Memoization/Dynamic Programming

Today's lecture discusses *memoization*, which is a method for speeding up algorithms based on recursion, by using additional memory to remember already-computed answers to subproblems. It is quite similar to *dynamic programming*, which is the iterative version. (**Mantra:** memoization is to recursion as dynamic programming is to for loops.)

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*, and its close relative *memoization*. Memoization is identical to recursion, except where we also on the side store a lookup table. If the input to our recursive function has never been seen before, we compute as normal *then* store the answer in the lookup table at the end. If we have seen the input before, then it's in the lookup table and we simply fetch it and retrieve it. The analogy to keep in mind is then "dynamic programming is to memoization as iteration is to recursion". Whereas the lookup table is built recursively in memoization, in dynamic programming it is built bottom-up.

In order for this approach to be effective, we have to think of subproblems as being ordered by size. In memoization we initially call our recursive function on the original input, then recursively solve "smaller" subproblems until we reach base cases.

For this dictionary problem, think of the string as being an array $s[1 \dots n]$. Then there is a natural subproblem for each substring $s[i \dots j]$. Consider a function $f(i,j)$ that is *true* if $s[i \dots j]$ is the concatenation of words from the dictionary and *false* otherwise. The size of a subproblem is naturally $d = j - i + 1$. And we have a recursive description of $f(i,j)$ in terms of smaller subproblems:

$$f(i,j) = (\text{indict}(s[i \dots j]) \text{ OR}_{k=i}^{j-1} (f(i,k) \text{ AND } f(k+1,j))).$$

The above recursive description now naturally leads to a recursive implementation.

Algorithm $f(i,j)$:

```
    // base case
1.  if indict(s[i...j]): return true
2.  for k = 1,...,j-1:
        if f(i,k) and f(k+1,j):
            return true
3.  return false
```

Algorithm CANBEPARSED(`s[1..n]`):

1. initialize a global variable `s` equal to the input string `s`
2. **return** $f(1,n)$

If one traces out the recursion tree during computation, starting from $f(1,n)$, it is apparent that f is called repeatedly with the same input parameters i, j. The memoized version below now fixes this efficiency issue:

Algorithm $f(i,j)$:

1. **if** `seen[i][j]`:
   return `D[i][j]`
2. `seen[i][j]` $\leftarrow$ *true*

   // base case
3. **if** indict($s[i \ldots j]$):
   `D[i][j]` $\leftarrow$ *true*
   return `D[i][j]`
4. **for** $k = 1, \ldots, j-1$:
   if $f(i,k)$ *and* $f(k+1,j)$:
   `D[i][j]` $\leftarrow$ *true*
   return `D[i][j]`
5. `D[i][j]` $\leftarrow$ *false*
   return `D[i][j]`

Algorithm MEMOIZEDCANBEPARSED(`s[1..n]`):

1. initialize a global variable `s` equal to the input string `s`
2. initialize a global doubly indexed array `seen`[1..n][1..n] with all values *false*
3. initialize a global doubly indexed array `D`[1..n][1..n]
4. **return** $f(1,n)$

What is the running time of MEMOIZEDCANBEPARSED? If we assum that we have an implementation of indict which always takes time at most T on inputs of size at most n, then we see that there are at most n^2 possible (i,j) that could be the input to f. For each of these possible inputs, ignoring all work done in recursive subcalls, we in addition (1) call indict, and (2) execute a for loop over k which takes at most n steps. Thus the sum of all work done across all possible inputs (i,j) is at most $n^2 \cdot (T+n) = O(n^2T + n^3)$. Note this is an upper bound on the running time, since once we compute some answer $f(i,j)$ and put it in the lookup table `D`, any future recursive calls to $f(i,j)$ return immediately with the answer, doing no additional work.

An easier way to see that the runtime is $O(n^2T + n^3)$ is to write a dynamic programming implementation of the solution, which builds `D` bottom-up, as below:

Algorithm DPCANBEPARSED(`s[1..n]`):

1. **for** $d = 1 \ldots n$:
   for $i = 1 \ldots n-d+1$:
   $j \leftarrow i+d-1$:
2. **if** indict($s[i \ldots j]$):
   `D[i][j]` $\leftarrow$ *true*
3. **else**:
   for $k = i \ldots j-1$:
   if `D[i][k]` *and* `D[k+1][j]`:
   `D[i][j]` $\leftarrow$ *true*
4. **return** `D[1][n]`

Now it is clearer that this implementation runs in time $O(n^2T + n^3)$. 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+1][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.

Before we go on, we should ask ourselves, can we improve on the above solution? It turns out that we can. The key is to notice that we don't really care about `D[i][j]` for all i and j. We just care about `D[1][j]` – can the string from the start be broken up into a collection of words. So now consider a one dimensional array $g(j)$ that will denote whether $s[1 \ldots j]$ is the concatenation of words from the dictionary. The size of a subproblem is now j. And we have a recursive description of $g(j)$ in terms of smaller subproblems:

$$g(j) = (\text{indict}(s[1 \ldots j]) \text{ OR}_{k=1}^{j-1} (g(k) \text{ AND indict}(s[k+1 \ldots j]))).$$

As an exercise, rewrite the pseudocode for filling the one-dimensional array using this recurrence. This algorithm runs in time $O(nT + n^2)$. You may also consider the following optimization: if the largest word in the dictionary is of length $L \ll n$, then you can modify the algorithm to run in time $O(nT + nL)$.

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. Let us assume we have positive costs c_i, c_d, and c_r for insert, delete, and replace.

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 subproblems: 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:

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

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

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 $D(i-1,j-1)+c_r$. We combine these two cases in our formula, using $D(i-1,j-1)+c_r I(i \neq j)$.

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

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

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

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

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 $O(nm)$.

In-Class Exercise

Find a partner, and try the following dynamic programming problem.

In various scientific or engineering applications one has multiply a chain of matrices. Given matrices $A_1, A_2, \ldots, A_n$, a natural goal is to multiply them using as few operations as possible. For convenience we will just count multiplication operations, as they are (under some architectures) the most expensive. Recall that matrix multiplication is not commutative, but is associative. Assume that we have a basic subroutine that we can call that multiplies two matrices; if the matrices are m by n and n by q, they will require mnq multiplication operations with our standard subroutine.

We can choose the order to multiply the matrices; this corresponds to parenthesizing the expression. For example, suppose we have an m by n matrix A, and n by p matrix B, and and p by s matirx C. The product ABC will be of size m by s, and can be calculated by either calculating $(AB)C$ or $A(BC)$. The first requires $mnp + mps$ multiplcation operations; the second requires $nps + mns$ multiplication operations. Try some numbers for m, n, p and s, and you'll see one approach can be much faster than another.

Come up with an algorithm that finds a parenthesization that minimizes the number of multiplication operations for computing $\prod_{i=1}^{n} A_i$ given the dimensions of each A_i.

Possible hint: let $M(j,k)$ be the the minimum number of multiplication operations for computing $\prod_{i=j}^{k} A_i$.

Traveling salesman problem

Suppose that you are given n cities and the distances d_{ij} between them. The traveling salesman problem (TSP) is to find the shortest tour that takes you from your home city to all the other cities and back again. As there are $(n-1)!$ possible paths, this can clearly be done in $O(n!)$ time by trying all possible paths. Of course this is not very efficient.

Since the TSP is NP-complete, we cannot really hope to find a polynomial time algorithm. But dynamic programming gives us a much better algorithm than trying all the paths.

The key is to define the appropriate subproblem. Suppose that we label our home city by the symbol 1, and other cities are labeled $2, \ldots, n$. In this case, we use the following: for a subset S of vertices including 1 and at least one other city, let $C(S, j)$ be the shortest path that start at 1, visits all other nodes in S, and ends at j. Note that our subproblems here look slightly different: instead of finding tours, we are simply finding *paths*. The reason for this is tours are hard to break dwon into recursive subproblems, but paths have a natural recursive formulation. The important point is that the shortest path from i to j through all the vertices in S consists of some shortest path from i to a vertex x, where $x \in S - \{j\}$, and the additional edge from x to j.

for all j do $C(\{1, j\}, j) := d_{1j}$
  for s = 3 to n do % s is the size of the subset
  for all subsets S of $\{1, \ldots, n\}$ of size s containing 1 do
    for all $j \in S$, $j \neq 1$ do
      $C(S, j) := \min_{i \neq j, i \in S}[C(S - \{j\}, i) + d_{ij}]$
  opt := $\min_{j \neq i} C(\{1, \ldots, n\}, j) + d_{j1}$

The idea is to build up paths one node at a time, not worrying (at least temporarily) where they will end up. Once we have paths that go through all the vertices, it is easy to check the tours, since they consists of a shortest path through all the vertices plus an additional edge. The algorithm takes time $O(n^2 2^n)$, as there are $O(n2^n)$ entries in the table (one for each pair of set and city), and each takes $O(n)$ time to fill. Of course we can add in structures so that we can actually find the tour as well. **Exercise: Consider how memory-efficient you can make this algorithm.**

Switching

We'll return to some further work on dynamic programming, and other algorithms, after we switch gears for a bit and think about machine models that are suitable for implementing some of the algorithms we've been talking about.