CSE 202 NOTES FOR OCTOBER 11, 2001

Dynamic Programming

Dynamic programming is like backtracking with one additional idea: save your work. If your recursive algorithm is calling itself on identical subproblems an exponential number of times, simply save the answers in some easily-named and constant-time-addressable data structure so that you can avoid all the re-computation. In doing this, you will eliminate the recursive structure of your solution by solving all of the subproblems in a bottom up order.

In other words, we follow these steps:

Find backtracking/recursive solution: Typically the simpler the recursive algorithm you start with, the simpler (more likely you are to find) the dynamic programming algorithm.

Identify and characterize the subproblems: Generally this involves looking at how the recursion works out from a decision tree point of view, and then parameterizing the subproblems.

Rewrite recursion in terms of renaming: This generally removes recursion from the problem.

Identify bottom-up order on parameters: This usually consists of properly initializing whatever data structure you may be using with the appropriate base-cases.

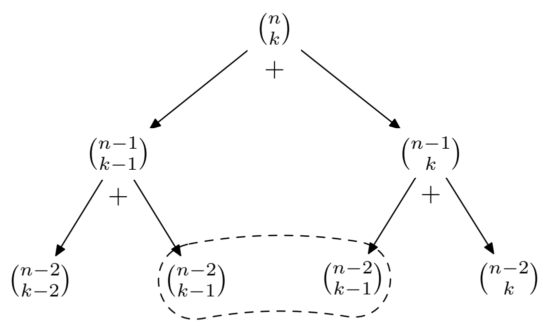
Rewrite the recursive algorithm:

```
Initialize any data structures with base cases
for every subproblem, in bottom-up order do
    do rewritten recursion
end for
return main problem
```

As a toy example, consider the problem of calculating the binomial function: $\binom{n}{k}$ (number of sets S of k elements drawn from a larger set U of n elements). If we took a backtracking approach, we would do something like the following:

```
if k ≤ 0 then
    return 1;
else
    return choose(n − 1,k)+choose(n − 1,k − 1)
end if
```

FIGURE 1. The chain of calls made by the backtracking `choose` function



However, if we were to draw out a tree of the recursive calls made to the choose algorithm, we'd have a situation like figure 1. In this case, we see that two of the calls are the same, namely, the calls to $\binom{n-2}{k-1}$. If we saved this work, we'd only have to do it once. The dynamic programming version of the algorithm is:

```
for m = 1 to n do
  for k = 0 to m do
    if k = 0 ∨ k = m then
      c[k, m] ← 1
    else
      c[k, m] ← c[k − 1, m − 1] + c[k, m − 1]
    end if
  end for
end for
```

The asymptotic time behavior of this algorithm is $O(n^2)$ instead of $O(2^n)$. Of course, there's a linear iterative algorithm for calculating $\binom{n}{k}$ for particular values of n and k:

```
r ← 1
for j = 1 upto k do
  r ← r × (n−j+1)/j
end for
return r
```

CARD COUNTING

Another example of dynamic programming is the age-old cheating technique of counting cards: given a deck of n cards $A[1..n]$, figure out how many hands of length l sum to value t. We will apply the above process.

Find backtracking solution: *What are the decision points?*

The cards in $A[1..n]$.

How does one decision affect the other decisions?

If we include a card, then we decrease the target t by the value of that card, otherwise we change the size of the deck.

Are the subproblems self-similar?

Yes. We have t', a new deck $A[2..n]$ and a hand-length of $l-1$ (if we included the top card, l otherwise).

The backtracking solution is fairly straightforward, then. At a high level,

$$\text{Hands}(A[1..n], l, t) = \begin{cases} \text{Hands}(A[2..n], l, t) \text{ if } A[1] > t, n > 1 \\ \text{Hands}(A[2..n], l, t) + \text{Hands}(A[2..n], l-1, t-A[1]) \text{ if } l > 0, n > 1, A[1] \leq t \\ 1 \text{ if } n = 1, A[1] = t, l = 1 \\ 1 \text{ if } t = 0, l = 0 \\ 0 \text{ otherwise} \end{cases}$$

Which can be restated fairly easily as an algorithm.

Identify and characterize the subproblems: As we call `Hands`, we're changing the size of the array, the value of the target, and the length of the hand. So, the parameters of the recursive solution were A, l, and t, and our new parameters are $1 \leq I \leq n$, $A[I..n]$, $0 \leq l' \leq l$, and $0 \leq t' \leq t \leq lv$ (where v is the maximum value of a card). We can use I, l', and t' as our parameters, and notice that we will fill in a data structure $H[I, l', t']$ as we solve our subproblems. So let's call $H[I, l', t']$ the number of hands summing to t' in $A[I..n]$ of length l'. Notice that we have in no way changed the problem, we've simply renamed parts of it to be more data-structure oriented.

Rewrite the recursion with the renamed subproblems: Our new renamed recursive solution is now:

$$H[I', l', t'] = \begin{cases} H[I+1, l', t'] \text{ if } A[I] > t' \\ H[I+1, l', t'] + H[I+1, l'-1, t'-A[I] \text{ if } l' > 0, I > n, A[I] \leq t' \\ 1 \text{ if } I = n, A[I] = t', l' = 1 \\ 1 \text{ if } t' = 0, l' = 0 \\ 0 \text{ otherwise} \end{cases}$$

Identify the bottom-up order on the solution: In the recursive solution, the index I was increasing. In the DP solution, then, I should be decreasing.

Apply template: We rewrite our solution using the above template.

```
if t > lv then
  return 0;
end if
{Initialization}
Create H[1..n][0..l][0..t]
for I = 1 upto n do
  H[I][0][0] ← 1
end for
H[n][1][A[n]] ← 1 {This corresponds to the t' = A[n] case}
for I = 1 upto n do
  for t' = 1 upto t do
    H[I][0][t'] ← 0
  end for
end for
for l' = 0 upto l do
  for t' = 0 upto t do
    if l' ≠ 0 and t' ≠ 0 then
      H[n][l'][t'] ← 0
    else
      H[n][l'][t'] ← 1
```

```
    end if
  end for
end for
{Computation}
for I = n − 1 downto 1 do
  for l′ = 0 upto l do
    for t′ = 0 upto t do
      if A[I] > t then
        H[I][l′][t′] ← H[I + 1][l′][t′]
      else
        H[I][l′][t′] ← H[I + 1][l′ − 1][t′ − A[I]] + H[I + 1][l′][t′]
      end if
    end for
  end for
end for
return H[1][l][t]
```

The time of this dynamic programming algorithm is $O(nlt) = O(nl^2v)$ where v is the maximum value for any card.

As a side note, memoization is very similar to dynamic programming, except that you use the original recursion, modified by your naming scheme. This is a popular technique in Perl programs because of autovivification, but memoization suffers from lack of locality of reference. In fact, memoization is typically a worse technique except when the overlap between subproblems is sparse, at which point the savings in memory is generally more beneficial than locality of reference.