20bits

- About
- Writing

Introduction to Dynamic Programming

by Jesse Farmer on Saturday, November 15, 2008

Dynamic programming is a method for efficiently solving a broad range of search and optimization problems which exhibit the characteristics of overlappling subproblems and optimal substructure. I'll try to illustrate these characteristics through some simple examples and end with an exercise. Happy coding!

Contents

Overlapping Subproblems

A problem is said to have overlapping subproblems if it can be broken down into subproblems which are reused multiple times. This is closely related to recursion. To see the difference consider the factorial function, defined as follows (in Python):

```
def factorial(n):
        if n == 0: return 1
        return n*factorial(n-1)
```

Thus the problem of calculating `factorial(n)` depends on calculating the subproblem `factorial(n-1)`. This problem does **not** exhibit *overlapping* subproblems since `factorial` is called exactly once for each positive integer less than n.

Fibonacci Numbers

The problem of calculating the n^{th} Fibonacci number does, however, exhibit overlapping subproblems. The naÃ¯ve recursive implementation would be

```
def fib(n):
        if n == 0: return 0
        if n == 1: return 1
        return fib(n-1) + fib(n-2)
```

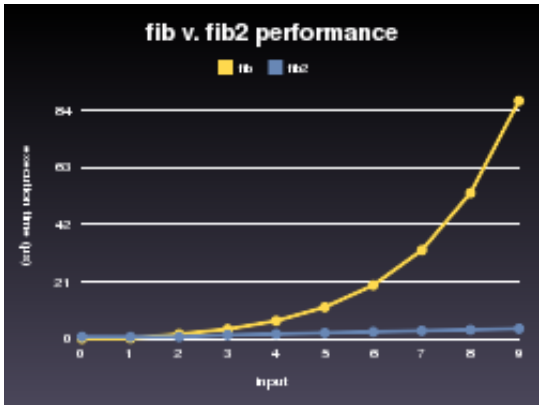
The problem of calculating `fib(n)` thus depends on both `fib(n-1)` and `fib(n-2)`. To see how these subproblems overlap look at how many times fib is called and with what arguments when we try to calculate `fib(5)`:

```
fib(5)
fib(4) + fib(3)
fib(3) + fib(2) + fib(2) + fib(1)
fib(2) + fib(1) + fib(1) + fib(0) + fib(1) + fib(0) + fib(1)
fib(1) + fib(0) + fib(1) + fib(1) + fib(0) + fib(1) + fib(0) + fib(1)
```

At the k^{th} stage we only need to know the values of `fib(k-1)` and `fib(k-2)`, but we wind up calling each multiple times. Starting from the bottom and going up we can calculate the numbers we need for the next step, removing the massive redundancy.

```
def fib2(n):
        n2, n1 = 0, 1
        for i in range(n-2):
                n2, n1 = n1, n1 + n2
        return n2+n1
```

In Big-O notation the `fib` function takes $O(c^n)$ time, i.e., exponential in n, while the `fib2` function takes O(n) time. If this is all too abstract take a look at this graph comparing the runtime (in microseconds) of `fib` and `fib2` versus the input parameter.



The above problem is pretty easy and for most programmers is one of the first examples of the performance issues surrounding recursion versus iteration. In fact, I've seen many instances where the Fibonacci example leads people to believe that recursion is inherently slow. This is not true, but in cases where we can define a problem with overlapping subproblems recursively using the above technique will always reduce the execution time.

Now, for the second characteristic of dynamic programming: optimal substructure.

Optimal Substructure

A problem is said to have optimal substructure if the globally optimal solution can be constructed from locally optimal solutions to subproblems. The general form of problems in which optimal substructure plays a roll goes something like this. Let's say we have a collection of objects called A. For each object o in A we have a "cost," c(o). Now find the subset of A with the maximum (or minimum) cost, perhaps subject to certain constraints.

The brute-force method would be to generate every subset of A, calculate the cost, and then find the maximum (or minimum) among those values. But if A has n elements in it we are looking at a search space of size 2^n if there are no constraints on A. Oftentimes n is huge making a brute-force method computationally infeasible. Let's take a look at an example.

Maximum Subarry Sum

Let's say we're given an array of integers. What (contiguous) subarray has the largest sum? For example, if our array is [1,2,-5,4,7,-2] then the subarray with the largest sum is [4,7] with a sum of 11. One might think at first that this problem reduces to finding the subarray with all positive entries, if one exists, that maximizes the sum. But consider the array [1,5,-3,4,-2,1]. The subarray with the largest sum is [1, 5, -3, 4] with a sum of 7.

First, the brute-force solution. Because of the constraints on the problem, namely that the subsets under consideration are contiguous, we only have to check $O(n^2)$ subarrays (why?). Here it is, in Python:

```
def msum(a):
        return max([(sum(a[j:i]), (j,i)) for i in range(1,len(a)+1) for j in range(i)])
```

This returns both the sum and the offsets of the subarray. Let's see if we can't find an optimal substructure to exploit.

We are given an input array `a`. I'm going to use Python notation so that `a[0:k]` is the subarray starting at 0 and including every element up to and including `k-1`. Let's say we know the subarray of `a[0:i]` with the largest sum (and that sum). Using just this information can we find the subarray of `a[0:i+1]` with the largest sum?

Let a[j:k+1] be the optimal subarray, t the sum of a[j:i], and s the optimal sum. If t+a[i] is greater than s then set a[j:i+1] as the optimal array and set s = t. If t + a[i] is negative, however, the contiguity constraint means that we cannot include a[j:i+1] in our subarray since any such subarray will have a smaller sum than a subarray without it. So, if t+a[i] is negative set t = 0 and set the left-hand bound of the optimal subarray to i+1.

To visualize consider the array [1,2,-5,4,7,-2].

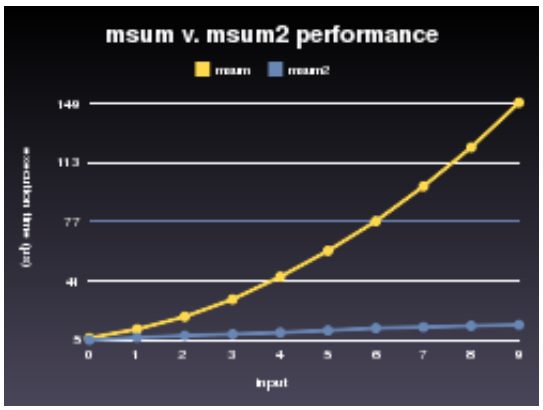
```
Set s = -infinity, t = 0, j = 0, bounds = (0,0)
(1   2  -5   4   7  -2
(1)| 2  -5   4   7  -2  (set t=1.  Since t > s, set s=1 and bounds = (0,1))
(1   2)|-5   4   7  -2  (set t=3.  Since t > s, set s=3, and bounds = (0,2))
 1   2  -5(| 4   7  -2  (set t=-2. Since t < 0, set t=0 and j = 3 )
 1   2  -5  (4)| 7  -2  (set t=4.  Since t > s, set s=4 and bounds = (3,4))
 1   2  -5  (4   7)|-2  (set t=11. Since t > s, set s=11 and bounds = (3,5))
 1   2  -5  (4   7) -2| (set t=9.  Nothing happens since t < s)
```

This requires only one pass through the array and at each step we're only keeping track of three variables: the current sum from the left-hand edge of the bounds to the current point (t), the maximal sum (s), and the bounds of the current optimal subarray (bounds). In Python:

```
def msum2(a):
        bounds, s, t, j = (0,0), -float('infinity'), 0, 0

        for i in range(len(a)):
                t = t + a[i]
                if t > s: bounds, s = (j, i+1), t
                if t < 0: t, j = 0, i+1
        return (s, bounds)
```

In this problem the "globally optimal" solution corresponds to a subarray with a globally maximal sum, but at each each step we only make a decision relative to what we have already seen. That is, at each step we know the best solution *thus far*, but might change our decision later based on our previous information and the current information. This is the sense in the problem has optimal substructure. Because we can make decisions locally we only need to traverse the list once, reducing the run-time of the solution to O(n) from $O(n^2)$. Again, a graph:



The Knapsack Problem

Let's apply what we're learned so far to a slightly more interesting problem. You are an art thief who has found a way to break into the impressionist wing at the Art Institute of Chicago. Obviously you can't take everything. In particular, you're constrained to take only what your knapsack can hold — let's say it can only hold W pounds. You also know the market value for each painting. Given that you can only carry W pounds what paintings should you steal in order to maximize your profit?

First let's see how this problem exhibits both overlapping subproblems and optimal substructure. Say there are n paintings with weights w_1, ..., w_n and market values v_1, ..., v_n. Define A(i,j) as the maximum value that can be attained from considering only the first i items weighting at most j pounds as follows.

Obviously A(0,j) = 0 and A(i,0) = 0 for any $i \leq n$ and $j \leq W$. If $w_i > j$ then A(i,j) = A(i-1, j) since we cannot include the i^{th} item. If, however, $w_i \leq j$ then A(i,j) then we have a choice: include the i^{th} item or not. If we do not include it then the value will be A(i-1, j). If we do include it, however, the value will be v_i + A(i-1, j - w_i). Which choice should we make? Well, whichever is larger, i.e., the maximum of the two.

Expressed formally we have the following recursive definition

$$A(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ A(i-1,j) & \text{if } w_i > j \\ \max\{A(i-1,j), v_i + A(i-1, j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

This problem exhibits both overlapping subproblems and optimal substructure and is therefore a good candidate for dynamic programming. The subproblems overlap because at any stage (i,j) we might need to calculate A(k,l) for several k < i and l < j. We have optimal substructure since at any point we only need information about the choices we have already made.

The recursive solution is not hard to write:

```
def A(w, v, i,j):
    if i == 0 or j == 0: return 0
    if w[i-1] > j:  return A(w, v, i-1, j)
    if w[i-1] <= j: return max(A(w,v, i-1, j), v[i-1] + A(w,v, i-1, j - w[i-1]))
```

Remember we need to calculate A(n,W). To do so we're going to need to create an n-by-W table whose entry at (i,j) contains the value of A(i,j). The first time we calculate the value of A(i,j) we store it in the table at the appropriate location. This technique is called memoization and is one way to exploit overlapping subproblems. There's also a Ruby module called memoize which does it for Ruby.

To exploit the optimal substructure we iterate over all i <= n and j <= W, at each step applying the recursion formula to generate the A(i,j) entry by using the memoized table rather than calling A() again. This gives an algorithm which takes O(nW) time using O(nW) space and our desired result is stored in the A(n,W) entry in the table.

Everyday Dynamic Programming

The above examples might make dynamic programming look like a technique which only applies to a narrow range of problems, but many algorithms from a wide range of fields use dynamic programming. Here's a very partial list.

1. The Needleman-Wunsch algorithm, used in bioinformatics.
2. The CYK algorithm which is used the theory of formal languages and natural language processing.
3. The Viterbi algorithm used in relation to hidden Markov models.
4. Finding the string-edit distance between two strings, useful in writing spellcheckers.
5. The D/L method used in the sport of cricket.

That's all for today. Cheers!