COP 4516 Dynamic Programming: Lecture #1

Standard Algorithms to Know

Computing Binomial Coefficients (Brassard 8.1)
World Series Problem (Brassard 8.1)
Making Change (Brassard 8.2)
Knapsack (Brassard 8.4 Goodrich 5.3)
Subset Sum (special instance of knapsack where weights=values)
Floyd-Warshall's (Brassard 8.5 Cormen 26.2)
Chained Matrix Multiplication (Brassard 8.6, Cormen 16.1 Goodrich 5.3)
Longest Common Subsequence (Cormen 16.3)
Edit Distance (Skiena 11.2)
Polygon Triangulation (Cormen 16.4)

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 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.

Using this idea, we can code up a dynamic programming solution to the Fibonacci number question that is far more efficient than the recursive version:

```
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.

To see an illustration of the difference in speed, I wrote a short main to test this:

```
public static void main(String[] args) {

    long start = System.currentTimeMillis();
    System.out.println("Fib 30 = "+fib(30));
    long mid = System.currentTimeMillis();
    System.out.println("Fib 30 = "+fibrec(30));
    long end = System.currentTimeMillis();

    System.out.println("Fib Iter Time = "+(mid-start));
    System.out.println("Fib Rec Time = "+(end-mid));
  }
// Output:
// Fib Iter Time = 4
// Fib Rec Time = 258
```

Example #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.

In pseudocode, here's how binomial combinations can be computed using dynamic programming:

```
int combo(int n, int k) {

  int pascaltri[][] = new int[n+1][n+1];
  for (int i=0; i<n+1; i++) {
    pascaltri[i][0] = 1;
    pascaltri[i][i] = 1;
  }

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

Example #2: 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 add up to the target.
b) Allow for multiple copies of each item in S.

The recursive solution looks something like this:

```
SubsetSum(Set S, int Target) {
  if (Target == 0) return true;
  else if (S == empty) return false;
  else
    return SubsetSum(S – {S[length-1]}, Target) ||
          SubsetSum(S – {S[length-1]}, Target-S[length-1]);
}
```

The basic idea is as follows: All subsets of S either contain S[length-1] or don't contain that idea. If a subset exists that adds up to T, 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 T.

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 T minus the value taken.

These cases, a and b, correspond to the two recursive calls.
Now, to turn this into dynamic programming.

If you take a look at the structure of the recursive calls, the key input parameter is the target value. What we could do, is just store whether or not we've seen a subset that adds to a particular value in a boolean array. Then, we can iterate through each element and update our array as necessary.

The solution roughly looks like this:

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

for (int i=0; i<S.length; i++) {
   for (int j=T; j>=S[i]; j--)
      if (foundIt[j - S[i]])
          foundIt[j];
}
```

If we want to remember which values make up the subset, then instead of a boolean array, store an integer array, and in each index, just store the last value that you added to the set to get you there.

```
int[] Subset = new int[T+1];
for (int i=0; i<T+1; i++) Subset[i] = 0;

for (int i=0; i<S.length; i++) {
   for (int j=T; j>=S[i]; j--)
      if (Subset[j - S[i]] != 0 || j == S[i])
          Subset[j] = S[i];
}
```

From here, to recreate the set, you can just "jump" backwards through the integer array. If Subset[15] = 3, then 3 is an element that adds up to 15. Then you take 15 – 3 to get 12 and look at Subset[12]. If this is 7, for example, then 7 is in the set too, and then you can look at Subset[5]. If this is 5, then that means that your set that added up to 15 was 3, 7 and 5.

Finally, if we want to allow mutiple 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<=T; j++)
      if (Subset[j - S[i]] != 0 || j == S[i])
          Subset[j] = S[i];
}
```

Examle #3: Knapsack Problem (Subset Sum Generalized)

Your goal is to maximize the value of a knapsack that can hold at most W units worth of goods from a list of items I_0, I_1, ... I_{n-1}. Each item has two attributes:

1) Value - let this be v_i for item I_i.
2) Weight - let this be w_i for item I_i.

Now, instead of being able to take a certain weight of an item, you can only either take the item or not take the item.

The naive way to solve this problem is to cycle through all 2^n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack. But, we can find a dynamic programming algorithm that will USUALLY do better than this brute force technique.

Our first attempt might be to characterize a sub-problem as follows:

Let S_k be the optimal subset of elements from $\{I_0, I_1, ... I_k\}$. But what we find is that the optimal subset from the elements $\{I_0, I_1, ... I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1, ... I_k\}$ in any regular pattern. Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

To illustrate this, consider the following example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

The maximum weight the knapsack can hold is 20.

The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$ but the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$. In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$. (Instead it build's upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

So, now, we must rework our example. In particular, after trial and error we may come up with the following idea:

Let B[k, w] represent the maximum total value of a subset S_k with weight w. Our goal is to find B[n, W], where n is the total number of items and W is the maximal weight the knapsack can carry.

Using this definition, we have B[0, w] = v_0, if w >= w_0.

= 0, otherwise

Now, we can derive the following relationship that B[k, w] obeys:

$B[k, w] = B[k - 1,w]$, if $w_k > w$
$= \max \{ B[k - 1,w], B[k - 1,w - w_k] + v_k\}$

In English, here is what this is saying:

1) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w is the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_{k-1}\}$ with weight w, if item k weighs greater than w.

Basically, you can NOT increase the value of your knapsack with weight w if the new item you are considering weighs more than w – because it WON'T fit!!!

2) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_1, I_2, ... I_{k-1}\}$ with weight w, if item k should not be added into the knapsack.

OR

3) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_{k-1}\}$ with weight $w-w_k$, plus item k.

You need to compare the values of knapsacks in both case 2 and 3 and take the maximal one.

Recursively, we will STILL have an $O(2^n)$ algorithm. But, using dynamic programming, we simply have to do a double loop - one loop running n times and the other loop running W times.

Question: In which cases would a running time of O(nW) be worse than a running time of $O(2^n)$?
Here is a dynamic programming algorithm to solve the 0-1 Knapsack problem:

Input: S, a set of n items as described earlier, W the total weight of the knapsack. (Assume that the weights and values are stored in separate arrays named w and v, respectively.)

Output: The maximal value of items in a valid knapsack.

```
int w, k;
for (w=0; w <= W; w++)
    B[w] = 0

for (k=0; k<n; k++) {

    for (w = W; w>= w[k]; w--)  {

        if (B[w – w[k]] + v[k] > B[w])
            B[w] = B[w – w[k]] + v[k]
    }
}
```

Note on run time: Clearly the run time of this algorithm is O(nW), based on the nested loop structure and the simple operation inside of both loops. When comparing this with the previous $O(2^n)$, we find that depending on W, either the dynamic programming algorithm is more efficient or the brute force algorithm could be more efficient. (For example, for n=5, W=100000, brute force is preferable, but for n=30 and W=1000, the dynamic programming solution is preferable.)

Let's run through an example:

i	Item	w_i	v_i
0	I_0	4	6
1	I_1	2	4
2	I_2	3	5
3	I_3	1	3
4	I_4	6	9
5	I_5	4	7

W = 10

Item	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	6	6	6	6	6	6	6
1	0	0	4	4	6	6	10	10	10	10	10
2	0	0	4	5	6	9	10	11	11	15	15
3	0	3	4	7	8	9	12	13	14	15	18
4	0	3	4	7	8	9	12	13	14	16	18
5	0	3	4	7	8	10	12	14	15	16	19

Example #4: 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];

    // Initialize table that will store LCS's of all prefix strings.
    // This initialization is for all empty string cases.
    for (i=0; i<=lenx; i++)
                table[i][0] = 0;
    for (i=0; i<=leny; i++)
                table[0][i] = 0;

    // 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];
}
```

Example #5: Change Problem (Number of Ways)

"The Change Store" was an old SNL skit (a pretty dumb one...) where they would say things like, "You need change for a 20? We'll give you two tens, or a ten and two fives, or four fives, etc."

If you are a dorky minded CS 2 student, you might ask yourself (after you ask yourself why those writers get paid so much for writing the crap that they do), "Given a certain amount of money, how many different ways are there to make change for that amount of money?"

Let us simplify the problem as follows:

Given a positive integer n, how many ways can we make change for n cents using pennies, nickels, dimes and quarters?

Recursively, we could break down the problem as follows:

To make change for n cents we could:
1) Give the customer a quarter. Then we have to make change for n-25 cents
2) Give the customer a dime. Then we have to make change for n-10 cents
3) Give the customer a nickel. Then we have to make change for n-5 cents
4) Give the customer a penny. Then we have to make change for n-1 cents.

If we let T(n) = number of ways to make change for n cents, we get the formula

$T(n) = T(n-25)+T(n-10)+T(n-5)+T(n-1)$
Is there anything wrong with this?

If you plug in the initial condition $T(1) = 1$, $T(0)=1$, $T(n)=0$ if $n<0$, you'll find that the values this formula produces are incorrect. (In particular, for this recurrence relation $T(6)=3$, but in actuality, we want $T(6)=2$.)

So this can not be right. What is wrong with our logic? In particular, it can been seen that this formula is an OVERESTIMATE of the actual value. Specifically, this counts certain combinations multiple times. In the above example, the one penny, one nickel combination is counted twice. Why is this the case?
The problem is that we are counting all combinations of coins that can be given out where ORDER matters. (We are counting giving a penny then a nickel separately from giving a nickel and then a penny.)

We have to find a way to NOT do this. One way to do this is IMPOSE an order on the way the coins are given. We could do this by saying that coins must be given from most value to least value. Thus, if you "gave" a nickel, afterwards, you would only be allowed to give nickels and pennies.

Using this idea, we need to adjust the format of our recursive computation:

To make change for n cents using the largest coin d, we could

1)If d is 25, give out a quarter and make change for n-25 cents using the largest coin as a quarter.
2)If d is 10, give out a dime and make change for n-10 cents using the largest coin as a dime.
3)If d is 5, give out a nickel and make change for n-5 cents using the largest coin as a nickel.
4)If d is 1, we can simply return 1 since if you are only allowed to give pennies, you can only make change in one way.

Although this seems quite a bit more complex than before, the code itself isn't so long. Let's take a look at it:

```
public static int makeChange(int n, int d) {

    if (n < 0)
        return 0;
    else if (n==0)
        return 1;
    else  {
        int sum = 0;
        switch (d)  {
            case 25: sum+=makeChange(n-25,25);
            case 10: sum+=makeChange(n-10,10);
            case 5: sum += makeChange(n-5,5);
            case 1: sum++;
         }
         return sum;
     }
}
```

There's a whole bunch of stuff going on here, but one of the things you'll notice is that the larger n gets, the slower and slower this will run, or maybe your computer will run out of stack space. Further analysis will show that many, many method calls get repeated in the course of a single initial method call.

In dynamic programming, we want to AVOID these reoccuring calls. To do this, rather than making those three recursive calls above, we could store the values of each of those in a two dimensional array.

Our array could look like this

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4
10	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6
25	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6

Essentially, each row label stands for the number of cents we are making change for and each column label stands for the largest coin value allowed to make change.

(Note: The lightly colored squares with 1, 2 and 1 are added to calculate the lightly colored square with 4, based on the recursive algorithm.)

Now, let us try to write some code that would emulate building this table by hand, from left to right.

```
public static int makeChangedyn(int n, int d) {

    // Take care of simple cases.
    if (n < 0)
        return 0;
    else if ((n>=0) && (n < 5))
        return 1;

     // Build table here.
     else {

          int[] denominations = {1, 5, 10, 25};
          int[][] table = new int[4][n+1];

          // Initialize table
          for (int i=0; i<n+1;i++)
              table[0][i] = 1;
          for (int i=0; i<5; i++) {
              table[1][i] = 1;
              table[2][i] = 1;
              table[3][i] = 1;
          }
          for (int i=5;i<n+1;i++) {
              table[1][i] = 0;
              table[2][i] = 0;
              table[3][i] = 0;
          }

         // Fill in table, row by row.
         for (int i=1; i<4; i++) {
             for (int j=5; j<n+1; j++) {
                 for (int k=0; k<=i; k++) {
                     if ( j >= denominations[k])
                         table[i][j]+= table[k][j-denominations[k]];
                  }
              }
          }
          return table[lookup(d)][n];
     }
}
```

An alternate way to code this up is to realize that we DON'T need to add many different cases up together. Instead, we note that the number of ways to make change for n cents using denomination d can be split up into counting two groups:

1) The number of ways to make change for n cents using denominations LESS than d

2) The number of ways to make change for n cents using at least ONE coin of denomination d.

The former is simply the value in the table that is directly above the one we are trying to fill.

The latter is the value on the table that is on the same row, by d spots to the left.

Visually, consider just adding two values from our previous example:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4
10	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6
25	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6

(Also note that the lightly colored three was computed by adding the 1 and 2 that were lightly colored in the previous example.)

Here is the code to implement this slight change, just substitute this line for the for loop with k in the previous code:

```
if ( j >= denominations[i])
       table[i][j] = table[i-1][j] + table[i][j - denominations[k]];
else
       table[i][j] = table[i-1][j]
```

Example #6: Testing the Catcher

If you read this carefully, this problem is really a longest non-increasing sequence problem.

The recursive solution is as follows:

For the first value in the list, we have two options: (a) take it, (b) don't take it

Work out which of these two options is better and return the maximum of the two strategies.

Our recursive function takes in four parameters:

1) The whole list
2) The size of the list
3) The index of the current value we are considering
4) The height of the last missile taken previous to the current.

One thing to note is that sometimes, the current missile can not be intercepted. This is precisely when its height is greater than the height of the previous missle intercepted.

Initially, the last parameter is set to a value greater than the height of any missile.

In the code, if it's possible to intercept the current missile, then take the maximum of:

1) 1 + maximum number of missiles that can be intercepted from the rest of the list such that the maximum height is that of the current missile.

2) the maximum number of missiles that can be intercepted from the rest of the list such that the maximum height is that of the previous missile taken.

Now, the question is, how can we turn this into DP?

For each sublist starting from the beginning, perhaps we could store the maximum number of missiles that can be intercepted from that sublist. Thus, for the list:

0	1	2	3	4	5	6	7
80	70	60	50	65	45	60	61

We could store (in an auxiliary array) the values:

0	1	2	3	4	5	6	7
1	2	3	4	4	5	5	5

The key to DP is being able to construct this table just using previously filled in values to the table. The problem here is just because we know that the longest list using the first 7 elements (in indexes 0 through 5) is 5, doesn't mean we can decide whether or not we can do better using the last element, 35. In fact, the information we need is what was the height of the last missile in the list of four intercepted missiles. If this is greater than or equal to 61, then we can add 61 to the list, otherwise we can't.

So, we have two options:

1) Store the last missile height the corresponds to each of the longest sequences
2) Stipulate that the entry in the array corresponds to the longest sequence of missiles that can be intercepted with the LAST missile being intercepted.

It turns out that the characterization for choice 2 works quite well. Here is the adjusted auxiliary array if we choose to store this information:

0	1	2	3	4	5	6	7
1	2	3	4	3	5	4	3

To finish up the problem, we simply find the maximum value stored in this auxiliary array.

Now, how do we construct this auxiliary array?

We will fill in each value one by one, in order.

When we fill in the k^{th} element, we must ask ourselves the following question:

Assuming that the last missile intercepted was any of the previous, which of these previous interceptions leads to the maximum number of missile interceptions that end with this current missile?
Here's an example:

Consider filling out the last element in the auxiliary array for the example above.

For each previous missile that is at a height greater than or equal to 61, we must find the one that gives us the maximum number of intercepted missiles.

0	1	2	3	4	5	6	7
80	70	60	50	65	45	60	61

0	1	2	3	4	5	6	7
1	2	3	4	3	5	4	3

We will loop through the array of missile heights, checking to see if those missiles are at height 61 or higher. There are only two. For these two missiles, we check the corresponding entry in the auxiliary array. We see that if 80 is the last missile we take, then by taking 61, we have taken 2 missiles. But then we see that if we take 70 as our last missile, by taking 61 afterwards, we have taken 3 missiles. This is better than 2, so we store 3 in the auxiliary array.

Problems from acm.uva.es site

10131, 10069, 10154, 116, 10003, 10261, 10271, 10201

Take a look at these problems for more practice on dynamic programming.

References

Brassard, Gilles & Bratley, Paul. Fundamentals of Algorithmics (text for COT5405)
Prentice Hall, New Jersey 1996 ISBN 0-13-335068-1

Cormen, Tom, Leiserson, Charles, & Rivest, Ronald. Introduction to Algorithms
The MIT Press,Cambridge, MA 1992 ISBN 0-262-03141-8 (a newer version exists)

Goodrich, Michael & Tamassia, Roberto. Algorithm Design (text for COP3530)
John Wiley & Sons, New York 2002 ISBN 0-471-38365-1

Skiena, Steven & Revilla, Miguel. Programming Challenges
Springer-Verlag, New York 2003 ISBN 0-387-00163-8