Lecture VII
DYNAMIC PROGRAMMING

We introduce an algorithmic paradigm called **dynamic programming**. It was popularized by Richard Bellman, circa 1954. The word "programming" here is the same term as found in "linear programming", and has the connotation of a systematic method for solving problems. The term is even identified[1] with the filling-in of entries in a table. The semantic shift from this to our contemporary understanding of the word "programming" is an indication of the progress in the field of computation.

Linear Bin Packing, again. Recall the linear bin packing problem from Chapter V. The input is (M, w) where $w = (w_1, \ldots, w_m)$ is a sequence of numbers satisfying $w_i \leq M$. We want to partition w into the minimum number of subgroups of the from $w(i, j) = (w_{i+1}, \ldots, w_j)$, $0 \leq i \leq j \leq n$, of size $\sum_{\ell=i+1}^{j} w_\ell$ at most M. We now allow the w_i's to be negative. The greedy algorithm breaks down in this case. However, it is easy to give an $O(n^2)$ algorithm. But first, we must generalize the problem so that, when solving the problem (M, w), we also solve the subproblems $(M, w(0, i))$ for all $i = 1, \ldots, n$. For instance, after solving $(M, (w_1, w_2, w_3, w_4))$, it is assumed that the solutions to $(M, (w_1, w_2, w_3))$, $(M, (w_1, w_2))$ and $(M, (w_1))$ are also known.

Now it is easy to see that from a solution of $(M, w(0, n-1))$, we can solve (M, w) in $O(n)$ time. Thus the overall complexity is $T(n) = O(n) + T(n-1) = O(n^2)$.

This is a typical dynamic program solution. What features do we see here? First, from a standard problem (M, w), we generated a polynomial number of subproblems $(M, w(0, i))$, $i = 1, \ldots, n$. Second, from the solutions to all the smaller subproblems, we can reconstruct the solution to the main problem in polynomial time (in this case, $O(n)$ time).

§1. Longest Common Subsequence Problem

We consider a simple problem on strings. Throughout this chapter, we fix some alphabet Σ and strings are just elements of Σ^*. Let $X = x_1 x_2 \cdots x_m$ be a string where the x_i's are **letters** (or symbols) from some alphabet. The **length** of X is m, denoted $|X|$. The **empty string** is denoted ϵ and it has length $|\epsilon| = 0$. The ith letter of X is denoted $X[i] = x_i$ $(i = 1, \ldots, m)$. A **subsequence** $Z = z_1 z_2 \cdots z_k$ of X is a string such that for some

$$1 \leq i_1 < i_2 < \cdots < i_k \leq m$$

we have $Z[\ell] = X[i_\ell]$ for all $\ell = 1, \ldots, k$. For example, `ln`, `lg` and `log` are subsequences of the string `long`.

A related concept is the notion of a "substring". The above subsequence Z is a **substring** when $i_j = i_1 + j - 1$ for all $j = 1, \ldots, k$. For instance, `on` and `g` are substrings of `long` but `ln`, `lg` and `log` are not. Thus, subsequences are substrings but the converse may not be true.

A **common subsequence** of X, Y is a string $Z = z_1 z_2 \cdots z_k$ that is a subsequence of both X and Y. We further call Z a **longest common subsequence** if its length $|Z| = k$ is maximum among all common subsequences of X and Y. Since the longest common subsequence may not be unique, let $LCS(X, Y)$ denote the set of longest common subsequences of X, Y. Remark that there is multiset form of $LCS(X, Y)$, but that is treated in the exercises. Also, let $\lambda(X, Y) := |LCS(X, Y)|$ and $L(X, Y)$ be the length of any

[1] Such tables are sometimes filled out by deploying a row of human operators, each assigned to filling in some specific table entries and to pass on the partially-filled table to the next person.

$Z \in LCS(X,Y)$. Note that $\lambda(X,Y) \geq 1$ since "at worst", $LCS(X,Y)$ is the singleton comprising the empty string.

For example, if

$$X = \texttt{longest}, \qquad Y = \texttt{length} \tag{1}$$

then $LCS(X,Y) = \{\texttt{lngt}\}$, $\lambda(X,Y) = 1$ and $L(X,Y) = 4$.

There are several versions of the **longest common subsequence (LCS) problem**. Given two strings

$$X = x_1 x_2 \cdots x_m, \quad Y = y_1 y_2 \cdots y_n,$$

the problem is to compute (respectively) one of the following:

- (Length version) The length $L(X,Y)$ of the longest common subsequence;
- (Instance version) Any longest common subsequence $Z \in LCS(X,Y)$;
- (Cardinality version) The size $\lambda(X,Y)$ of $LCS(X,Y)$.
- (Set version) The set $LCS(X,Y)$.

We will mainly focus on the first two versions. The last version can be exponential if members of the set $LCS(X,Y)$ is explicitly written out; we may be willing to accept some reasonably explicit[2] representation of $LCS(X,Y)$. We will consider representations of $LCS(X,Y)$ below.

A brute force solution to the length version of the LCS problem would be to list all subsequences of length ℓ (for $\ell = m, m-1, m-2, \ldots, 2, 1$) of X, and for each subsequence to check if it is also a subsequence of Y. This is an exponential algorithm since X has 2^m subsequences.

The following is a simple but crucial observation. Let us write X' for the prefix of X obtained by dropping the last symbol of X. This notation assumes $|X| > 0$ so that $|X'| = |X| - 1$. It is easy to verify the following formula for $L(X,Y)$:

$$L(X,Y) = \begin{cases} 0 & \text{if } \ m = 0 \text{ or } \ n = 0 \\ 1 + L(X',Y') & \text{if } \ x_m = y_n \\ \max\{L(X',Y), L(X,Y')\} & \text{if } \ x_m \neq y_n \end{cases} \tag{2}$$

There is a subtlety in this formula when $x_m = y_n$. The "obvious" formula for this case is

$$L(X,Y) = \max\{1 + L(X',Y'), L(X',Y), L(X,Y')\}.$$

The right hand side is simplified to the form in (2) because

$$L(X',Y) \leq 1 + L(X',Y').$$

There is a similar inequality involving $L(X,Y')$. The formula (2) constitutes the "dynamic programming principle" for the LCS problem – it expresses the solution for inputs of size $N = |X| + |Y|$ in terms of the solution for inputs of sizes $\leq N - 1$. We will discuss the dynamic programming principle in §4. There is a corresponding formula for $LCS(X,Y)$:

$$LCS(X,Y) = \begin{cases} \{\epsilon\} & \text{if } \ m = 0 \text{ or } \ n = 0 \\ LCS(X',Y')x_m & \text{if } \ x_m = y_n \\ LCS(X',Y) & \text{if } \ x_m \neq y_n \quad \text{and} \quad L(X',Y) > L(X,Y') \\ LCS(X,Y') & \text{if } \ x_m \neq y_n \quad \text{and} \quad L(X,Y') > L(X',Y) \\ LCS(X,Y') \cup LCS(X',Y) & \text{if } \ x_m \neq y_n \quad \text{and} \quad L(X,Y') = L(X',Y). \end{cases} \tag{3}$$

[2]Of course, the pair (X,Y) itself is an implicit representation of $LCS(X,Y)$. Hence "reasonably explicit" means that we can confirm membership in $LCS(X,Y)$, or enumerate the members of $LCS(X,Y)$, in a reasonably efficient manner.

For any string X and natural number $i \geq 0$, let X_i denote the prefix of X of length i (if $i > |X|$, let $X_i = X$). The dynamic programming principle for $LCS(X, Y)$ suggests studying the following collection of subproblems:

$$L(X_i, Y_j), \qquad (i = 0, \ldots, m; j = 0, \ldots, n).$$

There are $O(mn)$ such subproblems. Note that X_0 is the empty string ϵ, so that

$$LCS(X_0, Y_j) = \{\epsilon\}, \qquad L(X_0, Y_j) = 0. \tag{4}$$

Simplification: From the above recurrence equations, we see that the flow of control of our algorithm for $LCS(X, Y)$ is driven by the function $L(X, Y)$. In fact, both equations (2) and (3) share a common flow of control. Therefore, if we have an algorithm for $L(X, Y)$, it will be easy to insert some simple modifications to derive an algorithm for $LCS(X, Y)$. Hence, we will first develop the algorithm for the simpler problem of $L(X, Y)$. Such a simplifying step is typical of solutions that exploit the dynamic programming approach.

Here now is the dynamic programming solution for $L(X, Y)$. The algorithm sets up an $(1 + m) \times (1 + n)$ matrix $L[0..m, 0..n]$ where $L[i, j]$ is to store the value $L(X_i, Y_j)$. We fill in the entries of this matrix as follows. First fill in the 0th column and 0th row with zeros, as noted in (4). Now fill in successive rows, from left to right, using equation (2) above.

In illustration, we extend[3] the example (1) to the strings $X =$ `lengthen` and $Y =$ `elongate`:

		e	l	o	n	g	a	t	e
	0	0	0	0	0	0	0	0	0
l	0								
e	0								
n	0				x				
g	0					$1 + x$			
t	0								
h	0								
e	0								u
n	0							v	$max(u, v)$

We illustrate the formula (3) in action in two entries: the entry corresponding to the 'g'-row and 'g'-column is filled with $1+x$ where x is the entry in the previous row and column. The entry corresponding to last row and last column is $\max(u, v)$ where u and v are the two adjacent entries. [It turns out that $x = 2, u = 5, v = 4$.] The reader may verify that $L(X, Y) = 5$ and $LCS(X, Y) = \{$`lngte`, `engte`$\}$ in this example. We leave as an exercise to program this algorithm in your favorite language.

Complexity Analysis. Each entry is filled in constant time. Thus the overall time complexity is $\Theta(mn)$. The space is also $\Theta(mn)$. Actually, it is not hard to see that the space $O(\min\{m, n\})$ suffices.

Extensions. We said that it should be easy to modify the code for computing $L(X, Y)$ so that you can find members of $LCS(X, Y)$. Let us use this observation: each entry $L[i, j]$ derives its values from one of

[3] No pun in-tended.

the values in $L[i-1,j], L[i,j-1], L[i-1,j-1]$. Keep track of this information by using another matrix $M[1..m, 1..n]$ with the following entries. If $x_i = y_j$ then $M[i,j] = 0$, and if $x_i \neq y_j$ then

$$M[i,j] = \begin{cases} 1 & \text{if } L[i,j] = L[i-1,j] > L[i,j-1], \\ 2 & \text{if } L[i,j] = L[i,j-1] > L[i-1,j], \\ 3 & \text{if } L[i,j] = L[i-1,j] = L[i,j-1]. \end{cases}$$

The matrix can be filled in as we fill in the matrix L. Subsequently, we can easily use M find a member or enumerate all members of $LCS(X,Y)$. Moreover, given any Z, we can check if $Z \in LCS(X,Y)$ (how?). In this sense, we may say M (together with X, Y) is a reasonably explicit representation of $LCS(X,Y)$. Basically, what we construct is a compressed trie for $LCS(X,Y)$ However, this trie is really a dag (directed acyclic graph) of size $O(mn)$.

Improvements. There are various ways to improve the basic algorithm. One is to exploit knowledge about the alphabet. For instance, Paterson and Masek gives an algorithm with $\Theta(mn/\log(\min(m,n)))$ time when the alphabet of the strings is bounded.

Our algorithm fill in the entries of the matrix L in a bottom-up fashion. We can also fill them in a top-down fashion. Namely, we begin by trying to fill the entry $L[m,n]$. There are 2 possibilities: (i) If $x_m = y_n$, we must recursively fill in $L[m-1,n-1]$ and then use this value to fill in $L[m,n]$. (ii) Otherwise, we must recursively fill in $L[m-1,n]$ and $L[m,n-1]$ first. In general, while trying to fill in $L[i,j]$ we must first check if the entry is already filled in (why?). If so, we can return the value at once. Clearly, this approach may lead to much fewer than mn entries being looked at. We leave the details to an exercise.

Applications. Computational problems on strings has been studied since the early days of computer science. One primary motivation was their application in text editors. For instance, the problem of finding a pattern in a larger string is a basic task in text editors. The advent of computational genomics in the 1990's has brought new attention to problems on strings. To understand this application, we need to recall that the fundamental unit of study here is the DNA, where a DNA can be regarded as a string over an alphabet of four letters: *A, C, G, T*. These corresponds to the four bases: adenine, guanine, cytosine and thymine. DNA's can be used to identify species as well as individuals. More generally, the variations across species can be used as a basis for measuring their genetic similarity. The LCS problem is one of many that has been formulated to measure such similarities. We will see another formulation in the next section.

Exercises

Exercise 1.1: Find the set $LCS(X,Y)$ where

$$X = 00110011, \quad Y = 10100101.$$

Show your working (the matrix) and justify your method of extracting the longest common subsequences. ♢

Exercise 1.2: Compute $LCS(X,Y)$ for $X =$ `AATTCCCCGACTGCAATTCACGCACC` and $Y =$ `GGCTTTTATTCTCCCTGTAAGT`. Note: these are DNA sequences from a modern human and a Neanderthal, respectively. ♢

Exercise 1.3:

(a) Give a direct recursive algorithm for computing $L(X,Y)$ based on equation (2) and show that it

takes exponential time. (In other words, equation (2) alone does not ensure efficiency of solution.)
(b) Let $lcs(X, Y)$ denote any member of $LCS(X, Y)$. Give the analogue of (3) for $lcs(X, Y)$. ♢

Exercise 1.4: Let $S = \{X_1, \ldots, X_k\}$ be a set of strings. A string Z such that each X_i is a subsequence of Z is called a **superstring** of S. We can consider the corresponding "shortest superstring problem" for any given S. In some sense, this is the dual of the LCS problem. Is there a dynamic programming solution for the shortest superstring problem? ♢

Exercise 1.5: Joe Quick observed that the recurrence (2) for computing $L(X, Y)$ would work just as well if we look at suffixes of X, Y (*i.e.*, by omitting prefixes). On further reflection, Joe concluded that we could double the speed of our algorithm if we work from *both ends* of our strings! That is, for $0 \leq i < j$, let $X_{i,j}$ denote the substring $x_i x_{i+1} \cdots x_{j-1} x_j$. Similarly for $Y_{k,\ell}$ where $0 \leq k < \ell$. Derive an equation corresponding to (2) and describe the corresponding algorithm. Perform an analysis of your new algorithm, to confirm and or reject the Quick Hypothesis. ♢

Exercise 1.6: What are the forbidden configurations in the matrix M? For instance, we have the following constraints: $0 \leq M[i, j] - M[i-1, j] \leq 1$ and $0 \leq M[i, j] - M[i, j-1] \leq 1$. Also, $M[i, j] = M[i-1, j] = M[i, j-1] = M[i-1, j-1]$ is impossible. Note that these constraints are based only on adjacency matrix entries. Is it possible to exactly characterize the set of all allowable configurations of M based on such adjacency constraints? ♢

Exercise 1.7:
(a) Write the code in your favorite programming language to fill the above table for $L(X, Y)$.
(b) Modify the code so that the program retrieves some member of $LCS(X, Y)$.
(c) Modify (b) so that the program also reports whether $|LCS(X, Y)| > 1$. Remember that we do not count duplicates in $LCS(X, Y)$. ♢

Exercise 1.8: Let X, Y be strings.
(a) Prove that $L(XX, Y) \leq 2L(X, Y)$.
(b) Give an example where the inequality is strict. the best possible.
(c) Prove that $L(XX, YY) \leq 3L(X, Y)$. How tight is this upper bound? ♢

Exercise 1.9: Suppose we have a parallel computer with unlimited number of processors.
(a) How many parallel steps would you need to solve the $L(X, Y)$ problem using our recurrence (2)?
(b) Give a solution to Joe Quick's idea (previous exercise) of having an algorithm that runs twice as fast on our parallel computer. Hint: work the last two symbols of each input string X, Y in one step. ♢

Exercise 1.10: Let $\lambda(X, Y)$ denote size of the set $LCS(X, Y)$ and $\lambda(m, n)$ be the maximum of $\lambda(X, Y)$ when $|X| = m, |Y| = n$. Finally let $\lambda(n) = \lambda(n, n)$.
(a) Compute $\lambda(n)$ for $n = 1, 2, 3, 4$.
(b) Give upper and lower bounds for $\lambda(n)$. ♢

Exercise 1.11: Let $LCS'(X, Y)$ be the *multiset* of all the longest common subsequences of X and Y. Let $\lambda'(n, m)$ and $\lambda'(n)$ be defined as in the previous exercise. Re-do the previous exercise for $\lambda'(n)$. ♢

Exercise 1.12: Modify the algorithm for $L(X, Y)$ to compute the following functions:
(a) $\lambda'(X, Y)$
(b) $\lambda(X, Y)$ ♢

Exercise 1.13: Instead of the bottom-up filling of tables, let us do a recursive top-down approach. That is, we begin by trying to fill in the entry $L[m, n]$. If $x_m = y_n$, we recursively try to fill in the entries for $L[m-1, n-1]$; otherwise, recursively solve for $L[m-1, n]$ and $L[m, n-1]$. Can you quantify the improvements in this approach? ♢

Exercise 1.14: (a) Solve the problem of computing the length $L(X, Y, Z)$ of the longest common subsequence of three strings X, Y, Z.
(b) What can you say about the complexity of the further generalization to computing $L(X_1, \ldots, X_m)$ (for $m \geq 3$). ♢

Exercise 1.15: A common subsequence of X, Y is said to be **maximal** if it is not the proper subsequence of another common subsequence of X, Y. For example, *let* is a maximal subsequence of *longest* and *length*. Let $LCS^*(X, Y)$ denotes the set of maximal common subsequences of X and Y. Design an algorithm to compute $LCS^*(X, Y)$. ♢

Exercise 1.16: A **Davenport-Schinzel sequence on n symbols** (or, **n-sequence** for short) is a string $X = x_1, \ldots, x_\ell \in \{a_1, \ldots, a_n\}^*$ such that $x_i \neq x_{i+1}$. The **order** of X is the smallest integer k such that there does not exist a subsequence of length $k+2$ of the form

$$a_i a_j a_i a_j \cdots a_i a_j a_i \quad \text{or} \quad a_i a_j a_i a_j \cdots a_j a_i a_j$$

where a_i and a_j alternate and $a_i \neq a_j$. Define $\lambda_k(n)$ to be the maximum length of a n-sequence of order at most k.
(a) Show that $\lambda_1(n) = n$ and $\lambda_2(n) = 2n - 1$. NOTE: for an order 2 string, a symbol may n times.
(b) Suppose X is an n-sequence of order 3 in which a_n appears at most $\lambda_3(n)/n$ times. After erasing all occurrences of a_n, we may have to erase occurrences a_i $(i = 1, \ldots, n-1)$ in case two copies of a_i becomes adjacent. We erase as few of these a_i's as necessary so that the result X' is a $(n-1)$-sequence. Show that $|X| - |X'| \leq \lambda_3(n)/n + 2$.
(c) Show that $\lambda_3(n) = O(n \log n)$ by solving a recurrence for $\lambda_3(n)$ implied by (b).
(d) Give an algorithm to determine the order of an n-sequence. Bound the complexity $T(n, k)$ of your algorithm where n is the length input sequence and $k \leq n$ the number of symbols. ♢

Exercise 1.17: Consider the generalization of LCS in which we want to compute the LCS for any input set of strings.
(a) If the input set have bounded size, give a polynomial time solution.
(b) (Maier, 1978) If the input set is unbounded, show that the problem is NP-complete. ♢

End Exercises

§2. Edit Distance Problem

A closely related problem is the **edit distance problem**. Again, the alphabet Σ is fixed. For any index $i \geq 1$ and letter $a \in \Sigma$, we define the following **editing operations**

$$Ins(i, a), \quad Del(i), \quad Rep(i, a).$$

These operations, when applied to a string X, will **insert** the letter a so that it is now in position i, **delete** the ith letter, and **replace** the ith letter by a (respectively). Let

$$Ins(i, a, X), \quad Del(i, X), \quad Rep(i, a, X) \tag{5}$$

denote the respective results.

For example, suppose $X = aatcga$. Then $Ins(3, g, X) = aagtcga$, $Del(5, X) = aatca$ and $Rep(5, t, X) = aatcta$. In general, if $Y = Ins(i, a, X)$, then $|Y| = 1 + |X|$ and

$$Y[j] = \begin{cases} X[j] & \text{if} \quad j = 1, \ldots, i-1 \\ a & \text{if} \quad j = i \\ X[j+1] & \text{if} \quad j = i+1, \ldots, |X| \end{cases}$$

The other operations can be similarly characterized.

The notations in (5) are unambiguous only when i is in the "proper range". For insertion, this means $1 \le i \le |X| + 1$, but for deletion and replacement, this means $1 \le i \le |X|$. But when i is not in the proper range, we may introduce conventions for interpreting (5). The operations $Del(i)$ and $Ins(i, a)$ are inverses of each other in the following sense:

$$Del(i, Ins(i, a, X)) = X, \qquad Ins(i, b, Del(i, X)), \tag{6}$$

for some b. Whatever our conventions for handling improper indices, we would want equations such as (6) to hold.

For simplicity, however, we simply declare such operations to be undefined. In the following, we will implicitly assume that i is in the proper range whenever we apply these operations.

Let $D(X, Y)$ be the minimum number of editing operations that will transform X to Y. Clearly,

$$D(X, Y) \le \max\{|X|, |Y|\}. \tag{7}$$

The **triangular inequality** holds: for any strings X, Y, Z, it is clear that

$$D(X, Z) \le D(X, Y) + D(Y, Z). \tag{8}$$

In fact, $D(X, Y)$ is a metric since it satisfies the usual axioms for a metric:

(i) $D(X, Y) \ge 0$ with equality iff $X = Y$.

(ii) $D(X, Y) = D(Y, X)$.

(iii) $D(X, Y)$ satisfies the triangular inequality (8).

It is interesting to view the set Σ^* of all strings over a fixed alphabet Σ as vertices of an infinite bigraph $G(\Sigma)$ in which $X, Y \in \Sigma^*$ are connected by an edge iff there exists an operation of the form (5) that transforms X to Y. Paths in $G(\Sigma)$ are called **edit paths**. Thus $D(X, Y)$ is the length of the shortest path from X to Y in $G(\Sigma)$.

In analogy to (2), we have the

$$D(X, Y) = \begin{cases} \max\{|X|, |Y|\} & \text{if} \quad m = 0 \text{ or } \quad n = 0 \\ D(X', Y') & \text{if} \quad x_m = y_n \\ 1 + \min\{D(X', Y), D(X, Y'), D(X', Y')\} & \text{if} \quad x_m \ne y_n \end{cases} \tag{9}$$

We leave it as an exercise to prove the correctness of this equation. It follows that $D(X,Y)$ can also be computed in $O(mn)$ time by the same technique of filling in entries in an $m \times n$ matrix M.

Suppose, we want to actually compute the sequence of $D(X,Y)$ edit operations that convert X to Y. Again, we expect to annotate the matrix M with some additional information to help us do this. For this purpose, let us decode equation (9) a little. There are four cases:
(a) In case $x_m = y_n$, the edit operation is a no-op.
(b) If $D(X,Y) = 1 + D(X',Y)$, the edit operation is $Del(m, X)$.
(c) If $D(X,Y) = 1 + D(X,Y')$, the edit operation is $Ins(m+1, y_n, X)$.
(d) If $D(X,Y) = 1 + D(X',Y')$, the edit operation is $Rep(m, y_n, X)$.
Hence it is enough to store two additional bits per matrix entry to reconstruct *one* possible sequence of $D(X,Y)$ edit operation.

What is the relation between $L(X,Y)$ and $D(X,Y)$? There is no universal relation, valid for all X and Y. Here are some inequalities:

Lemma 1 *Let X and Y have lengths m and n. Then*

$$D(X,Y) \leq m + n - 2L(X,Y).$$

and

$$D(X,Y) \geq \max\{m,n\} - L(X,Y).$$

Proof. The first inequality is easy. Suppose $\widehat{X}$ is the substring that is left after we delete a longest common subsequence Z of X and Y. Let $\widehat{Y}$ be similarly defined for Y after Z, viewed as a subsequence of Y is deleted. This implies that

$$D(X,Y) \leq |\widehat{X}| + |\widehat{Y}| = m + n - 2L(X,Y).$$

since we can transform $\widehat{X}$ to $\widehat{Y}$ using $|\widehat{X}| + |\widehat{Y}|$ edit operations, without disturbing Z (so to speak). This proves our upper bound.

The lower bound on $D(X,Y)$ is based on the intuition that any transformation from $\widehat{X}$ to $\widehat{Y}$ requires at least $\max\{|\widehat{X}|, \widehat{Y}|\}$ operations. But we have to prove that the shortest edit path from X to Y must essentially go along such a route. An alternative argument goes as follows: assume $m \geq n$ and we will show $L(X,Y) \geq m - D(X,Y)$. Suppose we transform X to Y in a sequence of $D(X,Y)$ edit steps. But in $D(X,Y)$ steps, there is a subsequence Z of X of length $m - D(X,Y)$ that is unaffected. Hence Z is also a subsequence of Y. So $L(X,Y) \geq |Z| = m - D(X,Y)$. **Q.E.D.**

The above lemma is the best possible in the following sense: for each positive $\ell \leq \min\{m,n\}$, there are strings X, Y such that $D(X,Y) = m + n - 2\ell$ where $L(X,Y) = \ell$. For this purpose, choose $X = X'Z$ and $Y = ZY'$ such that Z is the unique string in $LCS(X,Y)$ For the lower bound on $D(X,Y)$, we choose $X = X'Z$ and $Y = Y'Z$ where Z is any string in $LCS(X,Y)$. Then $D(X,Y) = \max\{m,n\} - \ell$. See Exercises for more details.

Alignment Problem. Let us a variation of the editing distance problem. It is motivated by computational biology where we want to "line up two DNA sequences" to compare them. Suppose X, Y are two strings. You are allowed to insert, delete and replace, just as before. What is new is the cost function:

- Each insert or delete costs 2 points.

- Each replace costs 1 point.
- Each original match costs -2 points. The term "original match" will be explained next.

Here is one description of how you apply this cost model. Let the input strings be X, Y. We proceed in two phases. In Phase 1, you are allowed to do any number of inserts or deletes to X, Y to produce X^+, Y^+ so that they have the same lengths. The cost is $2k$ points where k is the number of inserts or deletes. This completes Phase 1. In Phase 2, we have an obvious way to align X^+ and Y^+, since $|X^+| = |Y^+|$. At each position of this alignment, we compare a letter x in X^+ to a letter y in Y^+. If $x \neq y$ we must do a replacement at the cost of 1 point. If $x = y$ and both x and y are "original letters" from X, Y then the cost is -2. Otherwise the cost is 0 (since this means that x or y must have been inserted). Sum up the costs for replacements and original matches. This ends Phase 2.

This gives us a particular procedure for aligning X and Y, and we have an associated cost (= total cost of the two phases). Let $A(X, Y)$ be the minimum over the costs of all procedures for aligning X and Y.

For instance, $X = cga$ and $Y = acaat$. We can first insert a in the front of X to get $X' = acga$ (costs 2), and delete t from Y to get $Y' = acaa$. To indicate the various copies of a letter, we use subscripts: $X' = a_1cga_2$ and $Y' = a_3ca_4a_5$. Now we make the obvious one-one correspondence between X' and Y': we get matches at positions $1, 2, 4$. The matches at positions 2 and 4 costs -2 units each. However, the match at position 1 (between a_1 with a_3) is a result of inserting a_1 and so it has already been counted. In other words, it is not an "original match". Since there is no match at 3, we need to replace g by an a to get a match – this replacement costs 1 unit. This shows that $A(X, Y) \leq 2 + 2 - 2 - 2 + 1 = 1$. Can you show a lower cost alignment of X, Y?

Recursive Formula for $A(X, Y)$. Let us develop a recursive formula for alignment. Suppose X^+, Y^+ are the strings at the end of Phase 1 in an optimal alignment procedure. Without loss of generality, assume that only insertions are performed in Phase 1, since deletions in one string can be simulated by a insertion in the other string. Let x and y the last characters in X, Y, respectively. Similarly, let the last characters in X^+ and Y^+ be x^+ and y^+. Since we do an optimal alignment, it is clear that at least one of x^+ and y^+ is from the initial string. (if both x^+, y^+ were inserted, this is clearly suboptimal). Hence there are three possibilities: as usual, let X', Y' denote the prefex of X, Y (resp.) after deleting its last letter.
1. $(x = x^+)$ and $(y \neq y^+)$. This means y^+ was inserted and so the cost is $A(X, Y) = Cost(Insertion) + A(X', Y) = 2 + A(X', Y)$ where X' is the prefix of X obtained by dropping the last character in X.
2. $(x \neq x^+)$ and $(y = y^+)$. Similar to the previous case, we get the recurrence $A(X, Y) = 2 + A(X, Y')$.
3. $(x = x^+)$ and $(y = y^+)$. This means the original characters at the end of X and Y are kept. There are two possibilities: (a) If $x = y$ then we get an original match with cost -2. The recurrence is $A(X, Y) = -2 + A(X', Y')$. (b) If $x \neq y$, the we need a replacement with cost 1, giving the recurrence $A(X, Y) = 1 + A(X', Y')$.

The base case is when $|X| = 0$ or $|Y| = 0$. In this case, $A(X, Y) = 2\max\{|X|, |Y|\}$. Hence the overall recurrence is

$$A(X,Y) = \begin{cases} 2\max\{|X|,|Y|\} & \text{if } |X| \cdot |Y| = 0, \\ \min\{2 + A(X',Y), 2 + A(X,Y'), -2 + A(X',Y')\} & \text{if x=y}, \\ \min\{2 + A(X',Y), 2 + A(X,Y'), 1 + A(X',Y')\} & \text{else} \end{cases}$$

We can simplify the recurrence in case $x = y$: we claim that $\min\{2+A(X', Y), 2+A(X, Y'), -2+A(X', Y')\} = -2 + A(X', Y')$. To see this, we note that $A(X', Y') \leq 2 + A(X', Y)$ since we can insert y at the end of Y' to create Y. Hence $2 + A(X', Y) \geq A(X', Y') > -2 + A(X', Y')$. Similarly, $2 + A(X, Y') \geq A(X', Y') > -2 + A(X', Y')$. Hence we get:

$$A(X,Y) = \begin{cases} 2\max\{|X|,|Y|\} & \text{if } |X| \cdot |Y| = 0, \\ -2 + A(X',Y') & \text{if x=y}, \\ \min\{2 + A(X',Y), 2 + A(X,Y'), 1 + A(X',Y')\} & \text{else} \end{cases}$$

Now the optimal alignment cost can be solved by a filling in of a matrix in the standard way.

We remark that it is interesting to see how this model gives a cost for original matching, even though no explicit operation was perform. Moreover, since this matching cost is negative, we see that $A(X, Y)$ can be negative.

Generalizations. There are many possible generalization of the above string problems.

- We can introduce costs associated to each type of editing operations. The implicit cost model above is the unit cost for every operation.
- The fundamental primitive in these problems is the comparison of two letters: is letter $X[i]$ equal to letter $Y[j]$ (a "match") or not (a "non-match")? We can generalize this by allowing "approximate" matching (allowing some amount of non-match) or allow generalized "patterns" (e.g., wild card letters or regular expressions).
- We can also generalize the notion of strings. Thus "multidimensional strings" is just an arrays of letters, where the array has some fixed dimension. Thus, strings are just 1-dimensional arrays. It is natural to view 2-dimensional arrays as raster images.
- Another generalization of strings is based on trees. A **string tree** is a rooted tree T in which each node v is labeled with a letter $\lambda(v)$ (from some fixed alphabet). The tree may be ordered or unordered. In a natural way, T represents a collection (order or unordered) of strings. Let P and T be two string trees. We say that P is a **(string) subtree** of T if there is 1-1 map μ from the nodes of P to the nodes of T such that
  - μ is label-preserving: $v \in P$ and $\mu(v) \in T$ has the same label.
  - μ is "parent preserving": if u is the parent of v in P then $\mu(u)$ is the parent of $\mu(v)$ in T. For ordered trees, we further insist that μ be order preserving.

  In particular, if v_0 is the root of P then $\mu(P)$ is a subtree (in the usual sense of rooted trees) of T rooted a $\mu(v_0)$. We say there is a "match" at $\mu(v_0)$. Hence a basic problem is, given P and T, find a match of P in T, if any. Consider the edit distance problem for string trees. The following edit operations may be considered: (1) Relabeling a node. (2) Inserting a new child v to a node u, and making some subset of the children of u to be children of v. In the case of ordered trees, this subset must form a consecutive subsequence of the ordered children of u. (3) Deleting a child v of a node u. This is the inverse of the insertion operation. We next assign some cost γ to each of these operations, and define the edit distance $D(T, T')$ between two string trees T and T' to be the miminum cost of a sequence of operations that transforms T to T'. A natural requirement is hat $D(T, T')$ is a metric: so, $D(T, T') \geq 0$ with equality iff $T = T'$, $D(T, T') = D(T', T)$ and the triangular inequality be satisfied.

Remarks: Levenshtein (1966) introduce the editing metric for strings in the context of binary codes. Sankoff and Kruskal (1983) considered the LCS problem in computational biology applications. Applications of string tree matching problems arise in term-rewriting systems, logic programming and evolutionary biology. We refer to the collection in [1] for a state-of-the-art overview, circa 1997.

Exercises

Exercise 2.1: Compute the edit distances $D(X, Y)$ where X, Y are given:
(a) $X = 00110011$ and $Y = 10100101$.
(b) $X =$ `agacgttcgttagca` and $Y =$ `cgactgctgtatgga`. ♢

Exercise 2.2: Compute $A(X,Y)$ where X, Y are the strings `AATTCCCGA` and `GCATATT`. You must organize this computation systematically as in the LCS problem. ♢

Exercise 2.3: Prove (9). This is an instructive exercise. ♢

Exercise 2.4: Let x, y, z be distinct letters.
(a) Prove that $D(X,Y) = m + n - 2\ell$. where $X = x^{m-\ell}z^{\ell}$ and $Y = z^{\ell}y^{n-\ell}$. Note that the upper bound on $D(X,Y)$ follows from the lemma in the text.
(b) Let $X = x^{m-\ell}z^{\ell}$ and $Y = y^{n-\ell}z^{\ell}$. Prove that $D(X,Y) = \max\{m,n\} - \ell$. ♢

Exercise 2.5: Let X, Y be strings. Clearly, $L(XX, YY) \geq 2L(X,Y)$.
(a) Give an example where the inequality is strict.
(b) Prove that $L(XX,Y) \leq 2L(X,Y)$ and this is the best possible.
(c) Prove that $L(XX,YY) \leq 3L(X,Y)$.
(d) We know from (a) and (c) that $L(XX,YY) = cL(X,Y)$ where $2 \leq c \leq 3$. Give sharper bounds for c. ♢

Exercise 2.6: Suppose we consider the edit distance $D(X,Y)$ problem in which each insert, delete or replacement operation has an individual cost, which may even depend on the actual characters involved. Specifically, for letters $a \neq b$, let

$$D(\lambda, b), D(a, \lambda), D(a, b)$$

denote, respectively, the cost to insert a letter b, to delete a letter a and to replace a by b. To what extent can dynamic programming be used to solve this problem? You may assume that the above costs are positive, but what other properties do you need? ♢

Exercise 2.7: Suppose we allow the operation of **transpose**, $\ldots ab \ldots \to \ldots ba \ldots$. Let $T(X,Y)$ be the minimum number of operations to convert X to Y, where the operations are the usual string edit operations plus transpose.
(i) Compute $T(X,Y)$ for the following inputs: $(X,Y) = (ab,c)$, $(X,Y) = (abc,c)$, $(X,Y) = (ab,ca)$ and $(X,Y) = (abc,ca)$.
(ii) Show that $T(X,Y) \geq 1 + \min\{T(X',Y), T(X,Y'), T(X',Y')\}$.
(iii) In what sense can you say that $T(X,Y)$ cannot be reduced to some simple function of $T(X',Y), T(X,Y')$ and $T(X',Y')$?
(iv) Derive a recursive formula for $T(X,Y)$. ♢

Exercise 2.8: In computational biology applications, there is interest in another kind of edit operation: namely, you are allowed to reverse a substring: if X, Y, Z are strings, then we can transform the XYZ to XY^RZ in one step where Y^R is the reverse of R. Assume that substring reversal is added to our insert, delete and replace operations. Give an efficent solution to this version of the edit distance problem. ♢

End Exercises

§3. Triangulating an Abstract Polygon

We now address a different family of problems amenable to the dynamic programming approach. These problems have an abstract structure that is best explained using the notion of convex polygons.

The standard notion of a polygon P is a geometric one, and may be represented by a sequence $(v_1, \ldots, v_n)$ of **vertices** where $v_i \in \mathbb{R}^2$ is a point in the Euclidean plane. We say P is **convex** if no v_i in contained in the interior of the triangle $\Delta(v_j, v_k, v_\ell)$ formed by any other triple of points. Figure 1 shows a convex polygon with $n = 7$ vertices. An **edge** of P is a line segment $[v_i, v_{i+1}]$ between two consecutive vertices (the subscript arithmetic, "$i + 1$", is modulo n). Thus $[v_1, v_n]$ is also an edge. A **chord** is an line segment $[v_i, v_j]$ that is not an edge.

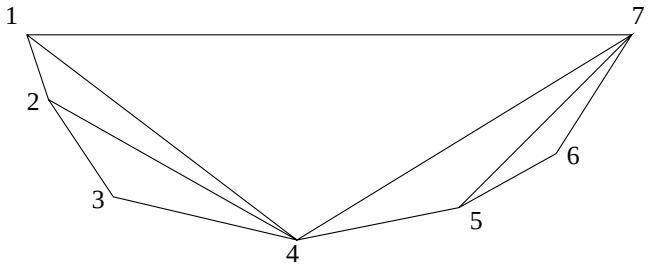


Figure 1: A triangulated 7-gon

Abstract Polygons. We now give an abstract, purely combinatorial version of these terms. Let $P = (v_1, \ldots, v_n)$, $n \geq 1$, be a sequence of n distinct symbols, called a **combinatorial convex polygon**, or an **(abstract) n-gon** for short. We call each v_i a **vertex** of P. Since the vertices are merely symbols (only the underlying linear ordering matters), it is often convenient to identify v_i with the integer i. In this case, we call $(v_1, \ldots, v_n) = (1, \ldots, n)$ the **standard n-gon**. Henceforth, we assume $n \geq 3$ to avoid trivial considerations.

Assume P is a standard n-gon. By a **segment** of P we mean an ordered pair of vertices, (i, j) where $1 \leq i < j \leq n$. This is sometimes written "ij". We classify a segment ij as an **edge** of P if $j = i + 1 (\mod n)$; otherwise the segment is called a **chord**. Thus, $1n$ is an edge. If $n \geq 3$, there are exactly n edges and $n(n-3)/2$ chords (why?). We say two segments ij and $k\ell$ **intersect** if

$$i < k < j < \ell \qquad \text{or} \qquad k < i < \ell < j;$$

otherwise they are **disjoint**. Note that an edge is disjoint from any other segment of P.

Triangulations. It is not hard to show by induction that a *maximal* set T of pairwise disjoint chords of P has size exactly $n-3$. If $n \geq 3$, a set T with exactly $n-3$ pairwise disjoint chords is called a **triangulation** of P. In the following, it is convenient to consider the degenerate case of a 2-gon; the empty set is, by definition, the unique triangulation of a 2-gon. E.g., figure 1 shows a triangulation

$$T = \{14, 24, 47, 57\}$$

of the standard 7-gon. A **triangle** of P is a triple (i, j, k) (or simply, ijk) where $1 \leq i < j < k \leq n$; its three edges are ij, jk and ik. E.g., the set of all triangles of the standard 5-gon are

$$123, 124, 125, 134, 135, 145, 234, 235, 245, 345.$$

We say ijk **belongs to** a triangulation T if each edge of the triangle is either a chord in T or an edge of P. Thus the triangles of the T in figure 1 are

$$\{124, 234, 147, 457, 567\}.$$

Every triangulation T has exactly $n-2$ triangles belonging to it, and each edge of P appears as the edge of exactly one triangle and each chord in T appears as the edge of exactly two triangles [Check: $n-2$ triangles has a combined total of $2(n-3)+n$ edges.] In particular, there is a unique triangle belonging to T which contains the edge $1n$. This triangle is $(1, i, n)$ for some $i = 2, \ldots, n-1$. Then the set T can be partitioned into three disjoint subsets

$$T = T_1 \uplus T_2 \uplus S_i$$

where $S_i = T \cap \{(1, i), (i, n)\}$, and T_1, T_2 are (respectively) triangulations of the i-gon $P_1 = (1, 2, \ldots, i)$ and the $(n-i+1)$-gon $P_2 = (i, i+1, \ldots, n)$. Note that $S_i = \{(1, i), (i, n)\}$ iff $2 < i < n-1$. Also, our convention about the triangulation of 2-gons is assumed when $i = 2$ or $i = n-1$.

Thus triangulations can be viewed recursively. This is the key to our ability to decompose problems based on triangulations. E.g., the triangulation T in figure 1 has the partition

$$T = T_1 \uplus T_2 \uplus S_4$$

where $S_4 = \{14, 47\}$, $T_1 = \{24\}$ and $T_2 = \{57\}$.

Weight functions and optimum triangulations. A **(triangular) weight function** on n vertices is a non-negative real function W such that $W(i, j, k)$ is defined for each triangle ijk of an abstract n-gon. The **W-cost** of a triangulation T is the sum of the weights $W(i, j, k)$ of the triangles ijk belonging to T. The **optimal triangulation problem** asks for a minimum W-cost triangulation of P, given its weight function W.

Example: In case $P = (v_1, \ldots, v_n)$ is a geometric polygon in the plane, a natural cost function is $W(i, j, k)$ is the perimeter $\|v_i - v_j\| + \|v_i - v_k\| + \|v_j - v_k\|$ of the triangle (v_i, v_j, v_k), where $\|\cdot\|$ denotes the Euclidean length function. It is easy to check that T is optimal iff it minimizes the sum $\sum_{(v_i,v_j)\in T} \|v_i - v_j\|$ of the lengths of the chords in T. This might be a reasonable "cost" if a carpenter has to saw a wooden P into $n-2$ triangles. It might also be regarded as the cost of "disposing of the sawdust".

In specifying W, we generally expected the "specification size" to be $\Theta(n^3)$. However, in many applications, the function W is implicitly defined by fewer parameters, typically $\Theta(n)$ or $\Theta(n^2)$. Here are some examples.

1. **Metric Sawdust Problem:** this is a generalization of the "sawdust example". Suppose each vertex i of P is associated with a point p_i of some metric space. Then $W(i, j, k) = d(p_i, p_j) + d(p_j, p_k) + d(p_k, p_i)$ where $d(p, q)$ is the metric between two points of the space.

2. **Generalized Perimeter Problem:** W is defined by a symmetric matrix $(a_{ij})_{i,j=1}^n$ such that $W(i, j, k) = a_{ij} + a_{jk} + a_{ik}$. We can view $a_{i,j}$ as the "distance" from node i to node j and $W(i, j, k)$ is thus the perimeter of the triangle ijk. This is another generalization of "metric sawdust". Here, W is specified by $\Theta(n^2)$ parameters. More generally, we might have

   $$W(i, j, k) = f(a_{ij}, a_{jk}, a_{ik})$$

   where $f(\cdot, \cdot, \cdot)$ is some function.

3. **Weight functions induced by vertex weights:** W is defined by a sequence $(a_1, \ldots, a_n)$ of objects where

   $$W(i, j, k) = f(a_i, a_j, a_k).$$

   for some function $f(\cdot, \cdot, \cdot)$. If a_i is a number, we can view a_i as the weight of the ith vertex. Two examples are $f(x, y, z) = x + y + z$ (sum) and $f(x, y, z) = xyz$ (product). The case of product corresponds to the matrix chain product problem studied in §4.

4. **Weight functions from differences of vertex weights:** W is defined by an increasing sequence $a_1 \leq a_2 \leq \cdots \leq a_n$ and $W(i,j,k) = a_k - a_i$. Note that the index j is not used in $W(i,j,k)$. In §5, we will see an example (optimum search trees) of such a weight function.

A dynamic programming solution. The cost of the optimal triangulation can be determined using the following recursive formula: let $C(i,j)$ be the optimal cost of triangulating the subpolygon $(i, i+1, \ldots, j)$ for $1 \leq i < j \leq n$. Then

$$C(i,j) = \begin{cases} 0 & \text{if } j = i+1, \\ \min_{i<k<j}\{W(i,k,j) + C(i,k) + C(k,j)\} & \text{else.} \end{cases} \tag{10}$$

The desired optimal triangulation has cost $C(1,n)$. Assuming that the value $W(i,j,k)$ can be obtained in constant time, and the size of the input is n, it is not hard to implement this outline to give a cubic time algorithm. We say more about this in the next section.

Exercises

Exercise 3.1: Find an optimal triangulation of the abstract pentagon whose weight function W is parameterized by $(a_1, \ldots, a_6) = (4,1,3,2,2,3)$:
(a) The weight function is given by $W(i,j,k) = a_i a_j a_k$.
(b) The weight function is given by $W(i,j,k) = |a_i - a_j| + |a_i - a_k| + |a_j - a_k|$. ♢

Exercise 3.2: Suppose P is a geometric simple polygon, not necessarily convex. We now define chords of P to comprise those segments that do not intersect the exterior of P. A triangulation is as usual a set of $n-3$ chords. Let W be a weight function on the vertices of P. Give an efficient method for computing the minimum weight triangulation of P. The goal here is to give a solution that is $O(k)$ where k is the number of chords of P. ♢

Exercise 3.3: (T. Shermer) Let P be a simple (geometric) polygon (so it need not be convex). Define the "bushiness" $b(P)$ of P to be the minimum number of degree 3 vertices in the dual graph of a triangulation of P. A triangulation is "thin" if it achieves b(P). Give an $O(n^3)$ algorithm for computing a thin triangulation. ♢

Exercise 3.4: Suppose that we want to **maximize** the "triangulation cost" (we should really interpret "cost" as "reward") for a given weight function $W(i,j,k)$. Does the same dynamic programming method solve this problem? ♢

Exercise 3.5: (Multidimensional Dynamic Programming?)
(a) Give a dynamic programming algorithm to optimally partition an n-gon into a collection of 3- or 4-gons. Assume we are given a non-negative real function $W(i,j,k,l)$, defined for all $1 \leq i \leq j \leq k \leq l \leq n$ such that $|\{i,j,k,l\}| \geq 3$. The value $W(i,j,k,l)$ should depend only on the set $\{i,j,k,l\}$: if $\{i,j,k,l\} = \{i',j',k',l'\}$, then $W(i,j,k,l) = W(i',j',k',l')$. For example, $W(2,2,4,7) = W(2,4,4,7)$. The weight of a partitioning is equal to the sum of the weights over all 3- or 4-gons in the partition. Analyze the running time of your algorithm. NOTE: this problem has a 2-dimensional structure on its subproblems, but it can be generalized to any dimensions.
(b) Solve a variant of part (a), namely, the partition should exclusively be composed of 4-gons when $n-4$ is even, and has exactly one 3-gon when $n-4$ is odd. ♢

End Exercises

§4. The Dynamic Programming Method

Let us note three ingredients essential for our dynamic programming solution (10) for the triangulation problem:

- **There are a small number of subproblems**. We usually interpret "small" to mean a polynomial number. We began with a weight function W on the n-gon $(1, \ldots, n)$. Each contiguous subsequence

$$(i, i+1, i+2, \ldots, j-1, j), \qquad (1 \leq i < j \leq n)$$

  induces a weight function $W_{i,j}$ on the $(j-i+1)$-gon $(i, i+1, \ldots, j-1, j)$. This gives rise to the **subproblem** $P_{i,j}$ of optimal triangulation of $(i, i+1, \ldots, j)$. The original problem is just $P_{1,n}$. There are $\Theta(n^2)$ subproblems. The "wrong" formulation can violate this smallness requirement (see Exercise).

- **An optimal solution of a problem induces optimal solutions on certain subproblems.** If T is an optimal triangulation on $(a_1, \ldots, a_n)$, then we have noted that $T = T_1 \uplus T_2 \uplus S_i$ where $S_i \subseteq \{1i, in\}$ and T_1, T_2 are triangulations of subpolygons of P. In fact, T_1, T_2 are optimal solutions to subproblems $P_{1,i}$ and $P_{i,n}$ for some $1 < i < n$. This property is called the **dynamic programming principle**, namely, an optimal solution to a problem induces optimal solutions on certain subproblems.

- **The optimal solution of a problem is easily constructed from the optimal solutions of subproblems.** If we have already found the cost of optimal triangulations for all smaller subproblems of $P_{i,j}$ then we can easily solve $P_{i,j}$ using equation (10).

The reader may check that the same ingredients were present in the LCS and edit distance problems.

Mechanics of the algorithm. Here is a natural way to organize the computation embodied in equation (10). First we have an upper triangular $n \times n$ matrix A to store the values of $C(i, j)$,

$$A[i, j] = C(i, j), \qquad (i < j)$$

See Figure 2.

We view the algorithm as a systematic filling in of the upper part of A. Note that filling in the entries $A[i, j]$ can be viewed as solving a subproblem of size $(j - i + 1)$. We proceed in $n - 1$ stages, where stage S_t $(t = 2, \ldots, n)$ corresponds to solving all subproblems of size t. There are exactly $n - t + 1$ problems of size t. Note that to solve a problem of size t $(t \geq 2)$ we need to minimize over a set of $t - 2$ numbers (see equation (10)), and this takes time $O(t)$. Thus stage t takes $O((t-2)(n-t+1)) = O(n^2)$ time. Summed over all stages, the time is $O(n^3)$. The space requirement is $\Theta(n^2)$, because of the matrix A.

The algorithm is easy to implement in any conventional programming language: it has a triply-nested "for-loop", with the outermost loop-counter controlling the stage number, t. The following gives a bottom-up implementation of equation (10):

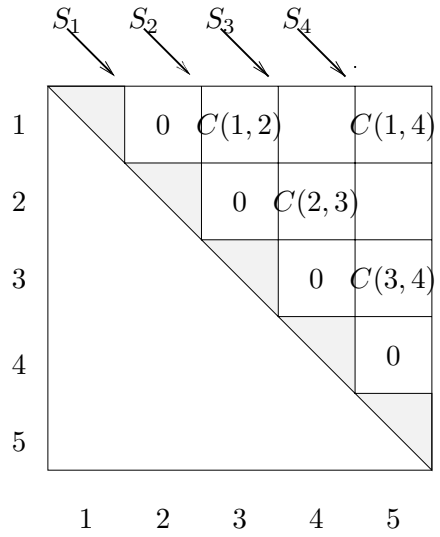


Figure 2: Filling in of a upper triangular matrix

```
Dynamic Programming for Optiam Triangulation
    for t ← 1 to n − 1    — do problems of size 2
        A[t, t + 1] ← 0.
    for t ← 2 to n − 1    — t + 1 is problem size
        for i ← 1 to n − t    — compute C[i, i + t]
            A[i, i + t] ← A[i, i + 1] + A[i + 1, i + t] + W(i, i + 1, i + t)
            for k ← i + 2 to i + t − 1
                A[i, i + t] ← min{A[i, i + t], A[i, k] + A[k, i + t] + W(i, k, i + t)}
```

This algorithm lends itself to hand simulation, a process that the student should become familiar with. For example

Splitters and the construction of Optimal Solutions. Suppose we want to find the actual optimal triangulation, not just its cost. Let us call any index k that minimizes the second expression on the right-hand side of equation (10) an (i, j)**-splitter**. If we can keep track of all the splitters, we can clearly construct the optimal triangulation. For this purpose, we employ an upper trianguler $n \times n$ matrix K where $K[i, j]$ stores an (i, j)-splitter. It is easy to see that the entry $K[i, j]$ can be filled in at the same time that $A[i, j]$ is filled in. Hence, finding optimal solutions is asymptotically the same as finding the cost of optimal solutions.

Top-down versus bottom-up dynamic programming. The above triply nested loop algorithm is a bottom-up design. However, it is not hard to construct a top-down design recursive algorithm: simply implement (10) by a recursion. However, it is important to maintain the matrices A (and K if desired) as global shared space. This technique has been called "memo-izing". Without memo-izing, the top-down solution can take exponential time, simply because there are exponentially many subproblems (see next section). A simple memoization does not speed up the algorithm. But we can, by computing bounds, avoid certain branches of the recursion. This can have potential speedup – see Exercise.

REMARK: The abstract triangulation problem has a "linear structure" on the subproblems. This linear structure can sometimes be artificially imposed on a problem in order to exploit the dynamic programming

framework (see Exercise on hypercube vertex selection).

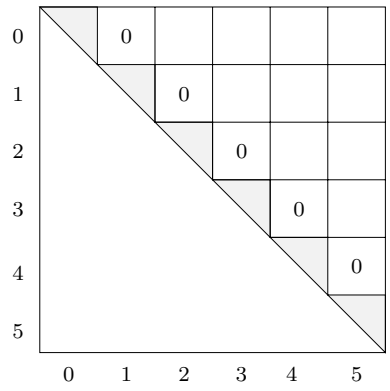
Exercises

Exercise 4.1: Jane Sharp noted an alternative to equation (10).

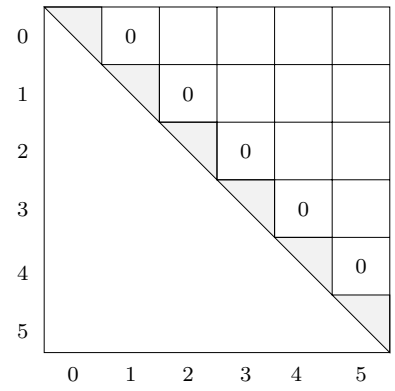
(a) Jane observed that every triangulation T must contain a triangle of the form $(i, i+1, i+2)$. Such a triangle is called an "ear". Prove this claim of Jane. (You may also prove the stronger claim that there are at least two ears.)

(b) Suppose we remove an ear from an n-gon. The result is an $(n-1)$-gon. If we knew an ear which appears in an optimum triangulation of an n-gon, we could recursively triangular the smaller $(n-1)$-gon. But since we do not know, we can try all possible $(n-1)$-gons obtained by removing an ear. What is wrong with this approach? (Try to write the analogue of equation (10), and think of the 3 ingredients needed for a dynamic programming approach.) ♢

Exercise 4.2: Let $(n_0, n_1, \ldots, n_5) = (2, 1, 4, 1, 2, 3)$. We want to multiply a sequence of matrices, $A_1 \times A_2 \times \cdots \times A_5$ where A_i is $n_{i-1} \times n_i$ for each i. Please fill in matrices (a) and (b) in Figure 3. Then write the optimal order of multiplying $A_1, \ldots, A_5$.



(a) Optimum Cost Matrix C



(b) Splitter Matrix K

Figure 3: (a) $C[i,j]$ is optimal cost to multiplying $A_i \times \cdots \times A_j$. (b) $K[i,j]$ indicates the optimal split, $(A_i \times \cdots \times A_{K[i,j]})(A_{K[i,j]+1} \times \cdots \times A_j)$

♢

Exercise 4.3: We are given three real coefficients a, b, c and a real function

$$h(x) = \begin{cases} 1 & \text{if } |x| < 1 \\ 0 & \text{else.} \end{cases}$$

Define the function $f(x, i)$ (where $i \geq 0$ is integer) as follows:

$$f(x,i) = \begin{cases} h(x) & \text{if } i = 0 \\ a \cdot f(2x-1, i-1) + b \cdot f(2x, i-1) + c \cdot f(2x+1, i-1) & \text{else.} \end{cases}$$

Suppose n, m are parameters and we want to compute the values $f(x, n)$ for all

$$x \in \{\pm \frac{i}{m} : i = 0, 1, \ldots, m-1\}$$

Assume that each arithmetic operation takes unit time.
(a) Determine the time to compute a single value $f(x)$ if we simply implement $f(x,n)$ by using straightforward recursion and every call to $f(x,n)$ is independent.
(b) The function $f(x,0)$ has support in $[-1,1]$. What can you say about the support of $f(x,n)$ where n is fixed?
(c) Use dynamic programming techniques to obtain an efficient solution.
NOTE: This problem is motivated by actual computations with wavelets. ♢

Exercise 4.4: (Recursive Dynamic Programming) The "bottom-up" solution of the optimal triangulation problem is represented by a triply-nested for-loop in the text. Now we want to consider a "top-down" solution, by using recursion. As usual, the weight $W(i,j,k)$ is easily computed for any $1 \le i < j < k \le n$.
(a) Give a naive recursive algorithm for optimal triangulation. Briefly explain how this algorithm is exponential.
(b) Describe an efficient recursive algorithm. You will need to use some global data structure for sharing information across subproblems.
(c) Briefly analyze the complexity of your solution.
(d) Does your algorithm ever run faster than the bottom-up implementation? Can you make it run faster on some inputs? HINT: for subproblem $P(i,j)$, we can try to compute upper and lower bounds on $C(i,j)$. Use this to "prune" the search. ♢

Exercise 4.5: Give a linear space $O(n)$ solution to problem of optimal triangulation. Write the recurrence for the space and time complexity of your algorithm. Solve for the running time. ♢

Exercise 4.6: Consider the problem of evaluating the determinant of an $n \times n$ matrix. The obvious co-factor expansion takes $\Theta(n \cdot n!)$ arithmetic operations. Gaussian elimination takes $\Theta(n^3)$. But for small n and under certain circumstances, the co-factor method may be better. In this question, we want you to improve the co-factor expansion method by using dynamic programming. What is the number of arithmetic operations if you use dynamic programming? Please illustrate your result for $n = 3$.

HINT: We suggest you just count the number of multiplications. Then argue separately that the number of additions is of the same order. ♢

Exercise 4.7: Generalize the previous exercise. Let the set of real constants $\{a_i : i = -N, -N+1, \ldots, -1, 0, 1, \ldots, N\}$ be fixed. Suppose that

$$f(x,i) = \begin{cases} h(x) & \text{if } i = 0 \\ \sum_{i=-N}^{N} a_i \cdot f(2x-1, i-1) & \text{else.} \end{cases}$$

Re-do parts (a)–(c) in the last exercise. ♢

Exercise 4.8: (Hypercube vertex selection) A **hypercube** or n-**cube** is the set $H_n = \{0,1\}^n$. Each $x = (x_1, \ldots, x_n) \in H_n$ is called a vertex of the hypercube. Let $\pi = (\pi_1, \ldots, \pi_n)$ and $\rho = (\rho_1, \ldots, \rho_n)$ be two positive integer vectors. The **price** and **reliability** of a vertex x is given by $\pi(x) = \sum_{i=1}^{n} x_i \pi_i$ and $\rho(x) = \prod_{i=1; x_i = 1}^{n} \rho_i$. The **hypercube vertex selection problem** is this: given π, ρ and a positive bound B_0, find $x \in H_n$ which maximizes $\rho(x)$ subject to $\pi(x) \le B_0$. Solve this problem in time $O(nB_0)$ (not $O(n \log B_0)$).
HINT: View $H_n = H_k \otimes H_{n-k}$ for any $k = 1, \ldots, n-1$ and $y \otimes z$ denotes concatenation of vectors $y \in H_k$, $z \in H_{n-k}$. Solve subproblems on H_k and H_{n-k} with varying values of B ($B = 1, 2, \ldots, B_0$). The choice of k is arbitrary, but what is the best choice of k? ♢

Exercise 4.9: Let $S \subseteq \mathbb{R}^2$ be a set of n points. Partially order the points $p = (p.x, p.y) \in \mathbb{R}^2$ as follows: $p \leq q$ iff $p.x \leq q.x$ and $p.y \leq q.y$. If $p \neq q$ and $p \leq q$, we write $p < q$. A point p is **S-minimal** if $p \in S$ and there does not exist $q \in S$ such that $q < p$. Let $\min(S)$ denote the set of S-minimal points.
(a) For $c \in R$, let $S(c)$ denote the set $\{p \in S : p.x \geq c\}$. E.g., let $S = \{p(1,3), q(2,1), r(3,4), s(4,2)\}$ as shown in figure 4. Then $\min(S(c))$ is equal to $\{p, q\}$ if $c \leq 1$; $\{q\}$ if $1 < c \leq 2$; $\{r, s\}$ if $2 < c \leq 3$;

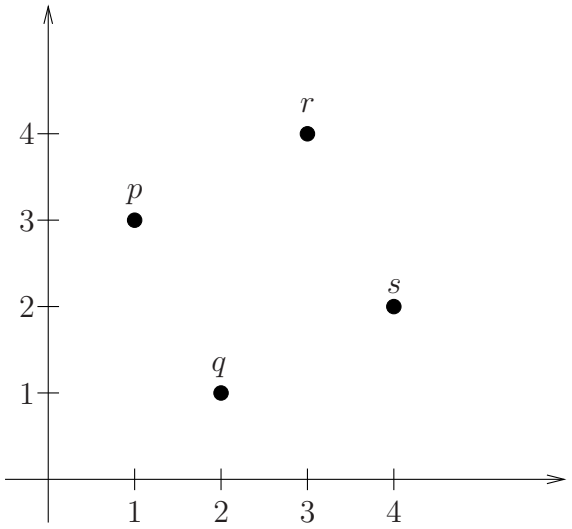


Figure 4: Set of 4 points.

$\{s\}$ if $3 < c$. Design a data structure $D(S)$ with two properties:

1. For any $c \in \mathbb{R}$ ("the query" is specified by c), you can use $D(S)$ to output the set $\min(S(c))$ in time
$$O(\log n + k)$$
where k is the size of $\min(S(c))$.
2. The data structure $D(S)$ uses $O(n)$ space.

(b) For any $q \in \mathbb{R}^2$, let $S(q)$ denote the set $\{p \in S : p.x \geq q.x, p.y \geq q.y\}$. Design a data structure $D'(S)$ such that for any $q \in \mathbb{R}^2$, you can use $D''(S)$ to output the set $\min(S(q))$ in time $O(\log n + k)$ where k is the size of $\min(S(q))$, and $D''(S)$ uses $O(n^2)$ space. ♢

Exercise 4.10: (Knapsack) In this problem, you are given $2n + 1$ positive integers,
$$W, w_i, v_i (i = 1, \ldots, n).$$
Intuitively, W is the size of your knapsack and there are n items where the ith item has size w_i and value v_i. You want to choose a subset of the items of maximum value, subject to the total size of the selected items being at most W. Precisely, you are to compute a subset $I \subseteq \{1, \ldots, n\}$ which maximizes the sum
$$\sum_{i \in I} v_i$$
subject to the constraint $\sum_{i \in I} w_i \leq W$.
(a) Give a dynamic programming solution that runs in time $O(nW)$.
(b) Improve the running time to $O(n, \min\{W, 2^n\})$. ♢

Exercise 4.11: (Optimal line breaking) This book (and most technical papers today) is typeset using Donald Knuth's computer system known as TEX. This remarkable system produces very high quality output because of its sophisticated algorithms. One such algorithm is the way in which it breaks a paragraph into individual lines.

A **paragraph** can be regarded as a sequence of words. Suppose there are n words, and their lengths are $a_1, \ldots, a_n$. The problem is to break the paragraph into lines, no line having length more than m. Between 2 words in a line we introduce one space; there is no spaces after the last word in a line. If a line has length k, then we assess a **penalty** of $m-k$ on that line. The penalty for a particular method of breaking up a paragraph is the sum of the penalty over all lines. The last line of a paragraph, by definition, suffers no penalty.
(a) Consider the obvious greedy method to solve this problem (basically fill in each line until the next word will cause an overflow). Give an example to show that this does not always give the minimum penalty solution.
(b) Give a dynamic programming solution to finding the optimal (i.e., minimal penalty) solution.
(c) Illustrate your method with Lincoln's Gettysburg address, assuming that $m = 80$. In the case of a terminal word (which is followed by a full-stop), we consider the full stop as part of the word.
(d) Suppose we assume that there are 2 spaces separating a full-stop and the following word (if any) in the line. Modify your solution in (a) to handle this.
(e) Now introduce optional hyphenation into the words. For simplicity, assume that every word has zero or one potential place for hyphenation (the algorithm is told where this hyphen can be placed). If an input word of length ℓ can be broken into two half-words of lengths ℓ_1 and ℓ_2, respectively, it is assumed that $\ell_1 \geq 2$ and $\ell_2 \geq 1$. Furthermore, we must include an extra unit (for the placement of the hyphen character) in the length of the line that contains the first half. Can you modify the above algorithm further? ♢

End Exercises

§5. Optimal Parenthesization

We can view a triangulation of an $(n+1)$-gon to be a "parenthesized expression" on n symbols. Let us clarify this connection.

Let $(e_1, e_2, \ldots, e_n)$, $n \geq 1$, be a sequence of n symbols. A **(fully) parenthesized expression** on $(e_1, \ldots, e_n)$ is one whose atoms are e_i (for $i = 1, \ldots, n$), each e_i occurring exactly once and in this order left-to-right, and where each matched pair of parenthesis encloses exactly two non-empty subexpressions. E.g., there are exactly two parenthesized expressions on $(1, 2, 3)$:

$$((12)3), \qquad (1(23)).$$

The reader may verify that there are 5 parenthesized expressions on $(1, 2, 3, 4)$.

A parenthesized expression on $(e_1, \ldots, e_n)$ corresponds bijectively to a **parenthesis tree** on $(e_1, \ldots, e_n)$. Such a tree is a full[4] binary tree T on n leaves, where the ith leaf in symmetric order is associated with e_i. If $n = 1$, then the tree has only one node. Otherwise, the left and right subtrees are (respectively) parenthesized expressions on $(e_1, \ldots, e_i)$ and $(e_{i+1}, \ldots, e_n)$ for some $i = 1, \ldots, n$.

There is a slightly more involved bijective correspondence between parenthesis trees on $(e_1, \ldots, e_n)$ and triangulations of an abstract $(n+1)$-gon. See Figure 5 for an illustration. If the $(n+1)$-gon is $(v_0, v_1, \ldots, v_n)$,

[4] A node of a binary tree is **full** if it has two children. A binary tree is **full** if every internal node is full.

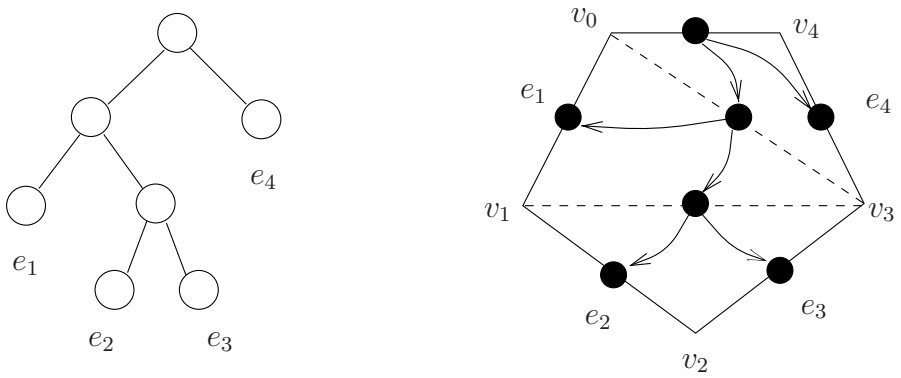


Figure 5: The parenthesis tree and triangulation corresponding to $((e_1(e_2e_3))e_4)$.

then the edges (v_{i-1}, v_i) is mapped to e_i $(i = 1, \ldots, n)$ under this correspondence, but the "distinguished edge" (v_0, v_n) is not mapped. We leave the details for an exercise.

If we associate a cost $W(i, j, k)$ for forming a parenthesis of the form "(E_1, E_2)" where E_1 (resp., E_2) is a parenthesized expression on $(e_i, \ldots, e_j)$ (resp., $(e_{j+1}, \ldots, e_k)$, then we may speak of the **cost** of a parenthesized expression – it is the same as the cost of the corresponding triangulation of P. Finding such an optimal parenthesized expression on $(e_1, \ldots, e_n)$ is clearly equivalent to finding an optimal triangulation of P.

Catalan numbers. It is instructive to count the number $P(n)$ of parenthesis trees on $n \geq 1$ leaves. In the literature, $P(n)$ is also denoted $C(n-1)$, in which case it is called a **Catalan number**. The index $n-1$ of the Catalan numbers is the number of pairs of parenthesis needed to parenthesize n symbols. Here $C(n) = 1, 1, 2, 5$ for $n = 0, 1, 2, 3$. Note that $C(0) = 1$, not 0. In general, for $n \geq 1$, the following recurrence is evident:

$$C(n) = \sum_{i=1}^{n} C(i-1)C(n-1-i). \tag{11}$$

We can interprete $C(n)$ as the number of binary trees with exactly n nodes (Exercise). In terms of $P(n)$, we get a similar recurrence:

$$P(n) = \sum_{i=1}^{n} P(i)P(n-i) \tag{12}$$

where we define $P(0) = 0$.

This recurrence has an elegant solution using generating functions (see Lecture VII),

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

By Stirling's approximation,

$$\binom{2m}{m} = \Theta\left(\frac{4^m}{\sqrt{m}}\right).$$

So $C(m)$ grows exponentially and there is no hope to find the optimal parenthesis tree by enumerating all parenthesis trees.

Matrix Chain Product. An instance of the parenthesis problem is the **matrix chain product** problem: given a sequence

$$A_1, \ldots, A_n$$

of rectangular matrices where A_i is $a_{i-1} \times a_i$ $(i = 1, \ldots, n)$, we want to compute the chain product

$$A_1 A_2 \cdots A_n$$

in the cheapest way. The sequence $(a_0, a_1, \ldots, a_n)$ of numbers is called the **dimension** of this chain product expression.

We need to clarify the cost model. Using associativity of matrix products, each method of computing this product corresponds to a distinct parenthesis tree on $(A_1, \ldots, A_n)$. For instance,

$$((A_1 A_2)A_3), \quad (A_1(A_2 A_3))$$

are the two ways of multiplying 3 matrices. We assume that it costs pqr to multiply a $p \times q$ matrix by a $q \times r$ matrix. Hence if the dimension of the chain product $A_1 A_2 A_3$ is (a_0, a_1, a_2, a_3), the first method to multiply these three matrices above costs

$$a_0 a_1 a_2 + a_0 a_2 a_3 = a_0 a_2 (a_1 + a_3)$$

while the second method costs

$$a_0 a_1 a_3 + a_1 a_2 a_3 = a_1 a_3 (a_0 + a_2).$$

Letting $(a_0, \ldots, a_3) = (1, d, 1, d)$, these two methods cost $2d$ and $2d^2$, respectively. Hence the second method may be arbitrarily more expensive than the first.

Corresponding to the dimension $(a_0, \ldots, a_n)$ of a chain product instance, we define an triangular weight function $W(i, j, k)$ for $0 \leq i < j < k \leq n$ to reflect our complexity model:

$$W(i, j, k) := a_i a_j a_k.$$

This is what we called the "product weight function" in §2. The optimal method of computing a matrix chain product is reduced to the optimal parenthesis tree problem. We have seen an $O(n^3)$ solution to this problem.

The original problem of matrix chain product can be solved in two stages: first find the optimal parenthesis tree, based on just the dimension of the chain. Then use the parenthesis tree to order the actual matrix multiplications. The only creative part of this solution is the determination of the optimal parenthesization.

Remark: For the product weight function, $W(a_i, a_j, a_k) = a_i a_j a_k$, the optimal triangulation problem can be solved in $O(n \log n)$ time, using a sophisticated algorithm due to Hu and Shing [4]. Ramanan [7] gave an exposition of this algorithm, and presented an $\Omega(n \log n)$ lower bound in an algebraic decision tree.

Exercises

Exercise 5.1: Show that $C(n)$ is the number of binary trees on n nodes. HINT: Use the recurrence (11) and structural induction on the definition of a binary tree. ♢

Exercise 5.2: Work out the bijective correspondence between triangulations and parenthesis trees stated above. ♢

Exercise 5.3: Verify by induction that $C(m)$ has the claimed solution. ♢

Exercise 5.4: Solve the recurrence (11) for $C(n)$ by using the following observation: consider generating function

$$G(x) = \sum_{i=0}^{\infty} C(i)x^i = 1 + x + 2x^2 + 5x^3 + \cdots .$$

HINT: What can you say about the coefficient of x^n in the squared generating function $G(x)^2$? Write this down as a recurrence equation involving $G(x)$ Solve this quadratic equation.

♢

Exercise 5.5: i) Consider an abstract n-gon whose weight function is a product function, $W(i,j,k) = w_i w_j w_k$ for some sequence $w_1, \ldots, w_n$ of non-negative numbers. Call w_i the "weight" of vertex i. Let $(\pi_1, \pi_2, \ldots, \pi_n)$ be a permutation of $\{1, \ldots, n\}$ such that

$$w_{\pi_1} \leq w_{\pi_2} \leq \cdots \leq w_{\pi_n}.$$

Show that there exists an optimal triangulation T of P such that vertex π_1 of least weight is **connected to** π_2 and also to π_3 in T. [We say vertex i is **connected to** j **in** T if either ij or ji is in T or is an edge of the n-gon.]
HINT: Use induction on n. Call a vertex i **isolated** if it is not connected to another vertex by a chord in T. Consider two cases, depending on whether π_1 is isolated in T or not.

ii) (Open) Can you exploit this result to obtain a $o(n^3)$ algorithm for the matrix chain product problem?

♢

End Exercises

§6. Optimal Search Trees

Suppose we store n keys

$$K_1 < K_2 < \cdots < K_n$$

in a binary search tree. The probability that a key K to be searched is equal K_i is $p_i \geq 0$, and the probability that K falls between K_j and K_{j+1} is $q_j \geq 0$. Naturally,

$$\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1.$$

In our formulation, we do not restrict the sum of the p's and q's to be 1, since we can simply interpret these numbers to be "relative weights". But we do require the q_j, p_i's to be non-negative.

We want to construct an full[5] binary search tree T whose nodes are labeled by

$$q_0, p_1, q_1, p_2, \ldots, q_{n-1}, p_n, q_n \tag{13}$$

in symmetric order. Note that the p_i's label the internal nodes and q_j's label the leaves.

[FIGURE]

In a natural way, T corresponds to a binary search tree in which the internal nodes are labeled by $K_1, \ldots, K_n$. But for our purposes, the actual keys K_i are irrelevant: only the probabilities p_i, q_j are of

[5] This amounts to an extended binary search tree, as described in Lecture 3.

interest. Each subtree $T_{i,j}$ $(1 \le i \le j \le n)$ of T corresponds to a binary search tree on the keys $K_i, \ldots, K_j$. We define the following **weight function**:

$$\begin{aligned} W(i-1,j) &:= q_{i-1} + p_i + q_i + \cdots p_j + q_j \\ &= q_{i-1} + \sum_{k=i}^{j} (q_k + p_k) \end{aligned}$$

for all $0 \le i \le j \le n$. Thus $W(i,i) = q_i$. The **cost** of T is given by

$$C(T) = W(0,n) + C(T_L) + C(T_R)$$

where T_L and T_R are the left and right subtrees of T. If T has only one node, then $C(T) = 0$, corresponding to the case where the node is labeled by some q_j. We say T is **optimal** if its cost is minimum. So the problem of **optimal search trees** is that of computing an optimal T, given the data in (13). Why is this definition of "cost" reasonable? Let us charge a unit cost to each node we visit when we lookup a key K. If K has the frequency distribution given by the probabilities p_i, q_j, then the expected charge to the root of T is precisely $W(i-1,j)$ if the leaves of T are $K_i, \ldots, K_j$. So $C(T)$ is the expected cost of looking up K in the search tree T.

Application. In constructing compilers for programming languages, we need a search structure for looking up if a given identifier K is a key word. Suppose $K_1, \ldots, K_n$ are the key words of our programming language and we have statistics telling us that an identifier K in a typical program is equal to K_i with probability p_i and lies between K_j and K_{j+1} with probability q_j. One solution to this compiler problem is to construct an optimal search tree for the key words with these probabilities.

Example. Assume that $(p_1, p_2, p_3) = (6, 1, 3)$ and the q_i's are zero. There are 5 possible search trees here (see figure 6). The optimal search tree has root labeled p_1, giving a cost of $6 + 2(3) + 3(1) = 15$. Note that the structurally "balanced tree" with p_2 at the root has a bigger cost of 19. Intuitively, we understand why it is better to have p_1 at the root – it has a much larger frequency than the other nodes.

Figure 6: The 5 possible binary search trees on (p_1, p_2, p_3).

Let us observe that the **dynamic programming principle** holds, *i.e.*, every subtree of $T_{i,j}$ $(1 \le i \le n)$ is optimal for its associated relative weights

$$q_{i-1}, p_i, q_i, \ldots, q_{j-1}, p_j, q_j.$$

Hence an obvious dynamic programming algorithm can be devised to find optimal search trees in $O(n^3)$ time. Exploiting additional properties of the cost function, Knuth shows this can be done in $O(n^2)$ time. The key to the improvement is due to a general inequality satisfied by the cost function, first clarified by F. Yao, which we treat next.

Exercises

Exercise 6.1: Describe the precise connection between the optimal search tree problem and the optimal triangularization problem. ♢

Exercise 6.2: Suppose the input frequencies are $(p_1, \ldots, p_n)$ (the q_i's are all zero). If the p_i's are distinct, Joe Quick has a suggestion: why not choose the largest p_i to be the root? Is this true for $n = 3$? Find the smallest n for which this is false, and provide a counter example for this n.

♢

Exercise 6.3: (Project) Collect several programs in your programming language X.
(a) Make a sorted list of all the key words in language X. If there are n key words, construct a count of the number of occurrences of these key words in your set of programs. Let $p_1, p_2, \ldots, p_n$ be these frequencies.
(b) Construct an optimum search tree for these key words (assuming q_i's are 0) these key words (assuming q_i's are 0).
(c) Construct from your programs the frequencies that a non-key word falls between the keywords, and thereby obtain $q_0, q_1, \ldots, q_n$. Construct an optimum search tree for these p's and q's. ♢

Exercise 6.4: The following class of recurrences was investigated by Fredman [2]:

$$M(n) = g(n) + \min_{0 \le k \le n-1} \{\alpha M(k) + \beta M(n-k-1)\}$$

where $\alpha, \beta > 0$ and $g(n)$ are given. This is clearly related to optimal search trees. We focus on $g(n) = n$.
(a) Suppose $\min\{\alpha, \beta\} < 1$. Show that $M(n) \sim \frac{n}{1-\min\{\alpha,\beta\}}$.
(b) Suppose $\min\{\alpha, \beta\} > 1$, $\log \alpha / \log \beta$ is rational and $\alpha^{-1} + \beta^{-1} = 1$. Then $M(n) = \Theta(n^2)$. ♢

Exercise 6.5: If the p_i's are all zero in the Optimal Search Tree problem, then the optimization criteria amounts to minimizing the external path length. Recall that the external path length of a tree whose leaves are weighted is equal to $\sum_u d(u)w(u)$ where u ranges over the leaves, with $w(u), d(u)$ denoting the weight and depth of u. Suppose we consider a **modified path length** of a leaf u to be $w(u) \sum_{i=0}^{d(u)} 2^{-i}$ (instead of $d(u)w(u)$). Solve the Optimal Search Tree under this criteria. REMARK: This problem is motivated by the processing of cartographic maps of the counties in a state. We want to form a hierarchical level-of-detail map of the state by merging the counties. After the merge of a pair of maps, we always simplify the result by discarding some details. If the weight of a map is the number of edges or vertices in its representation, then after a simplification step, we are left with half as many edges.
♢

Exercise 6.6: Consider the following generalization of Optimal Binary Trees. We are given a subdivision of the plane into simply connected regions. Each region has a positive weight. We want to construct a binary tree T with these regions as leaves subject to one condition: each internal node u of T determines a subregion R_u of the plane, obtained as the union of all the regions below u. We require R_u to be simply-connected. The cost of T is as usual the external path length (i.e., sum of the weights of each leaf multiplied by its depth).
(a) Show that this problem is NP-complete.
(b) Give provably good heuristics for this problem. ♢

End Exercises

§7. Weight Matrices

We reformulate the optimal search tree problem in an abstract framework.

Definition 1 *Let $n \geq 2$ be an integer. A* **triangular function** *W (of order n) is any partial function with domain $[0..n] \times [0..n]$ such $W(i,j)$ is defined iff $i \leq j$. We call W a* **weight matrix** *if it is a triangular function whose range is the set of non-negative real numbers. A quadruple (i,i',j,j') is* **admissible** *if*

$$0 \leq i \leq i' \leq j \leq j' \leq n.$$

We say W is **monotone** *if*

$$W(i',j) \leq W(i,j')$$

for all admissible (i,i',j,j'). The **quadrangle inequality for** *W for (i,i',j,j') is*

$$W(i,j) + W(i',j') \leq W(i,j') + W(i',j).$$

We say W is **quadrangular** *if it satisfies the quadrangular inequality for all admissible (i,i',j,j').*

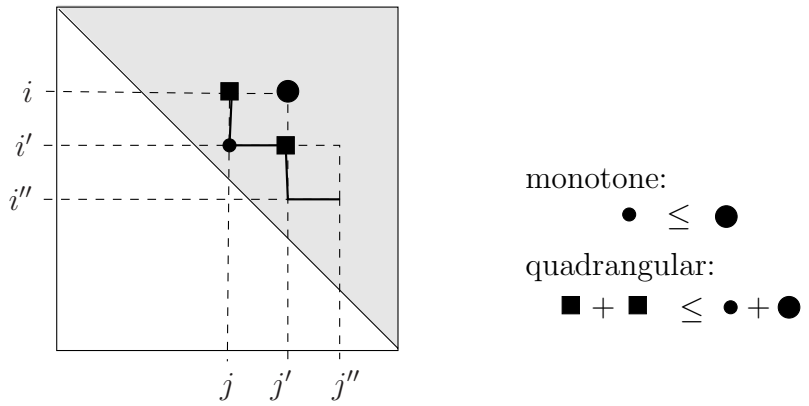


Figure 7: Monotone and quadrangular weight matrix.

It is sometimes convenient to write W_{ij} or $W_{i,j}$ instead of $W(i,j)$. If we view W_{ij} as the (i,j)-th entry of an n-square matrix W, then W is upper triangular matrix. Note that (i,i',j,j') is admissible iff the four points $(i,j),(i',j),(i,j'),(i',j')$ are all on or above the main diagonal of W (see Figure 7). Monotonicity and quadrangularity is also best seen visually (cf. Figure 7):

- Monotonic means that along any north-eastern path in the upper triangular matrix, the matrix values are non-decreasing.
- Quadrangularity means that for any 4 corner entries of a rectangle lying on or above the main diagonal, the south-west plus the north-east entries are not less than the sum of the other two.

Example: In the optimal search tree problem, the weight function W is implicitly specified by $O(n)$ parameters, *viz.*, $q_0, p_1, q_1, \ldots, p_n, q_n$, with

$$W(i,j) = \sum_{k=i-1}^{j} q_k + \sum_{k=i}^{j} p_k.$$

In this case, $W(i,j)$ can be computed in linear time from the q_k's and p_k's. The point is that, depending on the representation, $W(i,j)$ may not be available in constant time. The following is left as an exercise:

Lemma 2 *The weight matrix for the optimal search tree problem is both monotone and quadrangular. In fact, the quadrangular inequality is an equality.*

Definition 2 *Given a weight matrix W, its* **derived weight matrix** *is the triangular function*

$$W^* : [0..n]^2 \to \mathbb{R}_{\geq 0}$$

is defined as follows:

$$W^*(i,i) := W(i,i).$$

Assuming that $W^*(i,j)$ has been defined for all $j - i < \ell$, define

$$W^*(i,i+\ell) := W(i,i+\ell) + \min_{i<k\leq i+\ell}\{W^*(i,k-1) + W^*(k,i+\ell)\}.$$

Defining

$$W^*(i,j;k) := W(i,j) + W^*(i,k-1) + W^*(k,j), \tag{14}$$

*we call k an (i,j)-***splitter** *if $W^*(i,j) = W^*(i,j;k)$.*

Note: the literature (especially in operations research) describes the Monge property of matrices. This turns out to be the quadrangle inequality restricted to admissible quadruples (i,i',j,j') where $i' = i+1$ and $j' = j+1$.

Exercises

Exercise 7.1: (a) Computer the derived matrix of the following weight matrices:

$$W_1 = \begin{array}{|c|c|c|c|}\hline 1 & 1 & 1 & 1 \\ \hline & 2 & 2 & 2 \\ \hline & & 3 & 3 \\ \hline & & & 4 \\ \hline \end{array}, \qquad W_2 = \begin{array}{|c|c|c|c|c|}\hline 1 & 2 & 1 & 2 & 1 \\ \hline & 2 & 0 & 3 & 2 \\ \hline & & 1 & 0 & 1 \\ \hline & & & 4 & 2 \\ \hline & & & & 2 \\ \hline \end{array}.$$

(b) Suppose $W(i,j) = a_i$ for $i = j$ and $W(i,j) = 0$ for $i \neq j$. The a_i's are arbitrary constants. Succinctly describe the matrix W^*. ♢

Exercise 7.2: (Lemma 2) Verify that the weight matrix for the optimal search tree problem is indeed monotone and satisfies the quadrangular *equality*. ♢

Exercise 7.3: Write a program to compute the derivative of a matrix. It should run in $O(n^3)$ time on an n-square matrix. ♢

Exercise 7.4:

(a) Interpret the derived matrix for the optimal search tree problem.

(b) Does the derived matrix of a derived matrix have a realistic interpretation? ♢

Exercise 7.5: Generalize the concept of a triangular function W so that its domain is $[0..n]^k$ for any integer $k \geq 2$, and $W(i_1, \ldots, i_k)$ is defined iff $i_1 \leq i_2 \leq \cdots \leq i_k$. Then W is a **weight function** (of **order** n and **dimension** k) if it is triangular and has range over the non-negative real numbers. Formulate the "optimal k-gonalization" problem for an abstract n-gon. (This seeks to partition an n-gon into ℓ-gons where $3 \leq \ell \leq k$. Give a dynamic programming solution. ♢

End Exercises

§8. Quadrangular Inequality

The quadrangular inequality is central in the $O(n^2)$ solution of the optimal search tree problem. We will show two key lemmas.

Lemma 3 *If W is monotone and quadrangular, then the derived weight matrix W^* is also quadrangular.*

Proof. We must show the quadrangular inequality

$$W^*(i,j) + W^*(i',j') \leq W^*(i,j') + W^*(i',j), \qquad (0 \leq i \leq i' \leq j \leq j' \leq n). \tag{15}$$

First, we make the simple observation when $i = i'$ or $j = j'$, the inequality in equation (15) holds trivially.

The proof is by induction on $\ell = j' - i$. The basis, when $\ell = 1$, is immediate from the previous observation, since we have $i = i'$ or $j = j'$ in this case.

Case $i < i' = j < j'$: So we want to prove that $W^*(i,j) + W^*(j,j') \leq W^*(i,j') + W^*(j,j)$. Let $W^*(i,j') = W(i,j';k)$ and initially assume $i < k \leq j$. Then

$$\begin{array}{rcll} W^*_{i,j} + W^*_{j,j'} & \leq & [W_{i,j} + W^*_{i,k-1} + W^*_{k,j}] + W^*_{j,j'} & \text{(expanding } W^*_{i,j}\text{)} \\ & \leq & W_{i,j'} + W^*_{i,k-1} + [W^*_{k,j} + W^*_{j,j'}] & \text{(by monotonicity)} \\ & \leq & [W_{i,j'} + W^*_{i,k-1} + W^*_{k,j'}] + W^*_{j,j} & \text{(by induction)} \\ & = & W^*_{i,j'} + W^*_{j,j} & \text{(by choice of } k\text{)}. \end{array}$$

In case $j < k \leq j'$, we would initially expand $W^*_{j,j'}$ above.

Case $i < i' < j < j'$: Let $W^*(i,j') = W(i,j';k)$ and $W^*(i',j) = W(i',j;\ell)$ and initially assume $k \leq \ell$. Then

$$\begin{array}{rcll} W^*_{i,j} + W^*_{i',j'} & \leq & [W_{i,j} + W^*_{i,k-1} + W^*_{k,j}] + [W_{i',j'} + W^*_{i',\ell-1} + W^*_{\ell,j'}] & \text{(since } i < k \leq j, i' < \ell \leq j'\text{)} \\ & \leq & [W_{i,j'} + W_{i',j}] + W^*_{i,k-1} + W^*_{i',\ell-1} + [W^*_{k,j} + W^*_{\ell,j'}] & (W \text{ is quadrangular}) \\ & \leq & [W_{i,j'} + W_{i',j}] + W^*_{i,k-1} + W^*_{i',\ell-1} + [W^*_{k,j'} + W^*_{\ell,j}] & \text{(induction on } (k,\ell,j,j')\text{)} \\ & \leq & [W_{i,j'} + W^*_{i,k-1} + W^*_{k,j'}] + [W_{i',j} + W^*_{i',\ell-1} + W^*_{\ell,j}] & \\ & = & W^*_{i,j'} + W^*_{i',j} & \text{(by choice of } k,\ell\text{)}. \end{array}$$

In case $\ell < k$, we can begin as above with the initial inequality $W^*(i,j) + W^*(i',j') \leq W^*(i,j;\ell) + W^*(i',j';k)$. **Q.E.D.**

Splitting function K_W. The (i,j)-splitter k is not unique but we make it unique in the next definition by choosing the largest such k.

Definition 3 *Let W be an weight matrix. Define the* **splitting function** K_W *to be a triangular function*

$$K_W : [0..n]^2 \to [0..n]$$

defined as follows: $K_W(i,i) = i$ and for $0 \le i < j \le n$,

$$K_W(i,j) := \max\{k : W^*(i,j) = W(i,j;k)\}.$$

We simply write $K(i,j)$ for $K_W(i,j)$ when W is understood. Once the function K_W is determined, it is a straightforward matter to compute the derived matrix of W The following is the key to a faster algorithm.

Lemma 4 *If the derived weight matrix of W is quadrangular, then for all $0 \le i \le j < j$,*

$$K_W(i,j) \le K_W(i,j+1) \le K_W(i+1,j+1).$$

Proof. By symmetry, it suffices to prove that

$$K(i,j) \le K(i,j+1). \tag{16}$$

This is implied by the following claim: if $i < k \le k' \le j$ then

$$W^*(i,j;k') \le W^*(i,j;k) \quad \text{implies} \quad W^*(i,j+1;k') \le W^*(i,j+1;k). \tag{17}$$

To see the implication, suppose equation (16) fails, say $K(i,j) = k' > k = K(i,j+1)$. Then the claim implies $K(i,j+1) \ge k'$, contradiction.

It remains to show the claim. Consider the quadrangular inequality for the admissible quadruple $(k,k',j,j+1)$,

$$W^*(k,j) + W^*(k',j+1) \le W^*(k,j+1) + W^*(k',j).$$

Adding $W(i,j) + W(i,j+1) + W^*(i,k-1) + W^*(i,k'-1)$ to both sides, we obtain

$$W^*(i,j;k) + W^*(i,j+1;k') \le W^*(i,j+1;k) + W^*(i,j;k').$$

This implies equation (17). **Q.E.D.**

Main result. The previous lemma gives rise to a faster dynamic programming solution for monotone quadrangular weight functions.

Theorem 5 *Let W be weight matrix such that $W(i,j)$ can be computed in constant time for all $1 \le i \le j \le n$, and its derived matrix W^* is quadrangular. Then its derived matrix W^* and the splitting function K_W can be computed in $O(n^2)$ time and space.*

Proof. We proceed in stages. In stage $\ell = 1, \ldots, n-1$, we will compute $K(i,i+\ell)$ and $W^*(i,i+\ell)$ (for all $i = 0, \ldots, n-\ell$). It suffices to show that each stage takes takes $O(n)$ time. We compute $W^*(i,i+\ell)$ using the minimization

$$W^*(i,i+\ell) = \min\{W(i,i+\ell;k) : K(i,i+\ell-1) \le k \le K(i+1,i+\ell)\}.$$

This equation is justified by the previous lemma, and it takes time $O(K(i+1, i+\ell) - K(i, i+\ell-1) + 1)$. Summing over all $i = 1, \ldots, n-\ell$, we get the telescoping sum

$$\sum_{i=1}^{n-\ell} [K(i+1, i+\ell) - K(i, i+\ell-1) + 1] = n - \ell + K(n-\ell+1, n) - K(1, \ell) = O(n).$$

Hence stage ℓ takes $O(n)$ time. **Q.E.D.**

Remarks. We refer to [5] for a history of this problem and related work. The original formulation of the optimal search tree problem assumes p_i's are zero. For this case, T.C. Hu has an non-obvious algorithm that Hu and Tucker were able to show runs correctly in $O(n \log n)$ time. Mehlhorn [6] considers "approximate" optimal trees and show that these can be constructed in $O(n \log n)$ time. He describes a solution to the "approximate search tree" problem in which we dynamically change the frequencies; see "Dynamic binary search", (**SIAM J.Comp.**,8:2(1979)175–198). M. R. Garey gives an efficient algorithm when we want the optimal tree subject to a depth bound; see "Optimal Binary Search Trees with Restricted Maximum Depth, (**SIAM J.Comp.**,3:2(1974)101-110).

Exercises

Exercise 8.1: (a) Compute the optimal binary tree for the following sequence:

$$(q_0, p_1, q_1, \ldots, p_{10}, q_{10}) = (1, 2, 0, 1, 1, 3, 2, 0, 1, 2, 4, 1, 3, 3, 2, 1, 2, 5, 1, 0, 2).$$

(b) Compute the optimal binary tree for the case where the q's are the same as in (a), namely,

$$(q_0, q_1, \ldots, q_{10}) = (1, 0, 1, 2, 1, 4, 3, 2, 2, 1, 2)$$

and the p's are 0. ♢

Exercise 8.2: It is actually easy to give a "graphical" proof of lemma 4. In the figure 8, this amounts to showing that if $A + a \geq B + b$ then $A' + a' \geq B' + b'$.

♢

Exercise 8.3: If W is monotone and quadrangular, is W^* monotone? ♢

Exercise 8.4: Consider a binary search tree that has this shape (essentially a linear list):

Show that the following set of inequalities is necessary and sufficient for the above search tree to be optimal:

$$\begin{array}{lcll} p_2 + q_2 & \geq & p_1 + q_0 & (E_2) \\ p_3 + q_3 & \geq & p_2 + q_1 + p_1 + q_0 & (E_3) \\ \cdots & & & \\ p_n + q_n & \geq & p_{n-1} + q_{n-2} + p_{n-2} + \cdots + p_1 + q_0 & (E_n) \end{array}$$

HINT: use induction to prove sufficiency.
Remark: So search trees with such shapes can be verified to be optimal in linear time. In general, can an search tree be verified to be optimal in $o(n^2)$ time? ♢

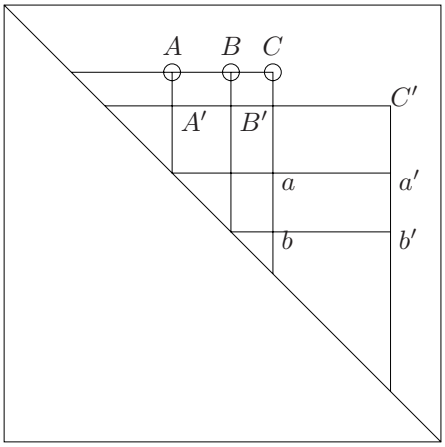


Figure 8: Derived weight matrix.

Exercise 8.5: (a) Generalize the above result so that all the internal nodes to the left of the root are left-child of its parent, and all the internal nodes to the right of the root are right-child of its parent. (b) Can you generalized this to the case where all the internal nodes lie on one path (ignoring directions along the tree edges – the path first traverses up the tree to the root and then down the tree again). ♢

Exercise 8.6: Given a sequence $a_1, \ldots, a_n$ of real numbers. Let $A_{ij} = \sum_{k=i}^{j} a_k$, $B_{ij} = \min\{A_{kj} : k = i, \ldots, j\}$ and $B_j = B_{1j}$. Compute the values $B_1, \ldots, B_n$ in $O(n)$ time. ♢

End Exercises

§9. Conclusion

This chapter shows the versatility of the on dynamic programming approach to a variety of problems. A serious drawback of dynamic programming is its high polynomial cost, $O(n^k)$ for $k \geq 2$, in both time and space. Hence there is interest in exploiting "sparsity conditions" when they occur. Sometimes, the implicit matrix to be searched has special properties (Monge conditions). See the survey of Giancarlo [3] for such examples.

References

[1] A. Apostolico and Z. Galil, editors. *Pattern Matching Algorithms*. Oxford University Press, 1997.

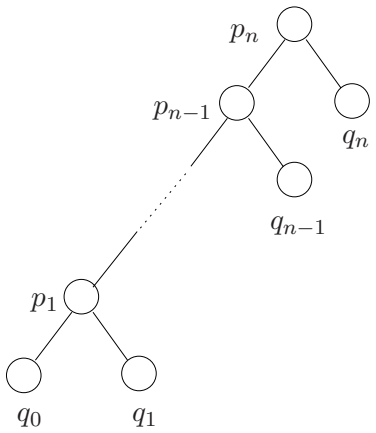


Figure 9: Linear list search tree.

[2] M. L. Fredman. *Growth Properties of a class of recursively defined functions*. PhD thesis, Stanford University, 1972. Technical Report No. STAN-CS-72-296. PhD Thesis.

[3] R. Giancarlo. Dynamic programming: Special cases. In A. Apostolico and Z. Galil, editors, *Pattern Matching Algorithms*, pages 201–232. Oxford University Press, 1997.

[4] T. C. Hu and M.-T. Shing. An $O(n)$ algorithm to find a near-optimum partition of a convex polygon. *J. Algorithms*, 2:122–138, 1981.

[5] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Boston, 1972.

[6] K. Mehlhorn. *Datastructures and Algorithms 1: Sorting and Sorting*. Springer-Verlag, Berlin, 1984.

[7] P. Ramanan. A new lower bound technique and its application: Tight lower bound for a polygon triangulation problem. *SIAM J. Computing*, 23:834–851, 1994.