Dynamic Programming (Intro – still continued)

UCF Programming Team

Fall 2021

Example 4: Subset Sum

Problem: Given a set of numbers S and a target value T, determine whether or not a subset of the values in S adds up exactly to T.

Variations on the Problem:

a) List the set of values that adds up to the target.
b) Allow for multiple copies of each item in S.

Recursive Solution

```
boolean SubsetSum(Set S, int T)
{
   if (T == 0)
      return(true);

   if (S == empty)
      return(false);

   boolean notUseFirstElem = SubsetSum(S – {S[0]}, T);
   boolean useFirstElem    = SubsetSum(S – {S[0]}, T - S[0]);
   return(notUseFirstElem || useFirstElem);
}
```

Now, let's turn this into dynamic programming. First, let's solve it with Iterative DP.

If you take a look at the structure of the recursive calls, the key input parameter is the target value. What we can do is: just store (in a Boolean array) whether or not we've seen a subset that adds to a particular value. Then, we can iterate through each element of the set S and update our Boolean array as necessary. The solution looks like this:

```
boolean[] foundIt = new boolean[T+1]; /* if foundIt[n] is
            true, it means we have a subset that adds up to n */
```

index	0	1	2	. . .	50	. . .
foundIt					true/ false	

```
/* initialize the table */
foundIt[0] = true;
for (int k = 1;  k < T+1;  ++k)
   foundIt[k] = false;

/* update the table */
/* loop thru each element of the set S */
for (int k = 0;  k < S.length;  ++k)
  {
   /* check to see if the element S[k] can be used */
   for (int j = T;  j >= S[k];  --j)
      if (foundIt[j - S[k]])
         /* we have a subset that adds up to "j - S[k]" so we
            can add S[k] to that subset to get a subset
            that adds up to "j" */
         foundIt[j] = true;

  }/* end for (k) */
```

Let's trace the code some:
Let's assume T = 200, i.e., we want to know if there is a subset that adds up to 200.
The outside 'for loop' starts with k = 0.
Let's assume S[0] = 8.
The inside 'for loop' will look like:

```
   for (int j = 200;  j >= 8;  --j)
      if (foundIt[j - 8])
         foundIt[j] = true;
```

Now, let's solve it with Recursive DP (using memoization, i.e., memoize it).

```
int[] memo;

int[] S; /* set of elements (already loaded) */

int solve(int T)
{
   memo = new int[T+1];

   /* initialize memo */
   Arrays.fill(memo, -1);
   memo[0] = 1;

   return(SubsetSum(S,T));
}


int SubsetSum(Set S, int T)
{
   if (T == 0)
      return(1);

   if (S == empty)
      return(0);

   if (memo[T] > -1)
      /* we've already checked/searched for a subset with
         this sum */
      return(memo[T]);

   int notUseFirstElem = SubsetSum(S - {S[0]}, T);
   int useFirstElem    = SubsetSum(S - {S[0]}, T - S[0]);
   int result;
   if ( (notUseFirstElem == 1) || (useFirstElem == 1) )
      result = 1;
   else
      result = 0;
   memo[T] = result;
   return(result);
}
```

Example 5: Game of Gold (from Learn Site: DP-1)

Problem: A set of gold bags, lined up in a row. Two people taking turns picking a bag; one can pick only the bag from either end of the line. Assuming both play optimally, print how many more/less gold the first player will have.

bag	15	28	7	. . .	5	9	12

Java solution on the Google Drive.

/* *****

Coaches' summary ("one-liner") to help when/how to use DP:

If you already know DP, you could just remember it!

***** */