11 – Dynamic Programming (1)

Introduction

Weighted Interval Scheduling

Outline

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

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:

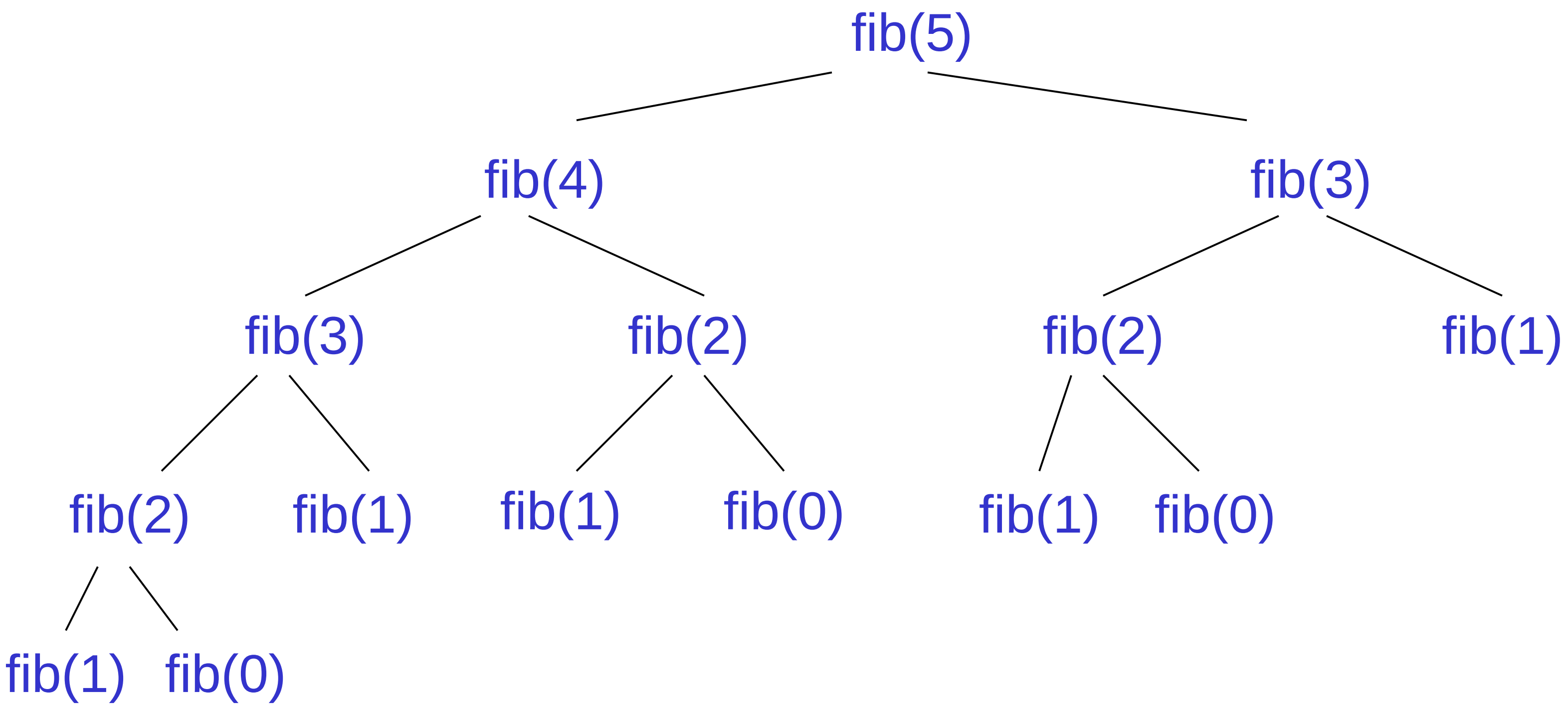
$$f(n) = \left\lfloor \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n \right\rceil$$

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 $T_0, \ldots, T_k$ 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 $T_0, \ldots, T_k$ in this order and store the solutions.

Fibonacci numbers

1. Recursive definition of the Fibonacci numbers, based on the standard equation.

2. $T = \{f(0),\ldots,f(n)\}$

3. $T_i = f(i), \quad i = 0,\ldots,n$

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*
F[0] := 0; *F*[1] := 1;
for *k* := 2 **to** *n* **do**
    F[*k*] := *F*[*k-1*] + *F*[*k-2*];
return *F*[*n*];

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!

Recursive computation using memoization

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);
```

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];
```

Weighted interval scheduling

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

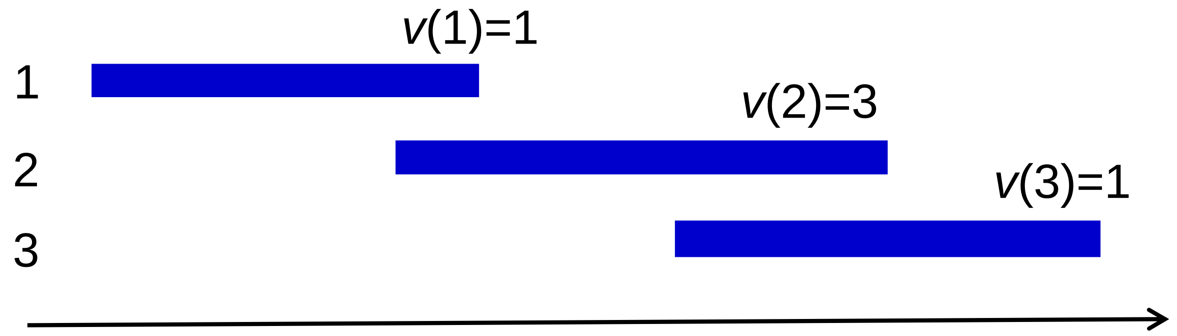
Request i: $[s(i), f(i))$ $s(i)$ = start time $f(i)$ = finish time

$v(i)$ = value/weight

Two requests are compatible if they do not overlap.

Goal: Select $S \subseteq \{1,\ldots,n\}$ of mutually compatible requests so as to maximize $\Sigma_{i \in S}\, v(i)$.

Greedy* (Earliest Deadline First) is not optimal.



Predecessor function

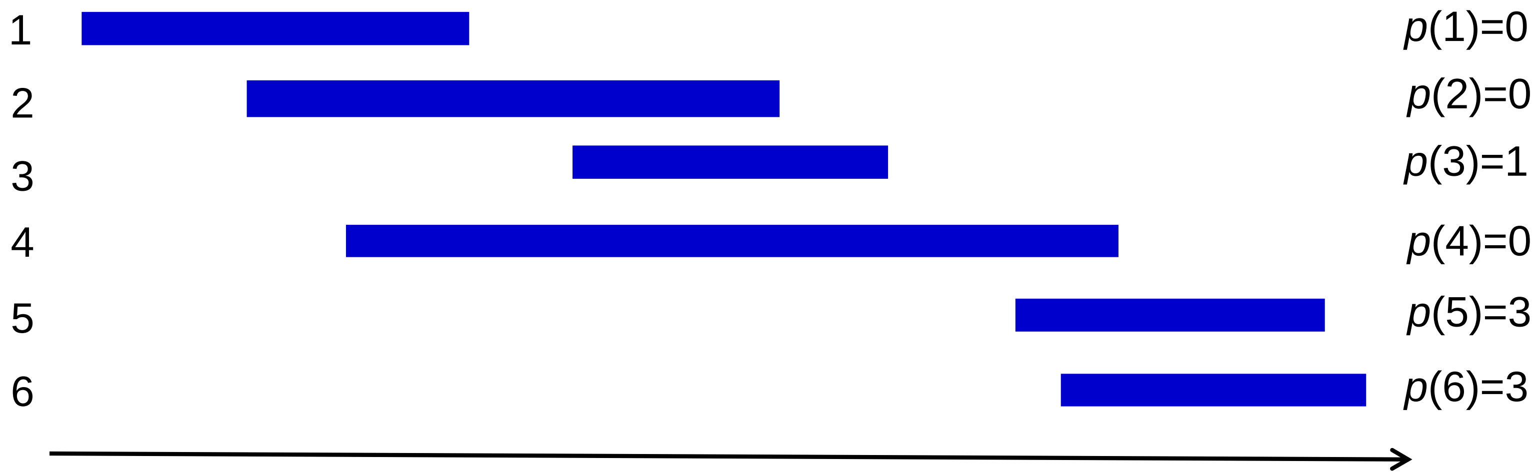
In the following, requests are numbered such that

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

For $j = 1,\ldots,n$

$p(j) =$ 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 $\{1,\ldots,n\text{-}1\}$
- $n \in O$: remaining requests in O are an optimal subset of $\{1,\ldots,p(n)\}$

For $j = 1,\ldots,n$

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

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

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

Dynamic programming approach

For $j = 1,\ldots,n$

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

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

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

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

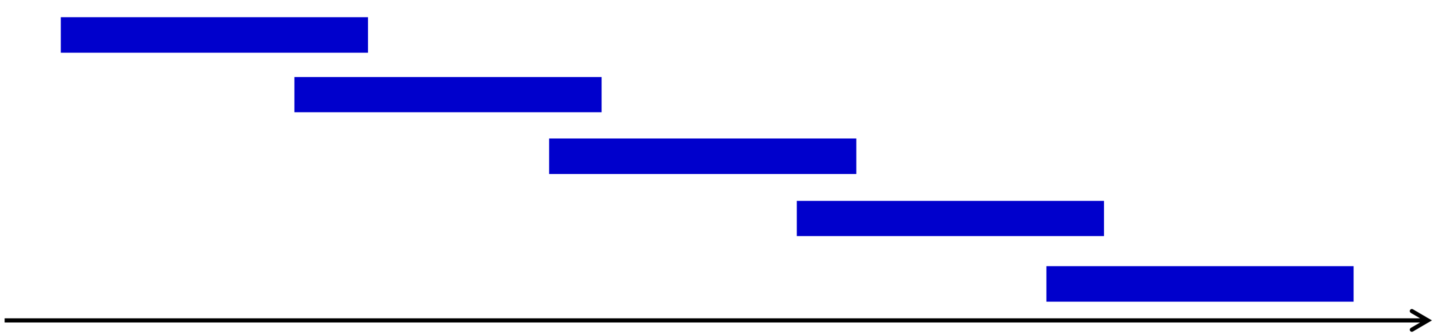
Request j belongs to an optimal solution for $\{1,\ldots,j\}$ if and only if

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

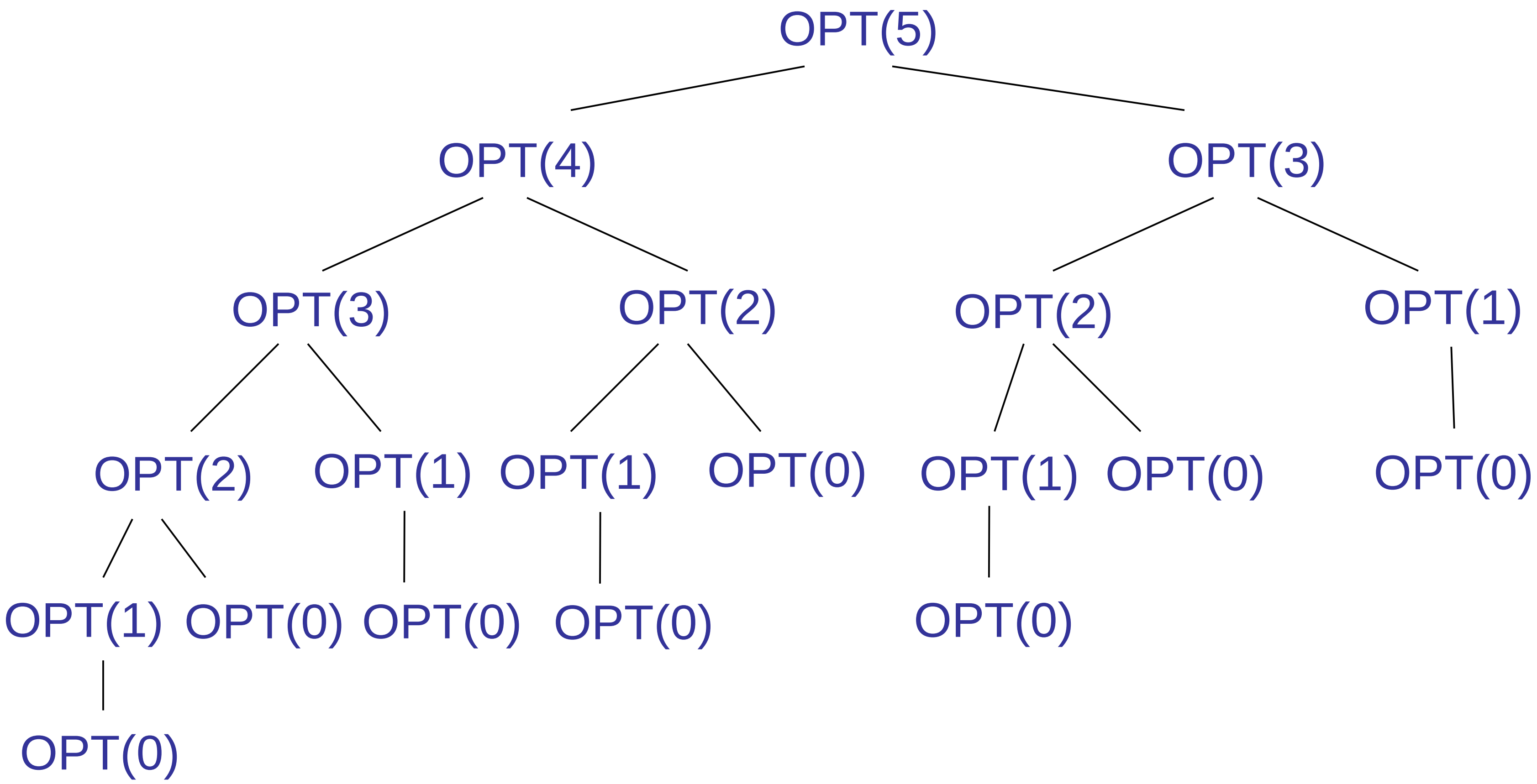
Straightforward implementation

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

procedure ComputeOpt(j : *integer*)
if $j = 0$
    then return 0;
    else return max{$v(j)$ + OPT($p(j)$) , OPT(j-1)};



Instance taking exponential time



Iterative solution

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

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

Running time: O(*n*)

Recursion using memoization

procedure ComputeOpt(*j* : *integer*)

if $j = 0$ **then**
    return 0;
else if M[*j*] is not empty **then**
    return M[*j*];
else
    M[*j*] := max{$v(j)$ + ComputeOPT($p(j)$) , ComputeOpt(j-1)};
    return M[*j*];
endif;

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

if $j = 0$ **then**
    Output nothing;
else if $v(j) + M[p(j)] \geq M[j-1]$ **then**
    Output j together with the result of FindSolution($p(j)$);
else
    Output the result of FindSolution(j-1);
endif;

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

