بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

حسین فلسفین

دیدیم که پیچیدگی زمانی ***every-case*** الگوریتم ما برای مسئلهٔ فروشندهٔ دوره گرد برابر با

$$T(n) = \sum_{k=1}^{n-2} (n-1-k)k\binom{n-1}{k}.$$

است.

$$\begin{aligned}(n-1-k)\binom{n-1}{k} &= (n-k-1)\frac{(n-1)!}{k!(n-k-1)!} = \frac{(n-1)!}{k!(n-k-2)!} \\ &= (n-1)\frac{(n-2)!}{k!(n-k-2)!} = (n-1)\binom{n-2}{k}\end{aligned}$$

$$T(n) = (n-1)\sum_{k=1}^{n-2} k\binom{n-2}{k}$$

$$k\binom{n-2}{k} = k\frac{(n-2)!}{k!(n-2-k)!} = \frac{(n-2)!}{(k-1)!(n-2-k)!} =$$
$$(n-2)\frac{(n-3)!}{(k-1)!(n-2-k)!} = (n-2)\binom{n-3}{k-1}$$

$$T(n) = (n-1)(n-2)\sum_{k=1}^{n-2}\binom{n-3}{k-1} =$$

$$(n-1)(n-2)\sum_{k=0}^{n-3}\binom{n-3}{k} = (n-1)(n-2)2^{n-3} \in \Theta(n^2 2^n)$$

At this point you may be wondering what we have gained, because our new algorithm is still $\Theta(n^2 2^n)$.

* *20 cities by brute-force algorithm:* $19!\ \mu sec$ *= 3857 years :(*

* *20 cities by dynamic programming algorithm:*

$$(20-1)(20-2)2^{20-3}\ \mu sec \text{ = 45 seconds :)}$$

* *60 cities by dynamic programming algorithm: Many years :(*

Because the memory used in this algorithm is also large, we will analyze the memory complexity, which we call $M(n)$. The memory used to store the arrays $D[v_i][A]$ and $P[v_i][A]$ is clearly the dominant amount of memory. So we will determine how large these arrays must be. $V - \{v_1\}$ contains $n-1$ vertices, thus it has 2^{n-1} subsets A. The first index of the arrays D and P ranges in value between 1 and n. Therefore,

$$M(n) = 2n2^{n-1} = n2^n \in \Theta(n2^n).$$

NP-HARDNESS: No one has ever found an algorithm for the Traveling Salesperson Problem whose worst-case time complexity is better than exponential. Yet no one has ever proved that such an algorithm is not possible.

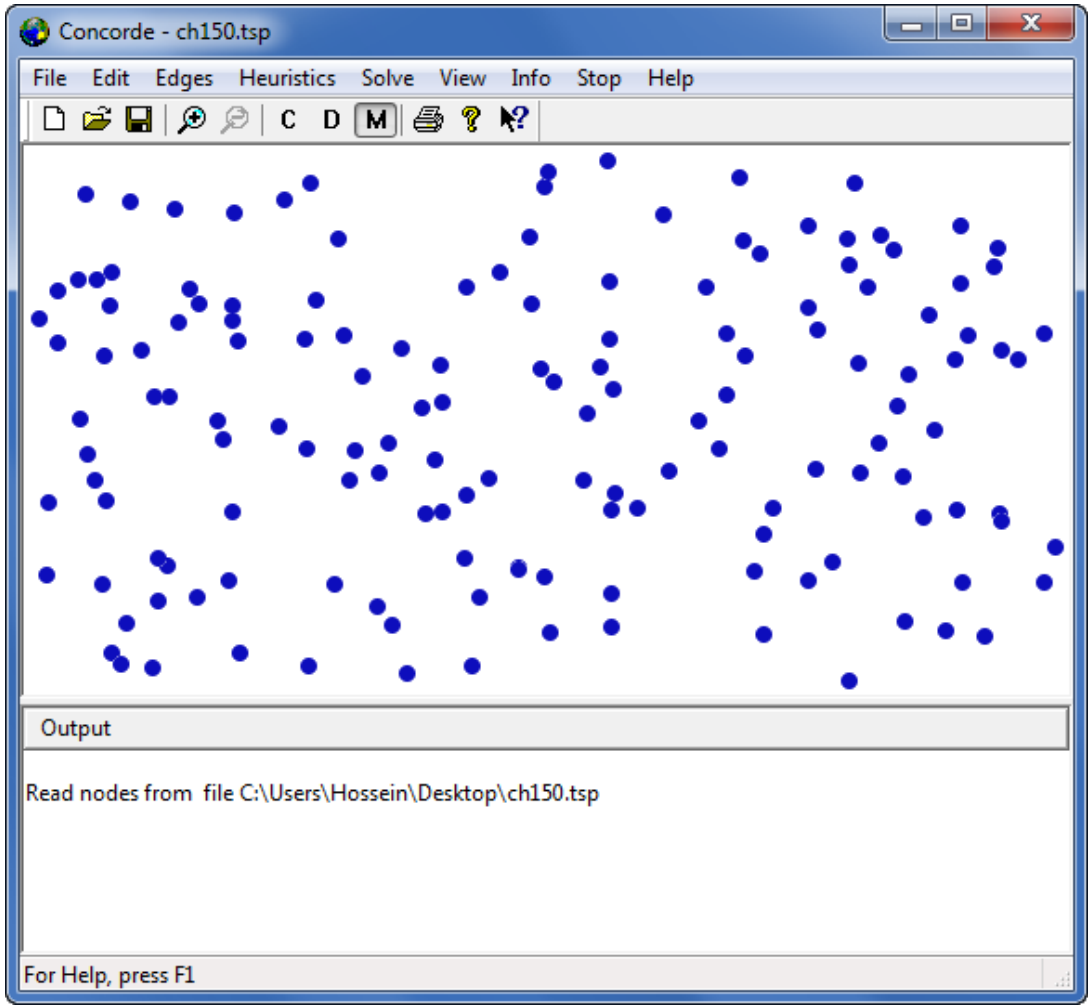
The Millennium Prize Problems

http://www.claymath.org/millennium-problems/p-vs-np-problem

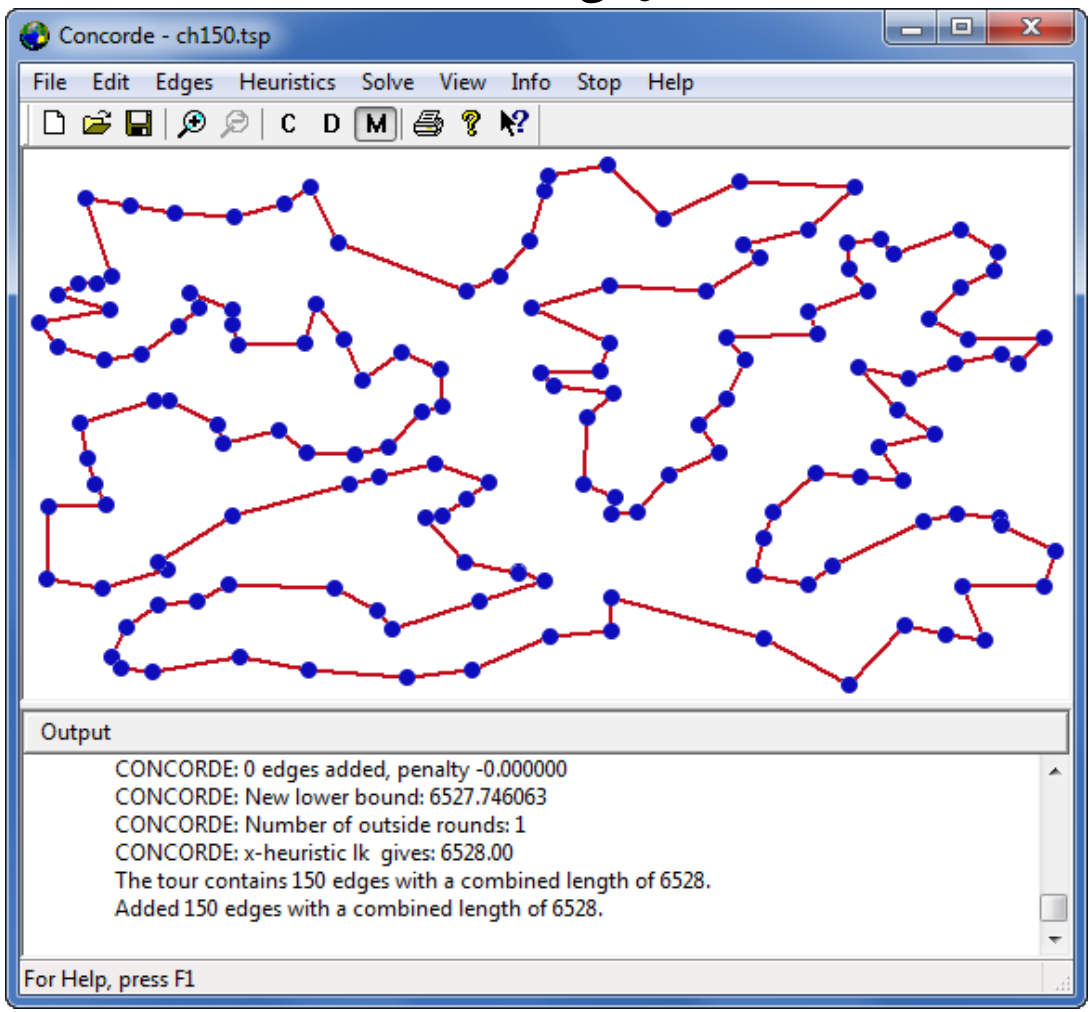
Clay Mathematics Institute

Concorde TSP Solver

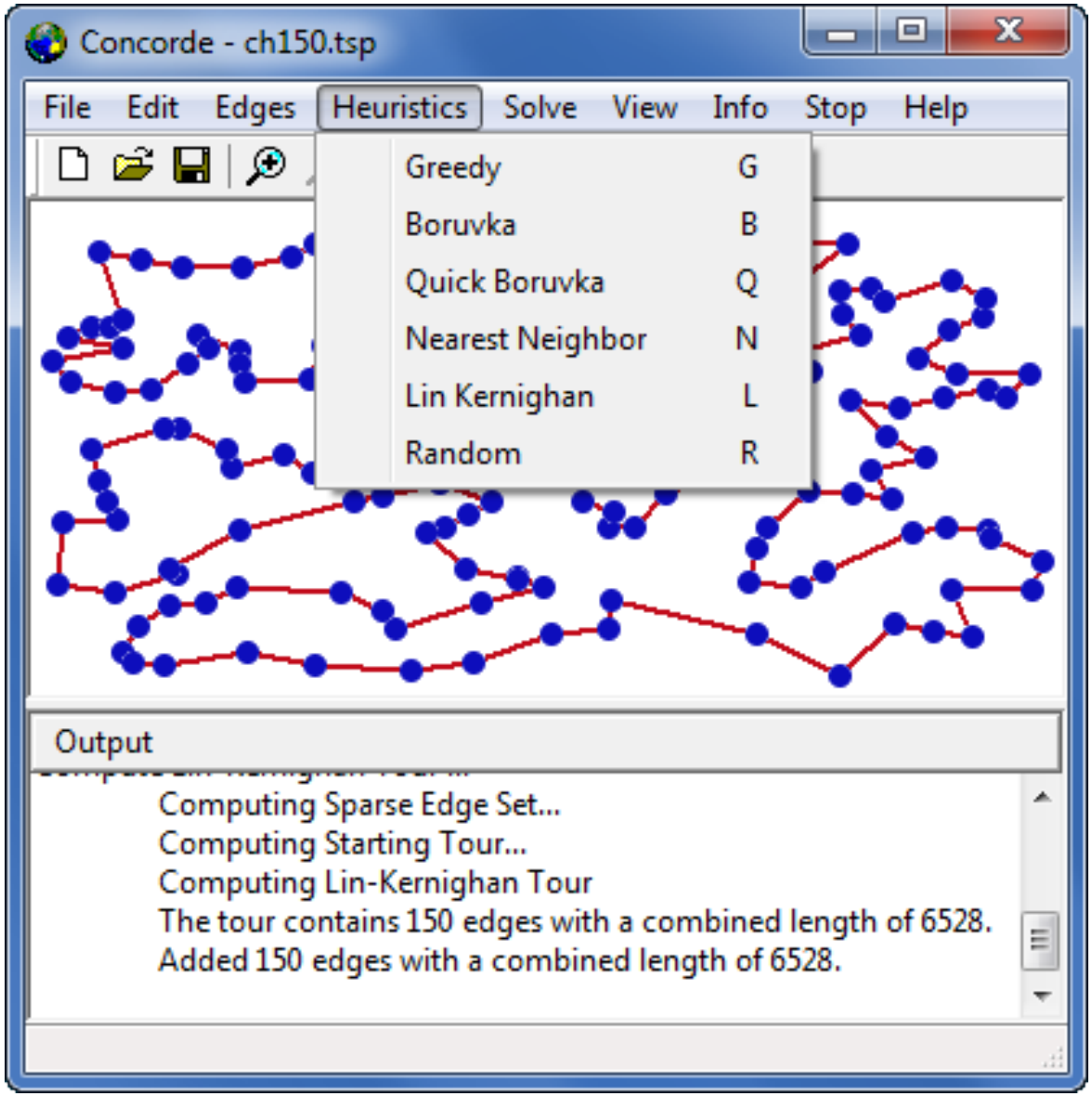
http://www.math.uwaterloo.ca/tsp/concorde/index.html



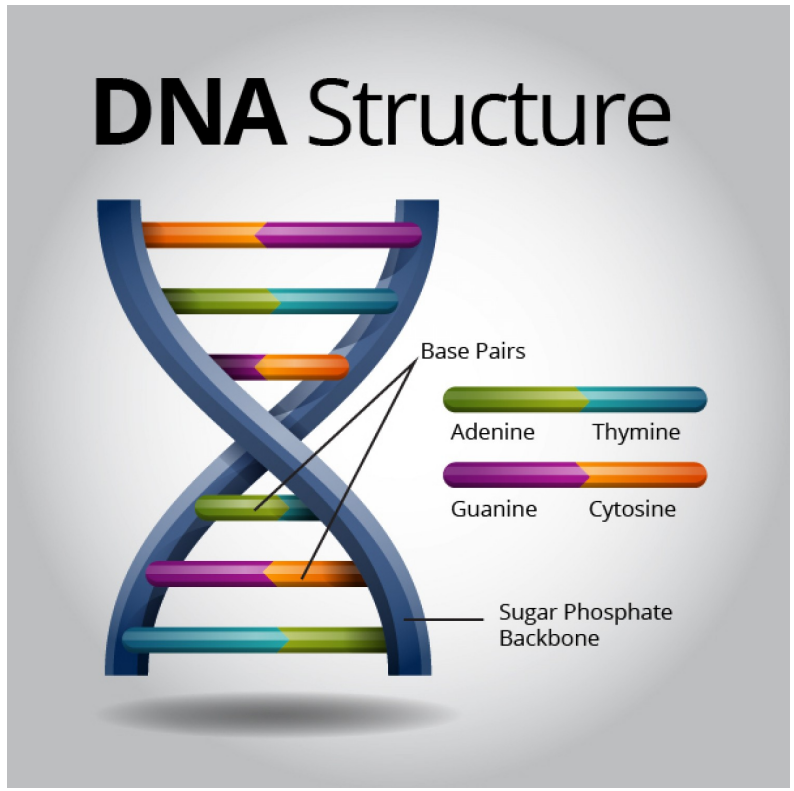
در کنکورد، الگوریتم دقیق، مبتنی بر راهبرد شاخه و کران (بعبارت بهتر: شاخه و برش) است.



کنکورد دربردارندهٔ چند الگوریتم غیردقیق ابتکاری (مکاشفه‌ای!؟) نیز هست:



Longest common subsequence



Biological applications often need to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine, and thymine. Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set $\{A, C, G, T\}$. *For example, the DNA of one organism may be* $S_1 =$ `ACCGGTCGAGTGCGCGGAAGCCGGCCGAA`, *and the DNA of another organism may be* $S_2 =$ `GTCGTTCGGAATGCCGTTGCTCTGTAAA`.

One reason to compare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are. We can define similarity in many different ways. One way to measure the similarity of strands S_1 and S_2 is by finding a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively. The longer the strand S_3 we can find, the more similar S_1 and S_2 are. In our example, the longest strand S_3 is `GTCGTCGGAAGCCGGCCGAA`.

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

For example, if

$$X = \langle A, B, C, B, D, A, B \rangle$$

and

$$Y = \langle B, D, C, A, B, A \rangle,$$

the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y. The sequence $\langle B, C, A \rangle$ is not a longest common subsequence (LCS) of X and Y, however, since it has length 3 and the sequence $\langle B, C, B, A \rangle$, which is also common to both X and Y, has length 4. The sequence $\langle B, C, B, A \rangle$ is an LCS of X and Y, as is the sequence $\langle B, D, A, B \rangle$, since X and Y have no common subsequence of length 5 or greater.

In the longest-common-subsequence problem, we are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum length common subsequence of X and Y.

In a brute-force approach to solving the LCS problem, we would enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence we find. Each subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of X. Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, we define the ith prefix of X, for $i = 0, 1, \ldots, m$, as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. For example, if $X = \langle A, B, C, B, D, A, B \rangle$, then $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence.

Theorem (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ ***and*** $Y = \langle y_1, y_2, \ldots, y_n \rangle$ ***be sequences, and let*** $Z = \langle z_1, z_2, \ldots, z_k \rangle$ ***be any LCS of*** X ***and*** Y***.***

1. ***If*** $x_m = y_n$***, then*** $z_k = x_m = y_n$ ***and*** Z_{k-1} ***is an LCS of*** X_{m-1} ***and*** Y_{n-1}***.***

2. ***If*** $x_m \neq y_n$***, then*** $z_k \neq x_m$ ***implies that*** Z ***is an LCS of*** X_{m-1} ***and*** Y***.***

3. ***If*** $x_m \neq y_n$***, then*** $z_k \neq y_n$ ***implies that*** Z ***is an LCS of*** X ***and*** Y_{n-1}***.***

Proof (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a *longest* common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length-$(k - 1)$ common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k - 1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

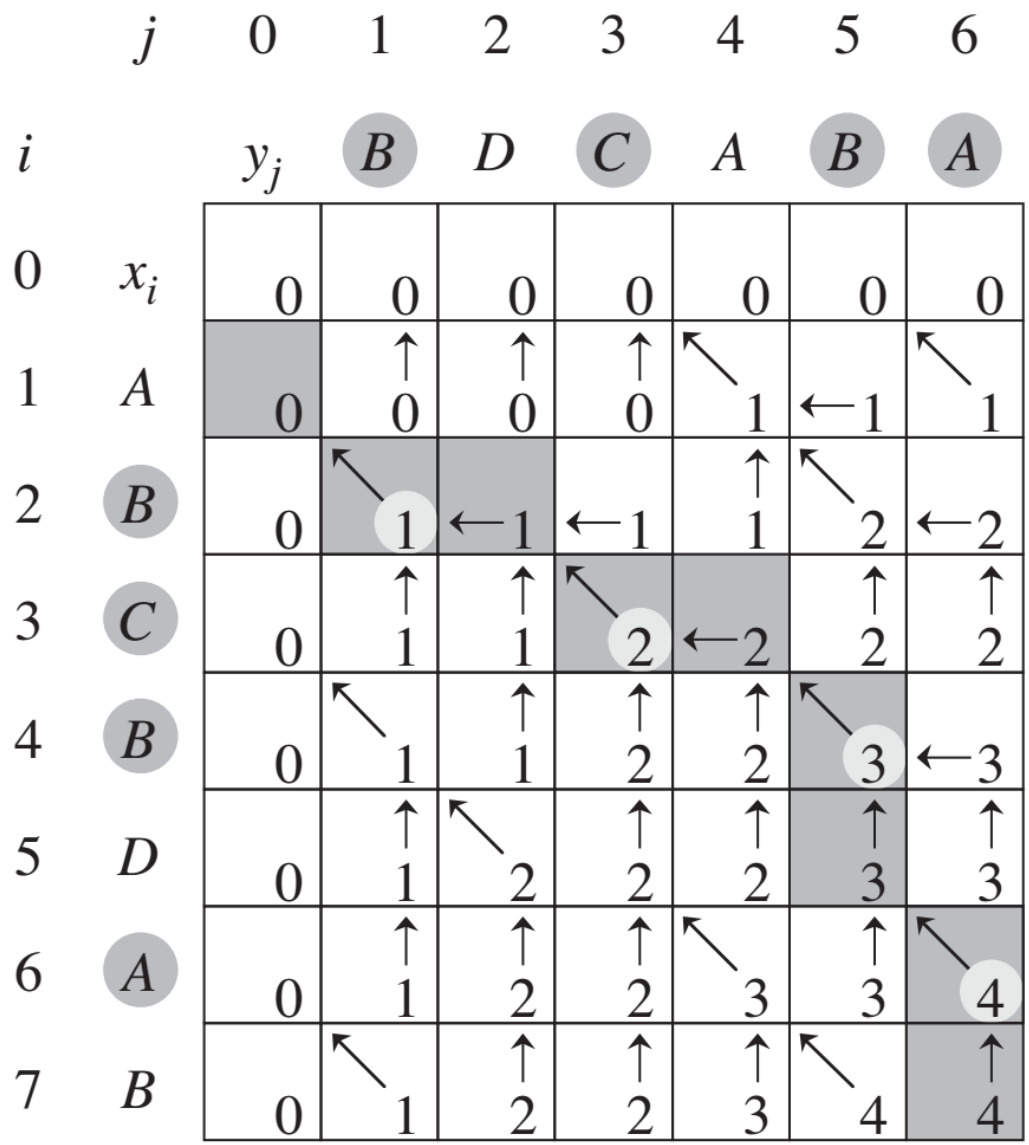
(3) The proof is symmetric to (2). ■

The above theorem implies that we should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ *and* $Y = \langle y_1, y_2, \ldots, y_n \rangle$*. If* $x_m = y_n$*, we must find an LCS of* X_{m-1} *and* Y_{n-1}*. Appending* $x_m = y_n$ *to this LCS yields an LCS of* X *and* Y*. If* $x_m \neq y_n$*, then we must solve two subproblems: finding an LCS of* X_{m-1} *and* Y *and finding an LCS of* X *and* Y_{n-1}*. Whichever of these two LCSs is longer is an LCS of* X *and* Y*. Because these cases exhaust all possibilities, we know that one of the optimal subproblem solutions must appear within an LCS of* X *and* Y*.*

Our recursive solution to the LCS problem involves establishing a recurrence for the value of an optimal solution. Let us define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

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

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```


j 0 1 2 3 4 5 6
i
y_j B D C A B A
0 x_i 0 0 0 0 0 0 0
1 A 0 ↑0 ↑0 ↑0 ↖1 ←1 ↖1
2 B 0 ↖1 ←1 ←1 ↑1 ↖2 ←2
3 C 0 ↑1 ↑1 ↖2 ←2 ↑2 ↑2
4 B 0 ↖1 ↑1 ↑2 ↑2 ↖3 ←3
5 D 0 ↑1 ↖2 ↑2 ↑2 ↑3 ↑3
6 A 0 ↑1 ↑2 ↑2 ↖3 ↑3 ↖4
7 B 0 ↖1 ↑2 ↑2 ↑3 ↖4 ↑4

Procedure LCS-LENGTH stores the $c[i, j]$ values in a table $c[0..m, 0..n]$, and it computes the entries in row-major order. (That is, the procedure fills in the first row of c from left to right, then the second row, and so on.) The procedure also maintains the table $b[1..m, 1..n]$ to help us construct an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$. The procedure returns the b and c tables; $c[m, n]$ contains the length of an LCS of X and Y.

The running time of the procedure is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

```
PRINT-LCS(b, X, i, j)
1   if i == 0 or j == 0
2       return
3   if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6   elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8   else PRINT-LCS(b, X, i, j − 1)
```

The b table returned by LCS-LENGTH enables us to quickly construct an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. We simply begin at $b[m, n]$ and trace through the table by following the arrows. Whenever we encounter a "$\nwarrow$" in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH found. With this method, we encounter the elements of this LCS in reverse order. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is

PRINT-LCS$(b, X, X.length, Y.length)$.

The procedure takes time $O(m + n)$, since it decrements at least one of i and j in each recursive call.

Longest Common Substring

The longest common substring of two strings $S[1..m]$ and $T[1..n]$ can be obtained by finding the longest common suffixes between each pair of prefixes of the strings, keeping all the longest ones. In general, the length $M[i,j]$ of a longest common suffix between two prefixes $S[1..i]$ and $T[1..j]$ of the strings S and T is given by the recurrence

$$M[i,j] = \begin{cases} 0, & \text{if } S[i] \neq T[j], \\ M[i-1,j-1]+1, & \text{if } S[i] = T[j]. \end{cases}$$

Therefore, if we let $M[i,j]$ denote the number of characters in the longest common suffix of $S[1..i]$ and $T[1..j]$. When $S[i] \neq T[j]$, there is no way the last pair of characters could match, so $M[i,j] = 0$. But if $S[i] = T[j]$, then we have $M[i,j] = M[i-1,j-1]+1$.

		G 1	C 2	T 3	T 4	C 5	C 6	G 7	G 8	C 9	T 10	C 11	G 12	T 13	A 14	T 15	A 16	A 17	T 18	G 19	T 20	G 21	T 22	G 23	G 24
T	1	0	0	1	1	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0	0
G	2	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	2	0	2	0	2	1
C	3	0	2	0	0	1	1	0	0	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
T	4	0	0	3	1	0	0	0	0	0	3	0	0	1	0	1	0	0	1	0	1	0	1	0	0
T	5	0	0	1	4	0	0	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1	0	0
C	6	0	1	0	0	5	1	0	0	1	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
T	7	0	0	2	1	0	0	0	0	0	2	0	0	1	0	1	0	0	1	0	1	0	1	0	0
G	8	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	2	0	2	0	2	1
A	9	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0
C	10	0	1	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
T	11	0	0	2	1	0	0	0	0	0	2	0	0	1	0	1	0	0	1	0	1	0	1	0	0
A	12	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	2	1	0	0	0	0	0	0	0
T	13	0	0	1	1	0	0	0	0	0	1	0	0	1	0	3	0	0	2	0	1	0	1	0	0
A	14	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	4	1	0	0	0	0	0	0	0
A	15	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	5	0	0	0	0	0	0	0
T	16	0	0	1	1	0	0	0	0	0	1	0	0	1	0	2	0	0	**6**	0	1	0	1	0	0
A	17	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	3	1	0	0	0	0	0	0	0
G	18	1	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	1	0	1	0	1	1

Polyphonic & Apocalyptic

		p	*o*	*l*	*y*	*p*	*h*	*o*	*n*	*i*	*c*
	0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
a	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
p	*0*	*1*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*
o	*0*	*0*	*2*	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*
c	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*
a	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
l	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
y	*0*	*0*	*0*	*0*	*2*	*0*	*0*	*0*	*0*	*0*	*0*
p	*0*	*1*	*0*	*0*	*0*	*3*	*0*	*0*	*0*	*0*	*0*
t	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
i	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*
c	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*2*

The Greedy Approach

A greedy algorithm grabs data items in sequence, each time taking the one that is deemed "best" according to some criterion, without regard for the choices it has made before or will make in the future. One should not get the impression that there is something wrong with greedy algorithms because of the word "greedy." They often lead to very efficient and simple solutions.

Like dynamic programming, greedy algorithms are often used to solve optimization problems. However, the greedy approach is more straightforward. In dynamic programming, a recursive property is used to divide an instance into smaller instances. In the greedy approach, there is no division into smaller instances. A greedy algorithm arrives at a solution by making a sequence of choices, each of which simply looks the best at the moment. That is, each choice is locally optimal. The hope is that a globally optimal solution will be obtained, but this is not always the case. For a given algorithm, we must determine whether the solution is always optimal.