Memoization and Dynamic Programming: Lecture #1

Motivation

We have looked at several algorithms that involve recursion. In some situations, these algorithms solve fairly difficult problems efficiently, but in other cases they are inefficient because they recalculate certain function values many times. The example given in the text is the fibonacci example. Recursively we have:

```
public static int fibrec(int n) {
     if (n < 2)
          return n;
     else
          return fibrec(n-1)+fibrec(n-2);
}
```

The problem here is that lots and lots of calls to Fib(1) and Fib(0) are made. It would be nice if we only made those method calls once, then simply used those values as necessary.

In fact, if I asked you to compute the 10th Fibonacci number, you would never do it using the recursive steps above. Instead, you'd start making a chart:

$F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, $F_9 = 34$, $F_{10} = 55$.

First you calculate F_3 by adding F_1 and F_2, then F_4, by adding F_3 and F_4, etc.

The idea of memoization and dynamic programming is to avoid making redundant method calls. Instead, one should store the answers to all necessary method calls in memory and simply look these up as necessary.

For memoization, we simply make tiny changes to the recursive code (making our own lookup table and manually avoiding recursion in the cases we've previously carried out a particular computation).

For dynamic programming, we remove all recursion from our solution and instead, carefully order our calculations of solutions to subproblems so that any time we would like to look up an answer to a "recursive call", we can look it up because we've already calculated it.

First, let's look at a memoized version of Fibonacci. Before this code gets executed, we need to set up the following as a static variable in the class. (Alternatively, we can pass the memo table into the method as a second parameter.)

```
public static int[] memo;
```

In main, once we know the input size of the query, we would do:

```
memo = new int[n+1];
Arrays.fill(memo, -1);
```

Now, here is the method:

```
public static int fibrec(int n) {
    if (n < 2) return n;
    if (memo[n] != -1) return memo[n];
    memo[n] = fibrec(n-1)+fibrec(n-2);
    return memo[n];
}
```

In a nutshell, memo is our memory. First we set it to all -1s. This means we don't know anything. In general, we'll set memo[n] to the n^{th} Fibonacci number. So, all the changes we make to the code are as follows:

1) Create a memo table that has appropriate storage for all possible recursive calls outside of the recursive method. Set this table to a sentinel value that can not be the answer to a real recursive call. Typically -1 works well.

2) In our base case, immediately check to see if we've solved this before. If so, just return the answer stored in the memo table.

3) Before we return the answer to a query, store it in the memo table.

Basically, any time in our computation tree we hit a call to fib(k), where we've previously calculated it, we avoid that whole computation tree!

Now, for dynamic programming, we realized that we want to calculate all Fibonacci numbers from the smallest to the largest. We keep the same storage as our memo table (but I'll rename it in this example) but fill it in incrementally and instead of making a recursive call ever, we'll just directly access the table any time we want to know some Fibonacci number. Here is the code:

```
public static int fib(int n) {

     int[] fibnumbers = new int[n+1];
     fibnumbers[0] = 0;
     fibnumbers[1] = 1;

     for (int i=2; i<n+1;i++)
          fibnumbers[i] = fibnumbers[i-1]+fibnumbers[i-2];
     return fibnumbers[n];
}
```

The only requirement this program has that the recursive one doesn't is the space requirement of an entire array of values. (But, if you think about it carefully, at a particular moment in time while the recursive program is running, it has at least n recursive calls in the middle of execution all at once. The amount of memory necessary to simultaneously keep track of each of these is in fact at least as much as the memory the array we are using above needs.)

Usually however, a dynamic programming algorithm presents a time-space trade off. More space is used to store values, but less time is spent because these values can be looked up.

Can we do even better (with respect to memory) with our Fibonacci method above? What numbers do we really have to keep track of all the time?

```
public static int fib(int n) {

     int fibfirst = 0;
     int fibsecond = 1;

     for (int i=2; i<n+1;i++) {
          fibsecond = fibfirst+fibsecond;
          fibfirst = fibsecond - fibfirst;
     }
     return fibsecond;
}
```

So here, we calculate the nth Fibonacci number in linear time (assuming that the additions are constant time, which is actually not a great assumption) and use very little extra storage.

Problem #1: Binomial Coefficients

There is a direct formula for calculating binomial coefficients, it's $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

However, it's instructive to calculate binomial coefficients using dynamic programming since the technique can be used to calculate answers to counting questions that don't have a simple closed-form formula.

The recursive formula for binomial coefficients is $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, with $\binom{n}{0} = \binom{n}{n} = 1$.

In code, this would look roughly like this:

```
int combo(int n, int k) {
  if (n == 0 || n == k)
    return 1;
  else
    return combo(n-1,k-1)+combo(n-1,k);
}
```

However, this ends up repeating many instances of recursive calls, and ends up being very slow. We can implement a dynamic programming solution by creating a two dimensional array which stores all the values in Pascal's triangle. When we need to make a recursive call, instead, we can simply look up the answer in the array. To turn a recursive solution into a DP one, here's what has to be done:

a) Characterize all possible input values to the function and create an array to store the answer to each possible problem instance that is necessary to solve the problem at hand.

b) Seed the array with the initial values based on the base cases in the recursive solution.

c) Fill in the array (in an order so that you are always looking up array slots that are already filled) using the recursive formula, but instead of making a recursive call, look up that value in the array where it should be stored.

The code is on the next page:

```
public static int combo(int n, int k) {
    int[][] tri = new int[n+1][n+1];
    for (int i=0; i<n+1; i++) {
        tri[i][0] = 1;
        tri[i][i] = 1;
    }

    for (int i=2; i<n+1; i++)
        for (int j=1; j<i; j++)
            tri[i][j] = tri[i-1][j-1] + tri[i-1][j];

    return pascaltri[n][k];
}
```

The key idea here is that pascaltri[i][j] always stores $\binom{i}{j}$. Since we fill in the array in increasing order, by the time we look up values in the array, they are already there. Basically, what we are doing, is building up the answers to subproblems from small to large and then using the smaller answers as needed.

Side Problem: Computing a "large" row of Pascal's Triangle mod a prime

Recall that $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. If we wanted to calculate $\binom{n}{k} \bmod p$ for some prime p, we can store a table of all the values of n! mod p up to our maximum value of n. The issue becomes division. We can't directly divide under mod, since mod requires integers and regular division is likely to give us non-integers. Instead, when dealing with mod, anything that looks like a division is replaced with multiplication by a modular inverse. We define *a'* to be the modular inverse of *a mod p*, iff $aa' \equiv 1 \bmod p$. Java has a method in its BigInteger class called modInverse() that will return the modular inverse of a mod p. If a and p were BigInteger objects, we could solve for the modular inverse as follows:

```
BigInteger aInv = a.modInverse(p);
```

If we had longs and wanted to utilize this method, we could write our own wrapper method:

```
public static long modInv(long a, long p) {
    BigInteger bigA = new BigInteger(""+a);
    BigInteger bigP = new BigInteger(""+p);
    return bigA.modInverse(bigP).longValue();
}
```

Returning to our problem of calculating $\binom{n}{k} \bmod p$ quickly, what we could do is simply multiply *n! mod p* by the inverse of *k! mod p* times the inverse of *(n-k)! mod p*. All of these values would be stored in two look-up tables one for the regular values mod p and the other for the inverse values mod *p*.

Problem #2: Longest Common Subsequence

The problem is to find the longest common subsequence in two given strings. A subsequence of a string is simply some subset of the letters in the whole string in the order they appear in the string. In order to denote a subsequence, you could simply denote each array index of the string you wanted to include in the subsequence. For example, given the string "GOODMORNING", the subsequence that corresponds to array indexes 1, 3, 5, and 6 is "ODOR."

Here is the basic idea behind solving the problem:

If the last characters of both strings s1 and s2 match, then the LCS will be one plus the LCS of both of the strings with their last characters removed.

If the initial characters of both strings do NOT match, then the LCS will be one of two options:

1) The LCS of x and y without its last character.
2) The LCS of y and x without its last character.

Thus, in this case we will simply take the maximum of these two values. Also, we could just as easily have compared the *first* two characters of x and y and used a similar technique.

Let's examine the code for both the recursive solution to LCS and the dynamic programming solution:

```
// Arup Guha
// 3/2/05

// The method below solves the longest common subsequence
// problem recursively.
import java.io.*;

public class LCS {

  // Precondition: Both x and y are non-empty strings.
  //               0 < len1 <= x.length() , 0 < len2 <= y.length
  public static int lcsrec(String x, String y) {

    // If one of the strings has one character, search for that
    // character in the other string and return the appropriate
    // answer.
    if (x.length() == 1)
        return find(x.charAt(0), y);
    if (y.length() == 1)
        return find(y.charAt(0), x);

    // Solve the problem recursively.

    // Corresponding last characters match.
    if (x.charAt(len1-1) == y.charAt(len2-1))
        return 1+lcsrec(x.substring(0, x.length()-1),
                                  y.substring(0,y.length()-1));

    // Corresponding characters do not match.
    else
        return max(lcsrec(x, y.substring(0, y.length()-1)),
                             lcsrec(x.substring(0,x.length()-1), y));

  }
```

Now, our goal will be to take this recursive solution and build a dynamic programming solution. The key here is to notice that the heart of each recursive call is the pair of indexes, telling us which prefix string we are considering. In some sense, we can build the answer to "longer" LCS questions based on the answers to smaller LCS questions. This can be seen trace through the recursion at the very last few steps.

If we make the recursive call on the strings RACECAR and CREAM, once we have the answers to the recursive calls for inputs RACECAR and CREA and the inputs RACECA and CREAM, we can use those two answers and immediately take the maximum of the two to solve our problem!

Thus, think of *storing* the answers to these recursive calls in a table, such as this:

	R	A	C	E	C	A	R
C							
R							
E							
A			XXX				
M							

In this chart for example, the slot with the XXX will store an integer that represents the longest common subsequence of CREA and RAC. (In this case 2.)

Now, let's think about building this table. First we will initialize the first row and column:

	R	A	C	E	C	A	R
C	0	0	1	1	1	1	1
R	1						
E	1						
A	1						
M	1						

Basically, we search for the first letter in the other string, when we get there, we put a 1, and all other values subsequent to that on the row or column are also one. This corresponds to the base case in the recursive code.

Now, we simply fill out the chart according to the recursive rule:

1) Check to see if the "last" characters match. If so, delete this and take the LCS of what's left and add 1 to it.

2) If not, then we try to possibilities, and take the maximum of those two possibilities. (These possibilities are simply taking the LCS of the whole first word and the second work minus the last letter, and vice versa.)

Here is the chart:

	R	A	C	E	C	A	R
C	0	0	1	1	1	1	1
R	1	1	1	1	1	1	2
E	1	1	1	2	2	2	2
A	1	2	2	2	2	3	3
M	1	2	2	2	2	3	3

Now, let's use this to develop the dynamic programming code.

```
public static int lcsdyn(String x, String y) {

    int i,j;
    int lenx = x.length();
    int leny = y.length();
    int[][] table = new int[lenx+1][leny+1];

    // Fill in each LCS value in order from top row to bottom row,
    // moving left to right.
    for (i = 1; i<=lenx; i++) {

        for (j = 1; j<=leny; j++) {

            // Last chars match
            if (x.charAt(i-1) == y.charAt(j-1))
                table[i][j] = 1+table[i-1][j-1];

            // Take best of two possible smaller cases.
            else
                table[i][j] = Math.max(table[i][j-1], table[i-1][j]);

            System.out.print(table[i][j]+" ");
        }
        System.out.println();
    }

    // This is our answer.
    return table[lenx][leny];
}
```

Problem #3: Longest Path in a DAG

Since a DAG doesn't have cycles, there are well-defined longest paths in DAGs. One way to solve this problem would be to use memoization. Recursively, a solution looks like this, assuming that a path always existed:

```
int longpath(ArrayList[] graph, int source, int end) {

    if (memo[source] != -1) return memo[source];
    if (source == end) return 0;

    int best = Integer.MIN_VALUE;

    for (int i=0; i<graph[source].size(); i++) {

        int cur = graph[source].get(i).weight +
         longpath(graph, graph[source].get(i).vertex, end);

       best = Math.max(best, cur);
    }

    memo[source] = best;
    return best;
}
```

Each array list must be an array list of objects that have the instance variables weight and vertex that store those corresponding items for that connection from the source variable and the memo array has to be declared as a static class variable.

In essence, the code just says, "Try all paths that lead from source. The longest these paths could be is the sum of the edge from source to the next vertex, plus the longest path from that next vertex to our end vertex. Of all of these, take the longest!"

Problem #4: Subset Sum

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.

The recursive solution looks something like this:

```
// Returns true iff a subset of values from s[k..len-1]
// sums to target.
public static boolean SS(int[] s, int k, int target) {

    if (target == 0) return true;
    if (k == s.length) return false;

    return SS(s, k+1, target) || SS(s, target-s[k]);
}
```

The basic idea is as follows: All subsets of S either contain S[k] or don't contain that idea. If a subset exists that adds up to the target, then we have two choices:

a) Don't take the value
b) Take the value

If we don't take the value, then if we want an overall subset that adds up to T, we must find a subset of the rest of the elements that adds up to target.

If we do take the value, then we want to add that element, to a subset of the rest of the set that adds up to target minus the value taken.

These cases, a and b, correspond to the two recursive calls. Memoized, we could do:

```
Boolean[] memo = new memo[target+1];
Arrays.fill(memo, null);

public static boolean SS(int[] s, int k, int target) {
    if (target == 0) return true;
    if (k == s.length) return false;
    if (memo[target] != null) return memo[target];
    memo[target] =SS(s, k+1, target) || SS(s, target-s[k]);
    return memo[target];
}
```

Here, we use the Wrapper class Boolean and use null as our sentinel value, since neither true nor false can be our sentinel value.

Now, when we convert this to dynamic programming, it'll look quite a bit different than our memoized version. We simulate the role of k with a for look through the current item we're considering and we simulate building off of old targets by looping through every possible previous target.

```
boolean[] foundIt = new boolean[target+1];
foundIt[0] = true;
for (int i=1; i<target+1; i++) foundIt[i] = false;

for (int i=0; i<s.length; i++)
   for (int j=target; j>=s[i]; j--)
      if (foundIt[j – s[i]])
          foundIt[j] = true;
```

Basically, in the outer loop we go through each element. Then, we look at all of the previous targets and see if we can build off of them. If we have a subset without item i that adds to j - s[i], then when we add item i to that subset, we'll get a new subset that adds to j, so we mark it as true. Note that it's important that we do the inner loop backwards, so that we don't change a value in the foundIt array based on element i, and then change a second element in the foundIt array based on the previously changed element. For example, if s[i] = 10, then if the loop worked forward, we'd mark foundIt[10] true, but later as the loop marched up, we'd mark foundIt[20] true because we have a subset that adds to 10 (just 10 itself) and then we're adding 10 to it again. But, we're not allowed to use the same item more than once! This is why the loop has to go backwards.

If we want to allow multiple copies of each element, run the inner for loop forwards. Can you see why that works?

```
for (int i=0; i<s.length; i++)
   for (int j=s[i]; j<=target; j--)
      if (foundIt[j – s[i]])
          foundIt[j] = true;
```

Problem #5: Knapsack - Subset Sum Generalized

Imagine in the subset sum problem that each item also had an associated value. So, our items are pairs - a value and a weight and our goal was to find some subset of items whose sum of weights equals some number while we try to maximize the total value of that subset. Thus, instead of just assigning a true or false value to each sub-case, we now want to assign a maximal integer value, for the best valued subset of each particular weight. Recursively, very little changes, except we try both branches and return the larger of the two values instead of their or.

Iteratively, our code looks like this:

```
public static int knapsack(items[] list, int target) {

  int[] best = new int[target+1];
  for (int i=0; i<list.length; i++)
    for (int j=target; j>=list[i].value; j--)
      best[j] = Math.max(best[j], best[j-list[i].value]+
                                  list[i].weight);

  return best[target];
}
```

Note: if we wanted the best value of any knapsack with weight target or less, we could just sweep through the best array at the end and return the largest value in it.

Once again, if we want to allow repeat values, we just run the inner loop forward so we can build off the same item.