CSC345 Discussion 11

Dynamic Programming and
Bones Battle/Column Sort Tournament

Dynamic Programming

Dynamic Programming

Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions using a memory-based data structure

Problem

Imagine you have a homework assignment with different parts labeled A through G. Each part has a “value” (in points) and a “size” (time in hours to complete). For example, say the values and times for our assignment are:

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
time	3	4	2	6	7	3	5

Problem

Say you have a total of 15 hours: which parts should you do?
If there was partial credit that was proportional to the amount of work done (e.g., one hour spent on problem C earns you 2.5 points) then the best approach is to work on problems in order of points/hour (a greedy strategy).
But, what if there is no partial credit? In that case, which parts should you do, and what is the best total value possible?

	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
time	3	4	2	6	7	3	5

This is an example of the Knapsack Problem

In the knapsack problem we are given a set of n items, where each item i is specified by a size s_i and a value v_i .

We are also given a size bound S (the size of our knapsack). The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (they all fit into the knapsack)

The Knapsack Problem

We can solve the knapsack problem in exponential time by trying all possible subsets.

With Dynamic Programming, we can reduce this to time O(nS).

Let’s do this top down by starting with a simple recursive solution.

Let’s start by just computing the best possible total value, and we afterwards can see how to actually extract the items needed.

The Knapsack Problem

This will take exponential time!

```
// Recursive algorithm: either we use the last element or we don't.
Value(n,S)    // S = space left, n = # items still to choose from
{
  if (n == 0) return 0;
  if (s_n > S) result = Value(n-1,S); // can't use nth item
  else result = max{v_n + Value(n-1, S-s_n), Value(n-1, S)};
  return result;
}
```

But, notice that there are only O(nS) different pairs of values the arguments can possibly take on, so this is perfect for memoizing. Let us initialize a 2-d array arr[i][j] to “unknown” for all i,j.

The Knapsack Problem

```
Value(n,S)
{
  if (n == 0) return 0;
  if (arr[n][S] != unknown) return arr[n][S];  // <- added this
  if (s_n > S) result = Value(n-1,S);
  else result = max{v_n + Value(n-1, S-s_n), Value(n-1, S)};
  arr[n][S] = result;                          // <- and this
  return result;
}
```

Since any given pair of arguments to Value can pass through the array check only once, and in doing so produces at most two recursive calls, we have at most 2n(S + 1) recursive calls total, and the total time is O(nS).

The Knapsack Problem

How can we get the items?
As usual for Dynamic Programming, we can do this by just working backwards: if arr[n][S] = arr[n-1][S] then we didn’t use the nth item so we just recursively work backwards from arr[n-1][S].
Otherwise, we did use that item, so we just output the nth item and recursively work backwards from arr[n-1][S-s_n].

Practice Problem

The Longest Common Subsequence (LCS) problem is as follows. We are given two strings: string S of length n, and string T of length m. Our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings.

For example, consider:
S = ABAZDC
T = BACBAD
In this case, the LCS has length 4 and is the string ABAD. Another way to look at it is we are finding a 1-1 matching between some of the letters in S and some of the letters in T such that none of the edges in the matching cross each other.

Practice Problem

Consider the two cases in LCS

Case 1: what if S[i] != T[j]? Then, the desired subsequence has to ignore one of S[i] or T[j] so we have:

$$LCS[i, j] = \max(LCS[i - 1, j], LCS[i, j - 1]).$$

Case 2: what if S[i] = T[j]? Then the LCS of S[1..i] and T[1..j] might as well match them up. For instance, if I gave you a common subsequence that matched S[i] to an earlier location in T, for instance, you could always match it to T[j] instead. So, in this case we have:

$$LCS[i, j] = 1 + LCS[i - 1, j - 1].$$

Practice Problem

This recursive solution to LCS runs in exponential time. Use Dynamic Programming to create an O(mn) solution.

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1); // no harm in matching up
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  return result;
}
```

Solution

```
LCS(S,n,T,m)
{
  if (n==0 || m==0) return 0;
  if (arr[n][m] != unknown) return arr[n][m];  // <- added this line (*)
  if (S[n] == T[m]) result = 1 + LCS(S,n-1,T,m-1);
  else result = max( LCS(S,n-1,T,m), LCS(S,n,T,m-1) );
  arr[n][m] = result;                          // <- and this line (**)
```

Solution

In this memoized version, our running time is now just O(mn). One easy way to see this is as follows. First, notice that we reach line at most mn times (at most once for any given value of the parameters). This means we make at most 2mn recursive calls total (at most two calls for each time we reach that line). Any given call of LCS involves only O(1) work (performing some equality checks and taking a max or adding 1), so overall the total running time is O(mn).