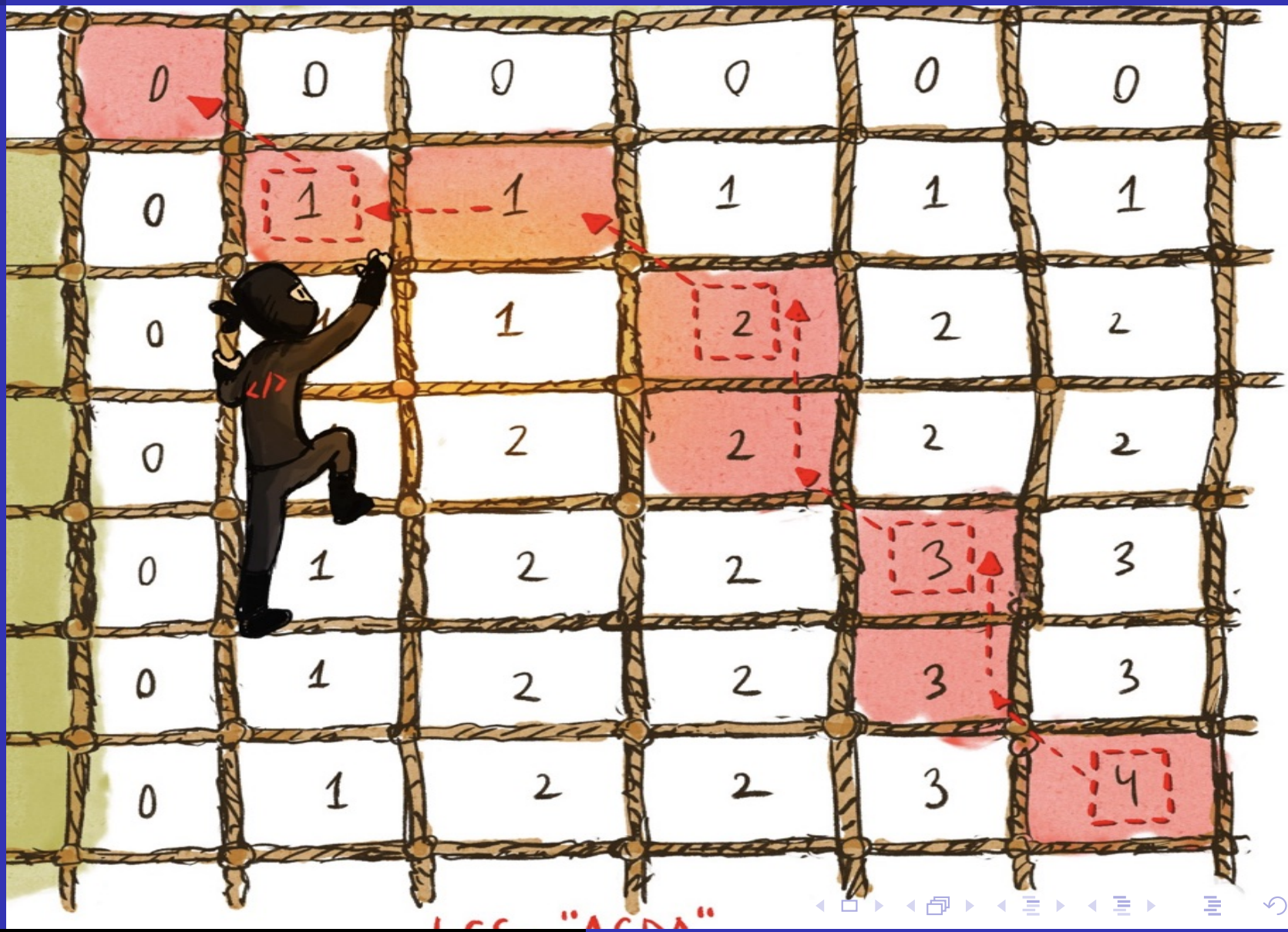
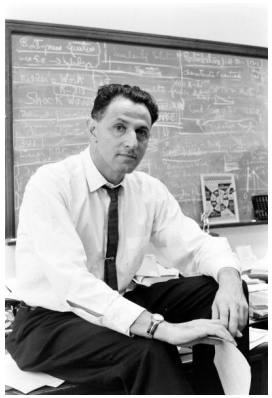
Dynamic Programming



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

Dynamic Programming works efficiently when:

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.

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.

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.

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, store 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. We solve recursively one subproblem

Difference with greedy

- Greedy problems have the greedy choice property: locally optimal choices lead to globally optimal solution. We solve recursively one subproblem
- 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

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).

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.

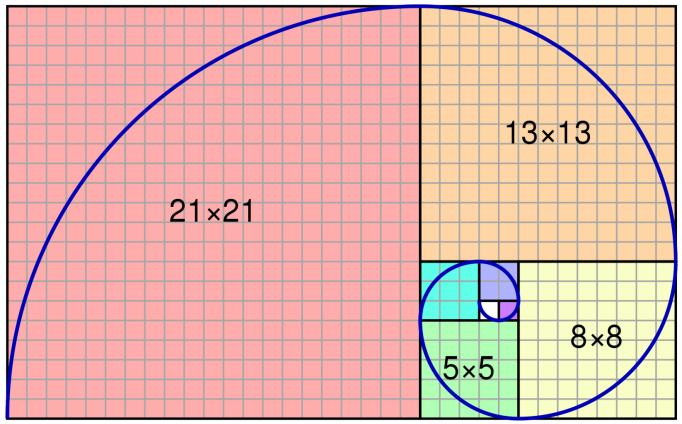
A first example: Fibonacci Recurrence.

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

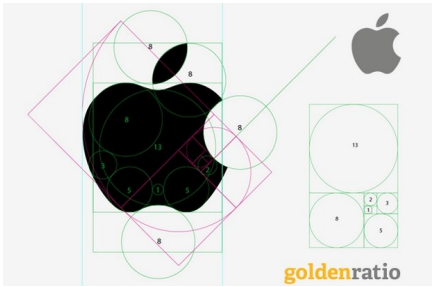


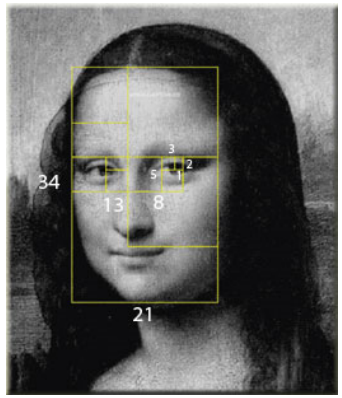
The golden ratio

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

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.

Computing the n-th Fibonacci number.

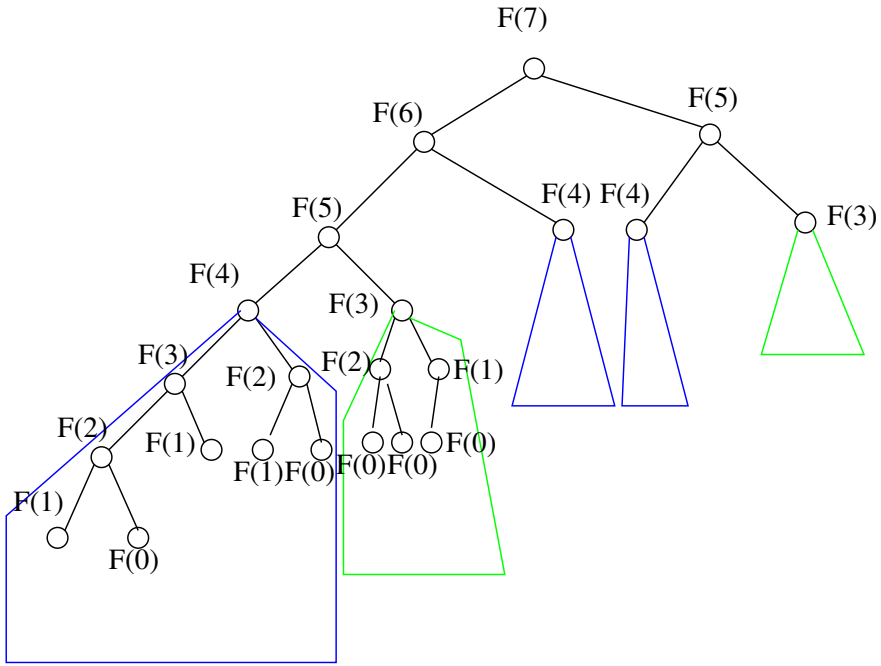
INPUT: $n \in \mathbb{N}$
QUESTION: Compute F_n.

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))

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

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 $F[i] \neq -1$ **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

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

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.

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)

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.

Computing the n-th Fibonacci number: cost

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

Computing the n-th Fibonacci number: cost

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

To get F_n the last algorithm needs $O(n)$ time and uses $O(1)$ space.

Computing the n-th Fibonacci number: cost

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

To get F_n the last algorithm needs $O(n)$ time and uses $O(1)$ space.

The initial recursive algorithm takes $O(1.6^n)$ time and uses $O(n)$ space

Do we have a polynomial time solution?

Computing the n-th Fibonacci number: cost

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

To get F_n the last algorithm needs $O(n)$ time and uses $O(1)$ space.

The initial recursive algorithm takes $O(1.6^n)$ time and uses $O(n)$ space

Do we have a polynomial time solution? NO

Computing the n-th Fibonacci number: cost

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

To get F_n the last algorithm needs $O(n)$ time and uses $O(1)$ space.

The initial recursive algorithm takes $O(1.6^n)$ time and uses $O(n)$ space

Do we have a polynomial time solution? NO the size of the input is $\log n$.

Computing the n-th Fibonacci number: cost

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

To get F_n the last algorithm needs $O(n)$ time and uses $O(1)$ space.

The initial recursive algorithm takes $O(1.6^n)$ time and uses $O(n)$ space

Do we have a polynomial time solution? NO the size of the input is $\log n$.
We use the term pseudopolynomial for algorithms whose running time is polynomial in the value of some numbers in the input.

Guideline to implement Dynamic Programming

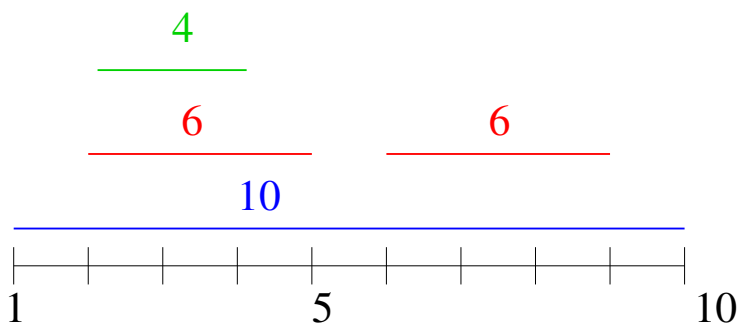
This first example of PD was easy, as the recurrence is given in the statement of the problem.

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, memoization/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 solution from optimal value.

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 S, then all activities that overlap with i cannot be in S.
  - i is not in S.

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.

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**$(S, C)\})$

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**$(S, C)\})$

The recursion tree have branching 2 and height $\leq n$, so size is $O(2^n)$.

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**$(S, C)\})$

The recursion tree have branching 2 and height $\leq n$, so size is $O(2^n)$.

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**$(S, C)\})$

The recursion tree have branching 2 and height $\leq n$, so size is $O(2^n)$.
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.

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., $f_1 \leq f_2 \leq \cdots \leq f_n$.

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., $f_1 \leq f_2 \leq \cdots \leq f_n$.

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 $(\max\{\textbf{WAS-2}(S \cup \{i\}, j), \textbf{WAS-2}(S, i-1)\})$

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., $f_1 \leq f_2 \leq \cdots \leq f_n$.

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., $f_1 \leq f_2 \leq \cdots \leq f_n$.

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? activities $j < k < i$ overlap with i any other that overlap with i also overlaps with j.

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., $f_1 \leq f_2 \leq \cdots \leq f_n$.

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? activities $j < k < i$ overlap with i any other that overlap with i also overlaps with j.

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., $f_1 \leq f_2 \leq \cdots \leq f_n$.

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? activities $j < k < i$ overlap with i any other that overlap with i also overlaps with j.

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

DP from WAS-2: a recurrence

- We need a $O(n \lg n)$ time for sorting.
- We have n activities with $f_1 \leq f_2 \leq \cdots \leq f_n$ and weights w_i, $1 \leq i \leq n$.

DP from WAS-2: a recurrence

- We need a $O(n \lg n)$ time for sorting.
- We have n activities with $f_1 \leq f_2 \leq \cdots \leq f_n$ and weights w_i, $1 \leq i \leq n$.
- Supproblems calls WAS-2(S, i)
  - S keeps track of the value of the solution
  - i defines de supproblem: W activity selection for activities $\{1, \ldots, i\}$, for $0 \leq i \leq n$.
  - $O(n)$ subproblems!

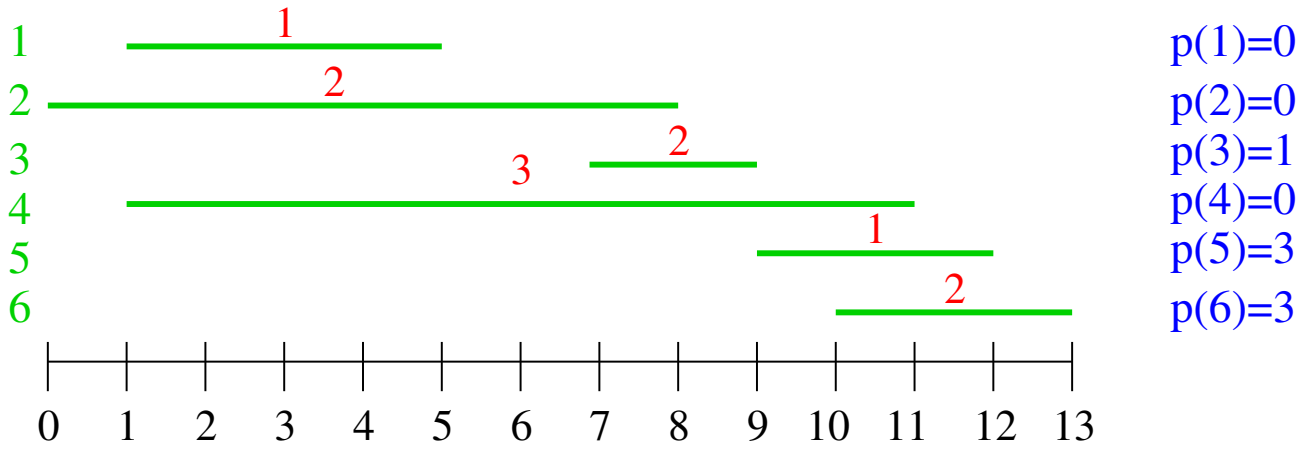
DP from WAS-2: a recurrence

- We need a $O(n \lg n)$ time for sorting.
- We have n activities with $f_1 \leq f_2 \leq \cdots \leq f_n$ and weights w_i, $1 \leq i \leq n$.
- Supproblems calls WAS-2(S, i)
  - S keeps track of the value of the solution
  - i defines de supproblem: W activity selection for activities $\{1, \ldots, i\}$, for $0 \leq i \leq n$.
  - $O(n)$ subproblems!
- 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).

DP from WAS-2: a recurrence

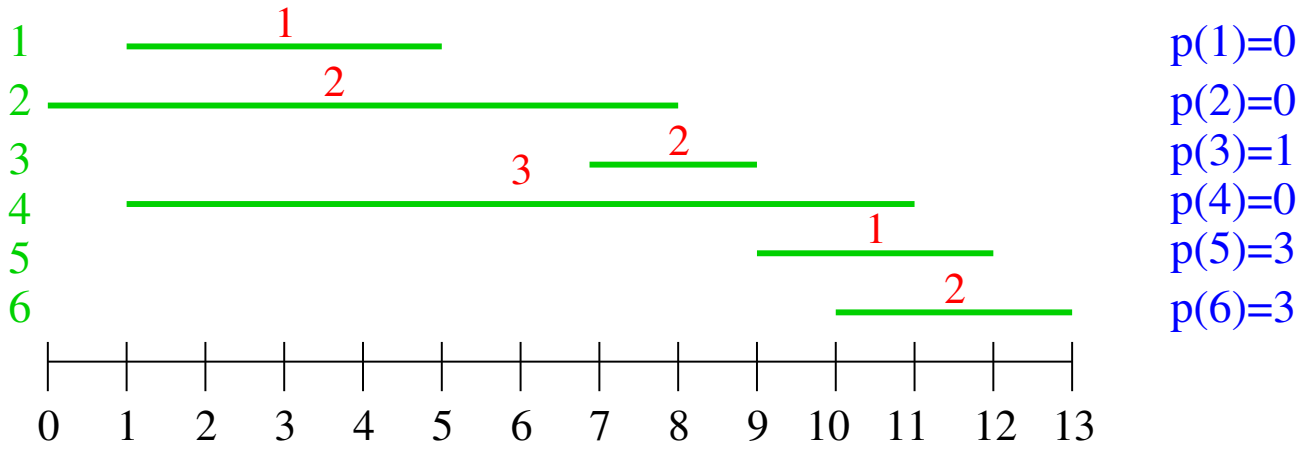
- We need a $O(n \lg n)$ time for sorting.
- We have n activities with $f_1 \leq f_2 \leq \cdots \leq f_n$ and weights w_i, $1 \leq i \leq n$.
- Supproblems calls WAS-2(S, i)
  - S keeps track of the value of the solution
  - i defines de supproblem: W activity selection for activities $\{1, \ldots, i\}$, for $0 \leq i \leq n$.
  - $O(n)$ subproblems!
- 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 an optimal solution O_j to the sub problem consisting of activities in the range 1 to j.

DP from WAS-2: a recurrence



Let Opt(j) be the value of an optimal solution O_j to the subproblem consisting of activities in the range 1 to j.
Reinterpreting WAS-2, we get

DP from WAS-2: a recurrence



Let Opt(j) be the value of an optimal solution O_j to the subproblem consisting of activities in the range 1 to j.
Reinterpreting WAS-2, we get

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

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,
- $O_j - \{j\}$ must be an optimal solution for $\{1, \ldots, p[j])\}$, otherwise then $O'_j = O_{p[j]} \cup \{j\}$ will be better (optimal substructure)

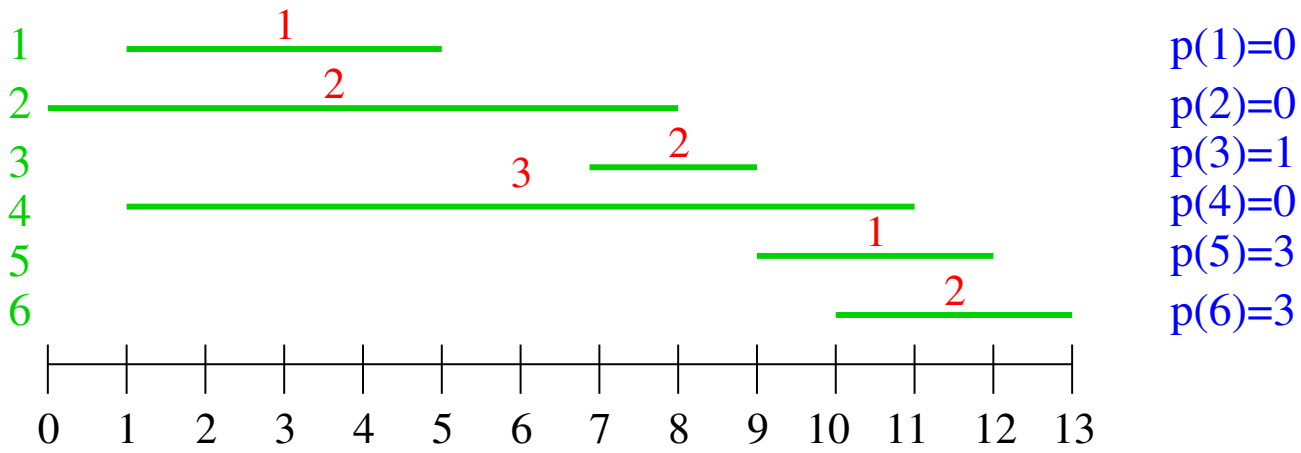
2.- If $j \notin O_j$: then O_j is an optimal solution to $\{1, \ldots, j-1)\}$.

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 tasks are 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
  $W[j] = \max(w_j + \textbf{R-Opt}(P[j])), \textbf{R-Opt}(j-1))$
  return $W[j]$

DP from WAS-2: Memoization

We assume that tasks are 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
  $W[j] = \max(w_j + \textbf{R-Opt}(P[j])), \textbf{R-Opt}(j-1))$
  return $W[j]$

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 tasks are 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 $\{1, \ldots, i\}$.

DP from WAS-2: Iterative

We assume that tasks are 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 $W[n]$

Time complexity: $O(n \lg n + n)$.
Notice: Both algorithms gave only the numerical max. weight
We have to keep more info to recover a solution form $W[n]$.

DP from WAS-2: Returning an optimal solution

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

DP from WAS-2: Returning an optimal solution

To get also the list of activities in an optimal solution, 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.
- Guided by optimal value, obtain an optimal solution .

0-1 Knapsack

(This example is from Section 6.4 in Dasgupta,Papadimritriou,Vazirani's book.)

0-1 Knapsack: Given as input a set of n items that can NOT be fractioned, item i has weight w_i and value v_i, and a maximum permissible weight W.

QUESTION: select a set of items S that maximize the profit.

Recall that we can NOT take fractions of items.

subproblems and and recurrence

Input: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Let $S \subseteq \{1, \ldots, n\}$ be an optimal solution to the problem The optimal benefit is $\sum_{i \in S} v_i$

subproblems and and recurrence

Input: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Let $S \subseteq \{1, \ldots, n\}$ be an optimal solution to the problem The optimal benefit is $\sum_{i \in S} v_i$
- With respect to the last item we have two cases:
  - $n \notin S$, then S is an optimal solution to the problem $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, W
  - $n \in S$, then $S - \{n\}$ is an optimal solution to the problem $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, $W - w_n$

subproblems and and recurrence

Input: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Let $S \subseteq \{1, \ldots, n\}$ be an optimal solution to the problem The optimal benefit is $\sum_{i \in S} v_i$
- With respect to the last item we have two cases:
  - $n \notin S$, then S is an optimal solution to the problem $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, W
  - $n \in S$, then $S - \{n\}$ is an optimal solution to the problem $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, $W - w_n$
- in both cases we get an optimal solution of a subproblem in which the last item is removed and in which the maximum weight can be W or a value smaller than W.

subproblems and and recurrence

Input: $(w_1, \ldots, w_n)$, $(v_1, \ldots, v_n)$, W.

- Let $S \subseteq \{1, \ldots, n\}$ be an optimal solution to the problem The optimal benefit is $\sum_{i \in S} v_i$
- With respect to the last item we have two cases:
  - $n \notin S$, then S is an optimal solution to the problem $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, W
  - $n \in S$, then $S - \{n\}$ is an optimal solution to the problem $(w_1, \ldots, w_{n-1})$, $(v_1, \ldots, v_{n-1})$, $W - w_n$
- in both cases we get an optimal solution of a subproblem in which the last item is removed and in which the maximum weight can be W or a value smaller than W.
- This identifies subproblems of the form $[i, x]$ that are knapsack instances in which the set of items is $\{1, \ldots, i\}$ and the maximum weight that can hold the knapsack is x.

Subproblems and recurrence

Let $v[i, x]$ be the maximum value (optimum) we can get from objects $\{1, 2, \ldots, i\}$ within total weight $\leq x$.

Subproblems and recurrence

Let $v[i, x]$ be the maximum value (optimum) we can get from objects $\{1, 2, \ldots, i\}$ within total weight $\leq x$.

To compute $v[i, x]$, the two possibilities we have considered give raise to the recurrence:

$$v[i, x] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ \max v[i-1, x-w_i] + v_i, v[i-1, x] & \text{otherwise} \end{cases}$$

DP algorithm: tabulating

Define a table $P[n+1, W+1]$ to hold optimal values for the corresponding subproblem.

Knapsack(i, x)
for $i = 0$ **to** n **do**
  $P[i, 0] = 0$
for $x = 1$ **to** W **do**
  $P[0, x] = 0$
for $i = 1$ **to** n **do**
  for $x = 1$ **to** W **do**
    $P[i, x] = \max\{P[i-1, x], P[i-1, x-w[i]] + v[i]\}$
return $P[n, W]$

The number of steps is $O(nW)$

DP algorithm: tabulating

Define a table $P[n+1, W+1]$ to hold optimal values for the corresponding subproblem.

Knapsack(i, x)
for $i = 0$ **to** n **do**
  $P[i, 0] = 0$
for $x = 1$ **to** W **do**
  $P[0, x] = 0$
for $i = 1$ **to** n **do**
  for $x = 1$ **to** W **do**
    $P[i, x] = \max\{P[i-1, x], P[i-1, x-w[i]] + v[i]\}$
return $P[n, W]$

The number of steps is $O(nW)$ which is

DP algorithm: tabulating

Define a table $P[n+1, W+1]$ to hold optimal values for the corresponding subproblem.

Knapsack(i, x)
for $i = 0$ **to** n **do**
  $P[i, 0] = 0$
for $x = 1$ **to** W **do**
  $P[0, x] = 0$
for $i = 1$ **to** n **do**
  for $x = 1$ **to** W **do**
    $P[i, x] = \max\{P[i-1, x], P[i-1, x-w[i]] + v[i]\}$
return $P[n, W]$

The number of steps is $O(nW)$ which is pseudopolynomial.

An example

i	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	28

$W = 11$.

		w										
		0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1	1	1	1	1
	2	0	1	6	7	7	7	7	7	7	7	7	7
I	3	0	1	6	7	7	18	19	24	25	25	25	25
	4	0	1	6	7	7	18	22	23	28	29	29	40
	5	0	1	6	7	7	18	22	28	29	34	35	40

For instance, $v[4, 10] = \max\{v[3, 10], v[3, 10 - 6] + 22\} = \max\{25, 7 + 22\} = 29$.
$v[5, 11] = \max\{v[4, 11], v[4, 11 - 7] + 28\} = \max\{40, 4 + 28\} = 40$.

Recovering the solution

To compute the actual subset $S \subseteq I$ that is the solution, we modify the algorithm to compute also a Boolean table $K[n+1, W+1]$, so that $K[i, x]$ is 1 when the max is attained in the second alternative ($i \in S$), 0 otherwise.

Recovering the solution

To compute the actual subset $S \subseteq I$ that is the solution, we modify the algorithm to compute also a Boolean table $K[n+1, W+1]$, so that $K[i,x]$ is 1 when the max is attained in the second alternative ($i \in S$), 0 otherwise.

Knapsack(i,x)
for $i = 0$ **to** n **do**
  $P[i,0] = 0$; $K[i,0] = 0$
for $x = 1$ **to** W **do**
  $P[0,x] = 0$; $K[0,x] = 0$
for $i = 1$ **to** n **do**
  for $x = 1$ **to** W **do**
    if $P[i-1,x] \geq$
    $P[i-1, x-w[i]] + v[i]$ **then**
      $P[i,x] = P[i-1,x]$;
      $K[i,x] = 0$
    else
      $P[i,x] =$
      $P[i-1, x-w[i]] + v[i]$;
      $K[i,x] = 1$
return $P[n,W]$

Complexity: $O(nW)$

An example

	0	1	2	3	4	5	6	7	8	9	10	11
0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
1	0 0	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1
2	0 0	1 0	6 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1
3	0 0	1 0	6 0	7 0	7 0	18 1	19 1	24 1	25 1	25 1	25 1	25 1
4	0 0	1 0	6 0	7 0	7 0	18 1	22 1	23 1	28 1	29 1	29 1	40 1
5	0 0	1 0	6 0	7 0	7 0	18 0	22 0	28 1	29 1	34 1	35 1	40 0

Recovering the solution

- To compute an optimal solution $S \subseteq I$, we use K to trace backwards the elements in the solution.
- $K[i, x]$ is 1 when the max is attained in the second alternative: $i \in S$.

Recovering the solution

- To compute an optimal solution $S \subseteq I$, we use K to trace backwards the elements in the solution.
- $K[i, x]$ is 1 when the max is attained in the second alternative: $i \in S$.

$x = W$, $S = \emptyset$
for $i = n$ **downto** 1 **do**
  if $K[i, x] = 1$ **then**
    $S = S \cup \{i\}$
    $x = x - w_i$
Output S

Complexity: $O(nW)$

An example

i	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	28

$W = 11$.

An example

i	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	28

$W = 11$.

	0	1	2	3	4	5	6	7	8	9	10	11
0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
1	0 0	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1	1 1
2	0 0	1 0	6 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1	7 1
3	0 0	1 0	6 0	7 0	7 0	18 1	19 1	24 1	25 1	25 1	25 1	25 1
4	0 0	1 0	6 0	7 0	7 0	18 1	22 1	23 1	28 1	29 1	29 1	40 1
5	0 0	1 0	6 0	7 0	7 0	18 0	22 0	28 1	29 1	34 1	35 1	40 0

$K[5, 11] \to K[4, 11] \to K[3, 5] \to K[2, 0]$. So $S = \{4, 3\}$

Complexity

The 0-1 KNAPSACK is NP-complete.

- 0-1 KNAPSACK, has complexity $O(nW)$, and its length is $O(n \lg M)$ taking $M = \max\{W, \max_i w_i, \max_i v_i\}$.
- If W requires k bits, the cost and space of the algorithm is $n2^k$, exponential in the length W. However the DP algorithm works fine when $W = \Theta(n)$, here $k = O(\log n)$.
- Consider the unary knapsack problem, where all integers are coded in unary (7=1111111). In this case, the complexity of the DP algorithm is polynomial on the size, i.e., UNARY KNAPSACK $\in$P.