

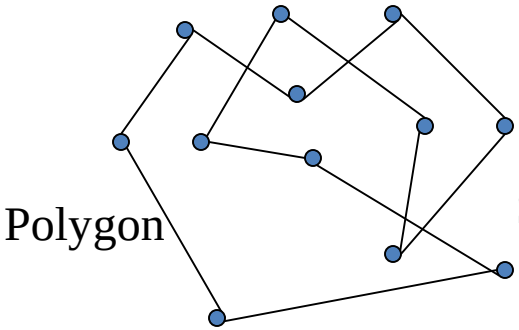
• If you know that most of the table will not be needed, here is a way to save space.

• Rather than storing the whole table explicitly as an array, you can store the ``defined" entries of the table in a hash table, using the index pair *(i, j)* as the hash key.

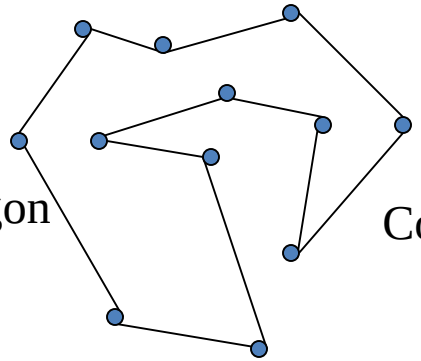
• See Chapter 11 in CLRS for more information on hashing.

Polygons and Triangulations

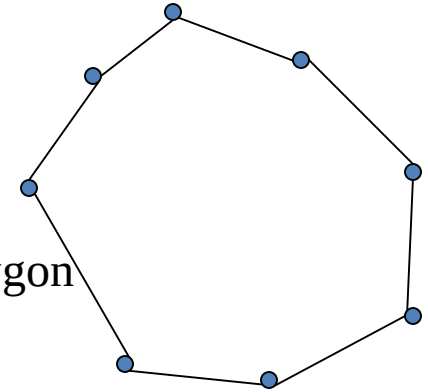
- Let's consider a geometric problem that outwardly appears to be quite different from chainmatrix multiplication, but actually has remarkable similarities.
- We begin with a number of definitions. Define a *polygon* to be a piecewise linear closed curve in the plane. In other words, we form a cycle 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.



Polygon



Simple Polygon



Convex Polygon

- Given a convex polygon, we assume that its vertices are labeled in counterclockwise order $P = \langle v_0, v_1, v_2, \ldots, v_{n-1} \rangle$. We assume that indexing is done modulo n, so $v_0 = v_n$. This polygon has n sides, $\overline{v_{i-1}v_i}$.

Triangulations (cont.)

• Given two non-adjacent vertices v_i, v_j $(i < j)$, 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 lies 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 is a maximal set T of pairwise non-crossing 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.

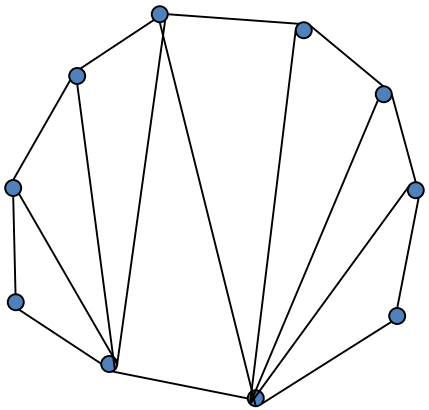
• Triangulations are of interest for a number of reasons. Many geometric algorithms operate by first decomposing a complex polygonal shape into rectangles. Then an algorithm can be applied triangle by triangle.

• Define the *dual graph* of the triangulation to be a graph whose vertices are the triangles, and in which two vertices share a side (are *adjacent*) if the two triangles share a common chord.

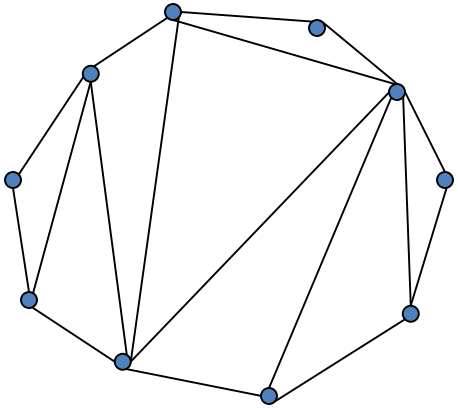
• Observe that the dual graph is a tree. Hence algorithms for traversing trees can be used for traversing the triangles of a triangulation.

Minimum-Weight Convex Polygon Triangulation

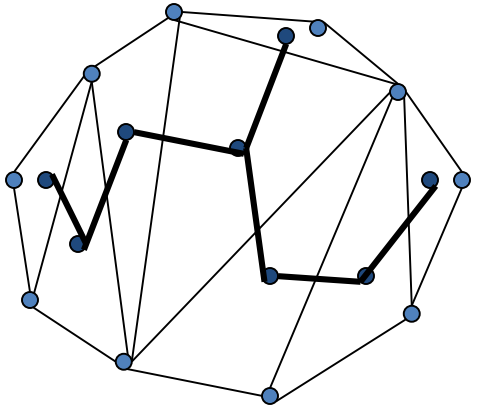
- In general, given a convex polygon, there are many possible triangulations. In fact, the number is exponential in *n*, the number of sides. Which triangulation is the ``best"?
- There are many criteria that are used depending on the application. One criterion is to imagine that you must ``pay" for the ink you use in drawing the triangulation, and you want to minimize the amount of ink you use. This suggests the following interesting problem.
- ***Minimumweight convex polygon triangulation:*** Given a convex polygon, determine the triangulation that minimizes the sum of the perimeters of its triangles.



A triangulation



Another triangulation



The dual tree

- 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 $v_i v_j$.

Correspondence 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.

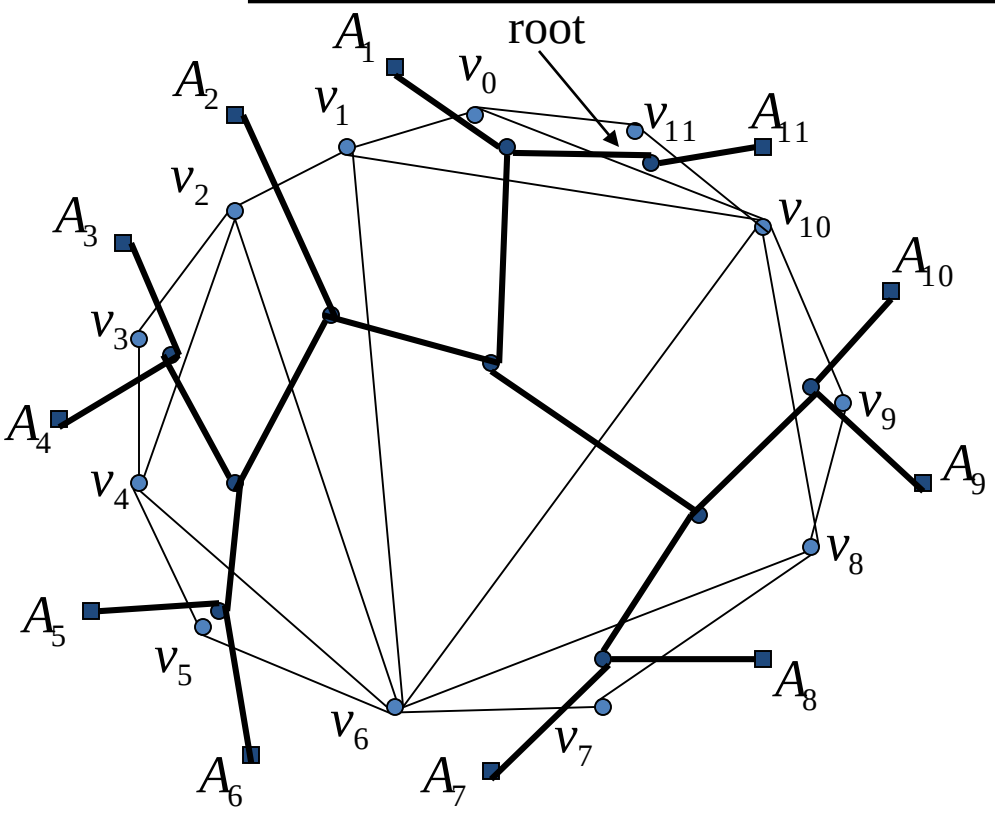
• To see that there is a similar correspondence here, consider an (n + 1)sided convex polygon $P = \langle v_0, v_1, v_2, \ldots, v_{n-1}, v_n \rangle$, and fix one side of the polygon (say $\overline{v_0 v_n}$).

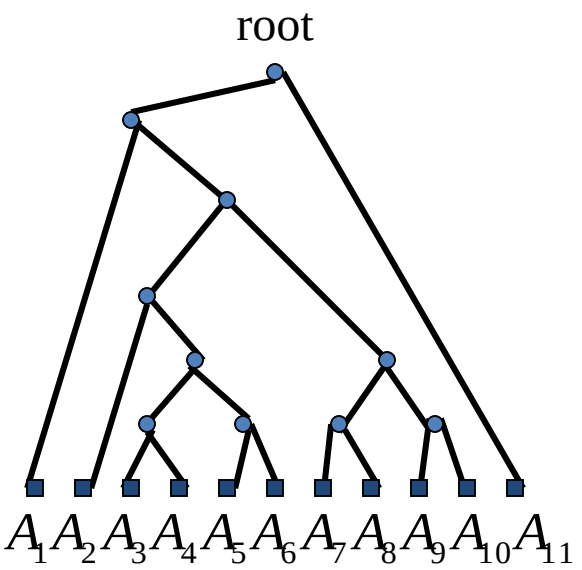
• Now consider a rooted binary tree whose root node is the triangle containing side $\overline{v_0 v_n}$, whose internal nodes are the nodes of the dual tree, and whose leaves correspond to the remaining sides of the polygon.

• Observe that partitioning the polygon into triangles is equivalent to a binary tree with *n* leaves, and vice versa. This is illustrated in the figure below.

• 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.

Correspondence to Binary Trees (cont.)





- 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 that 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.

Dynamic Programming Solution

- Let us consider an (n+1)sided polygon $P = \langle v_0, v_1, v_2, \ldots, v_{n-1}, v_n \rangle$.

- To derive a DP formulation we need to define a set of subproblems from which we can derive the optimum solution.

- For $1 \le i \le j \le n$, define t[i, j] to be the weight of the minimum weight triangulation for the subpolygon $\langle v_{i-1}, v_i, \ldots, v_j \rangle$, assuming that the chord $\overline{v_{i-1}v_j}$ is already present in the triangulation. (The reason that we start with v_{i-1} rather than v_i is to keep the structure as similar as possible to the chain-matrix multiplication problem. We will see this below.)

- 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 will be t[1, n].

- As a ***basis case***, we define the weight of the trivial ``2sided polygon'' to be zero, implying that t[i, i] (the line segment $\overline{v_{i-1}v_i}$) to be 0.

- In ***general***, to compute t[i,j], consider the subpolygon $\langle v_{i-1}, v_i, \ldots, v_j \rangle$, where $i<j$. One of the chords of this polygon is the side $\overline{v_{i-1}v_j}$. We may split this subpolygon by introducing a triangle whose base is this chord, and whose third vertex is any vertex v_k, where $i \le k \le j-1$.

Dynamic Programming Solution (cont.)

- This subdivides the polygon into the subpolygons $\langle v_{i-1}, v_i, \ldots, v_k \rangle$ and $\langle v_k, v_{k+1}, \ldots, v_j \rangle$ whose minimum weight are already known to us as t[i,k] and t[k+1,j].
- In addition we should consider the weight of newly added triangle $\Delta v_{i-1} v_k v_j$.
- Thus, we have the following recursive rule:

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

- Note that this has exactly the same structure as the recursive definition used in the chain-matrix multiplication algorithms.
- The same $\Theta(n^3)$ algorithm can be applied with only minor changes.

• • •

Homework #1: See on the class' web-page.