Algorithms Theory

14 – Dynamic Programming (1)

Very helpful and powerful algorithm design technique.

P.D. Dr. Alexander Souza

Winter term 11/12

Outline

- General approach, differences to a recursive solution

  Dynamic Programming = Recursion + Storing computed subproblem-solutions

- Basic example: Computation of the Fibonacci numbers

Method of dynamic programming

Recursive approach: Solve a problem by solving several smaller analogous subproblems of the same type. Then combine these solutions to generate a solution to the original problem.

Drawback: Repeated computation of solutions

Dynamic-programming method: Once a subproblem has been solved, store its solution in a table so that it can be retrieved later by simple table lookup.

Example: Fibonacci numbers

$f(0) = 0$
$f(1) = 1$
$f(n) = f(n-1) + f(n-2)$, for $n \geq 2$

Remark:

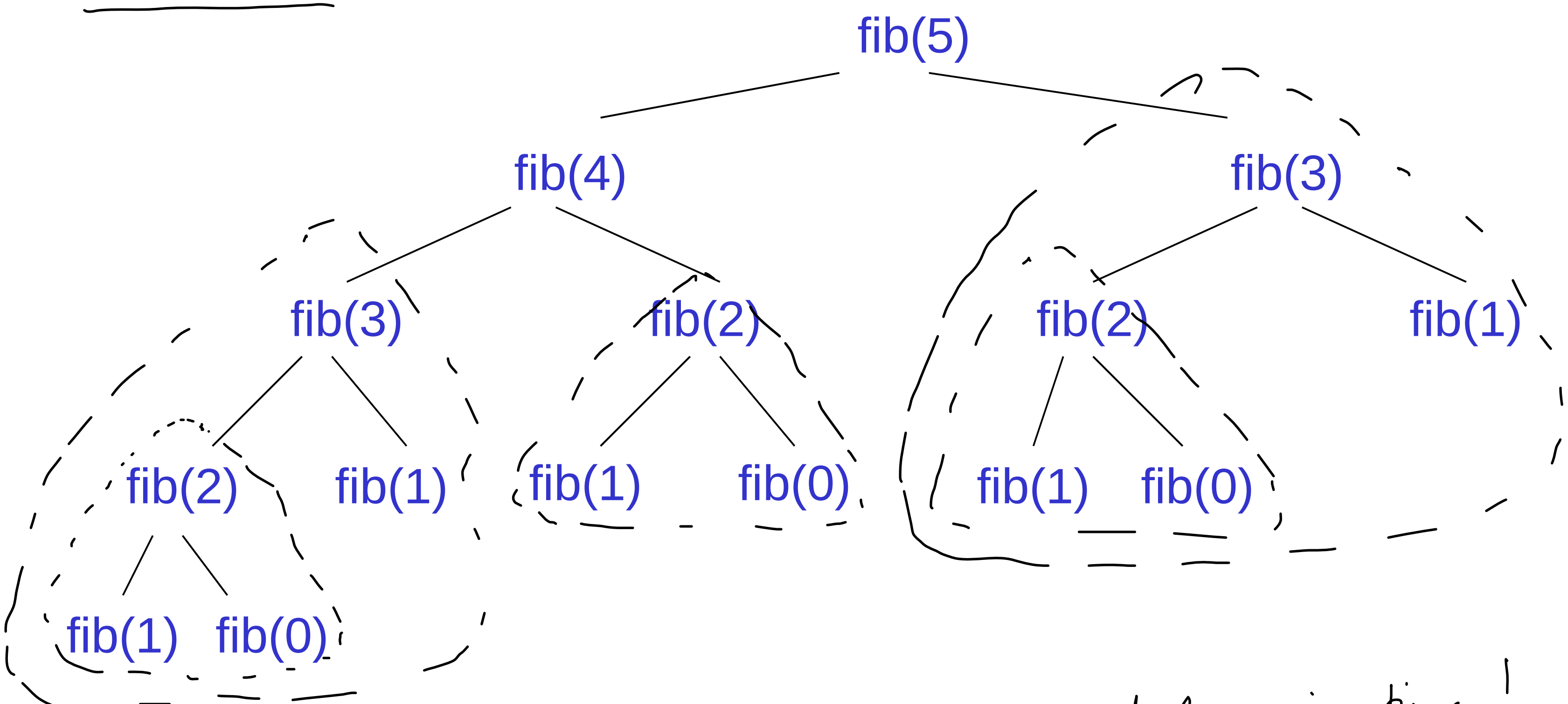
$$f(n) = \left[\frac{1}{\sqrt{5}} (1.618\ldots)^n \right]$$

Straightforward implementation:

procedure *fib* (*n* : *integer*) : *integer*
if (*n* == 0) or (*n* ==1)
  then return *n*
  else return *fib*(*n* – 1) + *fib*(*n* – 2)

Example: Fibonacci numbers

Recursion tree:



Repeated computation!

exponential running time !

$$T(n) \approx \left[\left(1+\frac{1}{\sqrt{5}}\right)\left(\frac{\sqrt{5}+1}{2}\right)^{n} - 1\right] \approx [1.447 \times 1.618^{n} - 1]$$

Dynamic programming

Approach:

1. Recursively define problem *P.*

2. Determine a set T consisting of all subproblems that have to be solved during the computation of a solution to P.

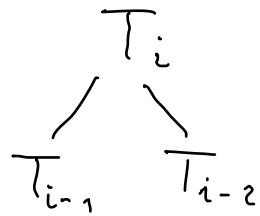
   $T = \{ T_0, ..., T_k \}$

3. Find an order $T_0, ..., T_k$ of the subproblems in T such that during the computation of a solution to T_i only subproblems T_j with $j < i$ arise.

4. Solve $T_0, ..., T_k$ in this order and store the solutions.

Example: Fibonacci numbers

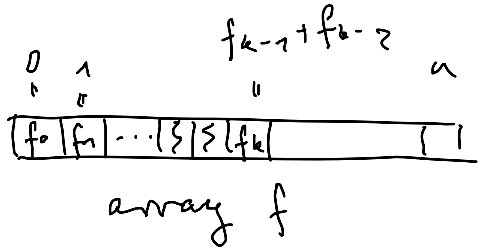
1. Recursive definition of the Fibonacci numbers, based on the standard equation. $f(n)$
2. $T = \{ f(0),..., f(n\text{-}1)\}$
3. $T_i = f(i), \quad i = 0,...,n - 1$



4. Computation of *fib*(*i*), for $i \geq 2$, only requires the results of the last two subproblems *fib*(*i* – 1) and *fib*(*i* – 2).

Example: Fibonacci numbers

Computation by dynamic programming, version 1:



procedure *fib*(*n* : *integer*) : *integer*

$f_0 := 0$; $f_1 := 1$

for $k := 2$ **to** n **do**

    $f_k := f_{k-1} + f_{k-2}$

return f_n

Running time $O(n)$

Space $O(n)$

Bottom-up

Example: Fibonacci numbers

Computation by dynamic programming, version 2:

procedure *fib* (*n* : *integer*) : *integer*

$f_{secondlast}$:= 0; f_{last} :=1

for *k* := 2 **to** *n* **do**

  $f_{current} := f_{last} + f_{secondlast}$

  $f_{secondlast} := f_{last}$

  $f_{last} := f_{current}$

if $n \leq 1$ **then return** *n* **else return** $f_{current}$;

Linear running time, constant space requirement!

Computing Fibonacci numbers with memoization

Recursive approach where we

Compute each number exactly once, store it in an array $F[0...n]$:

```
procedure fib (n : integer) : integer
  F[0] := 0;   F[1] := 1;
  for i :=2 to n do
      F[i] := ∞;
  return lookupfib(n)
```

"∞" $\equiv$ number not yet computed

Top-Down approach

The procedure *lookupfib* is defined as follows:

```
procedure lookupfib(k : integer) : integer
  if F[k] < ∞
     then return F[k]
     else F[k] := lookupfib(k – 1) + lookupfib(k – 2);
           return F[k]
```