Computer Science 431
Algorithms
The College of Saint Rose
Spring 2015

Topic Notes: Dynamic Programming

We next consider *dynamic programming*, a technique for designing algorithms to solve problems by setting up recurrences with overlapping subproblems (smaller instances), solving those smaller instances, remembering their solutions in a table to avoid recomputation, then using the subproblem solutions from the table to obtain a solution to the original problem.

The idea comes from mathematics, where "programming" means "planning".

Simple Example: Fibonacci Numbers

You have certainly seen the Fibonacci sequence before, defined by:

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0$$
$$F(1) = 1$$

A direct computation using this formula would involve recomputation of many Fibonacci numbers before $F(n)$.

But if instead, we store the answers to the subproblems in a table (in this case, just an array), we can avoid that recomputation. For example, when we compute $F(n)$, we first need $F(n-1)$, then $F(n-2)$. But the computation of $F(n-1)$ will also have computed $F(n-2)$. With a dynamic programming technique, that answer will have been stored, so $F(n-2)$ is immediately available once we have computed it once.

With the Fibonacci sequence, we can take a complete "bottom up" approach, realizing that we will need answers to all smaller subproblems ($F(0)$ up to $F(n-1)$) in the process of computing $F(n)$, we can populate an array with 0 in element 0, 1 in element 1, and all successive elements with the sum of the previous two. This makes the problem solvable in linear time, but uses linear space.

Moreover, with this approach, we need only keep the most recent two numbers in the sequence, not the entire array. So we can still get a linear time solution constant space.

Binomial Coefficients

Another example you've probably seen before is the computation of *binomial coefficients*: the values $C(n, k)$ in the binomial formula:

$$(a+b)^n = C(n,0)a^n + \cdots + C(n,k)a^{n-k}b^k + \cdots + C(0,n)b^n.$$

These numbers are also the n^{th} row of Pascal's triangle.

The recurrences that will allow us to compute C(n,k):

$$\begin{aligned} C(n,k) &= C(n-1,k-1) + C(n-1,k) \quad \text{for } \mathrm{n} > \mathrm{k} > 0 \\ C(n,0) &= 1 \\ C(n,n) &= 1 \end{aligned}$$

As with the Fibonacci recurrence, the subproblems are overlapping – the computation of $C(n-1,k-1)$ and $C(n-1,k)$ will involve the computation of some of the same subproblems. So a dynamic programming approach is appropriate.

The following algorithm will fill in the table of coefficients needed to compute $C(n,k)$:

```
binomial(n,k)
  for i=0 to n
    for j=0 to min(i,k)
      if j==0 or j==i
        C[i][j] = 1
      else
        C[i][j] = C[i-1][j-1] + C[i-1][j]
  return C[i][k]
```

It's been a while since we did a more formal analysis of an algorithm's efficiency.

The basic operation will be the addition in the `else`. This occurs once per element that we compute. In the first $k+1$ rows, the inner loop executes i times (the first double summation in the formula below). For the remaining rows, the inner loop executes $k+1$ times (the second double summation). There are no differences in best, average, and worst cases: we go through the loops in their entirety regardless of the input.

So we can compute the number of additions $A(n,k)$ as:

$$\begin{aligned} A(n,k) &= \sum_{i=1}^{k}\sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n}\sum_{j=1}^{k} 1 \\ &= \sum_{i=1}^{k}(i-1) + \sum_{i=k+1}^{n} k \\ &= \frac{(k-1)k}{2} + k(n-k) \in \Theta(nk). \end{aligned}$$

Knapsack Problem

We now revisit the *knapsack problem*, which we first considered with a brute-force approach.

Recall the problem:

Given n items with weights $w_1, w_2, ..., w_n$ and values $v_1, v_2, ..., v_n$, what is the most vaulable subset of items that can fit into a knapsack that can hold a total weight W.

When we first considered the problem, we generated all possible subsets and selected the one that resulted in the largest total value where the sums of the weights were less than or equal to W.

To generate a more efficient approach, we use a dynamic programming approach. We can break the problem down into overlapping subproblems as follows:

Consider the instance of the problem defined by first i items and a capacity j ($j \leq W$).

Let $V[i, j]$ be optimal value of such an instance, which is the value of an optimal solution considering only the first i items that fit in a knapsack of capacity j.

We can then solve it by considering two cases: the subproblem that includes the i^{th} item, and the subproblem that does not.

If we are not going to include the i^{th} item, the optimal subset's value (considering only items 0 through $i-1$) would be $V[i-1, j]$.

If we do include item i (which is only possible if $j - w_i \geq 0$), its value is obtained from the optimal subset of the first $i-1$ items that can fit within a capacity of $j - w_i$, as $v_i + V[i-1, j-w_i]$.

It's also possible that the i^{th} item does not fit (where $j - w_1 < 0$), in which case the optimal subset is $V[i-1, j]$.

We can formulate these into a recurrence:

$$V[i,j] = \begin{cases} \max\{V[i-1,j], v_i + V[i-1, j-w_i]\} & \text{if } j - w_1 \geq 0 \\ V[i-1,j] & \text{if } j - w_i < 0 \end{cases}$$

Combine with some initial conditions:

$V[0, j] = 0$ for $j \geq 0$, $V[i, 0] = 0$ for $i \geq 0$.

The solution to our problem is the value $V[n, W]$.

The text has an example in Figure 8.5, where the solution is obtained in a "bottom up" fashion, filling in the table row by row or column by column, until we get our ultimate answer at in the lower right corner. This guarantees we will compute each subproblem exactly once and is clearly $\Theta(nW)$ in time and space.

One problem with that approach is the fact that we are computing some subproblems that will never be used in a recurrence starting from $V[n, W]$.

An alternate approach, this time working from the top (the solution) down (the subproblems), is to solve each subproblem as it's needed. But...to avoid recomputing subproblems, we remember the answers to subproblems we have computed and just use them when they exist.

The approach uses a technique called *memory functions*. These work just like regular functions, but do what we described above. At the start of the function, it looks to see if the requested instance has already been solved. If so, we just return the previously-computed result. If not, we compute it, save it in the table of answers in case it's needed again, then return the answer.

The following pseudocode implements this memory function idea for the knapsack problem:

```
Globals:
W[1..n]: item weights
values[1..n]: item values
V[0..n][0..W]: subproblem answers, starting all 0's in first row/col,
               -1's elsewhere
mf_knapsack(i,j)
  if (V[i][j] < 0)
    if (j < W[i])
      V[i][j] = mf_knapsack(i-1, j)
    else
      V[i][j] = max(mf_knapsack(i-1, j),
                    values[i] + mf_knapsack(i-1, j-W[i]))
  return V[i][j]
```

Applying this approach to the text's example results in only 11 of the 20 values that were not initial conditions are computed, but only one value is reused (Figure 8.6). Larger instances would result in more subproblems being unneeded and more being reused.