Dynamic Programming I

Senior Computer Team

October 29, 2004

Dynamic programming, often abbreviated "DP", is a technique to efficiently solve recursion problems – problems whose solutions depend on solutions of smaller problems – by storing partial results. Many times, it is obvious that a given problem is a recursion problem. However, it may not be so obvious that using DP will tremendously speed up the solution. We start by looking at an example.

1 Fibonacci [Leonardo of Pisa]

Given a positive integer N, find the N-th Fibonacci number, F_N. Recall that these are defined as $F_1 = 1$, $F_2 = 1$, $F_N = F_{N-1} + F_{N-2}$ when $N > 2$.

Here recursion is obvious: by knowing previous Fibonacci numbers we can find a given one. Consider this program:

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

This program correctly computes the N-th Fibonacci number. Alas, its runtime is formidable; for $N = 100$, it runs for about 100 *centuries* on the fastest computers to date! The reason for this is plain: when a recursive function calls itself more than once in each instance, the runtime is exponential– for example, $fib(100)$ calls $fib(98)$ twice, which is a waste the second time, because we could have stored the answer the first time. That is the point of DP - *avoid solving the same subproblem twice*. So lets keep an array of stored answers, and check this before trying to solve any problem, then store whenever we do compute something:

```
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];
}
```

This is an example of *memoization* – where the recursion prototype is kept, but values that are computed are stored.

This implementation of dynamic programming has a flaw: you must place many functions on the stack, which can overflow it if you call many functions and each requires a fair amount of memory. Another way to do dynamic programming, which avoids this problem, is *computation in order*. We notice that, to compute a given Fibonacci number, we could care less what the later Fibonacci numbers are; we just want to know the previous ones. So, if we compute the Fibonacci numbers in order, we can eliminate recursion altogether.

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

This function takes linear time and memory. On most computers, including the USACO grader, bytes are much more expensive than operations– for example, USACO gives a limit of 16 megabytes and 1 second, or a couple hundred million operations. Therefore it would be nice if we could decrease the memory consumption. Notice that each stored value is only used for the next two values, and then ignored. Therefore we could recycle memory, only storing the last two values. This is called the *sliding window trick*.

```
int fib(int N)
{
   if (N < 3) return 1;
   int a = 1;
   int b = 1;
   int c = 2;
```

```
   for (int i = 4; i <= N; i++)
   {
      a = b;
      c += b;
      b = c - b;
   }
   return c;
}
```

This program, instead of linear memory and exponential time, now takes constant memory and linear time. It can be improved, however– see the problems at the end of this lecture.

2 Biggest Sum [Traditional]

Given an array of N numbers, find the largest sum of a consecutive subsequence of this array (the numbers are not necessarily positive).

It may not be obvious to you that this is a recursion problem. Don't worry: you are not alone. The most difficult part of solving DP problems usually is identifying them as such. For now, we will tell you what the recursive function is; on contests you will be on your own.

Let $f(m)$ 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: either we take a subsequence of length 1, i.e., just the $m+1$-st element, or we append the $m+1$-st element to a subsequence that ends at m. The largest value of such a subsequence is $f(m)$. Thus, $f(m+1) = \max(a_{m+1}, a_{m+1} + f(m))$ (a is the array). So the recursive function is

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

The in-order DP formulation thus is

```
int f(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];
}
```

To solve our problem, what we are really looking for is the maximum value of $f(m)$, since the sequence has to end somewhere. Also, a sliding window approach is handy here because to find $f(m)$ we only need $f(m-1)$. Thus, if b denotes $f(m-1)$, $f(m)$ is just $a_m + \max(0, b)$. So, the final piece of code, which computes the biggest value of a consecutive subsequence is as follows:

```
int biggest()
{
   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;
}
```

Our next example illustrates two-dimensional dynamic programming.

3 Number Triangles [Traditional; Kolstad]

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. We jump straight to the DP solution, as we assume that the reader got the hang of how to translate recursive paradigms into DP programs.

Let $best(r, c)$ be the sum of the best path that ends at the c-th element of row r. 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. We also have to take into account the boundary condition that $best(0, 0) = a_{0,0}$.

Here comes the DP: to compute *best* for a certain row, we only need to know the previous rows. Thus, we can find the *best* array in order. Namely,

```
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;
}
```

As an exercise, figure out how to apply the sliding window trick to this problem.

4 Problems

1. Consider the matrix $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Prove that the top-left element of A^n is F_{n+1}.

2. Using the result of problem 1, devise an algorithm that runs in $O(\log N)$ time to compute F_N.

3. Suppose that we have a circle of N numbers. Write an $O(N)$ algorithm that finds the consecutive block with the largest sum.

4. [USACO Camp 2003] Given a number triangle, compute the highest score you can achieve in two paths down, where you only count intersections of paths once ($N \leq 100$).

5. [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.