Chapter 10

Dynamic Programming

Recursion is an elegant way to solve problems. Often, a problem that looks complex can be solved by a short, clever recursive algorithm. However, for some problems, the natural recursive algorithm can be horribly inefficient. In particular, if a recursive function calls itself more than once, there is a danger of exponential growth. For example, a single call to a recursive function might result in two recursive calls, which in turn result in four calls, which result in eight, and so on. The run time of the algorithm can grow exponentially as a function of the distance between the top level of the recursion and the base case.

An example of this is the Fibonacci numbers, which are often used as an example of recursion even though the recursive computation is ridiculously inefficient. The Fibonacci numbers can be defined by the recursive function

```
int Fibonacci( int n ) {
   if (n <= 1)
      return 1;
   else
      return Fibonacci(n-1) + Fibonacci(n-2);
}
```

The number of times that this function calls itself grows exponentially as a function of the input n. The computation of Fibonacci(45) using this function involves over 3.6 billion recursive calls to the function; during the course of this computation, Fibonacci(1) is called 1,134,903,170 times.[1] The problem here is that the same values are computed over and over. The evaluation of Fibonacci(45) requires the evaluation of Fibonacci(44), Fibonacci(43), ..., Fibonacci(0), but there is really no need to compute any of these values more than once (let alone millions of times).

Suppose that, instead of recomputing Fibonacci(x) every time we need its need its value, we *save* the value the first time we compute it. The saved values that we compute can be stored in

[1] In fact, for $1 \leq x \leq n$, the number of times that Fibonacci(x) is called during the computation of Fibonacci(n) is Fibonacci($n - x$). The value of Fibonacci(n) itself grows as an exponential function of n, and 45 is the largest input that gives an answer that can be expressed as an ordinary signed 32-bit integer.

a table. When we need the value of Fibonacci(x) for some x, we first check the table. If there is already a saved value in the table, we can just return it without recomputing. Otherwise, we compute the value, and we store the value in the table before returning it. Note that when we use a previously computed value of, say, Fibonacci(42) from the table, we are saving not just one or two calls to the function but all the exponentially many calls that would have been necessary to compute the value if we were not using the table. The Fibonacci function can be written to use a table as follows, allowing for inputs up to $n = 45$:

```
int TABLE[46];  // Already computed values; TABLE[i] is 0
                // if Fibonacci(i) has not yet been computed.
                // If > 0, it is the precomputed value.

int Fibonacci_by_table( int n ) {
   if ( TABLE[n] == 0 ) {  // Value must be computed.
      if ( n <= 1 )
         TABLE[n] = 1;
      else
         TABLE[n] = Fibonacci_by_table(n-1)
                         + Fibonacci_by_table(n-2);
   }
   return TABLE[n];
}

int Fibonacci( int n ) {
   for (int i = 0; i < n; i++)
      TABLE[i] = 0;
   return Fibonacci_by_table(n);
}
```

The run time of Fibonacci_by_table(n) grows only linearly in n, instead of exponentially. Here, we have applied a process called **memoization** to the original recursive function. The name comes from the fact that we make a "memo" of a result the first time we compute it. Memoization can sometimes be used, as in this example, to produce an efficient version of an inefficient recursive algorithm.

Often, it is also possible to produce a non-recursive algorithm that filles the table of precomputed values from the "bottom up," that is, starting from the simplest cases and building up from them to the more complex cases. This idea leads to a non-recursive version of Fibonacci, which is somewhat more efficient than the memoized version:

```
int Fibonacci_nonrecursive( int n ) {
   int TABLE[n+1];
   for (int i = 0; i <= n; i++) {
      if ( i <= 1 )
```

```
            TABLE[i] = 1;
         else
            TABLE[i] = TABLE[i-1] + TABLE[i-2];
      }
      return TABLE[n];
   }
```

Of course, there are other ways to write non-recursive Fibonacci functions that don't use an array, but the point here is to show how building a table of precomputed values that can be used in later computations can lead to an efficient algorithm. This general approach is known as **dynamic programming**. (I have no explanation for the name.)

Dynamic programming does not apply to all problems, but when it does apply, it can produce an efficient algorithm for a problem that might at first seem to be intractable. Dynamic programming is likely to apply if you have a recursive algorithm for which the total number of sub-cases that exist is much smaller than the number of times that the algorithm calls itself. When that happens, the same sub-cases are being computed over and over, and it might be better to compute a table of results for all the possible sub-cases. In the rest of this chapter, we will look at a few applications of dynamic programming that are more interesting than the Fibonacci numbers.[2].

10.1 Shuffled Strings

We start with a relatively simple example. Suppose that x, y, and z are strings. We say that z is a "shuffle" of x and y if z can be obtained by mixing the characters from x and y in a way that preserves the left-to-right ordering of the characters from x and the characters from y. For example, "OBesFUScCheATIwON" is a shuffle of "eschew" and "OBFUSCATION".

Given x, y, and z, we can ask whether z is a shuffle of x and y. For this to be true, the last character of z must be equal either to the last character of x or to the last character of z, and the remaining characters of z must be a shuffle of the remaining characters in x and y. There are also base cases where x or y is empty. If x is the empty string, then z must be equal to y; if y is empty, then z must be equal to x.

This leads to an obvious recursive algorithm, shown here with some pseudocode:

```
bool isSuffle(string x, string y, string z) {
   int n = x.length();
   int m = y.length();
   int r = z.length();
   if (n == 0)
      return y == z;
   if (m == 0)
      return x == z;
```

[2]This material is largely from *Computer Algorithms* by Sara Baase, and *Introduction to Algorithms* by Cormen, Leiserson, and Rivest

```
   string x1 = the first n-1 characters in x
   string y1 = the first m-1 characters in y
   string z1 = the first r-1 characters in z
   return ( (z[r-1] == x[n-1] && isShuffle(x1,y,z1))
               || (z[r-1] == y[m-1] && isShuffle(x,y1,z1)) );
}
```

This algorithm has to potential to call itself twice, and when it calls itself, the size of the problem has only been reduced by one character. This means that there is a potential for exponential growth (although it will not occur for all strings). However, there is only a limited number of different subproblems, and so we can apply dynamic programming. The dynamic programming algorithm uses a n-by-m array, S, of boolean values, where $S[i][j]$ is true if and only if the first $i+j$ characters of z are a shuffle of the first i characters of x together with the first j characters of y. S will hold all the values that would be computed by the recursive *isShuffle* algorithm for all possible sub-cases, but each sub-case will only be computed once. We are really interested in $S[n][m]$, but we have to fill the array starting with smaller values of the indices. The recursive algorithm can be translated fairly easily into this dynamical programming version:

```
bool isShuffle_nonrecursive(string x, string y, string z) {
   int n = x.length();
   int m = y.length();
   int r = z.length();
   if (r != n + m)
      return false;  // obvious case
   bool S[n][m];
   S[0][0] = true;
   for (int i = 1; i < n; i++)
      S[i][0] = S[i-1][0] && (z[i-1] == x[i-1]);
   for (int j = 1; j < m; k++)
      S[0][j] = S[0][j-1] && (z[j-1] == y[j-1]);
   for (int i = 1; i < n; i++)
      for (int j = 1; j < m; j++) {
         S[i][j] = ( (z[i+j-1] == x[i-1]) && S[i-1][j] )
                     || ( (z[i+j-1] == y[j-1]) && S[i][j-1]);
      }
   return S[n-1][m-1];
}
```

This algorithm always runs in time $\Theta(nm)$. It is actually slower than the recursive algorithm in some cases, but it has much better worst-case performance. Next, we will look at another text-processing example that has a little more intrinsic interest.

10.2 Edit Distance

Suppose that you are given two strings $u = x_0x_1 \ldots x_{n-1}$ and $v = y_0y_1 \ldots y_{m-1}$. What is the smallest number of changes that can be made to transform u to v? Of course, the answer depends on what kinds of changes are allowed, but assuming that we know what types of operations are available, we can ask the question. If deleting and inserting symbols are possible operations, then u can be transformed to v in $n+m$ steps simply by deleting all the symbols of u and then inserting all the symbols of v, but it's likely that a shorter sequence of operations will do. More generally, we could specify that a certain "cost" is applied to each type of operation, and we could look for the minimum cost transformation of u to v.

This problem comes up, for example, in comparing DNA sequences for analogous genes from related species. The number of differences between the two sequences is related to the amount of time that the two species have been evolving separately, that is the time since their last common ancestor. Biologists use this idea to build evolutionary trees that show how groups of species are related.

We will consider a version of this problem where the editing operations are inserting a character, deleting a character, and changing a character. There is an obvious exponential time algorithm to solve this optimization problem: Look at all possible sequences of $n + m$ or fewer operations, test whether each sequence transforms u to v, and select the shortest sequence that performs this transformation. In practice, of course, we need a much more efficient algorithm. We will use a dynamic programming approach.

Given the strings $u = x_0x_1 \ldots x_{n-1}$ and $v = y_0y_1 \ldots y_{m-1}$, we define a two-dimensional n-by-m array D where $D[i][j]$ is the minimum number of differences between $x_0x_1 \ldots x_{i-1}$ and $y_0y_1 \ldots y_{j-1}$. We are really interested in $D[n-1][m-1]$, the minimum number of differences between the complete strings. But to compute this efficiently, we will fill in the entire array starting from $D[0][0]$.

First, note that $D[i][0] = i$ for any i since a string of length i can be converted into a string of length 0 by deleting each of the i characters, and clearly i operations are required. Similarly, $D[0][j] = j$ for any j. Let's consider how $D[i+1][j+1]$ can be computed from earlier entries in D. We want to know the minimum number of differences between $x_0x_1 \ldots x_i$ and $y_0y_1 \ldots y_j$. We can transform the first string into the second in several ways: (1) If $x_i = y_j$, we can simply transform $x_0x_1 \ldots x_{i-1}$ into $y_0y_1 \ldots y_{j-1}$ at a cost of $D[i][j]$, with no further cost; (2) If $x_i \neq y_j$, we can transform $x_0x_1 \ldots x_{i-1}$ into $y_0y_1 \ldots y_{j-1}$ and then change x_i into y_j at a total cost of $D[i][j] - 1$, with no further cost; (3) We could transform $x_0x_1 \ldots x_{i-1}$ into $y_0y_1 \ldots y_j$ and then delete x_i, at a cost of $D[i-1][j] + 1$; or (4) We could transform $x_0x_1 \ldots x_i$ and $y_0y_1 \ldots y_{j-1}$ and then insert y_j at a cost of $D[i][j-1] + 1$. To compute $D[i][j]$, we should look at all four possibilities and take the one of minimal cost. This leads to the following pseudocode algorithm for computing the values in the array D:

```
for (int j = 0; j < m; j++)
   D[0][j] = j;
for (int i = 1; i < n; i++) {
   D[i][0] = i;
   for (int j = 1; j < m; j++) {
```

```
            int cost;
            if ( x[i] == y[j] )
               cost = D[i-1][j-1];
            else
               cost = D[i-1][j-1] + 1;
            if ( D[i-1][j] + 1 < cost)
               cost = D[i-1][j] + 1;
            if ( D[i][j-1] + 1 < cost)
               cost = D[i][j-1] + 1;
            D[i][j] = cost;
         }
      }
```

We note that the computation of $D[i][j]$ depends only on values in the array that have been computed previously. The last value computed is $D[n-1][m-1]$, which is the output of the algorithm. The running time of the algorithm is a reasonably efficient $\Theta(nm)$.

10.3 Optimal Binary Search Tree

The time that it takes to find an item in a binary search tree is proportional to the height of the node that contains that item. If you imagine searching the same tree over and over for various items, the total search time will be minimized if the items that are more likely to be searched for are nearer to the root of the tree.

Suppose that we are given a collection of items x_0, x_1, ..., x_{n-1} and that we want to arrange them in a binary search tree for easy retrieval. The set of items is fixed: Once the tree is built, it will not be changed, and the same tree will be used over and over. Suppose we know that, for each i, the probability that a search is for item x_i is p_i. Consider a given binary search tree containing all the items. For each i, let the number of nodes on the path from the root to x_i be c_i. That is, c_i is the number of steps that it takes to search for x_i. We assume that we search for x_i with probability p_i, so that the expected number of steps in a search is $\sum_{i=0}^{n-1} p_i c_i$. We want to build the binary search tree that will minimize this quantity. Note that the value of c_i depends on the way that the x_i are arranged in the specific tree we are looking at.

Let's assume that the items are ordered such that $x_0 < x_1 < \cdots < x_{n-1}$. If we use x_k as the root of the tree, we know that $x_0, \ldots, x_{k-1}$ will have to go in the left subtree and $x_{k+1}, ..., x_{n-1}$ will have to go in the right subtree, in order to satisfy the binary search tree property.

One way to construct the optimal search tree would be to consider each x_k in turn for the root of the tree. In each case, we could then recursively find the optimal way of arranging the left and right subtrees of that root, and from that compute the expected search time in the best tree with root x_k. We then have n candidate trees, each with a different root. To get the optimal tree, we just choose the candidate that has the smallest expected search time. Unfortunately, the run time of this algorithm grows exponentially with n. But if we look at the subtrees considered at every level of the recursion, we see that every subtree consists a contiguous subsequence of the items,

of the form $x_i, x_{i+1}, \ldots, x_j$. There are only $\frac{n(n+1)}{2}$ such subsequences altogether, so the algorithm only computes $\frac{n(n+1)}{2}$ **different** optimal subtrees. We can get an efficient algorithm by computing each of these subtrees only once.

Using the dynamic programming approach, we want to store the information that we compute about the subtrees in a table. Define a two-dimensional table T where $T[i][j]$, for $i \leq j$, represents the expected number of steps needed for a search in the optimal binary tree that contains just the items $x_i, x_{i+1}, \ldots, x_j$. To make this clear, $T[i][j]$ counts the number of steps executed in this tree by an average search. When you search for an item not in the tree, the number of steps spent in the tree is zero; when you search for an item x_r in the tree, the number of steps is c_r, where c_r is the number of nodes on the path to x_r *in the optimal tree containing* $x_i, x_{i+1}, \ldots, x_j$. We know that x_r is selected with probability p_r, so the value of $T[i][j]$ is just $\sum_{r=i}^{j} p_r c_r$. Note that we are really interested only in $T[0][n-1]$, the expected search time in the full optimal tree.

We also define $R[i][j]$ to record the root of the optimal tree containing $x_i, x_{i+1}, \ldots, x_j$. That is, if the root contains item x_s, then $R[i][j] = s$. If we know $R[i][j]$ for all i and j, then we can use the data in R to recover the full optimal tree.

Note that when $j = i$, we are looking for the optimal subtree containing the single item x_i. Since there is only one possible tree in this case, containing x_i in its single node, we can say that $T[i][i] = p_i$ and that $R[i][i] = i$. As another example, suppose that $j = i + 1$. In this case, we are looking at trees that contain the two items x_i and x_{i+1}. There are two trees to consider, one with x_i as the root and one with x_{i+1} as the root. The expected search time in the first of these trees is $p_i + 2p_{i+1}$ and the expected time in second is $2p_i + p_{i+1}$. $T[i][i+1]$ is the minimum of these two values, and the value of $R[i][i+1]$ is either i or $i+1$, depending on whether the first tree or the second tree gives the minimum value.

Now, suppose that we want to compute $T[i][j]$ in general. We will show later that for $j > i$, $T[i][j]$ can be computed as

$$T[i][j] = \min_{i \leq k \leq j} \left(T[i][k-1] + T[k+1][j] + \sum_{r=i}^{j} p_r \right)$$

This formula assumes that $T[i][i-1]$ has been initialized to 0 for each i. We can also initialize $T[i][i] = p_i$ for each i. We can then start computing values for $T[i][j]$ for $j > i$, using the above formula. We just have to make sure that we compute them in the right order, so that so that all values that are used when we apply the formula for $T[i][j]$ have been computed previously. Note that for all the table entries $T[r][s]$ in the formula for $T[i][j]$, the difference between s and r is less than the difference between j and i, so we just have to compute $T[i][j]$ in order of increasing difference between i and j. Also, note that $R[i][j]$ is the value of k that gives the minimum value in the formula. Based on all this, we get the following dynamic programming algorithm. In this algorithm, we assume that the probabilities p_i have been stored in an array $P[i]$:

```
double T[N][N]; // Table of search times for optimal subtrees.
int R[N][N];    // Table of roots of optimal subtrees.
```

```
for (int i = 1; i < N; i++)    // Initialize entries below the diagonal.
   T[i][i-1] = 0;

for (int i = 0; i < N; i++) {  // Initialize entries on the diagonal.
   T[i][i] = P[i];
   R[i][i] = i;
}

for (int diff = 2; diff < N; diff++) { // Difference between i and j in T[i][j].
   for (int i = 0; i+diff < N; i++) {  // Initialize entries where j = i + diff.
      int j = i + diff;
      double sum = 0;
      for (int k = i; k <= j; k++)
         sum += P[k];
      double min_val = -1;
      int min_k;
      for (int k = i; k <= j; k++) {
         double val = T[i][k-1] + T[k+1][j] + sum;
         if (val > min_val) {
            min_val = min;
            min_k = k
         }
      }
      T[i][j] = min_val;
      R[i][j] = min_k;
   }
}
```

This algorithm runs in time $\Theta(n^3)$, so it is reasonably efficient. We are really interested in the optimal tree, rather than in the optimal times, so we should look at an algorithm that builds a binary tree based on the data in R. To be definite, let's assume that the items x_i are of type *ItemType* and that they are stored in an array $X[i]$. Here is the algorithm. To construct the complete optimal binary search tree, you just have to call *getOptimalTree*(1,$n-1$):

```
struct TreeNode {
   ItemType item;           // One of the items x
   TreeNode *left, *right;  // Pointers to subtrees.
};

TreeNode *getOptimalTree(int i, int j) {
   if (j < i)
      return NULL;
   else {
```

```
            TreeNode *root = new TreeNode();
            int k = R[i][j];  // Root of this tree.
            root->item = X[k];
            root->left = getOptimalTree(i,k-1);
            root->right = getOptimalTree(k+1,j);
        }
    }
```

The correctness of our algorithm depends on the correctness of the formula that we have given for $T[i][j]$. The rest of this section gives the rather mathematical justification for the formula. The goal is to compute the expected number of steps in a search in the optimal tree containing $x_i, x_{i+1}, \ldots, x_j$. To do this, we consider each of the possible choices of root for this tree. If we choose x_k for the root, where $i \leq k \leq j$, then the left subtree will contain $x_i, \ldots, x_{k-1}$ while the right subtree will contain $x_{k+1}, \ldots, x_j$. (One of these trees can in fact be empty. "Search time" in an empty tree is zero. This is accounted for by the fact that we set $T[r][r-1] = 0$ for each r.)

Assume that we have already computed $T[i][k-1]$ and $T[k+1][j]$. These values represent the expected number of steps spent during a search in the left and right subtrees. The question is how to compute the expected number of steps spent in the entire tree. This value is the sum of three terms: the expected number of steps when the search is for the root item, p_k, the expected number of steps when the search is for an item in the left subtree, and the expected number of steps when the search is for an item in the right subtree.

If the search is for the root item, x_k, then there is only one step in the tree, and this occurs with probability p_k, so searches for the root item contribute the term $p_k * 1$ to the sum. Suppose that the search is for an item in the left subtree. The expected number of steps *within that subtree* is $T[i][k-1] = \sum_{r=i}^{k-1} p_r c_r$, where c_r is the distance from x_r to the root of the subtree. If we look at the time spent within the tree as a whole, we use the same formula, except that c_r is replaced by $c_r + 1$ to reflect the fact that the root of the tree is one node further away from x_r than the root of the subtree. That is, the expected number of steps within the whole tree is $\sum_{r=i}^{k-1} p_r(c_r + 1)$. This is the same as $\left(\sum_{r=i}^{k-1} p_r c_r\right) + \left(\sum_{r=i}^{k-1} p_r\right)$, which is $T[i][k-1] + \left(\sum_{r=i}^{k-1} p_r\right)$. This is the amount contributed by items in the left subtree to the expected number of steps in the tree. Similarly, items in the right subtree contribute $T[k+1][j] + \left(\sum_{r=k+1}^{j} p_r\right)$ to the sum. Taking the total of the three contributions, we see that the expected number of steps for a search in a tree with root x_k is

$$\left(T[i][k-1] + \sum_{r=i}^{k-1} p_r\right) + p_k + \left(T[k+1][j] + \sum_{r=k+1}^{j} p_r\right)$$

This can be rewritten as

$$T[i][k-1] + T[k+1][j] + \sum_{r=i}^{j} p_r$$

To compute $T[i][j]$, the expected number of steps in the optimal tree, we just have to compute this

value for each k and take the minimum. That is,

$$T[i][j] = \min_{i \le k \le j} \left(T[i][k-1] + T[k+1][j] + \sum_{r=i}^{j} p_r \right)$$

as I asserted earlier.