CSE 3318 Notes 7: Dynamic Programming

(Last updated 8/2/22 9:06 AM)

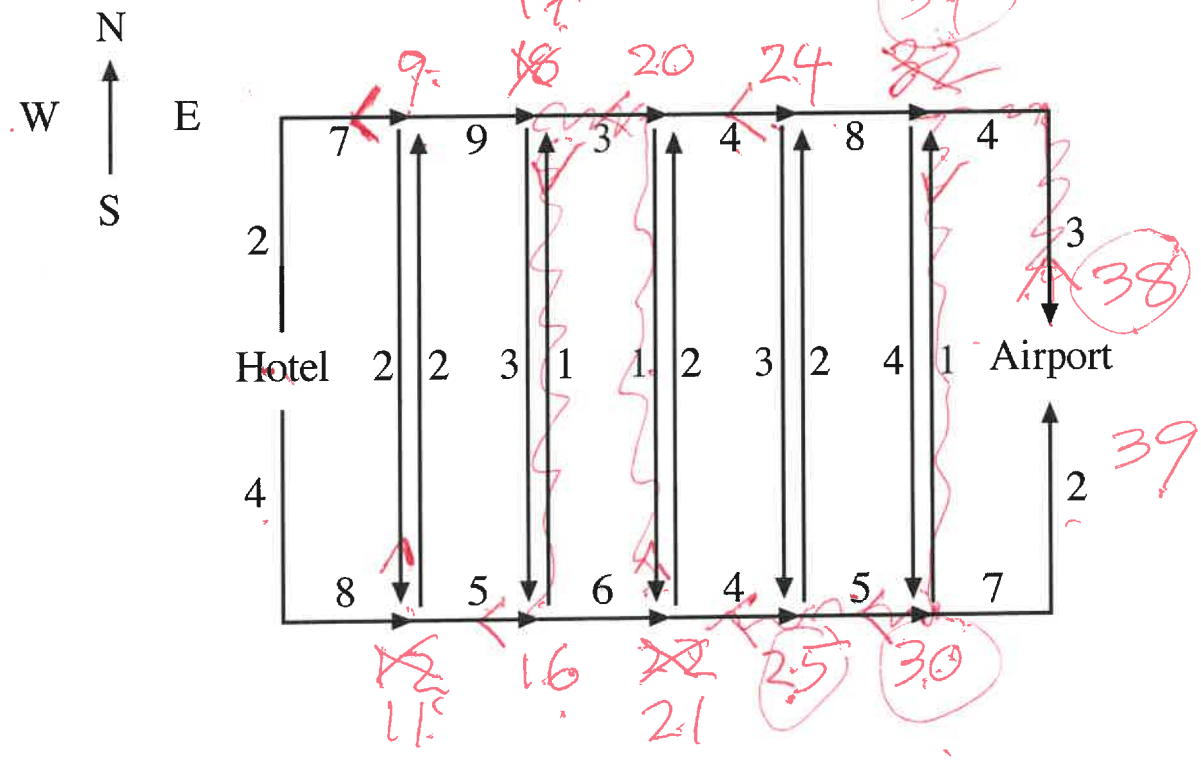
CLRS 14.1-14.4

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 for enumerating subproblems.
   a. Simple bottom-up approach - from small problems towards the entire big problem.
   b. Top-down approach with "memoization" - to attack large problems.
5. Backtrace for solution. *Most of the effort in dynamic programming is ignored at the end.*
   a. Predecessor/back pointers to get to the subproblems whose results are in the solution.
   b. Top-down recomputation of cost function (to reach the same subproblems as 5.a)

(Providing all solutions is an extra cost feature . . .)

7.A. A SMALL EXAMPLE – Shuttle-to-Airport



How many different paths (by brute force)?

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

(Could also use Dijkstra's algorithm, Notes 15, which is more general, but slower.)

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

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

   a. (`https://ranger.uta.edu/~weems/NOTES3318/shuttle1.c`) *explicitly save* indication of which of the two cases was used (continue - c, switch - s), or

   b. (`https://ranger.uta.edu/~weems/NOTES3318/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 (c)

Dynamic programming is:

1. Exhaustive search without brute force.
2. Optimal solution to big problem from optimal solutions to subproblems.

7.B. WEIGHTED INTERVAL SCHEDULING

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.

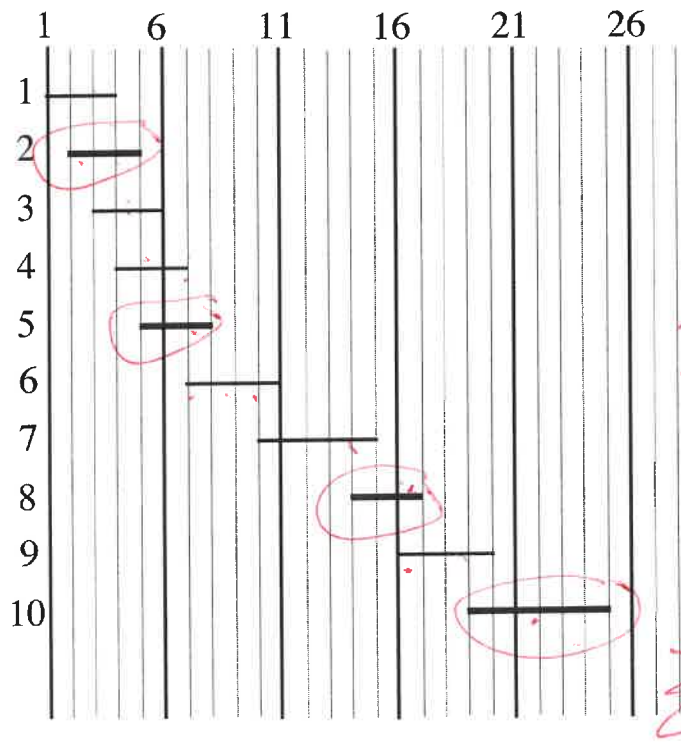
Output: A set of non-overlapping intervals to ***maximize*** the sum of their weights. (Two intervals i and j overlap if either $s_i < s_j < f_i$ or $s_i < f_j < f_i$.)

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. (`https://ranger.uta.edu/~weems/NOTES3318/wis.power.c`)

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 using `binSearchLast()` from Notes 01. See `https://ranger.uta.edu/~weems/NOTES3318/wis.bs.c`)



	v	p	m
1	4	0	4
2	5	0	5
3	6	0	6
4	3	1	7
5	5	2	10
6	1	4	10
7	3	5	13
8	4	6	14
9	2	7	15
10	5	8	19

$\Sigma = 19$

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 $M(i) = M(i-1)$.

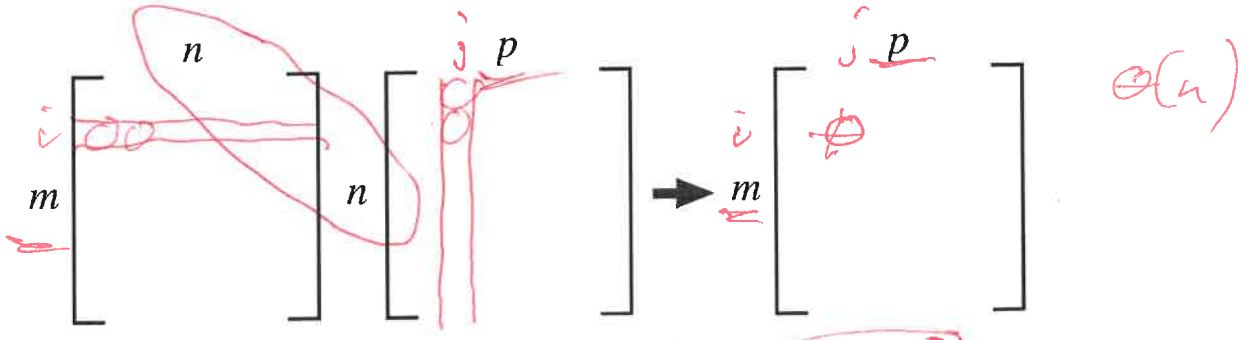
   This observation tells us to compute cost ***both*** ways and keep the maximum.

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

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

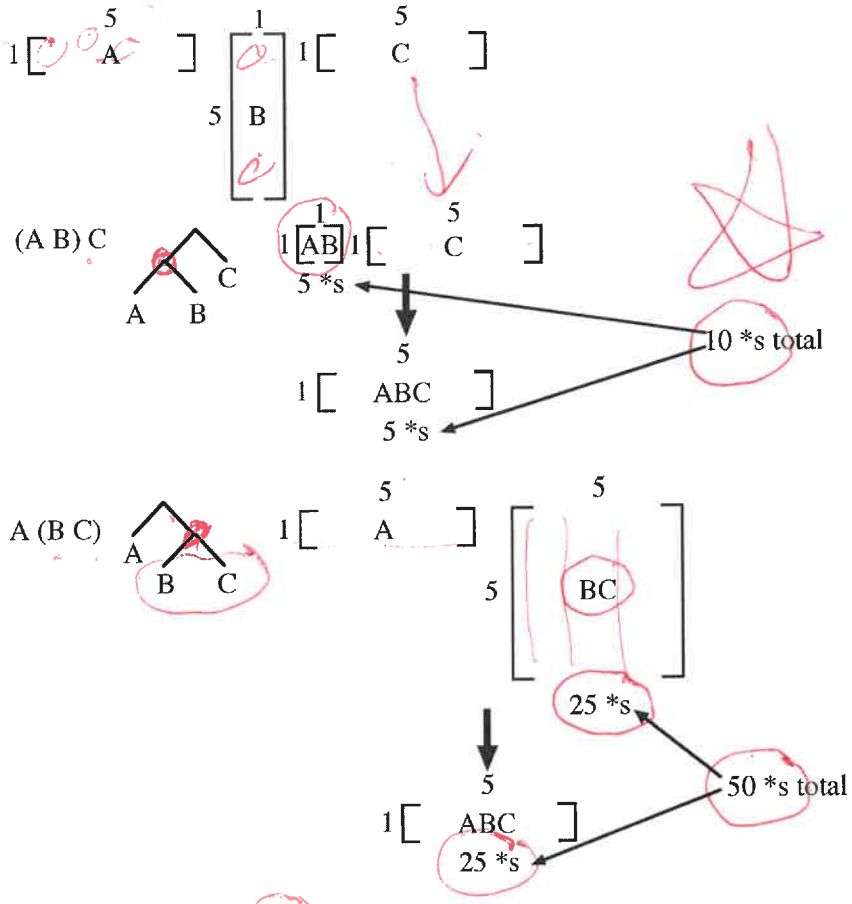
```
i=n;
while (i>0)
  if (v[i]+M[p[i]]>=M[i-1])
  {
    // Interval i is in solution
    i=p[i];
  }
  else
    i--;
```

7.C. 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:



Aside: Ways to parenthesize n matrices? (Catalan numbers)

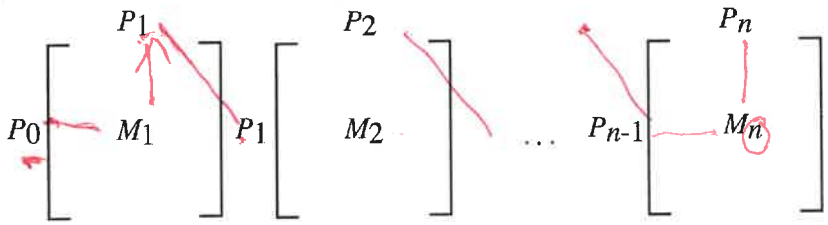
$$C_0 = 1 \qquad C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} \text{ for } n \geq 0 \qquad C_n = \frac{1}{n+1}\binom{2n}{n}$$

(`https://en.wikipedia.org/wiki/Catalan_number`)

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

1. Describe problem input.

   n matrices $\Rightarrow n + 1$ sizes



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 $C(i, j)$

P_k P_j

P_{i-1} $\left[\; C(i, k) \;\right]$ P_k $\left[\; C(k+1, j) \;\right]$

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

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 $n - 1$. Suppose $n = 5$:

0	1	2	3	4
$C(1,1)$	$C(1,2)$	$C(1,3)$	$C(1,4)$	$C(1,5)$
$C(2,2)$	$C(2,3)$	$C(2,4)$	$C(2,5)$	
$C(3,3)$	$C(3,4)$	$C(3,5)$		
$C(4,4)$	$C(4,5)$			
$C(5,5)$				

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

https://ranger.uta.edu/~weems/NOTES3318/optMM.c

```
// Optimal matrix multiplication order using dynamic programming

#include <stdio.h>

int p[20];
int n;
int c[20][20];
int trace[20][20];

void tree(int left,int right,int indent)
{
int i;

if (left==right)
{
  for (i=0;i<indent;i++)
    printf("    ");
  printf("%d\n",left);
  return;
}
tree(trace[left][right]+1,right,indent+1);
for (i=0;i<indent;i++)
  printf("    ");
printf("%d %d\n",left,right);
tree(left,trace[left][right],indent+1);
}

int main()
{
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]);
  }
```

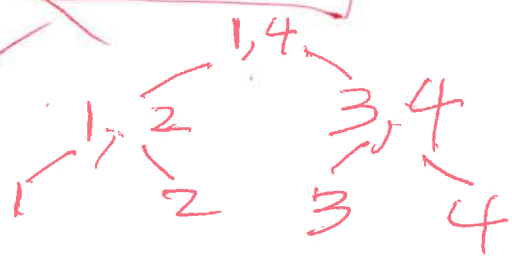
```
printf("    ");
for (i=1;i<=n;i++)
  printf("   %3d    ",i);
printf("\n");
for (i=1;i<=n;i++)
{
  printf("%2d ",i);
  for (j=1;j<=n;j++)
    if (i>j)
      printf(" ------- ");
    else
      printf(" %3d %3d ",c[i][j],trace[i][j]);
  printf("\n");

  printf("\n");
}

tree(1,n,0);
}
```

It is straightforward to use integration to determine that the k loop body executes about $\frac{n^3}{6}$ times.

```
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		

```
      4
    3 4
      3
1, 4
      2
    1 2
      1
```



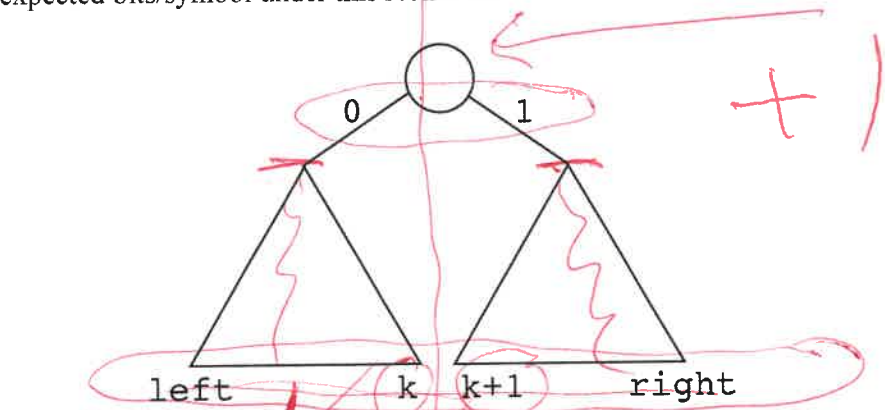

```
7
1 7 9 5 1 5 10 3
Compute c[1][2]
  k=1 gives cost  63=c[1][1]+c[2][2]+p[0]*p[1]*p[2]
  c[1][2]==63,trace[1][2]==1
Compute c[2][3]
  k=2 gives cost 315=c[2][2]+c[3][3]+p[1]*p[2]*p[3]
  c[2][3]==315,trace[2][3]==2
Compute c[3][4]
  k=3 gives cost  45=c[3][3]+c[4][4]+p[2]*p[3]*p[4]
  c[3][4]==45,trace[3][4]==3
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

```
      7
    5 7
        6
      5 6
        5
1 7
      4
    1 4
        3
      1 3
          2
        1 2
            1
```

(Aside) Like optimal matrix multiplication, the *order-preserving* Huffman code problem mentioned in Notes 06 requires a solution with the leaves in alphabetic order. The cost function is based on minimizing the expected bits/symbol under this restriction:

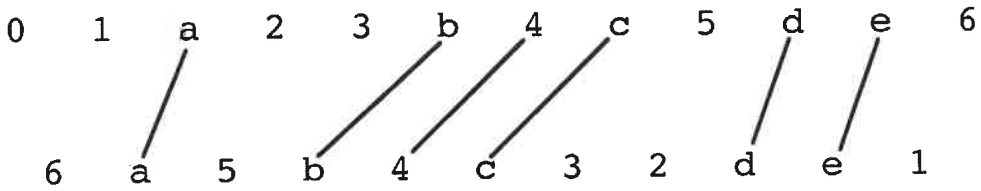


$$C(left,right) = \sum_{i=left}^{right} P_i + \min_{left \le k < right} \{C(left,k) + C(k+1,right)\}$$

$$C(i,i) = 0$$

7.D. LONGEST COMMON SUBSEQUENCE (not substring, https://ranger.uta.edu/~weems/NOTES3318/LCS.c)

Has important applications in genetics research.



1. Describe problem input.

   Two sequences:

   $X = x_1 x_2 \ldots x_m$

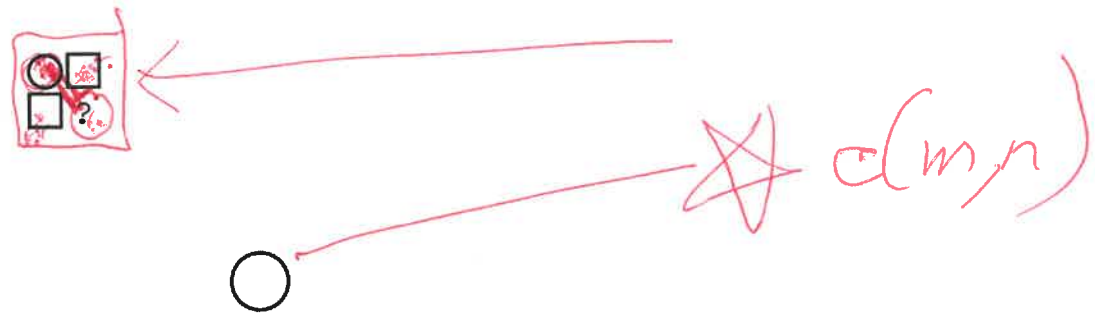
   $Y = y_1 y_2 \ldots y_n$

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 $j = 0$

			y_1	y_2		y_j		y_n
	0	1	2	...	j	...	n
0	0	0	0	...	0	...	0
x_1 1	0						
x_2 2	0						
.	.						
.	.						
.	.						
x_i i	0				?		
.	.						
.	.						
.	.						
x_m m	0						



3. Determine general case.

Suppose $C(i,j)$ has

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

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

Now suppose $x_i \neq y_j$:

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

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

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

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.)

7.F. SUBSET SUM (https://ranger.uta.edu/~weems/NOTES3318/subsetSum.c)

Given a "set" of n positive integer values, find a subset whose sum is a value m.

Optimization?

Generating subsets (combinations) would take exponential time (in n).

1. Describe problem input. Array $S = S_1, S_2, \ldots, S_n$ and m.

2. Determine cost function and base case.

   $C(i)$ = Smallest index j such that there is some combination of $S_1, S_2, \ldots, S_j$, that includes S_j and sums to i.

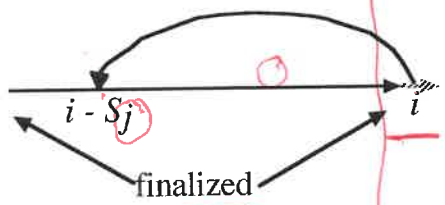
   $C(0) = 0$ (Will assume that $S_0 = 0$)

3. Determine general case for cost function.

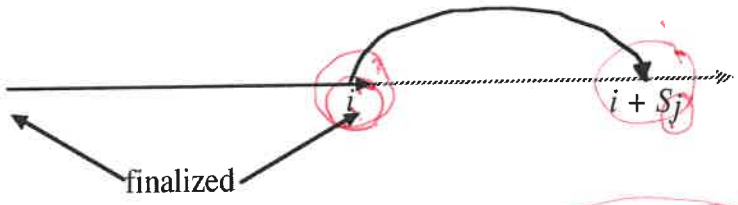
$$C(i) = \min_{j \text{ s.t. } C(i - S_j) \text{ is defined and } C(i - S_j) < j} \{j\}$$

4. Appropriate ordering of subproblems:

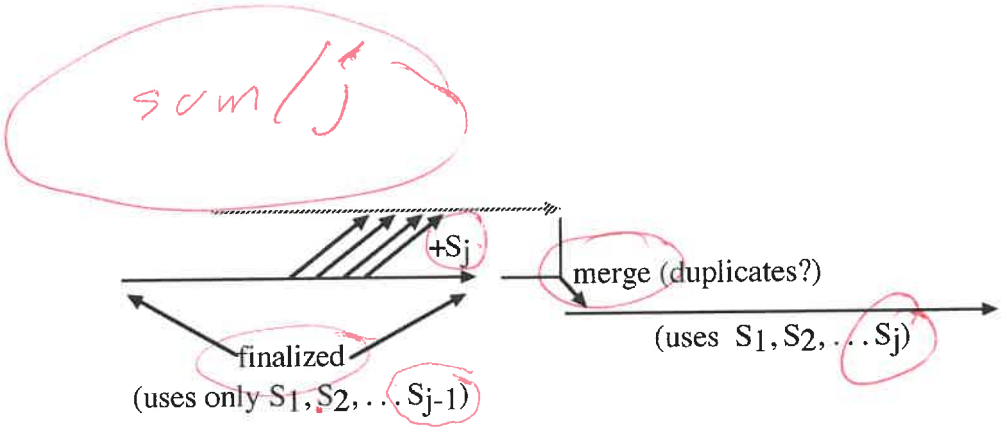
   a. Iterate over i looking backwards (like the cost function) to previous "finalized" solutions.



   b. (Aside, Dijkstra's algorithm-like) Iterate over finalized $C(i)$ to compute $i + S_j$ for each $j > C(i)$ and attempt update forward. After updates, $C(i+1)$ has final value.



   c. (Aside) Maintain ordered list of finalized solutions from using $S_1, S_2, \ldots, S_{j-1}$ and generate new ordered list that also uses S_j to reach some new values.



5. Backtrace for solution - if $C(m)$ is defined, then backtrace using C values to subtract out each value in subset. (Indices will appear in strictly decreasing order during backtrace.)

```
// Initialize table for DP
C[0]=0;  // DP base case
// For each potential sum, determine the smallest index such
// that its input value is in a subset to achieve that sum.
for (potentialSum=1; potentialSum<=m; potentialSum ++)
{
  for (j=1;j<=n;j++)
  {
    leftover=potentialSum-S[j];      // To be achieved with other values
    if (leftover>=0 &&               // Possible to have a solution
      C[leftover]<j)                 // Indices are included in
      break;                         // ascending order.
  }
  C[potentialSum]=j;
}

if (C[m]==n+1)
  printf("No solution\n");
else
{
  printf("Solution\n");
  printf("  i   S\n");
  printf("-------\n");
  for (i=m;i>0;i-=S[C[i]])
    printf("%3d %3d\n",C[i],S[C[i]]);
}
```

Example: $m = 12, n = 4$

i	0	1	2	3	4
S_i	0	3	6	7	9

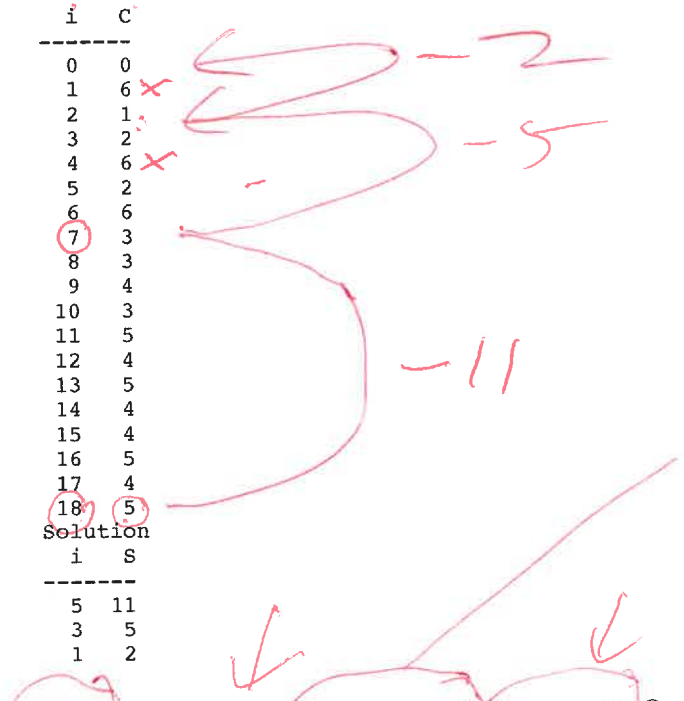
[The S_i values do not require ordering.]

i	0	1	2	3	4	5	6	7	8	9	10	11	12
C_i	0	5	5	1	5	5	2	3	5	2	3	5	4

Time is $\Theta(mn)$. Space is $\Theta(m)$. [What happens if m and each S_i are multiplied by the same constant?]

Another subset sum example (`https://ranger.uta.edu/~weems/NOTES3318/subsetSum.c`)

```
5 18
2 3 5 7 11
  i   S
-------
  0   0
  1   2
  2   3
  3   5
  4   7
  5  11
  i   C
-------
  0   0
  1   6
  2   1
  3   2
  4   6
  5   2
  6   6
  7   3
  8   3
  9   4
 10   3
 11   5
 12   4
 13   5
 14   4
 15   4
 16   5
 17   4
 18   5
Solution
  i   S
-------
  5  11
  3   5
  1   2
```



7.G. 0/1 (INDIVISIBLE, UNBOUNDED) KNAPSACK - OPTIMAL SOLUTION

(`https://ranger.uta.edu/~weems/NOTES3318/knapsackTypeRS.c`)

CLRS 14.1 calls this the *rod-cutting problem*.

n item *types*, each with an integer size and value (CLRS - type = rod, size = length, value = price).

Unlike conventional version (Notes 06), *unlimited* supply of each type.
m, the integer capacity of the knapsack (length of the longer rod to be cut)

Goal: Select a combination from the unlimited supply of items that

1) maximizes the sum of the values, and
2) the sum of the sizes does not exceed m.

1. Describe problem input. Array *size* of n weights, array *val* of n values, and m.

2. Determine cost function and base case.

$maxKnown(i)$ = Maximum sum of values achievable by some combination of items whose weights sum to no more than i.

$maxKnown(0) = 0$

3. Determine general case for cost function.

$$maxKnown(i) = \max_{k \text{ s.t. } maxKnown(i-size_k) \text{ is defined}} \{maxKnown(i - size_k) + val_k\}$$

4. Appropriate ordering of subproblems - since goal is to compute $maxKnown(m)$, extra cases could be computed. Use array of $maxKnown(i)$ values along with *unknown* indicator to implement *memoization* (top-down).

```
// From Sedgewick
int knap(int M, int level)
{ int i, space, max, maxi = 0, t;
  for (i=0;i<level;i++)
    printf(".");
  if (maxKnown[M] != unknown)
  {
    printf("Reusing knap(%d)=%d\n",M,maxKnown[M]);
    return maxKnown[M];
  }
  printf("Start knap(%d)\n",M);
  for (i = 0, max = 0; i < N; i++)
    if ((space = M-items[i].size) >= 0)
      if ((t = knap(space,level+1) + items[i].val) > max)
        { max = t; maxi = i; }
  maxKnown[M] = max; itemKnown[M] = items[maxi];
  for (i=0;i<level;i++)
    printf(".");
  printf("Finish knap(%d)\n",M);
  return max;       }
. . .
// Since knap() uses memoization, a bottom-up loop is not needed.
printf("Maximum for %d is %d\n",m,knap(m,0));
```

5. Backtrace for solution - backtrace using *maxKnown* and *itemKnown*.

Example: m=46

i	0	1	2	3
size	11	13	17	19
val	10	14	16	20

i	11	12	13	14	16	18	20	22	24	27	29	33	35	46
maxKnown	10	10	14	14	14	16	20	20	24	28	28	34	34	48
size	11	11	13	13	13	17	19	11	11	13	13	13	11	13
val	10	10	14	14	14	16	20	10	10	14	14	14	10	14

Time is $\Theta(mn)$. Space is $\Theta(m)$.

```
Start knap(46)
.Start knap(35)
..Start knap(24)
...Start knap(13)
....Start knap(2)
....Finish knap(2)
....Start knap(0)
....Finish knap(0)
...Finish knap(13)
...Start knap(11)
....Reusing knap(0)=0
...Finish knap(11)
...Start knap(7)
...Finish knap(7)
...Start knap(5)
...Finish knap(5)
..Finish knap(24)
..Start knap(22)
...Reusing knap(11)=10
...Start knap(9)
...Finish knap(9)
...Reusing knap(5)=0
...Start knap(3)
...Finish knap(3)
..Finish knap(22)
..Start knap(18)
...Reusing knap(7)=0
...Reusing knap(5)=0
...Start knap(1)
...Finish knap(1)
..Finish knap(18)
..Start knap(16)
...Reusing knap(5)=0
...Reusing knap(3)=0
..Finish knap(16)
.Finish knap(35)
.Start knap(33)
..Reusing knap(22)=20
..Start knap(20)
...Reusing knap(9)=0
...Reusing knap(7)=0
...Reusing knap(3)=0
...Reusing knap(1)=0
..Finish knap(20)
..Reusing knap(16)=14
..Start knap(14)
...Reusing knap(3)=0
...Reusing knap(1)=0
..Finish knap(14)
.Finish knap(33)
.Start knap(29)
..Reusing knap(18)=16
..Reusing knap(16)=14
..Start knap(12)
...Reusing knap(1)=0
..Finish knap(12)
..Start knap(10)
..Finish knap(10)
.Finish knap(29)
.Start knap(27)
..Reusing knap(16)=14
..Reusing knap(14)=14
..Reusing knap(10)=0
..Start knap(8)
..Finish knap(8)
.Finish knap(27)
Finish knap(46)
Maximum for 46 is 48
  i size val
------------
  0   11  10
  1   13  14
  2   17  16
  3   19  20
  i max size val
----------------
 11  10   11  10
 12  10   11  10
 13  14   13  14
 14  14   13  14
 16  14   13  14
 18  16   17  16
 20  20   19  20
 22  20   11  10
 24  24   11  10
 27  28   13  14
 29  28   13  14
 33  34   13  14
 35  34   11  10
 46  48   13  14
Solution has value 48:
  i size  val
-------------
  1   13   14
  2   13   14
  3   19   20
Unused capacity 1
```