Dynamic Programming

Applies when the following *Principle of Optimality* holds:

> In an optimal sequence of decisions or choices, each subsequence must be optimal.

Translation: There's a recursive solution.

Steps in designing a dynamic programming algorithm:

1. Characterize the solution structure.
2. Recursively define the solution.
3. Solve sub-problems in bottom-up fashion.
4. Construct final solution from sub-problem solutions.

Subproblem results usually are stored in an array.

Examples of Dynamic Programming

- *Fibonacci Numbers*

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = F(1) = 1$$

- *Binomial Coefficients*

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\binom{n}{0} = \binom{n}{n} = 1$$

Dynamic Programming

The *Subset-Sum* problem is the following:

> *Input:* An array $A[1 \ldots n]$ of positive integers, and a target integer $T > 0$.
>
> *Output:* True iff there is a subset of the A values that sums to T. If so, would also like to know what the subset is.

Can solve as follows:

Let $S[i, j]$ be defined as true iff there is a subset of elements $A[1 \ldots i]$ that sums to j. Then $S[n, T]$ is the solution to our problem.

In general:

$$S[i, j] = S[i - 1, j - A[i]] \vee S[i - 1, j]$$

Initial conditions are $S[i, 0] = \text{True}$, and $S[0, j] = \text{False}$, for $j > 0$.

Subset-Sum Problem

Translating to C/C++ code, looks like:

```
for  (i = 0;  i <= n;  i ++)
  S [i] [0] = TRUE;
for  (j = 1;  j <= T;  j ++)
  S [0] [j] = FALSE;

for  (i = 1; i <= n;  i ++)
  for  (j = 1;  j <= T;  j ++)
    S [i] [j] = S [i - 1] [j]
          || (A [i] <= j && S [i - 1] [j - A [i]]);
```

Time complexity is $\Theta(nT)$.

Can find the actual values by "stepping back" through the array as follows:

```
j = T;
for  (i = n;  i >= 1;  i --)
  if  (! S [i - 1] [j])
      {
       printf ("Use item %d = %d\n", i, A [i]);
       j -= A [i];
      }
```

All-Pairs Shortest Paths

Let $D[i, j, k]$ be the shortest distance from vertex i to vertex j *without passing through a vertex numbered higher than* k.

Floyd-Warshall Algorithm:

Set $D[i, j, 0]$ to the adjacency matrix of G.
for $(k = 1$ to $n)$
  for $(i = 1$ to $n)$
    for $(j = 1$ to $n)$

$$D[i, j, k] = \min \left\{ \begin{array}{l} D[i, j, k-1], \\ D[i, k, k-1] \\ \quad + D[k, j, k-1] \end{array} \right\}$$

Time complexity is $\Theta(n^3)$. The third subscript of D can be ignored.

Knapsack Problem

The *Knapsack* problem is the following:

> *Input:* An array $S[1 \ldots n]$ of positive integers, an array $V[1 \ldots n]$ of positive integers, and an integer $K > 0$.
>
> *Output:* The maximum sum of a subset of $V[i]$'s subject to the constraint that the sum of the corresponding $S[i]$'s is not larger than K.
>
> Would also like to know what elements are in the subset.

Can solve in a similar manner to the Subset-Sum problem:

Let $A[i, j]$ be defined as the maximum sum of a subset of elements $V[1 \ldots i]$ whose corresponding $S[i]$'s sum to exactly j. Then $\max\{A[n, j] \mid 1 \leq j \leq K\}$ is the solution to our problem.

In general:

$$A[i, j] =$$

Knapsack Problem

Initial conditions?

Code?

Time complexity?

What happens if we allow each object to be used more than once? *I.e.*, we wish to maximize

$$c_1V[1] + c_2V[2] + \cdots + c_nV[n]$$

where each c_i is a non-negative integer, subject to the constraint that

$$c_1S[1] + c_2S[2] + \cdots + c_nS[n] \leq K$$

Matrix-Chain Multiplication

Input: Dimensions $d_0, \ldots, d_n$ of matrices $M_1, \ldots, M_n$ to be multiplied, where M_i is $d_{i-1} \times d_i$.

Output: Best way to parenthesize the product so as to minimize the cost of the multiplications, where the cost of multiplying a $p \times q$ matrix times a $q \times r$ matrix is pqr.

Matrix-Chain Multiplication

$m[i,j] \stackrel{\text{def}}{=}$ the minimum cost of multiplying matrices $M_i * \cdots * M_j$

$s[i,j] \stackrel{\text{def}}{=}$ the "last" multiplication used to achieve $m[i,j]$,

In other words, $s[i,j] = k$ iff the min-cost parenthesization of $M_i * \cdots * M_j$ has the form $(M_i * \cdots * M_k) * (M_{k+1} * \cdots * M_j)$.

$$m[i,i] = 0$$

$$m[i,j] = \min_{i \le k < j} \left\{ \begin{array}{c} m[i,k] + m[k+1,j] \\ + d_{i-1} d_k d_j \} \end{array} \right\}$$

$s[i,j] = k$ that achieves the above min

Matrix-Chain Multiplication

```
for  (i = 1;  i <= n;  i ++)
  m [i] [i] = 0;

for  (d = 1;  d < n;  d ++)
  for  (i = 1;  i <= n - d;  i ++)
    {
     j = i + d;
     m [i, j] = m [i, i] + m [i + 1] [j]
                    + d [i - 1] * d [i] * d [j];
     s [i, j] = i;
     for  (k = i + 1;  k < j;  k ++)
       {
        Cost = m [i, k] + m [k + 1] [j]
                    + d [i - 1] * d [k] * d [j];
        if  (Cost < m [i] [j])
            {
             m [i] [j] = Cost;
             s [i] [j] = k;
            }
       }
    }

printf ("Min Cost = %d\n", m [1] [n]);
```

Complexity is $\Theta(n^3)$.

Matrix-Chain Multiplication

```
int  Print_Expression  (s, i, j)

//  Print the optimal parenthesization for
//  M [i] ... M [j] based on  the information
//  in  s .

  {
   int  k = s [i] [j];

   if  (i == j)
       {
        printf ("M%d", i);
        return  0;
       }

   if  (k != i)
       printf ("(");
   Print_Expression (s, i, k);
   if  (k != i)
       printf (")");
   printf (" * ");
   if  (k + 1 != j)
       printf ("(");
   Print_Expression (s, k + 1, j);
   if  (k + 1 != j)
       printf (")");

   return  0;
  }
```

Minimum Edit Distance

Input: Two strings: $s[1 \ldots m]$ and $t[1 \ldots n]$.

Output: The minimum number of operations to transform s into t. An operation is: insert a character; delete a character; or substitute a character.

$c[i,j] \stackrel{\text{def}}{=}$ the minimum cost to transform $s[1 \ldots i]$ into $t[1 \ldots j]$.

$$c[0,0] = 0$$

$$c[i,j] = \min \begin{cases} \begin{cases} c[i-1][j-1] & \text{if } s[i] = t[j] \\ 1 + c[i-1, j-1] & \text{otherwise} \end{cases} & \text{(subst)} \\ 1 + c[i-1, j] & \text{(delete)} \\ 1 + c[i, j-1] & \text{(insert)} \end{cases}$$

Minimum Edit Distance

Example:

Here is the c array for strings $s =$ `ababbab` and $t =$ `babbbaaba`

				$t[j]$									
				b	a	b	b	b	a	a	b	a
			0	1	2	3	4	5	6	7	8	9
		0	0	1	2	3	4	5	6	7	8	9
	a	1	1	1	1	2	3	4	5	6	7	8
	b	2	2	1	2	1	2	3	4	5	6	7
	a	3	3	2	1	2	2	3	3	4	5	6
$s[i]$	b	4	4	3	2	1	2	2	3	4	4	5
	b	5	5	4	3	2	1	2	3	4	4	5
	a	6	6	5	4	3	2	2	2	3	4	4
	b	7	7	6	5	4	3	2	3	3	3	4

Minimum Edit Distance

The code looks like this:

```
for  (i = 0;  i <= m;  i ++)
  c [i] [0] = i;
for  (j = 1;  j <= n;  j ++)
  c [0] [j] = j;

for  (i = 1;  i <= m;  i ++)
  for  (j = 1;  j <= n;  j ++)
    {
     Best = c [i - 1] [j - 1];
     if  (s [i] != t [j])
         Best ++;
     if  (1 + c [i - 1] [j] < Best)
         Best = 1 + c [i - 1] [j];
     if  (1 + c [i] [j - 1] < Best)
         Best = 1 + c [i] [j - 1];
     c [i] [j] = Best;
    }
```

Can get the minimum sequence of edit operations by tracing back through the c array.

Travelling Salesperson Problem

Input: A complete directed graph $G = (V, E)$ with non-negative edge weights.

Output: A directed cycle with the smallest possible sum of edge weights that passes through *all* vertices of G.

For $u \in V$, $S \subset V$, $u, v_1 \notin S$, define $D(u, S)$ to be the minimum-weight path starting at vertex u, passing through each vertex in S and ending at vertex v_1.

The solution to the entire problem is then $D(v_1, V \setminus \{v_1\})$.

Let $w_{u,v}$ denote the weight of edge $u \to v$. Then in general we have

$$D(u, S) = \min_{v \in S} \{w_{u,v} + D(v, S \setminus \{u\})\}$$
$$D(u, \emptyset) = w_{u,v_1}$$

Other Dynamic Programming Problems

Longest Common Subsequence

Given two sequences $S = s_1, s_2, \ldots, s_m$ and $T = t_1, t_2, \ldots, t_n$, find the longest sequence that is a subsequence of both.

Longest Ascending Subsequence

Given a sequences $S = s_1, s_2, \ldots, s_n$ of numbers, find the longest subsequence of it in which every number is at least as great as its predecessor.

Bitonic Travelling Salesperson

Given a set of points in the plane, find the minimum-distance cycle that hits every point and has the property that, if we start at the leftmost point, the cycle goes strictly to the right until it hits the rightmost point, and then it goes strictly to the left back to the leftmost point.