CRICOS PROVIDER 00123M

Dynamic Programming 1

getting things done faster by remembering stuff

http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg

adelaide.edu.au

seek LIGHT

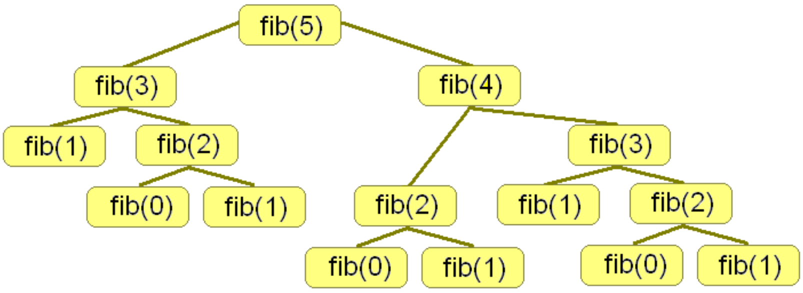
Non-dynamic Programming

- Many problems have elegant, but **inefficient**, recursive solutions.
- Example - Generating the nth Fibonacci Number….
  - See the call-tree for `fibonacci(5)`
  - Measure the performance..

PSSD - Dynamic Programming

Slide 2

The call tree



- How many times is fibonacci(2) called in the tree above?

It's good to remember

- The call-tree for fibonacci has many <u>overlapping</u> calls
- We can avoid these by remembering the result of the first call.
  - When that same call is made again - just return the remembered result.
  - This can save a lot of time
  - It prunes **big** branches off the call-trees.

Remembering-Fibonacci

```
private static HashMap store = new HashMap();

    public static int fibonacci(int n) {
        if (n <= 1)
            return 1;
        else if (store.containsKey(n)) {
            return (Integer) store.get(n);
        }else{
            int result =  fibonacci(n-1) +
                          fibonacci(n-2);
            store.put(n,result);
            return result;
        }
    }
```


Memory Functions

- The last program is an example of a Memory-Function.
- Memory functions store the results of previous calculations for when they are needed again.
- Memory functions are a special case of Dynamic Programming.

Dynamic Programming

- Dynamic Programming systematically stores the results of previous calculations to increase the efficiency of a solution.
- Dynamic Programming Requires
  - A Recurrence Relation - with boundary-conditions
    - In other words, that there is a recursive solution
  - Overlapping solutions to sub-problems that can be stored for later use.

Another Example - Counting Change

- From Abeson and Sussman (Structure and Interpretation of Computer Programs)
- Problem: given a set of coins (5c, 10c, 20c, 50c, 100c, 200c) how many ways can we change a given amount of money e.g. $2.10?
- A solution can be defined recursively….

Counting Change - Solution

- The following statement is true:
- The number of ways to count an amount a using n kinds of coins equals:
  - The number of ways to change amount a using all but the first kind of coin **plus.**
  - The number of ways to change amount a-d using all n kinds of coins, where d is the denomination of the first coin.
- That is the recurrence relation… now the boundary conditions.
  - a is exactly 0c, there is exactly one way to change 0c.
  - a < 0c, there are zero ways to change <0c.
  - n is 0, there are zero ways to change any amount of money.

Counting Change Code

```
private static int [] denoms = {5,10,20,50,100,200};
 public static int countChange(int amount){
      return countChange(amount,0);
 }

public static int countChange(int amount, int denomPointer){
  if (amount == 0){
      return 1;
  }else if ((amount < 0) || (denomPointer >= denoms.length)){
      return 0;
  }else{
      return countChange(amount - denoms[denomPointer],
                          denomPointer) +
             countChange(amount, ++denomPointer);
  }
}
```

DP for Counting Change

- Problem: given a set of coins (5c, 10c, 20c, 50c, 100c, 200c) how many ways can we change a given amount of money e.g. $2.10?
  - We saw a recursive solution last week
- As with fibonacci we can increase the efficiency by remembering previous calls
  - However, this time each call has two parameters.

Counting Change Memory Function

```
private static int [] denoms = {5,10,20,50,100,200};
private static HashMap store = new HashMap();
. . .
public static int countChange(int amount, int denomPointer){
   String currParams = amount + " " + denomPointer;
   if (amount == 0){
      return 1;
   }else if ((amount < 0) || (denomPointer >= denoms.length)){
      return 0;
   }else if(store.containsKey(currParams)){
      return (Integer) store.get(currParams);
   }else{
      int result = countChange(amount - denoms[denomPointer],
                              denomPointer) +
                   countChange(amount, ++denomPointer);
      store.put(currParams,result);
      return result;
   }
}
```

Going Forwards

- Memory functions work top down
  - They start at the desired solution and work back toward the boundary conditions.
- Many dynamic programming solutions work from the bottom-up.
  - They start at the boundary conditions and work forwards toward the desired solution.
- We remember a table of solutions to sub-problems as we go
  - For this reason dynamic programming is sometimes called tabulation.

A Fibonacci Table

- One dimensional table for a single sequence of numbers.

n	Fib(n)
0	1
1	1
2	2
3	3
4	5
5	8
6	13
7	21
8	34
9	55

Tabulating fibonacci

```
public static int fibonacci(int n)  {
  int table[] = new int[n+1];
  table[1] = table[0] = 1;
  for(int i = 2; i <= n; i++){
      table[i] = table[i-1] + table[i-2];
  }
  return table[n];
}
```

- Can you think of a further optimisation?

A Counting-change table

- So, for example, there is one way to change 40c with 20c, 50c, 100c and 200c coins, but nine ways if we are allowed to use 5c and 10c coins too.

Coin type availible

Amount	200	100, 200	50, 100, 200	20, 50, 100, 200	10, 20, 50, 100, 200	5, 10, 20, 50, 100, 200
0	1	1	1	1	1	1
5	0	0	0	0	0	1
10	0	0	0	0	1	2
15	0	0	0	0	0	2
20	0	0	0	1	2	4
25	0	0	0	0	0	4
30	0	0	0	0	2	6
35	0	0	0	0	0	6
40	0	0	0	1	3	9

Tabulating Counting Change

```
private static int [] denoms = {200,100,50,20,10,5};

int table [] [] = new int[(amount/5)+1][denoms.length];
.. more initialisation here ..

for(int a = 5; a <= amount; a+=5){
   for(int c=0; c< denoms.length; c++){
      int lessCoinsCount = 0;
      if (c > 0){
         lessCoinsCount = table[a/5][c-1];
      }
      int lessAmountCount = 0;
      int coinVal = denoms[c];
      if ((a - coinVal) >= 0){
          lessAmountCount = table[(a-coinVal)/5][c];
      }
      table[a/5][c] = lessCoinsCount + lessAmountCount;
   }
 }
```


Optimisations on your Optimisations

- Sometimes, you don't have to generate the whole table.
- Often, you don't have to remember everything you have generated so far
  - You can forget some of the "old" values.
  - Very problem-dependent.

When to use Dynamic Programming

- When brute-force won't do, and....
- When there's overlapping sub-problems in the recurrence-relation, and...
- It is practical to save the solutions to the required sub-problems.
- Examples:
  - Warshall's algorithm, Floyd's Algorithm, Knapsack problem, binomial coefficients..... and many many more...