Algorithmics

Dynamic programming

Vicente García Díaz – garciavicente@uniovi.es

University of Oviedo, 2016

Table of contents

1. Basic concepts
2. Examples of use
    1. Fibonacci series
    2. Combinations
    3. The knapsack problem (0/1)
    4. The problem of the change
    5. Cheaper travel on the river

Basic concepts

Problems with Divide and Conquer

- The idea **was** to divide the original problem in subproblems and combine them to solve the original problem
  - Drawbacks:
    - Not suitable when the **number** of subproblems is very **high** and then the complexity is not polynomial
    - Not suitable when generating a number of subproblems that are **repeated** and therefore are solved several times in the same execution

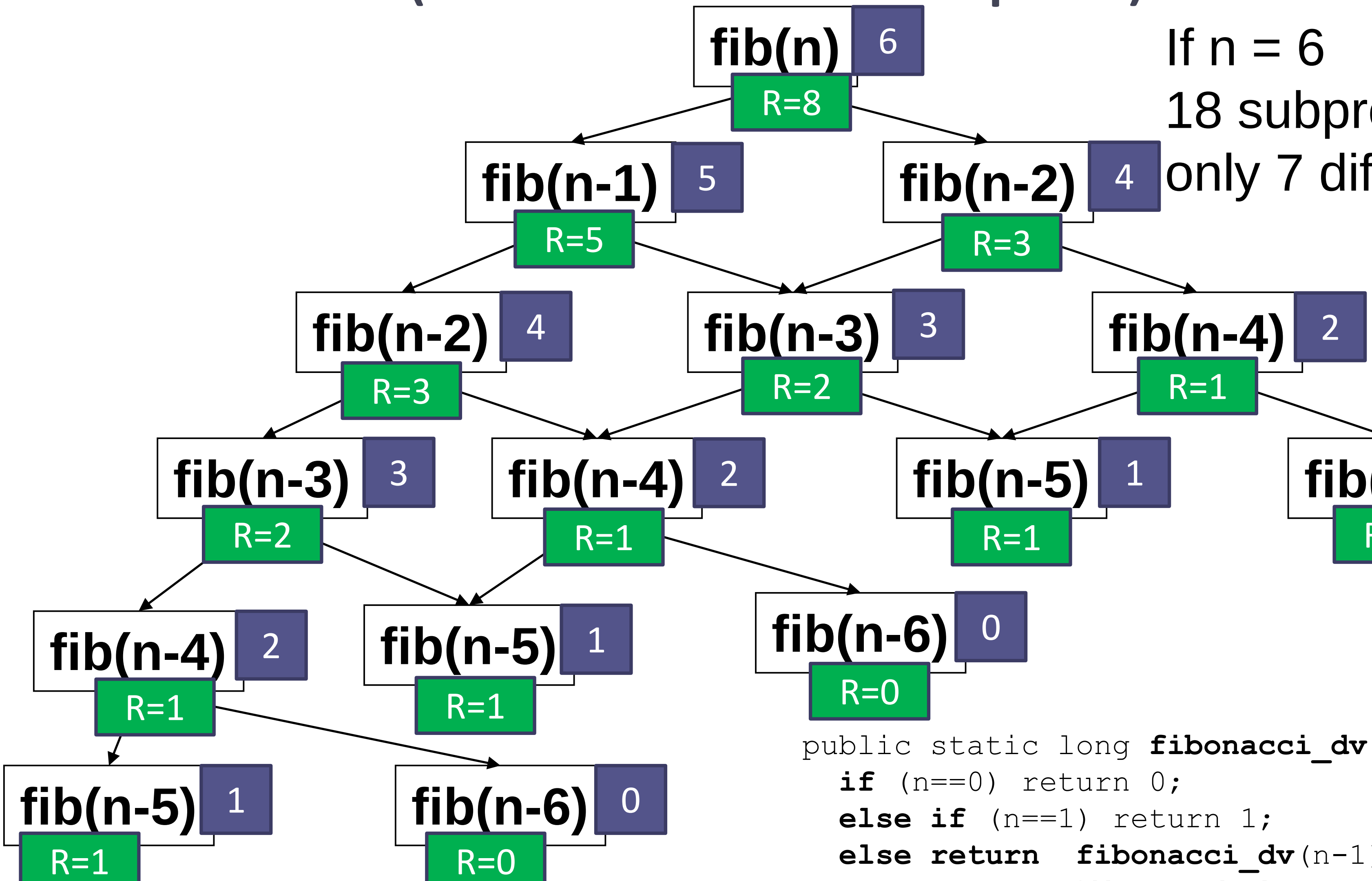
Dynamic Programming

- The idea **is** to divide the original problem in subprobems and combine them to solve the original problem
  - Improvement:
    - When the number of different problems is polynomial, we can solve each subproblem once and store the solution for later use
    - The idea is to **avoid calculating the same subproblem twice**, usually maintaining a table of known results

Pseudocode (Divide and Conquer)

```
public static long fibonacci_dv(int n){
  if (n==0) return 0;
  else if (n==1) return 1;
  else return fibonacci_dv(n-1) + fibonacci_dv(n-2);
}
```

Call tree (Divide and Conquer)



If n = 6
18 subproblems
only 7 differ

```
public static long fibonacci_dv(long n){
  if (n==0) return 0;
  else if (n==1) return 1;
  else return  fibonacci_dv(n-1) +
               fibonacci_dv(n-2);}
```

Pseudocode

```
public static int fibonacci_pd(int n){
  int[] f = new int[n+1]; //0, 1, 2, 3, 4, 5, 6

  f[0]= 0; f[1]= 1; //we know it
  for (int i= 2; i<n+1; i++)
      f[i]= f[i-1]+f[i-2];
  return f[n];
}
```

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**

fib2()?

Divide and Conquer vs Dynamic Programming

- Divide and Conquer
  - Descending technique (progressive refinement)
  - We start with the whole problem
    - We divide it into subproblems
- Dynamic Programming
  - Ascending technique
  - We start with the subproblems
    - We compose solutions until reaching the solution for the whole initial problem

Examples of use

Fibonacci series

- Goal
  - Calculate the Fibonacci function (0,1,1,2,3,5,8,13,21,34,55,89,...)

$$
\begin{cases}
f = 0 & \text{if } n = 0 \\
f = 1 & \text{if } n = 1 \\
f = f(n-1) + f(n-2) & \text{if } n>1
\end{cases}
$$

- Complexity comparison
  - Divide & Conquer -> $O(1.6^n)$
  - Dynamic Programming -> $O(n)$

$$\frac{50!}{6!(50-6)!} = \frac{50!}{6!(44!)}$$
$$= \frac{50\times49\times48\times47\times46\times45}{6\times5\times4\times3\times2}$$
$$= 15{,}890{,}700$$

Combinations

- In mathematics a **combination** is a way of selecting several things out of a larger group, where (unlike *permutations*) order does not matter

- In smaller cases it is possible to count the number of combinations
  - For example given **three fruit**, say an apple, orange and pear, **there are three combinations of two** that can be drawn from this set: an apple and a pear; an apple and an orange; or a pear and an orange

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$ If 0 < k < n $$\binom{n}{0} = \binom{n}{n} = 1$$

Fuente: http://en.wikipedia.org/wiki/Combination

Examples of use

Combinations

$$\frac{50!}{6!(50-6)!} = \frac{50!}{6!(44!)}$$
$$= \frac{50 \times 49 \times 48 \times 47 \times 46 \times 45}{6 \times 5 \times 4 \times 3 \times 2}$$
$$= 15{,}890{,}700$$

A possible solution with DP

	0	1	2	3	...	k-1	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
...	...	...	...	...	...		
...	...	...	...	...	...	...	
n-1						C(n-1,k-1) +	C(n-1,k)
n							C(n,k)

- Complexity?
  - O(n*k)

combinationsDivideAndConquer()?

Goal

- **n** objects and a backpack to transport them
- Each object i= 1,2, ...n has a weight of w_i and a value of v_i
- The backpack can carry a total weight not exceeding W
- **The idea is to maximize the value of objects, while respecting the weight limitation**
- Objects **cannot be fragmented**; we take an entire object or we leave it

Data for a specific problem

- Number of objects: `n=3`
- Weight limit of the backpack: `W=10`

Object	1	2	3
w_i	6	5	5
v_i	8	5	5

Strategy (I)

- Table **V**
  - Rows: *i* objects
  - Columns: maximum weight of the backpack

- **V[i,j]** → maximum value of the items we would carry
  - We include only until object **i** for each case
  - The weight limit is **j**

- Solution to our problem **V[n,W]** → **V[3,10]**

Strategy (II)

- Function that calculates values in the matrix:

$$V(i,j)=\begin{cases} -\infty \text{ if } j<0 \\ 0 \text{ if } i=0 \,\&\, j\geq 0 \\ \max(V(i\text{-}1,j), V(i\text{-}1,j\text{-}w_i)+v_i) \text{ other case} \end{cases}$$

- **i**, is the number of objects we try to put in the backpack
- **j**, is the maximum weight of the backpack

Table values

- For *n*=3 (objects), *W*=10 (maximum load)

Maximum weights

Objects

	0	1	2	3	4	5	6	7	8	9	10
1											
2											
3											

V[i,j]

V[n,W]

Table values. Cell out

- For n=3 (objects), W=10 (maximum load)

Maximum weights

Objects

i \ j	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0								
2	0	0	0								
3	0	0									

Obj.	1	2	3
w_i	6	5	5
v_i	8	5	5

V[i,j] Max(V(i-1,j),V(i-1,j-w_i)+v_i)

Table values. Cell in

- For n=3 (objects), W=10 (maximum load)

Maximum weights

Objects

i \ j	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0					
2	0	0	0	0	0	5					
3	0	0	0	0	0						

Obj.	1	2	3
w_i	6	5	5
v_i	8	5	5

V[i,j] $\text{Max}(V(i-1,j),V(i-1,j-w_i)+v_i)$

The problem of the change (I)

- Design an algorithm to pay a certain amount of money, using the fewest possible coins
- Example:
  - We have to pay €2.89
  - Solution: 1 coin of €2, 1 coin of 50 cents, 1 coin of 20 cents, 1 coin of 10 cents, 1 coin of 5 cents, 2 coins of 2 cents
    - → Optimal solution ☺
- Greedy heuristic: Take the coin of the biggest possible value without exceeding what we have left to return → It does not work for all the cases

The problem of the change (II)

- **Can you do it** *(using the dynamic programming technique)*??

- Key differences with “Knapsack problem”
  - Main methods:
    - `float knapsack01(int maxWeight, float[]benefits, int[]weights)`
    - `int change(int amount, int[]coins)`
  - Now we don’t look for the greatest value, we look for the smallest
  - We need to sort the coins from the smallest to the biggest (the first one should have a value of 1)

`change()?`

Cheaper travel on the river

Cheaper travel on a trip

- We are in a river that has `n` docks
- In each of them you can rent a boat for going to any other dock downstream (it is impossible to go upstream)
- There is a fee table that indicates the cost of traveling from dock `i` to to dock `j` (`i<j`)
- It may happen that a trip from `i` to `j` is more expensive than a succession of shorter trips, in which case we would take a boat from `i` to a dock `k` first and a second boat to go from `k` to `j`
- Our problem is to design an efficient algorithm to determine the minimum cost for each pair of docks `i,j (i<j)`
    - Indicate, in function of `n`, the time used by the algorithm **`riverTravel()`**?

Try the quizz

https://www.goconqr.com/en-US/p/4759638

Bibliography

JUAN RAMÓN PÉREZ PÉREZ; (2008) ***Introducción al diseño y análisis de algoritmos en Java***. Issue 50. ISBN: 8469105957, 9788469105955 (Spanish)