1 Dynamic Programming: Take 1

We examine problems that can be solved by dynamic programming, perhaps the single most useful algorithmic design technique. Dynamic programming can be explained many ways. Rather than explain what a dynamic programming algorithm is, we explain how one might develop one:

1. Find a recursive algorithm for the problem. It often helps to first find a recursive algorithm to count the number of feasible solutions.

2. Spot redundancy in the calculation

3. Eliminate the redundancy. This can often be best accomplished by performing the calculations bottom up, instead of top down.

2 Fibonacci Numbers and Binomial Coefficients

See section 1.2.2 and section 3.1 from the text.

3 Longest Common Subsequence Problem

The input to this problem is two sequences $A = a_1, \ldots, a_m$ and $B = b_1, \ldots, b_n$. The problem is to find the longest sequence that is a subsequence of both A and B. For example, if $A = ccdedcdec$ and $B = cdedcdec$, then $cddec$ is subsequence of length 5 of both sequences. Let $T(i, j)$ be the length of the longest common subsequence of $a_1, \ldots, a_i$ and $b_1, \ldots, b_j$. Then one can see that if $a_i = b_j$ then $T(i, j) = T(i-1, j-1) + 1$. Otherwise, one can see that $T(i, j) = \max(T(i, j-1), T(i-1, j))$.

Note that there are only nm possible subproblems since there are only m choices for i and n choices for j. Hence, by treating the $T(i, j)$'s as array entries (instead of recursive calls) and updating the table in the appropriate way we can get the following $O(n^2)$ time algorithm.

```
For i=0 to m do T(i,0)=0

For j=0 to n do T(0,j)=0

For i=1 to m do
    For j= 1 to n do
        if a(i) = b(j) then T(i,j)=T(i-1,j-1) + 1
        else T(i,j)= MAX( T(i, j-1), T(i-1, j))
```

4 Computing Number of Binary Search Trees and Optimal Binary Search Trees

Section 3.5 from the text.

5 Chained Matrix Multiplication

Section 3.4 from the text.

6 Maximum Weight Independent Set in a Tree

The input to this problem is a tree T with weights on the vertices. The goal is to find the independent set in T with maximum aggregate weight. An independent set is a collection of mutually nonadjacent vertices.

Root the tree at an arbitrary node r, and process the tree in postorder. We generalize the induction hypothesis. Consider an arbitrary node v with branches to k descendants $w_1, w_2, \ldots, w_k$. We can create an independent set for the subtree rooted at v in essentially two ways depending on whether or not we include v in the independent set. If we decide not to include v, Consider an arbitrary root v with branches to k descendants $w_1, w_2, \ldots, w_k$. We can create an independent set for the subtree rooted at v in essentially

two ways depending on whether or not we include v in the independent set. If we decide not to include v, we can combine any independent sets for the subtrees rooted at $w_1, w_2, \ldots, w_k$ to create an independent set for the subtree rooted at v since there are no edges between the subtrees. On the other hand, if we **do** include v in the independent set we can only use independent sets for the subtrees that do **not** include their respective root $w_1, w_2, \ldots, w_k$. Otherwise we would have both v and some w_j in the set and it would not be an independent set anymore.

Therefore for each node v the algorithm computes the following information:

1. $big(v)$ = the maximum weight of an independent set for the subtree rooted at v, and
2. $bignotroot(v)$ = the maximum weight of an independent set for the subtree rooted at v that does **not** include v.

At node v, the algorithm first recursively computes $big(w_i)$ and $bignotroot(w_i)$ for each descendant subtree $w_1, w_2, \ldots, w_k$. It then computes $bignotroot(v)$ and $big(v)$ using the following recurrence relations that correspond to the two cases identified above:

$$bignotroot(v) = \sum_{i=1}^{k} big(w_i)$$

$$big(v) = \max(bignotroot(v), weight(v) + \sum_{i=1}^{k} bignotroot(w_i)$$

If v is a leaf then $bignotroot(v) = 0$ and $big(v) = weight(v)$.

7 Longest Increasing Subsequence

The longest increasing subsequence (LIS) problem is defined as follows.

INPUT: A sequence $X = x_1, \ldots, x_n$ of integers

OUTPUT: The longest increasing subsequence of X. A sequence is increasing if each number in the sequence is larger than the previous number.

Let us try to develop a recursive algorithm for this problem. Let $LIS(k)$ be the longest increasing subsequence of the first k integers in X. Consider the following first stab at an induction hypothesis:

Induction Hypothesis 1: We know how to compute $LIS(k)$.

Assume that we have computed $LIS(k-1)$, and we now want to compute $LIS(k)$. The following sequence $1, 4, 2, 3$ shows that we can't just know any longest sequence. Before we consider 3, we need to know the LIS $1, 2$, not the LIS $1, 4$. The obvious solution is to remember the LIS that ends in the smallest number. Hence we change the definition of LIS as follows: Let $LIS(k)$ be the the longest increasing subsequence of the first k integers in X that ends in the smallest number.

Induction Hypothesis 2: We know how to compute $LIS(k)$.

The following sequence $12, 13, 14, 1, 2, 4$ shows that knowing the LIS that ends in the smallest number is not sufficient information. Notice that in this example the length of the LIS doesn't change, but a new one is formed with a smaller last number. The most obvious way to fix this problem is to remember the best (the one that ends in the smallest number) sequence of length one less than the LIS. Let $LIS2(k)$ be the the increasing subsequence of the first k integers in X that has length one less than the length of $LIS(k)$ and that ends in the smallest number.

Induction Hypothesis 3: We know how to compute $LIS(k)$, and $LIS2(k)$.

The following sequence $12, 13, 14, 15, 1, 2, 3$ shows that there is not sufficient inductive information to compute $LIS2(k)$. We need to know the sequence of the first $k-1$ integers in X of length two shorter than $LIS(k-1)$ that ends in the smallest number. One can see that eventually we have to inductively know the sequence of each length that ends in the smallest number.

Hence, we define $LIS(k, s)$ as the the smallest last number of any subsequence of length s from among the first k numbers of X. This then leads us to the following algorithm.

```
For k= 0 to n do
    for s= 1 to n do
          LIS(k, s) = plus infinity
```

```
For k= 0 to n do
     LIS(k, 0) = minus infinity

For k= 1 to n do
     for s= 1 to n do
          if ( LIS(k-1, s-1) < x(k) < LIS(k-1, s)
                 then LIS(k,s)=x(k)
                 else LIS(k,s)=LIS(k-1,s)
```

This is an example of where when trying to compute something recursively/inductive it is sometimes easier to compute more information than you need. This is sometimes called *strengthening the inductive hypothesis*. This is not paradoxical because the strengthening of the inductive hypothesis also gives you more information to work with when the recursion returns from the smaller subproblem. The most common mistake made here is to use the additional information returned from the recursion to solve the original problem, not the more generalized problem.

One point that I want you to draw from this example is that its not always easy to determine how one should strengthen the inductive hypothesis.

8 Dynamic Programming: Take 2

In this section we discuss another method for developing dynamic programming algorithms that avoids having to develop a recursive algorithm, which is almost always the most difficult part in the previously outlined method. This method is particularly applicable when the feasible solutions are subsets or subsequences. The steps to developing a dynamic programming algorithm using this method are as follows:

1. Determine how to generate all possible feasible solutions. Once again it might be easier to first determine how to count the number of feasible solutions. Enumeration usually follows easily from counting.

2. Develop a pruning rule to eliminate partial feasible solutions that either are redundant, or that can not be extended to an optimal solution.

3. Transform the algorithm to an iterative table based algorithm.

9 Subset Sum Example

The subset sum problem is defined as follows:

INPUT: A collection $x_1, \ldots, x_n$ of positive integers, and a positive integer L.
OUTPUT: A subset of the x_i's that sum to L, or a statement that no such subset exists.

We can generate the subsets inductively using a binary tree as follows. The nodes of depth i are all subsets of $x_1, \ldots, x_i$. The children of a node S at depth $i-1$ are S and $S \cup \{x_i\}$. Thus all the 2^n subsets can be found at the leaves. The depth of this tree is $n+1$.

The two pruning rules are

1. Eliminate the subtree rooted at any subset with sum greater than L.

2. If there are two subsets S and T at the same depth, with the same aggregate sum, then one can arbitrary select one of the two nodes and eliminate the subtree rooted at that node.

Note that these two pruning rules mean that there are at most $L+1$ nodes left unpruned at any level. Hence, this gives us an algorithm with running time is polynomial in n and L. We now wish to derive an iterative algorithm. Let $Sum[k, S]$ be true if there is a subset of the first k numbers that sums to S. We compute $Sum[k, S]$ as follows:

```
Sum[0,0]=True
For S=1 to L do Sum[0, S]= False


For k=1 to n do
     For S = 0 to L do
          Sum[k, S] = Sum[k-1, S] or Sum[k-1, S - x(k)]
```

10 Knapsack Example

The knapsack problem can be defined as follows:

INPUT: A collection of objects $O_1, \ldots, O_n$ with positive integer weights $w_1, \ldots, w_n$, and positive integer values $v_1, \ldots, v_n$. A positive integer L.

OUTPUT: The highest valued subset of the objects with weight at most L.

We can generate the subsets inductively using a binary tree as in the subset sum problem. The two pruning rules are

1. Eliminate the subtree rooted at any subset with sum greater than L.

2. If there are two subsets S and T at the same depth, with the same total weight, then eliminate the one of least value.

Note that these two pruning rules mean that there are at most $L + 1$ nodes left unpruned at any level. Let $Value[k, S]$ be highest value one can obtain from a subset of the first k numbers with aggregate weight S. We compute $Value[k, S]$ as follows:

```
Value[0,0]=0
For S=1 to L do Sum[0, S]= minus infinity


For k=1 to n do
     For S = 0 to L do
          Value[k, S] = Max( Value[k-1, S],
                             Value[k-1, S - w(k)] + v(k) )
```

11 Longest Increasing Subsequence Problem: Take 2

We now apply this method to the longest increasing subsequence problem. We can generate feasible solutions (subsequences) as in the subset sum problem. One obvious the pruning rule is:

1. If you have two subsequences S and T at the same depth that have the same length, prune the one that ends in the larger number.

This pruning rule will give you at most n unpruned nodes at any level. If you turn this into an iterative code, you will get the same code as we got before. Another possible pruning rule would be:

2. If two sequences S and T at the same depth have the same last number, prune the shorter sequence.

This pruning rule will also give you at most n unpruned nodes at any depth, and an (n^2) time algorithm.

Note how much easier it was to develop an algorithm for the LIS problem this way, as opposed to trying recursion and generalizing the induction hypothesis.

12 The Single Source Shortest Path Problem

The input to this problem is a directed positive edge weighted graph with a designated vertex s. The problem is to find the shortest simple path from s to each other vertex. For more information see section 4.2 of the text.

Here feasible solutions are simple paths that start from the source vertex s. The nodes in level k of the tree represent all paths from s of k or less hops. Note that by simplicity we we need only consider the tree to depth n, the number of vertices.

The pruning rule is that we need only remember the shortest path to a particular vertex. We thus get the following code. Here $D[k, i]$ is the shortest path from s to i of k or less hops.

```
For k= 1 to n do
     For i= 1 to n do
D[k, i] = min (D[k-1, i], D[k, i])
     for each edge e = (i, j) do
          D[k, j]=min( D[k, j], D[k -1, i] + the length of e )
```

This is Bellman-Ford shortest path algorithm and has running time $O(VE)$.

13 The Shortest Path Problem with Negative Edge Weights

The input to this problem is a directed edge weighted graph with a designated vertex s. The edge weights may be positive or negative. The problem is to find the shortest simple path from s to each other vertex. For more information see section 4.2 of the text.

Here feasible solutions are simple paths that start from the source vertex s. The nodes in level k of the tree represent all paths from s of k or less hops. Note that by simplicity we we need only consider the tree to depth n, the number of vertices.

The pruning rule is that if two paths end at the same vertex and contain the same vertices then we may prune the shorter one. Make sure you understand why the pruning rule that we used for positive weights does not work here.

We thus get the following code. Here $D[k, S, i]$ is the shortest path from s to i of k or less hops that visits exactly the vertices in S.

```
For k= 1 to n do
     For i= 1 to n do
      For S= 1 to 2^n do
   D[k, S, i] = min (D[k-1, S, i], D[k, S, i])
          for each edge e = (i, j) do
       if j is not in S then
                  D[k, S+ j, j]=min( D[k, S+j, j], D[k, S, i] + the length of e )
```

The running time of this code is $O(VE2^V)$, where V is the number of vertices and E is the number of edges.

14 The Traveling Salesman Problem

See section 3.6 from the text.

The input to this problem is a directed edge weighted graph with a designated vertex s. The edge weights may be positive or negative. The problem is to find the shortest simple path from s that visits all of the vertices.

Here feasible solutions are simple paths that start from the source vertex s. The nodes in level k of the tree represent all paths from s of exactly k. Note that by simplicity we we need only consider the tree to depth n, the number of vertices.

The pruning rule is that if two paths end at the same vertex and contain the same vertices then we may prune the shorter one.

We thus get the following code. Here $D[k, S, i]$ is the shortest path from s to i of k or less hops that visits exactly the vertices in S.

```
For k= 1 to n do
     For i= 1 to n do
      For S= 1 to 2^n do
           for each edge e = (i, j) do
       if j is not in S then
                   D[k, S+ j, j]=min( D[k, S+j, j], D[k, S, i] + the length of e )
```

The running time of this code is $O(VE2^V)$, where V is the number of vertices and E is the number of edges.