The 0/1 Knapsack Problem

In this problem, there are a number of objects. Each object has a weight and a profit associated with it. For example, the objects might be bars of gold, silver, platinum, etc. There is an *unlimited* pile of each object in front of you. You also have a knapsack in which to carry objects. Unfortunately, the knapsack can only hold a given weight before it breaks. The problem is to fill the knapsack with a mixture of objects so that it won't break and so that you carry away the maximum profit.

Now suppose we have the following objects

	Object 1	**Object 2**	**Object 3**	**Object 4**
Profit	10	5	3	1
Weight	9	6	4	2

If, for example, the knapsack capacity was 17, the maximum profit is obtained by taking one of Object 1 plus either one Object 2 and one Object 4 or two of Object 3. In either case, the maximum profit is 16.

For a given weight, this problem can be solved by Dynamic Programming. We begin by creating a recursive solution. The recursive solution is easy to write. The basic pseudocode is:

```
integer Maximum-Profit( Weight )

  if( Weight < 0 ) then
    Let Most-Profit = -∞  --suppress negative weights
  else if ( Weight < 2 ) then
    Let Most-Profit = 0 --weight must be at least 2 for any profit
  else
    let a = 10 + Maximum-Profit( Weight - 9 )
    let b = 5 + Maximum-Profit( Weight - 6 )
    let c = 3 + Maximum-Profit( Weight - 4 )
    let d = 1 + Maximum-Profit( Weight - 2 )
    let Most-Profit = Maximum( a, b, c, d )

  return Most-Profit
```

Dynamic Programming Solution

To convert the recursive solution to a dynamic programming solution, we use the general rule, "Turn rounded parentheses into square brackets." That is, recursive method calls become array lookups. The basic pseudocode for the dynamic programming solution follows.

```
void Maximum-Profit( Weight )
   --Initialize two arrays, Profits[] and Weights[], of size 0 to Weight.
   --Initialize both arrays to zeros.
   --Declare 6 integer variables: a, b, c, d, p, and w

   for currentWeight = 0 to Weight do
      Let a = 0
      if( currentWeight – 9 ≥ 0 ) then
         Let a = 10 + Profits[ currentWeight – 9 ]

      Let b = 0
      if( currentWeight – 6 ≥ 0 ) then
         Let b = 5 + Profits[ currentWeight – 6 ]

      Let c = 0
      if( currentWeight – 4 ≥ 0 ) then
         Let c = 3 + Profits[ currentWeight – 4 ]

      Let d = 0
      if( currentWeight – 2 ≥ 0 ) then
         Let d = 1 + Profits[ currentWeight – 2 ]

      Let p = Maximum( a, b, c, d )

      if( p = a ) then
         Let w = 9
      else if ( p = b ) then
         Let w = 6
      else if( p = c ) then
         Let w = 4
      else
         Let w = 2

      Let Profits[ currentWeight ] = p
      Let Weights[ currentWeight ] = w

   --To print a solution
   Print “Maximum Profit is ” + Profits[ Weight ]

   --To print the objects used to get the maximum profit
   Let i = Weight
   while( i > 0 ) do
      Print “An object of weight ” + Weights[ i ] + “ was used.”
      Let i = i – Weights[ i ]
```

Memoized Solution

For a memoized solution, the two arrays `Profits[]` and `Weights[]` must be declared and initialized in a scope that is larger than the method `Maximum-Profit()`. If you have any experience with Lisp and know what a helper function is, this is a good situation to use a helper function. The array `Weights[]` can be initialized to all zeros, but the array `Profits[]` must be initialized to a unique value that represents “empty”. This unique value must be different than a zero.

```
integer Maximum-Profit( Weight )

   if( Weight < 0 ) then
       Let Most-Profit = -∞  --suppress negative weights
     else if ( Weight < 2 ) then
       Let Most-Profit = 0 --weight must be at least 2 for any profit
     else
      --Declare 5 integer variables: a, b, c, d, and w

      if( Weight – 9 < 0 ) then
         Let a = -∞  --suppress negative weights
         else
            if( Profits[ Weight – 9 ] is empty ) then
               Let Profits[ Weight – 9 ] = Maximum-Profit( Weight – 9 )

            Let a = 10 + Profits[ Weight – 9 ]

      if( Weight – 6 < 0 ) then
         Let b = -∞  --suppress negative weights
         else
            if( Profits[ Weight – 6 ] is empty ) then
               Let Profits[ Weight – 6 ] = Maximum-Profit( Weight – 6 )

            Let b = 5 + Profits[ Weight – 6 ]

      if( Weight – 4 < 0 ) then
         Let c = -∞  --suppress negative weights
         else
            if( Profits[ Weight – 4 ] is empty ) then
               Let Profits[ Weight – 4 ] = Maximum-Profit( Weight – 4 )

            Let c = 10 + Profits[ Weight – 4 ]

      if( Weight – 2 < 0 ) then
         Let d = -∞  --suppress negative weights
         else
            if( Profits[ Weight – 2 ] is empty ) then
               Let Profits[ Weight – 2 ] = Maximum-Profit( Weight – 2 )

            Let d = 1 + Profits[ Weight – 2 ]

      Let Most-Profit = Maximum( a, b, c, d )

      if( Most-Profit = a ) then
         Let w = 9
      else if ( Most-Profit = b ) then
         Let w = 6
      else if( Most-Profit = c ) then
         Let w = 4
      else
         Let w = 2

      Let Weights[ Weight ] = w

   return Most-Profit
```