CSE 2320 Notes 7: Dynamic Programming

(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 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 16, 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, \infty
Lower-to-Upper = LU = lu_1 lu_2 \ldots lu_n = 2, 1, 2, 2, 1, \infty

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. (http://ranger.uta.edu/~weems/NOTES2320/shuttle1.c ) explicitly save indication of which of the two cases was used (continue - c, switch - s), or

b. (http://ranger.uta.edu/~weems/NOTES2320/shuttle2.c ) recheck during backtrace for which case was used.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>0</td>
<td>9  (c)</td>
<td>17 (s)</td>
<td>20 (c)</td>
<td>24 (c)</td>
<td>31 (s)</td>
<td>38 (c)</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>11 (s)</td>
<td>16 (c)</td>
<td>21 (s)</td>
<td>25 (c)</td>
<td>30 (c)</td>
<td>39 (c)</td>
</tr>
</tbody>
</table>
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. (http://ranger.uta.edu/~weems/NOTES2320/wis.power.c)

1. Describe problem input.

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

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 1. See http://ranger.uta.edu/~weems/NOTES2320/wis.bs.c)

\[
\begin{array}{cccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
1 & 6 & 11 & 16 & 21 & 26 & & & & & \\
& 5 & 0 & 4 & 0 & 3 & 2 & 2 & 1 & 1 & 5 & 0 & 4 & 5 & 3 & 7 & 2 & 6 & 1 & 8 \\
\end{array}
\]
2. Determine cost function and base case.

\[ M(i) = \text{Cost for optimal non-overlapping subset for the first } i \text{ input intervals.} \]

\[ M(0) = 0 \]

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 \textit{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 \( M(n) \).

```plaintext
def backtrack(M, n):
    i = n
    while i > 0:
        if (v[i] + M[p[i]] >= M[i - 1]):
            i = p[i]
        else:
            i -= 1
```

7.C. \textbf{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:

\[
\begin{align*}
(A \ B) \ C & \quad \text{10 \ast s total} \\
A \ (B \ C) & \quad \text{50 \ast s total}
\end{align*}
\]

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

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

(http://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

\[
\begin{pmatrix}
P_0 & \ldots & P_n \\
M_1 & \ldots & M_n
\end{pmatrix}
\]
2. Determine cost function and base case.

\[ C(i, j) = \text{Cost for optimally multiplying } M_i \ldots M_j \]

\[ C(i, i) = 0 \]

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:

\[ C(5, 5) + C(6, 9) + P_4 P_5 P_9 \]
\[ C(5, 6) + C(7, 9) + P_4 P_6 P_9 \]
\[ C(5, 7) + C(8, 9) + P_4 P_7 P_9 \]
\[ C(5, 8) + C(9, 9) + P_4 P_8 P_9 \]

Compute all four and keep the smallest one.

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

\[
P_{i-1} \begin{bmatrix} P_k & P_j \\ C(i, k) & P_k & C(k+1, j) \end{bmatrix}
\]

\[ C(i, j) = \min_{i \leq k < j} \left\{ C(i, k) + C(k + 1, j) + P_{i-1} P_k P_j \right\} \]

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 \( n = 5 \):

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C(1,1) )</td>
<td>( C(1,2) )</td>
<td>( C(1,3) )</td>
<td>( C(1,4) )</td>
<td>( C(1,5) )</td>
<td></td>
</tr>
<tr>
<td>( C(2,2) )</td>
<td>( C(2,3) )</td>
<td>( C(2,4) )</td>
<td>( C(2,5) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( C(3,3) )</td>
<td>( C(3,4) )</td>
<td>( C(3,5) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( C(4,4) )</td>
<td>( C(4,5) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( C(5,5) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. Backtrace for solution – explicitly save the \( k \) value that gave each \( C(i, j) \).
// 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
",left);
        return;
    }
    tree(trace[left][right]+1,right,indent+1);
    for (i=0;i<indent;i++)
        printf("   ");
    printf("%d %d
",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]=9999999;
            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]);
        }
}

http://ranger.uta.edu/~weems/NOTES2320/optMM.c
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");
}
tree(1,n,0);
}

It is straightforward to use integration to determine that the k loop body executes about $\frac{2^k}{6}$ times.

\[
\begin{array}{cccc}
  4 & 2 & 4 & 3 & 5 & 2 \\
  c[1][2] = 24, \text{trace}[1][2] = 1 \\
  c[2][3] = 60, \text{trace}[2][3] = 2 \\
  c[3][4] = 30, \text{trace}[3][4] = 3 \\
  c[1][3] = 54, \text{trace}[1][3] = 2 \\
  c[2][4] = 100, \text{trace}[2][4] = 2 \\
  \\
  7 & 1 & 3 & 9 & 5 & 1 & 5 & 10 & 3 \\
  c[1][2] = 63, \text{trace}[1][2] = 1 \\
  c[2][3] = 315, \text{trace}[2][3] = 2 \\
  c[3][4] = 45, \text{trace}[3][4] = 3 \\
  c[4][5] = 25, \text{trace}[4][5] = 4 \\
  c[5][6] = 50, \text{trace}[5][6] = 5 \\
  c[6][7] = 150, \text{trace}[6][7] = 6 \\
  c[1][3] = 108, \text{trace}[1][3] = 2 \\
  c[2][4] = 108, \text{trace}[2][4] = 2 \\
  c[2][4] = 108, \text{trace}[2][4] = 2 \\
  \\
  1 & 7 & 2 & 9 & 5 & 1 & 5 & 10 & 3 \\
  c[1][2] = 63, \text{trace}[1][2] = 1 \\
  c[2][3] = 315, \text{trace}[2][3] = 2 \\
  c[3][4] = 45, \text{trace}[3][4] = 3 \\
  c[4][5] = 25, \text{trace}[4][5] = 4 \\
  c[5][6] = 50, \text{trace}[5][6] = 5 \\
  c[6][7] = 150, \text{trace}[6][7] = 6 \\
  c[1][3] = 108, \text{trace}[1][3] = 2 \\
  c[2][4] = 108, \text{trace}[2][4] = 2 \\
  c[2][4] = 108, \text{trace}[2][4] = 2 \\
\end{array}
\]
(Aside) Like optimal matrix multiplication, the *order-preserving* Huffman code problem mentioned in Notes 06 requires a solution with the leaves ordered (according to an alphabet). The cost function is based on minimizing the expected bits/symbol under this restriction:

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

$$C(i, i) = 0$$
7.D. LONGEST COMMON SUBSEQUENCE (not substring, http://ranger.uta.edu/~weems/NOTES2320/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.

\[ C(i, j) = \text{length of LCS for } x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_j \]
\[ C(i, j) = 0 \text{ if } i = 0 \text{ or } j = 0 \]

3. Determine general case.

Suppose \( C(i, j) \) has

\[ x_1 x_2 \ldots x_{i-1} A \quad y_1 y_2 \ldots y_{j-1} A \]

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

Now suppose \( x_i \neq y_j \):

\[ x_1 x_2 \ldots x_{i-1} B \quad y_1 y_2 \ldots y_{j-1} B \]

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 explicitly 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
a 0 1 1 1 1 1
b 0 1 1 2 2 2
a 0 1 2 2 2 3
b 0 1 2 3 3 3
a 0 1 2 3 3 4
b 0 1 2 3 4 4
```

7.E. LONGEST INCREASING SUBSEQUENCE

Monotone: For an input sequence $Y = y_1 y_2 \ldots y_n$, find a longest subsequence in increasing ($\leq$) order.

Strict: For an input sequence $Y = y_1 y_2 \ldots y_n$, find a longest subsequence in strictly increasing ($<$) order.

Both versions may be solved inefficiently by reduction to LCS:

Monotone: $\Theta(n^2)$ worst-case time by taking LCS of sequence and its elements sorted in ascending order.

```
1122346778
6178213472
LCS is 11347, length==5
```

```
6 1 7 8 2 1 3 4 7 2
0 0 0 0 0 0 0 0 0 0
1 0 0 1 1 1 1 1 1 1 1 1
1 0 0 1 1 1 1 1 2 2 2 2 2
2 0 0 1 1 1 2 2 2 2 2 2 2 3
2 0 0 1 1 1 2 2 2 2 2 2 3
3 0 0 1 1 1 2 2 3 3 3 3 3
4 0 0 1 1 1 2 2 3 4 4 4 4 4
6 0 1 1 1 1 2 2 3 4 4 4 4 4
7 0 1 1 2 2 2 2 3 4 5 5 5 5
7 0 1 1 2 2 2 2 3 4 5 5 5 5
8 0 1 1 2 3 3 3 3 4 5 5 5 5
```
Strict: \(\Theta(mn)\) worst-case time, where \(m\) is the number of unique integers occurring in input.

Both versions may be solved in \(\Theta(n \log n)\) worst-case time, using an appropriate DP cost function and \(n\) binary searches.

Monotone (http://ranger.uta.edu/~weems/NOTES2320/LIS.c):

1. Describe problem input. \(Y = y_1y_2 \ldots y_n\)

2. Determine cost function and base case.

\[C(i) = \text{Length of longest increasing subsequence ending with } y_i.\]

\[C(0) = 0\]

3. Determine general case for cost function.

\[C(i) = 1 + \max_{j<i \text{ and } y_j \leq y_i} \{C(j)\} \quad (\text{The } j \text{ that gives } C(i) \text{ may be saved for backtrace.})\]

4. Appropriate ordering of subproblems - iterate over the prefix length, saving \(C\) and \(j\) for each \(i\).

<table>
<thead>
<tr>
<th>(i)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y_i)</td>
<td>60</td>
<td>10</td>
<td>70</td>
<td>80</td>
<td>20</td>
<td>10</td>
<td>30</td>
<td>40</td>
<td>70</td>
<td>20</td>
</tr>
<tr>
<td>(C)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>(j)</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

5. Backtrace for solution.

Find the rightmost occurrence of the maximum \(C\) value. The corresponding \(y\) will be minimized.

Appears to take \(\Theta(n^2)\), but \texttt{binSearchLast()} from Notes 1 may be used to find each \(C\) and \(j\) pair in \(\Theta(n \log n)\) time to give \(\Theta(n \log n)\) overall:
// Initialize table for binary search for DP
bsTabC[0]=-999999; // Must be smaller than all input values.
bsTabI[0]=0; // Index of predecessor (0=grounded)
for (i=1;i<=n;i++)
    bsTabC[i]=999999; // Must be larger than all input values.

C[0]=0; // DP base case
j[0]=0;

for (i=1;i<=n;i++)
{
    // Find IS that y[i] could be appended to.
    // See CSE 2320 Notes 01 for binSearchLast()
    k=binSearchLast(bsTabC,n+1,y[i]);
    C[i]=k+1; // Save length of LIS for y[i]
    j[i]=bsTabI[k]; // Predecessor of y[i]
    bsTabC[k+1]=y[i]; // Decrease value for this length IS
    bsTabI[k+1]=i;
}

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>y_i</td>
<td>60</td>
<td>10</td>
<td>70</td>
<td>80</td>
<td>20</td>
<td>10</td>
<td>30</td>
<td>40</td>
<td>70</td>
<td>20</td>
</tr>
</tbody>
</table>

C

j

1

2

3

4

5

Strict (http://ranger.uta.edu/~weems/NOTES2320/LSIS.c): Similar to monotone with the following exceptions:

2. Determine cost function and base case.

\[ C(i) = \text{Length of longest strictly increasing subsequence ending with } y_i. \]

\[ C(0) = 0 \]

3. Determine general case for cost function.

\[ C(i) = 1 + \max_{j < i, y_j < y_i} \{ C(j) \} \] (The j that gives \( C(i) \) must be saved to allow backtrace.)
Finally, any $y_i$ that is found by binSearchLast() will be ignored.

```c
for (i=1;i<=n;i++)
{
  // Find SIS that y[i] could be appended to.
  // See CSE 2320 Notes 01 for binSearchLast()
  k=binSearchLast(bsTabC,n+1,y[i]);
  // y[i] only matters if it is not already in table.
  if (bsTabC[k]<y[i]) {
    C[i]=k+1;         // Save length of LIS for y[i]
    j[i]=bsTabI[k];   // Predecessor of y[i]
    bsTabC[k+1]=y[i]; // Decrease value for this length IS
    bsTabI[k+1]=i;
  } else {
    C[i]=(-1);        // Mark as ignored
    j[i]=(-1);
  }
}
```

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_i$</td>
<td>60</td>
<td>10</td>
<td>70</td>
<td>80</td>
<td>20</td>
<td>10</td>
<td>30</td>
<td>40</td>
<td>70</td>
<td>20</td>
</tr>
</tbody>
</table>

7.F. **SUBSET SUM** (http://ranger.uta.edu/~weems/NOTES2320/subsetSum.c)

Given a “set” of $n$ positive integer values, find a subset whose sum adds to a value $m$.

**Optimization?**

Enumerating subsets (combinations) would take exponential time.

1. Describe problem input. Array $S = S_1, S_2, ..., S_n$ and $m$. 
2. Determine cost function and base case.

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

\[ C(0) = 0 \text{ (Will assume that } S_0 = 0) \]

3. Determine general case for cost function.

\[ C(i) = \min \{j\} \text{ s.t. } C(i-S_j) \text{ is defined and } C(i-S_j) < j \]

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.)

```c
// 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 &&
            C[leftover]<j)                 // Possible to have a solution
            break;                        // Indices are included in
                                            // 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)
        printf("%3d %3d\n",C[i],S[C[i]]);
}
```

Example: $m=12, n=4$

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_i$</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

[The $S_i$ values do not require ordering.]

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

7.G. 0/1 (INDIVISIBLE, UNBOUNDED) KNAPSACK - OPTIMAL SOLUTION
(http://ranger.uta.edu/~weems/NOTES2320/knapsackTypeRS.c)

CLRS 15.1 calls this the *rod-cutting problem*.

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

Unlike conventional version (Notes 6), *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 \( \text{size} \) of \( n \) weights, array \( \text{val} \) of \( n \) values, and \( m \).

2. Determine cost function and base case.

\[
\text{maxKnown}(i) = \text{Maximum sum of values achievable by some combination of items whose weights sum to no more than } i.
\]

\[
\text{maxKnown}(0) = 0
\]

3. Determine general case for cost function.

\[
\text{maxKnown}(i) = \max_{k \text{ s.t. } \text{maxKnown}(i-\text{size}_k) \text{ is defined}} \{\text{maxKnown}(i-\text{size}_k) + \text{val}_k\}
\]

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

```java
// From Sedgewick
static int knap(int M)
{
    int i, space, max, maxi = 0, t;
    if (maxKnown[M] != unknown) return maxKnown[M];
    for (i = 0, max = 0; i < N; i++)
        if ((space = M-items[i].size) >= 0)
            if (((t = knap(space) + items[i].val) > max)
                { max = t; maxi = i; }
            maxKnown[M] = max; itemKnown[M] = items[maxi];
    return max;
}
```

// Since knap() uses memoization, a bottom-up loop is not needed.
System.out.format("Maximum for %d is %d\n",m,knap(m));

5. Backtrace for solution - backtrace using \( \text{maxKnown} \) and \( \text{itemKnown} \).

Example: \( m=46 \)

\[
\begin{array}{cccccc}
  i & 0 & 1 & 2 & 3 & \\
  \text{size} & 11 & 13 & 17 & 19 & \\
  \text{val} & 10 & 14 & 16 & 20 & \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
  i & 11 & 12 & 13 & 14 & 16 & 18 & 20 & 22 & 24 & 27 & 29 & 33 & 35 & 46 & \\
  \text{maxKnown} & 10 & 10 & 14 & 14 & 14 & 16 & 20 & 20 & 24 & 28 & 28 & 34 & 34 & 48 & \\
  \text{val} & 10 & 10 & 14 & 14 & 14 & 16 & 20 & 10 & 10 & 14 & 14 & 14 & 14 & 10 & 14 & \\
\end{array}
\]

Time is \( \Theta(mn) \). Space is \( \Theta(m) \).
Start knap(46)
..Finish knap(33)
.Finish knap(29)
..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(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

Solution has value 48:

```
<table>
<thead>
<tr>
<th>i</th>
<th>size</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>
```

Unused capacity 1
7.H. The Parking Problem (aside, http://ranger.uta.edu/~weems/NOTES2320/parkingOff.oneBased.c)

Requirements:

Implement a $\Theta(kn)$-time, $\Theta(k + n)$-space dynamic programming algorithm to determine a minimum cost sequence of parking permits to cover $n$ not-necessarily-adjacent days you need to drive downtown. Each element of a solution sequence will be one of $k$ available permit types, each covering a different number of consecutive days at some cost.

The first line of the input will be positive integers for $k$ and $n$. $k \leq 10$ and $n \leq 100$. The next $k$ lines will be pairs of positive integers for the permit types. The two values of a pair will be the number of days and cost, respectively. Note that the $k$ pairs will appear in strictly increasing order for the number of days and likewise for the costs. (For example, nobody will spend $20 for a three-day permit if a four-day permit is just $15.) Each of the remaining $n$ lines will contain an integer corresponding to a day you must park. These values appear in strictly increasing order.

The output is 1) the table of subproblems, i.e. their cost and backtrace information, 2) the cost of the final solution, and 3) the sequence of permits needed and the range of days covered by each. The sequence may be output in reverse order. The last day for the last permit should be the last day you need to park. The start date for the first permit may be earlier than the first day you need to park.

Getting Started:

1. The left column gives an input instance and the right column gives a solution

<table>
<thead>
<tr>
<th>Day</th>
<th>Permit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

   DP table:
   - prefixCost[0]=0 permitTypeUsed[0]=0
   - prefixCost[1]=5 permitTypeUsed[1]=1
   - prefixCost[8]=28 permitTypeUsed[8]=1
   - prefixCost[10]=35 permitTypeUsed[10]=1

   Cost is 35

   Permits used are:
   - Permit type 1 cost 5 begin 15 end 15
   - Permit type 2 cost 7 begin 12 end 13
   - Permit type 3 cost 9 begin 7 end 9
   - Permit type 1 cost 5 begin 4 end 4
   - Permit type 3 cost 9 begin 1 end 3

2. This problem is slightly similar to the weighted interval scheduling problem. In particular, an optimal solution is determined for every prefix of the input sequence of $n$ days.

3. Achieving $\Theta(kn)$ time requires maintaining a table with $k$ entries whose entry $i$ indicates the subscript of the latest input day such that a permit of type $i$ cannot cover both this day and the day whose subproblem is under consideration.
1. Describe problem input.

The $k$ permit types are in arrays \texttt{permitDays}_1 \ldots \texttt{permitDays}_k$ and \texttt{permitAmount}_1 \ldots \texttt{permitAmount}_k.

The $n$ days to be covered by permits are in the array \texttt{day}_1 \ldots \texttt{day}_n.

2. Determine cost function and base case.

\[ C(i) = \text{Minimum cost for permits to cover the first } i \text{ input days.} \]

\[ C(0) = 0 \]

3. Determine general case.

\[ C(i) = \min_{1 \leq j \leq k} \{ \texttt{permitAmount}_j + C(\max\{\texttt{lag} \mid 0 \leq \texttt{lag} < i \land \texttt{day}_{\texttt{lag}} < \texttt{day}_i - \texttt{permitDays}_j + 1\}) \} \]

Translated: For each day, try every permit. To the cost of a candidate permit, add the cost of previous permits to optimally cover all driving days before the candidate permit.

4. Appropriate ordering of subproblems. Compute \( C(i) \) in ascending \( i \) order.

Using the cost function directly will lead to \( \Theta(kn^2) \) time due to table scanning for \( \texttt{lag} \).

\( \Theta(kn) \) approach - have an array for the \( k \) \( \texttt{lag} \) values and attempt to increment before trying the corresponding permit for \( i \).

5. Backtrace for solution - easily done if the \( j \) giving the minimum is saved for each \( i \). Not difficult to use recomputation (see code).

For much more about the parking permit problem and its role in online computation, see:


http://ieeexplore.ieee.org.esproxy.uta.edu/xpl/articleDetails.jsp?arnumber=1530721