



Dynamic programming

Javier Esparza

Computer Science
School of Informatics
The University of Edinburgh

Dynamic programming

- A technique to improve the efficiency of the divide-and-conquer approach
 - Applicable when problems share subproblems
 - First Idea: save the solution to subproblems in tables
 - Second Idea: compute the recursive calls 'bottom-up'
-

Fibonacci Numbers

$$fib(0) = 0$$

$$fib(1) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2)$$

```
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);
        }
    }
}
```

Naive memoisation

- Create an array of values for all possible calls of the recursive procedure
 - In the Fibonacci example: in order to compute `fib(n)`, create an array `[0..n]` for `fib(0)`, `fib(1)`, ...
 - Before computing a subproblem, check if it has already been computed
-

```
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;
    }
}
```

Dynamic Programming: Computing bottom-up

- First solve the smaller sub-problems and work bottom-up towards our final goal.
- Advantage: no stack required, no need to store local variables

```
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];
        }
    }
}
```

Improving the solution

```
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)
    }
}
```

Recurrence relations

- Given a number n , compute the value of $C(n)$, defined as

$$C(n) = \begin{cases} 1 & n = 0 \\ \frac{2}{n} \sum_{i=0}^{n-1} C(i) + n & n \geq 1 \end{cases}$$

- Computing values in strictly increasing order of n is still appropriate, but
 - to calculate $C(n)$, *all* values $C(0)$ to $C(n - 1)$ are required.
-

```
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];  
    }  
}
```

All-pairs Shortest paths

- Given: a collection of towns, a map of roads between the towns
Compute: for each pair of towns, the length of the shortest route between them.
 - Divide-and-conquer strategy:
Say that a route crosses a town if it is one of the *intermediate* towns in the route.
Let a, b, c be towns. A shortest route from a to b either
 - does not cross c , or
 - it is the concatenation of shortest routes from a to c and from b to c which do not cross c .
-

Pseudocode

```
int shortestRoute(a, b, I) {
  if (I is empty) return distance(a,b);
  else {
    choose c from I;
    p1 = shortestRoute(a, b, I\{c});
    p2 = shortestRoute(a, c, I\{c});
    p3 = shortestRoute(c, b, I\{c});
    return( min(p1, p2+p3) )
  }
}
```

Floyd-Warshall algorithm

- Invented in 1962
 - Applies dynamic programming to divide-and-conquer solution
 - Based on arbitrary ordering $\{t_1, \dots, t_n\}$ of the towns
 - Computes shortest routes matrices M_i , $0 \leq i \leq n$, where M_i has $\{t_1, \dots, t_i\}$ as intermediate towns (so in M_0 there are no intermediate towns)
 - Computes bottom up: M_0, M_1, \dots, M_n
 - Complexity $\Theta(n^3)$
 - Only the previous matrix would need to be stored, but even that can be avoided!
-

```
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];  
}
```