Dynamic Programming

Dynamic programming is a widely applicable way of solving problems. The basic idea is to use solutions to smaller cases of a problem to build up solutions to bigger cases. One way to do this is to start small and save the solutions to the smaller problems as we find them. Another way is a top-down recursive approach, where we also save solutions to problems. Let's see how this works for computing Fibonacci numbers.

The Fibonacci numbers are the sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Each number in the sequence is the sum of the two before it. For instance, $34 = 21 + 13$. Mathematically, people often use the notation F_n to denote Fibonaccis. If we start at $n = 0$, then $F_0 = 1$, $F_1 = 1$, $F_2 = 2$, $F_3 = 3$, $F_4 = 5$, $F_5 = 8$, etc. The general rule for generating new Fibonaccis is $F_n = F_{n-1} + F_{n-2}$. For instance, with $n = 6$, we get $F_6 = F_5 + F_4$, which gives $F_6 = 5 + 8 = 13$. Notice how each Fibonacci number is generated from previous Fibonacci numbers. This is a the key to dynamic programming, where we use smaller cases to build up to bigger cases.

We will look at two common approaches to generating Fibonaccis. One approach is recursive. It is basically a direct translation of the mathematical formula into computer code. See below.

```
def fib(n):
    if n==0 or n==1:
        return 1
    return fib(n-1) + fib(n-2)
```

A second way, shown below, is a "bottom-up" approach that builds up a list of Fibonaccis, starting with small values and working toward larger ones.

```
def fib(n):
    F = [1, 1]
    for i in range(2, n+1):
        F.append(F[i-1] + F[i-2])
    return F[-1]
```

The recursive solution is a bit simpler (if you are comfortable with recursion), but it tends to run very slowly for large values of n because the same values tend to be recomputed over and over again. For instance, to compute F_{10}, we need to do F_9 and F_8. To do F_9, we also need F_8, so F_8 gets computed twice. And it keeps getting worse: F_7 ends up having to be computed 3 times and F_6 ends up being computed 5 times. The total number of computations ends up being exponential, with the algorithm running in $O(2^n)$ time. The list-based approach runs in $O(n)$ time, which is a huge improvement. On my laptop, the recursive version with $n = 30$ takes about the same amount of time as the list version with $n =$50,000.

Memoization

Often, recursive solutions are simpler to code and understand (again, once you're comfortable with recursion). But the exponential running time we sometimes get is too big of a price to pay. But there is something we can do, called *memoization*, to fix the problem. The key idea is every time we compute a value of the function, we store the value in a cache somewhere. If we need future values, we can look them up in the cache instead of recomputing them via the function. For the Fibonacci code, we could do the following:

```
cache = {}
def fib(n):
    if n==0 or n==1:
        return 1
    if n in cache:
        return cache[n]
    x = fib(n-1)+fib(n-2)
    cache[n] = x
    return x
```

This code will run nearly as fast as the iterative code using a list. But there are two problems: One is that we have cluttered up our short and clear recursive solution with stuff dealing with the cache. The other is that we are using a global variable, which is not particularly good style, especially in larger programs. Below is a better solution that we can reuse for other problems:

```python
def memo(f):
    cache = {}
    def g(*args):
        if args in cache:
            return cache[args]
        x = f(*args)
        cache[args] = x
        return x
    return g

@memo
def fib(n):
    if n==0 or n==1:
        return 1
    return fib(n-1) + fib(n-2)
```

The basic overview is that we are creating a function called `memo` that takes a function and returns an improved version of the function (the `g` function created in the code) that remembers past values. The `@memo` syntax is called a *decorator*. It's basically shorthand for `fib = memo(fib)`.

The `memo` function itself is a little tricky to understand. First, it creates a local `cache` variable. Then we create a new function called `g`. This function is sometimes referred to as a *closure*, and it has access to the `cache` variable. The `*args` syntax in the declaration of `g` is a way to specify a function with an unknown number of parameters. This allows the `memo` function to be applied to any function, regardless of how many parameters it has. The `*args` syntax three lines below is a little different. To understand it, suppose we have a function `f(a, b)`. One way to call it is simply as `f(1, 2)`. Another way is as `f(*[1, 2])`. That is, we have a list `[1,2]`, and the `*` operator tells Python to expand that list out to fill out the parameters of the function. This is a neat trick that is occasionally useful, and it's useful for our `memo` function.

We can reuse this `memo` function anytime we have a recursive function and we want to get a speedup by remembering previously computed values. All we have to do is include the memo function and then add the `@memo` decorator right before the recursive function's declaration.

Change-making problem

The change-making problem is about determining which coins are used to make a given amount of change. For instance, common American coins are 1, 5, 10, and 25 cents (pennies, nickels, dimes, and quarters). If we need to make 73 cents, we can do that by taking 2 quarters, 2 dimes, and 3 pennies.

There is a nice greedy algorithm, used by millions of people, that works for American coins: take as many quarters as possible, then take as many dimes as possible, then as many nickels as possible, and finally take as many pennies as possible. Unfortunately, the greedy algorithm doesn't work for all money systems. For instance, if the only types of coins are a 2-cent and a 5-cent piece, and we have to make change for 8 cents, the greedy algorithm would say to take a 5-cent piece to start, and then we would be stuck. However, there is a dynamic programming approach that will work for all money systems.

Let's look at this using the American change example of 1, 5, 10, and 25 cents. Remember that the idea of dynamic programming is to use smaller solutions to build up bigger ones. Here we will approach things in a bottom-up way, starting with trying to make change for 1 cent, then 2 cents, then 3 cents, etc. These solutions will be useful in finding solutions for larger amounts of cents.

In general, to try to make change for i cents, we first check if i is one of the coin values. If it is, then we're done. Otherwise, we loop over the list of coin values in reverse order, looking at the previously found answers for $i-25$, $i-10$, $i-5$, and $i-1$, or at least the ones for which the subtraction is positive. For instance, if i is 9, we would only look at $i-5$ and $i-1$. When we find one for which we have a solution, we add the new coin to it. For instance, if $i = 37$, we would first look at $i-25$, which is 12. The solution for that would turn out to be

$(1,1,10)$, and we add the quarter into it to get $(1,1,10,25)$ as our solution for $i = 37$. Here is one way to code all of this:

```
def coin(C, val):
    T = [()]
    for i in range(1, val+1):
        if i in C:
            T.append((i,))
        else:
            for x in C[::-1]:
                if i >= x and T[i-x] != ():
                    T.append(T[i-x] + (x,))
                    break
            else:
                T.append(())
    return T[val]
```

Here `C` is the list of coin values, `[1, 5, 10, 25]` for American coins, assumed to be sorted. The loop builds up a table of how to make change for values from 1 up to `val`. The first if statement checks if the current value is one of the coin values. If it is, the entry for that value is a tuple consisting of a single item, namely that value. Otherwise, we look at earlier table values as described earlier (e.g., $i-25$, $i-10$, $i-5$, and $i-1$ for American coins). The else case near the bottom is for the situation where none of the earlier values we examine gives us anything we can work with. In that case, there isn't any way to make change. For instance, if the coin values are 2, 4, and 6, and we are looking at $i = 7$, then $i-6$, $i-4$, and $i-2$ would all return empty tuples, and so there $i = 7$ would end up with an empty tuple as well, indicating there is no way to make change for it.

Here is a recursive version of the above. It doesn't suffer from the exponential explosion that the Fibonacci code does, so it's not necessary to memoize it. If we did want to do that, we would just need to copy in the `memo` function from earlier and add `@memo` right before the declaration of `coin2`.

```
def coin2(C, val):
    if val in C:
        return (val,)
    for x in C[::-1]:
        if val >= x:
            t = coin2(C, val-x)
            if t:
                return t + (x,)
    return tuple()
```

Longest increasing subsequence

For this problem, we are given a sequence of numbers like 2 4 8 1 5 3 7 4 9, and we want to take a portion of it (a subsequence). The items taken need not be all next to each other. For example, 4 5 9 is a subsequence of the sequence above. This is indicated below with the asterisks indicating the positions in the original sequence.

```
2 4 8 1 5 3 7 4 9
  *     *       *
```

The goal of this problem is to find the longest one of these subsequences in which the terms are increasing, which means the terms will be continually growing larger or staying constant.

Remember that the idea behind dynamic programming is to figure out how to use smaller cases to solve larger ones. For the longest increasing sequence problem, what we'll do is for each stopping index, find the longest increasing subsequence that ends at that index. Below is the sequence we are looking at as well as a table of the longest increasing sequence ending at each index.

Index	0	1	2	3	4	5	6	7	8
Sequence	2	4	8	1	5	3	7	4	9

Stopping index	Best subsequence
0	2
1	2 4
2	2 4 8
3	1
4	2 4 5
5	2 3
6	2 4 5 7
7	2 4 4
8	2 4 5 7 9

The first couple of rows we can figure out pretty quickly just by looking at the sequence. But as we get farther into the sequence, it gets trickier to keep track of all the possible subsequences.

The way dynamic programming comes into this is we use earlier rows of the table to build up later rows. For instance, let's see how to get the sequence for stopping index 6 (where the sequence value is 7). To do this we look at the sequences from each of the rows from 0 to 5 and try adding a 7 to it. Here are rows 0 to 5:

0	2	2 7
1	2 4	2 4 7
2	2 4 8	~~2 4 8 7~~
3	1	1 7
4	2 4 5	2 4 5 7
5	2 3	2 3 7

Notice that for row 2, we can't add a 7 because then the sequence wouldn't be increasing. The longest of these is the 2 4 5 7, and that becomes our entry for row 6.

Coding it

While it's possible to write this recursively, it's probably easier to just write it the bottom up way with lists.

```python
def lis(L):
    best = [[L[0]]]
    for i in range(1, len(L)):
        M = [x + [L[i]] for x in best if L[i] >= x[-1]]
        best.append([L[i]] if M==[] else max(M, key=len))
    return max(best, key=len)
```

The `best` list initially starts with the first item of the list. This corresponds to the first row of the table we built above. To create row `i` of the table (entry `i` of `best`), we try to attach `L[i]` to each of the results of the previous row, only doing so if `L[i]` fits onto the end of the sequence in an increasing way. This is what the first line of the for loop does, building up a list of all the possibilities. In the next line, we find the possibility with the largest length. This is done with the `max` function, feeding it a `key=len` argument. Finally, in the return line, we find the longest entry in the `best` list.

The cut rod problem

Suppose we are given a rod that we can either sell as one big piece or break into smaller pieces. Certain lengths are worth more than others. The goal is to figure out how to cut the rod in order to get as much money as possible. For instance, say the rod is 5 units long and we have the following table of costs fpr various lengths:

length	1	2	3	4	5
cost	\$3	\$5	\$10	\$12	\$14

So if we don't cut it at all, we can sell it for \$14. If we break it into a piece of length 4 and a piece of length 1, we can sell it for $\$12 + \$3 = \$15$, an improvement. If we break it into five pieces of length 1, we can sell it for $\$3 + \$3 + \$3 + \$3 + \$3 = \$15$. What's the best possible breakdown?

The trick to this is to start small and build up the solution from there. We imagine we don't have a full length 5 piece yet. We start by assuming all we have is a length 1 piece. Then we move up to a size 2 piece and see what we can do with it. We keep increasing until we get to length 5. For each possible length, we look at what we can get for it without cutting it, and we look at how to use the info about smaller pieces to determine how we should cut it.

For length 1, there is only thing we can do, and that is to leave it alone for a cost of $3.

For length 2, we can leave it alone for $5 or we can break it into two pieces of length 1 for a total cost of $\$3 + \$3 = \$6$. We record the fact that if we have a length 2 rod, the best cost we can get for it is $6. We can summarize what we have for lengths 1 and 2 in a table as below:

length	1	2
best	$3	$6

For length 3, if we leave it alone, it's worth $10. We now take the best breakdown of length 2 and add a length 1 piece to it, and we take the best (and only) breakdown of length 1 and add a length 2 piece to it. For these we get costs of $\$6 + \$3 = \$9$ and $\$3 + \$5 = \$8$. The best option, therefore, for length 3 is to leave it alone. We record this in the table:

length	1	2	3
best	$3	$6	$10

For length 4, we have the following possibilities:

- Best breakdown of length 3, plus a length 1 piece: $\$10 + \$3 = \$13$
- Best breakdown of length 2, plus a length 2 piece: $\$6 + \$5 = \$11$
- Best breakdown of length 1, plus a length 3 piece: $\$3 + \$10 = \$13$
- Leave it alone: $12

Note that in each sum, the left value comes from the table we're building and the right value comes from the initial table of costs. We see that $13 is the best we can do. The table is now this:

length	1	2	3	4
best	$3	$6	$10	$13

Finally, for length 5, we do a similar process:

- Best breakdown of length 4, plus a length 1 piece: $\$13 + \$3 = \$16$
- Best breakdown of length 3, plus a length 2 piece: $\$10 + \$5 = \$15$
- Best breakdown of length 2, plus a length 3 piece: $\$6 + \$10 = \$16$
- Best breakdown of length 1, plus a length 4 piece: $\$3 + \$12 = \$15$
- Leave it alone: $14

So the best cost we can get is $16. Here is the final table:

length	1	2	3	4	5
best	$3	$6	$10	$13	$16

Coding it

We can try a recursive approach. For each length i from 0 to $n-1$, we look at the best breakdown of a length i piece and add a length $n-1-i$ piece to it. The case of leaving the rod alone is built into this if we think of combining a full length piece with a length 0 piece (which has cost 0). We can program this like below:

```python
def cut_rod(n, cost):
    if n==0:
        return 0
    return max(cut_rod(i, cost) + cost[n-1-i] for i in range(n))
```

This is very direct, but it suffers from the same exponential running time that the earlier recursive Fibonacci code suffers from. To deal with that, be sure to memoize it. Here is a bottom-up approach that uses lists:

```
def cut_rod(n, cost):
    best = [0]
    for j in range(1, n+1):
        best.append(max(best[i] + cost[j-i-1] for i in range(j)))
    return best[-1]
```

Notice the similarity between this and the recursive approach. In place of recursively calling the function, we use the `best` list. The append line is essentially the same as the return line from the recursive code, except for using `j` instead of `n`.

One issue with the above approaches is they return only the best cost. They don't give us the sequence of cuttings to achieve it. We can modify the code above to keep track of the cuttings, like below:

```
def cut_rod(n, cost):
    best = [(0, ())]
    for j in range(1, n+1):
        best.append(max((best[i][0] + cost[j-i-1], best[i][1] + (j-i,))
                    for i in range(j)))
    return best[-1]
```

In this code, instead of `best` storing just the best cost, it now stores the best sequence as well. We store it as a tuple, and when we append things into best, we take the best sequence and add the new cut to it. Note that there may be more than one best sequence. This code only returns one of them.

Knapsack problem

This is a famous problem in computer science. The idea is we have a list of items, each with a price and weight. We have a knapsack that has a given weight limit, and we want to fill it with as much as it can hold in order to maximize the total value of its contents. There are several versions of the problem. The version we will look at here is called the $(0,1)$ knapsack problem. In it, we are not allowed to take more than one of any item. In particular, for each item, we choose whether to skip it or to put it into the knapsack. Here is a sample list of items:

item	weight	price
A	1	4
B	3	8
C	2	7

We will solve this with dynamic programming, which means we use smaller cases to build up our solution to larger cases. One of the ways to get smaller cases is to start with less available items. Initially, we assume there is nothing available (the base case). Then we assume only A is available to be chosen. Then we assume A and B are available. And finally we assume A, B, and C are all available. Another way to get smaller cases is to start with small knapsack sizes and build up to larger ones.

Our approach to this problem will involve doing both of these things at once. In particular, we will build up the table below. The numbers across the top are the knapsack weight limits. The rows indicate what items are available. The entries are the best cost we can get for the knapsack.

	0	1	2	3	4	5	6
—	0	0	0	0	0	0	0
A	0	4	4	4	4	4	4
AB	0	4	4	8	12	12	12
ABC	0	4	7	8	12	15	19

For instance, the entry in the AB row under capacity 4 says that if we only have A and B available to choose from, and the knapsack has a weight limit of 4 pounds, then we can get a total knapsack price of \$12. Note that A is 1 pound and B is 3 pounds, so $1+3=4$ will fit in the knapsack, and the total price of A and B is $4+8=12$.

Let's figure out how to create the table. The first row is all 0s because if we aren't given any items to choose from, then we can't put anything into the knapsack. This will be useful as a base case when we program things. For the second row, only A is available, and since this is the $(0,1)$ knapsack problem, we can either take one of A or take none of A; we can't take more than one. Since A has weight 1, once our knapsack capacity is 1, then we can take A for a price of \$4. Since we can't take more than one of A, all the other entries after the first in row 2 will be 4.

Once we get past the first two rows, dynamic programming really comes into play. Let's look at how to get the entry in the ABC row for capacity 5. The entries in the table we will need to figure this out are shown below:

items	0	1	2	3	4	5	6
—							
A							
AB				8		12	
ABC						?	

One thing we could do is take whatever worked in the row above, namely whatever the optimal knapsack was when we just had A and B to choose from. In other words, we could just ignore C entirely. This would give us a price of \$12.

However, since we do now have C available, we should check if we can do better by adding C. First, we need to make sure C can even fit into the knapsack, making sure that C's weight does not exceed the knapsack capacity. C's weight is 2 and the capacity is 5, so we are good.

Now, think about if we were to jettison 2 pounds of stuff from the knapsack and replace it with C. How do we do that? We look at the entry in row AB, column 3 (where $3 = 5 - 2$). That entry holds whatever the best possible knapsack with items A and B and a capacity of 3 pounds. That knapsack has a price of \$8. If we add C at a price of \$7 to this knapsack, we would have a total price of \$15, which is better than the \$12 we would get by just going with A and B. So \$15 is what the entry will be.

In general, assume the table we are creating is called `T`, with `T[i][j]` being the entry in row `i`, column `j`, where `i` indicates that the first `i` items are available, and `j` indicates that the knapsack can hold `j` pounds of stuff. To get entry `T[i][j]`, take the larger of the following two values:

1. `T[i-1][j]` — What we get by ignoring the new item available in row `i`.
2. `T[i-1][j-W[i-1]] + P[i-1]` — What we get by dropping `W[i-1]` pounds of weight and replacing those pounds with the new item. Note that `T[i-1][j-W[i-1]]` holds the best thing we can do with the reduced weight. Also note that we only do this if `W[i-1]` does not exceed the current knapsack size, `j`.)

With that in mind, here is how we can program it. In the function below, `W` is a list of weights, `P` is a list of prices, and `cap` is what our knapsack's capacity is.

```python
def knapsack(W, P, cap):
    T = [[0]*(cap+1) for i in range(len(W)+1)]
    for i in range(1, len(W)+1):
        for j in range(1, cap+1):
            x = 0
            if W[i-1] <= j:
                x = T[i-1][j-W[i-1]] + P[i-1]
            T[i][j] = max(T[i-1][j], x)
    return T[-1][-1]
```

The code starts by creating the table, which is a two-dimensional list. We initially set the first row to all 0s. This is our base case for when there are no items available. Then we have nested loops. The `i` variable goes through the rows of the table and the `j` variable goes through the columns. In the loop, we apply the rule given above. And that's it. With dynamic programming, the hard part comes in figuring out how to break the problem up into smaller subproblems. Once that is done, the code usually involves building up a list or table in a similar way to this.

Note that `T[-1][-1]`, which is the bottom right entry in the table, is the actual best value we can get for the knapsack. If you want to nicely print the table, here is some code for that:

```python
def print_2dlist(a):
    for i in range(len(a)):
        for j in range(len(a[0])):
            print('{:2d}'.format(a[i][j]), end=' ')
        print()
    print()
```

If we actually want the list of items that give the optimal cost, we can do that by making each table entry be a tuple containing both the cost and the items. It's a nice exercise to try, though it's a bit tedious.

Edit distance

In this problem, we are given two strings and we want to transform one of them into the other using as few edits as possible. Here are the edits that are allowed:

1. Replacing one letter with another
2. Inserting a letter
3. Deleting a letter

For example, suppose we want to know the edit distance from the word *apple* to the word *sample*. We change *apple* to *sample* by inserting an *s* at the start and replacing the first *p* with *m*. This is two edits in total, and it's as good as we can do since it's not possible change *sample* to *apple* in just one edit. So the edit distance from *apple* to *sample* is 2. Often if a word is misspelled, words with close edit distances to the misspelled word are the most likely words the person meant. Edit distance is also used for determining how closely related two DNA sequences are.

To use dynamic programming, we need to break this the problem into smaller problems. To do this, we will look at substrings of each of the words, specifically substrings that start at the start of the string (called *prefixes*). For instance, the prefixes of *apple* are the empty string, *a*, *ap*, *app*, *appl*, and *apple*. For each of these prefixes, we will look at the edit distance from it to the prefixes of the other word (which will be *sample* in this example). Below is the table of all the edit distances. We'll go over how to generate it.

	–	s	a	m	p	l	e
–	0	1	2	3	4	5	6
a	1	1	1	2	3	4	5
p	2	2	2	2	2	3	4
p	3	3	3	3	2	3	4
l	4	4	4	4	3	2	3
e	5	5	5	5	4	3	2

1. _ to *s* — insert *s* (cost 1)
2. *a* to *a* — no change
3. *p* to *m* — replace *p* with *m* (cost 1)
4. *p* to *p* — no change
5. *l* to *l* — no change
6. *e* to *e* — no change

First note that we add a blank character at the start of each word. This will be helpful as a base case for programming. As an example of what the table tells us, the entry in the second *p* row and *m* column says that the edit distance from *sam* to *app* is 3 (namely three replacements). Similarly, the entry in the *l* row and *l* column says that the edit distance from *appl* to *sampl* is 2 (namely inserting an *s* and changing a *p* to an *m*). The bottom right entry in the table is the final edit distance.

Moving diagonally down and right in the table corresponds to either a replacement (adding 1 to the total edit distance) or letters that match in the two positions (adding nothing to the total edit distance). Moving right corresponds to an insertion and moving down corresponds to a deletion.

Highlighted in the table is an optimal path from the start at the top left to the end at the bottom right. On the right is a description of the path. To find that path, start at the bottom right and work your way to the top left, making sure the values on the path always stay equal or go down.

In general, assume the table we are creating is called `T`, with `T[i][j]` being the entry in row `i`, column `j`, where `i` and `j` indicate the lengths of the two substrings. To get the entry `T[i][j]`, we look at the entries directly above, directly to the left, and directly above and left of the current entry. See below.

```
T[i-1][j-1]   T[i-1][j]
T[i][j-1]     T[i][j]
```

There are three ways in which these entries affect the edit distance:

- Going down from `T[i][j-1]` to `T[i][j]` corresponds to inserting a letter, a cost of 1 to the edit distance.
- Going right from `T[i-1][j]` to `T[i][j]` corresponds to deleting a letter, a cost of 1 to the edit distance.
- Going diagonally from `T[i-1][j-1]` to `T[i][j]`, we look at the current letters in both words. If they match, then there is no cost in terms of edit distance to go from `T[i-1][j-1]` to `T[i][j]`. If they are different, then the cost is 1, corresponding to a letter replacement.

We want to find which of these gives the minimum cost. Here is an example. In it, we want the entry in the *l* row and *m* column.

	_	s	a	m	p	l	e
_							
a							
p							
p			3	3			
l			4	?			
e							

The ? entry comes by finding the minimum of the following:

1. Take the entry directly above, 3, and add 1 to get 4.
2. Take the entry directly left, 4, and add 1 to get 5.
3. Take the entry diagonally above and left, a 3. Since the current letters, *m* and *l*, don't match, we have to add 1 to get 4.

The smallest of these is 4, so that's what the entry will be.

Just like with the knapsack problem, the hard part is figuring out the breakdown into subproblems. We did that above. Now that we have the breakdown, we can write the code to build the table. Just like with the knapsack code, we first create the table (a 2d array), and then use nested loops to build up the table.

```python
def edit_distance(a, b):
    T = [[0]*(len(b)+1) for i in range(len(a)+1)]
    for i in range(len(a)+1):
        T[i][0] = i
    for j in range(len(b)+1):
        T[0][j] = j

    for i in range(1,len(a)+1):
        for j in range(1,len(b)+1):
            x = T[i][j-1] + 1
            y = T[i-1][j] + 1
            z = T[i-1][j-1] + (0 if a[i-1]==b[j-1] else 1)
            T[i][j] = min(x, y, z)
    return T[-1][-1]
```

The code starts by creating the table and filling up the first row and column. These initial entries are all about edit distances from the empty string. They increase 0, 1, 2, 3, etc. as the size of the substrings increase. The rest of the code consists of nested loops that build the table up row by row and column by column. Inside the loops, we compute the three values mentioned above, and the minimum of those becomes the new table entry. The final answer is the bottom right entry in the table. Just like with some of the other problems we've looked at, it is possible to modify the code to record the sequence of edits.