"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 physical interpretation of "programming" to our contemporary understanding 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` in 2008. In 2020, it asks if I meant `CGT AATCC` and the search results for `CGTAATCC` (see Figure 1) included a reference to this particular paragraph of our lecture notes!

`CCGTCC.com` is the homepage for members of Casino Chip & Gaming Token Collectors Club

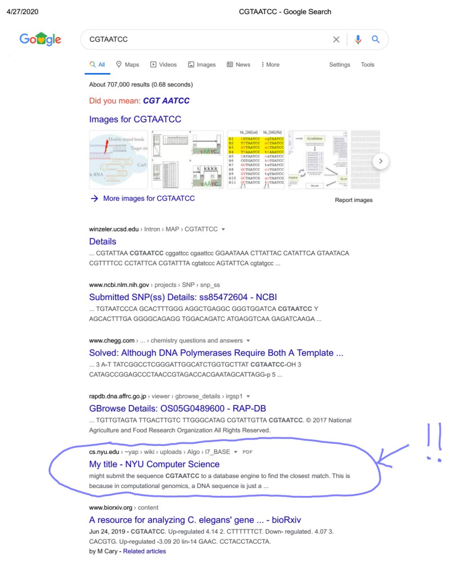
But a biologist might submit the sequence `CGTAATCC` to a database engine to find the closest match. This is because in computational genomics, a DNA sequence is just a string over the symbols `A,C,G,T` representing the four bases (adenine, cytosine, guanine and thymine) in DNAs. The strings in Google 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 to show you "closely related" strings meant that these algorithms have (1) some implied some measure of similarity or distance between strings, and (2) some database of strings in which to search for

[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 these possibilities like `string cheese`, `string theory`, `stringbuilder`, etc. In 2012, as in 2008, it offers only one alternative `strings`. In 2020, it simply gives the results for `strings` (with an option to really search for `strnigs`). The results included stores on Google map selling musical string instruments; a literal search for `strnigs` appears to shoe-horn some results on `strings`.

[3]In 2011, it asked if I wanted `CHEATCC` which led to websites with video game cheats, cheat codes, hints and tips. Here are Google's offer for 2012: `cheatcc`, `ggatcc`, `cgtalk`, `cgtuts`.


4/27/2020 CGTAATCC - Google Search

CGTAATCC

All · Maps · Videos · Images · News · More · Settings · Tools

About 707,000 results (0.68 seconds)

Did you mean: ***CGT AATCC***

Images for CGTAATCC

→ More images for CGTAATCC Report images

winzeler.ucsd.edu › Intron › MAP › CGTATTCC
Details
... CGTATTAA **CGTAATCC** cggattcc cgaattcc GGAATAAA CTTATTAC CATATTCA GTAATACA CGTTTTCC CCTATTCA CGTATTTA cgtatccc AGTATTCA cgtatgcc ...

www.ncbi.nlm.nih.gov › projects › SNP › snp_ss
Submitted SNP(ss) Details: ss85472604 - NCBI
... TGTAATCCCA GCACTTTGGG AGGCTGAGGC GGGTGGATCA **CGTAATCC** Y AGCACTTTGA GGGGCAGAGG TGGACAGATC ATGAGGTCAA GAGATCAAGA ...

www.chegg.com › ... › chemistry questions and answers
Solved: Although DNA Polymerases Require Both A Template ...
... 3 A-T TATCGGCCTCGGGATTGGCATCTGGTGCTTAT **CGTAATCC**-OH 3 CATAGCCGGAGCCCTAACCGTAGACCACGAATAGCATTAGG-p 5 ...

rapdb.dna.affrc.go.jp › viewer › gbrowse_details › irgsp1
GBrowse Details: OS05G0489600 - RAP-DB
... TGTTGTAGTA TTGACTTGTC TTGGGCATAG CGTATTGTTA **CGTAATCC**. © 2017 National Agriculture and Food Research Organization All Rights Reserved.

cs.nyu.edu › ~yap › wiki › uploads › Algo › l7_BASE PDF
My title - NYU Computer Science
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 ...

!!

www.biorxiv.org › content
A resource for analyzing C. elegans' gene ... - bioRxiv
Jun 24, 2019 - **CGTAATCC**. Up-regulated 4.14 2. CTTTTTTCT. Down- regulated. 4.07 3. CACGTG. Up-regulated -3.09 20 lin-14 GAAC. CCTACCTACCTA.
by M Cary - Related articles

Figure 1: Google search for `CGTAATCC` on April 27, 2020

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. First Glimpses of Dynamic Programming

¶2. Divide and Conquer with a twist. In Chapter II, we looked at divide-and-conquer problems. Dynamic programming is also a form of divide-and-conquer because it is based on solving subproblems. But it has some distinctive features. A simple illustration is provided by the computation of Fibonacci numbers, $F(n) = F(n-1) + F(n-2)$. Assume $F(0) = 0$ and $F(1) = 1$. On input $n > 1$, the obvious recursive algorithm calls itself twice, on the arguments $n-1$ and $n-2$ respectively. 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 $\geq 1$
    $(a, b) \leftarrow F_2(n-1)$        ◁ Recursive call!
    Return$(a + b, a)$
```

The running time $T_2(n)$ for this algorithm satisfies the recurrence $T_2(n) = T_2(n-1) + 1$ and thus $T_2(n) = n$. Thus, we have achieved an exponential speedup over naive recursion. Now, another "recursive idea" can give us a running time $T_3(n)$ with yet another exponential speedup: $T_3(n) = O(\log n)$. See Exercise.

Here we see the seed of the dynamic programming idea — by keeping around solutions to subproblems, we avoid their recomputation, and avoid what would otherwise be an exponential complexity. In the Fibonacci computation, we keep track of solutions to two subproblems. This is not typical of the dynamic programming problems we will study: generally, we must keep track of solutions to a polynomial number of subproblems. Thus, we cannot bound the complexity of dynamic programming problems using the Master Recurrence $T(n) = aT(n/b) + f(n)$. This is because the number of subproblems in this recurrence is bounded by a constant $a > 0$.

¶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 riders in a queue. The goal was 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 (OK, our joy ride interpretation is now stretched). In any case, the greedy algorithm breaks down. For instance let $M = 5$ and $w = (1, 2, 3, 4, -5)$. The greedy solution use 3 cars $(1, 2), (3), (4, -5)$ but the optimal solution uses only one car since $1 + 2 + 3 + 4 - 5 = 5$. To achieve this optimal solution, we must give up our online policy (i.e., to decide on each rider without looking at what comes after in the queue). But we keep the First Come First Ride (FCFR) policy. 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 may 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 $(w_i, \ldots, w_n)$ where $w_i + w_{i+1} + \cdots + w_n \le M$. Notice that b_{i-1} is the optimal number of cars for the subproblem $P_{i-1} = (w_1, \ldots, w_{i-1})$. Hence we conclude that $P_n = 1 + b_{i-1}$ (the "1+" comes from the last car). But what is i? Well, we can search for this i using the following formula:

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

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 optimal values $b_i$
      $b[0] \leftarrow 0$.
      for $k = 1, \ldots, n$
          $W \leftarrow 0$
          $B \leftarrow +\infty$      ◁ $B$ is current min value of $b_k$'s
          for $i = k, k-1, \ldots, 2, 1$
              $W \leftarrow W + w[i]$
  (*)         If $(W \le M)$, $B \leftarrow \min\{B, b[i-1]\}$      ◁ $B$ is updated
          $b[k] \leftarrow 1 + B$
```

We have two nested for-loops: the inner for-loop computes $b[k]$ by maintain the current minimum value B in the set

$$\{b[i-1] : i \text{ is valid for } k\}$$

where "i is valid for k" means $w_i + w_{i+1} + \cdots + w_k \le M$ (see (1)). Line (*) checks if i is valid, and if it is, B is updated to $\min\{B, b[i-1]\}$. After the inner loop, we set $b[k] \leftarrow 1 + B$. The initialization $B \leftarrow +\infty$ allows the possibility that $b[k] = \infty$ (i.e., problem instance P_k has no solution). The overall complexity is $T(n) = \sum_{k=1}^{n} O(k) = \Theta(n^2)$.

This solution is typical of dynamic programming: the solution for problem instance P_n can be efficiently computed from the solutions to a polynomial number of subproblems $(P_1, \ldots, P_n)$.

¶4. String Notations. Let us fix some common terminology for strings. An **alphabet** is just a finite set Σ whose elements are called **letters** (or characters or symbols). A **string** (or word) is just a finite sequence of letters. The set of all 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 for the cardinality $|S|$ 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 $|XY| = |X;Y| = |X| + |Y|$.

Exercises

Exercise 1.1: We have shown that the Fibonacci numbers $F(n)$ can be computed in $\Theta(n)$ arithmetic operations. Design an algorithm to compute this in $\Theta(\log n)$ arithmetic operations.

HINT: Reduce this to a problem of multiplying n copies of the 2×2 matrix $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$.

What are the entries of the nth power M^n? How many matrix products is needed to compute M^n (recall that matrix multiplication is associative)? ◇

Exercise 1.2: 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.

(a) 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`?
(b) We can also exchange any two non-consecutive characters of `strings`. there are 10 such choices, such as `rtsing, itrsng, ntrisg, gtrins; sirtng, snritg`, etc.
(c) Repeat the experiment, but starting with the sequence `strings`. ♢

Google in Fall 2011: (a) In the first experiment, Google correctly identify `string` as your intention.
(b) This was not done.
(c) For this experiment, it sometimes think you want `signs`.

Exercise 1.3: Compare the different search engines: Google, Yahoo, Bing, Amazon.com, eBay, Twitter, Wikipedia(en). Caveat: These are moving targets!

Fall 2012: suppose you typed `tsring`. Google, Yahoo, Bing, Wikipedia all think it is `string` you want. Amazon.com thinks you are looking for rings ("RingVille Sign" shows up). eBay think it is `string` and advertises many sellers of sexy outfits. Twitter says "over capacity (please try again)".
Nov 11, 2013: Google returns various facebook accounts with last name of `Tsring` (a Tibetan last name), but also helpfully suggests `string`. Both Yahoo and Bing assumes it is `string` that you want. Amazon.com suggests `tring` (as in T-ring in cameras) instead. Wikipedia asks if you mean `turing` as `tsring` does not exist. Twitter (just having gone public last week) gives 19 hits.

♢

Exercise 1.4: We have a game on a list L of *positive* integers. There are two players (Alice and Bob) who alternately remove a number from L in their turn. Assume that the play *must* remove a number when it is their turn. When a player removes a number the number is added to that player's score. Numbers can only be removed from the front or back end of L, and the game ends when L becomes empty. The player with the higher score wins; we have a tie when both players have the same score. Note that each player has the entire list is full view. Let Alice be make the first move. Show that Alice never loses if the size of L is *even*, but she may not win if the size of L is odd. Give a method to compute the optimal score for Alice. ♢

Exercise 1.5: (Maximum of subarray sums) Given an array $A[1..n]$ of positive and negative numbers, let $S[i,j]$ denote the sum of the numbers in the subarray $A[i..j]$ $(1 \le i \le j \le n)$. We may assume there are no zeros in A. Give a fast algorithm to compute the maximum of $M^* := \max\{S[i,j] : 1 \le i \le j \le n\}$. Give a linear time algorithm; prove its correctness.
♢

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 `notation` and `notions` clearly have much overlap. But do `notation` 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.

¶5. String Concepts. A **subsequence** of $X = x_1 \cdots x_m$ is a string $Z = z_1 z_2 \cdots z_k$ such that for some

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

we have $Z[\ell] = X[i_\ell]$ for all $\ell = 1, \ldots, k$. E.g., `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. E.g., `og` is a common subsequence of $X =$ `long` and $Y =$ `bog`. We call Z a **longest common subsequence** if its length $|Z| = k$ is maximum among all common subsequences of X and Y. A longest common subsequence of X, Y may not be unique. We define several related notions:

- Let $LCS(X,Y)$ denote the set of longest common subsequences of X, Y. Note that $LCS(X,Y) = \{\epsilon\}$ iff there is no letter in common between X and Y.
- Let $lcs(X,Y)$ denote *any* element of $LCS(X,Y)$: so $lcs(X,Y) \in LCS(X,Y)$.
- Let $L(X,Y) := |lcs(X,Y)|$ (length function). So $L(X,Y) = 0$ iff there is no letter in common between X and Y.
- Let $\lambda(X,Y) := |LCS(X,Y)|$ (cardinality function). Note that $\lambda(X,Y) \ge 1$ since "at worst", $LCS(X,Y)$ is the singleton comprising the empty string ϵ or a string with a single letter.

For example, if

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

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

Of course, the ultimate in similarity under the overlap 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.

¶6. Versions of LCS Problems. There are several versions of the **longest common subsequence (LCS) problem**. Given two strings X and Y, 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., $LCS(\texttt{longest}, \texttt{lengthen}) = \{\texttt{lngt, lnge}\}$.

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; but we can introduce suitable **implicit representations** of $LCS(X,Y)$ to avoid exponential size (see ¶15). See the Exercise for the **multiset version** of $LCS(X,Y)$, denoted $LCS^+(X,Y)$. E.g., if $X = \texttt{aaa}$, $Y = \texttt{a}$, then $LCS^+(X,Y) = \{a,a,a\}$ (i.e., a has multiplicity 3). Likewise, $\lambda^+(X,Y)$ can count the members of $LCS^+(X,Y)$ with multiplicity, so $\lambda^+(\texttt{aaa}, \texttt{a}) = 3$.

¶7. 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:

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

For example,

$$LCS(X_2, Y_2) = \{0a0a, 0a1a, 1a0a, 1a1a\}.$$

In general, we claim

$$L(X_n, Y_n) = 2n, \quad \lambda(X_n, Y_n) \ge 2^n$$

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. Actually, $\lambda(X_2, Y_2) = 4$ is misleading, as it suggests that the lower bound $\lambda(X_n, Y_n) \ge 2^n$ is tight. Asymptotically, $\lambda(X_n, Y_n)$ grows almost quadratically faster than 2^n:

$$\lambda(X_n, Y_n) = \Theta(4^n/\sqrt{n})$$

(see Exercise).

¶8. 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 $|X'| = |X| - 1$.

Lemma 1 *We have the following recursive formula for $L(X,Y)$:*

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

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

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

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 $\le N-1$. We will discuss the dynamic programming principle in §4.

For any string X and natural number $i \ge 0$, 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.

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

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 $x_m \ne y_n$ 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 $c \ne b$. 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 subsequence of X' and Y'. Since $w'c$ is the longest common subsequence of $X'b$ and $Y'b$, we conclude that w' must be the longest common subsequence 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.

¶9. 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[4] the example (2) to the strings $X =$ `lengthen` and $Y =$ `elongate`:

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

			1	2	3	4	5	6	7	8
			e	l	o	n	g	a	t	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: Computing $lcs(X, Y)$: (a) Rules for filling table, (b) Recovery of $lcs(X, Y)$

To see the formula (4) in action, we consider two entries. (Equality Case) The entry corresponding to the 'g'-row and 'g'-column is filled with $1 + w$ where w is the entry in the previous row and column. (Inequality Case) 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 $LCS(X, Y) = \{$`lngte`$,$ `engte`$\}$ in this example. We leave as an exercise to program this algorithm in your favorite language.

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

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

¶11. Recursive Solution. A naive recursive solution can be exponential time. But a technique called **memo-izing** can avoid the exponential behavior. Our goal is to write a recursive routine to compute $L(X, Y)$. We need a global data structure, which is our matrix $(m + 1) \times (n + 1)$ matrix M where $m = |X|, n = |Y|$. Assume every entry of the matrix M is

[4] No pun in-tended.

initialized to a negative number (say, -1). Also, let X, Y be global data. Then our $L(X,Y)$ can be defined to be $L(|X|,|Y|)$, which calls the following subroutine:

```
L(i,j):
  ▷ BASE CASE
  If ($ij = 0$) then Return 0
  ▷ RECURSIVE CASES
  If ($M(i,j) < 0$) then        ◁ Need to compute value
      If ($x_i = y_j$) then
          $M[i,j] \leftarrow 1 + L(i-1, j-1)$
      else
          $M[i,j] \leftarrow \max\{L(i-1,j), L(i,j-1)\}$
  Return $M[i,j]$     ◁ Return computed value
```

The intuitive idea is that we compute each $L(i,j)$ at most once, and so the overall worst case is $O(mn)$. But we need to charge the work carefully. For instance, for any i,j, we could call $L(i,j)$ more than once. For each i,j, we say that a recursive call to $L(i,j)$ is **productive** if $M(i,j) < 0$ at the beginning of the call. All other calls are **non-productive**. At the end of the productive call, $M(i,j) \geq 0$. Hence, all subsequent calls to $L(i,j)$ will be non-productive. Also, all calls to $L(i',j')$ during the productive call to $L(i,j)$ has the property that $i'+j' < i+j$. In particular, $(i',j') \neq (i,j)$. This proves that we have at most one productive call for each (i,j).

All the work in this algorithm will be charged to the productive calls. This charge is $O(1)$. Since there are $\leq mn$ productive calls, the overall cost is $O(mn)$.

The recursive method can be more efficient than the non-recursive version of the algorithm: for example, the non-recursive algorithm has a lower bound of $\Omega(mn)$. But the recursive method can cost as little as $O(m+n)$ (in the best case). Indeed, this bound is achieved in the case where $X = Y$.

¶12. Recovery of Optimal Instance. Given the full matrix $L[0..m, 0..n]$, we can recover an optimal instance $lcs(X,Y)$. We describe a simple way to construct $lcs(X,Y)$. For a concrete example, consider $X = $ `final` and $Y = $ `infill`. The corresponding matrix $L[0..m, 0..n]$ $(m = 5, n = 6)$ is shown in Table 2. To avoid clutter, we use a "row-convention" whereby we only display an entry *only if it is larger than the entry to its left on the same row.*

row-convention for displaying matrix

Begin with the entry $L[m,n]$, which contain the value $L(X,Y)$. We are going to trace a path to any entry $L[i,j]$ containing the value 0. In general, we are at some entry $L[i,j]$, holding some value $\ell = L(X_i, Y_j)$. If $\ell = 0$, we are done. Otherwise, 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, and we may move to any cell that contains ℓ. When $\ell = 0$, this process has output the string $lcs(X,Y)$, but in reverse order.

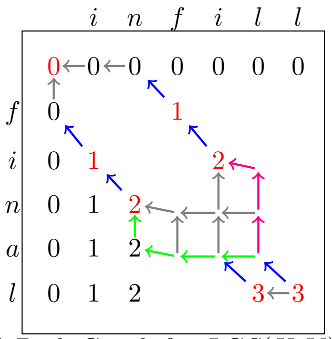
Let us apply this procedure to the matrix in Table 2. To break ties, let us move to the neighbor cell in the same row:

$$[5,6] \overset{l}{-} [4,5] - [4,4] - [4,3] - [4,2] - [3,2] \overset{n}{-} [2,1] \overset{i}{-} [2,0].$$

This corresponds to the blue-green path in Table 2(b), and outputs the string $lcs(X,Y) = $ `inl`.

		i	n	f	i	l	l
	0	0	0	0	0	0	
f	0	0		1			
i	0	1			2		
n	0	1	2				
a	0	1	2				
l	0	1	2			3	3

(a) Matrix for $L(X,Y)$


(b) Path Graph for $LCS(X,Y)$

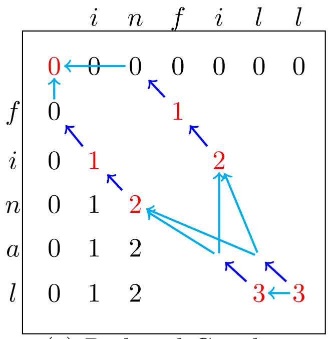

(c) Reduced Graph

Table 2: Input strings $X =$ `final`, $Y =$ `infill`.

Instead, if we break ties by moving to the neighbor in the same column, we obtain

$$[5,6] \overset{l}{-} [4,5]-[3,5]-[2,5]-[2,4] \overset{i}{-} [1,3] \overset{f}{-} [0,2].$$

This corresponds to the blue-magenta path in Table 2(b), and outputs the string $lcs(X,Y) =$ `fil`. Indeed, we may check that $LCS(X,Y) = \{$`inl`, `fil`$\}$.

¶13. Representation of $LCS(X,Y)$. Next let us recover the set $LCS(X,Y)$. Intuitively, we extend the $lcs(X,Y)$ algorithm above to follow "all the possible paths". For example, if $x_i \neq y_j$ and $L[i-1,j] = L[i,j-1]$, then we now need to move to $L[i-1,j]$ as well as $L[i,j-1]$. But we know the size $\lambda(X,Y)$ of this set is exponential in in the worst case. We must avoid an explicit listing of the members of $LCS(X,Y)$, but seek some compact representation that allows us to query membership in $LCS(X,Y)$ efficiently. So, instead of following all possible path, we collect all these paths into a "path graph" as illustrated in Table 2(b). This graph is a DAG with source $[m,n] = [5,6]$ and sink $[0,0]$. We may count 11 distinct paths from source to sink. But we already noted that $\lambda(X,Y) = |LCS(X,Y)| = 2$ in this example. Upon reflection, we see that there are not really 11 distinct meaningful matches. So what do these paths count? There are two issues:

- Only the diagonal (blue) edges represent matches; the horizontal and vertical edges do not. The solution is to define a "reduced graph" from the "path graph" by keeping only vertices corresponding to matches (such vertices have red entries connected by blue diagonal edges). See Table 2(c). The original diagonal (blue) edges are now supplemented by (cyan) edges that connect two vertices if they are connected by a monotone path of horizontal and vertical edges in the path graph.
- Each string in $LCS(X,Y)$ might by more than one one choice of subsequences in X and/or in Y. For instance, `inl` arise in two ways since the letter `l` can come from `infill` in two ways. Similarly, `fil` arise in two ways. In our reduced graph Table 2(c), we count four paths from $[m,n]$ to $[0,0]$. These corresponds to these four ways of obtaining an LCS. Thus, the reduced graph could be further reduced for the purposes of non-redundant representation of $LCS(X,Y)$. We leave this further reduction as an exercise. For now, we are happy with the reduced graph as a fairly compact representation of $L(X,Y)$.

¶14. Small Space Solution. The above algorithm for $lcs(X,Y)$ uses $O(mn)$ space. For Google applications, this may be acceptable because m,n is typically small. In computational

how long a search string would you type?

genomics, this is not acceptable because of long gene sequences. 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 that in optimization problems (including dynamic programming), it is usually easy to modify the code for computing the optimal value to actually deliver the optimal construction. For LCS problem, we expect to convert the program for $L(X,Y)$ to also deliver $lcs(X,Y)$ or some representation of $LCS(X,Y)$. We now have a counter-example the general validity of this claim: you could not recover $lcs(X,Y)$ using the small space solution here. We need another method.

¶15. 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 $|X^i| = i$. For instance, if $|X| = m$ then $X^{\#}$ is the same as X^{m-1}. 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 the matrix entry $L[i,j]$ hold the value $L(X^{m-i}, Y^{n-j})$. 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 gained really nothing new with this backward approach. But when combined with the forward approach, we obtain something new.

¶16. Recovery of Optimal Instance in Small Space. Now we address the possibility of computing $lcs(X,Y)$ in small space. We 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,

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

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:

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

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 $T(m,n) \le 2mn$: if $n = 1$, this is true. Otherwise,

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

¶17. 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 $G_0(X,Y)$ as follows. The node set of $G_0(X,Y)$ 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 $G_0(X,Y)$ 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 $G_0(X,Y)$ is quite wasteful: let $G_1(X,Y)$ be a compact version of $G_0(X,Y)$ in which we only retain node (m,n) (source), node $(0,1)$ (sink), and any node (i,j) where $x_i = x_j$. We introduce an edge (i,j)–(i',j') in $G_1(X,Y)$ iff there is a path from (i,j) to (i',j') such that for all (i'',j'') in this path, $L[i'',j''] = L[i,j]$. Furthermore, any node not reachable from the source is discarded. The nodes of $G_1(X,Y)$ are naturally partitioned into **levels** where the level of a node (i,j) is $L[i,j]$ (the source is its own level). Moreover, edges only go from level $k+1$ to level k ($k = 1,\ldots,L[X,Y]$). Also, the nodes $(0,0)$ and (m,n) are retained, corresponding to the unique **sink** and **source**. We introduce two kinds of edges: (1) Normal edges are (i,j)–(i',j') where $L[i,j] = L[i',j'] + 1$ and there is a path from (i,j) to (i',j') in $G_0(X,Y)$. 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 2, the result is a level graph in

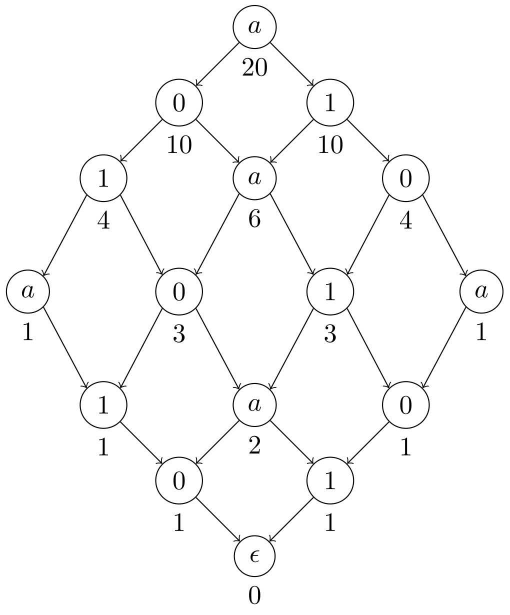


Figure 2: Graph $G_1(X_3, Y_3)$ represents $LCS(X_3, Y_3)$

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 2, we show the graph $G_1(X, Y)$ for the pair $(X, Y) = (X_3, Y_3) = (01a01a01a, 10a10a10a)$ defined in (3). The name (i, j) of a node is not explicitly given, instead we 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 $LCS(X_3, Y_3)$. Note that $G_1(X_3, Y_3)$ shown has only 16 nodes, compared to 100 nodes for $G_0(X_3, Y_3)$.

External to each node in Figure 2, a numerical value is indicated: what are these?

¶18. 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.

¶19. 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: $A,\ C,\ G,\ T$. 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, please compute $L(X,Y)$:
(a) $X =$ `lengthening` and $Y =$ `elongation`.
(b) $X =$ `prelengthening` and $Y =$ `postelongation`. ♢

Exercise 2.2: Compute $L(X,Y)$ and $lcs(X,Y)$ for $X =$ `AATT, CCCC, GACT, GCAA, TTCA, CGCA, CC` and $Y =$ `GGCT, TTTA, TTCT, CCCT, GTAA, GT`. These are parts of DNA sequences from a modern human and a Neanderthal, respectively. You may do a hand simulation or write a simple program (please show the program) to compute the solution. ♢

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: Give the analogue of (8) for $lcs(X,Y)$. ♢

Exercise 2.6: Develop an output-sensitive algorithm for $LCS(X,Y)$ as outlined in the text. In particular, you must determine how to detect branching of paths that are necessary or which might be unnecessary. ♢

Exercise 2.7: Consider the example in the text ¶6, $X_n = (01a)^n$ and $Y_n = (10a)^n$. Let $\lambda_n := \lambda(X_n, Y_n)$. We want to improve the bound $\lambda_n \geq 2^n$ in the text.
(a) Compute $L(X_2,Y_2)$ by filling in the the usual matrix. Further determine $\lambda(X_2,Y_2) = |LCS(X_2,Y_2)|$.
(b) Prove that $L(X_n,Y_n) = 2n$.
(c) Using your result in part (a), show that $\lambda_n = \Omega(\sqrt{6}^n)$.
(d) Provide an exact closed formula for λ_n, and show that this is $\Theta(4^n/\sqrt{n})$. HINT: There is the usual digraph constructed from the the entries of the matrix $L[0..3n, 0..3n]$, with unique source $(3n,3n)$ and sink $(0,0)$. Count the number of paths from $(3n,3n)$ to $(0,0)$. ♢

Exercise 2.8: (Laura Florescu) Let us now define $X_1 = 01ab$ and $Y_1 = 10ba$. For $n \geq 2$, let $X_n = (X_1)^n$ and similarly for Y_n. Determine $L(X_n,Y_n)$. ♢

Exercise 2.9: Show the following combinatorial identity:

$$\binom{2n}{n} = \sum_{i=0}^{n} \binom{n}{i}^2. \tag{14}$$

HINT: Count the number of monotone paths from $(0,0)$ to (n,n) in the $n \times n$ grid.

$\diamondsuit$

Exercise 2.10: Suppose we further compress $G_1(X,Y)$ into a digraph denoted $G_2(X,Y)$ by collapsing groups of nodes in each level. Specifically, each node (i,j) other than the source or sink can be labeled by the symbol $a \in \Sigma$ where $a = x_i = y_j$. Then all the nodes with the same label in a given level can be identified. Let node u be collapsed into the node $\overline{u}$. The edges of $G_2(X,Y)$ are of the form $\overline{u}–\overline{v}$ corresponding to edges $u–v$ in $G_1(X,Y)$. Now, each level has at most $|\Sigma|$ nodes. What is wrong with using $G_2(X,Y)$ to represent $LCS(X,Y)$? Give a concrete example. $\diamondsuit$

Exercise 2.11: Give an $O(mn)$ time algorithm to compute $\lambda(X,Y)$. $\diamondsuit$

Exercise 2.12: 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 $|G(S)| \leq 4|SCS(S)|$. $\diamondsuit$

Exercise 2.13: What are the forbidden configurations in the matrix $L[0..m, 0..n]$ used in the computation of $L(X,Y)$?
(a) Suppose $L[i,j] = 89$, what are the possible values of $L[i-1, j-1]$?
(b) Show the following constraints: $0 \leq L[i,j] - L[i-1,j] \leq 1$ and $0 \leq L[i,j] - L[i,j-1] \leq 1$.
(c) If $x_i = y_j$ then $L[i,j] = L[i-1,j] = L[i,j-1]$ is impossible.
(d) The above constraints are based only on adjacency matrix entries. Are there global constraints not dependent on these local constraints? $\diamondsuit$

Exercise 2.14:
(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 $LCS(X,Y)$. $\diamondsuit$

Exercise 2.15: 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 2.16: 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 2.17: 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 2.18: Modify the algorithm for $L(X, Y)$ to compute the following functions:
(a) $\lambda'(X, Y)$
(b) $\lambda(X, Y)$ ♢

Exercise 2.19: 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 2.20: (a) Give the dynamic programming principle for the length $L(X, Y, Z)$ of the longest common subsequence of three strings X, Y, Z.
(b) Give an algorithm for the computing $L(X, Y, Z)$. ♢

Exercise 2.21: Let us generalize the previous question to an arbitrary number of strings, to computing $L(X_1, \ldots, X_m)$ where m is not fixed. But for simplicity, assume that all X_i's have the same length of n.
(a) Give an algorithm for this problem.
(b) Bound the complexity of your algorithm in terms of m and n. ♢

Exercise 2.22: 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 2.23: 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 2.24: 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 2.25: (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 2.26: 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

¶20. Similarities and Differences. The Least Common Subsequence (LCS) metric measures the *similarity* between two strings (based on the notion of common subsequence). We now introduce another approach called "edit distance" which measures the *differences* between two strings (based on the "amount change necessary to eliminate the differences"). If X, Y are two strings, the first approach computes the length $L(X,Y)$ of their longest common subsequence. The new approach computes the minimum edit distance $D(X,Y)$ to transform X to Y. These two approaches are complementary: imagine that X is fixed, but we can change Y incrementally $(Y_1, Y_2, \ldots)$. If $L(X,Y_i)$ increases with i, the two strings strings are becoming more similar. But if $D(X,Y_i)$ increases, they are becoming more different. How shall we measure change? We borrow the idea of editing a string using a text editor in a computer. Text editors have a repertoire of basic operations, and 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. If $X = \texttt{cat}$ and $Y = \texttt{dog}$, 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).

our favorite editor? a "vi" derivative called "gvim"

This simple model can be generalized in two ways: first, we can associate a positive cost with each operation, and $D(X,Y)$ is just the minimum total cost to convert X to Y. Counting operations amounts to charging one unit per operation. Second, we may broaden the repertoire to admit more types of operations. E.g., if we have the operation to replace any letter in a string by any other letter, then $D(\texttt{cat}, \texttt{dog}) = 3$ as three replacement operations suffice: *Replace*$(\texttt{c,d})$, *Replace*$(\texttt{a,o})$, *Replace*$(\texttt{t,g})$. Instead of the text editing interpretation of this problem, we can also interpret strings as genetic material, and the basic operations as "elementary genetic modifications". Again, we could have insertions and deletions, but **transposition** of a pair of letters is also common. With transposition, $D(\texttt{log}, \texttt{lgo}) = 1$. Because of such interpretations, $D(X,Y)$ is a popular measure of similarity in computational biology.

¶21. The Standard Edit Distance. We now specify the standard repertoire of edit operations. Fix any alphabet Σ. For any index $i \ge 1$ 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

$$Ins(i,a,X), \quad Del(i,X), \quad Rep(i,a,X) \tag{15}$$

denote the strings that are produced by these respective operations. For example, if $X = AATCGA$ then

$$Ins(3, G, AA\underline{T}CGA) = AA\underline{G}TCGA,$$

$$Del(5, AATC\underline{G}A) = AATC|A$$

$$Rep(5, T, AATC\underline{G}A) = AATCTA.$$

In general, if $Y = Ins(i,a,X)$, then $|Y| = 1 + |X|$ and

$$Y[j] = \begin{cases} X[j] & \text{if} \quad j = 1,\ldots,i-1, \\ a & \text{if} \quad j = i, \\ X[j-1] & \text{if} \quad j = i+1,\ldots,|X|. \end{cases} \tag{16}$$

If $Y = Del(i, X)$, then $|Y| = |X| - 1$ and

$$Y[j] = \begin{cases} X[j] & \text{if} \quad j = 1, \ldots, i-1, \\ X[j+1] & \text{if} \quad j = i, \ldots, |X|-1. \end{cases} \tag{17}$$

These two operations $Del(i)$ and $Ins(i, a)$ are inverses of each other in the following sense:

$$\left.\begin{array}{lcll} |Ins(i,a,X)| & = & |X|+1 & \\ |Del(i,X)| & = & \max\{0, |X|-1\} & \\ Del(i, Ins(i,a,X)) & = & X & \\ Ins(i,b,Del(i,X)) & = & X & \text{for some } b \in \Sigma. \end{array}\right\} \tag{18}$$

What if i is improper? The definitions in (16) and (17) tacitly assume that i is in the "proper range": For insertion, this means $i \le |X| + 1$, but for deletion and replacement, this means $i \le |X|$. When i is **improper** for the operation, we could simply declare such operations to be undefined. Another solution is to declare these as no-op (X is not changed) whenever i is not in proper range. Here is our convention:

- For $Ins(i, a, X)$, if i is improper, we insert a after the last letter of X.
- For $Del(i, X)$ and $Rep(i, a, X)$, if i is improper, we delete or replace the last letter of X.

We check that these conventions extends the validity of (18).

We define the **edit distance** $D(X, Y)$ between X and Y to be the minimum number of standard edit operations necessary to transform X to Y. For example, $D(\texttt{TAG}, \texttt{CAT}) \le 2$ since

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

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

The operation of replacement (Rep) is easily replaced (sic!) by an insertion and an deletion, so its use is not essential for "completeness" of editing operations. However, we will see that it is the basis of a very important generalization when we turn to the alignment problem.

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 $D : \Sigma^* \times \Sigma^* \to \mathbb{R}_{\ge 0}$, constitutes a **metric space**. This amounts to satisfying the following natural properties:

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

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

(iii) (Triangular Inequality)

$$D(X, Z) \le D(X, Y) + D(Y, Z). \tag{19}$$

We also have the following bounds:

$$|X| - |Y| \le D(X, Y) \le |X| \tag{20}$$

where $|X| \ge |Y|$ (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 operations just to decrease the length of X to that of Y. The upper bound is sufficient because, by using $|Y|$ replacement operations, we can 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 $D(\texttt{google}, \texttt{search}) = 6$.

¶22. 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 (15) that transforms X to Y. Paths in $G(\Sigma)$ are called **edit paths** and edit distances have the following interpretation:

$$D(X,Y) \text{ is the length of the shortest (link-distance) path from } X \text{ to } Y \text{ in } G(\Sigma). \qquad (21)$$

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

$$\boxed{D(X,Y) = \begin{cases} \max\{|X|,|Y|\} = \max\{m,n\} & \text{if } mn = 0 \quad \text{(Base case)} \\ D(X',Y') & \text{if } x_m = y_n \quad \text{(Equality case)} \\ 1 + \min\{D(X',Y), D(X,Y'), D(X',Y')\} & \text{if } x_m \ne y_n \quad \text{(Inequality case)} \end{cases} \qquad (22)}$$

Note that it is clear that the expression on the right-hand side is an upper bound on $D(X,Y)$. So correctness amounts to showing that it is also a lower bound. We leave the correctness proof to an Exercise. The reader should compare this formula with the $L(X,Y)$ formula in (4): both formulas have the equality and inequality cases. But the "1+" term appear for different cases. In the inequality case, we have a term $D(X',Y')$ above, but no corresponding $L(X',Y')$ term in (4).

It follows that $D(X,Y)$ can be computed in $O(mn)$ time by the technique of filling in entries in a $(1+m)\times(1+n)$ matrix D (see ¶9).

We illustrate using our old example $X = \texttt{lengthen}$ and $Y = \texttt{elongate}$: consider the matrix $D[0..8, 0..8]$ in Table 3. The rows (resp., columns) are labeled by the successive prefixes of $X = x_1,\ldots,x_m$ (resp., $Y = y_1,\ldots,y_n$). Thus the zero-th row is labeled by $X_0 = \epsilon$ (empty string), the first row by $X_1 = x_1$, second row by $X_2 = x_1x_2$, etc. Similarly the columns are labeled by the prefixes of Y.

We fill in the entries of the matrix D in a row-by-row (or column-by-column) fashion. The three cases of (22) are illustrated in Table 3(a):

- The base case of (22) is used to fill in the zero-th row and zero-th column. The zero-th row has entries $D[0,j] = j$ $(j = 0,1,\ldots,|Y|)$. Similarly the zero-th column has entries $D[i,0] = i$ $(i = 0,1,,\ldots,|X|)$.
- The equality case of (22) is used to fill the (i,j)-th entry when $x_i = y_j$. In this case $D[i,j] \leftarrow D[i-1,j-1]$. In Table 3, the $(3,4)$-entry is an equality case since $x_3 = y_4 = \texttt{n}$. Thus, $D[3,4] \leftarrow D[2,3] = z$.

		0	1 e	2 l	3 o	4 n	5 g	6 a	7 t	8 e
0		0	1	2	3	4	5	6	7	8
1	l	1								
2	e	2			z					
3	n	3				z				
4	g	4								
5	t	5								
6	h	6								
7	e	7							v	u
8	n	8							w	$1+\min(u,v,w)$

(a) Mechanics of filling in the matrix

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

(b) Filled matrix

Table 3: Matrix for computing $D(\texttt{lengthen}, \texttt{elongate})$

- The inequality case is used for the (i,j)-th entry when $x_i \neq y_j$. Then $D[i,j] \leftarrow 1 + \min\{D[i-1,j], D[i-1,j-1], D[i,j-1]\}$. In Table 3, the $(8,8)$-entry is an inequality case, and so $D[8,8] \leftarrow 1 + \min\{u,v,w\}$ as shown.

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 (22) 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.

¶23. Connection to LCS Problem. We had alluded to a connection between $L(X,Y)$ and $D(X,Y)$. They represent two opposing approaches to string similarity: maximizing $L(X,Y)$ versus minimizing $D(X,Y)$. We have these inequalities:

Lemma 2 *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 $m \ge n$. Then for each $n/2 \le \ell \le n$, 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 $0 \le \ell \le m$, there are strings X, Y such that $D(X,Y) = m - \ell$. E.g., $X = a^{m-\ell}b^{\ell}$ and $Y = b^{\ell}$.

¶24. General cost function Δ. 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 type (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. But if $x = *$ or $y = *$, we interpret $\Delta(*,y)$ as the cost of inserting y, and $\Delta(x,*)$ as the cost of deleting x. For the time being, we assume $\Delta(x,y)$ is always non-negative. Let $A_\Delta(X,Y)$, or simply $A(X,Y)$, if Δ is understood, be *tentatively* defined as the minimal cost of a sequence of edit operations to convert X to Y. We call $A_\Delta(X,Y)$ the **alignment cost** between strings X, Y. Note that we no longer call A_Δ a "distance" function because when we give the definitive definition, it may no longer be a metric (the triangular inequality will fail).

Tentative definition of alignment cost $A(X,Y)$ as edit distance

For example, consider the alignment cost function

$$\Delta(x,y) = \begin{cases} 2 & \text{if } \; x = * \text{ or } y = * \\ 0 & \text{if } \; x = y \\ 1 & \text{else.} \end{cases} \tag{23}$$

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 $X = \texttt{bulk}$ and $Y = \texttt{ucky}$. In the uniform cost model, we have $D(X,Y) = 3$, obtained by the following sequence of delete, insert, replace operations:

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

With the cost model of (23), 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:

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

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

The "alignment" terminology 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

Using the cost function (23) above, the cost of the alignment (X_*, Y_*) is

$$\Delta(\texttt{A},*) + \Delta(\texttt{C},\texttt{C}) + \Delta(*,\texttt{A}) + \Delta(\texttt{T},\texttt{T}) = 2+0+2+0 = 4.$$

Alternatively, we could simply align (X, Y) with cost

$X:$	A	C	T
$Y:$	C	A	T

with cost

$$\Delta(\texttt{A},\texttt{C}) + \Delta(\texttt{C},\texttt{A}) + \Delta(\texttt{T},\texttt{T}) = 1+1+0 = 2.$$

The goal is to alignment is find the minimal cost alignment. We will shortly convert from the editing model to the alignment model.

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:

- It appears that in genomics, mother nature is more likely to replace `A` by `T` than to replace `A` by `C`. This can be modeled using a cost function where $\Delta(\texttt{A},\texttt{C}) > \Delta(\texttt{A},\texttt{T}) > 0$.
- Consider the string edit problem over the alphabet $\{$`a,b,c, ...,x,y,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 $\Delta(\texttt{a},\texttt{b}) > \Delta(\texttt{v},\texttt{b})$.

¶25. What is missing in the editing model? So far, we have only looked at non-negative costs. But something new arises if we allow negative costs. To focus on this, let us introduce a simple class of cost functions. In general, the cost function Δ requires specifying almost $(|\Sigma| + 1)^2$ numbers. For most of our illustrations, we prefer the following **simplified cost model** given by:

$$\Delta(x, y) = \begin{cases} \delta_= & \text{if } \ x = y, \\ \delta_{\neq} & \text{if } \ x \neq y \\ \delta_* & \text{if } \ x = * \ \text{ or } \ y = *. \end{cases} \tag{24}$$

subject to

$$-\delta_* < \delta_= \leq 0 < \delta_{\neq} < \delta_* \tag{25}$$

For example, (23) is an instance of this model with $\delta_= = 0, \delta_{\not=} = 1, \delta_* = 2$. So our simplified cost model is specified by three numerical parameters. The value δ_* is called the **gap penalty**. The inequalities of (25) are 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:

$$\delta_= = -2, \quad \delta_{\neq} = 1, \quad \delta_* = 3. \tag{26}$$

Let us show that the triangular inequality (cf. (19)) fails under the simplified cost function (26). Let $X = \texttt{a}$, $Y = \texttt{aa}$, and $Z = \texttt{aaa}$. Under the alignment cost specified by (26), we have $A(\texttt{a},\texttt{aa}) = 3 - 2 = 1$, $A(\texttt{aa},\texttt{aaa}) = 3 - 4 = -1$, and $A(\texttt{a},\texttt{aaa}) = 6 - 2 = 4$. Thus

These claims are intuitive – it cannot be captured under the editing model.

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

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

¶26. Motivation for negative costs. We have motivated the need for a cost model based on the letters operated upon. But why do we need negative costs? Using our simplified cost function (24), 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:

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

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

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

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

¶27. 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:

$$\boxed{A(X,Y) = \begin{cases} (m+n)\delta_* & \text{if } mn = 0, \\ \min\{A(X',Y') + \Delta(x_m, y_n), & \\ \quad A(X',Y) + \Delta(x_m, *), & \\ \quad A(X,Y') + \Delta(*, y_n)\} & \text{else.} \end{cases} \qquad (27)}$$

Note that for simplicity, (27) 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 (27). The remaining entries of M is filled in a row by row fashion, using the general case of (27). The desired value $A(X, Y)$ is found in the $(m+1, n+1)$-entry of M.

Example. Assume that Δ in (24) is given by

$$\delta_= = -1, \quad \delta_{\neq} = 1, \quad \delta_* = 2. \qquad (28)$$

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

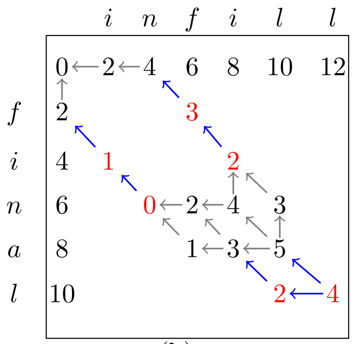
$$M = \begin{array}{c|cccccc} & \varepsilon & \texttt{A} & \texttt{A} & \texttt{T} & \texttt{T} & \texttt{C} \\ \hline \varepsilon & 0 & 2 & 4 & 6 & 8 & 10 \\ \texttt{G} & 2 & 1 & 3 & 5 & 7 & 9 \\ \texttt{C} & 4 & 3 & 2 & 4 & 6 & 6 \\ \texttt{A} & 6 & 3 & 2 & 3 & 5 & 7 \\ \texttt{T} & 8 & 5 & 4 & 1 & 2 & 4 \end{array}$$

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

Here is another example using the cost function (28), but with $X = \texttt{final}$ and $Y = \texttt{infill}$. The matrix is given in Table 4(a).

		i	n	f	i	l	l
	0	2	4	6	8	10	12
f	2	1	3	3	5	7	9
i	4	1	2	4	2	4	6
n	6	3	0	2	4	3	5
a	8	5	2	1	3	5	4
l	10	7	4	3	2	2	4

(a)



(b)

Table 4: (a) Computing $A(\texttt{final}, \texttt{infill})$, (b) optimal alignments

What are the optimal alignments with cost 4? In Table 4(b), we draw arrows connecting entries that are produced by the optimal paths starting from the bottom-most right-most entry (viz., [5,6]) to the origin [0,0].

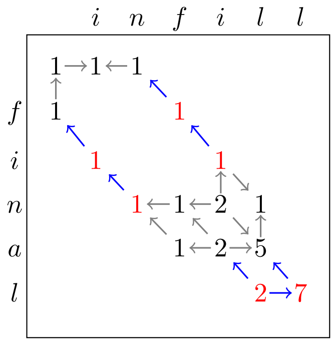


Table 5: Counting number of optimal alignments for $X = \texttt{final}$ and $Y = \texttt{infill}$

[NOTE: the arrows have messed up directions!!]

Suppose we want to count the number of optimal alignments. Begin with the optimal alignment paths in Table 4(b). We reverse the direction of all the edges so that they are now directed from left to right, and from top to bottom. Associate a **count** with each node on these paths, as shown in Table 5 using these rules. Start with a count of 1 for the origin at the top-left corner. The edge carries the count of its source node. The count of a target node is the sum of all the incoming counts. Using these rules, we finally obtain a count of 7 for the bottom-most right-most node. This means there are 7 optimal alignments, listed below. Two optimal alignments have matches for the substring `fil`; the other five have matches for the substring `inl`:

$$\begin{bmatrix} i & n & \underline{f} & \underline{i} & l & * & \underline{l} \\ * & * & \underline{f} & \underline{i} & n & a & \underline{l} \end{bmatrix}, \quad \begin{bmatrix} i & n & \underline{f} & \underline{i} & * & l & \underline{l} \\ * & * & \underline{f} & \underline{i} & n & a & \underline{l} \end{bmatrix},$$

$$\begin{bmatrix} * & \underline{i} & \underline{n} & f & i & \underline{l} & l \\ f & \underline{i} & \underline{n} & a & * & \underline{l} & * \end{bmatrix}, \quad \begin{bmatrix} * & \underline{i} & \underline{n} & f & i & \underline{l} & l \\ f & \underline{i} & \underline{n} & * & a & \underline{l} & * \end{bmatrix},$$

$$\begin{bmatrix} * & \underline{i} & \underline{n} & f & i & l & \underline{l} \\ f & \underline{i} & \underline{n} & a & * & * & \underline{l} \end{bmatrix}, \quad \begin{bmatrix} * & \underline{i} & \underline{n} & f & i & l & \underline{l} \\ f & \underline{i} & \underline{n} & * & * & a & \underline{l} \end{bmatrix},$$

$$\begin{bmatrix} * & \underline{i} & \underline{n} & f & i & l & \underline{l} \\ f & \underline{i} & \underline{n} & * & a & * & \underline{l} \end{bmatrix}.$$

¶28. Model of Alignment. The above algorithm for computing $A(X, Y)$ using (27) 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.

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 was 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

$$\begin{array}{c|cccc} \Delta & \mathtt{A} & \mathtt{G} & \mathtt{C} & \mathtt{T} \\ \hline \mathtt{A} & -3 & 2 & 3 & 1 \\ \mathtt{G} & 2 & -3 & 2 & 1 \\ \mathtt{C} & 3 & 2 & -3 & 1 \\ \mathtt{T} & 1 & 1 & 1 & -3 \end{array} . \tag{29}$$

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

Recall that we previously interpreted $D(X, Y)$ as a minimum cost path problem in the infinite graph $G(\Sigma)$ defined in ¶22. 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 $\Delta(\mathtt{a}, \mathtt{a})$ 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 $\Delta(\mathtt{a}, \mathtt{a})$. Then we must conclude that $A(X, X) = -\infty$ since we can replace $\mathtt{a}$ 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 insert 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_* when we place X_* directly 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 $\ell = |X_*| = |Y_*|$. Of course, this "replacement" here 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 $\Delta_*(X, Y)$:

$$\Delta_*(X, Y) := \min_{(X_*, Y_*)} \Delta(X_*, Y_*) \tag{30}$$

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 (30), an optimal alignment must satisfy $X_*[i] \neq *$ or $Y_*[i] \neq *$ for each i. Thus, we have $|X_*| = |Y_*| \leq |X| + |Y|$.

E.g., Let X = `AATTC` and Y = `GCAT`, as in a previous example. If X_* = `AATTC` and Y_* = `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.

¶29. Correctness of the Dynamic Programming Solution. We formally defined the alignment cost to be $\Delta_*(X, Y)$ in (30). In ¶27, we described a dynamic programming algorithm to compute a quantity that we denote by $A_\Delta(X, Y)$. The correctness of the 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.

One of issues of formalizing the editing model is the treatment of replacement, in particular the treatment of **self-replacement** in case $\Delta(x, x) < 0$. Surely we must disallow applying this self-replacement more than once. In our recursive definition of $A_\Delta(X, Y)$, this property is in-built. But if we apply a sequence of editing operations, how do we detect this illegitimate use of self-replacement?

The key idea is the following concept: say a sequence σ of edit operations is **canonical** if that each insertion in σ occurs before any replacement, each replacement in σ occurs before any deletion. For any string X, there is a well-defined **cost** $COST_\Delta(\sigma, X)$, which is just sum of the cost of each operation in σ as they are sequentially applied to X.

Lemma 3 *Given any sequence σ of edit operations that transforms a string X to another string Y, there exists a canonical sequence σ' such that*

(i) $COST_\Delta(\sigma, X) = COST_\Delta(\sigma', X)$
(ii) The number of inserts, replacements, and deletes in σ and σ' are the same.

Proof. Each operation in σ is an edit operation of the type

$$Del(i, X), \quad Ins(i+1, y_j, X) \quad Rep(i, y_j, X).$$

Since X is understood, we may simply write $Del(i)$, $Ins(i+1, y_j)$ or $Rep(i, y_j)$. It is convenient to also let $\delta_\le(i,j)$ be the Kronecker Delta function where $\delta_\le(i,j) = 1$ iff $i \le j$ and $\delta_\le(i,j) = 0$ other wise. We convert σ into a canonical sequence by repeatedly application of the following transformations:

$$\begin{aligned}
Del(i); Ins(j,b) &\to \begin{cases} Ins(j+1,b); Del(i) & \text{if } i < j \\ Ins(j,b); Del(i+1) & \text{if } i \ge j \end{cases} \\
Del(i); Rep(j,b) &\to \begin{cases} Rep(j+1,b); Del(i) & \text{if } i \le j \\ Rep(j,b); Del(i) & \text{if } i > j \end{cases} \\
Rep(i,a); Ins(j,b) &\to \begin{cases} Ins(j,b); Rep(i,a) & \text{if } i < j \\ Ins(j,b); Rep(i+1,a) & \text{if } i \ge j \end{cases}
\end{aligned}$$

These can be written more compactly using the Kronecker δ-function:

$$\begin{aligned}
Del(i); Ins(j,b) &\to Ins(j + \delta(i<j), b); Del(i + \delta(i \ge j)) && (31) \\
Del(i); Rep(j,b) &\to Rep(j + \delta(i \le j), b); Del(i) && (32) \\
Rep(i,a); Ins(j,b) &\to Ins(j,b); Rep(i + \delta(i \ge j), a). && (33)
\end{aligned}$$

Using (31), if a deletion is followed by an insertion in σ, we can replace these two operations by an insertion followed by a deletion. Similarly for the other two transformation rule. When no more transformations are possible, σ is normalized. The justification of (31)-(33) is an Exercise. We can also verify that the cost of each pair of operations (as given by $\Delta(x,y)$) is unchanged after our transformation. **Q.E.D.**

Henceforth, the canonical form of σ is the sequence described in the proof of the preceding lemma. We say a sequence σ of edit operations is **non-redundant** for a string X if, in the canonical form σ produced by the above lemma, all the replacement operations are distinct. To see this, suppose σ' is the canonical form of σ, and the operation $Rep(i,a)$ appears twice in σ'. It is clear that we can remove such redundancy. If σ (and therefore σ') has optimal cost, such redundancy could only come the fact that $Rep(i,a)$ is a self-replacement with negative cost. To avoid such redundancy, we must restrict our edit sequences to non-redundant ones.

We say Δ is **well-founded** if $\Delta_*(X,Y)$ is finite for all X, Y.

Theorem 4 (Correctness)
If Δ is well-founded then the recursive definition of $A_\Delta(X,Y)$ yields $\Delta_*(X,Y)$.

Proof. Assume that we have a sequence σ of edit operations that transforms X to Y. By the above lemma, we may assume that σ is canonical. We can split σ into $\sigma_I; \sigma_R; \sigma_D$ correspond to the insertions, followed by replacements, followed by deletions. Suppose σ_I transforms X to X_I, and σ_R transforms X_I to X_R. Clearly, σ_D transforms X_R to Y. We now define an alignment (X_*, Y_*) of X, Y as follows: We apply the insertions of σ_I to transform X into X_* as follows: instead of inserting a character, we simply insert $*$. We apply the deletions of σ_D to transform X_R to Y_* as follows: instead of deleting a character, we simply replace it with $*$. Now it is clear that the cost of the operations σ is equal to $\Delta_*(X_*, Y_*)$. Conversely, given any alignment (X_*, Y_*), we can construct a normalized sequence of editing operations of the form $\sigma_I; \sigma_R; \sigma_D$ to transform X to Y with cost $\Delta_*(X_*, Y_*)$. **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.

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

$$\Delta(x,y) = \begin{cases} \delta_* & \text{if } \ x = * \ \text{ or } \ y = *, \\ \delta_= & \text{if } \ x = y, \\ \delta_1 & \text{if } \ x, y \text{ are both consonants or both vowels}, \\ \delta_2 & \text{else}. \end{cases} \tag{34}$$

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

$$\delta_* = 3, \quad \delta_= = 0, \quad \delta_1 = 1, \quad \delta_2 = 2, \tag{35}$$

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.

¶31. 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.

- In the introduction to this Lecture, we mentioned the "database problem" in string searching: given a string X, and a database B (i.e., a set of strings), we want to return a string $X' \in B$ such that $d(X, Y)$ is minimized where d is some distance measure, or $m(X, Y)$ is maximized where m is some match measure. For instance, $d(X, Y)$ can be edit distance $D(X, Y)$ and $m(X, Y)$ is $LCS(X, Y)$. The **database problem** is this: given B, preprocess it so that for any X, we can retrieve any (or the set of) $X' \in B$ that optimizes the similarity between X and X'.

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 (34) and and (35). For this purpose, assume `y` is a consonant. Also, express $\Delta(X, Y)$ as a direct alignment cost. ♢

Exercise 3.4: What $A(X,Y)$ where $X =$ `final` and $Y =$ `infill`? Assume the cost of deletion/insertion is 2, cost of a match is -1, cost of non-match is 1. How many alignments achieves this cost? Give a graph representation of these alignments. NOTE: we want two numbers and graph. ♢

Exercise 3.5: 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 3.6: 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 3.7: Prove (22). This is an instructive exercise. ♢

Exercise 3.8: Let x, y, z be distinct letters, and $0 \le m \le n$.
(a) Prove that $D(X,Y) = m + n - 2\ell$ where $m \ge \ell \ge m/2$, $X = x^{m-\ell}z^{\ell}$ and $Y = z^{\ell}y^{n-\ell}$.
(b) Let $X = x^{m-\ell}z^{\ell}$ and $Y = y^{n-\ell}z^{\ell}$ $(0 \le \ell \le n)$ Prove that $D(X,Y) = n - \ell$. ♢

Exercise 3.9: Let X, Y be strings. Clearly, $L(XX, YY) \ge 2L(X,Y)$.
(a) Give an example where the inequality is strict.
(b) Prove that $L(XX, Y) \le 2L(X,Y)$ and this is the best possible.
(c) Prove that $L(XX, YY) \le 3L(X,Y)$.
(d) We know from (a) and (c) that $L(XX, YY) = cL(X,Y)$ where $2 \le c \le 3$. Give sharper bounds for c. ♢

Exercise 3.10: 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 3.11: 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 3.12: 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 3.13: 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 3.14: 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 $T(X,Y) \ge 1 + \min\{T(X',Y), T(X,Y'), T(X',Y')\}$.
(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 3.15: 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. ♢

Exercise 3.16: We had proved that any sequence of edit operations can be turned into the $Ins-Rep-Del$ normal form. In showing that this is canonical, we looked at transposing pairs of operations of the form

$$Del; Ins, \quad Rep; Ins, \quad Del; Rep$$

(a) Please determine the effects of transposing the remaining three types of operations:

$$Ins; Del, \quad Ins; Rep, \quad Rep; Del$$

(b) Explore the other five other possible "normal forms"

- $Ins - Del - Rep$
- $Del - Ins - Rep$
- $Del - Rep - Ins$
- $Rep - Del - Ins$
- $Rep - Ins - Del$

♢

End Exercises

§4. Polygon Triangulation

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

¶32. Sawdust or Glue Minimization. You are a carpenter and need to saw a wooden board in the shape of an octagon into 6 triangles. See Figure 3 for two possible ways to do this. How can you do this in order to minimize the amount of sawdust? The amount of sawdust is proportional to the total length of your sawing. In Figure 3, do you see why the sawdust in (b) is less than the sawdust in (a)?

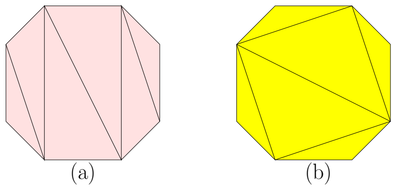
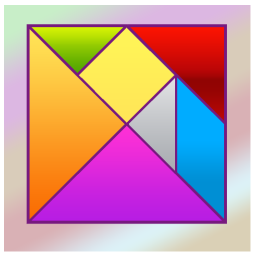


Figure 3: Two ways to triangular a regular octagon



OK, like a tangram

Equivalently, your factory wants to manufacture triangular pieces that must be assembled (like a jigsaw puzzle) into an octagon by gluing along edges. How should you design these triangles so as to minimize the amount of glue needed in reassembly? We shall see that the optimal solution can be found using dynamic programming.

The carpentry notion of a polygon P is a geometric one that 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. 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. We say P is **convex** if every chord v_iv_j is contained in P. Figure 4 shows a convex polygon with $n = 7$ vertices.

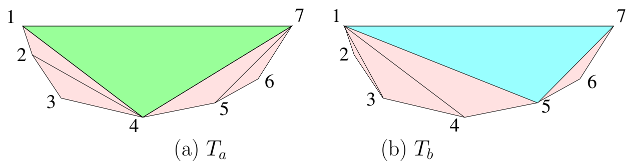


Figure 4: Two triangulations of the standard 7-gon

¶33. Abstract Polygons. We now give an abstract, purely combinatorial version of these terms. Let $P = (v_1, \ldots, v_n)$, $n \geq 2$, 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 an integer, so ordering in P can encoded by the relation $v_i < v_j$ on integers. In the simplest case, we identify v_i with the integer i. Then $P = (v_1, \ldots, v_n) = (1, \ldots, n)$ is called the **standard n-gon**.

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 segment can also be denoted by "i–j". We classify a segment i–j as an **edge** of P if $j = i + 1$ or $(i–j) = (1–n)$; otherwise the segment is called a **chord**. E.g., consider the standard 7-gon represented graphically in Figure 4. Two sets T_a, T_b of chords are shown:

$$T_a = \{1–4, 2–4, 4–7, 5–7\}, \quad T_b = \{1–3, 1–4, 1–5, 5–7\}. \tag{36}$$

You may check that an n-gon $(n \geq 3)$ has exactly n edges and $n(n-3)/2$ chords (Exercise).

Thus a 3-gon has 3 edges and no chords, but a 7-gon has 7 edges and 14 chords. We say two distinct segments $i–j$ and $i'–j'$ **intersect** if

$$i < i' < j < j' \qquad \text{or} \qquad i' < i < j' < j;$$

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

¶34. Triangulations. We define a **triangulation** of an n-gon to be a maximal set T of pairwise disjoint chords. For instance, the two sets T_a, T_b in (36) represent two different triangulations of the standard 7-gon. They are graphically shown in Figure 4. It is not hard to show by induction that any triangulation T of the n-gon has exactly $n-3$ chords. This formula breaks down for $n = 2$. Thus the empty set of chords is the unique triangulation of a 3-gon. It is also convenient to consider the empty set as the unique triangulation of the 2-gon.

A **triangle** of P is a triple (i, j, k) (or simply, ijk) where $1 \le i < j < k \le n$; its three **edges** are $i–j$, $j–k$ and $i–k$. 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_a in Figure 4 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 $1–n$. 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^a in Figure 4 has the partition

$$T^a = T_4 \uplus T'_4 \uplus S_4$$

where $S_4 = \{1–4, 4–7\}$, $T_4 = \{2–4\}$ and $T'_4 = \{5–7\}$. Note that $S_i = \{1–i, i–n\}$ for $i = 3, \ldots, n-2$, $S_2 = \{2–n\}$ 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.

¶35. 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. Here are two concepts of costs of a triangulation:

- The **total W-cost** of a triangulation T is the sum of the weights $W(i, j, k)$ of the triangles ijk belonging to T. Given W, the **Optimal Triangulation Problem** is to compute the minimum of the total W-cost of any triangulation.

- The **max W-cost** of a triangulation T is the maximum of the weights $W(i,j,k)$ of $ijk \in T$. Given W, the **Min-Max Triangulation Problem** is to compute the minimum of the max W-cost of any triangulation.

Notice that we have defined both problems in terms of computing just the minimum of the costs of triangulations. But we could have equally defined the problem in terms of computing the actual triangulations that achieves this minimum. Below, we will focus of the optimal triangulation problem which is based on total W-cost. The case of Min-Max cost seems harder.

¶36. Example: We introduced this section with the minimum sawdust problem. We had to cut up a convex polygonal board P into triangles. Here, $P = (v_1, \ldots, v_n)$ is a geometric convex polygon in the plane, and a natural cost function is the perimeter

$$W(i,j,k) := \|v_i - v_j\| + \|v_i - v_k\| + \|v_j - v_k\| \qquad (37)$$

of the triangle (v_i, v_j, v_k). Here, $\|\cdot\|$ denotes the Euclidean length. 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, carpenter's sawdust problem is just an optimal triangulation problem with the weight function (37).

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. *$\Theta(n)$ parameters*
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. *$\Theta(n^2)$ parameters*
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 §6. *$\Theta(n)$ parameters*
4. **Weight functions from differences of vertex weights:** W is defined by an increasing sequence $a_1 \le a_2 \le \cdots \le a_n$ and $W(i,j,k) = a_k - a_i$. Note that the index j is not used in $W(i,j,k)$. In §7, we will see an example (optimum search trees) of such a weight function. *$\Theta(n)$ parameters*

¶37. 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 $1 \le i < j \le n$. Then

$$C(i,j) = \begin{cases} 0 & \text{if } j = i+1, \\ \min_{i<k<j}\{C(i,k) + W(i,k,j) + C(k,j)\} & \text{else.} \end{cases} \quad (38)$$

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

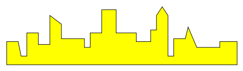
Exercise 4.1: Show that an n-gon ($n \ge 3$) has exactly $n(n-3)/2$ chords. ♢

Exercise 4.2: You are now a gardener with a convex polygonal garden. You want divide the garden into triangular plots, by introducing paths connecting the corners of the garden. But your goal now is to maximize the smallest area. Give a dynamic programming solution. ♢

Exercise 4.3: Find the minimum total W-cost of triangulations of the abstract hexagon whose weight function W is parametrized by $(a_1, \ldots, a_6) = (4, 1, 3, 2, 2, 3)$:
(a) Why is the weight function $W(i,j,k) = a_i + a_j + a_k$ not interesting?
(b) The weight function is given by $W(i,j,k) = a_i a_j a_k$.
(c) The weight function is given by $W(i,j,k) = |a_i - a_j| + |a_i - a_k| + |a_j - a_k|$.
(d) The weight function is given by $W(i,j,k) = a_i^2 + a_j^2 + a_k^2$. ♢

Exercise 4.4: Redo the previous exercise, but for the Min-Max Triangulation problem instead. ♢

Exercise 4.5: 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. ♢


Manhattan skyline (a non-simple polygon)

Exercise 4.6: In three dimensions, we may consider the problem of **optimal tetrahedralization** of a (geometric) polyhedron, i.e., subdivide it into a finite number of tetrahedra, subject to some optimization criterion. Note that the tetrahedrons are required to have vertices from the original polyhedron. Unfortunately, two problems arise:
(a) A convex polyhedron it can be decomposed into different numbers of tetrahedra. Show that a cube can be decomposed into 5 tetrahedra, and it can also be decomposed into 6 tetrahedra.
(b) Construct a non-convex polyhedron that cannot be decomposed into tetrahedra. ♢

Exercise 4.7: A more profound generalization of triangulation comes from considering the triangulation (tetrahedralization) of convex polytope in 3-dimensions. The number of tetrahedra is no longer unique (see previous Exercise). Give an abstract formulation of this problem. HINT: certain subsets of the vertices are said to be "convex". ♢

Exercise 4.8: (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 4.9: 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 4.10: (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 $1 \le i \le j \le k \le l \le n$ such that $|\{i,j,k,l\}| \ge 3$. 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

¶38. What is common? At this point, we have seen several problems whose solution is method is characterized as dynamic programming: $LCS(X,Y), D(X,Y)$ and optimal triangulation. Let us note the three ingredients found in these solutions.

- **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
$$(i, i+1, i+2, \ldots, j-1, j), \qquad (1 \le i < j \le n)$$
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.
- **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 $S_i \subseteq \{1\text{–}i, i\text{–}n\}$ and T_1, T_2 are triangulations of subpolygons of

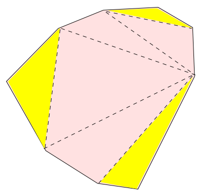
P. In fact, T_1, T_2 are optimal solutions to subproblems $P_{1,i}$ and $P_{i,n}$ for some $1 < i < n$. This property is called the **dynamic programming principle**, namely, an optimal solution to a problem induces optimal solutions on certain subproblems.

So this principle lends its name to this method. Or vice-versa?

- **The optimal solution of a problem can be easily constructed from the optimal solutions of subproblems.** We interpret "easily constructed" to be polynomial time. Thus, (38) allows us to compute the optimal triangulation of $P_{i,j}$ from the optimal triangulations for all smaller subproblems of $P_{i,j}$.

The reader may also verify these ingredients in the LCS and edit distance problems.

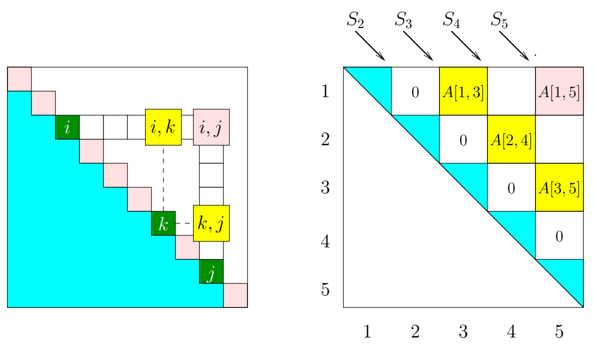
¶39. What Can Go Wrong? Here is an alternative solution to the optimal triangulation problem. Begin with the observation that every triangulation T must contain a triangle of the form $(i, i+1, i+2)$ (Exercise). Such a triangle is called an **ear**. Suppose we remove an ear from an n-gon. The result is an $(n-1)$-gon. If we knew an ear that appears in an optimum triangulation, we could recursively triangulate the remaining $(n-1)$-gon. Since we do not know which ear to remove, as usual, we try all possible ears. This gives rise to the following (initial) analogue of equation (38):


triangulation with 3 ears

$$C(i,j) = \begin{cases} 0 & \text{if } j = i+1, \\ \min_{i<k<j}\{W(k-1,k,k+1) + C(i,k-1,k+1,j)\} & \text{else.} \end{cases} \tag{39}$$

where $C(i, k-1, k+1, j)$ is the optimal solution for polygon $P(i, k-1, k+1, j)$ that remains after we remove the ear $(k-1, k, k+1)$. What is wrong with this approach? We realize that the number of parameters specifying the subproblems has increased from 2 to 4. If we recurse, the number of parameters will be $\Omega(n)$. Thus we are forced to look at an exponential number of subproblems. This demonstrates that equation (38) encodes some important properties that make it successful.

it deserves more respect!



(a) kth update for $A[i,j]$ (b) Entries for Stage 3

Figure 5: Filling in of a upper triangular matrix

¶40. Mechanics of the algorithm. To organize the computation embodied in equation (38), we use an upper triangular $n \times n$ matrix A to store the values of the optimal solution $C(i,j)$, $A[i,j] = C(i,j)$ for $1 \le i < j \le n$. This is illustrated in Figure 5.

We view the algorithm as a systematic filling in of the matrix A. Consider the problem of updating a single entry $A[i,j]$. According to (38), this can be achieved by a sequence of "updates". The k-th update corresponds to

$$A[i,j] \leftarrow \min\{A[i,j],\ A[i,k]+W(i,k,j)+A[k,j]\} \tag{40}$$

as illustrated in Figure 5(a). Here, k ranges from $i+1$ to $j-1$. We also need to initialize $A[i,j]$ to ∞ to allow the minimization to be meaningful for $k=i+1$.

The entry $A[i,j]$ represents a subproblem of **size** $(j-i+1)$. In order to do the update (40), the subproblems $A[i,k]$ and $A[j,k]$ need to be available. These subproblems have smaller size than $A[i,j]$. Therefore, we fill in the entries in order of increasing size. More precisely, in **stage** t, we solve all the subproblems of size t. The subproblems of size $t=2$ are trivial, $A[i,i+1]=0$ for all i. In Figure 5(a), the set of entries corresponding to subproblems of size t is denoted by S_t. For instance, $S_3=\{A[1,3],A[2,4],A[3,5]\}$ (these are colored yellow). In stage 3, we fill the entries in S_3. In stage 5, there is only one entry, $S_5=\{A[1,5]\}$.

Clearly, t ranges from 2 to n. There are exactly $n-t+1$ problems of size t. According to (38), to solve a problem of size t ($t\ge 2$) we need to minimize over a set of $t-2$ numbers, 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 (38):

```
  Dynamic Programming for Optimal Triangulation
▷ Initialize solutions to problems of size 2
    for $t \leftarrow 1$ to $n-1$
        $A[t,t+1] \leftarrow 0$
▷ Main Loop
    for $t \leftarrow 2$ to $n$     ◁ Do stage $t$
        for $i \leftarrow 1$ to $(n-t-1)$     ◁ There are $n-t+1$ subproblem of size $t$
            $A[i,i+t-1] \leftarrow +\infty$     ◁ Value to begin the minimization
            for $k \leftarrow i+1$ to $(i+t-2)$
                $A[i,i+t-1] \leftarrow \min\{A[i,i+t-1],\ A[i,k]+W(i,k,i+t-1)+A[k,i+t-1]\}$
```

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

Tensors. The reader would surely have guessed that we could generalize this scheme to objects that are more general than matrices. There is a class of mathematical objects called **tensors**. Each tensor has a rank, and matrices are just tensors of rank 2. The generalization of dynamic programming to tensors amounts to filling the entries of a rank k tensor in a systematic way. It is harder to visualize this process. But in terms of a computer program, this presents no extra difficulty: we would just have a $(k+1)$-ply nested for-loop.

¶41. 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 (38) 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. Using the slick "argmin" notation,

$$K[i,j] \leftarrow \operatorname*{argmin}_{i<k<j} \{A[i,k] + W(i,k,j) + A[k,j]\}$$

Clearly, 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.

¶42. 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 (38) 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 in the worst-case. But we can, by computing bounds, avoid certain branches of the recursion. This can have potential speedup – see Exercise.

¶43. Space-Efficient Solutions. We can often 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, as we saw in LCS, such space efficient solutions do not easily extend into solutions that could reconstruct the optimal solutions.

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: Prove the "ear lemma" in the text. In fact, you can prove the stronger lemma that there are at least two ears if $n \ge 4$ ◇

Exercise 5.2: Joe Quick observed that the recurrence (4) for $L(X,Y)$ would work just as well if we look at suffixes of X, Y (*i.e.*, by omitting prefixes). Joe concluded that we could double the speed of our algorithm if we work from *both ends* of our strings. That is, for $0 \le i < j$, let $X_{i,j}$ denote the substring $x_i x_{i+1} \cdots x_{j-1} x_j$. Similarly for $Y_{k,\ell}$ where $0 \le k < \ell$. 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 5.3: 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. $\diamondsuit$

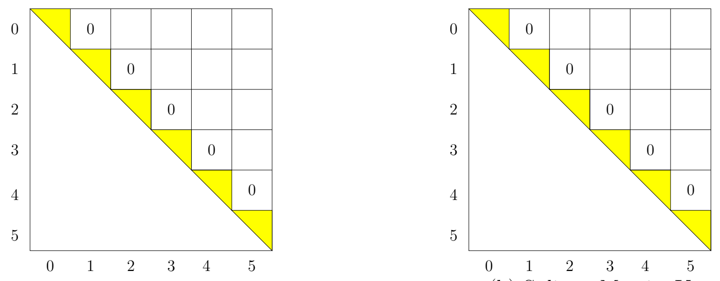
Exercise 5.4: Consider the linear bin packing problem where the ith item is not a single weight, but a pair of non-negative weights, (v_i, w_i). If we put the ith to jth items into a bin, then we require $\sum_{k=i}^{j} v_k$ and $\sum_{k=i}^{j} w_k$ to be each bounded by M. Again the goal to use the minimum number of bins. $\diamondsuit$

Exercise 5.5: Consider the optimal triangulation of the abstract hexagon using the weight function $W(i,j,k) = a_i^2 + a_j^2 + a_k^2$ where $(a_1, \ldots, a_6) = (4, 1, 3, 2, 2, 1)$.
(a) What is the optimal cost?
(b) What is the optimal triangulation?
Please show your working by filling in the following matrix. Note that the diagonal $A[i,i]$ contains a_i^2. This is useful when filling out the entries.

	1	2	3	4	5	6
1	16	0				
2		1	0			
3			9	0		
4				4	0	
5					4	0
6						1

$\diamondsuit$

Exercise 5.6: 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 6. Then write the optimal order of multiplying $A_1, \ldots, A_5$.



(a) Optimum Cost Matrix A (b) Splitter Matrix K

Figure 6: (a) $A[i,j]$ is optimal cost to multiply $A_i \times \cdots \times A_j$. (b) $K[i,j]$ indicates the optimal split.

$\diamondsuit$

Exercise 5.7: (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 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). $\diamondsuit$

Exercise 5.8: The following problem is motivated by 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. $\diamondsuit$

Exercise 5.9: (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. $\diamondsuit$

Exercise 5.10: Generalize the previous exercise (part (a)–(c) only). Let the set of real constants $\{a_i : i = -N, -N + 1, \ldots, -1, 0, 1, \ldots, N\}$ be fixed. Suppose that

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

$\diamondsuit$

Exercise 5.11: 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: Count the number of multiplications. Then argue separately that the number of additions is of the same order. $\diamondsuit$

Exercise 5.12: (Hypercube vertex selection) A **hypercube** or n-**cube** is the set $H_n = \{0,1\}^n$. Each $x = (x_1, \ldots, x_n) \in H_n$ is called a vertex of the hypercube. Let $\pi = (\pi_1, \ldots, \pi_n)$ and $\rho = (\rho_1, \ldots, \rho_n)$ be two positive integer vectors. The **price** and **reliability** of a vertex x is given by $\pi(x) = \sum_{i=1}^n x_i \pi_i$ and $\rho(x) = \prod_{i=1; x_i=1}^n \rho_i$. The **hypercube vertex selection problem** is this: given π, ρ and a positive bound B_0, find $x \in H_n$ which maximizes $\rho(x)$ subject to $\pi(x) \le B_0$. Solve this problem in time $O(nB_0)$ (not $O(n \log B_0)$).
HINT: View $H_n = H_k \otimes H_{n-k}$ for any $k = 1, \ldots, n-1$ and $y \otimes z$ denotes concatenation of vectors $y \in H_k$, $z \in H_{n-k}$. Solve subproblems on H_k and H_{n-k} with varying values of B ($B = 1, 2, \ldots, B_0$). The choice of k is arbitrary, but what is the best choice of k? $\diamondsuit$

Exercise 5.13: 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: $p \le q$ iff $p.x \le q.x$ and $p.y \le q.y$. If $p \ne q$ and $p \le q$, 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 $\{p \in S : p.x \ge c\}$. E.g., let $S = \{p(1,3), q(2,1), r(3,4), s(4,2)\}$ as shown in Figure 7. Then $\min(S(c))$ is equal to $\{p, q\}$

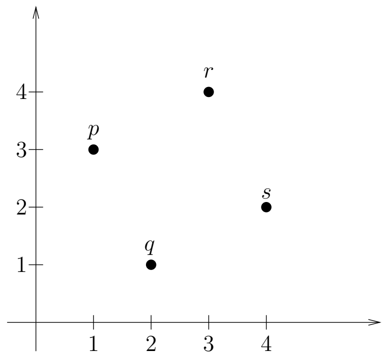


Figure 7: Set of 4 points.

if $c \le 1$; $\{q\}$ if $1 < c \le 2$; $\{r, s\}$ if $2 < c \le 3$; $\{s\}$ 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
$$O(\log n + k)$$
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 $\{p \in S : p.x \ge q.x, p.y \ge q.y\}$. 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. $\diamondsuit$

Exercise 5.14: (The Paragraphing Problem) 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 where each word occupies a given width of space. So a paragraph X of n words can be represented by a sequence $X = (x_1, \ldots, x_n)$ where $x_i > 0$ is space needed to typeset the ith word. The **Paragraphing Problem** is to break a given paragraph into lines, no line having width more than some given $M > 0$. Between 2 words in a line we introduce one space (of unit width); there is no space after the last word in a line. A line with the jth to kth words will have width $W_{j,k} = \sum_{i=j}^{k} x_i + k - j + 1$, and we require $W_{j,k} \le M$. For instance, $X = (4, 2, 3, 1, 6, 3)$ and $M = 7$ then we can break up the paragraph into 4 lines as follows: $(4, 2; 3, 1; 6; 3)$. Here comes the optimization aspect of our problem: call $M - W_{j,k}$ the **gap** of the above line, and we assess a **penalty** of $f(M - W_{j,k})$ on the line where f is a given "penalty function". We consider two such functions: linear $f(x) = x$, and quadratic $f(x) = x^2$. Note that $f(0) = 0$ in these cases (there is no penalty for zero gaps). The **total penalty** for a particular breakup of a paragraph is the sum of the penalty over all lines. The last line of a paragraph, by definition, suffers no penalty regardless of its gap. For instance, the total penalty of the breakup $(4, 2; 3, 1; 6; 3)$ under the linear and quadratic penalty functions are $1 + 3 + 1 = 5$ and $1 + 16 + 1 = 18$, respectively. Our goal is to find a breakup of the paragraph which minimizes the total penalty.
(a) Consider the obvious greedy method (cf. Lect.V.1) to solve this problem (fill in each line until the next word will cause an overflow, $W > M$). Show that under the linear penalty function, the greedy method is optimal.
(b) Show that under the quadratic penalty, the greedy algorithm is suboptimal.
(c) Give a dynamic programming solution to finding the optimal solution under the quadratic penalty function. Analyze its complexity. HINT: it seems easiest to first solve the
(d) Implement your solution of (c) in a programming language of your choice, and run it on Lincoln's Gettysburg address (Lecture V), assuming that each word has width equal to the number of characters, and $M = 80$. In the case of a terminal word (which is followed by a full-stop), we consider the full stop as part of the word. $\diamondsuit$

Exercise 5.15: (Refined Paragraphing Problem) Let us refine the Paragraphing Problem of the previous exercise. The previous problem has no concept of a sentence. Let us assume that some words end in a full-stop, and such words represent the end of a sentence. Moreover, we introduce the rule of introducing two spaces to separate the last word of sentence from the first word of the next sentence. In a bygone era of typewriters, this is called the *English Rule*.

Typing only one space is called the "French Rule", of course.

(a) Is the greedy method still optimal under the linear penalty function?
(b) Give a dynamic programming solution for an arbitrary penalty function.
(c) Now introduce optional hyphenation into the words. For simplicity, assume that every word has zero or one potential place for hyphenation (the algorithm has this information for each word). 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 $\ell_1 \ge 2$ and $\ell_2 \ge 1$. 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. Please modify the algorithm in (b) to handle hyphenation. $\diamondsuit$

Exercise 5.16: (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 $\sum_{i \in I} w_i \le W$.
(a) Give a dynamic programming solution that runs in time $O(nW)$.
(b) Improve the running time to $O(n, \min\{W, 2^n\})$. ♢

End Exercises

§6. Optimal Parenthesization

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

Let $(e_1, e_2, \ldots, e_n)$, $n \ge 1$, 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. Thus, there are exactly two parenthesized expressions on $(1, 2, 3)$, namely, $((12)3)$, $(1(23))$. The 5 parenthesized expressions on (e_1, e_2, e_3, e_4) are given by:

$$e_1(e_2(e_3e_4)), \quad e_1((e_2e_3)e_4), \quad (e_1e_2)(e_3e_4), \quad ((e_1e_2)e_3)e_4, \quad (e_1(e_2e_3))e_4. \tag{41}$$

A parenthesized expression on $(e_1, \ldots, e_n)$ can be viewed as a **parenthesis tree** on $(e_1, \ldots, e_n)$. Such a tree is a full[5] binary tree T on n leaves where e_i is the ith leaf in symmetric order. 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 $i = 1, \ldots, n$.

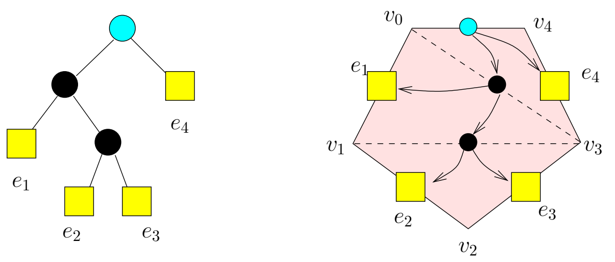


Figure 8: The parenthesis tree and triangulation corresponding to $((e_1(e_2e_3))e_4)$.

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 8 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

[5] A binary tree is **full** if every internal node is full, i.e., has two children. Alternatively, this is just an external binary tree.

this correspondence, but the "distinguished edge" (v_0, v_n) is not mapped. We leave the details for an exercise.

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

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

$$\pi = (\pi_1, \ldots, \pi_{n-1}) \tag{42}$$

of $\{1, 2, \ldots, n-1\}$. Before explaining this in general, we illustrate this encoding for the five parenthesized expressions on (e_1, e_2, e_3, e_4) as shown in (41). The corresponding permutations are given by

$$(1,2,3), \qquad (1,3,2), \qquad (2,1,3), \qquad (3,2,1), \qquad (3,1,2).$$

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 $\pi = (1,2)$ or $\pi = (2,1)$.

We now explain how the permutation (42) 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[6] a parenthesis tree on $A_{1+\pi_1}, \ldots, A_n$. Thus we have demonstrated:

Lemma 5 *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 (42) 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.

¶46. 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**, after the Belgian mathematician Eugéne Charles Catalan (1814–1894). The index $n-1$ of the Catalan numbers is the number of pairs of parenthesis needed to parenthesize n symbols. Equivalently, $n-1$ is the number of internal nodes of the parenthesis tree on n external nodes. The first few numbers in this sequence is given by the following table:

Catalan numbers were first described by Minggatu (1692–1763), Mongolian astronomer in the Qing court in China

n	0	1	2	3	4	5	6	7	8	9	10	11
$C(n)$	1	1	2	5	14	42	132	429	1430	4862	16796	58786

[6]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 $A_1, \ldots, A_{n-\pi_1}$.*

Note that $C(0) = 1$, not 0. From the injection of Lemma 5, we conclude that $P(n) = C(n-1) \le (n-1)!$. Our current goal is to give a more precise census of parenthesis trees. There are many interesting interpretations of $C(n)$. For computer science, the most natural interpretation is that *$C(n)$ is the number of binary trees with exactly n nodes.* This immediately gives the recurrence: for $n \ge 1$,

$$C(n) = \sum_{i=1}^{n} C(i-1)C(n-i) \tag{43}$$

in which index i selects the ith node as root, and so recursively, there are $C(i-1)$ (resp., $C(n-i)$) possible left (resp., right) subtrees. For instance, we compute $C(1) = C(0)C(0) = 1$ and $C(2) = C(0)C(1) + C(1)C(0) = 2$, $C(3) = C(0)C(2) + C(1)C(1) + C(2)C(0) = 5$. This recurrence has an elegant solution using generating functions (§VIII.11),

$$C(n) = \frac{1}{n+1}\binom{2n}{n}. \tag{44}$$

To show this, let $G(X) = \sum_{n\ge 0} C(n)X^n$ be the generating function for Catalan numbers. Then notice that

$$G(X)^2 = \left(\sum_{n\ge 0} C(n)X^n\right)\left(\sum_{n\ge 0} C(n)X^n\right) = \sum_{n\ge 0} X^n \sum_{i=0}^{n} C(i)C(n-i) = \sum_{n\ge 0} X^n C(n+1)$$

Therefore $G(X) = 1 + X \cdot G(X)^2$. Solving this quadratic equation,

$$G(X) = \frac{1 \pm \sqrt{1-4X}}{2X}.$$

Which of the two solutions $G_+(X) := \frac{1+\sqrt{1-4X}}{2X}$ or $G_-(X) := \frac{1-\sqrt{1-4X}}{2X}$ is the correct one? To answer this question, we use the binomial theorem (§II.5) to expand $(1+x)^{1/2}$ into $\sum_{n\ge 0}\binom{1/2}{n}x^n = 1 + \frac{1}{2}x + \frac{1}{2}(\frac{1}{2}-1)\frac{x^2}{2!} + \frac{1}{2}(\frac{1}{2}-1)(\frac{1}{2}-2)\frac{x^3}{3!} + \cdots$. For $n \ge 1$, the nth term of this expansion is

$$\begin{aligned}
\frac{x^n}{n!}\left(\tfrac{1}{2}\right)\left(\tfrac{1}{2}-1\right)\left(\tfrac{1}{2}-2\right)\cdots\left(\tfrac{1}{2}-n+1\right) &= \frac{x^n}{2^n n!}(1)(-1)(-3)\cdots(-2n+3)\\
&= -\frac{(-x)^n}{2^n n!}(1)(3)(5)\cdots(2n-3)\\
&= -\frac{(-x)^n}{2^n n!}\frac{(2n-2)!}{(2)(4)(6)\cdots(2n-2)}\\
&= -\frac{(-x)^n}{2^n n!}\frac{(2n-2)!}{2^{n-1}(n-1)!}\\
&= -2\frac{(-x)^n}{4^n n}\frac{(2n-2)!}{(n-1)!(n-1)!}\\
&= -2\frac{(-x)^n}{4^n n}\binom{2n-2}{n-1}.
\end{aligned}$$

The substitution $x \mapsto -4X$ into $(1+x)^{1/2}$ yields

$$\sqrt{1-4X} = 1 - 2\sum_{n\ge 1}\frac{(4X)^n}{4^n n}\binom{2n-2}{n-1} = 1 - 2\sum_{n\ge 1}\frac{X^n}{n}\binom{2n-2}{n-1}.$$

It follows that all but the first term of $G_+(X)$ would be negative, and so it cannot[7] be the generating function. On the other hand

$$G_-(X) = \frac{1-\sqrt{1-4X}}{2X} = \frac{2\sum_{n\ge 1}\frac{X^n}{n}\binom{2n-2}{n-1}}{2X} = \sum_{n\ge 0}\frac{X^n}{n+1}\binom{2n}{n}.$$

[7]The first term is $1/X$, so $G_+(X)$ is not even a power series. It has a pole at $X = 0$.

This proves our formula (44) for $C(n)$.

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.

¶47. On Matrix Chain Products. 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,

$$((A_1 A_2) A_3), \quad (A_1 (A_2 A_3)) \tag{45}$$

are the two ways of multiplying 3 matrices. Let $T(p, q, r)$ be the cost 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 (45) 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 (45) 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 a 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 ¶30.

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.

¶48. Near Optimal Solutions. For the product weight function, $W(a_i, a_j, a_k) = a_i a_j a_k$, the optimal triangulation problem can be solved in $O(n \log n)$ time, using a sophisticated algorithm due to Hu and Shing [6]. Ramanan [9] gave an exposition of this algorithm, and presented an $\Omega(n \log n)$ lower bound in an algebraic decision tree. But a simpler solution is possible if we only need to approximate the optimal solution Chandra[8] 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 (42), this corresponds to the following parenthesis tree on $A_1, \ldots, A_n$:

$$(\cdots((A_1A_2)A_3)\cdots)A_n. \tag{46}$$

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

$$(A_1 \cdots (A_{i_0-2}(A_{i_0-1}A_{i_0}))\cdots)(\cdots(A_{i_0+1}A_{i_0+2})\cdots A_n). \tag{47}$$

Then the cost of this computation is at most $2T_{opt}(a_0, \ldots, a_n)$.

Exercises

Exercise 6.1: Show that $C(n)$ is the number of binary trees on n nodes. HINT: Use the recurrence (43) 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: (Chandra)
(i) Show that the method (46) 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 (47) has cost at most $2T_{opt}$. ♢

Exercise 6.4: (i) Consider an abstract n-gon whose weight function is a product function, $W(i,j,k) = w_iw_jw_k$ 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

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

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? ♢

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

End Exercises

§7. Optimal Binary Search Trees

¶49. Probabilistic Model of Searching. We want to store n keys

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

in a binary search tree. The probability that key K to be searched is equal to K_i is p_i, and the probability that K falls between K_j and K_{j+1} is q_j. Thus the q's represent failed search probabilities. As illustrated in Figure 9, the goal is to construct an extended BST T (see §III.V) on n nodes (and $n+1$ nil-nodes) with the keys K_i's store in the nodes in sorting order, and the $2n+1$ nodes labeled with the probabilities

$$q_0, p_1, q_1, p_2, \ldots, q_{n-1}, p_n, q_n \tag{48}$$

in symmetric order. Note that the p_i's label the (regular) nodes and q_j's label the nil nodes. We want a BST that is "optimal" with respect to the given probabilities. Intuitively, our optimality criterion will cause a key K_i with a relatively high probability p_i to have a relatively smaller depth. E.g., if the BST in Figure 9 were optimal, this suggests that $q_5 + p_5 + p_6$ is relatively large compared to $q_2 + p_3 + q_3$. Thus, an optimal BST might not be very balanced in the usual sense of BST. Note that the optimal tree is a static BST, in contrast to our usual concerns with dynamic BST.

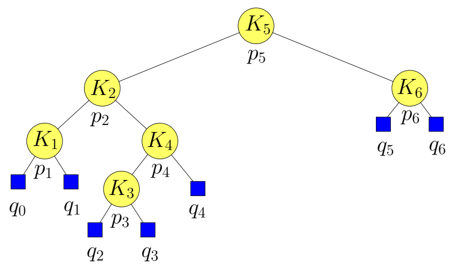


Figure 9: Optimal Binary Search Tree

¶50. Application. In constructing compilers for programming languages, we need a search structure for looking up if a given identifier K is a "key word", i.e., reserved words that have special meaning in the language. For instance, the `C` programming language has the following set[9] of key words

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

Let $K_1, \ldots, K_n$ be the key words of our programming language. By analyzing a large collection of sample programs, we can estimate about the probability p_i that an identifier K in a typical

[9]The 32 key words here is based on the so-called C90 version of the language, an ISO standard. The latest version has more key words.

program of this collection is equal to K_i; similarly, we can estimate the probability q_j that K lies between K_j and K_{j+1}. One solution to this compiler problem is to construct an optimal search tree for the key words with these probabilities. You might question how to select these sample programs. But we can imagine reasonable scenarios where we could get good samples. For instance, suppose Google has to periodically recompile a suite of software, because of bug fixes and the like. Then statistics can be gathered from this suite of programs are likely to be quite stable over small time frames. If we had smart compilers that allows us to change its lookup key-word lookup routines, then we could to speed up the re-compilation.

¶51. Optimality. We make precise our notion of optimality. Each node of the BST represents a subtree $T_{i,j}$ that is a BST on a sequence $K_i, K_{i+1}, \ldots, K_j$ of keys (for some $1 \le i \le j \le n$). For instance, in Figure 9, the node containing K_2 represents the subtree $T_{1,4}$. The root of the tree on n nodes represents $T_{1,n}$. Define the **weight function** W where

$$W(i,j) := \begin{cases} q_i & \text{if } i = j, \\ q_i + p_{i+1} + q_{i+1} + \cdots p_j + q_j = q_{i-1} + \sum_{k=i}^{j}(p_k + q_k) & \text{if } i < j. \end{cases} \tag{49}$$

for all $0 \le i \le j \le n$. The **cost** of T on n nodes is recursively given by

$$C(T) := \begin{cases} 0 & \text{if } n = 0, \\ W(0,n) + C(T_L) + C(T_R) & \text{if } n > 0, \end{cases}$$

where T_L and T_R are the left and right subtrees of T. We say T is **optimal** if its cost $C(T)$ is minimum among all BST's with the probability (48). Why is this definition of "cost" reasonable? Let us charge a unit cost to each node we visit during a lookup on a key K. If K has the frequency distribution given by the probabilities (48), then the expected charge to any node is precisely $W(i-1,j)$ if the subtree at u has the keys $K_i, \ldots, K_j$. So $C(T)$ is the expected cost of looking up K in the search tree T.

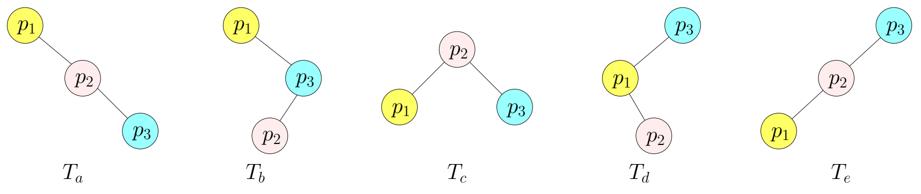


Figure 10: 5 binary search trees on (p_1, p_2, p_3).

¶52. Example. Consider $n = 3$ in which the q's are zero. The 5 possible BST's are shown in Figure 10. Then the cost of these trees are, resp.,

$$\left.\begin{array}{lll} C(T_a) = p_1 + 2p_2 + 3p_3, & C(T_b) = p_1 + 3p_2 + 2p_3, & C(T_c) = 2p_1 + p_2 + 2p_3, \\ C(T_d) = 2p_1 + 3p_2 + p_3, & C(T_e) = 3p_1 + 2p_2 + p_3. & \end{array}\right\}$$

Suppose $(p_1, p_2, p_3) = (6, 1, 3)$. Then the optimal tree is T_b with cost $6 + 3(1) + 2(3) = 15$. Note that the "balanced tree" T_c has a bigger cost of 19.

With the probabilistic interpretation, we naturally have

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

But in the following, we only need the p's and q's to be non-negative, and interpret them to be "relative weights". Also, for our purposes, the actual keys K_i are irrelevant: only the probabilities p_i, q_j are of interest. There are also two important special cases for this problem: we have seen an example where all the q's are zero. We can also consider the case where all the p's are zero.

Exercises

Exercise 7.1: Describe the precise connection between the optimal search tree problem and the optimal triangularization problem. ♢

Exercise 7.2: Suppose $n = 3$ and q_i's are all zero as in Figure 10. Let $p_i = \max\{p_1, p_2, p_3\}$. When is the optimal tree NOT rooted at p_i? ♢

Exercise 7.3: Suppose $n = 3$ and all the q's are zero. We have five possible BST's as shown in Figure 10: $T_a, T_b, \ldots, T_e$. Characterize the domain D_i of (p_1, p_2, p_3) for which the tree T_i is the optimal binary search tree ($i = a, b, \ldots, e$). Are these domains connected? convex? (Characterize them) ♢

Exercise 7.4: Same as the previous question, except that the p's are now zero. Characterize the domain D_i of $(q_0, \ldots, q_4)$ for which the tree T_i is the optimal. ♢

Exercise 7.5: The key words in a programming language are not completely independent. For instance, the key words `if` and `else` surely satisfy the property that (for syntactically correct programs), $\Pr(\texttt{if}) \ge \Pr(\texttt{else})$. For language with `begin` and `end` key words, we expect them to have the same probability. Can one exploit such relations? ♢

Exercise 7.6: (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 7.7: 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 7.8: 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. $\diamondsuit$

Exercise 7.9: (Open) The problem of Optimal BST is static problem. Explore the idea of "dynamic optimal BST" whereby the search probabilities slowly change over a sequence of lookups. $\diamondsuit$

Exercise 7.10: 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. $\diamondsuit$

End Exercises

§8. Quadrangular Inequality

¶53. Speeding Up. Our goal is to improve the complexity of the dynamic programming solution in the last section from $O(n^3)$ to $O(n^2)$. The basis of this improvement hinges on some inequalities of our weight matrix $W(i, j)$.

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. By exploiting additional properties of the cost function, Knuth shows this can be improved to $O(n^2)$ time. The key to the improvement is a general inequality satisfied by the cost function which was first clarified by F. Yao. To treat this, we reformulate the optimal search tree problem in an abstract framework.

In the optimal BST 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 example representing such a matrix:

$$W_1 = \begin{array}{|c|c|c|c|c|} \hline 1 & 2 & 1 & 2 & 1 \\ \hline & 2 & 0 & 3 & 2 \\ \hline & & 1 & 0 & 1 \\ \hline & & & 4 & 2 \\ \hline & & & & 2 \\ \hline \end{array}. \qquad (n=4) \tag{50}$$

Definition 1 *Let $n \geq 2$ be an integer. A **triangular function** W (of order n) is a function that assigns a non-negative value $W(i,j)$ to each (i,j) satisfying $0 \leq i \leq j \leq n$. We can represent W by an upper triangular matrix of dimensions $(n+1)\times(n+1)$ in the natural way, as in (50). A quadruple (i,i',j,j') (of order n) is **admissible** if*

$$1 \leq i \leq i' \leq j \leq j' \leq n.$$

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

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

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

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

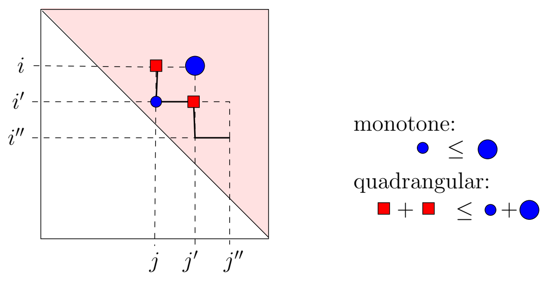


Figure 11: Monotone and quadrangular weight matrix.

We sometimes write W_{ij} or $W_{i,j}$ instead of $W(i,j)$. 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 11). Monotonicity and quadrangularity are visually illustrated in Figure 11:

- Monotonic means that along any northward-eastward 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.

The following is easily verified:

Lemma 6 *The weight matrix for the optimal binary 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

$$W^*(i,j;k) := W(i,j) + W^*(i,k-1) + W^*(k,j), \qquad (51)$$

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

The operations research literature describes a certain **Monge property** of matrices (see, e.g., [4]). This turns out to be the quadrangle inequality restricted to admissible quadruples (i,i',j,j') where $i' = i+1$ and $j' = j+1$.

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

Lemma 7 *If W is monotone and quadrangular, then the derived weight matrix W^* is also quadrangular.*

*Proof.*We must show the quadrangular inequality

$$W^*(i,j) + W^*(i',j') \leq W^*(i,j') + W^*(i',j), \qquad (0 \leq i \leq i' \leq j \leq j' \leq n). \qquad (52)$$

First, we make the simple observation when $i = i'$ or $j = j'$, the inequality in equation (52) 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.

¶54. 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.

¶55. 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

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

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

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

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

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

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

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

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

Lemma 8 *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

$$K(i,j) \le K(i,j+1). \tag{53}$$

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

$$W^*(i,j;k') \le W^*(i,j;k) \quad \text{implies} \quad W^*(i,j+1;k') \le W^*(i,j+1;k). \tag{54}$$

To see the implication, suppose equation (53) 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 (54). **Q.E.D.**

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

Theorem 9 *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 (or levels). 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.**

¶58. 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 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 6) 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. $\diamondsuit$

Exercise 8.6: (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. $\diamondsuit$

Exercise 8.7: It is actually easy to give a "graphical" proof of lemma 8. In the Figure 12, this amounts to showing that if $A + a \geq B + b$ then $A' + a' \geq B' + b'$.

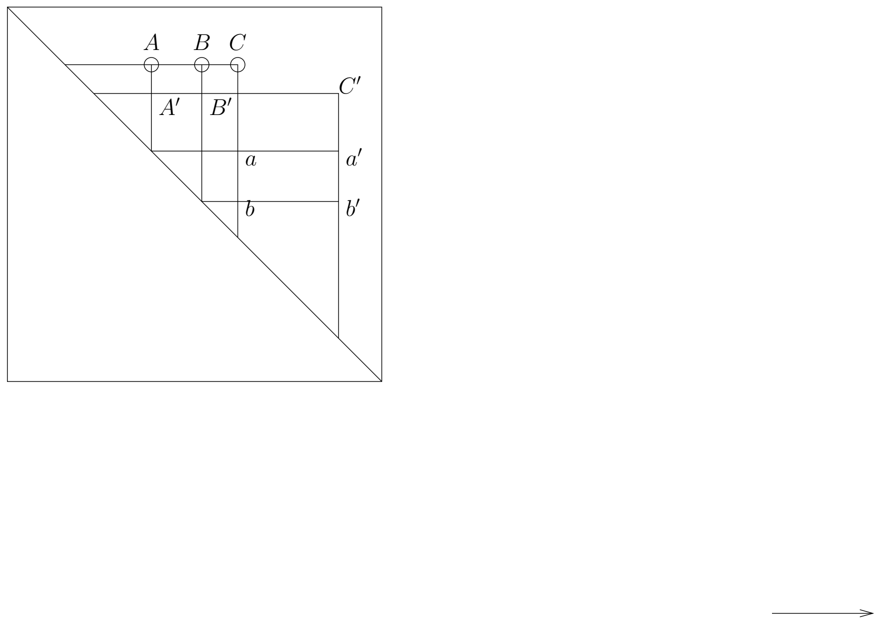


Figure 12: Derived weight matrix.

$\diamondsuit$

Exercise 8.8: If W is monotone and quadrangular, is W^* monotone? $\diamondsuit$

Exercise 8.9: 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

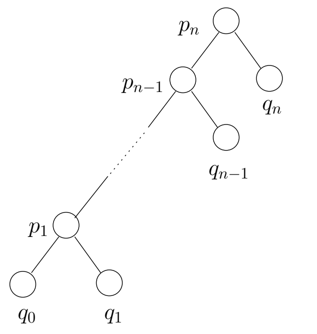


Figure 13: Linear list search tree.

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

Exercise 8.11: 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

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.