Dynamic Programming (Intro – continued)

UCF Programming Team

Fall 2021

<u>Fibonacci Sequence</u>

```
Fib(n) = Fib(n-1) + Fib(n-2)
Fib(0) = 0  /* Note: some definitions have F(0) = 1 */
Fib(1) = 1
```

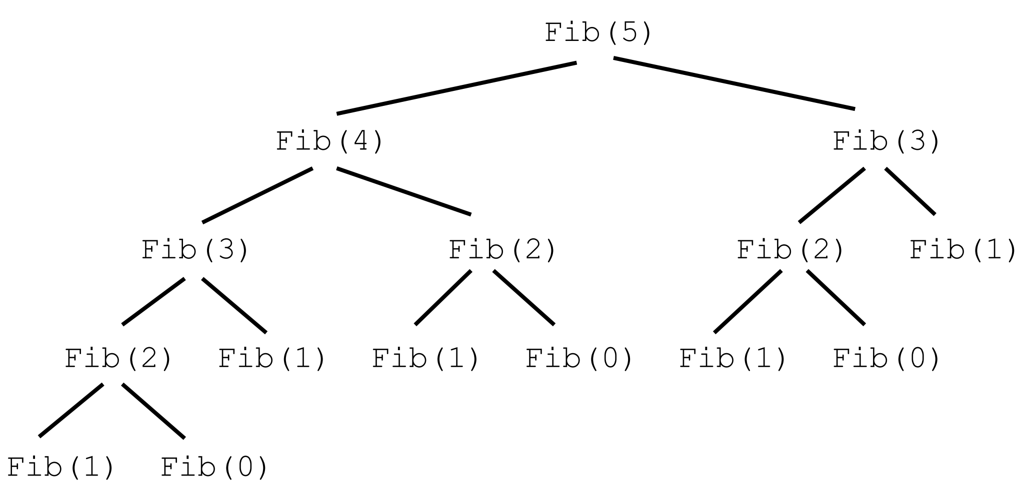
<u>Recursive Solution</u>

```
int Fib(int n)
{
   if ( (n == 0) || (n == 1) )
      Return(n);

   int result = Fib(n-1) + Fib(n-2);
   return(result);
}
```

$O(2^n)$

Fib(50) would result in roughly 10^{15} recursive calls



Iterative DP

```
int Fib(int n)
{
   int[] fib_num = new int[n+1];

   fib_num[0] = 0;
   fib_num[1] = 1;

   for (int k = 2;  k <= n;  ++k)
      fib_num[k] = fib_num[k – 1] + fib_num[k – 2];

   return(fib_num[n]);
}
```

Side Note: for this particular example, we don't really need to remember all the values; we need to remember only the last two values:

```
int Fib(int n)
{
   if ( (n == 0) || (n == 1) )
      Return(n);

   int last, curr, next;

   last = 0;
   curr = 1;

   for (int k = 2;  k <= n;  ++k)
   {
      next = last + curr;
      last = curr;
      curr = next;
   }

   return(curr);
}
```

Recursive DP - Memoization (Memo“ize”)

```
int[] memo;
int solve(int n)
{
   memo = new int[n+1];
   Arrays.fill(memo, -1);  /* initialize memo */
   return(Fib(n));
}

int Fib(int n)
{
   if ( (n == 0) || (n == 1) )
      return(n);

   if (memo[n] > -1)
      /* we’ve already seen/solved this problem */
      return(memo[n]);

   int result = Fib(n-1) + Fib(n-2);
   memo[n] = result;
   return(result);
}
```

Example 2: Coin Change

Given a set S of coin denominations (with an infinite quantity of each denominations) and a target sum *n*, determine the minimum number of coins you need to supply the change.

S_0, S_1, S_2, …

For USA Currency (1, 5, 10, 25), greedy works.

But, for this problem, greedy does not work in general:

1, 5, 6

Change for 10

greedy

$6 + 1 + 1 + 1 + 1 \longrightarrow 5$ coins

optimal

$5 + 5 \longrightarrow 2$ coins

Cashier(n,c) = the minimum number of coins needed to give n cents of change using only coins of denominations S_c or larger

S_0, S_1, S_2, …

e.g., let's say Cashier(65,4) = 12; this means 12 is the minimum number of coins needed to give 65 cents of change using only coins of denominations S_4, S_5, S_6, …

Solution: Cashier(n,0)

Recurrence Relation:

$$Cashier(n,c) = \min\begin{cases} Cashier(n - S_c, c) + 1 \\ Cashier(n, c+1) \end{cases}$$

Cashier(0,c) = 0

Cashier(n<0, c) = infinity

Cashier(n>0, |S|) = infinity /* |S| is the cardinality of S */

```
int[][] memo; /* 2D; solving Cashier(n,c) */

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

int solve(int n)
{
   memo = new int[n+1][S.length];

   /* initialize memo */
   for (int[] m : memo)
      Arrays.fill(m, -1);

   return(Cashier(n,0));
}


int Cashier(int n, int c)
{
   if (n == 0)
      return(0);

   if (n < 0)
      return(infinity);

   if (c == S.length)
      return(infinity);

   if (memo[n][c] > -1)
      /* we've already seen/solved this problem */
      return(memo[n][c]);

   int result = Math.min( 1 + Cashier(n-S[c], c) ,
                          Cashier(n, c+1) );
   memo[n][c] = result;
   return(result);
}
```

Example 3: 0-1 Knapsack

Given a capacity C and a set of n items (each item having a weight and a value):

$S = \{(W_1, V_1),\ (W_2, V_2),\ \ldots\ (W_n,\ V_n)\}$

Find some subset T of S such that:

$$\sum_{i \in T} W_i \le C \quad and \quad \sum_{i \in T} V_i \text{ is maximal}$$

Bag(w,k) = the maximum total value of the subset that uses items S_k, S_{k+1}, … where the total weight for this subset ≤ w.

S_0, S_1, S_2, …

e.g., let's say Bag(200,5) = 90; this means 90 is the maximum total value of the subset that uses items S_5, S_6, S_7, … where the total weight for this subset ≤ 200.

Solution: Bag(C,0)

Recurrence Relation:

$$Bag(w,k) = \max\begin{cases} Bag(w - W_k, k+1) + V_k & if\ w \ge W_k \\ Bag(w, k+1) & \end{cases}.$$

Bag(0,k) = 0

Bag(w<0, k) = -infinity

Bag(w>0, |S|) = 0 /* |S| is the cardinality of S */

```
int[][] memo; /* 2D; solving Bag(w,k) */

int[] SW, SV; /* set of items (weight, value); already loaded */

int solve(int w)
{
   memo = new int[w+1][SW.length];

   /* initialize memo */
   for ( int[] m : memo )
      Arrays.fill(m, -1);

   return(Bag(w,0));
}


int Bag(int w, int k)
{
   if ( w == 0 )
      return(0);

   if ( w < 0 )
      return(-infinity);

   if ( k == SW.length )
      return(0);

   if ( memo[w][k] > -1 )
      return(memo[w][k]);

   int result = Bag(w, k+1);
   if ( w >= SW[k] )
      result = Math.max( result,
                         SV[k] + Bag(w-SW[k], k+1) );
   memo[w][k] = result;
   return(result);
}
```