11.1 The String reconstruction problem

The greedy approach doesn't always work, as we have seen. It lacks flexibility; if at some point, it makes a wrong choice, it becomes stuck.

For example, consider the problem of *string reconstruction*. Suppose that all the blank spaces and punctuation marks inadvertently have been removed from a text file. You would like to reconstruct the file, using a dictionary. (We will assume that all words in the file are standard English.)

For example, the string might begin "thesearethereasons". A greedy algorithm would spot that the first two words were "the" and "sea", but then it would run into trouble. We could backtrack; we have found that sea is a mistake, so looking more closely, we might find the first three words "the","sear", and "ether". Again there is trouble. In general, we might end up spending exponential time traveling down false trails. (In practice, since English text strings are so well behaved, we might be able to make this work– but probably not in other contexts, such as reconstructing DNA sequences!)

This problem has a nice structure, however, that we can take advantage of. The problem can be broken down into entirely similar subproblems. For example, we can ask whether the strings "theseare" and "thereasons" both can be reconstructed with a dictionary. If they can, then we can glue the reconstructions together. Notice, however, that this is not a good problem for divide and conquer. The reason is that we do not know where the right dividing point is. In the worst case, we could have to try every possible break! The recurrence would be

$$T(n) = \sum_{i=1}^{n-1} T(i) + T(n-i).$$

You can check that the solution to this recurrence grows exponentially.

Although divide and conquer directly fails, we still want to make use of the subproblems. The attack we now develop is called *dynamic programming*. The way to understand dynamic programming is to see that divide and conquer fails because we might recalculate the same thing over and over again. (Much like we saw very early on with the Fibonacci numbers!) If we try divide and conquer, we will repeatedly solve the same subproblems (the case of small substrings) over and over again. The key will be to avoid the recalculations. To avoid recalculations, we use a lookup table.

In order for this approach to be effective, we have to think of subproblems as being ordered by size. We solve the subproblems bottom-up, from the smallest to the largest, until we reach the original problem.

For this dictionary problem, think of the string as being an array $s[1 \ldots n]$. Then there is a natural subproblem for each substring $s[i \ldots j]$. Consider a two dimensional array $D(i, j)$ that will denote whether $s[i \ldots j]$ is the concatenation of words from the dictionary. The size of a subproblem is naturally $d = j - i$.

So now we write a simple loops which solves the subproblems in order of increasing size:

```
for d := 1 to n − 1 do
  for i := 1 to n − d do
      j := i + d;
      if indict(s[i . . . j]) then D(i, j) := true else
        for k := i + 1 to j − 1 do
            if D(i,k) and D(k, j) then D(i, j) := true;
```

This algorithm runs in time $O(n^3)$; the three loops each run over at most n values. Pictorially, we can think of the algorithm as filling in the upper diagonal triangle of a two-dimensional array, starting along the main diagonal and moving up, diagonal by diagonal.

We need to add a bit to actually find the words. Let $F(i, j)$ be the position of end of the first word in $s[i \ldots j]$ when this string is a proper concatenation of dictionary words. Initially all $F(i, j)$ should be set to nil. The value for $F(i, j)$ can be set whenever $D(i, j)$ is set to true. Given the $F(i, j)$, we can reconstruct the words simply by finding the words that make up the string in order. Note also that we can use this to improve the running time; as soon as we find a match for the entire string, we can exit the loop and return success! Further optimizations are possible.

Let us highlight the aspects of the dynamic programming approach we used. First, we used a recursive description based on subproblems: $D(i, j)$ is true if $D(i,k)$ and $D(k, j)$ for some k. Second, we built up a table containing the answers of the problems, in some natural bottom-up order. Third, we used this table to find a way to determine the actual solution. Dynamic programming generally involves these three steps.

11.2 Edit distance

A problem that arises in biology is to measure the distance between two strings (of DNA). We will examine the problem in English; the ideas are the same. There are many possible meanings for the distance between two strings; here we focus on one natural measure, the *edit distance*. The edit distance measures the number of editing operations it would be necessary to perform to transform the first string into the second. The possible operations are as follows:

- Insert: Insert a character into the first string.
- Delete: Delete a character from the first string.
- Replace: Replace a character from the first string with another character.

Another possibility is to not edit a character, when there is a Match. For example, a transformation from *activate* to *caveat* can be represented by

D	M	R	D	M	I	M	M	D
a	c	t	i	v		a	t	e
	c	a		v	e	a	t	

The top line represents the operation performed. So the *a* in activate id deleted, and the *t* is replaced. The *e* in caveat is explicitly inserted.

The *edit distance* is the minimal number of edit operations – that is, the number of Inserts, Deletes, or Replaces – necessary to transform one string to the other. Note that Matches do not count. Also, it is possible to have a *weighted edit distance*, if the different edit operations have different costs. We currently assume all operations have weight 1.

We will show how compute the edit distance using dynamic programming. Our first step is to define appropriate subproblems. Let us represent our strings by $A[1 \ldots n]$ and $B[1 \ldots m]$. Suppose we want to consider what we do with the last character of A. To determine that, we need to know how we might have transformed the first $n-1$ characters of A. These $n-1$ characters might have transformed into any number of symbols of B, up to m. Similarly, to compute how we might have transformed the first $n-1$ characters of A into some part of B, it makes sense to consider how we transformed the first $n-2$ characters, and so on.

This suggests the following submproblems: we will let $D(i, j)$ represent the edit distance between $A[1 \ldots i]$ and $B[1 \ldots j]$. We now need a recursive description of the subproblems in order to use dynamic programming. Here the recurrence is:

$$D(i, j) = \min[D(i-1, j)+1, D(i, j-1)+1, D(i-1, j-1)+I(i \neq j)].$$

In the above, $I(i \neq j)$ represents the value 1 if $i \neq j$ and 0 if $i = j$. We obtain the above expression by considering the possible edit operations available. Suppose our last operation is a Delete, so that we deleted the ith character of A to transform $A[1 \ldots i]$ to $B[1 \ldots j]$. Then we must have transformed $A[1 \ldots i-1]$ to $B[1 \ldots j]$, and hence the edit distance

would be $D(i-1,j)+1$, or the cost of the transformation from $A[1\ldots i-1]$ to $B[1\ldots j]$ plus one for the cost of the final Delete. Similarly, if the last operation is an Insert, the cost would be $D(i,j-1)+1$.

The other possibility is that the last operation is a Replace of the ith character of A with the jth character of B, or a Match between these two characters. If there is a Match, then the two characters must be the same, and the cost is $D(i-1,j-1)$. If there is a Replace, then the two characters should be different, and the cost is $D(i-1,j-1)+1$. We combine these two cases in our formula, using $D(i-1,j-1)+I(i\neq j)$.

Our recurrence takes the minimum of all these possibilities, expressing the fact that we want the best possible choice for the final operation!

It is worth noticing that our recursive description does not work when i or j is 0. However, these cases are trivial. We have

$$D(i,0)=i,$$

since the only way to transform the first i characters of A into nothing is to delete them all. Similarly,

$$D(0,j)=j.$$

Again, it is helpful to think of the computation of the $D(i,j)$ as filling up a two-dimensional array. Here, we begin with the first column and first row filled. We can then fill up the rest of the array in various ways: row by row, column by column, or diagonal by diagonal!

Besides computing the distance, we may want to compute the actual transformation. To do this, when we fill the array, we may also picture filling the array with pointers. For example, if the minimal distance for $D(i,j)$ was obtained by a final Delete operation, then the cell (i,j) in the table should have a pointer to $(i-1,j)$. Note that a cell can have multiple pointers, if the minimum distance could have been achieved in multiple ways. Now any path back from (n,m) to $(0,0)$ corresponds to a sequence of operations that yields the minimum distance $D(n,m)$, so the transformation can be found by following pointers.

The total computation time and space required for this algorithm is $O(nm)$.