"It is one of the striking generalizations of biochemistry – which surprisingly is hardly ever mentioned in the biochemical text-books – that the twenty amino acids and the four bases, are, with minor reservations, the same throughout Nature. As far as I am aware the presently accepted set of twenty amino acids was first drawn up by Watson and myself in the summer of 1953 in response to a letter of Gamow's."

— Francis Crick, *On the Genetic Code*
Nobel Lecture, 11 December 1962

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.

¶1. From Google to Genomics. Dynamic programming techniques are particularly effective for problems on strings, i.e., sequences of symbols from some alphabet. Currently, there are two major consumers of string algorithms: search engines such as Google, and computational biology. Thus, if you ask Google to search the word `strnigs`, it will ask[2] if you meant `strings`. You can be sure that a slew of string algorithms are at work behind this innocent response. Or, when I search for `CGTAATCC`, Google asked[3] if I meant `CCGTCC`. It turns out that `CCGTCC.com` is the homepage for members of *Casino Chip & Gaming Token Collectors Club*. But a biologist might submit the sequence `CGTAATCC` to a database engine to find the closest match. This is because in computational genomics, a DNA sequence is just a string over the symbols `A,C,T,G`. The strings in Google and genomics have different characteristics: Google strings are words or phrases – these are much shorter than strings in biology which represent DNA or RNA sequences whose lengths go into millions. Google strings have medium size alphabets while strings in genomics have small alphabet (size 4). If we were looking at protein sequences, the alphabet size would be 20. The corresponding algorithms should try to exploit such properties.

Whether we are talking about strings in Google search or in genomics, the ability in both cases to show you closely related strings meant that these algorithms have (1) some measure of similarity or distance between strings, and (2) some database of strings in which to look for similar strings. We shall look at two notions of similarity of strings in this chapter. Computing these similarity measures efficiently calls for dynamic programming techniques.

In most applications of dynamic programming, the underlying objects have some kind of linear structure, much like strings. Other classes of such objects include polygons and binary trees. Thus, we will look at corresponding problems of optimal triangulation of (abstract) polygons, and the constructing optimal binary search trees.

[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.

[2] That was in 2008. In 2011, it no longer asks, but lists some possibilities like `string cheese`, `string theory`, `stringbuilder`, etc.

[3] That was in 2008. In 2011, it asked if I wanted `CHEATCC` which led to websites with video game cheats, cheat codes, hints and tips.

§1. First Glimpses of Dynamic Programming

¶2. Divide and Conquer with a twist. Dynamic programming is a form of divide-and-conquer because it is based on solving subproblems. But it has some rather distinctive features. A simple illustration is provided by the computation of Fibonacci numbers, $F(n) = F(n-1) + F(n-2)$. On input $n > 1$, the obvious recursive algorithm calls itself twice on the arguments $n-1$ and $n-2$. The returned results are added together. The running time is given by the recurrence $T(n) = T(n-1)+T(n-2)+1$. Thus $T(n)$ is exponential (§III.6, AVL trees). A little reflection shows that linear time suffices: instead of computing $F(n)$, let us define a new function $F_2(n)$ to compute the pair $(F(n), F(n-1))$ of consecutive Fibonacci numbers. To compute $F_2(n)$, we only need one recursive call to $F_2(n-1)$:

```
F_2(n):
    If (n = 1), Return(1, 0)      ◁ Assume input n is ≥ 1
    (a, b) ← F_2(n − 1)           ◁ Recursive call!
    Return(a + b, a)
```

The running time recurrence now satisfies the recurrence $T_2(n) = T_2(n-1) + 1 = n$. Here we see the seed of the dynamic programming idea — that by keeping around solutions to subproblems, we can avoid recomputing them and avoid what would otherwise be an exponential complexity. In the Fibonacci number computation, we only keep the solution to two subproblems. In this sense, Fibonacci numbers is not typical of dynamic programming problems, which typically need solutions to a polynomial number of subproblems.

¶3. Joy Ride, again. Recall the joy ride or linear bin packing problem in Chapter V. The input are the weights $(w_1, \ldots, w_n)$ of a queue of riders. We want to place these riders into a minimum number of cars, where each car has a weight capacity of M. Riders must be placed into cars in their queue order. The new twist here is that we allow negative weights (clearly our joy ride interpretation is stretched by this generalization). In any case, the greedy algorithm breaks down. For instance let $M = 5$ and $w = (5, 5, 5, 5, -20)$. The greedy solution has 4 cars $(5), (5), (5), (5, -20)$ but the optimal solution uses only one car. But to achieve this optimal solution, we must give up our online requirement (i.e., to decide on each rider without looking at what comes after in the queue). In this example, the optimal solution has to look at the entire queue before it can properly decide on the second rider (whether this rider should be in the first or second car). Thus, we must content ourselves with designing an **offline algorithm** in which each decisions can be based on the whole input.

Ah, negative weights are children with helium balloons!

We now give an $O(n^2)$ solution for the offline linear bin packing problem. But first, we must generalize the problem so that, instead of just solving the instance $P_n = (w_1, \ldots, w_n)$, we simultaneously solve a sequence $P_1, P_2, \ldots, P_n$ of subproblem instances, where $P_i = (w_1, \ldots, w_i)$. Let b_i be the minimum number of cars for instance P_i. We also define $b_0 := 0$. Now the last car for instance P_n has the form $(w_i, \ldots, w_n)$ for some i where $w_i + w_{i+1} + \cdots + w_n \leq M$. This justifies the following formula:

$$b_n = 1 + \min_{i=1,\ldots,n} \{b_{i-1} : \sum_{j=i}^{n} w_j \leq M\}. \tag{1}$$

Assuming $b_0, b_1, \ldots, b_{n-1}$ have been computed, we can compute b_n using this formula in $O(n)$ time. For instance, suppose $M = 5$ and $w = (1, 5, -2, 5, 1)$ Then $b_1 = 1$ (obviously), $b_2 = 2$, $b_3 = 1$ and $b_4 = 2$. Let us compute b_5 using the formula (1):

$$b_5 \leftarrow 1 + \min\{b_4, b_2\} = 1 + \min\{2, 2\} = 3.$$

So 3 cars is the optimal solution. Observe that if you were allowed to re-arrange the weights, then 2 cars would suffice; but that is not allowed in linear bin packing. We may program this solution as follows:

```
LINEAR BIN PACKING WITH NEGATIVE WEIGHTS:
    Input: array w[1..n] containing weights and M
    Output: array b[0..n] to store the values of optimal values b_i
        b[0] ← 0.
        for k = 1, ..., n
            W ← 0
            B ← +∞    ◁ B is current min value of b_k's
            for i = k, k−1, ..., 2, 1
                W ← W + w[i]
                If (W ≤ M) then B ← min{B, b[i−1]}
            b[k] ← 1 + B
```

Let $t(k)$ be the complexity of the k-th iteration of the outer for-loop. Clearly, $t(k) = \Theta(k)$. The overall complexity is $T(n) = \sum_{k=1}^{n} t(k) = \Theta(n^2)$.

This example is more typical of dynamic programming: the solution for problem instance P_n can be efficiently computed from the solutions to a polynomial number of subproblems (in this case, $P_1, \ldots, P_n$). In constrast, the problems with running times that satisfy the Master recurrence have a bounded number of subproblem instances.

¶4. String Notations. Let us fix some common terminology for strings. An **alphabet** is just a finite set Σ; its elements are called **letters** (or characters or symbols). A **string** (or word) is just a finite sequence of letters. The set of strings over Σ is denoted Σ^*. Let $X = x_1 x_2 \cdots x_m$ be a string where $x_i \in \Sigma$. The **length** of X is m, denoted $|X|$. Note that $|X|$ should not be confused with the usual notation $|S|$ for the cardinality of a set S. The **empty string** is denoted ϵ and it has length $|\epsilon| = 0$. Using an array-like notation, the ith letter of X is denoted $X[i] = x_i$ ($i = 1, \ldots, m$). Concatenation of two strings X, Y is indicated by juxtaposition, XY or sometimes $X; Y$. Thus $|XY| = |X| + |Y|$.

EXERCISES

Exercise 1.1: Let us probe what Google is doing with strings. The problem of transposing two consecutive letters in a string (i.e., a digraph) is a common human error in typing. Let us see if Google is looking out for this error:

Start with the sequence `string`. For each of the 5 digraphs in this sequence, we transpose them to get another string: `tsring, srting, stirng, strnig, strign`. Which of these does Google think is a mistype of `string`? Let us do the same experiment, but starting with the sequence `strings`. ♢

Exercise 1.2: Compare the different search engines: Google, Yahoo, Bing, Amazon.com, eBay, Twitter, Wikipedia(en). ♢

END EXERCISES

§2. Longest Common Subsequence

Many string problems come down to comparing two strings for similarity. In this Lecture, we will look at two such measures. The first of these measures captures the idea that two strings are similar if they have "substantial overlap". For instance, the two strings `notations` and `notions` clearly have much overlap. But do `notations` and `onttois` have as much overlap? This might be unclear, so we will introduce the concept of "subsequences" to give precise meaning to the overlap idea.

A **subsequence** of $X = x_1, \ldots, x_m$ is a string $Z = z_1 z_2 \cdots z_k$ 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 **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 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. Also, let $lcs(X, Y)$ denote[4] any element of $LCS(X, Y)$: so $lcs(X, Y) \in LCS(X, Y)$. Define the numerical functions $L(X, Y) := |lcs(X, Y)|$ (length function) and $\lambda(X, Y) := |LS(X, Y)|$ (cardinality function). Note that $\lambda(X, Y) \geq 1$ since "at worst", $LCS(X, Y)$ is the singleton comprising the empty string ϵ.

For example, if

$$X = \texttt{longest}, \qquad Y = \texttt{lengthen} \tag{2}$$

then $LCS(X, Y) = \{\texttt{lngt}, \texttt{lnge}\}$, $\lambda(X, Y) = 2$ and $L(X, Y) = 4$.

Of course, the ultimate in similarity under LCS measure is when $L(X, Y) = \min\{|X|, |Y|\}$. We also mention the related concept of "substring". A subsequence Z is called a **substring** of X if $X = Z'ZZ''$ for some strings Z', Z''. For instance, `on` and `g` are substrings of `long` but `ln`, `lg` and `log` are not. Thus, substrings are subsequences but the converse may not hold.

¶5. Versions of LCS Problems. 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) Compute $L(X, Y)$
  e.g., $L(\texttt{longest}, \texttt{lengthen}) = 4$.
- (Instance version) Compute $lcs(X, Y)$
  e.g., $lcs(\texttt{longest}, \texttt{lengthen}) = \texttt{lngt}$ or `lnge`.
- (Cardinality version) Compute $\lambda(X, Y)$
  e.g., $\lambda(\texttt{longest}, \texttt{lengthen}) = 2$.
- (Set version) Compute $LCS(X, Y)$
  e.g., $LCS(\texttt{longest}, \texttt{lengthen}) = \{\texttt{lngt, lnge}\}$.

[4] Clearly $lcs(X, Y)$ is not really a functional notation.

We will mainly focus on the first two versions. The last version can be exponential if members of the set $LCS(X,Y)$ are explicitly written out; we may prefer some "reasonably explicit" representation[5] of $LCS(X,Y)$. We will consider representations of $LCS(X,Y)$ below. See the Exercise for the multiset interpretation of $LCS(X,Y)$.

¶6. Exponential nature of $\lambda(X,Y)$. A brute force solution to the cardinality 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. But can $\lambda(X,Y)$ be truly exponential? Indeed, here is an example: let

$$X_n = 01a01a01a\ldots = (01a)^n, \qquad Y_n = 10a10a10a\ldots = (10a)^n. \tag{3}$$

We claim that $L(X_n, Y_n) = 2n$. It follows that the following 4 strings belong to $LCS(X_2, Y_2)$: $0a0a$, $0a1a$, $0a1a$, $1a1a$. More generally, we have $\lambda(X_n, Y_n) \geq 2^n$ since we can match all the a's in X_n and Y_n, and in each 01-block of X_n, we have 2 choices for matching the corresponding 10-block of Y_n. But we do not claim that $\lambda(X_n, Y_n) = 2^n$. Indeed, $\lambda(X_n, Y_n) = \Theta(4^n/\sqrt{n})$ (see Exercise).

¶7. The Dynamic Programming Principle for LCS. The following is a 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 $L(X,Y)$:

$$L(X,Y) = \begin{cases} 0 & \text{if } mn = 0 \\ 1 + L(X',Y') & \text{if } x_m = y_n \\ \max\{L(X',Y), L(X,Y')\} & \text{if } x_m \neq y_n \end{cases} \tag{4}$$

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

$$L(X,Y) = \max\{1 + L(X',Y'), L(X',Y), L(X,Y')\}. \tag{5}$$

The right hand side in (5) simplifies to the form in (4) because of

$$L(X',Y) \leq 1 + L(X',Y'), \tag{6}$$

and a similar inequality involving $L(X,Y')$. Formula (4) 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.

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 $L(X,Y)$ suggests the following collection of subproblem instances:

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

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

$$LCS(X_0, Y_j) = \{\epsilon\}, \qquad L(X_0, Y_j) = 0. \tag{7}$$

There are dynamic principles for $lcs(X,Y)$ and $LCS(X,Y)$ that are analogous to (4). Here we present the recursive formula for $LCS(X,Y)$, leaving $lcs(X,Y)$ as an exercise.

[5] See §14 for what this means.

$$LCS(X,Y) = \begin{cases} \{\epsilon\} & \text{if } mn = 0 \\ LCS(X',Y')x_m & \text{if } x_m = y_n \\ LCS(X',Y) & \text{if } x_m \neq y_n, \quad L(X',Y) > L(X,Y') \\ LCS(X,Y') & \text{if } x_m \neq y_n, \quad L(X,Y') > L(X',Y) \\ LCS(X,Y') \cup LCS(X',Y) & \text{if } x_m \neq y_n, \quad L(X,Y') = L(X',Y). \end{cases} \tag{8}$$

This formula can be viewed as an expansion of the three cases in the recursive formula for $L(X,Y)$ in (4). In particular, the case $x_m \neq y_n$ has been expanded into three subcases. Moreover, each of these subcases are clearly necessary. But the case $x_m = y_n$ ($= b$, say) is not entirely obvious. At a first glance, it seems that this case should be split into four subcases (in analogy to (5)):

$$LCS(X'b,Y'b) = \begin{cases} LCS(X',Y')b & \text{if } L(X,Y) > \max\{L(X',Y), L(X,Y')\} \\ LCS(X',Y')b \cup LCS(X',Y) & \text{if } L(X,Y) = L(X',Y) > L(X,Y') \\ LCS(X',Y')b \cup LCS(X,Y') & \text{if } L(X,Y) = L(X,Y') > L(X',Y) \\ LCS(X',Y')b \cup LCS(X,Y') \cup LCS(X',Y) & \text{if } L(X,Y) = L(X,Y') = L(X',Y) \end{cases}$$

To prove that these subcases are unnecessary, we claim:

$$LCS(X'b,Y'b) = LCS(X',Y')b. \tag{9}$$

One direction of this proof is easy: clearly, $LCS(X'b,Y'b)$ contains $LCS(X',Y')b$. Conversely, suppose $w \in LCS(X'b,Y'b)$. We must show that $w \in LCS(X',Y')b$. Write $w = w'c$ for some $c \in \Sigma$. We have two possibilities: (1) Suppose $c \neq b$. Then $w'c$ is a common subsequence of X' and Y'. Then $w'cb$ is a common subsequence of $X'b$ and $Y'b$. This contradicts the assumption that $w = w'c \in LCS(X'b,Y'b)$. (2) Suppose $c = b$. Then w' is a common subseqeunce of X' and Y'. Since $w'c$ is the longest common subseqeunce of $X'b$ and $Y'b$, we conclude that w' must be the longest common subseqeunce of X' and Y'. This implies $w \in LCS(X',Y')b$, as desired.

Simplification: The student should compare Equations (4) and (8) to see the relative simplicity of the former equation. Also the recurrence (8) tells us that the flow of control in the algorithm for $LCS(X,Y)$ is determined by the function $L(X,Y)$. In particular, we need to compute $L(X,Y)$ if we want to compute $LCS(X,Y)$. In fact, equations (4) and (8) share a common flow of control, with some refinements for $LCS(X,Y)$. Our strategy is to develop an algorithm for $L(X,Y)$ first. Then we indicate the necessary modifications to yield an algorithm for $LCS(X,Y)$. Such a modification is usually straightforward although we will see exceptions: see the $lcs(X,Y)$ in small space solution below.

¶8. Matrix encoding of subsolutions. To organize the dynamic programming solution for $L(X,Y)$, we use an $(1+m) \times (1+n)$ matrix $L[0..m, 0..n]$ where the (i,j)th entry $L[i,j]$ stores 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 (7). Now fill in successive rows, from left to right, using (4) above.

In illustration, we extend[6] the example (2) to the strings $X =$ `lengthen` and $Y =$ `elongate`:

[6] No pun in-tended.

			1 e	2 l	3 o	4 n	5 g	6 a	7 t	8 e
		0	0	0	0	0	0	0	0	0
1	l	0	0	1	1	1	1	1	1	1
2	e	0	1	1	1	1	1	1	1	2
3	n	0	1	1	1	2	2	2	2	2
4	g	0	1	1	1	2	3	3	3	3
5	t	0	1	1	1	2	3	3	4	4
6	h	0	1	1	1	2	3	3	4	4
7	e	0	1	1	1	2	3	3	4	5
8	n	0	1	1	1	2	3	3	4	5

Table 1: Recovery of $lcs(X, Y)$

		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	$max(u,v)$

To see the formula (4) in action, we consider 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. The reader may verify that $L(X, Y) = 5$ and $LCS(X, Y) = \{\texttt{lngte}, \texttt{engte}\}$ in this example. We leave as an exercise to program this algorithm in your favorite language.

Actually, $x = 2, u = 5, v = 4$.

¶9. Complexity Analysis. Each entry is filled in constant time. Thus the overall time complexity is $\Theta(mn)$. The space is also $\Theta(mn)$.

¶10. Recovery of Optimal Instance. Given the full matrix $L[0..m, 0..n]$, we can recover an optimal instance $lcs(X, Y)$. We now describe a simple way to construct $lcs(X, Y)$ in reverse order.

We will illustrate the reconstruction process using our example of $X = \texttt{lengthen}$ and $Y = \texttt{elongate}$, whose matrix $L[0..8, 0..8]$:

We begin with the entry $L[m, n]$. It should contain the value $L(X, Y)$. In general, suppose we are at some entry $L[i, j]$ of the matrix holding the value $\ell = L(X_i, Y_j)$. If $\ell = 0$, we are done. Assume $\ell > 0$. If $x_i = y_j$, then we can output x_i and move to the entry $L[i - 1, j - 1]$ containing $\ell - 1$. If $x_i \neq y_j$, then either $L[i - 1, j]$ or $L[i, j - 1]$ contains ℓ. We move to any cell that contains ℓ. Repeat

this procedure. If we want to recover the entire set $LCS(X,Y)$, we will need to follow all the possible paths.

Following this prescription, we can start tracing from $L[8,8]$ in Table . This will trace a unique path until $L[2,3]$ at which point we branch. This results in two maximal paths, corresponding to the two strings in $LCS(X,Y)$.

¶11. Small Space Solution. The above algorithm uses $O(mn)$ space. For Google applications, this may be acceptable because m,n is typically small (how long a search string would you type?). In computational genomics, this is not acceptable because of long gene sequences. We note that to fill in any row, we just need the values from two rows. In fact the space for one row is all that we need: as new entries are filled in, it can overwrite the corresponding entry of the previous row. Since a row has n entries, we just need $O(n)$ space. As rows and columns are interchangeable, we can also work with columns, so $O(\min\{m,n\})$ space suffices.

We said it is usually easy to modify the code for computing $L(X,Y)$ to compute either $lcs(X,Y)$ or some representation of $LCS(X,Y)$. But this is not always true – for instance, you could not recover $lcs(X,Y)$ using the small space solution in ¶11.

¶12. Backward Equation. We exploit another symmetry in strings. We had been developing our equations using prefixes of X and Y. We could have equally worked with suffixes. If $X^{\#}$ denote the suffix of X obtained by omitting the first letter, then the analogue of (4) is:

$$L(X,Y) = \begin{cases} 0 & \text{if } mn = 0 \\ 1 + L(X^{\#}, Y^{\#}) & \text{if } x_1 = y_1 \\ \max\{L(X^{\#}, Y), L(X, Y^{\#})\} & \text{if } x_1 \neq y_1 \end{cases} \tag{10}$$

Let X^i denote the suffix of X length i, so $|X^i| = i$. If we use the same matrix L as before, we now need to fill in the entries in reverse order as follows:

Neat! $X_i X^{m-i} = X$

Let $L[i,j]$ denote $L(X^{m-i}, Y^{n-j})$. Thus, we could fill in the last row and last column with 0's immediately. If we work in row order, we can next fill in row $i-1$ using (10), assuming row i is already filled in. The final entry to be filled in, $L[0,0]$, contains our answer $L(X,Y)$.

So far, we have not gained anything new by looking at this backward approach. But we will next see that, when combined with the forward approach, we obtain something new.

¶13. Recovery of Optimal Instance in Small Space. Now we address the possibility of computing $lcs(X,Y)$ in small space. Note that the small space solution for $L(X,Y)$ does not easily extend to recovery of an optimal instance $lcs(X,Y)$. We now describe a solution from Hirshberg (1975) [5]. See, e.g., [2], for similar space efficient methods for geometric problems.

The solution uses an interesting divide-and-conquer technique. For simplicity, assume that n is a power of two. Observe that

$$L(X,Y) = L(X_{i^*}, Y_{n/2}) + L(X^{m-i^*}, Y^{n/2}) \tag{11}$$

for some $i^* = 0, \ldots, m$. Indeed,

$$L(X,Y) = \max_{i=0,\ldots,n} \left\{ L(X_i, Y_{n/2}) + L(X^{m-i}, Y^{n/2}) \right\}. \tag{12}$$

How can we compute the i^* such that (11) holds? We use the usual (forward) recurrence to compute

$$\left\{L(X_i, Y_{n/2}) : i = 0, \ldots, m\right\}.$$

We use the backward recurrence (10) to compute

$$\left\{L(X^{m-i}, Y^{n/2}) : i = 0, \ldots, m\right\}.$$

This takes $O(m)$ space and $O(mn)$ time. Then using (12), we can determine i^* as the value that maximizes the function $L(X_i, Y_{n/2}) + L(X^{m-i}, Y^{n/2})$.

Knowing the i^* in (11), we could divide our lcs problem recursively into two subproblems. The key observation is that (11) can be extended into an equation for the optimal instance:

$$lcs(X,Y) = \begin{cases} \epsilon & \text{if } L(X,Y) = 0, \\ Y[1] & \text{if } n = 1 \text{ and } L(X,Y) = 1, \\ lcs(X_i, Y_{n/2}); lcs(X^{m-i}, Y^{n/2}) & \text{if } n \geq 2 \text{ and } L(X,Y) = L(X_i, Y_{n/2}) + L(X^{m-i}, Y^{n/2}). \end{cases} \tag{13}$$

where ";" denotes concatenation of strings.

The space complexity of this solution is easily shown to be $O(m)$. What about the time complexity? We have

$$T(m,n) = T(i, n/2) + T(m-i, n/2) + mn.$$

It is easy to verify by induction that $T(m,n) \leq 2mn$: if $n = 1$, this is true. Otherwise,

$$\begin{aligned} T(m,n) &= T(i,n/2) + T(m-i,n/2) + mn \\ &\leq 2\left(i\frac{n}{2}\right) + 2\left((m-i)\frac{n}{2}\right) + mn = 2mn. \end{aligned}$$

¶14. Efficient Representation of $LCS(X,Y)$. We now address the problem of representing the set $LCS(X,Y)$. There are two extremes: The pair (X,Y) itself would be a representation, but it is "too implicit". An explicit list of the strings in $LCS(X,Y)$ is "too explicit" (with exponential size). A "reasonably explicit" representation should have three properties: it is polynomial in size, we can enumerate (without repetition) the strings in $LCS(X,Y)$ in linear time per element, and for any given string s, we can check if $s \in LCS(X,Y)$ in linear time.

Indeed, the matrix $L[0..m, 0..n]$ can be viewed as such a representation. It is best interpreted as a digraph $G(X,Y)$ as follows. The node set of $G(X,Y)$ is $V = \{0,1,\ldots,m\} \times \{0,1,\ldots,n\}$. For each node $(i,j) \in V$, there are between 1 to 3 edges issuing from (i,j):
(i) Matching edge: if $x_i = y_j$ and $L[i,j] = 1 + L[i-1, j-1]$, then we have an edge from (i,j) to $(i-1, j-1)$.
(ii) Non-matching edges: If $L[i,j] = L[i-1,j]$ (resp., $L[i,j] = L[i,j-1]$), we have an edge from (i,j) to $(i-1,j)$ (resp., $(i,j-1)$). Each maximal path in $G(X,Y)$ represents a string in $LCS(X,Y)$ – the string corresponds to the sequence of symbols $X[i] = Y[j]$ encountered in at node (i,j) along the path. Conversely, each string in $LCS(X,Y)$ is represented by at least one path.

But this graph is quite wasteful, and we will define a compressed version denoted $G^*(X,Y)$. For illustration, we use the pair of strings $(X,Y) = (X_3, Y_3) = (01a01a01a, 10a10a10a)$ as defined in (3). The graph $G(X_3, Y_3)$ has 100 nodes, but the compressed version $G^*(X_3, Y_3)$ shown in Figure 1 with only 15 nodes.

The idea is retain those nodes (i,j) of $G(X,Y)$ corresponding to matches $X[i] = Y[j]$. Also, the node $(0,0)$ is retained, and is called the **sink**. We introduce two kinds of edges: (1) Normal edges are

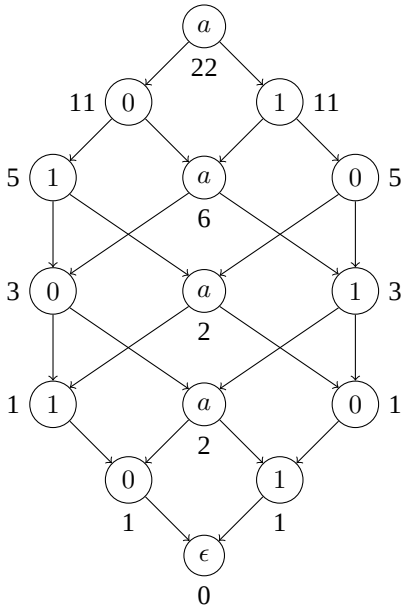


Figure 1: Representation of $LCS(X_3, Y_3)$

$(i,j)-(i',j')$ where $L[i,j] = L[i',j'] + 1$ and there is a path from (i,j) to (i',j') in $G(X,Y)$. Note that in such a path, every node (i'',j'') except the last will satisfy $L[i'',j''] = L[i,j]$. (2) Sink edges from (i,j) to $(0,0)$ whenever $L[i,j] = 1$. As we see in Figure 1, the result is a level graph in the sense that each node (i,j) has a level $\ell \geq 0$, corresponding to the length of the longest path from (i,j) to the sink, and edges can only go from a level ℓ to level $\ell - 1$.

In Figure 1, the name (i,j) of a node is not explicitly given, but we have labelled the node with the letter $X[i] = Y[j]$ corresponding to the match. By reading this sequence of labels along a maximal path, you get a string on $LCS(X_3, Y_3)$.

External to each node in Figure 1, a numerical value is indicated: what are these?

¶15. Other Improvements. We can exploit knowledge about the alphabet. For instance, Paterson and Masek gave an algorithm with $\Theta(mn/\log(\min(m,n)))$ time when the alphabet of the strings is bounded.

Our algorithm fills 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.

¶16. Applications. Computational problems on strings has been studied since the early days of computer science. One motivation is their application in text editors. For instance, the problem of finding a pattern in a larger string is a basic task in text editors. Another interesting application is in computer virus detection. The growth of the world wide web has been accompanied by the proliferation of computer viruses. It turns out that each virus will send messages X, Y which are rather similar to each other.

We use $L(X, Y)$ as a measure of similarity. If it is known that Y is from a virus, and $L(X, Y)$ exceeds some threshold, we infer that X is probably from the same virus. See Exercise below.

The advent of computational genomics in the 1990's has brought new attention to problems on strings. 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 correspond to the four bases: adenine, cytosine, guanine 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 have been formulated to measure similarity.

EXERCISES

Exercise 2.1: Extending our running example, compute $L(X, Y)$ where $X =$ `lengthening` and $Y =$ `elongation`. ♢

Exercise 2.2: Compute $lcs(X, Y)$ for $X =$ `AATTCCCCGACTGCAATTCACGCACC` and $Y =$ `GGCTTTTATTCTCCCTGTAAGT`. These are parts of DNA sequences from a modern human and a Neanderthal, respectively. ♢

Exercise 2.3: Show (6). ♢

Exercise 2.4: Give a direct recursive algorithm for computing $L(X, Y)$ based on equation (4) and show that it takes exponential time. (In other words, equation (4) alone does not ensure efficiency of solution.) ♢

Exercise 2.5: Let $lcs(X, Y)$ denote any member of $LCS(X, Y)$. Give the analogue of (8) for $lcs(X, Y)$. ♢

Exercise 2.6: (V.Sharma and Yap) Consider the example in (3).
(a) Compute $L(X_2, Y_2)$ by filling in the the usual matrix. Moreover, determine $\lambda(X_2, Y_2) = |LCS(X_2, Y_2)|$ by counting the number of maximum paths in the matrix.
(b) Prove that $L(X_n, Y_n) = 2n$.
(c) We indicated that $\lambda(X_n, Y_n) = |LCS(X_n, Y_n)| \geq 2^n$. Prove that $\lambda(X_n, Y_n) = \Omega(\sqrt{6}^n)$.
(d) Construct the graph $G_1(X_3, Y_3)$ as described in the text. Use this graph to count $\lambda(X_3, Y_3)$. What does this imply about $\lambda(X_n, Y_n)$?
(e) Write H_n for the graph $G_1(X_n, Y_n)$. Give a simple description of H_1 up to isomorphism.
(f) Based on this description, provide an exact closed formula for $\lambda(X_n, Y_n)$. ASIDE: Can you also show that this number is equal to $\sum_{i=0}^{n} \binom{n}{i}^2$? ♢

Exercise 2.7: Let $S = \{X_1, \ldots, X_k\}$ be a set of strings where k is not fixed. Wlog assume that no X_i is a substring of another X_j ($i \neq j$). A string Z such that each X_i is a substring (not subsequence) of Z is called a **superstring** of S. Let $SCS(S)$ denote the **shortest common superstring** of S. We are interested in computing $SCS(S)$. In some sense, this is the dual of the LCS problem. It is quite important in DNA sequencing where a long DNA sequence might be chemically cut into short substrings, and we want to reconstruct the original sequence as a shortest superstring.
(a) Is there a dynamic programming principle for this general problem?

(b) Give an efficient algorithm for $k = 2$.
(c) Let $merge(X, Y)$ denote the shortest string of the form $Z = UVW$ where $X = UV$ and $Y = VW$ where U and W are non-empty. Let U the **overlap** of X, Y denoted $ov(X, Y)$. We are interested in choosing X, Y where the overlap length $|ov(X, Y)|$ is maximum. Consider a simple greedy algorithm in which, at each iteration, we pick two strings $X, Y \in S$ with the maximum overlap length $|ov(X, Y)|$, and replace X, Y by $merge(X, Y)$. When there is only one string left, we output this as an approximation to $SCS(S)$. Let $G(S)$ denote the output of the greedy algorithm. Show that $|G(S)| \leq 4|SCS(S)|$. ♢

Exercise 2.8: Joe Quick observed that the recurrence (4) 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 (4) and describe the corresponding algorithm. Perform an analysis of your new algorithm, to confirm and or reject the Quick Hypothesis. ♢

Exercise 2.9: 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 (4)?
(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 2.10: What are the forbidden configurations in the matrix M used for computing $L(X, Y)$?
(a) Suppose $M[i, j] = 89$, what are the possible values of $M[i-1, j-1]$?
(b) 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 2.11:
(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 2.12: Let X, Y be strings.
(a) Prove that $L(XX, Y) \leq 2L(X, Y)$.
(b) Show that for every n, there are X, Y with $L(X, Y) = n$ and inequality in (b) is an equality.
(c) Prove that $L(XX, YY) \leq 3L(X, Y)$.
(d) Similar to part (b) but for the inequality of (c). ♢

Exercise 2.13: 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 2.14: Let $LCS'(X,Y)$ be the *multiset* of all the longest common subsequences of X and Y. That is, for each longest common subsequence $Z \in LCS(X,Y)$, we say Z has multiplicity $k\ell$ where Z occurs k (resp., ℓ) times as a subsequence of X (resp., Y). Let $\lambda'(n,m)$ and $\lambda'(n)$ be defined as in the previous exercise. Re-do the previous exercise for $\lambda'(n)$. ♢

Exercise 2.15: Modify the algorithm for $L(X,Y)$ to compute the following functions:
(a) $\lambda'(X,Y)$
(b) $\lambda(X,Y)$ ♢

Exercise 2.16: 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 2.17: (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 2.18: 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 2.19: Researchers are using LCS computation to fight computer viruses. A virus that is attacking a machine has a predictable pattern of messages it sends to the machine. We view the concatenation of all these messages that a potential virus sends as a single string. Call the first 1000 bytes than from any source (i.e., potential virus) the **signature** of that source. Let X be the signature of an unknown source and Y is the signature of a known virus. It is known empirically that if $L(X,Y) > 500$, then X is from the same virus, and if $L(X,Y) < 200$, it is different.
(a) Design a practical and efficient algorithm for the decision problem $L(X,Y,k)$ which outputs "PROBABLY VIRUS" if $L(X,Y) > k$ and "PROBABLY NOT VIRUS" otherwise. Give the pseudo-code for an efficient practical algorithm. NOTE: The obvious algorithm is to use the standard algorithm to compute $L(X,Y)$ and then compare n to k. But we want you to do better than this. HINT: There are two ideas we want you to exploit – most students only think of one idea.
(b) Quantify the complexity of your algorithm, and compare its performance to the obvious algorithm (which first computes $L(X,Y)$). First do your analysis using the general complexity parameters of $m = |X|, n = |Y|$ and k, and also $\ell = L(X,Y)$. Also discuss this for the special case of $m = n = 1000$ and $k = 500$. ♢

Exercise 2.20: 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 2.21: (Hirshberg and Larmore, 1987.) A concept of "Set LCS" quite distinct from our definition goes as follows. We want to compute the "LCS" of $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$ where $x_i \in \Sigma$ (for some alphabet Σ as before) but $y_j \in 2^\Sigma$. We view Y as a set of strings over Σ, $Y = \{\overline{y}_1 \cdots \overline{y}_n\}$ where each $\overline{y}_i$ is a permutation of the set $y_i \subseteq \Sigma$. An element $\overline{y}_1 \cdots \overline{y}_n \in Y$ is called a **flattening** of Y. A **SLCS** of X and Y is defined to be a common of X and any flattening of Y of maximum length. Give an $O(mN)$ algorithm for SLCS where $N = \sum_{j=1}^{n} |y_j|$. N.B. The motivation comes from computer-driven music where a "polyphonic score" is defined to be a sequence of sets of notes (represented by Y). Each $y_j \subseteq \Sigma$ may be viewed as a chord. X is a solo score that is to be played to accompany the polyphonic score. ♢

Exercise 2.22: 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

§3. Edit Distance

In the previous section, we tried to capture similarity between two strings by the amount of overlap. In this section, we look at a different view of similarity – *what is the minimal amount of change necessary to make the two strings equal?* For instance, two identical strings are most similar because the amount of change necessary to make them equal is zero. These represent opposite views of similarity: in one case we try to maximize the overlap, and in the other case we try to minimize the amount of change.

what is our favorite editor? the "vi" derivative called "gvim"

But first we need a way to measure change. It is based on the idea of editing a string in a text editor on a modern computer. Text editors has a repertoire of basic operations, and we we just count the number of basic operations necessary to convert one string X to another Y. The minimum such number $D(X, Y)$ is called the **edit distance** between X and Y. A "complete" repertoire for any editor may comprise just two types of operations: the insertion and deletion of a single character into a string. This allows us to transform any X into any other Y. We also need to associate a positive cost with each operation: the simplest model is to charge one unit per operation. Under this unit cost model, if $X =$ `cat` and $Y =$ `dog`, and we only have insertion and deletion operations then it is easy to see that $D(X, Y) = 6$ (we need to delete 3 letters and to insert 3 letters). On the other hand, if we have the operation to replace any letter in a string by another letter, then $D(X, Y) = 3$ as three replacement operations suffice. One could also give $D(X, Y)$ a computational biology interpretation, by postulating editing operations that are apropos of genetic modification.

Here is one application of the $D(X,Y)$ measure: in looking up strings in a database, a query string X can be used to find the Y that is closest to X, i.e., which minimizes the edit distance $D(X,Y)$. The ultimate in similarity between X and Y is then captured by the relation $D(X,Y) = 0$. It is interesting to compare $L(X,Y)$ in the LCS problem with the edit distance $D(X,Y)$: In the LCS problem, X and Y are more similar for larger values of $L(X,Y)$. But in edit distance, X and Y are more similar for smaller values of $D(X,Y)$. We explore some connection between $D(X,Y)$ and $L(X,Y)$ below.

¶17. The Standard Edit Distance. We now specify the standard repertoire of edit operations. Fix any alphabet Σ. For any index $i \geq 1$ and letter $a \in \Sigma$, define the following three **standard edit operations**:

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

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

$$Ins(i,a,X), \quad Del(i,X), \quad Rep(i,a,X) \tag{14}$$

denote the strings that are produced by these respective operations. For example, if $X =$ *AATCGA* then $Ins(3,G,X) =$ *AAGTCGA*, $Del(5,X) =$ *AATCA* and $Rep(5,T,X) =$ *AATCTA*.

The operations in (14) assume that i is in the "proper range": For insertion, this means $1 \leq i \leq |X|+1$, but for deletion and replacement, this means $1 \leq i \leq |X|$. When i is not in the proper range, we simply[7] declare such operations to be undefined. The operations $Del(i)$ and $Ins(i,a)$ are inverses of each other in the following sense:

$$X = Del(i, Ins(i,a,X)), \qquad X = Ins(i,b,Del(i,X)), \tag{15}$$

for some $b \in \Sigma$. 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.

Definition of edit distance $D(X,Y)$

We define $D(X,Y)$ be the minimum number of standard edit operations that will transform X to Y. For example, $D(\mathsf{TAG},\mathsf{CAT}) \leq 2$ since

$$\mathsf{TAG} = Rep(3,\mathsf{G},Rep(1,\mathsf{T},\mathsf{CAT})).$$

Moreover, $D(\mathsf{TAG},\mathsf{CAT}) \geq 2$ since a single edit operation cannot make these two strings equal. Therefore we conclude that $D(\mathsf{TAG},\mathsf{CAT}) = 2$.

Our immediate goal is devise an efficient algorithm to compute $D(X,Y)$ for any X, Y. But first, let us explore some simple properties. The first remark is that the set Σ^* of strings, together with the edit distance function $D : \Sigma^* \times \Sigma^* \to \mathbb{R}_{\geq 0}$, constitutes a **metric space**. This amounts to satisfying the following natural properties:

(i) (Non-negativity) $D(X,Y) \geq 0$ with equality iff $X = Y$.

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

(iii) (Triangular Inequality)

$$D(X,Z) \leq D(X,Y) + D(Y,Z). \tag{16}$$

[7] It is also easy to introduce conventions for interpreting (14) so that these operations are well-defined for all i.

We also have the following bounds:

$$|X| - |Y| \leq D(X,Y) \leq |X| \tag{17}$$

where $|X| \geq |Y|$ (this assumption is without loss of generality because of (ii)). In proof, the lower bound on $D(X,Y)$ is necessary because we need at least $|X|-|Y|$ delete operation just to decrease the length of X to that of Y. The upper bound is sufficient because, by using $|Y|$ replacement operations, we can make modify X so that it has Y as a prefix, and this can be followed by $|X| - |Y|$ deletions. These bounds are achievable. E.g., the upper bound is attained with $D(\texttt{google}, \texttt{search}) = 6$.

¶18. An Infinite Edit Distance Graph. 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 (14) that transforms X to Y. Paths in $G(\Sigma)$ are called **edit paths** and edit distances has the following interpretation:

$$D(X,Y) \text{is the length of the shortest (link-distance) path from } X \text{ to } Y \text{ in } G(\Sigma). \tag{18}$$

In analogy to (4), we have the following recursive formula:

$$D(X,Y) = \begin{cases} \max\{|X|,|Y|\} & \text{if } mn = 0 \\ D(X',Y') & \text{if } x_m = y_n \\ 1 + \min\{D(X',Y), D(X,Y'), D(X',Y')\} & \text{if } x_m \neq y_n \end{cases} \tag{19}$$

It is a simple exercise to prove the correctness of this formula. It follows that $D(X,Y)$ can also be computed in $O(mn)$ time by the technique of ¶8, by filling in entries in a $m \times n$ matrix M.

Suppose we want to compute, not just the number $D(X,Y)$, but the sequence of $D(X,Y)$ edit operations to convert X to Y. We have seen this idea before — we expect to be able to annotate the matrix M with some additional information to help us do this. For this purpose, let us decode equation (19) 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.

¶19. Connection to LCS Problem. We had alluded to a connection between $L(X,Y)$ and $D(X,Y)$. 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. Upper bound: if $Z \in LCS(X,Y)$ then we have $D(X,Z) \leq m - L(X,Y)$ and $D(Z,Y) \leq n - L(X,Y)$, Hence $D(X,Y) \leq D(X,Z) + D(Z,Y) \leq m + n - 2L(X,Y)$.

Lower bound: assume $m \geq n$, so it suffices to show $L(X,Y) \geq m - D(X,Y)$. Suppose we transform X to Y in a sequence of $D(X,Y)$ edit steps. Clearly, $D(X,Y) \leq m$. 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, i.e., $L(X,Y) \geq |Z| = m - D(X,Y)$. **Q.E.D.**

These bounds are essentially the best possible: assume $m \geq n$. Then for each $n/2 \leq \ell \leq n$, there are strings X, Y such that $D(X,Y) = m + n - 2\ell$ where $L(X,Y) = \ell$. E.g., $X = a^{m-\ell}b^{\ell}$ and $Y = b^{\ell}c^{n-\ell}$. For the lower bound, for each $0 \leq \ell \leq m$, there are strings X, Y such that $D(X,Y) = m - \ell$. E.g., $X = a^{m-\ell}b^{\ell}$ and $Y = b^{\ell}$.

¶20. Edit distance under general cost function. Our edit distance was based on unit cost for every operation. We now generalize this by allowing different costs for different types of operations. The "type" of an operation is determined, not only by its nature (insert/delete/replace) but also by the letters that are operated upon.

An **alignment cost function** is given by

$$\Delta : (\Sigma \cup \{*\})^2 \to \mathbb{R}$$

where $*$ is a symbol not in the alphabet Σ. For $x, y \in \Sigma$, we interpret $\Delta(x,y)$ as the cost to replace x by y. Also, for $b \in \Sigma$, we interpret $\Delta(*,b)$ as the cost of inserting b, and $\Delta(b,*)$ as the cost of deleting b. Under this interpretation, it is natural to impose the following requirement on Δ:

$$\Delta(*,*) \quad \geq \quad 0. \tag{20}$$

Informal definition of alignment distance $A(X,Y)$

The **alignment distance** between strings X, Y under this cost function is denoted $A_\Delta(X,Y)$, or simply $A(X,Y)$, if Δ is understood. If Δ is non-negative, then the definition of $A_\Delta(X,Y)$ is easy to define, and corresponds to our intuition coming from edit distance $D(X,Y)$. So for the time being, we assume $\Delta \geq 0$.

The terminology "alignment" is new and needs some motivation. The concept comes from genomics where we think of computing $A_\Delta(X,Y)$ as an issue of "aligning" X with Y so that there is a one-one correspondence between letters of X and Y, and all we do is to replace corresponding letters that are mismatched (i.e., different). Of course, letter-for-letter replacements alone will not be enough, so we need to generalize this notion to include insertions and deletions (i.e., by aligning letters with $*$). For instance, if $X = \texttt{ACT}$ and $Y = \texttt{CAT}$ then a possible alignment of these two strings can be represented by the pair $(X_*, Y_*) = (\texttt{AC*T}, \texttt{*CAT})$, which can be visualized as follows:

X_* :	A	C	*	T
Y_* :	*	C	A	T

The cost of this alignment is then taken to be

$$\Delta(\texttt{A},*) + \Delta(\texttt{C},\texttt{C}) + \Delta(*,\texttt{A})\Delta(\texttt{T},\texttt{T}).$$

We will shortly give a formal model of this alignment.

It is an easy observation that our original edit distance $D(X,Y)$ amounts to the alignment cost function where $\Delta(x,y) = 1$ if $x \neq y$ and $\Delta(x,y) = 0$ otherwise. But in general, the ability of Δ to assign cost based on the letters is rather useful:

- In genomics, it appears that that replacing A by C is less likely than replacing A by T. This can be modeled using a cost function where $\Delta(\mathtt{A},\mathtt{C}) > \Delta(\mathtt{A},\mathtt{T})$.
- Consider the string edit problem over the alphabet $\{\mathtt{a},\mathtt{b},\mathtt{c},\ \ldots,\mathtt{x},\mathtt{y},\mathtt{z}\}$: in many keyboard layouts, the key for b is adjacent to that for v, but relatively far from key a. Since it is easy to confuse two adjacent keys on a keyboard, we may model typing errors with a cost function where $\Delta(\mathtt{a},\mathtt{b}) > \Delta(\mathtt{v},\mathtt{b})$.

For example, consider the alignment cost function

$$\Delta(x,y) = \begin{cases} 2 & \text{if } x = * \text{ or } y = * \\ 0 & \text{if } x = y \\ 1 & \text{else.} \end{cases} \tag{21}$$

Thus we charge two units for insertion or deletion, but one unit for replacement. There is no charge when $x = y$ since, intuitively, this is a null operation. Suppose $X = \mathtt{bulk}$ and $Y = \mathtt{ucky}$. In the uniform cost model, we have $D(X,Y) = 3$, obtained by the following sequence of delete, insert, replace operations:

$$\mathtt{bulk} \overset{\mathtt{Del}}{\to} \mathtt{ulk} \overset{\mathtt{Ins}}{\to} \mathtt{ulky} \overset{\mathtt{Rep}}{\to} \mathtt{ucky}.$$

With the cost model of (21), these operations have a total cost of $5 = 2+2+1$. This cost is suboptimal because we can achieve a cost of $4 = 1+1+1+1$ by a straightforward sequence of replacements:

$$\mathtt{bulk} \to \mathtt{uulk} \to \mathtt{uclk} \to \mathtt{uckk} \to \mathtt{ucky}.$$

It is also easy to see that the cost cannot be less than 4. So we conclude that $A(\mathtt{bulk},\mathtt{ucky}) = 4$.

So far, we have only looked at non-negative costs. But something quite interesting arises if we allow negative costs. To focus on this issue, let us introduce a simple class of cost functions. In general, the cost function Δ requires specifying almost $(|\Sigma|+1)^2$ numbers. But consider the following cost function,

$$\Delta(x,y) = \begin{cases} \delta_= & \text{if } x = y, \\ \delta_{\neq} & \text{if } x \neq y \\ \delta_* & \text{if } x = * \text{ or } y = *. \end{cases} \tag{22}$$

which is completely specified by three parameters $\delta_=, \delta_{\neq}$ and δ_*. The value δ_* is called the **gap penalty**. We shall assume that

$$\delta_= \leq 0 < \delta_{\neq} < \delta_* \tag{23}$$

This is well-motivated in genomics where an insertion or deletion in a DNA sequence is a significant change and relatively rare. Here is one set of such parameters:

$$\delta_* = 3, \quad \delta_= = -2, \quad \delta_{\neq} = 1. \tag{24}$$

Let us note that the triangular inequality (cf. (16)) fails under the cost function (24). Let $X = \mathtt{a}$, $Y = \mathtt{aa}$, and $Z = \mathtt{aaa}$. Under the alignment cost specified by (24), we have $A(\mathtt{a},\mathtt{aa}) = 3-2 = 1$, $A(\mathtt{aa},\mathtt{aaa}) = 3-4 = -1$, and $A(\mathtt{a},\mathtt{aaa}) = 6-2 = 4$. Thus

$$A(\mathtt{a},\mathtt{aa}) + A(\mathtt{aa},\mathtt{aaa}) < A(\mathtt{a},\mathtt{aaa}).$$

Another example where triangular inequality fails is $X = \mathtt{ab}$, $Y = \mathtt{bb}$ and $Z = \mathtt{ba}$.

¶21. What is missing in the editing model? We have motivated the need for a cost model based on the letters operated upon. But why do we need negative costs? In our simplified cost function (22), we might think that a non-zero value for $\delta_=$ is most curious. Shouldn't $\delta_=$ always be 0? In other words, if there is no need to replace x, then no cost should be associated. First of all, it seems clear that there is little point in making $\delta_=$ positive. On the other hand, there is a strong case for allowing negative $\delta_=$. In terms of minimizing cost, a negative value is actually a good thing.

Imagine that the FBI has a DNA bank containing the DNA sequences collected at crime scenes. To correlate these crimes, the FBI computes the alignment costs of pairs of DNA's coming from different crime scenes. Let us define the correlation between two crime scenes to be the minimum $A(X, Y)$ where X occurs at one scene and Y at the second scene. With this definition, we would like our alignment cost to exhibit the following kind of inequality:

$$A(\mathtt{C},\mathtt{C}) > A(\mathtt{CC},\mathtt{CC}) > A(\mathtt{CCC},\mathtt{CCC}).$$

In other words, a matching pair in the alignment should be a positive factor, not neutral, for crime correlation. This amounts to $\delta_= < 0$, not $\delta_= = 0$. Similarly, we want the inequality

$$A(\mathtt{CG},\mathtt{CGG}) > A(\mathtt{CGAAA},\mathtt{CGGAAA})$$

even though a single insertion suffice to align both pairs of strings.

¶22. Algorithm to compute alignment cost. We now give a dynamic programming method to compute $A_\Delta(X, Y)$. The method is reminiscent of the LCS problem. Suppose $X = x_1 \ldots x_{m-1} x_m = X' x_m$ and $Y = y_1 \ldots y_{n-1} y_n = Y' y_n$. Then we have the recursive rule:

$$A(X,Y) = \begin{cases} \delta_*(m+n) & \text{if } mn = 0, \\ \min\{A(X',Y') + \Delta(x_m, y_n), & \\ \quad A(X',Y) + \Delta(x_m, *), & \\ \quad A(X,Y') + \Delta(*, y_n)\} & \text{else.} \end{cases} \tag{25}$$

Note that for simplicity, (25) assumes that all deletion and insertion have the same cost of δ_*. To systematically carry out the computation, we set up a $(m+1) \times (n+1)$ matrix M. The first row and first column corresponds the the base case, and can be filled in first using the base case of (25). The remaining entries of M is filled in a row by row fashion, using the general case of (25). The desired value $A(X, Y)$ is found in the $(m+1, n+1)$-entry of M.

Example. Assume that Δ in (22) is given by

$$\delta_= = -1, \quad \delta_{\neq} = 1, \quad \delta_* = 2. \tag{26}$$

For $X = \mathtt{GCAT}$ and $Y = \mathtt{AATTC}$, our matrix computation yields:

$M =$

	ε	A	A	T	T	C
ε	0	2	4	6	8	10
G	2	1	3	5	7	9
C	4	3	2	4	6	6
A	6	3	2	3	5	7
T	8	5	4	1	2	4

This proves that $A(X, Y) = 4$.

The original alignment problem came from S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins", *J.Molecular Biology*, 48(3):443-53, 1970. It is the first application of dynamic programming to computational biology. The cost function Δ is represented by a so-called **similarity matrix**. A typical similarity matrix is

Δ	A	G	C	T
A	-3	2	3	1
G	2	-3	2	1
C	3	2	-3	1
T	1	1	1	-3

. (27)

where the negative scores along the diagonal corresponds to $\delta_= = -3$. The gap penalty δ_* separately given.

¶23. Model of Alignment. The above algorithm for computing $A(X, Y)$ using (25) follows the LCS model and standard edit distance model. But the justification of its correctness in the presence of negative costs requires a new model of what we are minimizing.

Recall that we previously interpreted $D(X, Y)$ as a minimum cost path problem in the infinite graph $G(\Sigma)$ defined in ¶18. We could extend this interpretation to $A_\Delta(X, Y)$, where we now attach appropriate costs for edges of $G(\Sigma)$. This model works well as long as we do not have negative costs. But suppose $\Delta(\mathsf{a}, \mathsf{a})$ is negative. If a occurs in any string X, then the graph $G(\Sigma)$ will have an edge from X to itself (i.e., a self-loop) with cost $\Delta(\mathsf{a}, \mathsf{a})$. Then we must conclude that $A(X, X) = -\infty$ since we can replace a by itself as many times as we wish and induce an arbitrarily negative cost. It is easily seen that this implies $A(X, Y) = -\infty$ for all $X, Y \in \Sigma^*$. Something is clearly wrong with this interpretation. This problem will arise as long as there is a cycle in $G(\Sigma)$ with negative cost. So in order to define $A(X, Y)$ properly, we must restrict the possible paths from X to Y (in particular, we must not re-use edges). We introduce an "alignment model" to capture this.

The issue of negative cycles in shortest path is a well-known phenomenon

To compute the alignment distance for X, Y, we first inserting zero or more $*$'s into X and Y so that the resulting strings X_*, Y_* have the same length. Such a pair (X_*, Y_*) is called an **alignment** of X, Y. Thus the ith character $X_*[i]$ in X_* is aligned with the ith character $Y_*[i]$ in Y_* if we place X_* above Y_*. The cost of this alignment is the sum of the cost of "replacing" each $X_*[i]$ by $Y_*[i]$. We may extend the original cost function Δ to alignments as follows:

$$\Delta(X_*, Y_*) := \sum_{i=1}^{\ell} \Delta(X_*[i], Y_*[i])$$

where $\ell = |X_*|$. Of course, this "replacement" covers insertion and deletions as well. Finally, define the **alignment cost** for X, Y to be the minimum of $\Delta(X_*, Y_*)$ over all alignments (X_*, Y_*), and denote this minimum by $\Delta_*(X, Y)$:

$$\Delta_*(X, Y) := \min_{(X_*, Y_*)} \Delta(X_*, Y_*) \tag{28}$$

where (X_*, Y_*) ranges over all alignments of X, Y. Call (X_*, Y_*) an **optimal alignment** if $\Delta(X_*, Y_*) = \Delta_*(X, Y)$. Under assumption (20), an optimal alignment must satisfy $X_*[i] \neq *$ or $Y_*[i] \neq *$ for each i. Thus, we $|X_*| = |Y_*| \leq |X| + |Y|$.

E.g., Let $X = \mathsf{AATTC}$ and $Y = \mathsf{GCAT}$, as in a previous example. If $X_* = \mathsf{AATTC}$ and $Y_* = \mathsf{GCAT*}$, then $\Delta(X_*, Y_*) = 1 + 1 + 1 - 1 + 2 = 4$. If the alignment cost of X, Y is equal to $A(X, Y)$

as we have been trying to suggest, then this particular alignment (X_*, Y_*) must be optimal. That is because we have previously computed $A(X,Y) = 4$. This is the result to be shown next.

¶24. Correctness of the Dynamic Programming Solution. We formally defined the alignment cost to be $\Delta_*(X,Y)$ in (28). In ¶22, we described a dynamic programming algorithm to compute a quantity that we will define (by fiat!) to be $A_\Delta(X,Y)$. The correctness of the dynamic programming algorithm amounts to the equality $A_\Delta(X,Y) = \Delta_*(X,Y)$ for all strings X, Y. Unfortunately, this is not true without further restrictions on Δ. We say Δ is **well-founded** if

$$\Delta(a,*) + \Delta(*,a) \geq 0$$

for all $a \in \Sigma$. To see why this is necessary, suppose $\Delta(a,*) + \Delta(*,a) < 0$ for some a. Then

$$\Delta(\epsilon,\epsilon) = -\infty$$

where ϵ is the empty string. To see this, note that $(X_n, Y_n) := (\mathtt{a}^n *^n, \ *^n \mathtt{a}^n)$ is an alignment for (ϵ,ϵ) for any $n \geq 0$. The cost of this alignment is $n(\Delta(a,*) + \Delta(*,a))$, which can be arbitrarily negative. This argument can be extended to showing that $\Delta_*(X,Y) = -\infty$ for all X, Y.

THEOREM 2 (Correctness).
Δ is well-founded iff for all $X, Y \in \Sigma^*$, $\Delta_*(X,Y)$ is finite. When $\Delta_*(X,Y)$ is finite, it is equal to $A_\Delta(X,Y)$.

Proof. To show that $\Delta_*(X,Y)$ is finite, we show that if (X_*, Y_*) is an alignment of (X,Y), and if $|X_*| > |X| + |Y|$ then the cost of (X_*, Y_*) is not minimal. ... **Q.E.D.**

An alternative to the preceding alignment model is the following "marking model". Initially, we "mark" each letter in X. Each replacement or deletion operation is applicable only to marked letters. The result of a replacement or insertion operation is an unmarked letter. At the end of our sequence of operations, we must obtain a copy of Y with only unmarked letters. We leave it as an Exercise to show that this is equivalent to our alignment model.

Above we have noted that with negative costs, we may not satisfy the triangular inequality. We now prove a positive result in the other direction:

LEMMA 3. *Suppose the alignment cost function Δ is "triangular" in the sense that for all $x, y, z \in \Sigma \cup \{*\}$, we have $\Delta(x,z) \leq \Delta(x,y) + \Delta(y,z)$. Then the alignment distance A_Δ satisfies the triangular inequality: $\Delta(X,Z) \leq \Delta(X,Y) + \Delta(Y,Z)$.*

Proof. Suppose (X_*, Y_*) and (Y_{**}, Z_{**}) are the optimal alignments for $A_\Delta(X,Y)$ and $A_\Delta(Y,Z)$, respectively. Then we claim that $\Delta(X,Y) + \Delta(Y,Z) \geq \Delta(X,Z)$. This can be shown by constructing an alignment $(\widehat{X}, \widetilde{Z})$ that has alignment cost at most $A_\Delta(X,Y) + A_\Delta(Y,Z)$. ... incomplete **Q.E.D.**

¶25. Example. Let us give a non-biological example, motivated by string editing. Let $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c}, \ \ldots, \mathtt{x},\mathtt{y},\mathtt{z}\}$ be the letters of the English alphabet. Define

$$\Delta(x,y) = \begin{cases} \delta_* & \text{if } x = * \text{ or } y = *, \\ \delta_= & \text{if } x = y, \\ \delta_1 & \text{if } x, y \text{ are both consonants or both vowels,} \\ \delta_2 & \text{else.} \end{cases} \tag{29}$$

This cost function generalizes the editing distance cost in which we take into account the nature of letters that cause mismatch. For instance, with the choice

$$\delta_* = 3, \quad \delta_= = 0, \quad \delta_1 = 1, \quad \delta_2 = 2, \tag{30}$$

then $A(\texttt{there},\texttt{their}) = 4$ since we can replace the last two letters in the first word by their corresponding letter in the second word. This has cost 4 since using $\Delta(\texttt{r},\texttt{i}) = \Delta(\texttt{e},\texttt{r}) = 2$. There is no cheaper way to effect this transformation.

¶26. Generalizations. There are many possible generalizations of the above string problems.

- We can introduce cost models that are "context sensitive". For instance, transforming `XabY` to `XbaY` can be viewed as two replacements (with total cost of $2\Delta(a,b)$). But if we look at the context of these two replacements, and realize that they can be viewed as a transposition, then we might want to assign a smaller cost.
- 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 minimum 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.
- Let $D = \{Y_1, \ldots, Y_n\}$ be a fixed set of strings, called the dictionary. Define $A(X,D) = \min\{A(X,Y_i) : i = 1, \ldots, n\}$. We would like to preprocess D so that for any given X, we can quickly compute the set of words in the dictionary that is closest to X according to the alignment distance.

Remarks: Levenshtein (1966) introduce the editing metric for strings in the context of binary codes. Needleman and Wunsch (1970), "A general method applicable to the search for similarities in the amino

acid sequence of two proteins" (J.Mol.Biol., 48(3)443-53), is considered to be the first application of dynamic programming to biological sequence comparisons. Smith and Waterman (1981) proposed a variation of the Needleman-Wunsch algorithm to find all *local* alignments between two sequences. In contrast, the Needleman-Wunsch algorithm addresses the *global* alignment problem. 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. The volume by Apostolico and Galil [1] contains a state-of-the-art overview for pattern matching algorithms, circa 1997.

EXERCISES

Exercise 3.1: Compute the edit distances $D(X, Y)$ where X, Y are given:
(a) $X = 00110011$ and $Y = 10100101$.
(b) $X =$ `AGACGTTCGTTAGCA` and $Y =$ `CGACTGCTGTATGGA`.
(c) $X =$ `CGTAATCC` and $Y =$ `CCGTCC`. Recall that Google thought these two strings are similar, and may refer to `CCGTCC.com`. ♢

Exercise 3.2: Compute the alignment distance $A_\Delta(X, Y)$ for the examples (a)-(c) in the previous question. Let Δ be specified by the parameters $\delta_= = -1$, $\delta_\neq = 1, \delta_* = 2$. ♢

Exercise 3.3: Compute the alignment distance $A(X, Y)$ between $X =$ `google` and $Y =$ `yahoo` using the alignment cost (29) and and (30). For this purpose, assume `y` is a consonant. Also, express $\Delta(X, Y)$ as a direct alignment cost. ♢

Exercise 3.4: Suppose we compute optimal alignment $A(X, Y)$ by filling a matrix $M[0..m, 0..n]$ where $|X| = m, |Y| = n$. Let $M[i, j]$ be the optimal cost to align X_i with Y_j where X_i is the prefix of X of length i and similarly for Y_j. Assume the alignment cost function of the previous google-yahoo question. Suppose $M[i, j] = k$. What are the possible values for $M[i-1, j-1]$ as a function of k? What about $M[i-1, j+1]$ as a function of k? Justify your answer. ♢

Exercise 3.5: Compute $A(X, Y)$ where X, Y are the strings `AATTCCCGA` and `GCATATT`. Assume Δ has gap penalty 2, $\Delta(x, x) = -2$ and $\Delta(x, y) = 1$ if $x \neq y$. You must organize this computation systematically as in the LCS problem. ♢

Exercise 3.6: Prove (19). This is an instructive exercise. ♢

Exercise 3.7: Let x, y, z be distinct letters, and $0 \leq m \leq n$.
(a) Prove that $D(X, Y) = m + n - 2\ell$ where $m \geq \ell \geq m/2$, $X = x^{m-\ell}z^\ell$ and $Y = z^\ell y^{n-\ell}$.
(b) Let $X = x^{m-\ell}z^\ell$ and $Y = y^{n-\ell}z^\ell$ $(0 \leq \ell \leq n)$ Prove that $D(X, Y) = n - \ell$. ♢

Exercise 3.8: 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 3.9: You work for Typing-R-Us, a company that produces smart word processing editors. When the user mistypes a word, you want to lookup the dictionary for the set of closest matching words.
(a) Design an alignment cost function Δ which takes into account the keyboard layout. Assuming the QWERTY layout, you would like to define $\Delta(x, y)$ to be small when x, y are close to each other in this layout. Also, row distance is much smaller than column distance. Assume $\Sigma =$ {A,B,C ,..., X,Y,Z}.
(b) Using your Δ function, compute A(QWERTY, QUIET) and A(QWERTY, QUICKLY). ♢

There are 3 rows of letters in this layout: The first row is QWERTYUIOP. The next two rows are ASDFGHJKL and ZXCVBNM.

Exercise 3.10: In the text, we described a "marking model" to formalize the allowable sequence of operations to transform X to Y (and $A(X, Y)$ is the minimum cost of such an allowable sequence). Prove that this model is equivalent to our alignment model. ♢

Exercise 3.11: Let $D = \{Y_1, \ldots, Y_n\}$ be a fixed set of strings, called the dictionary. Let $A(X, D) = \min\{A(X, Y_i) : i = 1, \ldots, n\}$ be the minimum alignment distance between a string X and any string Y in D. How can you preprocess D so that $A(X, D)$ can be computed in faster than the obvious method? ♢

Exercise 3.12: Let Σ^{**} denote strings of strings. A natural language text can be thought of as an element of Σ^{**}. If $v, w \in \Sigma^*$, let $\Delta(v, w) = \frac{L(v,w)}{|v|+|w|}$. For $X, Y \in \Sigma^{**}$, let $A(X, Y)$ be the alignment distance using the above Δ function. Also, the gap penalty δ_* is some arbitrary positive value. ♢

Exercise 3.13: 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 3.14: 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 efficient solution to this version of the edit distance problem. ♢

END EXERCISES

§4. Polygon Triangulation

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 2 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.

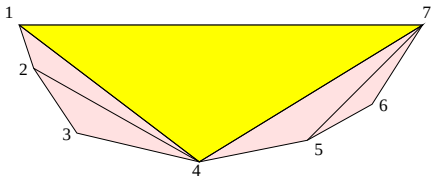


Figure 2: A triangulated 7-gon

¶27. 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 $j = i + 1 (\text{mod}\, n)$; otherwise the segment is called a **chord**. Thus, $1n$ is an edge. If $n \geq 3$, there are exactly n edges and $\binom{n-1}{2} = (n-1)(n-2)/2$ 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.

¶28. 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 2 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 2 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

$$T = T_i \uplus T_i' \uplus S_i$$

where $S_i = T \cap \{(1,i),(i,n)\}$, and T_i, T_i' are (respectively) triangulations of the i-gon $P_i = (1, 2, \ldots, i)$ and the $(n-i+1)$-gon $P_i' = (i, i+1, \ldots, n)$. E.g., the triangulation T in figure 2 has the partition

$$T = T_4 \uplus T_4' \uplus S_4$$

where $S_4 = \{14, 47\}$, $T_4 = \{24\}$ and $T_4' = \{57\}$. Note that $S_i = \{(1,i),(i,n)\}$ iff $2 < i < n-1$, $S_2 = \{(2,n)\}$ and $S_{n-1} = \{(1,n-1)\}$. 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.

¶29. 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.

¶30. Example: Suppose a carpenter has to saw a board P that is shaped as a convex n-gon into $n-2$ triangles. He wants to minimize the amount of sawing to be done. You can interpret this to mean minimizing the amount of sawdust produced. How should be cut up the board?

In case $P = (v_1, \ldots, v_n)$ is a geometric convex 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. Thus, this provides the solution to our carpenter's the sawdust problem.

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 p, q in 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 §5.

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.

¶31. 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

$$C(i,j) = \begin{cases} 0 & \text{if } j = i+1, \\ \min_{i<k<j}\{W(i,k,j) + C(i,k) + C(k,j)\} & \text{else.} \end{cases} \tag{31}$$

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 4.1: Find an optimal triangulation of the abstract pentagon whose weight function W is parametrized 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 4.2: 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 4.3: A more profound generalization of triangulation comes from considering the triangulation (tetrahedralization) of convex polytope in 3-dimensions. Now, the number of tetrahedra is not unique. Give an abstract formulation of this problem. HINT: certain subsets of the vertices are said to be "convex". ♢

Exercise 4.4: (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 4.5: 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 4.6: (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

§5. The Dynamic Programming Method

Let us note the three ingredients necessary for a successful dynamic programming solution. We use the triangulation problem for illustration.

- **There are a small number of subproblems**. We interpret "small" to mean a polynomial number. In the 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 (31).

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

¶32. Mechanics of the algorithm. To organize the computation embodied in equation (31), we use 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 3.

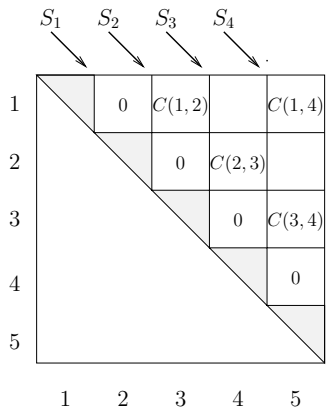


Figure 3: Filling in of a upper triangular matrix

We view the algorithm as a systematic filling in of the matrix 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 (31)), 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 (31):

```
DYNAMIC PROGRAMMING FOR OPTIMAL 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)}
```

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

In general, we would be filling entries of a rank k tensor (matrices are rank $k=2$ tensors). It is harder to visualize this process, but in terms a computer algorithm this presents no extra difficulty: we would just have a $(k+1)$-ply nested for-loop.

¶33. 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 (31) 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 triangular $n \times n$ 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.

¶34. 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 (31) 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.

¶35. Space-Efficient Solutions. We can usually reduce the space usage by a linear factor (quadratic to linear, cubic to quadratic, etc). For instance, in the LCS problem, it is sufficient to keep at most two rows (or two columns) of the matrix in memory. That is because the solution for row i depends only on the solutions of rows $(i-1)$ and row i. Indeed, space for only one row (or column) is already sufficient – as new entries are produced for row i, they overwrite the corresponding entries or row $i-1$. However, such space efficient solutions are not so easy to extend into solutions that reconstruct the optimal solutions. For instance, how do we compute a LCS using $O(n)$ space? To do this, we need a kind of divide and conquer technique: which we explore in the exercises.

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 5.1: Jane Sharp noted an alternative to equation (31).

(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.)

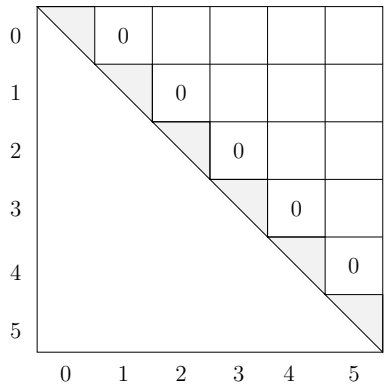
(b) Suppose we remove an ear from an n-gon. The result is an $(n-1)$-gon. If we knew an ear which appears in an optimum triangulation of an n-gon, we could recursively triangular the smaller $(n-1)$-gon. But since we do not know, we can try all possible $(n-1)$-gons obtained by removing an ear. What is wrong with this approach? (Try to write the analogue of equation (31), and think of the 3 ingredients needed for a dynamic programming approach.) ♢

Exercise 5.2: Consider the linear bin packing problem where the ith item is not a single weight, but a pair of non-negative weights, (v_i, w_i). If we put the ith to jth items into a bin, then we require $\sum_{k=i}^{j} v_k$ and $\sum_{k=i}^{j} w_k$ to be each bounded by M. Again the goal to use the minimum number of bins. ♢

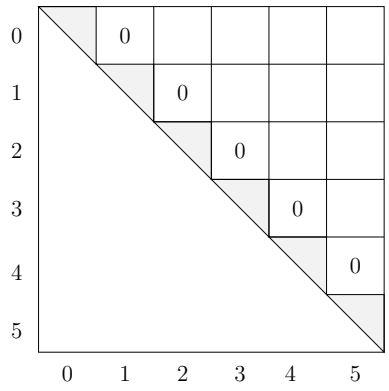
Exercise 5.3: Let $(n_0, n_1, \ldots, n_5) = (2, 1, 4, 1, 2, 3)$. We want to multiply a sequence of matrices, $A_1 \times A_2 \times \cdots \times A_5$ where A_i is $n_{i-1} \times n_i$ for each i. Please fill in matrices (a) and (b) in Figure 4. Then write the optimal order of multiplying $A_1, \ldots, A_5$.

♢

Exercise 5.4: (Google Interview Problem, Feb 2009) You are playing a game with an opponent. Both of you are looking at a list of numbers L. The players moves alternately. To make a move, the player must remove either the head or tail element from L. The score of a player is the sum of all the numbers that the player removes. Your goal is to maximize your score. Construct a dynamic



(a) Optimum Cost Matrix C



(b) Splitter Matrix K

Figure 4: (a) $C[i,j]$ is optimal cost to multiplying $A_i \times \cdots \times A_j$. (b) $K[i,j]$ indicates the optimal split, $(A_i \times \cdots \times A_{K[i,j]})(A_{K[i,j]+1} \times \cdots \times A_j)$

programming algorithm that maximizes your score against any opponent (the opponent might not be as interested in maximizing her own score as in minimizing yours). ♢

Exercise 5.5: The following problem is motivated by computations in wavelet theory. We are given three real non-negative coefficients a, b, c and a real function (the "barrier")

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

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

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

Let $f(x) = \lim_{i\to\infty} f(x,i)$. We call $f(x,i)$ the i-th approximation to $f(x)$. Assume that each arithmetic operation takes unit time.
(a) What is $f(0)$, $f(1/2)$ and $f(-1/2)$?
(b) The function $f(x,i)$ has support contained in the open interval $(-1,1)$ (for fixed i).
(c) Prove that $f(x)$ is well-defined (possibly infinite) for all x.
(d) Determine the time to compute a single value $f(x,n)$ if we implement a straightforward recursion (each call to $f(y,i)$ is independent).
(e) We want an efficient solution for the following problem: given n, m, we want to compute the values $f(i/m, n)$ for all

$$i \in D_m := \{-m+1, -m+2, \ldots, -1, 0, 1, \ldots, m-2, m-1\}.$$

Show that this can be computed in $O(mn)$ time and $O(m)$ space.
(f) Strengthen (e) to show we can compute a single value $f(i/m, n)$ in $O(n)$ time and $O(1)$ space. ♢

Exercise 5.6: (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 $1 \leq i < j < k \leq n$.

(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 5.7: 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 5.8: 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 5.9: 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 5.10: (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 $\pi(x) \leq B_0$. 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 5.11: 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 5. Then $\min(S(c))$ is equal to $\{p,q\}$ if $c \leq 1$; $\{q\}$ if $1 < c \leq 2$; $\{r,s\}$ if $2 < c \leq 3$; $\{s\}$ 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
$$O(\log n + k)$$

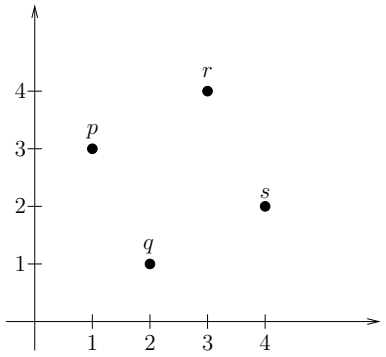


Figure 5: Set of 4 points.

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 5.12: (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 5.13: (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

§6. 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[8] 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 $i = 1, \ldots, n$.

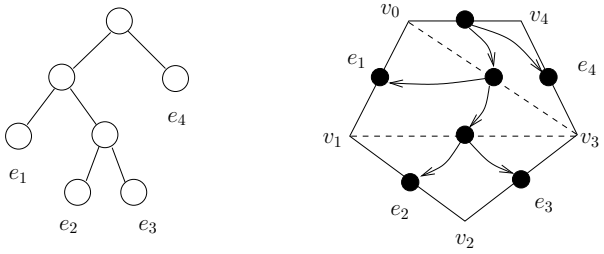


Figure 6: The parenthesis tree and triangulation corresponding to $((e_1(e_2e_3))e_4)$.

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 6 for an illustration. If the $(n+1)$-gon is $(v_0, v_1, \ldots, v_n)$, 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

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

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.

¶36. Encoding parenthesis trees as permutations. We can encode this parenthesis tree on $(e_1, \ldots, e_n)$ by a unique permutation

$$\pi = (\pi_1, \ldots, \pi_{n-1}) \tag{32}$$

of $\{1, 2, \ldots, n-1\}$. Before explaining this in full generality, consider all the 5 possible parenthesis trees on e_1, e_2, e_3, e_4:

$$e_1(e_2(e_3e_4)), \qquad e_1((e_2e_3)e_4), \qquad (e_1e_2)(e_3e_4), \qquad ((e_1e_2)e_3)e_4, \qquad (e_1(e_2e_3))e_4.$$

These are represented, respectively, by the permutations

$$(123), \qquad (132), \qquad (213), \qquad (321), \qquad (312).$$

If $n = 1$, the permutation is the empty sequence $\pi = ()$, and if $n = 2$, the permutation is just $\pi = (1)$. For $n = 3$, there are two permutations $\pi = (12)$ or $\pi = (21)$.

Let us now explain how the permutation (32) encodes a parenthesis tree: if $n = 1$, then $\pi = ()$ is the empty string. the first entry π_1 tells us that the last multiplication is to form the product $A_{1,\pi_1} \cdot A_{1+\pi_1,n}$ where we write $A_{i,j}$ for $\prod_{k=i}^{j} A_k$. Recursively, the next $\pi_1 - 1$ entries in π represents a parenthesis tree on $A_1, \ldots, A_{\pi_1}$, and the remaining $n - \pi_1 - 1$ entries in π represents[9] a parenthesis tree on $A_{1+\pi_1}, \ldots, A_n$. Thus we have demonstrated:

LEMMA 4. *There exists an injection from the set of parenthesis trees on n leaves to the set of permutations on $n-1$ symbols.*

It is clear that the first π_1 entries in (32) must therefore be a permutation on $\{1, 2, \ldots, \pi_1\}$. Therefore, not all permutations on $\{1, \ldots, n-1\}$ correspond to a permutation tree. For $n = 4$, we see that $\pi = (2, 3, 1)$ does not represent any parenthesis tree.

¶37. 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.

From the injection of Lemma 4, we conclude that $P(n) = C(n-1) \leq (n-1)!$. Our current goal is to give a more precise census of parenthesis trees. In general, for $n \geq 1$, the following recurrence is evident:

$$C(n) = \sum_{i=1}^{n} C(i-1)C(n-1-i). \tag{33}$$

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

$$P(n) = \sum_{i=1}^{n-1} P(i)P(n-1-i) \tag{34}$$

[9] Strictly speaking, the last $n - \pi_1 - 1$ entries represent a parenthesis tree on $A_{1+\pi_1}, \ldots, A_n$ in this sense: *if we subtract π_1 from each of these entries, we would obtain (recursively) a permutation representing a permutation tree on $A_1, \ldots, A_{n-\pi_1}$.*

where we define $P(0) = 0$. Thus $P(1) = P(2) = 1, P(3) = 2$.

This recurrence has an elegant solution using generating functions (see §VIII.9),

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

By Stirling's approximation,

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

So $C(m) = \Theta(4^m m^{-3/2})$ grows exponentially and there is no hope to find the optimal parenthesis tree by enumerating all parenthesis trees.

¶38. 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.

To be clear about what we mean by "cheapest way", we must 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,

$$((A_1 A_2)A_3), \quad (A_1(A_2 A_3)) \tag{35}$$

are the two ways of multiplying 3 matrices. Let $T(p, q, r)$ be the cost to to multiply a $p \times q$ matrix by a $q \times r$ matrix. For simplicity, assume the straightforward algorithm for matrix multiplication, so $T(p, q, r) = pqr$. Then, 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 in (35) to multiply these three matrices 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 in (35) 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.

Hence the key problem is this: given the dimension $(a_0, \ldots, a_n)$ of a chain product instance, determine the optimal cost $T_{opt}(a_0, \ldots, a_n)$ to compute such a product. We can solve this problem by reducing it to to the optimal parenthesis tree problem: define an triangular weight function $W(i, j, k)$ for $0 \le i < j < k \le n$ 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.

CLAIM: $T_{opt}(a_0, \ldots, a_n)$ is the minimum W-cost triangulation of the abstract $(n+1)$-gon on the vertex set $\{0, 1, \ldots, n\}$.

We have seen an $O(n^3)$ dynamic programming solution to compute this minimum W-cost triangulation (or equivalently, the corresponding parenthesis tree). 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: 1. Chandra[10] has shown a simple method of multiplying matrices that is within a factor of 2 from T_{opt}. Consider the permutation $\pi = (1, 2, \ldots, n-1)$: according to encoding scheme of (32), this corresponds to the following parenthesis tree on $A_1, \ldots, A_n$:

$$(\cdots((A_1A_2)A_3)\cdots)A_n. \tag{36}$$

This is essentially the left-to-right multiplication of the sequence of matrices. It can be shown that the cost of this method of multiplication is $O(T_{opt}^2)$, and this is tight (Exercise). But suppose we choose i_0 such that $a_{i_0} = \min\{a_0, a_1, \ldots, a_n\}$. Now consider the parenthesis tree represented by the permutation

$$\pi = (i_0 - 1, i_0 - 2, \ldots, 1, i_0 + 1, i_0 + 2, \ldots, n-1, i_0)$$

where the last i_0 is omitted if $i_0 = 0$ or $i_0 = n$. This corresponds to the parenthesis structure

$$(A_1 \cdots (A_{i_0-2}(A_{i_0-1}A_{i_0}))\cdots)(\cdots(A_{i_0+1}A_{i_0+2})\cdots A_n). \tag{37}$$

Then the cost of this computation is at most $2T_{opt}(a_0, \ldots, a_n)$.
2. For the product weight function, $W(a_i, a_j, a_k) = a_ia_ja_k$, the optimal triangulation problem can be solved in $O(n \log n)$ time, using a sophisticated algorithm due to Hu and Shing [6]. Ramanan [9] gave an exposition of this algorithm, and presented an $\Omega(n \log n)$ lower bound in an algebraic decision tree.

EXERCISES

Exercise 6.1: Show that $C(n)$ is the number of binary trees on n nodes. HINT: Use the recurrence (33) and structural induction on the definition of a binary tree. ♢

Exercise 6.2: Work out the bijective correspondence between triangulations and parenthesis trees stated above. ♢

Exercise 6.3: Verify by induction that $C(m)$ has the claimed solution. ♢

Exercise 6.4: Solve the recurrence (33) 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.

♢

[10] "Computing Matrix Chain Products in Near-Optimal Time", Ashok K. Chandra, IBM Research Report RC 5625 (#24393), 10/6/75.

Exercise 6.5: (Chandra)
(i) Show that the method (36) for multiplying the matrix chain $A_1, \ldots, A_n$ is $O(T_{opt}^2)$ where T_{opt} is the optimal cost of multiplying the chain.
(ii) Show that the bound $O(T_{opt}^2)$ is asymptotically tight.
(iii) Show that the method (37) has cost at most $2T_{opt}$. ♢

Exercise 6.6: (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

§7. Optimal Binary 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[11] binary search tree T whose nodes are labeled by

$$q_0, p_1, q_1, p_2, \ldots, q_{n-1}, p_n, q_n \tag{38}$$

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

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

of 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 (38). 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.

¶39. 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.

¶40. 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 7). 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 7: 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 7.1: Describe the precise connection between the optimal search tree problem and the optimal triangularization problem. ♢

Exercise 7.2: 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 7.3: (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 7.4: The following class of recurrences was investigated by Fredman [3]:

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

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 7.5: 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 7.6: 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

§8. 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').*

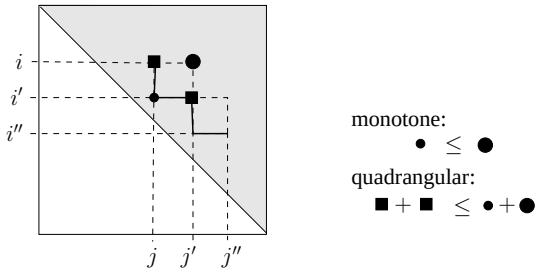


Figure 8: Monotone and quadrangular weight matrix.

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 8). Monotonicity and quadrangularity is also best seen visually (cf. Figure 8):

- 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.

¶41. 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 5. *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

$$W^*(i,j;k) := W(i,j) + W^*(i,k-1) + W^*(k,j), \tag{39}$$

*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 8.1: (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 8.2: (Lemma 5) Verify that the weight matrix for the optimal search tree problem is indeed monotone and satisfies the quadrangular *equality*. ♢

Exercise 8.3: Write a program to compute the derivative of a matrix. It should run in $O(n^3)$ time on an n-square matrix. ♢

Exercise 8.4:

(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 8.5: 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

§9. 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 6. *If W is monotone and quadrangular, then the derived weight matrix W^* is also quadrangular.*

Proof. We must show the quadrangular inequality

$$W^*(i,j) + W^*(i',j') \leq W^*(i,j') + W^*(i',j), \qquad (0 \leq i \leq i' \leq j \leq j' \leq n). \tag{40}$$

First, we make the simple observation when $i = i'$ or $j = j'$, the inequality in equation (40) 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.

¶42. 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.

¶43. 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.**

¶44. 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 $0 \leq i < j \leq n$,

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

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 7. *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

$$K(i,j) \le K(i,j+1). \tag{41}$$

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

$$W^*(i,j;k') \le W^*(i,j;k) \quad \text{implies} \quad W^*(i,j+1;k') \le W^*(i,j+1;k). \tag{42}$$

To see the implication, suppose equation (41) 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 (42). **Q.E.D.**

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

THEOREM 8. *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.**

¶46. Remarks. We refer to [7] 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 [8] 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 9.1: (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 9.2: It is actually easy to give a "graphical" proof of lemma 7. In the figure 9, this amounts to showing that if $A + a \geq B + b$ then $A' + a' \geq B' + b'$.

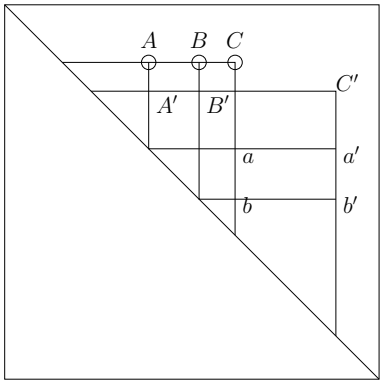


Figure 9: Derived weight matrix.

♢

Exercise 9.3: If W is monotone and quadrangular, is W^* monotone? ♢

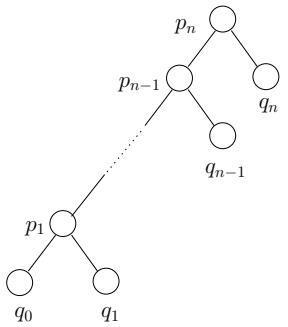


Figure 10: Linear list search tree.

Exercise 9.4: 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? ♢

Exercise 9.5: (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 9.6: 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

§10. 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 may not be practical in some applications. 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 [4] for such examples.

References

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

[2] D. Z. Chen, O. Daescu, X. Hu, and J. Xu. Finding an optimal path without growing the tree. *J. Algorithms*, 49(1):13–41, 2003.

[3] 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.

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

[5] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Comm. of the ACM*, 18(6):341–343, 1975.

[6] 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.

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

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

[9] 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.