Web Algorithms

Eng. Fabio Persia, PhD

Algorithmic techniques: dynamic programming (Part 1)

Characteristics

- Like divide-and-conquer paradigm, break up a problem into smaller subproblems, solve recursively each subproblem, and combine solutions of subproblems to form solution to original problem.

- Easy-to-compute recurrence that allows one to determine the solution to a subproblem from the solution to smaller subproblems.

- Differently from divide-and-conquer, subproblems are not independent, but overlap, that is during the decomposizions same subproblems occur frequently

- Idea: each subproblems is solved only once, thus reducing time complexity

- Differently from divide-and-conquer, usually bottom-up approach instead of top-down, that is starting from smaller subproblems solving in progress bigger ones, till the initial problem

Applications

Areas

- Computer science: theory, graphics, AI, compilers, systems,…
- Bioinformatics
- Control theory
- Information theory
- Operations research
- …

Some famous dynamic programming algorithms

- Unix diff for comparing two files
- genetic sequence alignment (Smith-Waterman)
- shortest paths in networks (Bellman-Ford)
- …

Let's give a closer look …

- The *Divide-and-Conquer* paradigm is based on the decomposition of problems in smaller subproblems:
  - recursively solves subproblems
  - combines the solutions of the subproblems to determine the solution of the initial problem

- If a problem of size n is decomposed in k subproblems of sizes $n_1,\ldots,n_k<n$, respectively, then the time complexity can be expressed by the recurrence

$$T(n) = T(n_1)+\ldots+T(n_k)+C(n),$$

  with $C(n)$ time of combining the k subsolutions

- The recurrence can be solved with different methods, as for instance resorting on the famous *Master Theorem*

- A classical example of application of *Divide-and-Conquer* is the computation of *Fibonacci* numbers

- The algorithm comes directly from the recursive definition of such numbers:
  - base case ($n \le 2$): $F(1)=F(2)=1$
  - inductive case ($n>2$): $F(n)=F(n-1)+F(n-2), n$

- Let's see the resulting algorithm….

Algorithm Fibonacci(n)

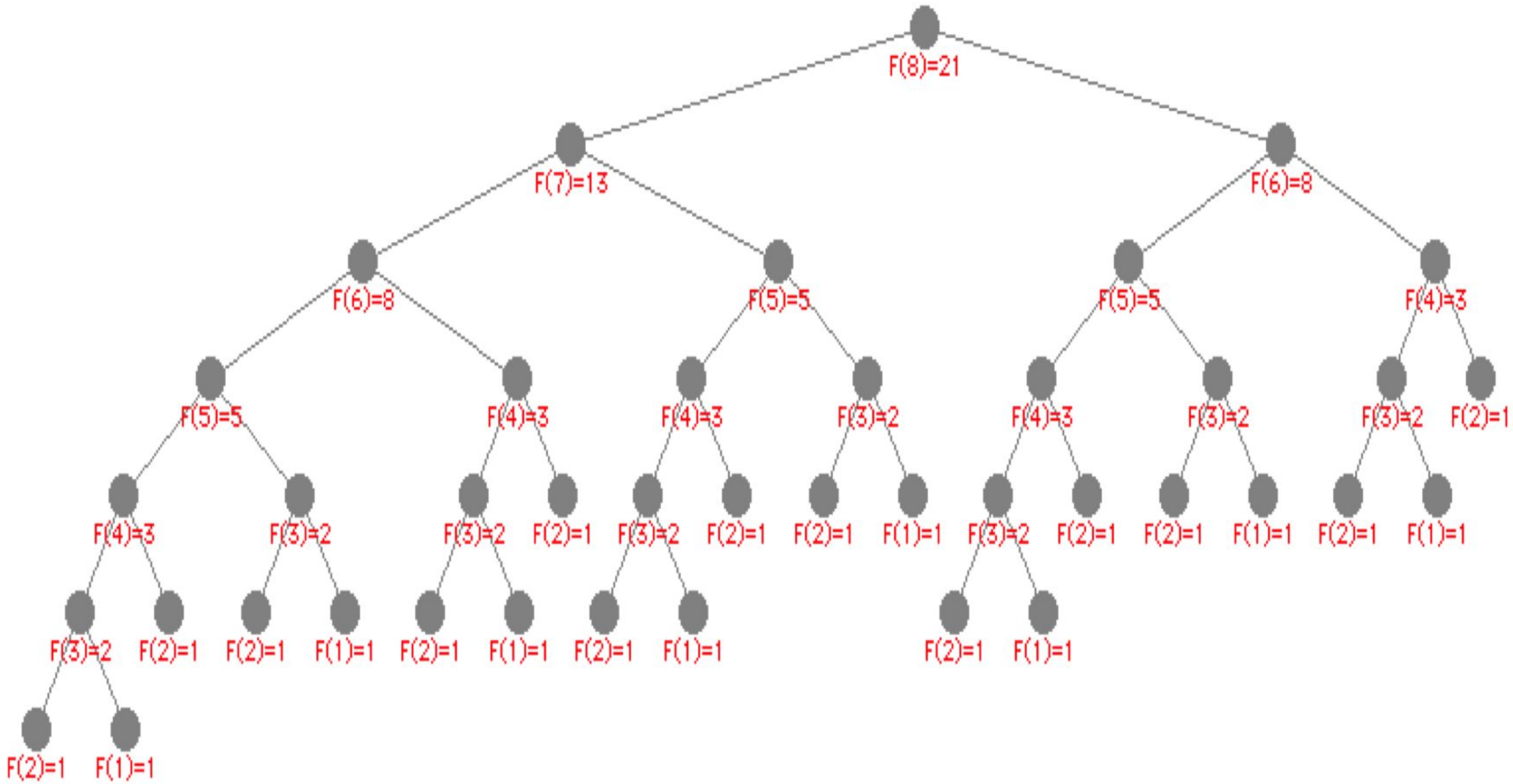
Begin
if (n=1) or (n=2) return 1
else return Fibonacci(n-1)+Fibonacci(n-2)
End

Time Complexity:

- $T(n) = T(n-1) + T(n-2) + \Theta(1)$, that gives

$$T(n)=O(2^n)$$

Let's have a look to the tree of the recursive calls:



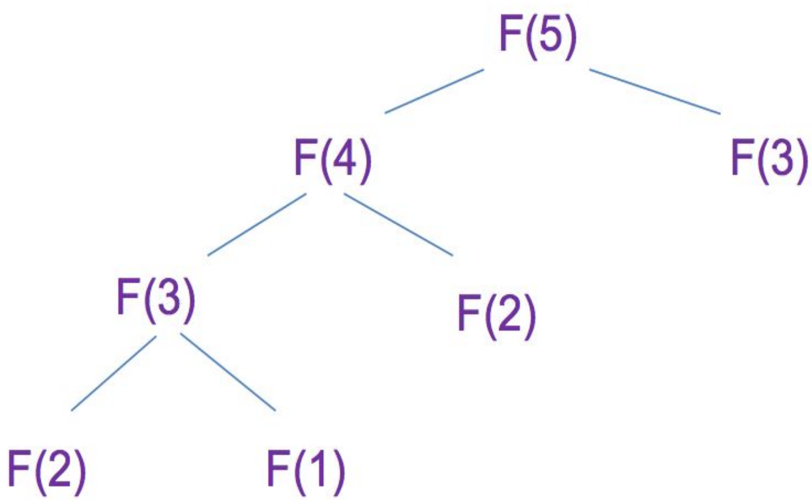
Remark:

- inefficient: same subproblems are solved again many times
- dynamic programming: store the solution of every subproblem in a table or array, thus avoiding solving it again
- in the resulting algorithm, *F* is an external global array visible to all the recursive calls:

Algorithm Fibonacci2(n)

```
Begin
    if (n=1) or (n=2) let F[n]=1 and return F[n]
        else
          if F[n] has been already assigned return F[n]
              else
                  let F[n]=Fibonacci2(n-1)+Fibonacci2(n-2)
                  return F[n]
End
```

Let us see the new tree of the recursive calls for $n=5$



1	1	2	3	5

1	1	2	3	

1	1	2		

1	1			

	1			

nstant


```
Algorithm Fibonacci3(n)

Begin
    F[1]=1
    F[2]=1
      For i=3 to n
          F[i]=F[i-1]+F[i-2]
    return F[n]
End
```

Example n=5

1				

1	1			

1	1	2		

1	1	2	3	

1	1	2	3	5

Performance comparison

Actual running time on different platforms:

	Fibonacci(58)	*Fibonacci2(58)*
Pentium IV 1700MHz	***15820 s (≃ 4 hours)***	***0.7 millionths of s***
Pentium III 450 MHz	***43518 s (≃ 12 hours)***	***2.4 millionths of s***
PowerPC G4 500 MHz	***58321 s (≃ 16 hours)***	***2.8 millionths of s***

Summarizing…

In dynamic programming:

- the initial problem can be recursively decomposed in subproblems
- same subproblems occur many times and are solved once
- the solution of a subproblem can be obtained combining the ones of smaller subproblems

Two possible implementations:

- *top-down* with table annotation (memoization)
- *bottom-up*

Top-down versus bottom-up

- Top-down
  - exploits table annotation
  - pros: solves only the strictly needed subproblems
  - cons: overhead of recursive chain of calls

- Bottom-up
  - is the typical choice in dynamic programming
  - cons: solves also unnecessary subproblems
  - pros: it is anyway generally more efficient because it eliminates the weight of recursion, which affects more the overall performance

Divide-and-Conquer versus dynamic progr.

Divide-and-Conquer:

- Recursive technique
- Top-down approach (problems split in subproblems)
- Profitable when subproblems are *indipendent* (i.e. different)
- Otherwise, same subproblems solved multiple times

Dynamic programming:

- Iterative technique
- Typically bottom-up approach
- Profitable when subproblems *overlap* (i.e. coincide)
- Each subproblem solved only once