

11 – Dynamic Programming (1)

Introduction

Weighted Interval Scheduling

- General approach, differences to a recursive solution
- Basic example: Computation of the Fibonacci numbers
 - Weighted interval scheduling

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

$$f(0) = 0$$

$$f(1) = 1$$

$$f(n) = f(n-1) + f(n-2), \text{ for } n \geq 2$$

Remark:

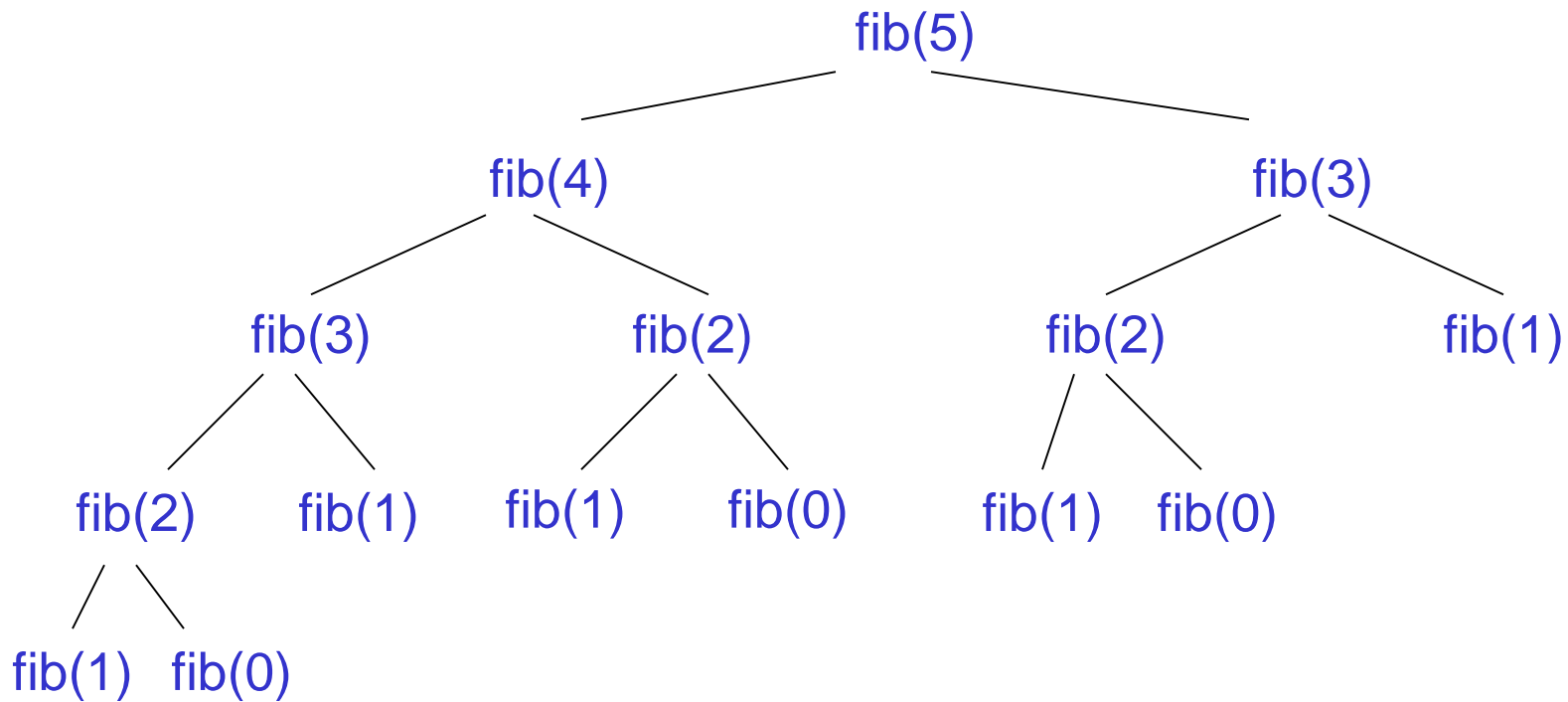
$$f(n) = \left[\frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n \right]$$

Straightforward implementation:

```
procedure fib (n : integer) : integer  
if (n = 0) or (n = 1)  
    then return n;  
    else return fib(n - 1) + fib(n - 2);
```

Fibonacci numbers

Recursion tree:



Repeated computation!

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 .
3. Find an order T_0, \dots, 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, \dots, T_k in this order and store the solutions.

Fibonacci numbers

1. Recursive definition of the Fibonacci numbers, based on the standard equation.
2. $T = \{f(0), \dots, f(n)\}$
3. $T_i = f(i), \quad i = 0, \dots, n$
4. Computation of $fib(i)$, for $i \geq 2$, only requires the results of the last two subproblems $fib(i-1)$ and $fib(i-2)$.

Fibonacci numbers

Computation by dynamic programming, version 1:

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

1 $F[0] := 0; F[1] := 1;$

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

3 $F[k] := F[k-1] + F[k-2];$

4 **return** $F[n];$

Fibonacci numbers

Computation by dynamic programming, version 2:

```
procedure fib (n : integer) : integer  
1  F(secondlast) := 0; F(last) := 1;  
2  for k := 2 to n do  
3      F(current) := F(last) + F(secondlast);  
4      F(secondlast) := F(last);  
5      F(last) := F(current);  
6  if n ≤ 1 then return n else return F(current);
```

Linear running time, constant space requirement!

Recursive computation using memoization



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

procedure *fib* ($n : \text{integer}$) : *integer*

1 $F[0] := 0$; $F[1] := 1$;

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

3 $F[i] := \infty$;

4 **return** *lookupfib*(n);

The procedure *lookupfib* is defined as follows:

procedure *lookupfib*($k : \text{integer}$) : *integer*

1 **if** $F[k] < \infty$

2 **then return** $F[k]$;

3 **else** $F[k] := \text{lookupfib}(k - 1) + \text{lookupfib}(k - 2)$;

4 **return** $F[k]$;

Weighted interval scheduling

Problem: Set $S = \{1, \dots, n\}$ of n requests for a resource.

Request i : $[s(i), f(i))$ $s(i)$ = start time $f(i)$ = finish time

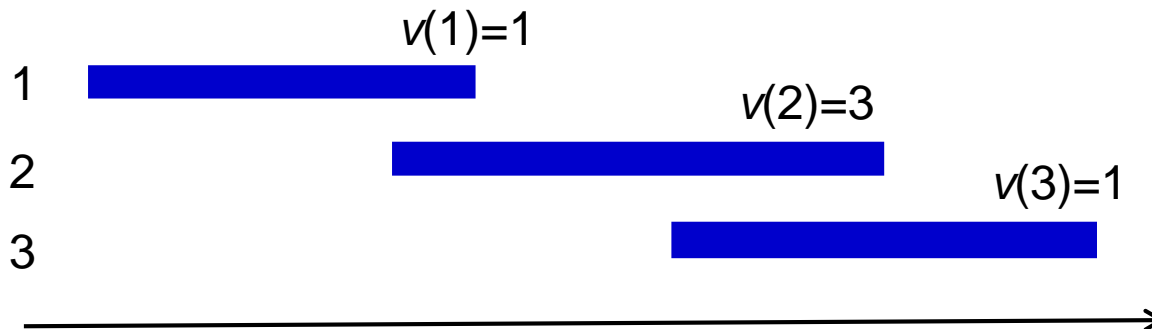
$v(i)$ = value/weight

Two requests are compatible if they do not overlap.

Goal: Select $S \subseteq \{1, \dots, n\}$ of mutually compatible requests so as to maximize

$$\sum_{i \in S} v(i).$$

Greedy* (Earliest Deadline First) is not optimal.



Predecessor function

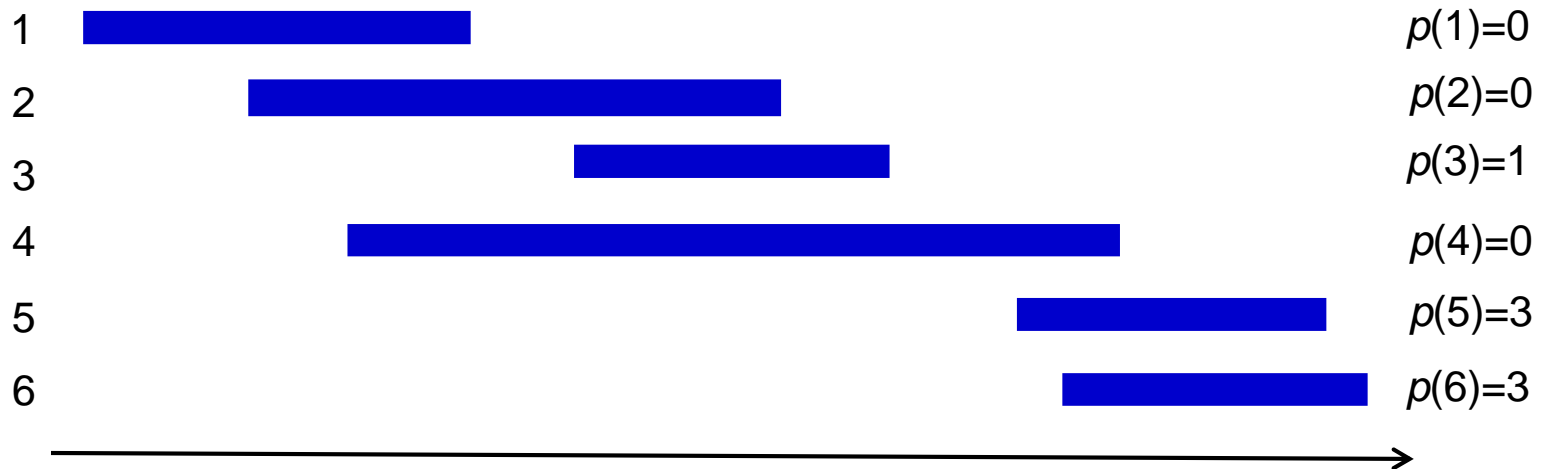
In the following, requests are numbered such that

$$f(1) \leq f(2) \leq f(3) \leq \dots \leq f(n).$$

For $j=1, \dots, n$

$p(j)$ = largest $i < j$ such that requests i and j do **not overlap**

$p(j) = 0$ if no request $i < j$ is disjoint from j



Dynamic programming approach

O = optimal subset of requests

- $n \notin O$: O is an optimal subset of $\{1, \dots, n-1\}$
- $n \in O$: remaining requests in O are an optimal subset of $\{1, \dots, p(n)\}$

For $j = 1, \dots, n$

O_j = optimal subset of requests from $\{1, \dots, j\}$

$OPT(j)$ = value of an optimal solution $OPT(0) := 0$

- $j \notin O_j$: O_j is an optimal subset of $\{1, \dots, j-1\}$
- $j \in O_j$: remaining requests in O_j are an optimal subset of $\{1, \dots, p(j)\}$

Dynamic programming approach

For $j = 1, \dots, n$

$O_j = \text{optimal subset}$ of requests from $\{1, \dots, j\}$

$\text{OPT}(j) = \text{value}$ of an optimal solution $\text{OPT}(0) := 0$

- $j \notin O_j$: O_j is an optimal subset of $\{1, \dots, j-1\}$
- $j \in O_j$: remaining requests in O_j are an optimal subset of $\{1, \dots, p(j)\}$

$$\text{OPT}(j) = \max\{ v(j) + \text{OPT}(p(j)) , \text{OPT}(j-1) \}$$

Request j belongs to an optimal solution for $\{1, \dots, j\}$ if and only if

$$v(j) + \text{OPT}(p(j)) \geq \text{OPT}(j-1).$$

Straightforward implementation

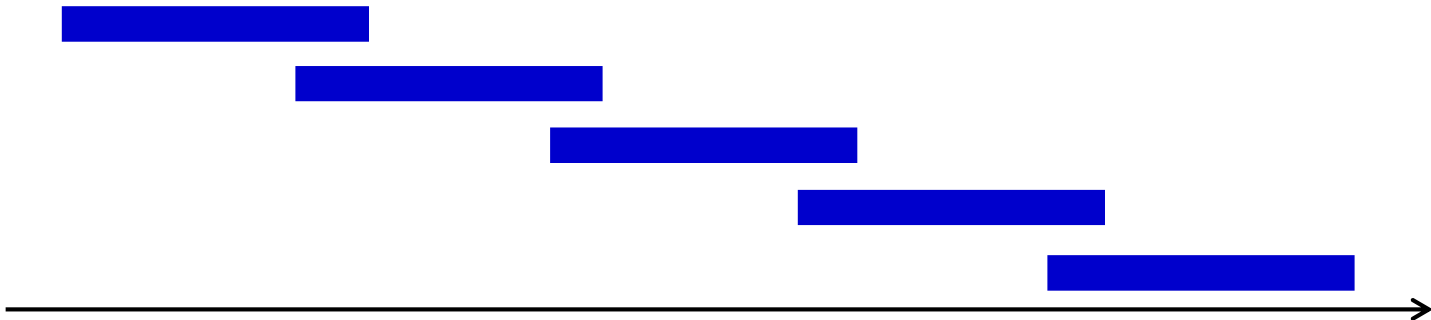
Assume that values $p(j)$, for $j=1, \dots, n$, have been computed.

procedure ComputeOpt(j : *integer*)

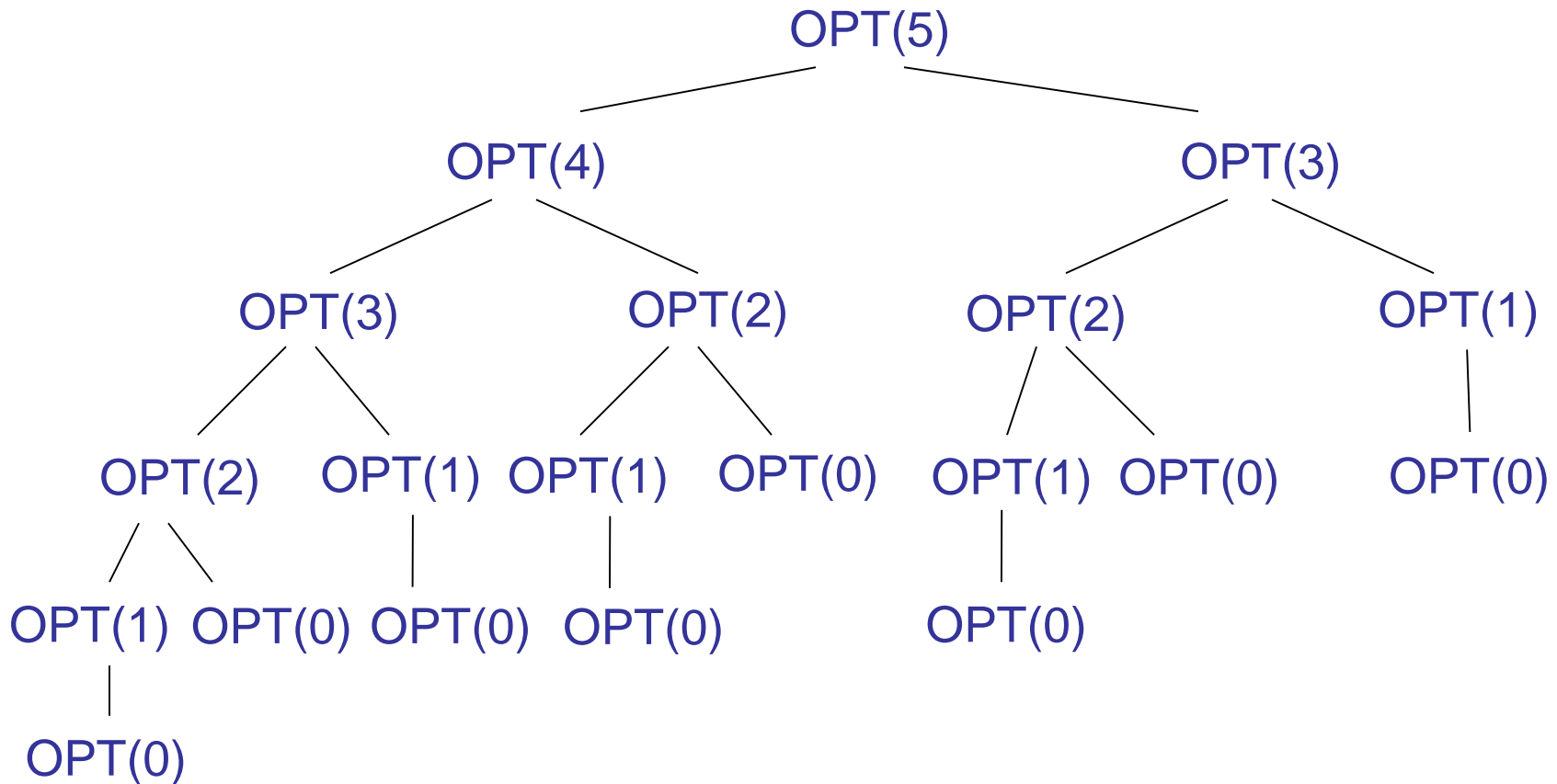
1 **if** $j = 0$

2 **then return** 0;

3 **else return** $\max\{v(j) + \text{ComputeOPT}(p(j)) , \text{ComputeOPT}(j-1)\}$;



Instance taking exponential time



Iterative solution

Array $M[0..n]$ contains the values of the optimal solutions.

```
procedure ComputeOpt( $n$  : integer)  
1   $M[0] := 0$ ;  
2  for  $j := 1$  to  $n$  do  
3     $M[j] := \max\{v(j) + M[p(j)] , M[j-1]\}$ ;  
4  endfor;
```

Running time: $O(n)$

Recursion using memoization

```
procedure ComputeOpt(j : integer)
1  if j = 0 then
2    return 0;
3  else if M[j] is not empty then
4    return M[j];
5  else
6    M[j] := max{v(j) + ComputeOPT(p(j)) , ComputeOpt(j-1)};
7    return M[j];
8  endif;
```

Proposition: The running time of $\text{ComputeOpt}(n)$ is $O(n)$ if the requests are sorted in order of non-decreasing finish times and the values $p(j)$, $1 \leq j \leq n$, are computed.

Proof: The running time is a constant times the number of recursive calls to ComputeOpt . Two calls are issued whenever a new array entry is filled. Hence there are a total of at most $2n$ calls.

Computing a solution

```
procedure FindSolution(j : integer)
1 if j = 0 then
2   Output nothing;
3 else if  $v(j) + M[p(j)] \geq M[j-1]$  then
4   Output j together with the result of FindSolution(p(j));
5 else
6   Output the result of FindSolution(j-1);
7 endif;
```

FindSolution calls itself only on strictly smaller values. Therefore FindSolution(n) issues less than n recursive calls and the running time is $O(n)$.