CS302 Lecture Notes - Dynamic Programming

- November 18, 2009
- Latest update: November 17, 2014
- James S. Plank
- Directory: /home/plank/cs302/Notes/DynamicProgramming

To augment this lecture, there are two sets of tutorials on dynamic programming from Topcoder. They are excellent:

- "Dynamic Programming: From Novice to Advanced," by Dumitru
- "Dynamic Programming," by vorthys

Below are more Topcoder problems with hints that can help you practice:

- ConvertibleStrings - I go over this one in class.
- PageNumbers - I go over this one in class, too. This is a tricky problem where thinking recursively really helps.
- AlienAndSetDiv2 - I go over this one in class as well. This is a great problem where you can either memoize on an array, or you can use bit arithmetic to make it go smoking fast!

- Mafia - This is a dynamic program where you turn the state of your simulation into a string, and then make recursive calls on modified strings. Obviously, you memoize on the string too.
- QuickSort - This is fairly straightforward DP where you use a string as the memoization key.
- StringWeightDiv2 - This is a counting problem where again, the tricky part is using recursion to help you.

Dynamic Programming in a Nutshell

Dynamic programming is nothing more than the following:

- Step 1: You spot a recursive solution to a problem. When you program that solution, it will be correct, but you'll find that it's incredibly slow, because it makes many duplicate procedure calls.
- Step 2: You cache the answers to recursive calls so that when they are repeated, you can return from them instantly. This is called *memoization*.
- Step 3: If you want to, you can typically figure out how to eliminate the recursive calls, and instead populate the cache with one or more for loops. This is faster than memoization, but usually not by much.
- Step 4: Sometimes, you can eliminate the cache completely. This makes the program even faster and more memory efficient.

I find that steps 3 and 4 are often optional. However, they usually represent the best solutions to a problem.

I will illustrate with many examples.

#1: Fibonacci numbers

Fibonacci numbers have a recursive definition:

$$\text{Fib}(0) = 1$$
$$\text{Fib}(1) = 1$$
$$\text{If } n > 1,\ \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

This definition maps itself to a simple recursive function, which you've seen before in CS140. However, we'll go through it again. This is Step 1: writing a recursive answer to a problem. I bundle this into a class because it makes steps 2 and 3 easier. It's in fib1.cpp:

```
#include <iostream>
#include <vector>
using namespace std;

class Fib {
```

```
  public:
    int find_fib(int n);
};

int Fib::find_fib(int n)
{
  if (n == 0 || n == 1) return 1;
  return find_fib(n-1) + find_fib(n-2);
}

main(int argc, char **argv)
{
  Fib f;

  if (argc != 2) {
    cerr << "usage: fib n\n";
    exit(1);
  }

  cout << f.find_fib(atoi(argv[1])) << endl;
}
```

The problem with this is that its performance blows up exponentially, so that, for example, calculating Fib(45) takes quite a long period of time. When we teach this in CS140, we turn it into a for() loop that starts with Fib(0) and Fib(1) and builds up to Fib(*n*).

However, with dynamic programming, we proceed to step two: memoization. We accept the recursive definition, but simply create a cache for the answers to that after the first time Fib(*i*) is called for some *i*, it returns its answer from the cache. We implement it in fib2.cpp below. We initialize the cache with -1's to denote empty values. fib2.cpp

```
#include <iostream>
#include <vector>
using namespace std;

class Fib {
  public:
    int find_fib(int n);
    vector <int> cache;
};

int Fib::find_fib(int n)
{
  if (cache.size() == 0) cache.resize(n+1, -1);
  if (cache[n] != -1) return cache[n];
  if (n == 0 || n == 1) {
    cache[n] = 1;
  } else {
    cache[n] = find_fib(n-1) + find_fib(n-2);
  }
  return cache[n];
}
```

Next, we perform Step 3, which makes the observation that whenever we call find_fib(*n*), it only makes recursive calls to values less than *n*. That means that we can build the cache from zero up to *n* with a for loop (fib3.cpp):

```
#include <iostream>
#include <vector>
using namespace std;

class Fib {
  public:
    int find_fib(int n);
    vector <int> cache;
};

int Fib::find_fib(int n)
{
  int i;
  if (n == 0 || n == 1)  return 1;

  cache.resize(n+1, -1);
```

```
  cache[0] = 1;
  cache[1] = 1;
  for (i = 2; i <= n; i++) cache[i] = cache[i-1] + cache[i-2];

  return cache[n];
}
```

Finally, when we reflect that we only ever look at the last two values in the cache, we can omit the cache completely. This is Step 4 (in fib4.cpp):

```
int Fib::find_fib(int n)
{
  int v[3];
  int i;

  if (n == 0 || n == 1) return 1;

  v[0] = 1;
  v[1] = 1;
  for (i = 2; i <= n; i++) {
    v[2] = v[0] + v[1];
    v[0] = v[1];
    v[1] = v[2];
  }

  return v[2];
}
```

#2: The coin list problem from Dumitru

Here's the problem: Given a list of *N* coin denominations (V1, V2, ..., VN), and a sum *S*. Find the minimum number of coins the sum of which is *S* (we can use as many coins of one denomination as we want), or report that it's not possible to select coins in such a way that they sum up to *S*.

We start with Step 1: finding the recursion. Often the best way to do this is to try some examples. I think its easier to think of postage stamps instead of coins, because they can come in wacky denominations. And then I think of the sum as a total amount of postage, and I'm trying to put stamps on the package to equal the amount of postage. So, for example, suppose our denominations are 1, 5, 6 and 10, and our sum is 11. The minimum way of constructing the sum is one 5 and one 6, or one 10 and one 1. Since the problem asks for the minimum number of "coins", the answer is 2. Similarly, if our sum is 18, the answer is three: three 6's.

Let's make a quick table of sums and answers:

S	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
Answer	*1*	*2*	*3*	*4*	*1*	*1*	*2*	*3*	*4*	*1*	*2*	*2*	*3*	*4*	*2*	*2*	*3*	*3*
Coins	*1*	*1,1*	*1,1,1*	*1,1,1,1*	*5*	*6*	*6,1*	*6,1,1*	*6,1,1,1*	*10*	*5,6*	*6,6*	*6,6,1*	*6,6,1,1*	*10,5*	*10,6*	*10,6,1*	*6,6,6*

There doesn't appear to be a nice pattern, but think recursively. If I want to make the sum *S*, and my coins are 1, 5, 6 and 10, then I can make the sum in four ways:

- I can make the sum *S-1* and add a 1 coin.
- I can make the sum *S-5* and add a 5 coin.
- I can make the sum *S-6* and add a 6 coin.
- I can make the sum *S-10* and add a 10 coin.

So, let's suppose my procedure is *M(S)*. If I define it recursively, then my answer is going to be the minimum of *M(S-1)+1*, *M(S-5)+1*, *M(S-6)+1* and *M(S-10)+1*.

See how to write the procedure? You loop through all the values, making recursive calls. It's in coins1.cpp. I use *s+1* as a sentinel value for the minimum, and if the sum cannot be constructed, then I return -1.

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
```

```
class Coins {
  public:
    vector <int> v;
    int M(int i);
};

int Coins::M(int s)
{
  int i, j, min, sv;

  min = s+1;

  for (i = 0; i < v.size(); i++) {
    if (s == v[i]) return 1;
    if (s > v[i]) {
      j = M(s-v[i]) + 1;
      if (j != 0 && j < min) min = j;
    }
  }
  if (min == s+1) return -1;
  return min;
}

main(int argc, char **argv)
{
  Coins c;
  int i;
  int sum;

  if (argc != 2) {
    cerr << "usage: coins s -- values on standard input\n";
    exit(1);
  }
  sum = atoi(argv[1]);
  while (cin >> i) c.v.push_back(i);

  cout << c.M(sum) << endl;
}
```

A quick test shows that it works:

```
UNIX> sh
> for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ; do echo $i `echo 1 5 6 10 | coins1 $i`; done
1 1
2 2
3 3
4 4
5 1
6 1
7 2
8 3
9 4
10 1
11 2
12 2
13 3
14 4
15 2
16 2
17 3
18 3
>
```

However, like most simple recursive implementations, it is too slow. It bogs down when *S* reaches the high 40's. So, we do a simple memoization. This looks pretty much just like the memoization in the Fibonacci numbers, although I use two sentinel values in the cache: If cache[s] equals -2, then I have not calculated the value. If cache[s] equals -1, then it is impossible to make the sum s. Here is the M() method (in coins2.cpp):

```
int Coins::M(int s)
{
  int i, j, min, sv;

  if (cache.size() <= s) cache.resize(s+1, -2);
  if (cache[s] != -2) return cache[s];
```

```
  if (s == 0) {
    cache[s] = 0;
    return 0;
  }

  min = s+1;

  for (i = 0; i < v.size(); i++) {
    if (s >= v[i]) {
      j = M(s-v[i]) + 1;
      if (j != 0 && j < min) min = j;
    }
  }
  if (min == s+1) min = -1;
  cache[s] = min;
  return min;
}
```

This is much faster:

```
UNIX> echo 1 5 6 10 | coins2 505
51
UNIX> echo 1 5 6 10 | coins2 5057
507
UNIX>
```

For Step 3, as with the Fibonacci numbers, we make the observation that we are always making recursive calls from larger s to smaller s. Thus, we can build the cache from low to high values of s without using recursion. The code is in coins3.cpp. Note how similar it is to coins2.cpp - the difference is that it looks into the cache instead of making recursive calls, and it builds the cache from low to high rather than making recursive calls from high to low. I also removed the cache from the Coins class and made it a local variable to M():

```
int Coins::M(int s)
{
  int i, j, val, min, sv;
  vector <int> cache;

  cache.resize(s+1);

  cache[0] = 0;

  for (j = 1; j <= s; j++) {
    min = s+1;

    for (i = 0; i < v.size(); i++) {
      if (j >= v[i]) {
        val = cache[j-v[i]] + 1;
        if (val != 0 && val < min) min = val;
      }
    }
    if (min == s+1) min = -1;
    cache[j] = min;
  }
  return cache[s];
}
```

Finally, there is a way to do step four if you think about it. This about that last call above -- "echo 1 5 6 10 | coins2 5057". Does your cache really need 5057 elements? If you don't see the answer, ask me in class -- this is a good test question.

#3: The Maximum Subsequence Problem

This one comes from the topcoder article by vorthys. You are to write a program that takes two strings and returns the length of the largest common subsequence of the two strings. A subsequence is a subset of the letters of a string that appears in the order given by the subsequence. For example, "dog" is a subsequence if "dodger", but "red" is not, because the characters don't appear in "dodger" in the correct order.

As always, we start with Step 1, which is to spot the recursive solution. To do so, let's take a look at two examples. Here's the first example:

S1 = "AbbbcccFyy", S2 = "FxxxyyyAc"

And here's the second example.

S1 = "AbbbcccFyy", S2 = "FxxxyyyAccc"

You should see that these examples are related. In the first, the longest subsequence is "Fyy", and in the second, the longest subsequence is "Accc". This example highlights the difficulty of this problem -- If you start looking for subsequences from the beginning of either string, the result may be affected by the end of the string. We'll return to these examples in a bit. Let's try another example:

S1 = "900970994705376997373585670781133734891 4", S2 = "940488836737907475451095492239929191 2221"

Man, that looks hard. Don't even bother trying to solve it. However, one thing you do know -- the maximum subsequence starts with a 9. Why? It has to -- you can prove this to yourself by contradiction: If the subsequence is *S* and *S* doesn't start with 9, then there is a valid subsequence which is "9"+*S*.

I'm hoping these examples help you spot the recursion: You will compare the first letters of both substrings, and do different things depending on whether they equal each other:

- If S1[0] == S2[0], then the answer is one plus the maximum subsequence of S1.substr(1) and S2.substr(1). (This method returns the substring starting at index 1 and going to the end of the string).
- If S1[0] != S2[0], then you know that the maximum subsequence either does not start with S1[0] or does not start with S2[0]. Thus, you should determine the maximum subsequence of S1 and S2.substr(1), and the maximum subsequence of S2 and S1.substr(1). The answer has to be one of these.

Let's hack that up. It's in subseq1.cpp. (There's a main() that reads the two strings from the command line.

```
int LCS(string s1, string s2)
{
  int r1, r2;

  if (s1.size() == 0 || s2.size() == 0) return 0;
  if (s1[0] == s2[0]) return 1 + LCS(s1.substr(1), s2.substr(1));

  r1 = LCS(s1, s2.substr(1));
  r2 = LCS(s2, s1.substr(1));
  return (r1 > r2) ? r1 : r2;
}
```

It seems to work on our examples:

```
UNIX> g++ -o subseq1 subseq1.cpp
UNIX> subseq1 AbbbcccFyy FxxxyyyAc
3
UNIX> subseq1 AbbbcccFyy FxxxyyyAccc
4
UNIX> subseq1 900970994705376997373585670781133734891 4 940488836737907475451095492239929191 2221
```

However, it hangs on that last one. Why? Exponential blow-up of duplicate calls. Let's memoize. This is an easy memoization on the strings -- just concatenate them with a character that can't be in either string -- I'll choose a colon here. I'm going to turn the solution into a class so that storing the cache is easier. It's in subseq2.cpp:

```
class Subseq {
  public:
    map <string, int> cache;
    int LCS(string s1, string s2);
};

int Subseq::LCS(string s1, string s2)
{
  string key;
  int r1, r2;

  if (s1.size() == 0 || s2.size() == 0) return 0;

  key = s1 + ":";
```

```
  key += s2;
  if (cache.find(key) != cache.end()) return cache[key];

  if (s1[0] == s2[0]) {
   cache[key] = 1 + LCS(s1.substr(1), s2.substr(1));
  } else {
    r1 = LCS(s1, s2.substr(1));
    r2 = LCS(s2, s1.substr(1));
    cache[key] = (r1 > r2) ? r1 : r2;
  }
  return cache[key];
}
```

This works fine on our examples:

```
UNIX> g++ -o subseq2 subseq2.cpp
UNIX> subseq2 AbbbcccFyy FxxxyyyAc
3
UNIX> subseq2 AbbbcccFyy FxxxyyyAccc
4
UNIX> subseq2 900970994705376997373585670781133734891 940488836737907475451095492239929191 2221
15
UNIX>
```

However, this solution should make you feel uneasy. It makes me feel uneasy. The reason is that each call to LCS makes a copy of its arguments, and each call to substr() creates a new string which is just one character smaller than the previous string. That's a lot of memory. To hammer home this point, the file sub-big.txt has two 3000-character strings. When we call subseq2 on it, it hangs because it is making so many copies of large strings:

```
UNIX> subseq2 `cat sub-big.txt`
```

Think about it -- each substring is a suffix of the previous one, so for S1 and S2 there are roughly 3000 suffixes. That means potentially 3000 * 3000 calls to LCS(), each time creating strings of size up to 3000. That's a huge amount of time and memory.

Since we are only using suffixes of S1 and S2, we can represent them with indices to their first characters, and we call LCS() on these indices rather than on the strings. I've put that code into subseq3.cpp. It doesn't memoize.

```
class Subseq {
  public:
    string s1, s2;
    int LCS(int i1, int i2);
};

int Subseq::LCS(int i1, int i2)
{
  int r1, r2;

  if (s1.size() == i1 || s2.size() == i2) return 0;

  if (s1[i1] == s2[i2]) {
    return 1 + LCS(i1+1, i2+1);
  } else {
    r1 = LCS(i1, i2+1);
    r2 = LCS(i1+1, i2);
    return (r1 > r2) ? r1 : r2;
  }
}
```

It works, although since it doesn't memoize, it hangs on the long input.

```
UNIX> subseq3 AbbbcccFyy FxxxyyyAc
3
UNIX> subseq3 AbbbcccFyy FxxxyyyAccc
4
UNIX> subseq3 900970994705376997373585670781133734891 940488836737907475451095492239929191 2221
```

We memoize in subseq4.cpp. Our cache is on the indices, and is thus of size s1.size() * s2.size():

```
class Subseq {
  public:
    string s1, s2;
```

```
    vector < vector <int> > cache;
    int LCS(int i1, int i2);
};

int Subseq::LCS(int i1, int i2)
{
  int r1, r2, i;

  if (s1.size() == i1 || s2.size() == i2) return 0;

  if (cache.size() == 0) {
    cache.resize(s1.size());
    for (i = 0; i < s1.size(); i++) cache[i].resize(s2.size(), -1);
  }

  if (cache[i1][i2] != -1) return cache[i1][i2];

  if (s1[i1] == s2[i2]) {
   cache[i1][i2] = 1 + LCS(i1+1, i2+1);
  } else {
    r1 = LCS(i1, i2+1);
    r2 = LCS(i1+1, i2);
    cache[i1][i2] = (r1 > r2) ? r1 : r2;
  }
  return cache[i1][i2];
}
```

Now this version is fast, and it even works on the huge input in under a second without compiler optimization!

```
UNIX> g++ -o subseq4 subseq4.cpp
UNIX> subseq4 AbbbcccFyy FxxxyyyAc
3
UNIX> subseq4 AbbbcccFyy FxxxyyyAccc
4
UNIX> subseq4 900970994705376997373585670781133734891 4404888367379074754510954922399291912221
15
UNIX> time subseq4 `cat sub-big.txt`
891
0.930u 0.040s 0:00.96 101.0%  0+0k 16+0io 0pf+0w
UNIX>
```

It's faster with optimization:

```
UNIX> g++ -O3 -o subseq4 subseq4.cpp
UNIX> time subseq4 `cat sub-big.txt`
891
0.310u 0.040s 0:00.33 106.0%  0+0k 0+0io 0pf+0w
UNIX>
```

We can perform step #3 on this by realizing that you perform the recursion on larger indices, so you build the cache from high to low. This is in subseq5.cpp:

```
int Subseq::LCS()
{
  int r1, r2, i, i1, i2;

  cache.resize(s1.size()+1);
  for (i = 0; i < cache.size(); i++) cache[i].resize(s2.size()+1, -1);

  for (i = 0; i <= s1.size(); i++) cache[i][s2.size()] = 0;
  for (i = 0; i <= s2.size(); i++) cache[s1.size()][i] = 0;

  for (i1 = s1.size()-1; i1 >= 0; i1--) {
    for (i2 = s2.size()-1; i2 >= 0; i2--) {
      if (s1[i1] == s2[i2]) {
       cache[i1][i2] = 1 + cache[i1+1][i2+1];
      } else {
        r1 = cache[i1][i2+1];
        r2 = cache[i1+1][i2];
        cache[i1][i2] = (r1 > r2) ? r1 : r2;
      }
    }
  }
  return cache[0][0];
}
```

This is much faster than the previous code:

```
UNIX> time subseq5 `cat sub-big.txt `
891
0.052u 0.036s 0:00.09 88.8% 0+0k 40+0io 0pf+0w
UNIX>
```

Finally, we can perform step 4 on this by only keeping two rows, and performing the arithmetic on *i1* mod 2. Now, the cache only contains (2*s2.size()) entries. The code is in subseq6.cpp:

```
int Subseq::LCS()
{
  int r1, r2, i, i1, i2, index, other;

  cache.resize(2);
  for (i = 0; i < cache.size(); i++) cache[i].resize(s2.size()+1, 0);

  for (i1 = s1.size()-1; i1 >= 0; i1--) {
    index = i1%2;
    other = (i1+1)%2;
    for (i2 = s2.size()-1; i2 >= 0; i2--) {
      if (s1[i1] == s2[i2]) {
       cache[index][i2] = 1 + cache[other][i2+1];
      } else {
        r1 = cache[index][i2+1];
        r2 = cache[other][i2];
        cache[index][i2] = (r1 > r2) ? r1 : r2;
      }
    }
  }
  return cache[0][0];
}
```

It runs at the same speed as before, but uses much less memory:

```
UNIX> time subseq6 `cat sub-big.txt `
891
0.052u 0.000s 0:00.05 100.0%  0+0k 0+0io 0pf+0w
UNIX>
```

Here's a curiousity -- take a look at the timings and the output and see if you can figure out what's going on:

```
UNIX> wc sub-big.txt
   2    2 5353 sub-big.txt
UNIX> wc sub-big-2.txt
   2    2 5186 sub-big-2.txt
UNIX> time subseq4 `cat sub-big-2.txt`
2592
0.000u 0.036s 0:00.04 75.0% 0+0k 0+0io 0pf+0w
UNIX> time subseq5 `cat sub-big-2.txt`
2592
0.060u 0.020s 0:00.08 100.0%  0+0k 0+0io 0pf+0w
UNIX> time subseq6 `cat sub-big-2.txt`
2592
0.044u 0.004s 0:00.05 80.0% 0+0k 0+0io 0pf+0w
UNIX>
```

Why is this interesting? Well, sub-big-2.txt is nearly the same size as sub-big.txt. subseq5 and subseq6 take about the same amount of time to run on it, but subseq4 is *much* faster. Why would that be the case?

Take a look at the output -- the maximum subsequence size is 2592, pretty much half the size of the input file. You can confirm that the two strings are in fact the same:

```
UNIX> sort -u sub-big-2.txt | wc
      1       1    2593
UNIX>
```

Why would subseq4 be so much faster than the others -- well, because it simply does the one recursion at every step, ending after 2592 recursive calls. subseq5, on the other hand, builds that 2592x2592 cache without realizing that pretty much none of the cache is getting used. Same with subseq6, except it only stores the last two rows.

#4: ConvertibleStrings

This is a nice example for two reasons:

- It uses strings as keys for memoization, and for that reason, the cache has to be a map keyed on strings.
- It may be viewed as a graph algorithm that you can solve using Dijkstra's shortest path algorithm, or using topological sort.

Here is the writeup.

#5: PageNumbers

Lecture notes are here - this is a hard problem. The hard part is spotting the recursion.

#6: Counting apples

I don't go over this one in class any more, but it makes for good reading.

This is Dumitru's Intermediate problem, and it is really no harder than the subsequence problem above. The problem is as follows:

> *"You are given a table composed of rows x cols cells, each having a certain quantity of apples. You start from the upper-left corner. At each step you can go down or right one cell. Find the maximum number of apples you can collect."*

You can view this as a graph problem -- the "table" is a graph, and there are only edges going down and right. Your job is to find the maximum weight path through the graph, where the weight of a path is the sum of all the node weights on the path.

Unlike finding the minimum weight path, finding the maximum weight path is an "NP-Complete" problem -- its solution is exponential (or at least, that's the best we can do with current knowledge). However, if we simply view it as a dynamic program, we can solve it without worrying about its running time complexity. Spotting the recursion here is pretty easy. The maximum weight path to the cell at row *r* and column *c* is equal to the number apples in the cell, plus the maximum of the maximum weight path to the cell to the left, and the cell above the given cell. If we then find the maximum weight path to the lower right-hand cell, we will have found the maximum weight path through the graph.

The code is in apples1.cpp

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

typedef vector <int> IArray;

class Apple {
  public:
    int rows;
    int cols;
    vector <IArray> apples;
    int find_max(int r1, int c1);
};

int Apple::find_max(int r, int c)
{
  int a;
  int r1, r2;

  a = apples[r][c];
  if (r == 0 && c == 0) return a;
  if (r == 0) return a + find_max(r, c-1);
  if (c == 0) return a + find_max(r-1, c);
  r1 = find_max(r, c-1);
  r2 = find_max(r-1, c);
  return (r1 > r2) ? a+r1 : a+r2;
}
```

```
main(int argc, char **argv)
{
  int r, c;
  Apple a;

  if (argc != 3) {
    cerr << "usage: apples1 rows cols -- apples on standard input\n";
    exit(1);
  }

  a.rows = atoi(argv[1]);
  a.cols = atoi(argv[2]);
  a.apples.resize(a.rows);
  for (r = 0; r < a.rows; r++) a.apples[r].resize(a.cols);

  for (r = 0; r < a.rows; r++) {
    for (c = 0; c < a.cols; c++) {
      cin >> a.apples[r][c];
      if (cin.fail()) {
        cerr << "Not enough apples\n";
        exit(1);
      }
    }
  }
  cout << a.find_max(a.rows-1, a.cols-1) << endl;
}
```

We can see it working on some small examples:

```
UNIX> cat a1.txt
5 10
6 4
UNIX> apples1 2 2 < a1.txt
19
UNIX> cat a2.txt
18 32 88
03 85 29
64 89 88
UNIX> apples1 3 3 < a2.txt
312
UNIX> calc 18+32+85+89+88
312.000000
UNIX>
```

Let's try a bigger example:

```
UNIX> cat a3.txt
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20
UNIX> apples1 10 20 < a3.txt
390
UNIX>
```

Is that right? The best path will be to take the top row all the way to the right and then drop down. That path will have a weight of *(19*20)/2 + 20*10* = 390. Yes, that is right. Let's try it with two times the number of rows:

```
UNIX> cat a3.txt a3.txt | apples1 20 20
```

It hangs, so we must memoize. Again, quite straightforward: (In apples2.cpp):

```
typedef vector <int> IArray;
int Apple::find_max(int r, int c)
{
  int a;
  int r1, r2;
  int retval;
```

```
  if (cache[r][c] != -1) return cache[r][c];

  a = apples[r][c];
  if (r == 0 && c == 0) {
    retval = a;
  } else if (r == 0) {
    retval = a + find_max(r, c-1);
  } else if (c == 0) {
    retval = a + find_max(r-1, c);
  } else {
    r1 = find_max(r, c-1);
    r2 = find_max(r-1, c);
    if (r1 > r2) {
      retval =  a+r1;
    } else {
      retval = a+r2;
    }
  }
  cache[r][c] = retval;
  return retval;
}
```

```
UNIX> cat a3.txt a3.txt | apples2 20 20
590
UNIX> cat a3.txt a3.txt a3.txt a3.txt a3.txt a3.txt a3.txt a3.txt a3.txt a3.txt | apples2 100 20
2190
```

Nice. In apples3.cpp, we remove the recursion. We do so by starting at the beginning of the cache and filling in to the higher values:

```
int Apple::find_max()
{
  int r1, r2;
  int retval;
  int r, c;

  cache[0][0] = apples[0][0];
  for (r = 1; r < rows; r++) cache[r][0] = apples[r][0] + cache[r-1][0];
  for (c = 1; c < cols; c++) cache[0][c] = apples[0][c] + cache[0][c-1];

  for (r = 1; r < rows; r++) {
    for (c = 1; c < cols; c++) {
      r1 = cache[r][c-1];
      r2 = cache[r-1][c];
      if (r1 > r2) {
        cache[r][c] =  apples[r][c]+r1;
      } else {
        cache[r][c] = apples[r][c]+r2;
      }
    }
  }
  return cache[rows-1][cols-1];
}
```

As with the maximum subsequence problem, we can reduce the cache size to two rows. I won't do it here -- see if you can do it yourself!