Notes 15 for CS 170

1 Introduction to Dynamic Programming

Consider the following algorithm for computing the n-th Fibonacci number F_n: Recursively apply the definition $F_n = F_{n-1} + F_{n-2}$ so that a function call to compute F_n results in two functions calls to compute F_{n-1} and F_{n-2}, and so on. The problem with this approach is that it is very expensive; it ends up calling a function to compute F_j for each $j < n$ possibly very many times, even after F_j had already been computed. We can improve this algorithm by building a table of values of Fibonacci numbers, computing F_n by looking up F_{n-1} and F_{n-2} in the table and simply adding them. This lowers the cost of computing F_n from exponential in n to just linear in n.

This works because we can sort the problems of computing F_n simply by increasing n, and compute and store the Fibonacci numbers with small n before computing those with large n.

Dynamic programming uses exactly the same idea in substantially more generality:

1. Express the solution to a problem in terms of solutions to smaller problems.

2. Solve all the smallest problems first and put their solutions in a table, then solve the next larger problems, putting their solutions into the table, solve and store the next larger problems, and so on, up to the problem one originally wanted to solve. Each problem should be easily solvable by looking up and combining solutions of smaller problems in the table.

For Fibonacci numbers, how to compute F_n in terms of smaller problems F_{n-1} and F_{n-2} is obvious. For more interesting problems, figuring out how to break big problems into smaller ones is the tricky part. Once this is done, the the rest of algorithm is usually straightforward to produce. We will illustrate by a sequence of examples, starting with "one-dimensional" problems that are most analogous to Fibonacci.

2 String Reconstruction

Suppose that all blanks and punctuation marks have been inadvertently removed from a text file, and its beginning was polluted with a few extraneous characters, so the file looks something like "lionceuponatimeinafarfarawayland..." You want to reconstruct the file using a dictionary.

This is a typical problem solved by dynamic programming. We must define what is an appropriate notion of *subproblem*. Subproblems must be ordered by *size*, and each subproblem must be easily solvable, once we have the solutions to all smaller subproblems. Once we have the right notion of a subproblem, we write the appropriate recursive equation expressing how a subproblem is solved based on solutions to smaller subproblems, and the program is then trivial to write. The complexity of the dynamic programming algorithm is precisely the total number of subproblems times the number of smaller subproblems we must examine in order to solve a subproblem.

In this and the next few examples, we do dynamic programming on a one-dimensional object—in this case a string, next a sequence of matrices, then a set of strings alphabetically ordered, etc. The basic observation is this: *A one-dimensional object of length n has about n^2 sub-objects* (substrings, etc.), where a sub-object is defined to span the range from i to j, where $i, j \leq n$. In the present case a subproblem is to tell whether the substring of the file from character i to j is the concatenation of words from the dictionary. Concretely, let the file be $f[1 \ldots n]$, and consider a 2-D array of Boolean variables $T(i,j)$, where $T(i,j)$ is true if and only if the string $f[i \ldots j]$ is the concatenation of words from the dictionary. The recursive equation is this:

$$T(i,j) = dict(x[i \ldots j]) \vee \bigvee_{i \leq k < j} [T(i,k) \wedge T(k+1,j)]$$

In principle, we could write this equation verbatim as a recursive function and execute it. The problem is that there would be *exponentially many* recursive calls for each short string, and 3^n calls overall.

Dynamic programming can be seen as a technique of implementing such recursive programs, that have heavy overlap between the recursion trees of the two recursive calls, so that the recursive function is called once for each distinct argument; indeed the recursion is usually "unwound" and disappears altogether. This is done by modifying the recursive program so that, in place of each recursive call a table is consulted. To make sure the needed answer is in the table, we note that the lengths of the strings on the right hand side of the equation above are $k-i+1$ and $j-k$, a both of which are shorter than the string on the left (of length $j-i+1$). This means we can fill the table in *increasing order of string length.*

for $d := 0$ to $n-1$ do ... $d+1$ is the size (string length) of the subproblem being solved
    for $i := 1$ to $n-d$ do ... the start of the subproblem being solved
    $j = i + d$
    if $dict(x[i \ldots j])$ then $T(i,j)$:=true else
        for $k := i$ to $j-1$ do
        if $T(i,k)$ =true and $T(k+1,j)$ =true then do $\{T(i,j)$:=true$\}$

The complexity of this program is $O(n^3)$: three nested loops, ranging each roughly over n values.

Unfortunately, this program just returns a meaningless Boolean, and does not tell us how to reconstruct the text. Here is how to reconstruct the text. Just expand the innermost loop (the last assignment statement) to

$\{T[i,j]$:=true, first$[i,j] := k$, exit for$\}$

where first is an array of pointers initialized to *nil.* Then if $T[i,j]$ is true, so that the substring from i to j is indeed a concatenation of dictionary words, then first$[i,j]$ points to a breaking point in the interval i, j. Notice that this improves the running time, by exiting the for loop after the first match; more optimizations are possible. This is typical of dynamic programming algorithms: Once the basic algorithm has been derived using dynamic programming, clever modifications that exploit the structure of the problem speed up its running time.

3 Edit Distance

3.1 Definition

When you run a spell checker on a text, and it finds a word not in the dictionary, it normally proposes a choice of possible corrections.

If it finds `stell` it might suggest `tell`, `swell`, `stull`, `still`, `steel`, `steal`, `stall`, `spell`, `smell`, `shell`, and `sell`.

As part of the heuristic used to propose alternatives, words that are "close" to the misspelled word are proposed. We will now see a formal definition of "distance" between strings, and a simple but efficient algorithm to compute such distance.

The distance between two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is the minimum number of "errors" (edit operations) needed to transform x into y, where possible operations are:

- insert a character.
  $insert(x, i, a) = x_1 x_2 \cdots x_i a x_{i+1} \cdots x_n$.
- delete a character.
  $delete(x, i) = x_i x_2 \cdots x_{i-1} x_{i+1} \cdots x_n$.
- modify a character.
  $modify(x, i, a) = x_1 x_2 \cdots x_{i-1} a x_{i+1} \cdots x_n$.

For example, if $x = aabab$ and $y = babb$, then one 3-steps way to go from x to y is

a	a	b	a	b		x
b	a	a	b	a	b	x' = insert(x,0,b)
b	a	b	a	b		x" = delete (x',2)
b	a	b	b			y = delete (x",4)

another sequence (still in three steps) is

a	a	b	a	b	x
a	b	a	b		x' = delete (x,1)
b	a	b			x" = delete(x',1)
b	a	b	b		y = insert (x",3,b)

Can you do better?

3.2 Computing Edit Distance

To transform $x_1 \cdots x_n$ into $y_1 \cdots y_m$ we have three choices:

- put y_m at the end: $x \to x_1 \cdots x_n y_m$ and then transform $x_1 \cdots x_n$ into $y_1 \cdots y_{m-1}$.
- delete x_n: $x \to x_1 \cdots x_{n-1}$ and then transform $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_m$.
- change x_n into y_m (if they are different): $x \to x_1 \cdots x_{n-1} y_m$ and then transform $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_{m-1}$.

This suggests a recursive scheme where the sub-problems are of the form "how many operations do we need to transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$.

Our dynamic programming solution will be to define a $(n+1) \times (m+1)$ matrix $M[\cdot, \cdot]$, that we will fill so that for every $0 \le i \le n$ and $0 \le j \le m$, $M[i, j]$ is the minimum number of operations to transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$.

The content of our matrix M can be formalized recursively as follows:

- $M[0, j] = j$ because the only way to transform the empty string into $y_1 \cdots y_j$ is to add the j characters $y_1, \ldots, y_j$.
- $M[i, 0] = i$ for similar reasons.
- For $i, j \ge 1$,

$$\begin{aligned} M[i,j] \quad &= \min\{ \quad M[i-1,j]+1, \\ &\qquad\qquad\; M[i,j-1]+1, \\ &\qquad\qquad\; M[i-1,j-1]+\mathit{change}(x_i,y_j)\} \end{aligned}$$

  where $\mathit{change}(x_i, y_j) = 1$ if $x_i \ne y_j$ and $\mathit{change}(x_i, y_j) = 0$ otherwise.

As an example, consider again $x = aabab$ and $y = babb$

	λ	b	a	b	b
λ	0	1	2	3	4
a	1	1	1	2	3
a	2	2	1	2	3
b	3	2	2	1	2
a	4	3	2	2	2
b	5	4	3	2	2

What is, then, the edit distance between x and y?

The table has $\Theta(nm)$ entries, each one computable in constant time. One can construct an auxiliary table $Op[\cdot, \cdot]$ such that $Op[\cdot, \cdot]$ specifies what is the first operation to do in order to optimally transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$. The full algorithm that fills the matrices can be specified in a few lines

```
algorithm EdDist(x,y)
     n = length(x)
     m = length(y)
     for i = 0 to n
       M[i,0] = i
     for j = 0 to m
       M[0,j] = j
     for i = 1 to n
       for j = 1 to m
         if x_i == y_j then change = 0 else change = 1
```

```
M[i,j] = M[i-1,j] + 1;  Op[i,j] = delete(x,i)
if M[i,j-1] + 1 < M[i,j] then
   M[i,j] = M[i,j-1] + 1;  Op[i,j] = insert(x,i,y_j)
if M[i-1,j-1] + change < M[i,j] then
   M[i,j] = M[i-1,j-1] + change
   if (change == 0) then Op[i,j] = none
   else Op[i,j] = change(x,i,y_j)
```

4 Longest Common Subsequence

A *subsequence* of a string is obtained by taking a string and possibly deleting elements.

If $x_1 \cdots x_n$ is a string and $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ is a strictly increasing sequence of indices, then $x_{i_1} x_{i_2} \cdots x_{i_k}$ is a subsequence of x. For example, **art** is a subsequence of **a**lgo**r**i**t**hm.

In the *longest common subsequence problem*, given strings x and y we want to find the longest string that is a subsequence of both.

For example, **art** is the longest common subsequence of **a**lgo**r**i**t**hm and p**ar**achu**t**e.

As usual, we need to find a recursive solution to our problem, and see how the problem on strings of a certain length can be reduced to the same problem on smaller strings.

The length of the l.c.s. of $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is either

- The length of the l.c.s. of $x_1 \cdots x_{n-1}$ and $y_1 \cdots y_m$ or;
- The length of the l.c.s. of $x_1 \cdots x_n$ and $y_1 \cdots y_{m-1}$ or;
- $1 +$ the length of the l.c.s. of $x_1 \cdots x_{n-1}$ and $y_1 \cdots y_{m-1}$, if $x_n = y_m$.

The above observation shows that the computation of the length of the l.c.s. of x and y reduces to problems of the form "what is the length of the l.c.s. between $x_1 \cdots x_i$ and $y_1 \cdots y_i$?"

Our dynamic programming solution uses an $(n+1) \times (m+1)$ matrix M such that for every $0 \leq i \leq n$ and $0 \leq j \leq m$, $M[i,j]$ contains the length of the l.c.s. between $x_1 \cdots x_i$ and $y_1 \cdots y_j$. The matrix has the following formal recursive definition

- $M[i,0] = 0$
- $M[0,j] = 0$
- $$\begin{aligned} M[i,j] \quad = \max\{ \quad & M[i-1,j] \\ & M[i,j-1] \\ & M[i-1,j-1] + eq(x_i,y_j)\} \end{aligned}$$

  where $eq(x_i,y_j) = 1$ if $x_i = y_j$, $eq(x_i,y_j) = 0$ otherwise.

The following is the content of the matrix for the words `algorithm` and `parachute`.

	λ	p	a	r	a	c	h	u	t	e
λ	0	0	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1	1	1
l	0	0	1	1	1	1	1	1	1	1
g	0	0	1	1	1	1	1	1	1	1
o	0	0	1	1	1	1	1	1	1	1
r	0	0	1	2	2	2	2	2	2	2
i	0	0	1	2	2	2	2	2	2	2
t	0	0	1	2	2	2	2	2	3	3
h	0	0	1	2	2	2	3	3	3	3
m	0	0	1	2	2	2	3	3	3	3

The matrix can be filled in $O(nm)$ time. How do you reconstruct the longest common substring given the matrix?