CS1Bh Lecture Note 4

Dynamic Programming

This lecture examines a problem solving technique known as *dynamic programming*. It is frequently used when a straightforward recursive solution to a problem has exponential run time, due to repeated redundant computation. We already encountered such a difficulty two lectures ago (Note 2), with the initial implementation of the recursive algorithm for spelling correction. The straightforward technique used to improve that implementation was just to remember if the result of a particular call of a method had already been computed, using a table indexed by the different argument values. In general, this technique is known as *memoisation*, the table being known as a *memo*, and it is frequently used by optimising compilers for programming languages where recursion is the main form of repeated execution. Such compilers automate the process of inserting code for memoisation, allowing the programmer to write 'obvious' recursive solutions, without suffering run time penalties.

Although memoisation is a very useful technique, it can potentially require a large amount of space because of creating arrays big enough to cover all possible argument values. Dynamic programming involves the use of a memo storing past results, but also involves changing the order of computation, bringing the memo to the fore by systematically building up and recording sub-problem results, to provide a time- and space-efficient overall solution. This is best illustrated through some examples.

4.1 Fibonacci numbers

Consider the calculation of a Fibonacci number. Recall that the series has the following form: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$, being defined as follows:

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \end{aligned}$$

If we directly implement this function using recursion, we have the following:

```
class RecFib{
    public static int fib(int n){
        switch(n){
        case 0:  return 0;
        case 1:  return 1;
        default: return fib(n-1) + fib(n-2);
        }
    }
}
```

This clearly implements the Fibonacci series correctly, since it follows directly from the definition. However, because two recursive calls are made at each level, it has $O(2^n)$ run time, which is extremely expensive. Underlying this exponential time is the fact that there are very many duplicated calls with the same argument, and we want to avoid this overhead.

4.1.1 Memoisation solution

With basic memoisation, we create an array of values, each one corresponding to a possible call to the recursive method. Thus if we wish to calculate `fib(n)`, we will have an array of values $[0 \ldots n]$ representing the results of `fib(0)`, `fib(1)`, etc. This is implemented in the `MemoFib` class shown overleaf. In the earlier example of memoisation in Note 2, we used a zero value in a memo location to denote 'not computed yet'. Here we use a slightly cleaner approach: a separate boolean array `done[]` (initialised to all false) records whether the calculation of this value has been done or not. At the start of the recursive method, we first check if the value has been calculated already — if so, it is returned. If not, the new value is calculated, the `memo` and `done` arrays are updated, and the value is returned.

4.1.2 Dynamic programming solution

The approach used so far in the Fibonacci example is a 'top down' one. We start with a goal `fib(n)` and then work out what is required to calculate it. Looking at this another way, we start from where we would like to finish and work backwards, determining all the smaller sub-problems needed to calculate the final answer.

Another way of viewing the problem is 'bottom up'. If we have computed `fib(1)` and `fib(0)` we can compute `fib(2)`. Then, if we have `fib(2)` and `fib(1)` we can calculate `fib(3)`, etc. So, rather than starting at the top and working down, we can first solve the smaller sub-problems and work bottom-up towards our final goal. In the case of Fibonacci, there is a straightforward systematic approach to ordering the sub-problems. We can solve them in strictly increasing order: `fib(0)`, `fib(1)`, `fib(2)`, etc. Note that this 'just in time' strategy means that, for each new value to be computed, we will have just computed the two values needed to work it out. Given this evaluation order, we get a new implementation, as in the `ArrayFib` class shown overleaf.

This code can be further improved if we make a simple observation. Because each new value only requires the two previous values, we do not actually need to store any values computed earlier than these two. So, finally we get a simple dynamic programming solution, as in the `Fibonacci` class shown overleaf.

MemoFib class

```
class MemoFib{
    static boolean [] done;
    static int [] memo;

    public static int fib(int n){
        done = new boolean[n];
        memo = new int[n];
        return memofib(n);
    }
    static int memofib(int n){
        if (done[n])
            return memo[n];
        else {
            int result;
            switch(n){
            case 0: result = 0;
            case 1: result = 1;
            default:
                result = memofib(n-1) + memofib(n-2);
            }
            done[n] = true;
            memo[n] = result;
            return result;
        }
    }
}
```

ArrayFib class

```
class ArrayFib{
    public static int fib(int n){
        int [] F = new int[n];

        switch(n){
        case 0:  return 0;
        case 1:  return 1;
        default:
            F[0] = 0;
            F[1] = 1;
            for (int i = 2; i < n; i++){
                F[i] = F[i - 1] + F[i - 2];
            }
            return F[n - 1] + F[n - 2];
        }
    }
}
```

Fibonacci class

```
class Fibonacci{
    public static int fib(int n){
        int prev = 0 ;
        int fib  = 1 ;
        for(int i = 1 ; i < n; i++) {
            int next = prev + fib ;
            prev     = fib  ;
            fib      = next ;
        }
        return fib; // fib = fib(n)
    }
}
```

4.2 Recurrence relations

As another, slightly more complex, example, we consider the use of dynamic programming to reduce the time to compute a solution to a recurrence relation. The following recurrence relation arises in the analysis of the expected run time of the *quicksort* sorting algorithm on an array of size n:

$$C(n) = \frac{2}{n}\sum_{i=0}^{n-1} C(i) + n$$

with $C(0) = 1$ (the analysis itself is CS3 material, so no need to worry about how these equations arise). To actually evaluate $C(n)$, an obvious recursive approach is:

```
class Recursive{
    public static double eval( int n){
        if (n==0) {
            return 1.0;
        }else{
            double sum = 0.0;
            for (int i = 0; i < n; i++){
                sum = sum + eval(i);
            }
            return 2.0 * sum/n + n;
        }
    }
}
```

Once again, there are multiple redundant computations of the $C(i)$, leading to an exponential run time. In looking for a dynamic programming approach, we note that this example is a little more complex than Fibonacci. To calculate a value, say $C(n)$, *all* values $C(0)$ to $C(n-1)$ are required. However, computing values in strictly increasing order of n is still appropriate here, giving:

```
class Dynamic{
    public static double eval(int n){
        double [] c = new double [n+1];
        c[0] = 1.0;
        for (int i = 1; i <= n; i++){
            double sum = 0.0;
            for (int j = 0; j < i; j++){
                sum = sum + c[j];
            }
            c[i] = 2.0 * sum/i + i;
        }
        return c[n];
    }
}
```

Thus, the problem can be solved in $O(n^2)$ time with $O(n)$ space. In fact, it can be improved to an even better $O(n)$ time algorithm, by keeping track of partial sums.

4.3 Shortest paths

As a final example, we consider a dynamic programming approach to find the shortest path between two points. For example, if we have a collection of towns, and details of the roads between the towns, in particular their lengths, the problem is to find a shortest route between any two towns. This solution technique was first described by Floyd and Warshall in 1962. The general form of the algorithm works for any *graph*, that is any collection of vertices connected by edges (alternative terminology: nodes connected by arcs). Here, for simplicity, we just use the language of towns connected by roads. Each road will have a length associated with it — this is a special-case instance here of associating a *weight* with each edge of a graph.

The algorithm can be considered as generating a sequence of virtual 'road maps', M_k, for $0 \leq k \leq n$, where n is the number of towns. Each map contains the same set of towns, which we denote by $\{t_1, \ldots, t_n\}$. However, the collection of roads in the maps evolves: map M_k has a virtual 'road' between towns t_i and t_j if and only if there is a sequence of real roads between t_i and t_j with any intermediate towns drawn *only* from $\{t_1, \ldots, t_k\}$. The map M_0 is just the real road map itself. Then, for example, map M_1 contains all the real roads, plus extra 'roads' representing any two-road routes between towns that go only via town t_1. When the algorithm is finished, the length of each 'road' in M_n indicates the shortest route between the towns it connects. If two towns are not connected by a 'road' in M_n, it means that there is no route between them.

Dynamic programming proceeds as follows. We begin with the supplied road map as M_0. There is a simple rule for constructing each M_k, for $1 \leq k \leq n$, from M_{k-1}:

> In map M_k, there is a road between towns t_i and t_j if either (a) there is a road in map M_{k-1}, or (b) there are roads between towns t_i and t_k, and between towns t_k and t_j, in map M_{k-1}. The length of this road is the smaller of the direct route (if it exists in M_{k-1}) and the indirect route (if it exists in M_{k-1}).

The final solution can then be read off from M_n.

We can now give a Java implementation of the dynamic programming algorithm. In it, a map is represented by an $n \times n$ array of integers. If there is a road between town i and town j, then its length is stored in element `[i,j]` and, by symmetry, element `[j,i]`. In the special case where $i = j$, zero is stored in element `[i,i]`. In all other cases, the value `Integer.MAX_VALUE` is stored, denoting a 'road of infinite length'.

```
class Floyd {
   private int n;
   private int M[][];

   public Floyd (int towns, int map[][]) {
      n = towns;
      M = new int[n][n];

      // Form initial map M0 in M
      for (int i=0; i<n; i++)
         for (int j=0; j<n; j++)
            M[i][j] = map[i][j];

      // Successively form maps M1, ..., Mn in M
      for (int k=0; k<n; k++)
         for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
               if (M[i][k]+M[k][j] < M[i][j])
                  M[i][j] = M[i][k]+M[k][j];
   }

   public int shortest (int town1, int town2) {
      return M[town1][town2];
   }
}
```

Note that the implementation of the algorithm is contained in the constructor method. We have done some optimisation in terms of storage space, since a single array `M` is used, rather than separate arrays for each of $M_0, M_1, \ldots, M_n$. It is fairly obvious to see from the construction rule opposite that only the previous M_{k-1} is needed when computing M_k. However, it is also safe go one step further, by relying on the observation that the (i, k) and (k, j) lengths will always be the same in both M_{k-1} and M_k (why?).

Gordon Brebner.
Javier Esparza, January 24th, 2003.