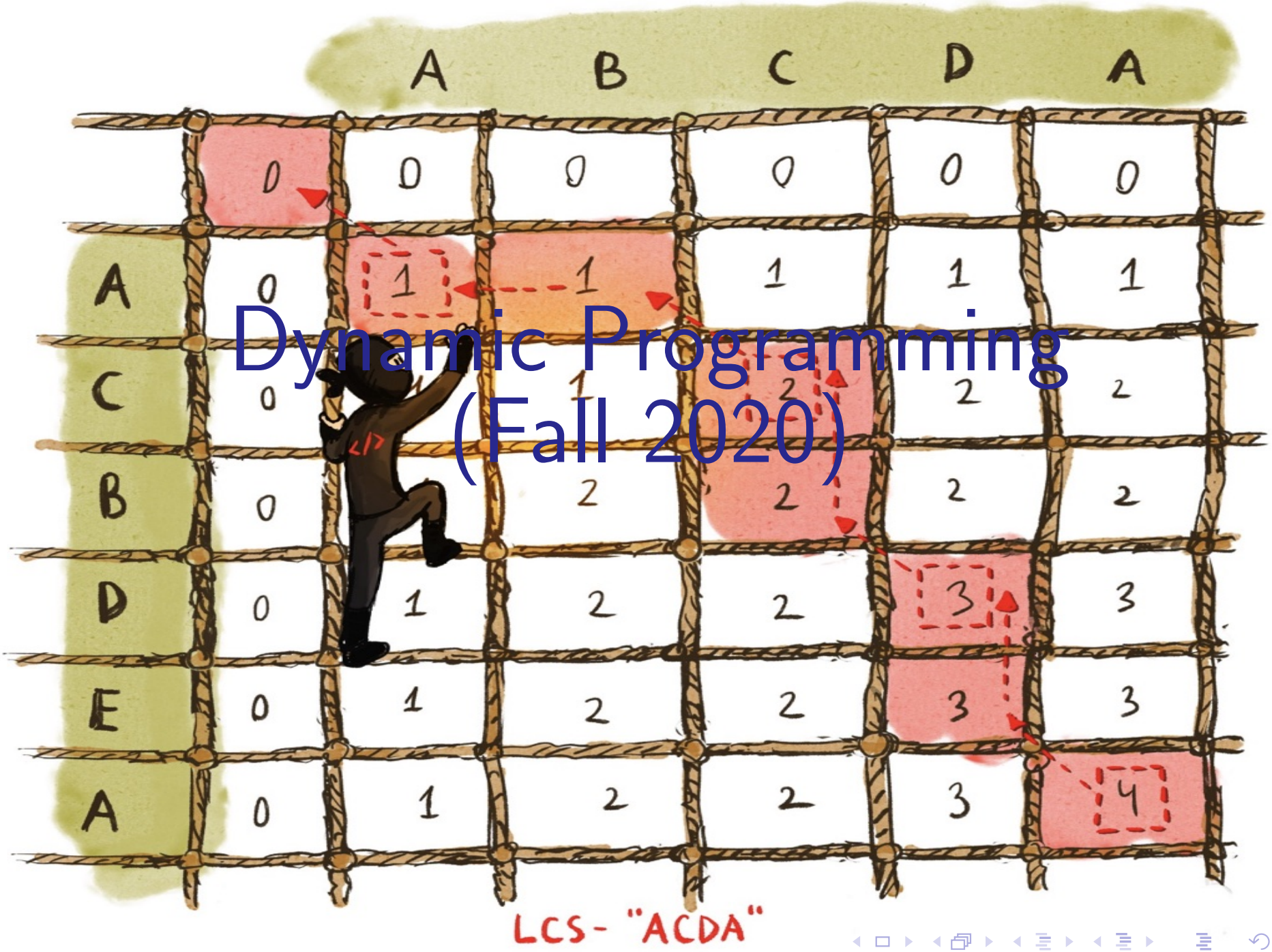


Dynamic Programming (Fall 2020)

Dynamic Programming.

For a gentle introduction to DP see Chapter 6 in DPV, KT and CLRS also have a chapter devoted to DP.

Richard Bellman: *An introduction to the theory of dynamic programming* RAND, 1953

Dynamic programming is a powerful technique for efficiently implement *recursive algorithms* by storing partial results and re-using them when needed.

Dynamic Programming: where?

Explore the space of all possible solutions by decomposing things into subproblems, and then building up correct solutions to larger problems. For example with a recursive backtracking algorithm

The number of solved subproblems can be exponential, but if the number of different problems is polynomial, we can get a polynomial algorithm by avoiding to repeat the computation of the same subproblem many times.

The key for getting a polynomial time algorithm using DP is to realize that the problem has a certain amount of recursive structure that we can exploit but having only a polynomial number of different subproblems.

Properties of Dynamic Programming

Dynamic Programming works efficiently when:

- Subproblems: There must be a way of breaking the global optimization problem into subproblems, each having a similar structure to the original problem but smaller size.
- Optimal sub-structure: An optimal solution to a problem must be a composition of optimal subproblem solutions, using a relatively simple combining operation.
- Repeated subproblems: The recursive algorithm solves a small number of distinct subproblems, but they are repeatedly solved many times.

This last property allows us to take advantage of memoization, istore intermediate values, using the appropriate dictionary data structure, and reuse when needed.

Difference with greedy

- Greedy problems have the greedy choice property: locally optimal choices lead to globally optimal solution.
- For some DP problems greedy choice is not possible globally optimal solution requires acces to many subproblems.
- I.e. In DP we solve all possible subproblems, while in greedy we are bound for the initial choice

Difference with divide and conquer

- Both require recursive programming with subproblems with a similar structure to the original
- D & C breaks a problems into a small number of subproblems each of them with size a fraction of the original size (size/b).
- In DP, we break into many subproblems with smaller size, but often, their sizes are not a fraction of the initial size.

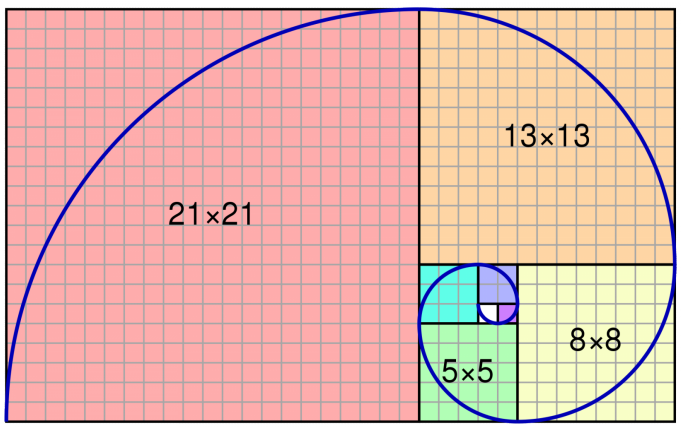
A first example: Fibonacci Recurrence.

The Fibonacci numbers are defines recursively as follows:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \qquad \text{for } n \geq 2$$

0,1,1,2,3,5,8,13,21,34,55,89,..

8th Fibonacci term

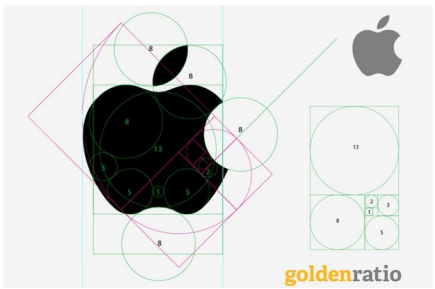


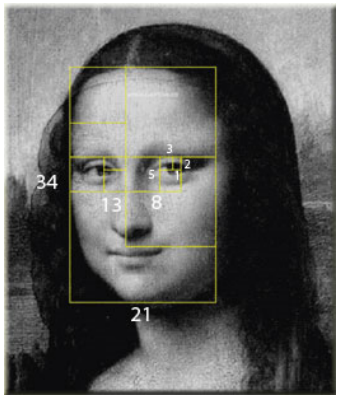
The golden ratio

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \varphi = 1.61803398875\ldots$$

Some examples of Fibonacci sequence in life

In nature, there are plenty of examples that follows a Fibonacci sequence pattern, from the shells of mollusks to the leaves of the palm. Below you have some further examples:





YouTube: Fibonacci numbers, golden ratio and nature

Computing the n-th Fibonacci number.

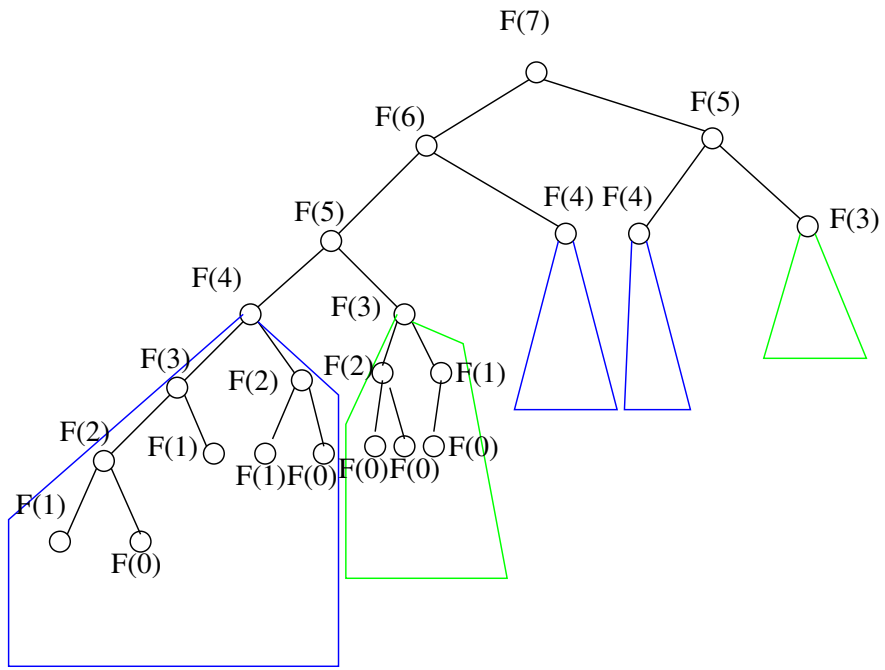
INPUT: $n \in \mathbb{N}$
QUESTION: Compute F_n.

A recursive solution:

Fibonacci (n)
if $n = 0$ **then**
  return 0
else if $n = 1$ **then**
  return 1
else
  return $(\textbf{Fibonacci}(n-1) + \textbf{Fibonacci}(n-2))$

Computing F_7.

As $F_{n+1}/F_n \sim (1+\sqrt{5})/2 \sim 1.61803$ then $F_n > 1.6^n$, and to compute F_n we need 1.6^n recursive calls.



Notice the computation of subproblem $F(i)$ is repeated many times

A DP implementation: memoization

To avoid repeating multiple computations of subproblems, keep a dictionary with the solution of the solved subproblems.

Fibo (n)
for $i \in [0..n]$ **do**
    $F[i] = -1$
$F[0] = 0$; $F[1] = 1$
return (**Fibonacci**(n))

Fibonacci (i)
if $F[i]!=-1$ **then**
    return ($F[i]$)
$F[i]=$ **Fibonacci**$(i-1)$ + **Fibonacci**$(i-2)$
return ($F[i]$)

Each subproblem requires $O(1)$ operations, we have $n+1$ subproblems, so the cost is $O(n)$.
We are using $O(n)$ additional space.

A DP algorithm: tabulating

To avoid repeating multiple computations of subproblems, carry the computation bottom-up and store the partial results in a table

DP-Fibonacci (n) {Construct table}
$F[0] = 0$
$F[1] = 1$
for $i = 2$ to n **do**
  $F[i] = F[i-1] + F[i-2]$
return $(F[n])$

F[0]	0
F[1]	1
F[2]	1
F[3]	2
F[4]	3
F[5]	5
F[6]	8
F[7]	13

To get F_n need $O(n)$ time and $O(n)$ space.

A DP algorithm: reducing space

In the tabulating approach, we always access only the previous two values. We can reduce space by storing only the values that we will need in the next iteration.

DP-Fibonacci (n) {Construct table}
$p1 = 0$
$p2 = 1$
for $i = 2$ to n **do**
  $p3 = p2 + p1$
  $p1 = p2$; $p2 = p3$
return (p3)

To get F_n need $O(n)$ time and $O(1)$ space.

Guideline to implement Dynamic Programming

This first example of PD was easy, as the recurrence is given in the statement of the problem.

1. *Characterize the structure of subproblems:* make sure space of subproblems is not exponential. Define variables.
2. Define recursively the value of an optimal solution: Find the correct recurrence, with solution to larger problem as a function of solutions of sub-problems.
3. *Compute, bottom-up, the cost of a solution:* using the recursive formula, tabulate solutions to smaller problems, until arriving to the value for the whole problem.
4. *Construct an optimal solution:* compute additional information to trace-back from optimal value.

Implementation of Dynamic Programming

Memoization: technique consisting in storing the results of subproblems and returning the result when the same sub-problem occur again.Mainly used to implement recursive algoriths using PD.

- In implementing the DP recurrence using recursion could be very inefficient because solves many times the same sub-problems.
- But if we could manage to solve and store the solution to sub-problems without repeating the computation, that could be a clever way to use recursion + memoization.
- To implement memoization use any dictionary data structure, usually tables or hashing.

Implementation of Dynamic Programming

- The other way to implement DP is using iterative algorithms.
- DP is a trade-off between time speed vs. storage space.
- In general, although recursive algorithms ha exactly the same running time than the iterative version, the constant factor in the O is quite more larger because the overhead of recursion. On the other hand, in general the memoization version is easier to program, more concise and more elegant.

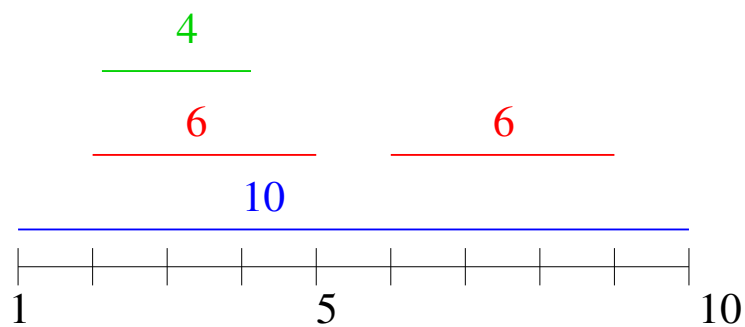
Top-down: Recursive and Bottom-up: Iterative

But using Memoization, DP can can be efficiently computed by using recursion .

Weighted Activity Selection problem

Weighted Activity Selection problem: Given a set $S = \{1, 2, \ldots, n\}$ of activities to be processed by a single resource. Each activity i has a start time s_i and a finish time f_i, with $f_i > s_i$, and a weight w_i. Find the set of mutually compatible activities such that it maximizes $\sum_{i \in S} w_i$

Recall: We saw that some greedy strategies did not provide always a solution to this problem.



W Activity Selection: looking for a recursive solution

- Let us think of a backtracking algorithm for the problem.
- The solution is a selection of activities, i.e., a subset $S \subseteq \{1, \ldots, n\}$.
- We can adapt the backtracking algorithm to compute all subsets.
- When processing element i, we branch
  - i is in the solution, then all activities that overlap with i must be removed.
  - i is not in the solution, then i must be removed.

W Activity Selection: looking for a recursive solution

This suggest to keep at each backtracking call a partial solution (S) and a candidate set (C), those activities that are compatible with the ones in S.

WAS-1 (S, C)
if $C = \emptyset$ **then**
  return $(W(S))$
Let i be an element in C; $C = C - \{i\}$;
Let A be the set of activities in C that overlap with i
return $(\max\{$**WAS-1**$(S \cup \{i\}, C - A),$ **WAS-1**$(S, C)\})$

How many subproblems appear here?

W Activity Selection: looking for a recursive solution

This suggest to keep at each backtracking call a partial solution (S) and a candidate set (C), those activities that are compatible with the ones in S.

WAS-1 (S, C)
if $C = \emptyset$ **then**
  return $(W(S))$
Let i be an element in C; $C = C - \{i\}$;
Let A be the set of activities in C that overlap with i
return $(\max\{$**WAS-1**$(S \cup \{i\}, C - A),$ **WAS-1**$(S, C)\})$

How many subproblems appear here? hard to count better than $O(2^n)$.

W Activity Selection: looking for a recursive solution

For the unweighted case, the greedy algorithm made use of a particular ordering that helped to discard overlapping tasks. Assume that the activities are sorted by finish time, i.e., $f_1 \leq f_2 \leq \ldots f_n$.

WAS-2 (S, i)
if $i == 1$ **then**
  return $(W(S) + w_1)$
if $i == 0$ **then**
  return $(W(S))$
Let j be the largest integer $j < i$ such that $f_j \leq s_i$, 0 if none is compatible.
return $(\max\{$**WAS-2**$(S \cup \{i\}, j),$ **WAS-2**$(S, i-1)\})$

WAS-2 $(\emptyset, n)$ will return the cost of an optimal solution. Why?

W Activity Selection: looking for a recursive solution

For the unweighted case, the greedy algorithm made use of a particular ordering that helped to discard overlapping tasks. Assume that the activities are sorted by finish time, i.e., $f_1 \leq f_2 \leq \ldots f_n$.

WAS-2 (S, i)
if $i == 1$ **then**
  return $(W(S) + w_1)$
if $i == 0$ **then**
  return $(W(S))$
Let j be the largest integer $j < i$ such that $f_j \leq s_i$, 0 if none is compatible.
return $(\max\{$**WAS-2**$(S \cup \{i\}, j),$ **WAS-2**$(S, i-1)\})$

WAS-2 $(\emptyset, n)$ will return the cost of an optimal solution. Why?
Easy to modify to get also the optimal solution.

W Activity Selection: looking for a recursive solution

For the unweighted case, the greedy algorithm made use of a particular ordering that helped to discard overlapping tasks. Assume that the activities are sorted by finish time, i.e., $f_1 \leq f_2 \leq \ldots f_n$.

WAS-2 (S, i)
if $i == 1$ **then**
  return $(W(S) + w_1)$
if $i == 0$ **then**
  return $(W(S))$
Let j be the largest integer $j < i$ such that $f_j \leq s_i$, 0 if none is compatible.
return $(\max\{$**WAS-2**$(S \cup \{i\}, j),$ **WAS-2**$(S, i-1)\})$

WAS-2 $(\emptyset, n)$ will return the cost of an optimal solution. Why?
Easy to modify to get also the optimal solution.

The algorithm has cost

W Activity Selection: looking for a recursive solution

For the unweighted case, the greedy algorithm made use of a particular ordering that helped to discard overlapping tasks. Assume that the activities are sorted by finish time, i.e., $f_1 \leq f_2 \leq \ldots f_n$.

WAS-2 (S, i)
if $i == 1$ **then**
  return $(W(S) + w_1)$
if $i == 0$ **then**
  return $(W(S))$
Let j be the largest integer $j < i$ such that $f_j \leq s_i$, 0 if none is compatible.
return $(\max\{$**WAS-2**$(S \cup \{i\}, j),$ **WAS-2**$(S, i-1)\})$

WAS-2 $(\emptyset, n)$ will return the cost of an optimal solution. Why?
Easy to modify to get also the optimal solution.

The algorithm has cost $O(2^n)$.

W Activity Selection: looking for a recursive solution

For the unweighted case, the greedy algorithm made use of a particular ordering that helped to discard overlapping tasks. Assume that the activities are sorted by finish time, i.e., $f_1 \leq f_2 \leq \ldots f_n$.

WAS-2 (S, i)
if $i == 1$ **then**
  return $(W(S) + w_1)$
if $i == 0$ **then**
  return $(W(S))$
Let j be the largest integer $j < i$ such that $f_j \leq s_i$, 0 if none is compatible.
return $(\max\{$**WAS-2**$(S \cup \{i\}, j),$ **WAS-2**$(S, i-1)\})$

WAS-2 $(\emptyset, n)$ will return the cost of an optimal solution. Why?
Easy to modify to get also the optimal solution.

The algorithm has cost $O(2^n)$.

The number of subproblems is

W Activity Selection: looking for a recursive solution

For the unweighted case, the greedy algorithm made use of a particular ordering that helped to discard overlapping tasks. Assume that the activities are sorted by finish time, i.e., $f_1 \leq f_2 \leq \ldots f_n$.

WAS-2 (S, i)
if $i == 1$ **then**
  return $(W(S) + w_1)$
if $i == 0$ **then**
  return $(W(S))$
Let j be the largest integer $j < i$ such that $f_j \leq s_i$, 0 if none is compatible.
return $(\max\{$**WAS-2**$(S \cup \{i\}, j),$ **WAS-2**$(S, i-1)\})$

WAS-2 $(\emptyset, n)$ will return the cost of an optimal solution. Why?
Easy to modify to get also the optimal solution.

The algorithm has cost $O(2^n)$.

The number of subproblems is $O(n)$.

W Activity Selection: looking for a recursive solution

For the unweighted case, the greedy algorithm made use of a particular ordering that helped to discard overlapping tasks. Assume that the activities are sorted by finish time, i.e., $f_1 \leq f_2 \leq \ldots f_n$.

WAS-2 (S, i)
if $i == 1$ **then**
  return $(W(S) + w_1)$
if $i == 0$ **then**
  return $(W(S))$
Let j be the largest integer $j < i$ such that $f_j \leq s_i$, 0 if none is compatible.
return $(\max\{$**WAS-2**$(S \cup \{i\}, j),$ **WAS-2**$(S, i-1)\})$

WAS-2 $(\emptyset, n)$ will return the cost of an optimal solution. Why?
Easy to modify to get also the optimal solution.

The algorithm has cost $O(2^n)$.

The number of subproblems is $O(n)$. What are the subproblems?

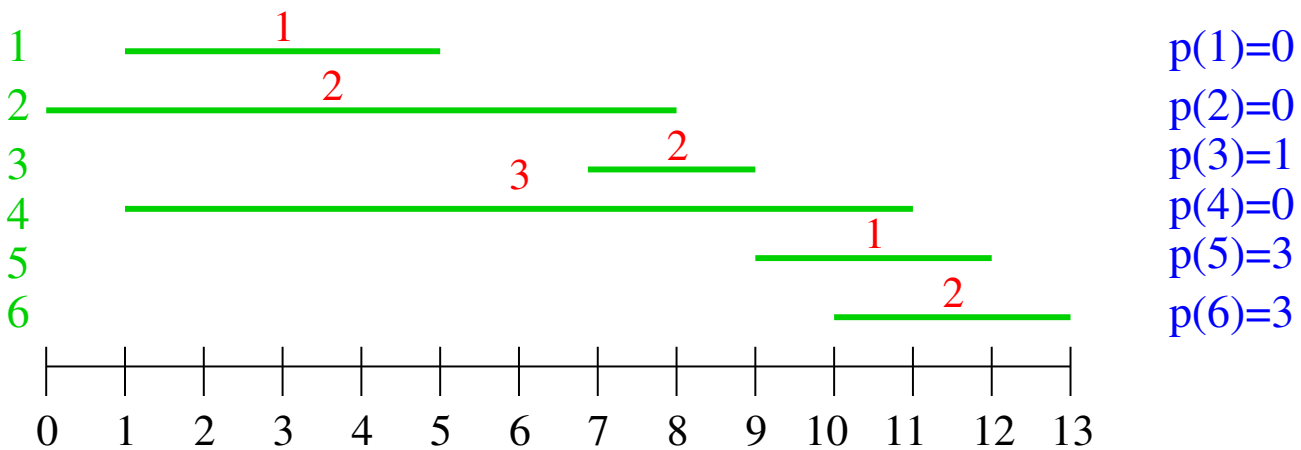
DP from WAS-2: a recurrence

We have $\{1, 2, \ldots, n\}$ activities with $f_1 \leq f_2 \leq \cdots \leq f_n$ and weights $\{w_i\}$.
Therefore, we may need a $O(n \lg n)$ pre-processing sorting step.

We only got subproblems defined by subsets of tasks $\{1, \ldots, i\}$, for $0 \leq i \leq n$

Define $p(i)$ to be the largest integer $j < i$ such that i and j are disjoints ($p(i) = 0$ if no disjoint $j < i$ exists).

Let Opt(j) be the value of the optimal solution to the problem consisting of activities in the range 1 to j. Let O_j be the set of jobs in an optimal solution for $\{1, \ldots, j\}$.



DP from WAS-2: a recurrence

We have $\{1, 2, \ldots, n\}$ activities with $f_1 \leq f_2 \leq \cdots \leq f_n$ and weights $\{w_i\}$.
Therefore, we may need a $O(n \lg n)$ pre-processing sorting step.

We only got subproblems defined by subsets of tasks $\{1, \ldots, i\}$, for $0 \leq i \leq n$

Define $p(i)$ to be the largest integer $j < i$ such that i and j are disjoints ($p(i) = 0$ if no disjoint $j < i$ exists).

Let $\mathrm{Opt}(j)$ be the value of the optimal solution to the problem consisting of activities in the range 1 to j. Let O_j be the set of jobs in an optimal solution for $\{1, \ldots, j\}$.

$$\mathrm{Opt}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{(\mathrm{Opt}(p[j]) + w_j), \mathrm{Opt}[j-1]\} & \text{if } j \geq 1 \end{cases}$$

DP from WAS-2: a recurrence

$$\text{Opt}(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{(\text{Opt}(p[j]) + w_j), \text{Opt}[j-1]\} & \text{if } j \geq 1 \end{cases}$$

Correctness: From the previous discussion, we have two cases:

1.- $j \in O_j$:

- As j is part of the solution,
- no jobs $\{p(j)+1, \ldots, j-1\}$ are in O_j,
- if $O_j - \{j\}$ must be an optimal solution for $\{1, \ldots, p[j])\}$, otherwise then $O'_j = O_{p[j]} \cup \{j\}$ will be better (optimal substructure)

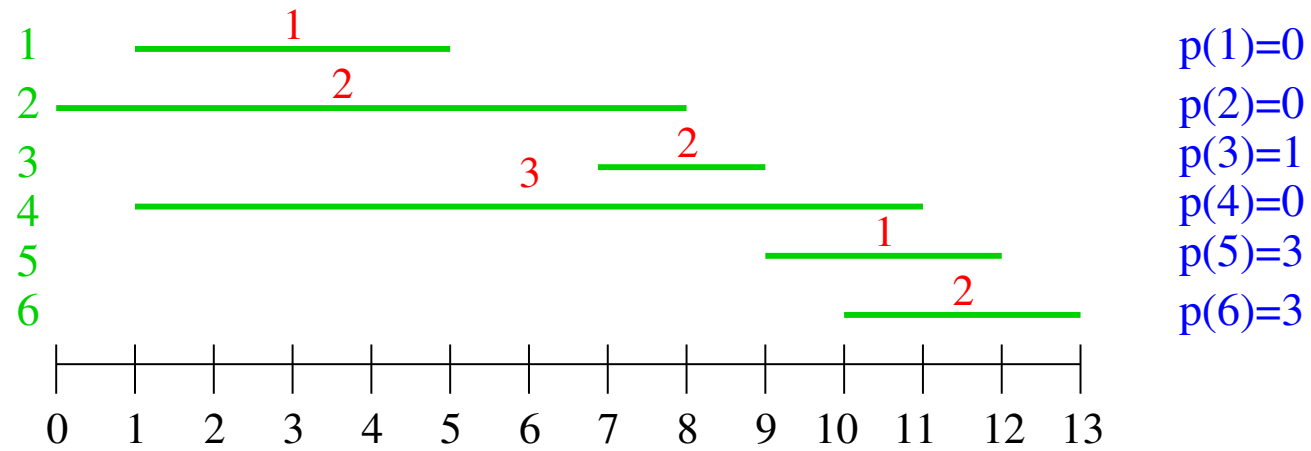
2.- If $j \notin O_j$: then O_j is an optimal solution to $\{1, \ldots, j-1)\}$.

DP from WAS-2: Preprocessing

Considering the set of activities S, we start by a pre-processing phase:

- Sort the activities by increasing values of finish times.
- To compute the values of $p[i]$,
  - sort the activities by increasing values of start time.
  - merging the sorted list of finishing times an the sorted list of start times, in case of tie put before the finish times.
  - $p[j]$ is the activity whose finish time precedes s_j in the combined order, activity 0, if no finish time precedes s_j
- We can thus compute the p values in $O(n \lg n + n) = O(n) \lg n$

DP from WAS-2: Preprocessing



Sorted finish times: 1:5, 2:8, 3:9, 4:11, 5:12, 6:13

Sorted start times: 2:0, 1:1, 4:1, 3:7, 5:9, 6:10

Merged sequence: 2:0, 1:1, 4:1,1:5,3:7,3:9,5:9,6:10, 5:12, 6:13

DP from WAS-2: Memoization

We assume that S is sorted and all $p(j)$ are computed and tabulated in $P[1 \cdots n]$

We keep a table $W[n+1]$, at the end $W[i]$ will hold the weight of an optimal solution for subproblem $\{1, \ldots, i\}$. Initially, set all entries to -1 and $W[0] = 0$.

R-Opt (j)
if $W[j]! = -1$ **then**
  return $(W[j])$
else
  return $\max(w_j + \textbf{R-Opt}(P[j])), \textbf{R-Opt}(j-1))$

DP from WAS-2: Memoization

We assume that S is sorted and all $p(j)$ are computed and tabulated in $P[1 \cdots n]$

We keep a table $W[n+1]$, at the end $W[i]$ will hold the weight of an optimal solution for subproblem $\{1, \ldots, i\}$. Initially, set all entries to -1 and $W[0] = 0$.

R-Opt (j)
if $W[j]! = -1$ **then**
  return $(W[j])$
else
  return $\max(w_j + \textbf{R-Opt}(P[j])), \textbf{R-Opt}(j-1))$

No subproblem is solved more than once, so cost is $O(n \log + n) = O(n \log n)$

DP from WAS-2: Iterative

We assume that S is sorted and all $p(j)$ are computed and tabulated in $P[1 \cdots n]$

We keep a table $W[n+1]$, at the end $W[i]$ will hold the weight of an optimal solution for subproblem $\{1, \ldots, i\}$.

Opt-Val (n)
$W[0] = 0$
for $j = 1$ to n **do**
  $W[j] = \max(W[P[j]] + w_j, W[j-1])$
return $W[n]$

Notice: Both algorithms gave only the numerical max. weight
We have to recover the list of activities that form the solution $W[n]$.

Time complexity: $O(n)$ (not counting the sorting).

DP from WAS-2:Returning the selected activities

To get also the list of selected activities, we use W to recover the decision taken in computing $W[n]$.

Find-Opt (j)
if $j = 0$ **then**
  return $\emptyset$
else if $W[p[j]] + w_j > W[j-1]$ **then**
  return $(\{j\} \cup \text{Find-Opt}(p[j]))$
else
  return $(\text{Find-Opt}(j-1))$

Time complexity: $O(n)$

DP for Weighted Activity Selection

- We started from a suitable recursive algorithm, which runs $O(2^n)$ but solves only $O(n)$ different subproblemes.
- Perform some preprocesing.
- Compute the weight of an optimal solution to each of the $O(n)$ subproblems.
- Obtain an optimal solution.