CSE 2320 Notes 16: Dynamic Programming

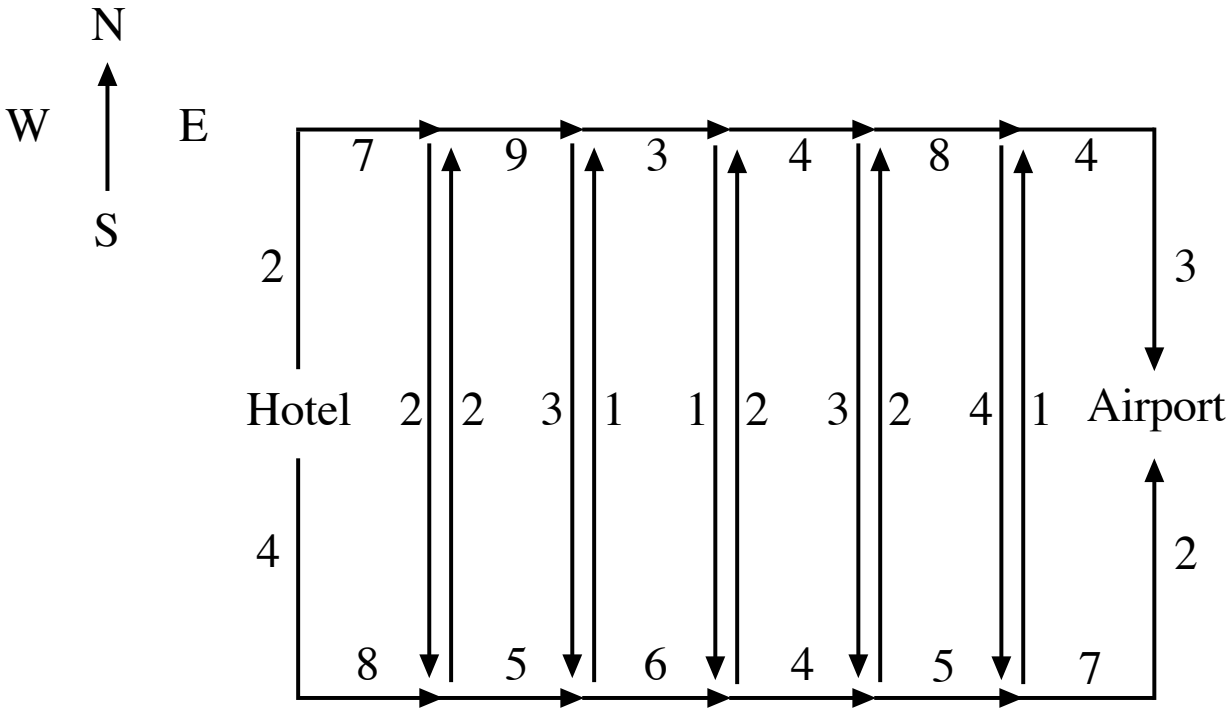
(Last updated 11/20/06 7:43 AM)

CLRS, 15.1-15.4

DYNAMIC PROGRAMMING APPROACH

1. Describe problem input.
2. Determine cost function and base case.
3. Determine general case for cost function. THE HARD PART!!!
4. Appropriate ordering of subproblems.
5. Backtrace for solution. *Most of the effort in dynamic programming is ignored at the end.*

A SMALL EXAMPLE – Shuttle-to-Airport (similar to "assembly-line scheduling" in CLRS)



How many different paths (by brute force)?

Observation: To find optimal route, need optimal route to each street corner.

Don't cheat and use _____!

1. Describe problem input.

   Four arrays of paths, each with n values

   Upper Direct = UD = $ud_1\ ud_2\ \ldots\ ud_n$ = 9 (2 + 7), 9, 3, 4, 8, 7 (4 + 3)

   Lower Direct = LD = $ld_1\ ld_2\ \ldots\ ld_n$ = 12 (4 + 8), 5, 6, 4, 5, 9 (7 + 2)

   Upper-to-Lower = UL = $ul_1\ ul_2\ \ldots\ ul_n$ = 2, 3, 1, 3, 4, 0

   Lower-to-Upper = LU = $lu_1\ lu_2\ \ldots\ lu_n$ = 2, 1, 2, 2, 1, 0

2. Determine cost function and base case.

   U(i) = Cost to reach upper corner i

   L(i) = Cost to reach lower corner i

   U(0) = 0

   L(0) = 0

3. Determine general case.

   $U(i) = \min \{ U(i - 1) + ud_i, L(i - 1) + ld_i + lu_i \}$

   $L(i) = \min \{ L(i - 1) + ld_i, U(i - 1) + ud_i + ul_i \}$

4. Appropriate ordering of subproblems.

   U(i) and L(i) cannot be computed without U(i - 1) and L(i - 1)

5. Backtrace for solution – either

   1) (`shuttle1.c`) *explicitly save* indication of which of the two cases was used (continue - c, switch - s), or

   2) (`shuttle2.c`) *recheck* during backtrace for which case was used.

	0	1	2	3	4	5	6
U	0	9 (c)	17 (s)	20 (c)	24 (c)	31 (s)	38 (c)
L	0	11 (s)	16 (c)	21 (s)	25 (c)	30 (c)	39 (s)

Dynamic programming is:

1. Exhaustive search without brute force.

2. Optimal solution to big problem from optimal solutions to subproblems.

WEIGHTED INTERVAL SCHEDULING (not in CLRS)

Input: A set of n intervals numbered 1 through n with each interval i having start time s_i, finish time f_i, and positive weight v_i,

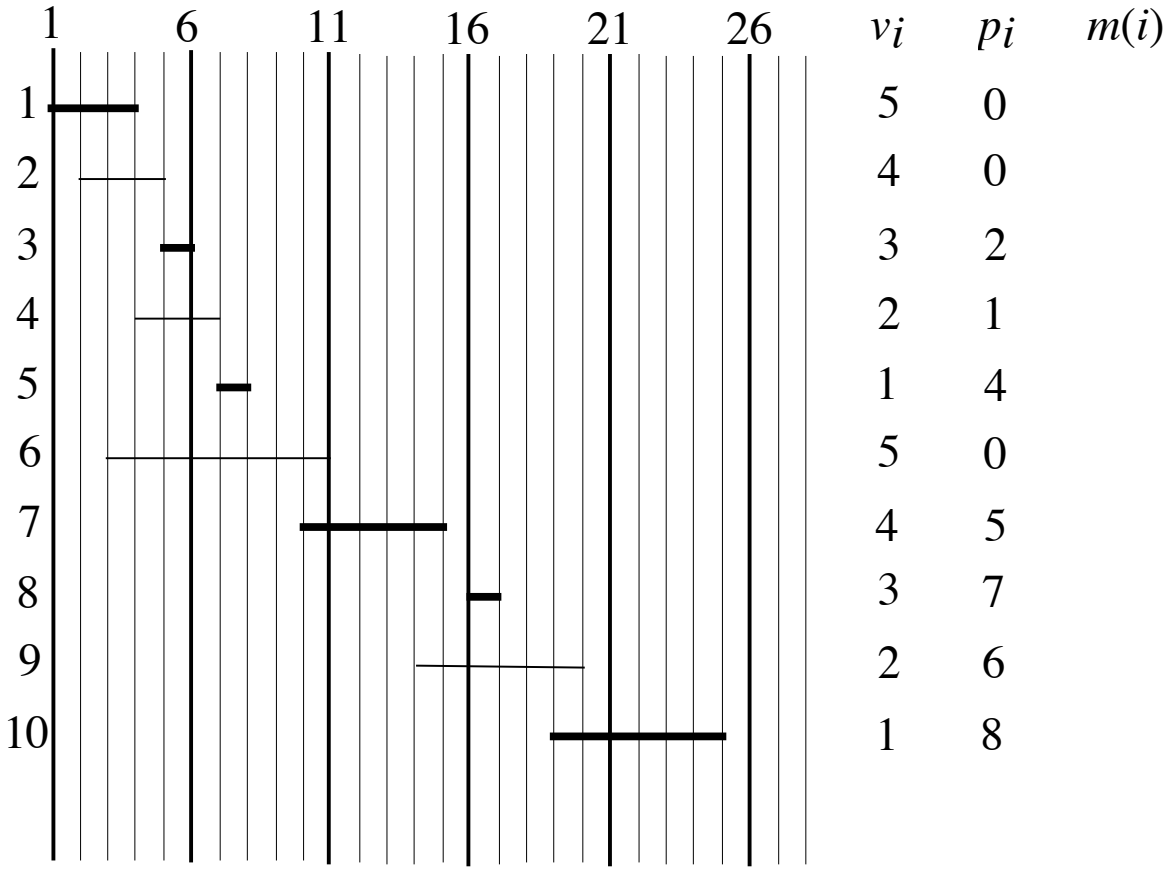
Output: A set of non-overlapping intervals to ***maximize*** the sum of their weights. (Two intervals i and j overlap if either $s_i < s_j < f_i$ or $s_i < f_j < f_i$.)

Brute-force solution: Enumerate the powerset of the input intervals, discard those cases with overlapping intervals, and compute the sum of the weights for each.

1. Describe problem input.

   Assume the n intervals are in ascending finish time order, i.e. $f_i \le f_{i+1}$.

   Let p_i be the *rightmost preceding interval* for interval i, i.e. the largest value $j < i$ such that intervals i and j do not overlap. If no such interval j exists, $p_i = 0$. (These values may be computed in $\Theta(n \log n)$ time. See `wis.c`.)



2. Determine cost function and base case.

   $M(i)$ = Cost for optimal non-overlapping subset for the first *i* input intervals.

   $M(0) = 0$

3. Determine general case.

   For $M(i)$, the main issue is: *Does the optimal subset include interval i?*

   If *yes*: optimal subset cannot include any overlapping intervals, so $M(i) = M(p_i) + v_i$.

   If *no*: optimal subset is the same as for $M(i-1)$, so $M(i) = M(i-1)$.

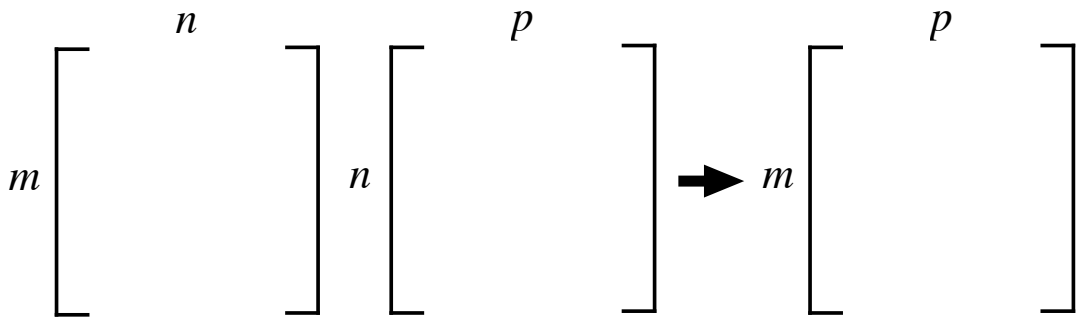
   This observation tells us to compute cost ***both*** ways and keep the maximum.

4. Appropriate ordering of subproblems. Simply compute $M(i)$ in ascending *i* order.

5. Backtrace for solution (with recheck). This is the subset of intervals for $M(n)$.

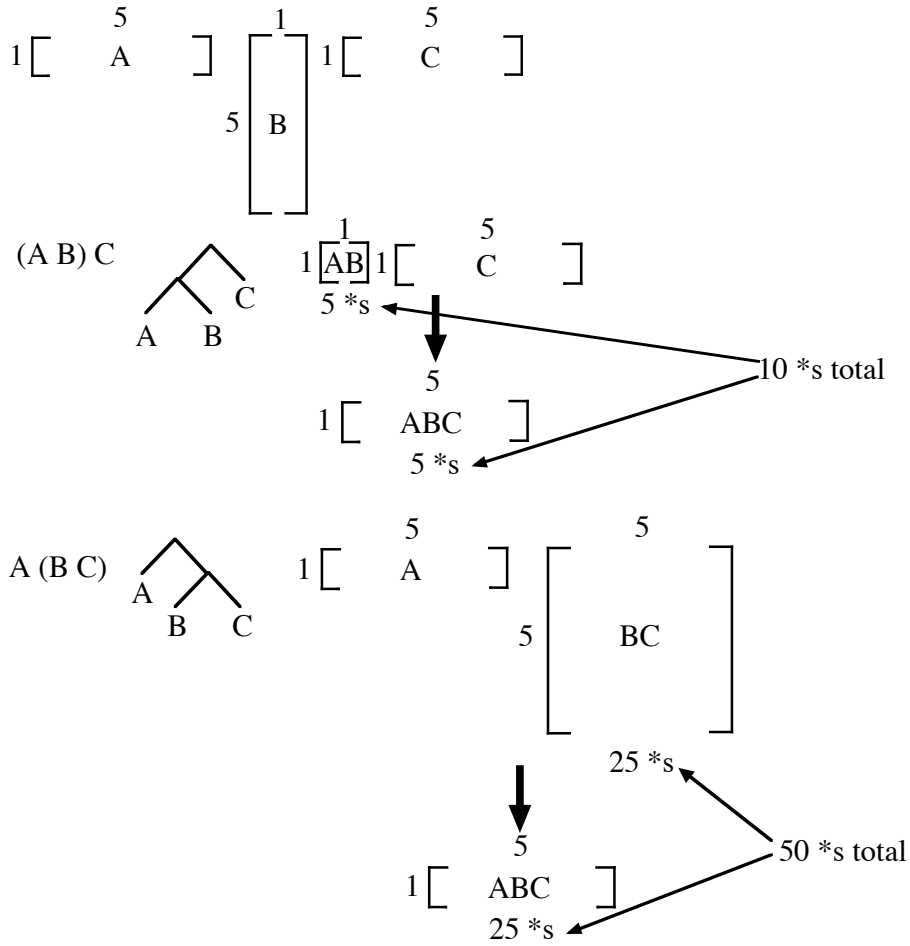
```
i=n;
while (i>0)
  if (v[i]+M[p[i]]>=M[i-1])
  {
    // Interval i is in solution
    i=p[i];
  }
  else
    i--;
```

OPTIMAL MATRIX MULTIPLICATION ORDERING (very simplified version of query optimization)



Only one strategy for multiplying two matrices – requires *mnp* scalar multiplications (and $m(n-1)p$ additions).

There are two strategies for multiplying three matrices:



Ways to parenthesize n matrices?

$$\frac{1}{n}\binom{2n-2}{n-1}$$ (CLRS, p. 333)

Observation: Final tree cannot be optimal if any subtree is not.

1. Describe problem input.

   n matrices $\Rightarrow$ $n + 1$ sizes

$$P_0 \begin{bmatrix} & P_1 \\ & \\ M_1 & \end{bmatrix} P_1 \begin{bmatrix} & P_2 \\ & \\ M_2 & \end{bmatrix} \quad \cdots \quad P_{n-1} \begin{bmatrix} & P_n \\ & \\ M_n & \end{bmatrix}$$

2. Determine cost function and base case.

   $C(i, j)$ = Cost for optimally multiplying $M_i \ldots M_j$

   $C(i, i) = 0$

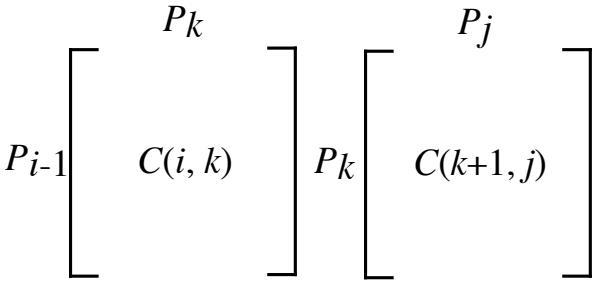
3. Determine general case.

   Consider a specific case $C(5, 9)$. The optimal way to multiply $M_5 \ldots M_9$ could be any of the following:

   $C(5, 5) + C(6, 9) + P_4P_5P_9$
   $C(5, 6) + C(7, 9) + P_4P_6P_9$
   $C(5, 7) + C(8, 9) + P_4P_7P_9$
   $C(5, 8) + C(9, 9) + P_4P_8P_9$

   Compute all four and keep the smallest one.

   Abstractly: Trying to find $C(i, j)$



$$C(i,j) = \min_{i \le k < j} \left\{ C(i,k) + C(k+1,j) + P_{i-1}P_kP_j \right\}$$

4. Appropriate ordering of subproblems.

   Since smaller subproblems are needed to solve larger problems, run value for $j - i$ for $C(i, j)$ from 0 to $n - 1$.

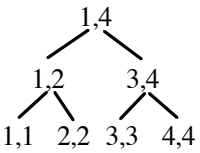
5. Backtrace for solution – explicitly save the k value that gave each $C(i, j)$.

Takes $\Theta(n^3)$ time – see exercise 15.2-4.

```
// Optimal matrix multiplication order using dynamic programming
#include <stdio.h>
main()
{
int p[20];
int n;
int c[20][20];
int trace[20][20];

int i,j,k;
int work;

scanf("%d",&n);
for (i=0;i<=n;i++)
  scanf("%d",&p[i]);
for (i=1;i<=n;i++)
  c[i][i]=trace[i][i]=0;
for (i=1;i<n;i++)
  for (j=1;j<=n-i;j++)
  {
    printf("Compute c[%d][%d]\n",j,j+i);
    c[j][j+i]=999999;
    for (k=j;k<j+i;k++)
    {
      work=c[j][k]+c[k+1][j+i]+p[j-1]*p[k]*p[j+i];
      printf("  k=%d gives cost %3d=c[%d][%d]+c[%d][%d]+p[%d]*p[%d]*p[%d]\n",
        k,work,j,k,k+1,j+i,j-1,k,j+i);
      if (c[j][j+i]>work)
      {
        c[j][j+i]=work;
        trace[j][j+i]=k;
      }
    }
    printf("  c[%d][%d]==%d,trace[%d][%d]==%d\n",j,j+i,
      c[j][j+i],j,j+i,trace[j][j+i]);
  }

printf("   ");
for (i=1;i<=n;i++)
  printf("   %3d   ",i);
printf("\n");
for (i=1;i<=n;i++)
{
  printf("%2d ",i);
  for (j=1;j<=n;j++)
    if (i>j)
      printf(" ------- ");
    else
      printf(" %3d %3d ",c[i][j],trace[i][j]);
  printf("\n");
  printf("\n");
}
}
```

```
4
2 4 3 5 2
Compute c[1][2]
  k=1 gives cost  24=c[1][1]+c[2][2]+p[0]*p[1]*p[2]
  c[1][2]==24,trace[1][2]==1
Compute c[2][3]
  k=2 gives cost  60=c[2][2]+c[3][3]+p[1]*p[2]*p[3]
  c[2][3]==60,trace[2][3]==2
Compute c[3][4]
  k=3 gives cost  30=c[3][3]+c[4][4]+p[2]*p[3]*p[4]
  c[3][4]==30,trace[3][4]==3
Compute c[1][3]
  k=1 gives cost 100=c[1][1]+c[2][3]+p[0]*p[1]*p[3]
  k=2 gives cost  54=c[1][2]+c[3][3]+p[0]*p[2]*p[3]
  c[1][3]==54,trace[1][3]==2
Compute c[2][4]
  k=2 gives cost  54=c[2][2]+c[3][4]+p[1]*p[2]*p[4]
  k=3 gives cost 100=c[2][3]+c[4][4]+p[1]*p[3]*p[4]
  c[2][4]==54,trace[2][4]==2
```

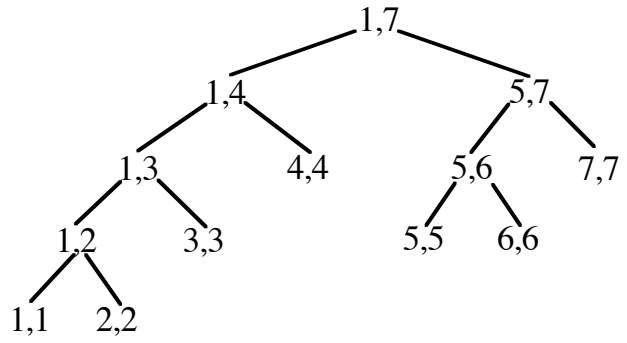
```
Compute c[1][4]
  k=1 gives cost  70=c[1][1]+c[2][4]+p[0]*p[1]*p[4]
  k=2 gives cost  66=c[1][2]+c[3][4]+p[0]*p[2]*p[4]
  k=3 gives cost  74=c[1][3]+c[4][4]+p[0]*p[3]*p[4]
  c[1][4]==66,trace[1][4]==2
       1       2       3       4
 1    0   0   24   1   54   2   66   2

 2  -------    0   0   60   2   54   2

 3  -------  -------    0   0   30   3

 4  -------  -------  -------    0   0
```




```
7
1 7 9 5 1 5 10 3
Compute c[1][2]
  k=1 gives cost  63=c[1][1]+c[2][2]+p[0]*p[1]*p[2]
  c[1][2]==63,trace[1][2]==1
Compute c[2][3]
  k=2 gives cost 315=c[2][2]+c[3][3]+p[1]*p[2]*p[3]
  c[2][3]==315,trace[2][3]==2
Compute c[3][4]
  k=3 gives cost  45=c[3][3]+c[4][4]+p[2]*p[3]*p[4]
  c[3][4]==45,trace[3][4]==3
Compute c[4][5]
  k=4 gives cost  25=c[4][4]+c[5][5]+p[3]*p[4]*p[5]
  c[4][5]==25,trace[4][5]==4
Compute c[5][6]
  k=5 gives cost  50=c[5][5]+c[6][6]+p[4]*p[5]*p[6]
  c[5][6]==50,trace[5][6]==5
Compute c[6][7]
  k=6 gives cost 150=c[6][6]+c[7][7]+p[5]*p[6]*p[7]
  c[6][7]==150,trace[6][7]==6
Compute c[1][3]
  k=1 gives cost 350=c[1][1]+c[2][3]+p[0]*p[1]*p[3]
  k=2 gives cost 108=c[1][2]+c[3][3]+p[0]*p[2]*p[3]
  c[1][3]==108,trace[1][3]==2
Compute c[2][4]
  k=2 gives cost 108=c[2][2]+c[3][4]+p[1]*p[2]*p[4]
  k=3 gives cost 350=c[2][3]+c[4][4]+p[1]*p[3]*p[4]
  c[2][4]==108,trace[2][4]==2
Compute c[3][5]
  k=3 gives cost 250=c[3][3]+c[4][5]+p[2]*p[3]*p[5]
  k=4 gives cost  90=c[3][4]+c[5][5]+p[2]*p[4]*p[5]
  c[3][5]==90,trace[3][5]==4
Compute c[4][6]
  k=4 gives cost 100=c[4][4]+c[5][6]+p[3]*p[4]*p[6]
  k=5 gives cost 275=c[4][5]+c[6][6]+p[3]*p[5]*p[6]
  c[4][6]==100,trace[4][6]==4
Compute c[5][7]
  k=5 gives cost 165=c[5][5]+c[6][7]+p[4]*p[5]*p[7]
  k=6 gives cost  80=c[5][6]+c[7][7]+p[4]*p[6]*p[7]
  c[5][7]==80,trace[5][7]==6
Compute c[1][4]
  k=1 gives cost 115=c[1][1]+c[2][4]+p[0]*p[1]*p[4]
  k=2 gives cost 117=c[1][2]+c[3][4]+p[0]*p[2]*p[4]
  k=3 gives cost 113=c[1][3]+c[4][4]+p[0]*p[3]*p[4]
  c[1][4]==113,trace[1][4]==3
Compute c[2][5]
  k=2 gives cost 405=c[2][2]+c[3][5]+p[1]*p[2]*p[5]
  k=3 gives cost 515=c[2][3]+c[4][5]+p[1]*p[3]*p[5]
  k=4 gives cost 143=c[2][4]+c[5][5]+p[1]*p[4]*p[5]
  c[2][5]==143,trace[2][5]==4
```

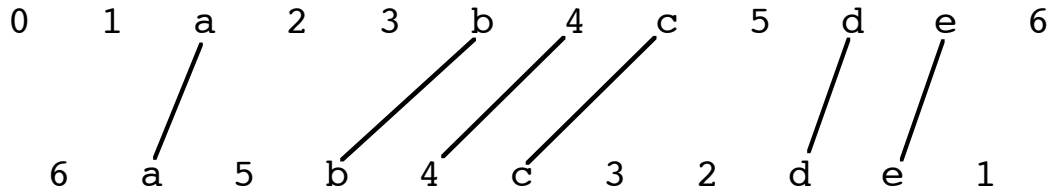
```
Compute c[3][6]
  k=3 gives cost 550=c[3][3]+c[4][6]+p[2]*p[3]*p[6]
  k=4 gives cost 185=c[3][4]+c[5][6]+p[2]*p[4]*p[6]
  k=5 gives cost 540=c[3][5]+c[6][6]+p[2]*p[5]*p[6]
  c[3][6]==185,trace[3][6]==4
Compute c[4][7]
  k=4 gives cost  95=c[4][4]+c[5][7]+p[3]*p[4]*p[7]
  k=5 gives cost 250=c[4][5]+c[6][7]+p[3]*p[5]*p[7]
  k=6 gives cost 250=c[4][6]+c[7][7]+p[3]*p[6]*p[7]
  c[4][7]==95,trace[4][7]==4
Compute c[1][5]
  k=1 gives cost 178=c[1][1]+c[2][5]+p[0]*p[1]*p[5]
  k=2 gives cost 198=c[1][2]+c[3][5]+p[0]*p[2]*p[5]
  k=3 gives cost 158=c[1][3]+c[4][5]+p[0]*p[3]*p[5]
  k=4 gives cost 118=c[1][4]+c[5][5]+p[0]*p[4]*p[5]
  c[1][5]==118,trace[1][5]==4
Compute c[2][6]
  k=2 gives cost 815=c[2][2]+c[3][6]+p[1]*p[2]*p[6]
  k=3 gives cost 765=c[2][3]+c[4][6]+p[1]*p[3]*p[6]
  k=4 gives cost 228=c[2][4]+c[5][6]+p[1]*p[4]*p[6]
  k=5 gives cost 493=c[2][5]+c[6][6]+p[1]*p[5]*p[6]
  c[2][6]==228,trace[2][6]==4
Compute c[3][7]
  k=3 gives cost 230=c[3][3]+c[4][7]+p[2]*p[3]*p[7]
  k=4 gives cost 152=c[3][4]+c[5][7]+p[2]*p[4]*p[7]
  k=5 gives cost 375=c[3][5]+c[6][7]+p[2]*p[5]*p[7]
  k=6 gives cost 455=c[3][6]+c[7][7]+p[2]*p[6]*p[7]
  c[3][7]==152,trace[3][7]==4
Compute c[1][6]
  k=1 gives cost 298=c[1][1]+c[2][6]+p[0]*p[1]*p[6]
  k=2 gives cost 338=c[1][2]+c[3][6]+p[0]*p[2]*p[6]
  k=3 gives cost 258=c[1][3]+c[4][6]+p[0]*p[3]*p[6]
  k=4 gives cost 173=c[1][4]+c[5][6]+p[0]*p[4]*p[6]
  k=5 gives cost 168=c[1][5]+c[6][6]+p[0]*p[5]*p[6]
  c[1][6]==168,trace[1][6]==5
Compute c[2][7]
  k=2 gives cost 341=c[2][2]+c[3][7]+p[1]*p[2]*p[7]
  k=3 gives cost 515=c[2][3]+c[4][7]+p[1]*p[3]*p[7]
  k=4 gives cost 209=c[2][4]+c[5][7]+p[1]*p[4]*p[7]
  k=5 gives cost 398=c[2][5]+c[6][7]+p[1]*p[5]*p[7]
  k=6 gives cost 438=c[2][6]+c[7][7]+p[1]*p[6]*p[7]
  c[2][7]==209,trace[2][7]==4
Compute c[1][7]
  k=1 gives cost 230=c[1][1]+c[2][7]+p[0]*p[1]*p[7]
  k=2 gives cost 242=c[1][2]+c[3][7]+p[0]*p[2]*p[7]
  k=3 gives cost 218=c[1][3]+c[4][7]+p[0]*p[3]*p[7]
  k=4 gives cost 196=c[1][4]+c[5][7]+p[0]*p[4]*p[7]
  k=5 gives cost 283=c[1][5]+c[6][7]+p[0]*p[5]*p[7]
  k=6 gives cost 198=c[1][6]+c[7][7]+p[0]*p[6]*p[7]
  c[1][7]==196,trace[1][7]==4
```

	1		2		3		4		5		6		7	
1	0	0	63	1	108	2	113	3	118	4	168	5	196	4
2	-------		0	0	315	2	108	2	143	4	228	4	209	4
3	-------	-------			0	0	45	3	90	4	185	4	152	4
4	-------	-------	-------				0	0	25	4	100	4	95	4
5	-------	-------	-------	-------					0	0	50	5	80	6
6	-------	-------	-------	-------	-------						0	0	150	6
7	-------	-------	-------	-------	-------	-------							0	0



LONGEST COMMON SUBSEQUENCE (not substring)

Has important applications in genetics research.



1. Describe problem input.

   Two sequences:

   $$X = x_1 x_2 \ldots x_m$$
   $$Y = y_1 y_2 \ldots y_n$$

2. Determine cost function and base case.

   $C(i, j)$ = length of LCS for $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$

   $C(i, j) = 0$ if $i = 0$ or $j = 0$

3. Determine general case.

   Suppose $C(i,j)$ has

   $$x_1 x_2 \ldots x_{i-1} A \qquad y_1 y_2 \ldots y_{j-1} A$$

   Since $x_i = y_j$, $C(i,j) = C(i-1,j-1) + 1$

   Now suppose $x_i \neq y_j$:

   $$x_1 x_2 \ldots x_{i-1} A \qquad y_1 y_2 \ldots y_{j-1} B$$

   But 'B' may appear in $x_1 x_2 \ldots x_{i-1}$ or 'A' may appear in $y_1 y_2 \ldots y_{j-1}$:

   $$C(i,j) = \max\{C(i,j-1), C(i-1,j)\} \text{ if } x_i \neq y_j$$

4. Appropriate ordering of subproblems.

   Before computing $C(i,j)$, must have $C(i-1,j-1)$, $C(i,j-1)$, and $C(i-1,j)$ available.

   Use $(m + 1) \times (n + 1)$ matrix to store C values.

5. Backtrace for solution – either explictly save indication of which of the three cases was used or recheck C values.

Takes $\Theta(mn)$ time. (Aside: Can be done using $\Theta(m+n)$ space.)

Example:

```
ababab
aabbaa
LCS is abaa, length==4
      a  a  b  b  a  a
   0  0  0  0  0  0  0
a  0  1  1  1  1  1  1
b  0  1  1  2  2  2  2
a  0  1  2  2  2  3  3
b  0  1  2  3  3  3  3
a  0  1  2  3  3  4  4
b  0  1  2  3  4  4  4
```