

# Dynamic Programming

Tyler Adams

April 18, 2010

This course is designed to give an early exposure to Dynamic Programming and how to apply it.

My goal is that you learn to appreciate one technique from the world of mathematics and discover a some patterns along the way.

## 1 Overview

Dynamic Programming is a technique which is used exclusively to evaluate recursive functions faster. Keep in mind that faster can often mean that hundreds of years becomes a few seconds. Luckily, there are many problems that can be solved through a recursion. Some Dynamic Programming implementations you may find obvious and others incredibly clever and complex. The range of difficulty in this course will be quite large.

This course will stay in theory for the most part (this is simply due to background and personal choice). Dynamic Programming, however, is highly applicable in the real world. One such application that you know very well is the core of computer navigations. We will discuss these later in the class.

## 2 Review of Recursion and Motivation

One should review recursion before optimizing it through the use of Dynamic Programming. If you are comfortable with recursion, you may want to skip this chapter.

A recursive function is one with a set of base cases as well as a sequence of instructions which will reduce any input to a function of the base cases.

## 2.1 Fibonacci

Recall the definition of the Fibonacci function.

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \text{ if } n \geq 2$$

Now I ask you to calculate  $F_{14}$  by hand in 30 seconds? Can you do it?

If the answer is yes then you may already know Dynamic Programming!

One way to solve this problem (given  $\infty$  time) is to simply plug the recursion in its raw form.

$$F_{14} = F_{13} + F_{12} = F_{12} + F_{11} + F_{11} + F_{10} = \dots \quad (1)$$

However, (1) takes a very long time. In fact as we increase  $n$ , we notice the number of recursive calls gets very large. In Computer Science we would say it grows in  $O(\phi^n)$  time. All you have to understand is that even fairly small  $n$  (like 100) yield huge times. So to calculate large Fibonacci numbers, one might want a new technique.

## 3 A first look at Dynamic Programming

Using this problem let's try the following. We notice that  $F_n = F_{n-1} + F_{n-2}$  requires us to only know the values of  $F_{n-1}$  and  $F_{n-2}$ . If the Fibonacci numbers are calculated and stored in a special order, the following is possible.

$$F_0 = 0$$

$$F_1 = 1$$

$$F_2 = F_1 + F_0 = 1 + 0 = 1$$

$$F_3 = F_2 + F_1 = 1 + 1 = 2$$

$$F_4 = F_3 + F_2 = 2 + 1 = 3$$

$$F_5 = F_4 + F_3 = 3 + 2 = 5$$

$$F_6 = F_5 + F_4 = 5 + 3 = 8$$

Continuing this method yields 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, *etc....* and eventually  $F_{14} = 377$ . Notice that approximately 14 operations were performed to achieve this result  $F_{14}$ . If recursion were used, at least 377 operations ( $1+1+\dots+1$  w/377 1s) would have been made. For both computers and humans, the former is much easier to perform.

This is Dynamic Programming. Smaller cases are used to build larger cases and eventually the desired case. This can bring huge speed reductions.

For the Fibonacci numbers,  $O(n)$  time complexity was achieved by using DP (Dynamic Programming). Notice that  $n$  is WAY smaller than  $\phi^n$ . Try plugging in  $n = 10, 100, 1000$  to get an idea of how much smaller this really is.

## 4 Math problems

This section will use Dynamic Programming to quickly evaluate relatively simple recursive relationships. This chapter does not demonstrate are not terribly creative applications, unlike those in Chapter 5. The goal of this chapter is to become familiar with the idea of taking a recursive function and evaluating it through DP.

### 4.1 Problem 11: AIME I 2008

Consider sequences that consist entirely of A's and B's and that have the property that every run of consecutive A's has even length, and every run of consecutive B's has odd length. Examples of such sequences are AA, B, and AABAA, while BBAB is not such a sequence. How many such sequences have length 14?

Let  $A(x)$  be the number of valid sequences that have length  $x$  and begin with an A.

Define  $B(x)$  similarly be the number with length  $x$  and begin with a B.

By construction

$$A(1) = 0, A(2) = 1 \tag{2}$$

$$B(1) = 1, B(2) = 0 \tag{3}$$

Let  $x \geq 3$ . Assume  $\alpha$  is a valid sequence with length  $x$  and begins with an A. Then the 3rd character of  $\alpha$  is either an A or a B. The tail beginning with the 3rd character must also be a valid sequence of length  $x - 2$ . If it begins with an A then the run of As in the beginning of the large sequence is even + 2 which is still even. If the smaller sequence begins with B, then the run of As is 2 long, which is even.

A valid sequence of length  $x - 2$  prepended with 2 As will automatically be a valid sequence of length  $x$ .

Given this relationship, it becomes apparent that for every  $\alpha$ , it corresponds

to a unique valid tail of length  $x - 2$ . Thus  $A(x) = B(x - 2) + A(x - 2)$ . Now let  $\beta$  be a valid sequence with length  $x$  and begins with a B. Then the 2nd character of  $\beta$  is either an A or a B. If it is an A and the tail beginning with the second character is valid, then  $\beta$  will correspond to a valid tail beginning with an A and length  $x - 2$ . If it is a B then the 3rd character must be a B (because the sequence must be odd). We also note that the subsequence beginning with the 3rd character must also have an odd number of Bs. Thus this sequence corresponds to a sequence length  $x - 2$  beginning with a B.

Thus  $B(x) = B(x - 2) + A(x - 1)$ .

Given the definitions, the goal is to evaluate  $A(14) + B(14)$ . Straight recursion would take forever. Memoization would be ugly and hard to manage. However, by simply building a table and reaching the 14th row, the answer can be easily and quickly calculated.

-	A	B
1	0	1
2	1	0
3	1	2
4	1	1
5	3	3
6	2	4
7	6	5
8	6	10
9	11	11
10	16	21
11	22	27
12	37	43
13	49	64
14	80	92

so  $A(14) + B(14) = 80 + 92 = \boxed{172}$

## 4.2 Problem 17: AMC12B 2010

"The entries of a 3x3 array include all the digits from 1 through 9, arranged so that the entires in every row and column are in increasing order. How many such arrays are there?"

Write out the sequence 1,2,3,...,9 with the following substitution. If the number is on the top row replace it with a t, and the middle row, replace it

with a m, bottom row, a b. For example with the filling:

1	3	7
2	4	8
5	6	9

becomes *tmtmbbtmb*. Notice that for any tail of the sequence (a subsequence which contains the last character) the number of bs  $\geq$  number of ms  $\geq$  number of ts (what would happen if this were false)? This suggests a function  $f(x, y, z)$  which returns the number of tails with x ts, y ms, and z bs would be appropriate. Every tail of a tail must also be valid. Combining these facts suggest defining  $f$  like such.

$f(x, y, z) =$

1 if  $x = y = z = 0$ . There is one empty sequence

0 if  $y < x$  or  $z < y$ . It will not have a proper tail

0 if  $x < 0$  or  $y < 0$  or  $z < 0$ . Bad inputs.

$f(x - 1, y, z) + f(x, y - 1, z) + f(x, y, z - 1)$  otherwise. Think of it like trying every possible tail after putting a t or m or b in front.

$f(3, 3, 3)$  is the goal since we have 3ts, 3ms, 3bs. Since we assume  $0 \leq x \leq y \leq z \leq 3$ , we can be smart about the values which we compute.

We construct the following table in such a way that if  $(a, b, c)$  comes before  $(\alpha, \beta, \gamma)$  (and both have non-zero function evaluations), then  $c \leq \gamma$  and if  $c = \gamma$ ,  $b \leq \beta$  and if  $b = \beta$  then  $a \leq \alpha$  and if  $a = \alpha$  then  $(a, b, c) = (\alpha, \beta, \gamma)$ . Thus if  $f(\alpha, \beta, \gamma)$  is dependent on  $f(a, b, c)$  then  $f(a, b, c)$  will appear before  $f(\alpha, \beta, \gamma)$

x	y	z	$f(x, y, z)$
0	0	0	1
0	0	1	$1 = f(0, 0, 0)$
0	1	1	$1 = f(0, 0, 1)$
1	1	1	$1 = f(0, 1, 1)$
0	0	2	$1 = f(0, 0, 1)$
0	1	2	$2 = f(0, 0, 2) + f(0, 1, 1)$
1	1	2	$3 = f(0, 1, 2) + f(1, 1, 1)$
0	2	2	$2 = f(0, 1, 2)$
1	2	2	$5 = f(0, 2, 2) + f(1, 2, 2)$
2	2	2	$5 = f(1, 2, 2)$
0	0	3	$1 = f(0, 0, 2)$
0	1	3	$3 = f(0, 0, 3) + f(0, 1, 2)$
1	1	3	$6 = f(0, 1, 3) + f(1, 1, 2)$
0	2	3	$5 = f(0, 1, 3) + f(0, 2, 2)$
1	2	3	$16 = f(0, 2, 3) + f(1, 1, 2) + f(1, 2, 2)$
2	2	3	$21 = f(1, 2, 3) + f(2, 2, 2)$
0	3	3	$5 = f(0, 2, 3)$
1	3	3	$21 = f(0, 3, 3) + f(1, 2, 3)$
2	3	3	$42 = f(1, 3, 3) + f(2, 2, 3)$
3	3	3	$42 = f(2, 3, 3)$

So the answer is  $\boxed{42}$ .

If you are confused by the summations, note that I did not bothering adding  $f(x,y,z)$  that I could tell would equal 0.

Author's note: I personally think that this application of Dynamic Programming is far cooler than the one before. Unlike the previous recursion, DP is non-obvious even after the recursion has been found and we used tricks to evaluate it quickly. Dynamic Programming can often require more than a recursive relationship. Sometimes other tricks are required to reduce speed to a manageable level.

## 5 Computer Science

Computer Science is not the science of computers. It is the science of what can be computed. The formal definition is complicated but if a problem can be solved by a machine which follows a set of instructions and terminates with a result, then it is computable. Dynamic Programming (which has its

roots before modern day programming) is a prime tool for this field. Dynamic Programming exists to compute recursive functions quickly. Therefore, what a better place to apply it than a field devoted to computation. Fast computation is of course important. When solving a problem with Dynamic Programming, there are two parts. The first is finding a recursive relationship between smaller cases and bigger cases. The second is finding a way to evaluate the subproblems. Think back to the previous problem. Finding  $f$  was the recursive relationship. Evaluating  $f$  by constructing a clever order was a way of evaluating the subproblems such that the final problem would be evaluated within time. Sometimes too much data is required to store the smaller cases required to build up a big case (we will touch upon this later).

## 5.1 Reading and writing to variables

Since we are going to enter the realm of computer science, I quickly want to introduce some notation and concepts. Programmers can skip this section. In algebra there is a concept of a variable. Sometimes the variable can vary, or othertimes it is fixed (which means it is an unknown). In computer science it is nice to have variables from which we can read data from and to which we can write data. Check out the following statement for variables  $x, y$   
 $x = 3$ . This is a declaration writing the value of  $x$  to 3. The variable on the left is being written upon.

$y = x$ . This is a declaration writing the value of  $y$  to the value of  $x$ . The variable on the right is being read.

## 5.2 How Fast is Dynamic Programming? Pretty fast!

Let's use a computer to gather some data. Blindly defining Fibonacci in Scheme yields.

```
(define ( fib x)
  (cond
    ((= x 0) 0)
    ((= x 1) 1)
    (else (+ (fib (- x 1)) (fib (- x 2))))))
```

Basically if  $x$  is 0 return 0,  $x$  is 1 return 1, otherwise return  $f(x-1)+f(x-2)$ . Now let's try to find  $F_{10}, F_{30}$ . We could try evaluating  $F_{100}$  but we would have to wait many years for the computer to finish.

Using DP to get  $F_x$  try the following.

Allocate readable and writeable variables  $F_0$  through  $F_x$

$F_0 = 0$

$F_1 = 1$

for( $i = 2; i \leq x$ ; increment  $i$  by 1)  $F_i = F_{i-1} + F_{i-2}$

now  $F_x$  is properly evaluated.

One might notice that after  $i = 3$ , one never calls upon  $F_1$  or  $F_0$  again. Check out this lightning fast and tight code

Allocate r/w variables a,b

if( $x$  is 0)  $F_x = 0$

if( $x$  is 1)  $F_x = 1$

otherwise

$a = 0$

$b = 1$

$c = 1$

```
for( $i = 2; i \leq x$ ; increment  $i$  by 1) {  
     $a = b$   
     $b = c$   
     $c = a + b$   
}
```

$c$  is the  $F_x$ .

in Scheme that code looks somewhat like

```
(define (fib x)  
  (cond  
    ((= x 0) 0)  
    ((= x 1) 1)  
    (else (fibhelp 0 1 x))))  
(define (fibhelp a b x)  
  (cond  
    ((= x 1) b)  
    (else (fibhelp b (+ a b) (- x 1)))))
```

Don't try to understand the code just yet. It's different because it can set  $a$  to be  $b$  and  $b$  to  $a + b$  simultaneously.

For now just imagine that it does the last piece of pseudo code and we'll look at some runtimes for  $F_{10}$ ,  $F_{35}$ ,  $F_{100}$  and  $F_{100,000}$ .

We can see the code does take advantage of building up smaller cases to solve our large case.

### 5.3 Shortest Path (Dijkstra)

Graph Theory is a field important to both mathematics and computer science. Mathematics discusses theorems about graphs (like colorings and cycles). Computer Science tackles computable problems. One such problem is the shortest path between two nodes in a directed graph with non-negative edge weights (wow that's a mouthful). In class I will quickly define graphs but I will omit that on this handout. To give a quick analogy, cow pastures with paths that connect them are graphs. The pastures are nodes (or vertices) and the paths are edges. The length of a path is its weight. If the paths are 1-way then the edge is directed. If Bessie the cow is in one pasture, what is the shortest route from her pasture to any other given pasture?

This problem is, surprisingly, recursive. The idea for shortest path lies on the fact that

$$\text{mindist}(\text{source}, \text{target}) = \min(\text{mindist}(\text{source}, x) + \text{dist}(x, \text{target})) \quad (4)$$

for all  $x$  neighbor of target (there exists an edge between  $x$  and target).  
and

$$\text{mindist}(\text{source}, \text{source}) = 0 \quad (5)$$

However, simply applying recursion to this scenario will not work. Note that blindly calling this function might not ever terminate. Therefore, a clever Dynamic Programming algorithm is actually *required* to solve this problem. Look at the following.

Begin by setting r/w variables  
 $\text{mindist}(\text{source}, \text{target}) = \infty$  for  $\text{target} \neq \text{source}$

$\text{mindist}(\text{source}, \text{source}) = 0$

We also mark all of the nodes as unvisited (r/w variables  $\text{visit}(x) = \text{false}$  for all  $x$ ).

We perform the following loop until we break from it.

For the unvisited node closest to the source, denoted  $y$ , (in the case of a tie we can pick arbitrarily).

mark  $y$  as visited ( $\text{visit}(y) = \text{true}$ )

for all neighbors  $x$  of  $y$

$\text{mindist}(\text{source}, x) = \min(\text{mindist}(\text{source}, x), \text{dist}(x, y) + \text{mindist}(\text{source}, y))$ . where  $\text{dist}(x, y)$  is the length of the shortest edge from  $x$  to  $y$

This is a lot of ugly notation. Let's break the concept down. We try to find a shorter path to  $x$  by taking the shortest path from source to  $y$  (somehow it was found) and then a known path from  $y$  to  $x$  (the shortest edge connecting them) and comparing it to the best path we have found so far. This will cover every possible path to  $y$  as any path to  $y$  from source must pass through a neighbor  $x$ , unless  $y$  is the source itself (in which case the distance is 0). This is essentially Dynamic Programming. To find the minimum path, Dijkstra builds up the smaller cases to reach the larger ones. This is then run until  $y = \text{target}$ . When  $y$  is target we can be sure that  $\text{mindist}(\text{source}, \text{target})$  is in fact the shortest path.

Proof that this algorithm works in a nutshell: induction.

The first case will be when  $y$  is the source. Trivially 0 is the shortest path to source so we have found it.

The induction hypothesis is that when we are at  $y$ ,  $\text{mindist}(\text{source}, y)$  is in fact the shortest path to  $y$  from source. The proof is not complex, but will be omitted in this lecture.

### 5.3.1 GPS/Mapquest Navigation

The problem of shortest path should sound familiar. GPS navigation devices and Google Maps are asked this problem daily. Roads look like edges and intersections look like nodes. By combining speed limit, length, shape, condition, and traffic, one could assign weights to roads such that the weight corresponds to the average time it takes to traverse the road. So with the proper modelling, UI, and data you too could make a navigation device.

## 5.4 Integer Knapsack Problem and other filling problems

These problems are fairly similar but have a few differences. Try to find some common themes and ideas.

### 5.4.1 Integer Knapsack Problem

Imagine that you are a farmer selling fruit at a market. You have a cart but it can only hold up to 100kg before breaking. You also know how much each piece of fruit weighs and how much it sells for (we assume the market is big enough that you cannot influence the price). What is the maximum profit? This is a pretty lame example but it illustrates a useful computer problem. Namely the knapsack problem. There are three knap sack problems, but we will only cover the integer knapsack problem in this course. There is a sack with capacity  $C$  and objects with weights  $w_i$  and values  $v_i$ . Maximize the value in the sack without exceeding the capacity or breaking the items.

Unsurprisingly this is more dynamic programming!

The maximum value stored for a given weight is denoted as  $v[w]$ .

$$v[w] = \begin{cases} 0 & \text{if } w \leq 0 \\ \max(v_i + v[w - w_i] & \text{for all } i \text{ such that } w_i \leq w \end{cases}$$

Again we see a recursion and again we will turn it upon its head to solve this problem.

We assume to start with that  $v[x] = 0$  for all  $x$ .

Then for all objects  $j$  and for each object iterating backwards through  $i$  from  $w - w_j$  to 0, we determine  $v[i + w_j] = \max(v[i + w_j], v_j + v[i])$ .

Then reading the value of  $v[w]$  will yield the maximum profit. Iterating backwards ensures that putting an object in the sack is part of what determines  $v[w]$ , then it does not get reprocessed (in otherwords we do not double count our objects by running backwards). Running forwards simulates having a variety of objects and having an infinite quantity of each one.

USACO GOLD PROBLEM (December 2009, Gold division)  
Problem 3: Video Game Troubles [Jeffrey Wang, 2007]

Farmer John's cows love their video games! FJ noticed that after playing these games that his cows produced much more milk than usual, surely because contented cows make more milk.

The cows disagree, though, on which is the best game console. One cow wanted to buy the Xbox 360 to play Halo 3; another wanted to buy the Nintendo Wii to play Super Smash Brothers Brawl; a third wanted to play Metal Gear Solid 4 on the PlayStation 3. FJ wants to purchase the set of game consoles (no more than one each) and games (no more than one each – and within the constraints of a given budget) that helps his cows produce the most milk and thus nourish the most children.

FJ researched  $N$  ( $1 \leq N \leq 50$ ) consoles, each with a console price  $P_i$  ( $1 \leq P_i \leq 1000$ ) and a number of console-specific games  $G_i$  ( $1 \leq G_i \leq 10$ ). A cow must, of course, own a console before she can buy any game that is specific to that console. Each individual game has a game price  $GP_j$  ( $1 \leq GP_j \text{ price} \leq 100$ ) and a production value ( $1 \leq PV_j \leq 1,000,000$ ), which indicates how much milk a cow will produce after playing the game. Lastly, Farmer John has a budget  $V$  ( $1 \leq V \leq 100,000$ ) which is the maximum amount of money he can spend. Help him maximize the sum of the production values of the games he buys.

This is a knapsack problem with a twist. Games are certainly have a weight (price) and a value (milk production value). However, in order to buy a game, one must own the game system which has no intrinsic value. Therefore we use the following trick.

Like in all knapsack problems,  $Val[x]$  will store the highest amount of Value one can get for  $\$x$

Create an array  $tempVal$

Let  $p[i][j]$  be the cost of game  $j$  for system  $i$

Let  $v[i][j]$  be the production value of game  $j$  for system  $i$

Let  $P[i]$  be the cost of system  $i$

for every game console  $i$

set  $tempVal[x] = Val[x]$  for all  $x$ .

for every game  $j$   
 iterating through every dollar  $k$  from  $V$  to 0  
 $tempVal[k + p[i][j]] = \max(tempVal[k + p[i][j]], tempVal[k] + v[i][j]).$   
 for every dollar  $k$  from  $P[i]$  to  $V$ .  
 $Val[x] = \max(Val[x], tempVal[x - P[i]]).$

We want to compare  $Val[x]$  and  $tempVal[x - P[i]]$  because  $tempVal[x - P[i]]$  has the production value that can be obtained by spending  $\$x$  because we now have to add the price of the game system.

### 5.4.2 Change

A problem many of you may have faced as a child is how many ways can one make change for a dollar (it is 293). It was probably tedious and required carefully keeping track of every way to make \$.05. However, with dynamic programming, this problem becomes rather simple.

Given coins of denominations  $d_i$ , observe that the number of ways to make  $x$  cents,

$$change[x] = change[x - 1] + change[x - 5] + change[x - 10] + change[x - 25] + change[x - 50] + change[x - 100]$$

$change[0] = 1$  as there is one way to make 0 cents out of any coin denominations.

$change[x] = 0$  if  $x < 0$  as there are no ways to make change for a negative amount.

The DP solution looks eerily similar to the knapsack problem. Note that  $v_i$  is the value of coin  $i$ .

Set  $change[0] = 1$  and

$change[i] = 0$  for all  $i \neq 0$

for all coin denominations  $d_j = 1, 5, 10, 25, 50, 100,$

for all values  $i = 0$  to 100 (or whatever amount we want to make change of)  
 $change[i + d_j] += change[i]$  (we add  $change[i]$  to  $change[i + d_j]$ )

then  $change[100]$  is the number of ways to make change for a dollar. Try it with number of ways to make change of \$.10 and \$.30 (we will do \$.30 in

lecture).<sup>1</sup>

## 5.5 Telephone Wire (USACO GOLD November 2007) and the sliding window trick

Many problems without an overall theme can be done recursively. Of course their implementations have to be Dynamic Programming. Watch out for memory usage though.

Problem 1: Telephone Wire [Jeffrey Wang, 2007]

Farmer John's cows are getting restless about their poor telephone service; they want FJ to replace the old telephone wire with new, more efficient wire. The new wiring will utilize  $N$  ( $2 \leq N \leq 100,000$ ) already-installed telephone poles, each with some height <sub>$i$</sub>  meters ( $1 \leq \text{height}_i \leq 100$ ). The new wire will connect the tops of each pair of adjacent poles and will incur a penalty cost  $C \cdot$  the two poles' height difference for each section of wire where the poles are of different heights ( $1 \leq C \leq 100$ ). The poles, of course, are in a certain sequence and can not be moved.

Farmer John figures that if he makes some poles taller he can reduce his penalties, though with some other additional cost. He can add an integer  $X$  number of meters to a pole at a cost of  $X^2$ .

Help Farmer John determine the cheapest combination of growing pole heights and connecting wire so that the cows can get their new and improved service.

When approaching this problem, think about the smaller cases. Personally when I saw it, I asked myself what would the answer be in the case with 1 pole, 2 poles, and 3 poles. I also asked myself what questions to ask. The question to ask is given the  $i$ th pole with a certain height, what is the minimum telephone cost (denote this  $f(i, h)$ ).

$f(1, h) = (h - H[1])^2$  for  $h \geq H[1]$  and infinity otherwise

$f(2, h) = (h - H[2])^2 + \min(C|h - h'| + f(1, h'))$  for all  $h' \geq H[1]$

$f(3, h) = (h - H[3])^2 + \min(C|h - h'| + f(2, h'))$  for all  $h' \geq H[2]$

---

<sup>1</sup>This and its solution are adapted from a USACO Problem, Counting Change.

This pattern alludes to  
 $f(i, h) = (h - H[i])^2 + \min(C|h - h'| + f(i - 1, h'))$  for all  $h' \geq H[i]$  where  $i \geq 2$

With this recursion, we can simply calculate and store  $f(i, h)$  for all  $h \leq 100$  (we use 100 because we know all of the poles are  $\leq 100$  meters tall). Creating a pole 101 meters tall would always cost more than creating a pole 100 meters tall. Once  $f(N, h)$  is calculated for all  $h$ , one must simply find the minimum over all  $h$ . However, this causes a problem. If one tries to store  $f(100000, 100)$  they must store up to 10 million integers. This requires too much storage space.

Luckily there is a solution to this problem, and you've already seen it. Remember the fibonacci numbers and how  $F_3$  was never needed again after  $F_5$  was calculated. Well, the same holds for this problem.  $f(1, h)$  is never called upon again after  $f(2, h')$  is calculated. Therefore we can use the sliding window trick<sup>2</sup>. Think of the data as being stored in a giant table with a window which can slide over it. To calculate data to the left of the window, only data that can be seen behind the window is needed. As new data is calculated, the window slides and we can reallocate our storage to accommodate the new data. The core of the recursion is the same and doing a dynamic programming algorithm is still possible but for a computer to carry out the algorithm, some work must be done behind the scenes. Here is generic pseudo code for the sliding window.

Slide the current state one state over (as to push off the oldest and now useless state).

Process old state to calculate current state.

Loop as necessary.

Personally I'm unhappy with the above explanation, let me give a few examples.

Reviewing fibonacci,

```
for( $i = 2; i \leq x; \text{increment } i \text{ by } 1$ ) {  
     $a = b$   
     $b = c$   
     $c = a + b$   
}
```

---

<sup>2</sup><http://tjsct.wikidot.com/sliding-window-trick>

The sliding window has old state under a,b  
 The current state is a+b.  
 b is slid over into a and c is slid into b. a is therefore pushed off as it is now useless  
 a+b is then pushed onto c  
 Then we loop.

For the telephone problem, the solution is more or less the same

```

fill  $f(1, h)$  as defined above
for( $i = 2; i \leq x$ ; increment  $i$  by 1) {
  set  $f(0, h)$  to be equal to  $f(1, h)$  for all  $h$ . (like setting  $a = b$ )
  for( $h = H[i]; h \leq 100$ ; increment  $h$  by 1) {
     $f(1, h) = (h - H[i])^2 + \min(C|h - h'| + f(0, h'))$  for all  $h' \geq H[i - 1]$ )
  }
}

```

The answer is then  $\min(f(1, h)$  for all  $H[x] \leq h \leq 100)$

This trick works for other recursions. Look at a modified Fibonacci where  
 $F_x = F_{x-1} + F_{x-2} + F_{x-3}$

The following sliding window trick works

```

if(x is 0)  $F_x = 0$ 
if(x is 1)  $F_x = 1$ 
if(x is 2)  $F_x = 1$ 
otherwise
   $a = 0$ 
   $b = 1$ 
   $c = 1$ 
   $d = 1$ 
  for( $i = 3; i \leq x$ ; increment  $i$  by 1) {
     $a = b$ 
     $b = c$ 
     $c = d$ 
     $d = a + b + c$ 
  }

```

Combining a Dynamic Programming approach with the sliding window trick is truly remarkable. Dynamic Programming is faster than simply doing recursive calls because it trades space for speed. Data about previous cases is stored and looked up at a later time. Regular recursive calls only need stack space (don't worry about it). Dynamic Programming needs a place to store information. This is of course Dynamic Programming's weakness. Sometimes too much data needs to be stored in order to build up larger cases<sup>3</sup>. However, with the sliding window trick, a very minimal amount of data needs to be stored. Thus one gets very fast code that requires little memory.

---

<sup>3</sup>A primary example is the multiple knapsack problem