The Citizen's Guide to Dynamic Programming

Jeff Chen
Stolen extensively from past lectures

October 6, 2007

"Unfortunately, programmer, no one can be told what DP is.
You have to try it for yourself."
–Adapted from *The Matrix*

Dynamic programming, often abbreviated "DP", is a technique for reducing the runtime of algorithms by taking advantage of their repetitive substates. Invented by Richard Bellman in 1953, DP is one of the most powerful problem solving tools you will ever learn. The maxim of DP is: don't do anything twice. It is most often used to simplify recursion problems– problems whose solutions depend on solutions of smaller problems – by storing partial results.

1 Fibonacci [Leonardo Fibonacci de Pisa]

Problem Statement: Given a positive integer N, find the nth Fibonacci number, F_n. Recall that $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ when $n > 2$.

Here recursion is obvious – the definition of Fibonacci is recursive. Let's code it up:

```
int fib(int N)
{
    if(N < 3)
       return 1;
    return (fib(N - 1) + fib(N - 2));
}
```

Unfortunately, our algorithm's runtime is formidable: for $n = 100$, it runs for about 100 *centuries* on the fastest computers to date – slightly edging out USACO's time limit. The reason for this is quickly palpable: when a recursive function calls itself more than once in each instance, the runtime is exponential – the function calls two functions which each call two functions, etc. etc. We note, however, that not all of these calls are necessary. For example, $fib(100)$ calls $fib(98)$ twice. This is a good clue that DP may be helpful. Before that, however, let's re-code our recursive solution:

```
int fib(int N)
{
   if(N < 3)
     return 1;
   if(!alreadyFound[N])
   {
     storedValue[N] = fib(N - 1) + fib(N - 2);
     alreadyFound[N]=1;
     }
   return storedValue[N];
}
```

Note two things: we never branch out on a $fib(n)$ that we already know, and we save each value of n that we have calculated. The latter is an example of *memoization* – a characteristic of dynamic programming in which the values of substates are stored for later use. The problem with this implementation is that it's still recursion; i.e. you must place many functions on the stack, which can overflow. Now what do we do?

Were You Aware? Recursion begins with the target state and recalls substates – from the top of the 'pyramid' down. *Computation in order* DP does the opposite – it starts at the lowest substate and iterates upwards until it reaches the answer. This should be pretty intuitive – manually, to find the n-th fibonacci number, you would start from F_2 and build your way up. If you prefer to calculate your fibonacci values by hand recursively, you are clinically insane.

In this case, we can have an array $F(n)$=the n-th fibonacci number. $F(1) = 1$, $F(2) = 1$. Now we can get $F(3)$, and then $F(4)$, and so on. So we keep an array of stored answers and iterate up to the desired value – $O(N)$ time, $O(N)$ memory.

```
int fib(int N)
{
   if(N < 3)
     return 1;
   F[1]=1;
   F[2]=1;
   for (int i = 3; i <= N; i++)
      F[i]=F[i-1]+F[i-2];
   return F[i];
}
```

This is already a good solution, but as problem-solvers, we constantly seek for optimization. On most computers, including the USACO grader, bytes are much more expensive than operations – for example, USACO gives a limit of 16MB compile-time memory, 1MB stack memory, and 1 second, or around 10,000,000 operations. It would be nice if we could decrease the memory consumption. But sir, how can we do that? Simple, children. We notice that each stored value is only used for the next two values, and then ignored. So why not kill the array too? We can store only the latest two values: this is loosely defined as the *sliding window trick*.

```
int fib(int N)
{
  if(N < 3) return 1;
  int a=1, b=1, c=2;
  for(int i=4;i<=N;i++)
  {
    a=b;
    c+=b;
    b=c-b;
   }
  return c;
}
```

We have now transformed an exponential time, linear memory problem to one of linear time and constant memory. Keep in mind, however, that you can't sliding window every single problem.

2 Biggest Sum [Traditional]

Problem Statement: Given an array of N integers, find the largest *sum* of a consecutive subsequence of this array. Note that the integers are not necessarily positive.

You can brute force this if you wish, but the complexity is $O(N^2)$. Let's try a better solution. The first step is to define a function which solves the problem, so let's have $f(m)$ denote the largest sum of a consecutive subsequence that ends at position m. Then, suppose we want to find $f(m+1)$. There are two possibilities: we could use only the $m+1$-st number to get a sequence of 1, or we append the $m+1$-st element to a subsequence that ends at m. Note that we don't give a hoot what this subsequence consists of, only what its size is. Thus, given an array of numbers a, $f(m+1) = \max(a_{m+1}, a_{m+1} + f(m))$, or f(m+1). So the recursive function is

```
int biggest(int m)
{
   if(m == 0)
      return a[0];
   return max(a[m], a[m] + f(m - 1));
}
```

Okay, let's memoize:

```
int biggest(int m)
{
   if(!alreadyFound[m])
   {
      alreadyFound[m]=1;
   if(m == 0)
      storedValue[m]=a[0];
   else storedValue[m]=max(a[m], a[m] + f(m - 1));
   }
   return storedValue[m];
}
```

Now we will flip the pyramid and change this to a computation in order DP:

```
int biggest(int m)
{
   if (m == 0) return a[0];
   storeF[0] = a[0];
   for(int i = 1; i <= m; i++)
      storeF[i] =  max(a[i], a[i] + storeF[i - 1]);
   return storeF[m];
}
```

Nice! The state is exactly the same; we simply started from the bottom to kill the recursion. Let's see what else we can do. What we are really looking for is the maximum value of $f(m)$. We find $f(m)$ by finding $f(m-1)$. Hence, we only need those two: sliding window time. Let b denote $f(m-1)$, so $f(m)$ is just $a_m + \max(0, b)$. So, our final code is as follows:

```
int biggest(int m)
{
   int best = a[0];
   int b = a[0];
   for (int i = 1; i <= m; i++)
   {
      b = a[i] + max(0, b);
      if (b > best) best = b;
   }
   return best;
}
```

3 Number Triangles [Traditional; Kolstad]

Problem Statement: Given a triangle of numbers, find the path from top to bottom with the greatest sum. For example, in the triangle

```
   1
  2 3
 1 5 9
9 1 1 1
```

the best path is 1-3-9-1, with sum 14.

Once again, we define a function. This time, we have 2 dimensions, but that's not much variation: let $best(r, c)$ be the sum of the best path that ends at the c-th element of row r. You should be reminded of Pascal's Triangle at this point – each value is built up from the 2 above it. This is exactly the same. To get to (r, c), we may take a path either from $(r-1, c-1)$ or $(r-1, c)$. We want to take the one that yields the bigger sum, thus

$$best(r, c) = a_{r,c} + \max(best(r-1, c-1), best(r-1, c)),$$

where $a_{r,c}$ is the element in the c-th column of the r-th row of the triangle. Don't forget the trivial base case $best(0, 0) = a_{0,0}$.

```
int solve()
{
   int i, j, max;
   best[0][0] = a[0][0];
   for (i=1; i<N; i++)
   {
         best[i][0] = a[i][0] + best[i-1][0];
         for (j=1; j<i ; j++)
              best[i][j] = a[i][j] + max(best[i-1][j], best[i-1][j-1]);
         best[i][i] = a[i][i] + best[i-1][i-1];
   }
   max = best[N - 1][0];
   for (i=1; i<N; i++)
         if (best[N - 1][i] > max)
              max = best[N - 1][i];
   return max;
}
```

4 Integral Knapsack [Traditional]

Problem Statement: Given an array of items with values V, each with a weight W, what is the total maximum value we can pick up without exceeding a weight C, assuming that we have an infinite supply of each item?

This is different than the fractional knapsack in that greedy will no longer consistently give the optimal answer, since we must pick whole items. Instead we must employ a DP. Let us first define the function. We will have $f(n)$=the max value for a weight of n, and $f(0) = 0$. Now we know $f(n)$ is made by combining items of value $V(i)$. Clearly it can only be made with combinations of items whose weights are each $< n$. Let's have an example problem. The weights of the items will be 1,5 and 10, while their respective values will be 2, 3 and 8. Let's say we are looking for $f(7)$. We will only have to consider the 1 and 5-weight items. The key observation is that since we are processing f iteritavely, we already know the values of $f(7-1)$ and $f(7-5)$. For each of these functions, adding the respective weight yields our desired function $f(7)$. If $f(2) = 5$, then by adding a 5-weight item, $f(7) = 5 + 3$. We can just loop through the items and find the largest $f(7)$ made this way. So we have solved it! Our dp is thus w=weights, v=values, and $f(i) = max(v_j + f(i - w_j)|w_j <= i)$. I'm not going to code this up for you; do it yourself, you lazy buffoons.

5 Problems

1. [Traditional] Given a grid, possibly missing some segments, compute the number of ways to go from the lower left corner to the upper right corner in the minimal distance.
2. [USACO/Traditional] Given V different coin values, compute the number of ways to construct a value N.
3. [USACO TP] Given an NxN grid of 0s and 1s, find the largest square of 1s in the array.

Final Tip

DP is unique in its pristine simplicity – but don't be fooled. Like tetris, Dynamic Programming is easy to learn but difficult to master. Practice, practice, practice. It is important to build intuition.