Advanced Models and Methods in Operations Research
Dynamic programming

Florian Fontan

November 7, 2023

Table of contents

Introduction

The partition problem and the subset sum problem

The knapsack problem

The single-night star observation scheduling problem

Dynamic programming as a tree search

Conclusion

Me

- Former student of ORCO (2015–2016)
- PhD in Operations Research
- Engineer at Artelys: `https://www.artelys.com/`
  - Artelys is an independent company specialized in optimization, decision support and modeling
  - Design and implementation of optimization algorithms for industrial clients
  - Development for the NLP/MINLP solver Artelys Knitro `https://www.artelys.com/solvers/knitro/`
- My GitHub: `https://github.com/fontanf/`

Organization

- 4 classes with me:
  - Dynamic programming
  - Heuristic tree search
  - Column generation heuristics
  - Project presentations
- 1.5 hours lecture / 1.5 hours practical training
- Goal of the classes: understanding the theory of the methods and being able to implement them to solve practical problems

Organization

- Evaluation:
  - Not in the final exam
  - Project by groups of 3 or 4
  - Implementation of the algorithms studied in the classes
  - First deadline with feedbacks
  - Second deadline with final grade
  - Please do not put your code on a public repository
  - You can keep it on a private repository. It can be valuable if you decide to apply for a company some day
- All materials (slides, projects...) are online
  `https://github.com/fontanf/teaching`
- E-mail: dev@florian-fontan.fr

Table of contents

Introduction

The partition problem and the subset sum problem

The knapsack problem

The single-night star observation scheduling problem

Dynamic programming as a tree search

Conclusion

Problem definition

Partition problem

- Instance: n items with weight w_j, $j = 1, \ldots, n$.
- Question: is it possible to partition the set of items into two subsets of equal weights?

Problem definition

Partition problem

- Instance: n items with weight w_j, $j = 1, \ldots, n$.
- Question: is it possible to partition the set of items into two subsets of equal weights?

We consider a slightly more general variant:

Subset sum problem (decision version)

- Instance:
  - n items with weight w_j, $j = 1, \ldots, n$
  - a capacity C
- Question: is there a subset of items with total weight C?

Problem definition

Partition problem

- Instance: n items with weight w_j, $j = 1, \ldots, n$.
- Question: is it possible to partition the set of items into two subsets of equal weights?

We consider a slightly more general variant:

Subset sum problem (decision version)

- Instance:
  - n items with weight w_j, $j = 1, \ldots, n$
  - a capacity C
- Question: is there a subset of items with total weight C?

Link between the partition problem and the subset sum problem?

Problem definition

Partition problem

- Instance: n items with weight w_j, $j = 1, \ldots, n$.
- Question: is it possible to partition the set of items into two subsets of equal weights?

We consider a slightly more general variant:

Subset sum problem (decision version)

- Instance:
  - n items with weight w_j, $j = 1, \ldots, n$
  - a capacity C
- Question: is there a subset of items with total weight C?

Link between the partition problem and the subset sum problem?
The partition problem is a subset sum problem with capacity $C = \frac{1}{2} \sum_{j=1}^{n} w_j$.

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define:

$$F(j, c) = \begin{cases} \text{True} & \text{if among items } 1, \ldots, j, \text{ there exists} \\ & \text{a subset of items with total weight } c \\ \text{False} & \text{otherwise} \end{cases}$$

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define:

$$F(j, c) = \begin{cases} \text{True} & \text{if among items } 1, \ldots, j, \text{ there exists} \\ & \text{a subset of items with total weight } c \\ \text{False} & \text{otherwise} \end{cases}$$

What is the value of

- $F(0, 0)$?

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define:

$$F(j, c) = \begin{cases} \text{True} & \text{if among items } 1, \ldots, j, \text{ there exists} \\ & \text{a subset of items with total weight } c \\ \text{False} & \text{otherwise} \end{cases}$$

What is the value of

- $F(0, 0)$? True

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define:

$$F(j, c) = \begin{cases} \text{True} & \text{if among items } 1, \ldots, j, \text{ there exists} \\ & \text{a subset of items with total weight } c \\ \text{False} & \text{otherwise} \end{cases}$$

What is the value of

- $F(0, 0)$? True
- $F(0, c)$?

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define:

$$F(j, c) = \begin{cases} \text{True} & \text{if among items } 1, \ldots, j, \text{ there exists} \\ & \text{a subset of items with total weight } c \\ \text{False} & \text{otherwise} \end{cases}$$

What is the value of

- $F(0, 0)$? True
- $F(0, c)$? True for $c = 0$, False otherwise

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define:

$$F(j, c) = \begin{cases} \text{True} & \text{if among items } 1, \ldots, j, \text{ there exists} \\ & \text{a subset of items with total weight } c \\ \text{False} & \text{otherwise} \end{cases}$$

What is the value of

- $F(0, 0)$? True
- $F(0, c)$? True for $c = 0$, False otherwise
- $F(j, 0)$?

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define:

$$F(j, c) = \begin{cases} \text{True} & \text{if among items } 1, \ldots, j, \text{ there exists} \\ & \text{a subset of items with total weight } c \\ \text{False} & \text{otherwise} \end{cases}$$

What is the value of

- $F(0, 0)$? True
- $F(0, c)$? True for $c = 0$, False otherwise
- $F(j, 0)$? True

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define:

$$F(j, c) = \begin{cases} \text{True} & \text{if among items } 1, \ldots, j, \text{ there exists} \\ & \text{a subset of items with total weight } c \\ \text{False} & \text{otherwise} \end{cases}$$

What is the value of

- $F(0, 0)$? True
- $F(0, c)$? True for $c = 0$, False otherwise
- $F(j, 0)$? True

What is the relation between the subset sum problem and F?

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define:

$$F(j, c) = \begin{cases} \text{True} & \text{if among items } 1, \ldots, j, \text{ there exists} \\ & \text{a subset of items with total weight } c \\ \text{False} & \text{otherwise} \end{cases}$$

What is the value of

- $F(0, 0)$? True
- $F(0, c)$? True for $c = 0$, False otherwise
- $F(j, 0)$? True

What is the relation between the subset sum problem and F?
The subset sum problem is equivalent to determining the value of $F(n, C)$.

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} \text{True} & \text{if } j = 0 \text{ and } c = 0 \\ \text{False} & \text{if } j = 0 \text{ and } c \neq 0 \\ F(j-1,c) & \textit{if } j \neq 0 \text{ and } c < w_j \\ F(j-1,c) & \text{otherwise} \\ \text{or } F(j-1,c-w_j) & \end{cases}$$

Divide-and-conquer

We can now solve the subset sum problem with a divide-and-conquer algorithm:

function F(w, j, c)
    if $j == 0$ **then**
        if $c == 0$ **then**
            return *True*
        else
            return *False*
    else if $c < w_j$ **then**
        return $F(j-1, c)$
    else
        return $F(j-1, c)$ or $F(j-1, c-w[j])$
function subsetsum(w, C)
    return $F(w, n, C)$

Divide-and-conquer

We can now solve the subset sum problem with a divide-and-conquer algorithm:

function F(w, j, c)
    if $j == 0$ **then**
        if $c == 0$ **then**
            return *True*
        else
            return *False*
    else if $c < w_j$ **then**
        return $F(j-1, c)$
    else
        return $F(j-1, c)$ or $F(j-1, c-w[j])$
function subsetsum(w, C)
    return $F(w, n, C)$

- Time complexity?

Divide-and-conquer

We can now solve the subset sum problem with a divide-and-conquer algorithm:

function F(w, j, c)
    if $j == 0$ **then**
        if $c == 0$ **then**
            return *True*
        else
            return *False*
    else if $c < w_j$ **then**
        return $F(j-1, c)$
    else
        return $F(j-1, c)$ or $F(j-1, c-w[j])$
function subsetsum(w, C)
    return $F(w, n, C)$

- Time complexity? $O(2^n)$

Divide-and-conquer

We can now solve the subset sum problem with a divide-and-conquer algorithm:

function F(w, j, c)
    if $j == 0$ **then**
        if $c == 0$ **then**
            return *True*
        else
            return *False*
    else if $c < w_j$ **then**
        return $F(j-1, c)$
    else
        return $F(j-1, c)$ or $F(j-1, c-w[j])$
function subsetsum(w, C)
    return $F(w, n, C)$

- Time complexity? $O(2^n)$
- Space complexity?

Divide-and-conquer

We can now solve the subset sum problem with a divide-and-conquer algorithm:

function F(w, j, c)
    if $j == 0$ **then**
        if $c == 0$ **then**
            return *True*
        else
            return *False*
    else if $c < w_j$ **then**
        return $F(j-1, c)$
    else
        return $F(j-1, c)$ or $F(j-1, c-w[j])$
function subsetsum(w, C)
    return $F(w, n, C)$

- Time complexity? $O(2^n)$
- Space complexity? $O(n)$

Divide-and-conquer

We can now solve the subset sum problem with a divide-and-conquer algorithm:

function F(w, j, c)
    if $j == 0$ **then**
        if $c == 0$ **then**
            return *True*
        else
            return *False*
    else if $c < w_j$ **then**
        return $F(j-1, c)$
    else
        return $F(j-1, c)$ or $F(j-1, c-w[j])$
function subsetsum(w, C)
    return $F(w, n, C)$

- Time complexity? $O(2^n)$
- Space complexity? $O(n)$

Is there a way to improve the time complexity?

Divide-and-conquer example

Consider an instance I of the subset sum problem with

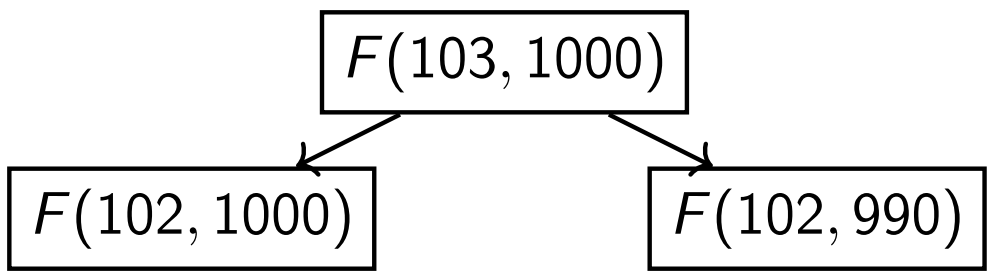
- $n = 103$
- $C = 1000$
- $w_{101} = 30,\ w_{102} = 20,\ w_{103} = 10.$

$$\boxed{F(103, 1000)}$$

Divide-and-conquer example

Consider an instance I of the subset sum problem with

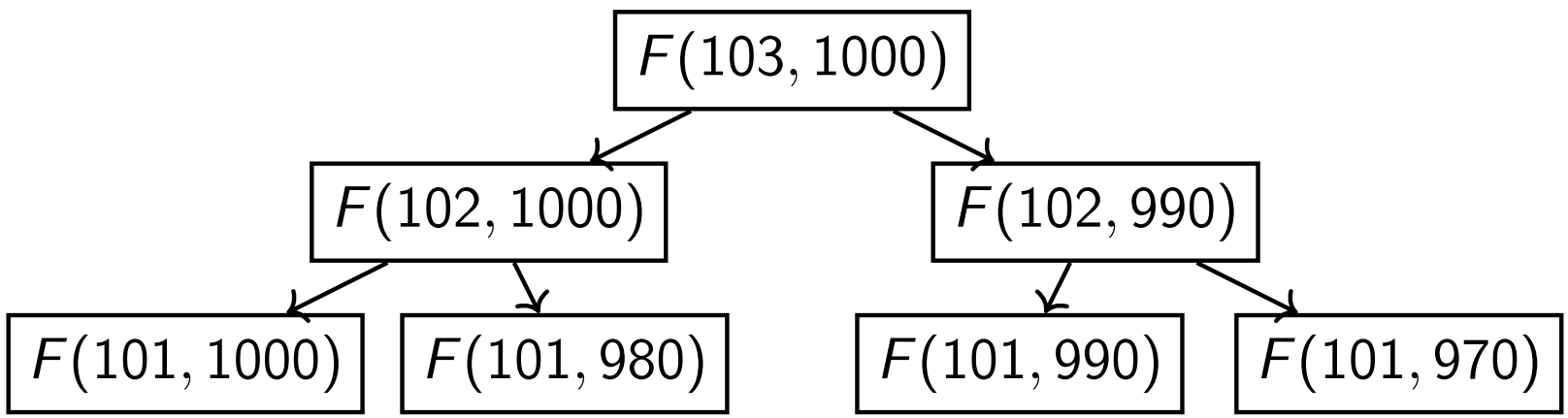
- $n = 103$
- $C = 1000$
- $w_{101} = 30,\ w_{102} = 20,\ w_{103} = 10.$



Divide-and-conquer example

Consider an instance I of the subset sum problem with

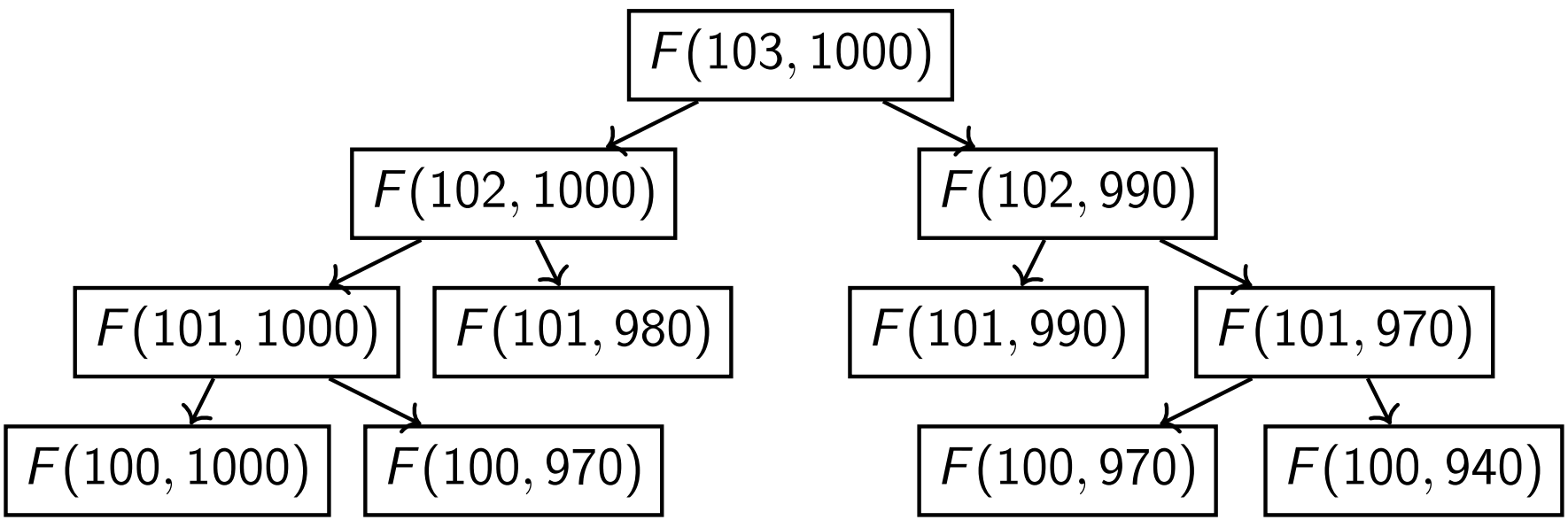
- $n = 103$
- $C = 1000$
- $w_{101} = 30,\ w_{102} = 20,\ w_{103} = 10.$



Divide-and-conquer example

Consider an instance I of the subset sum problem with

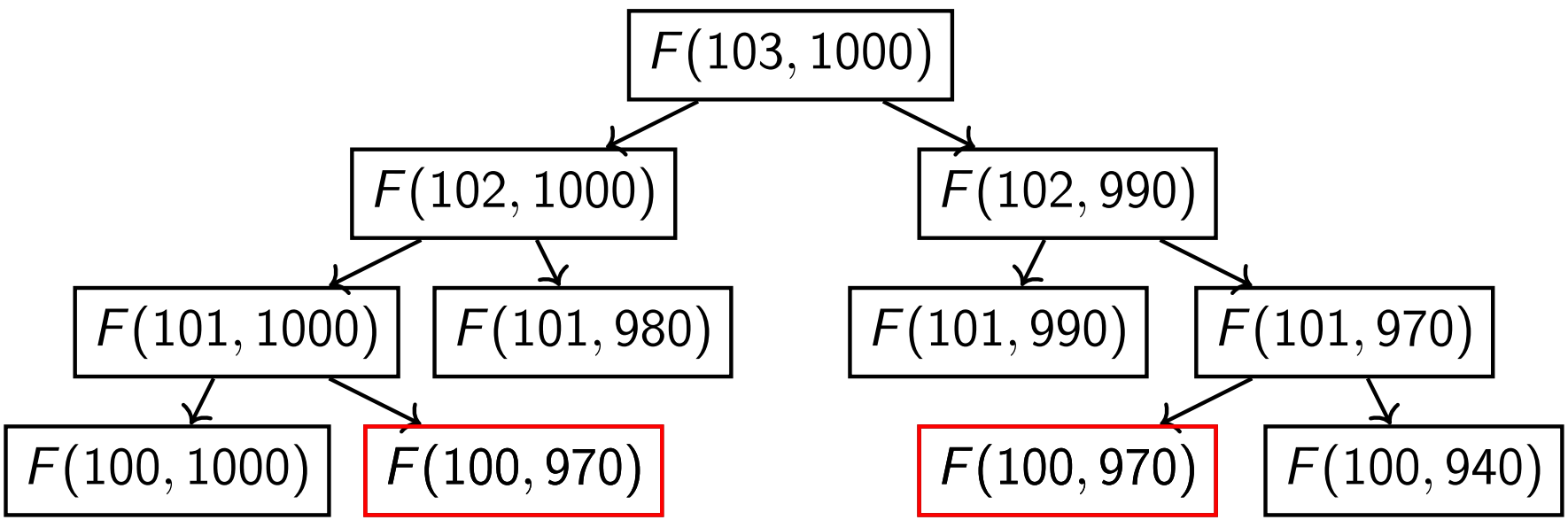
- $n = 103$
- $C = 1000$
- $w_{101} = 30,\ w_{102} = 20,\ w_{103} = 10.$



Divide-and-conquer example

Consider an instance I of the subset sum problem with

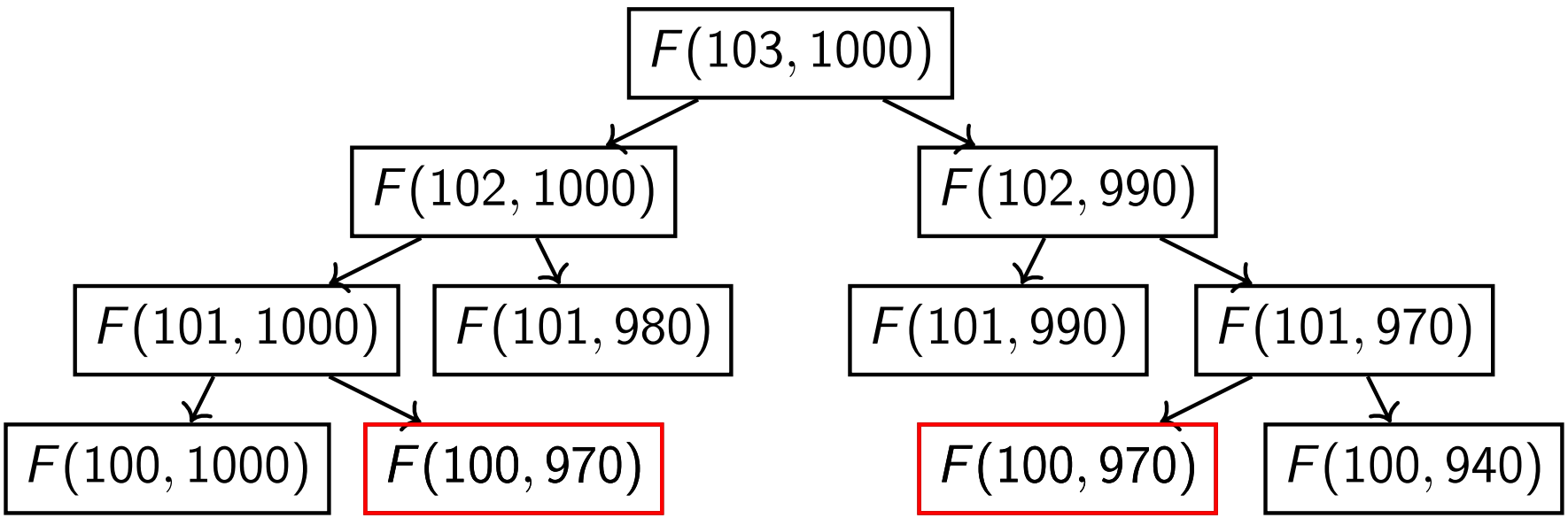
- $n = 103$
- $C = 1000$
- $w_{101} = 30,\ w_{102} = 20,\ w_{103} = 10.$



Divide-and-conquer example

Consider an instance I of the subset sum problem with

- $n = 103$
- $C = 1000$
- $w_{101} = 30,\ w_{102} = 20,\ w_{103} = 10.$



The same subproblems might be solved multiple times!

Dynamic programming: recursive implementation (top-down)

Same algorithm as before, but now we store the results of the subproblems to avoid solving multiple times the same subproblems:

procedure F(w, T, j, c)
    if $T[j][c] == \mathrm{NULL}$ **then**
        if $j == 0$ **then**
            if $c == 0$ **then**
                $T[j][c] \leftarrow$ True
            else
                $T[j][c] \leftarrow$ False
        else if $c < w[j]$ **then**
            $T[j][c] \leftarrow F(j-1, c)$
        else
            $T[j][c] \leftarrow F(j-1, c)$ or $F(j-1, c-w[j])$
    return $T[j][c]$
procedure subsetsum(w, C)
    $T \leftarrow$ array of size $(n+1) \times (C+1)$ initialized at NULL
    return $F(w, T, n, C)$

Dynamic programming: recursive implementation (top-down)

Same algorithm as before, but now we store the results of the subproblems to avoid solving multiple times the same subproblems:

```
procedure F(w, T, j, c)
    if T[j][c] == NULL then
        if j == 0 then
            if c == 0 then
                T[j][c] ← True
            else
                T[j][c] ← False
        else if c < w[j] then
            T[j][c] ← F(j − 1, c)
        else
            T[j][c] ← F(j − 1, c) or F(j − 1, c − w[j])
    return T[j][c]
procedure subsetsum(w, C)
    T ← array of size (n+1) × (C+1) initialized at NULL
    return F(w, T, n, C)
```

- Time complexity?

Dynamic programming: recursive implementation (top-down)

Same algorithm as before, but now we store the results of the subproblems to avoid solving multiple times the same subproblems:

procedure F(w, T, j, c)
  if $T[j][c] == \text{NULL}$ **then**
    if $j == 0$ **then**
      if $c == 0$ **then**
        $T[j][c] \leftarrow$ True
      else
        $T[j][c] \leftarrow$ False
    else if $c < w[j]$ **then**
      $T[j][c] \leftarrow F(j-1, c)$
    else
      $T[j][c] \leftarrow F(j-1, c)$ or $F(j-1, c-w[j])$
  return $T[j][c]$
procedure subsetsum(w, C)
  $T \leftarrow$ array of size $(n+1) \times (C+1)$ initialized at NULL
  return $F(w, T, n, C)$

- Time complexity? $O(nC)$

Dynamic programming: recursive implementation (top-down)

Same algorithm as before, but now we store the results of the subproblems to avoid solving multiple times the same subproblems:

procedure F(w, T, j, c)
  if $T[j][c] == \text{NULL}$ **then**
    if $j == 0$ **then**
      if $c == 0$ **then**
        $T[j][c] \leftarrow$ True
      else
        $T[j][c] \leftarrow$ False
    else if $c < w[j]$ **then**
      $T[j][c] \leftarrow F(j-1, c)$
    else
      $T[j][c] \leftarrow F(j-1, c)$ or $F(j-1, c-w[j])$
  return $T[j][c]$
procedure subsetsum(w, C)
  $T \leftarrow$ array of size $(n+1) \times (C+1)$ initialized at NULL
  return $F(w, T, n, C)$

- Time complexity? $O(nC)$
- Space complexity?

Dynamic programming: recursive implementation (top-down)

Same algorithm as before, but now we store the results of the subproblems to avoid solving multiple times the same subproblems:

procedure F(w, T, j, c)
  if $T[j][c] == \text{NULL}$ **then**
    if $j == 0$ **then**
      if $c == 0$ **then**
        $T[j][c] \leftarrow$ True
      else
        $T[j][c] \leftarrow$ False
    else if $c < w[j]$ **then**
      $T[j][c] \leftarrow F(j-1, c)$
    else
      $T[j][c] \leftarrow F(j-1, c)$ or $F(j-1, c-w[j])$
  return $T[j][c]$
procedure subsetsum(w, C)
  $T \leftarrow$ array of size $(n+1) \times (C+1)$ initialized at NULL
  return $F(w, T, n, C)$

- Time complexity? $O(nC)$
- Space complexity? $O(nC)$

Dynamic programming

Solving a problem recursively and storing the results of the subproblems to avoid recomputing them multiple times.

Dynamic programming: iterative implementation (bottom-up)

procedure subsetsum(w, C)
    $T \leftarrow$ array of size $(n+1) \times (C+1)$ initialized at NULL
    $T[0][0] \leftarrow$ True
    for $c = 1, \ldots, C$ **do**
        $T[0][c] \leftarrow$ False
    for $j = 1, \ldots, n$ **do**
        for $c = 0, \ldots, w[j] - 1$ **do**
            $T[j][c] \leftarrow T[j-1][c]$
        for $c = w[j], \ldots, C$ **do**
            $T[j][c] \leftarrow T[j-1][c]$ or $T[j-1][c-w[j]]$
    return $T[n, C]$

Dynamic programming: iterative implementation (bottom-up)

procedure subsetsum(w, C)
  $T \leftarrow$ array of size $(n+1) \times (C+1)$ initialized at NULL
  $T[0][0] \leftarrow$ True
  for $c = 1, \ldots, C$ **do**
    $T[0][c] \leftarrow$ False
  for $j = 1, \ldots, n$ **do**
    for $c = 0, \ldots, w[j] - 1$ **do**
      $T[j][c] \leftarrow T[j-1][c]$
    for $c = w[j], \ldots, C$ **do**
      $T[j][c] \leftarrow T[j-1][c]$ or $T[j-1][c-w[j]]$
  return $T[n, C]$

- Time complexity?

Dynamic programming: iterative implementation (bottom-up)

procedure subsetsum(w, C)
  $T \leftarrow$ array of size $(n+1) \times (C+1)$ initialized at NULL
  $T[0][0] \leftarrow$ True
  for $c = 1, \ldots, C$ **do**
    $T[0][c] \leftarrow$ False
  for $j = 1, \ldots, n$ **do**
    for $c = 0, \ldots, w[j] - 1$ **do**
      $T[j][c] \leftarrow T[j-1][c]$
    for $c = w[j], \ldots, C$ **do**
      $T[j][c] \leftarrow T[j-1][c]$ or $T[j-1][c-w[j]]$
  return $T[n, C]$

- Time complexity? $O(nC)$

Dynamic programming: iterative implementation (bottom-up)

procedure subsetsum(w, C)
  $T \leftarrow$ array of size $(n+1) \times (C+1)$ initialized at NULL
  $T[0][0] \leftarrow$ True
  for $c = 1, \ldots, C$ **do**
    $T[0][c] \leftarrow$ False
  for $j = 1, \ldots, n$ **do**
    for $c = 0, \ldots, w[j] - 1$ **do**
      $T[j][c] \leftarrow T[j-1][c]$
    for $c = w[j], \ldots, C$ **do**
      $T[j][c] \leftarrow T[j-1][c]$ or $T[j-1][c-w[j]]$
  return $T[n, C]$

- Time complexity? $O(nC)$
- Space complexity?

Dynamic programming: iterative implementation (bottom-up)

procedure subsetsum(w, C)
  $T \leftarrow$ array of size $(n+1) \times (C+1)$ initialized at NULL
  $T[0][0] \leftarrow$ True
  for $c = 1, \ldots, C$ **do**
    $T[0][c] \leftarrow$ False
  for $j = 1, \ldots, n$ **do**
    for $c = 0, \ldots, w[j] - 1$ **do**
      $T[j][c] \leftarrow T[j-1][c]$
    for $c = w[j], \ldots, C$ **do**
      $T[j][c] \leftarrow T[j-1][c]$ or $T[j-1][c-w[j]]$
  return $T[n, C]$

- Time complexity? $O(nC)$
- Space complexity? $O(nC)$

Dynamic programming: iterative implementation (bottom-up)

procedure subsetsum(w, C)
  $T \leftarrow$ array of size $(n+1) \times (C+1)$ initialized at NULL
  $T[0][0] \leftarrow$ True
  for $c = 1, \ldots, C$ **do**
    $T[0][c] \leftarrow$ False
  for $j = 1, \ldots, n$ **do**
    for $c = 0, \ldots, w[j] - 1$ **do**
      $T[j][c] \leftarrow T[j-1][c]$
    for $c = w[j], \ldots, C$ **do**
      $T[j][c] \leftarrow T[j-1][c]$ or $T[j-1][c-w[j]]$
  return $T[n, C]$

- Time complexity? $O(nC)$
- Space complexity? $O(nC)$
- In practice, 10 times faster than the recursive implementation

Dynamic programming example

Instance:

- $n = 5,\ w = \{4, 11, 6, 8, 7\}$
- $C = 17$

Dynamic programming example

Instance:

- $n = 5,\ w = \{4, 11, 6, 8, 7\}$
- $C = 17$

Reminder:

$$F(j,c) = \begin{cases} \text{True} & \text{if } j = 0 \text{ and } c = 0 \\ \text{False} & \text{if } j = 0 \text{ and } c \neq 0 \\ F(j-1,c) & \textit{if } j \neq 0 \text{ and } c < w_j \\ F(j-1,c) & \text{otherwise} \\ \text{or } F(j-1,c-w_j) & \end{cases}$$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Dynamic programming example

Instance:

- $n = 5,\ w = \{4, 11, 6, 8, 7\}$
- $C = 17$

Reminder:

$$F(j, c) = \begin{cases} \text{True} & \text{if } j = 0 \text{ and } c = 0 \\ \text{False} & \text{if } j = 0 \text{ and } c \neq 0 \\ F(j-1, c) & \textit{if } j \neq 0 \text{ and } c < w_j \\ F(j-1, c) & \text{otherwise} \\ \text{or } F(j-1, c-w_j) & \end{cases}$$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

Dynamic programming example

Instance:

- $n = 5,\ w = \{4, 11, 6, 8, 7\}$
- $C = 17$

Reminder:

$$F(j,c) = \begin{cases} \text{True} & \text{if } j = 0 \text{ and } c = 0 \\ \text{False} & \text{if } j = 0 \text{ and } c \neq 0 \\ F(j-1,c) & \textit{if } j \neq 0 \text{ and } c < w_j \\ F(j-1,c) & \text{otherwise} \\ \text{or } F(j-1,c-w_j) & \end{cases}$$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F

Dynamic programming example

Instance:

- $n = 5,\ w = \{4, 11, 6, 8, 7\}$
- $C = 17$

Reminder:

$$F(j,c) = \begin{cases} \text{True} & \text{if } j = 0 \text{ and } c = 0 \\ \text{False} & \text{if } j = 0 \text{ and } c \neq 0 \\ F(j-1,c) & \textit{if } j \neq 0 \text{ and } c < w_j \\ F(j-1,c) & \text{otherwise} \\ \text{or } F(j-1,c-w_j) & \end{cases}$$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F

Dynamic programming example

Instance:

- $n = 5,\ w = \{4, 11, 6, 8, 7\}$
- $C = 17$

Reminder:

$$F(j,c) = \begin{cases} \text{True} & \text{if } j = 0 \text{ and } c = 0 \\ \text{False} & \text{if } j = 0 \text{ and } c \neq 0 \\ F(j-1,c) & \textit{if } j \neq 0 \text{ and } c < w_j \\ F(j-1,c) & \text{otherwise} \\ \text{or } F(j-1,c-w_j) & \end{cases}$$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T

Dynamic programming example

Instance:

- $n = 5,\ w = \{4, 11, 6, 8, 7\}$
- $C = 17$

Reminder:

$$F(j,c) = \begin{cases} \text{True} & \text{if } j = 0 \text{ and } c = 0 \\ \text{False} & \text{if } j = 0 \text{ and } c \neq 0 \\ F(j-1,c) & \textit{if } j \neq 0 \text{ and } c < w_j \\ F(j-1,c) & \text{otherwise} \\ \text{or } F(j-1,c-w_j) & \end{cases}$$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T

Dynamic programming example

Instance:

- $n = 5,\ w = \{4, 11, 6, 8, 7\}$
- $C = 17$

Reminder:

$$F(j,c) = \begin{cases} \text{True} & \text{if } j = 0 \text{ and } c = 0 \\ \text{False} & \text{if } j = 0 \text{ and } c \neq 0 \\ F(j-1,c) & \textit{if } j \neq 0 \text{ and } c < w_j \\ F(j-1,c) & \text{otherwise} \\ \text{or } F(j-1,c-w_j) & \end{cases}$$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.
$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

Retrieving the solution with backtracking

The algorithms presented in the previous slides only return True or False. In case the answer is True, we would also like to get a solution, *i.e.* a subset of items with total weight C.

$w = \{4, 11, 6, 8, 7\}$

j / c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	T	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	T	F	F	F	T	F	F	F	F	F	F	F	F	F	F	F	F	F
2	T	F	F	F	T	F	F	F	F	F	F	T	F	F	F	T	F	F
3	T	F	F	F	T	F	T	F	F	F	T	T	F	F	F	T	F	T
4	T	F	F	F	T	F	T	F	T	F	T	T	T	F	T	T	F	T
5	T	F	F	F	T	F	T	T	T	F	T	T	T	T	T	T	F	T

$S = \{5, 3, 1\}$

Retrieving the solution with backtracking

```
function SSPbacktracking(w, C, T)
    S ← {}
    c ← C
    i ← n
    while i > 0 do
        if not T[i − 1][c] then
            S ← S ∪ {i}
            c ← c − w[j]
        i ← i − 1
    return S
```

Going further

- Write an algorithm computing $F(n, C)$ which only keeps two lines of the array in memory (spatial complexity $O(C)$)
- Write an algorithm computing $F(n, C)$ which only keeps a single line of the array in memory.
- How to return a solution when keeping only a single line in memory?
  - if the array is stored as an array of integers
  - if the array is stored as an array of bits

Table of contents

Introduction

The partition problem and the subset sum problem

The knapsack problem

The single-night star observation scheduling problem

Dynamic programming as a tree search

Conclusion

Problem definition

Knapsack problem

- Instance:
  - n items with weight w_j and profit p_j, $j = 1, \ldots, n$
  - a capacity C
- Problem: find a subset of items such that the total weight of the subset is less than or equal to C.
- Objective: maximize the total profit of the selected items.

Problem definition

Knapsack problem

- Instance:
  - n items with weight w_j and profit p_j, $j = 1, \ldots, n$
  - a capacity C
- Problem: find a subset of items such that the total weight of the subset is less than or equal to C.
- Objective: maximize the total profit of the selected items.

Link between the subset sum problem and the knapsack problem?

Problem definition

Knapsack problem

- Instance:
  - n items with weight w_j and profit p_j, $j = 1, \ldots, n$
  - a capacity C
- Problem: find a subset of items such that the total weight of the subset is less than or equal to C.
- Objective: maximize the total profit of the selected items.

Link between the subset sum problem and the knapsack problem? The subset sum problem is a knapsack problem with $p_j = w_j$ for all $j = 1, \ldots, n$.

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define $F(j, c)$ the maximum profit of a subset of items $1, \ldots, j$ with total weight less than or equal to c.

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define $F(j, c)$ the maximum profit of a subset of items $1, \ldots, j$ with total weight less than or equal to c.

What is the value of

- $F(0, 0)$?

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define $F(j, c)$ the maximum profit of a subset of items $1, \ldots, j$ with total weight less than or equal to c.

What is the value of

- $F(0, 0)$? 0

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define $F(j, c)$ the maximum profit of a subset of items $1, \ldots, j$ with total weight less than or equal to c.

What is the value of

- $F(0, 0)$? 0
- $F(0, c)$?

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define $F(j, c)$ the maximum profit of a subset of items $1, \ldots, j$ with total weight less than or equal to c.

What is the value of

- $F(0, 0)$? 0
- $F(0, c)$? 0

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define $F(j, c)$ the maximum profit of a subset of items $1, \ldots, j$ with total weight less than or equal to c.

What is the value of

- $F(0, 0)$? 0
- $F(0, c)$? 0
- $F(j, 0)$?

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define $F(j, c)$ the maximum profit of a subset of items $1, \ldots, j$ with total weight less than or equal to c.

What is the value of

- $F(0, 0)$? 0
- $F(0, c)$? 0
- $F(j, 0)$? 0

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define $F(j, c)$ the maximum profit of a subset of items $1, \ldots, j$ with total weight less than or equal to c.

What is the value of

- $F(0, 0)$? 0
- $F(0, c)$? 0
- $F(j, 0)$? 0

What is the relation between the knapsack problem and F?

Recursive function

For all $j = 0, \ldots, n$, $c = 0, \ldots, C$, let us define $F(j, c)$ the maximum profit of a subset of items $1, \ldots, j$ with total weight less than or equal to c.

What is the value of

- $F(0, 0)$? 0
- $F(0, c)$? 0
- $F(j, 0)$? 0

What is the relation between the knapsack problem and F?
The knapsack problem is equivalent to determining the value of $F(n, C)$.

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

Computing $F(n, C)$

We compute $F(j, c)$ with the following recursive formula:

$$F(j,c) = \begin{cases} 0 & \text{if } j = 0 \\ F(j-1,c) & \text{if } j \neq 0 \text{ and } c < w_j \\ \max \begin{cases} F(j-1,c) \\ F(j-1,c-w_j) + p_j \end{cases} & \text{otherwise} \end{cases}$$

Example: $C = 7$

Item	Weight	Profit
1	3	4
2	5	6
3	4	5
4	2	2

j / c	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	4	4	4	4	4
2	0	0	0	4	4	6	6	6
3	0	0	0	4	5	6	6	9
4	0	0	2	4	5	6	7	9

$$S = \{3, 1\}$$

Table of contents

Introduction

The partition problem and the subset sum problem

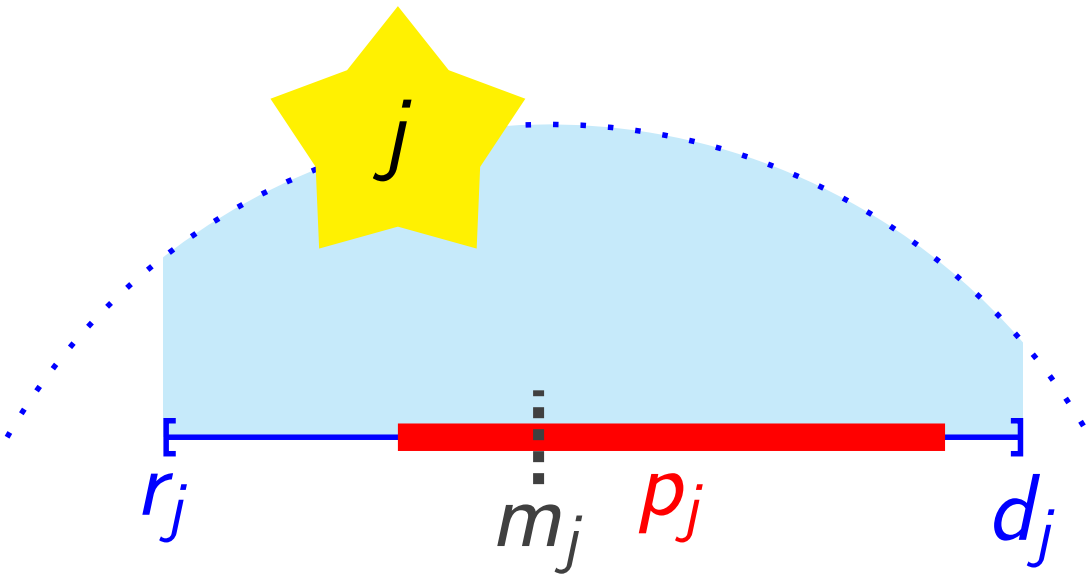
The knapsack problem

The single-night star observation scheduling problem

Dynamic programming as a tree search

Conclusion

A star

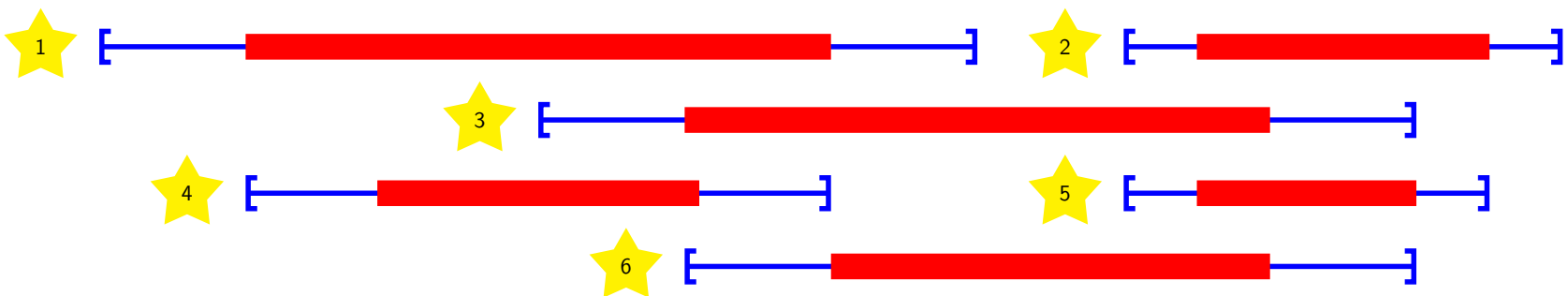


- $[r_j, d_j[$: visibility interval
- p_j: observation time
- w_j: scientific interest of the observation

Every observation j has a meridian $m_j \in [r_j, d_j[$ which is a mandatory instant of the observation.

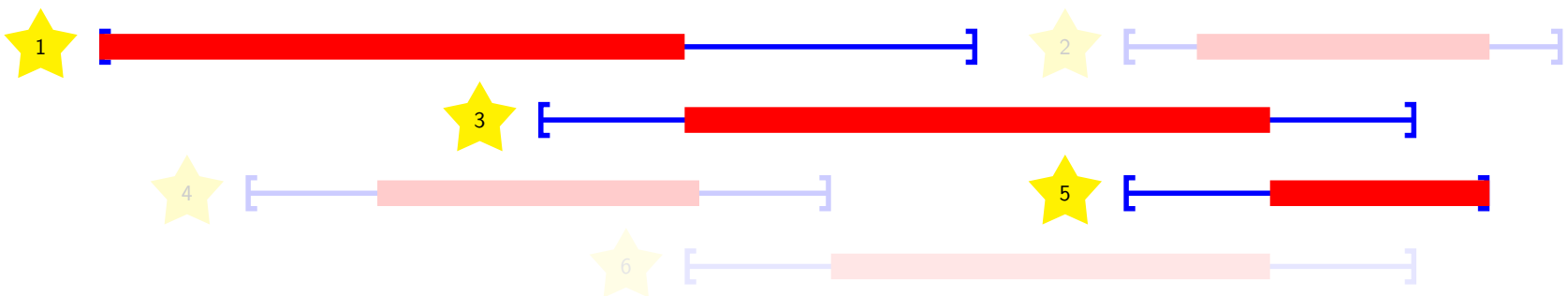
Problem definition

Instance: a set of stars $\mathcal{N}$; each star $j \in \mathcal{N}$ has a scientific interest w_j, an observation time p_j and a time window $[r_j, d_j[$



Problem definition

Instance: a set of stars $\mathcal{N}$; each star $j \in \mathcal{N}$ has a scientific interest w_j, an observation time p_j and a time window $[r_j, d_j[$



Problem: find a subset $\mathcal{N}' \subset \mathcal{N}$ as well as the start date s_j of each selected observation $j \in \mathcal{N}'$ such that:

- for all $j \in \mathcal{N}'$: $[s_j, s_j + p_j[\subset [r_j, d_j[$
- for all $(j_1, j_2) \in \mathcal{N}'^2$: $[s_{j_1}, s_{j_1} + p_{j_1}[\cap [s_{j_2}, s_{j_2} + p_{j_2}[= \emptyset$

Objective: maximize $\sum_{j \in \mathcal{N}'} w_j$ the profit of the selected observations.

Properties

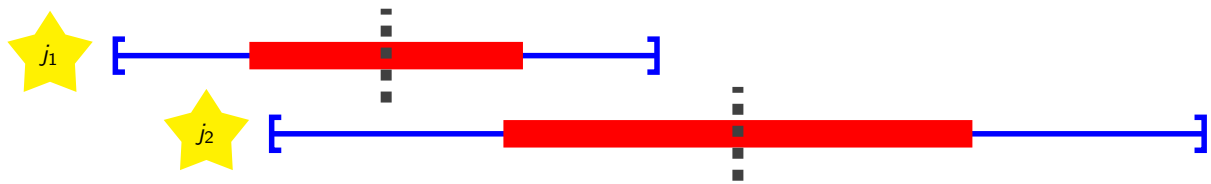
Property 1

There exists an optimal solution in which selected observations are scheduled in non-decreasing order of their mandatory instant.

Properties

Property 1

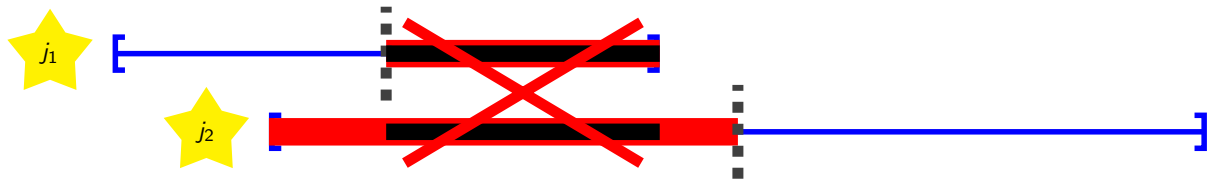
There exists an optimal solution in which selected observations are scheduled in non-decreasing order of their mandatory instant.



Properties

Property 1

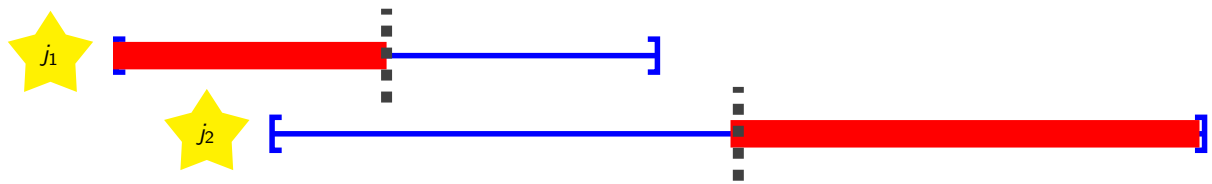
There exists an optimal solution in which selected observations are scheduled in non-decreasing order of their mandatory instant.



Properties

Property 1

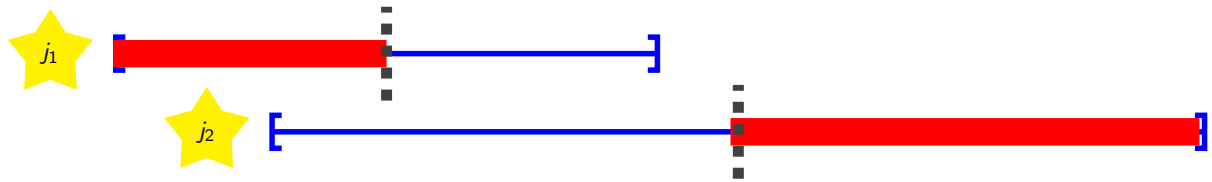
There exists an optimal solution in which selected observations are scheduled in non-decreasing order of their mandatory instant.



Properties

Property 1

There exists an optimal solution in which selected observations are scheduled in non-decreasing order of their mandatory instant.



Property 2

Consider a subset $\mathcal{N}' \subset \mathcal{N}$ and an observation $j_{\max}$ such that $d_{j_{\max}} = \max_{j \in \mathcal{N}'} d_j$. If there exists a feasible solution with selected observations $\mathcal{N}'$, then there exists a feasible solution with selected observations $\mathcal{N}'$ such that $s_{j_{\max}} = d_{j_{\max}} - p_{j_{\max}}$.

Recursive function

We consider than the observations are numbered in non-decreasing order of their mandatory instant.

Recursive function

We consider than the observations are numbered in non-decreasing order of their mandatory instant.

For all $j = 0, \ldots, n$, $t = 0, \ldots, T$, let us define $F(j, t)$ the maximum scientific interest of a subset of observations of $1, \ldots, j$ during the interval $[0, T]$.

Recursive function

We consider than the observations are numbered in non-decreasing order of their mandatory instant.

For all $j = 0, \ldots, n$, $t = 0, \ldots, T$, let us define $F(j, t)$ the maximum scientific interest of a subset of observations of $1, \ldots, j$ during the interval $[0, T]$.

$$F(j,t) = \begin{cases} 0 & j = 0 \\ F(j-1,t) & j \neq 0, t \in [0, r_j + p_j[\\ \max \begin{cases} F(j-1,t) \\ F(j-1, \min\{d_j, t\} - p_j) + w_j \end{cases} & \text{otherwise} \end{cases}$$

Recursive function

We consider than the observations are numbered in non-decreasing order of their mandatory instant.

For all $j = 0, \ldots, n$, $t = 0, \ldots, T$, let us define $F(j, t)$ the maximum scientific interest of a subset of observations of $1, \ldots, j$ during the interval $[0, T]$.

$$F(j,t) = \begin{cases} 0 & j = 0 \\ F(j-1,t) & j \neq 0, t \in [0, r_j + p_j[\\ \max \begin{cases} F(j-1,t) \\ F(j-1, \min\{d_j, t\} - p_j) + w_j \end{cases} & \text{otherwise} \end{cases}$$

Complexity: $O(nT)$

Recursive function

We consider than the observations are numbered in non-decreasing order of their mandatory instant.

For all $j = 0, \ldots, n$, $t = 0, \ldots, T$, let us define $F(j, t)$ the maximum scientific interest of a subset of observations of $1, \ldots, j$ during the interval $[0, T]$.

$$F(j,t) = \begin{cases} 0 & j = 0 \\ F(j-1,t) & j \neq 0, t \in [0, r_j + p_j[\\ \max \begin{cases} F(j-1,t) \\ F(j-1, \min\{d_j, t\} - p_j) + w_j \end{cases} & \text{otherwise} \end{cases}$$

Complexity: $O(nT)$

Sometimes, a bit of work is needed in order to exhibit the structure to design an efficient algorithm based on dynamic programming.

Table of contents

Introduction

The partition problem and the subset sum problem

The knapsack problem

The single-night star observation scheduling problem

Dynamic programming as a tree search

Conclusion

Breadth first search: example (knapsack problem)

Example:
$C = 5$

Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2

Breadth first search: example (knapsack problem)

Example:
$C = 5$

Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2

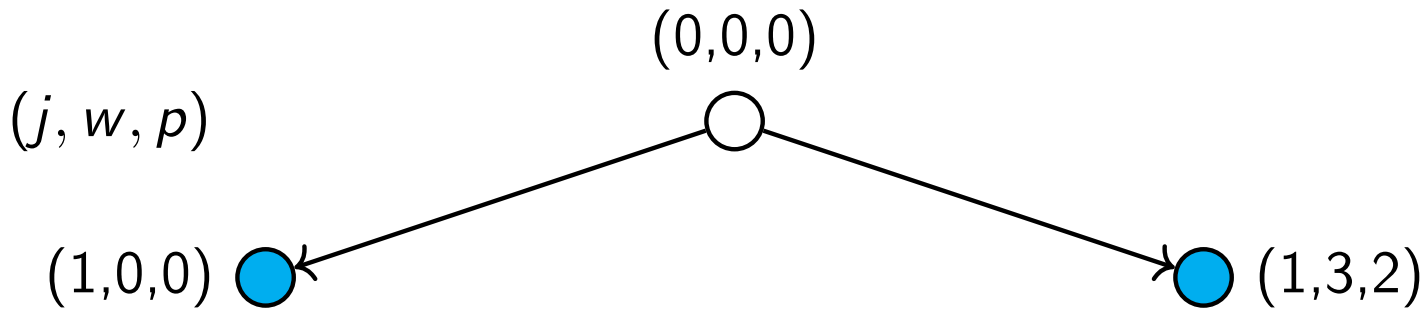
(j, w, p)

(0,0,0)

Breadth first search: example (knapsack problem)

Example:
$C = 5$

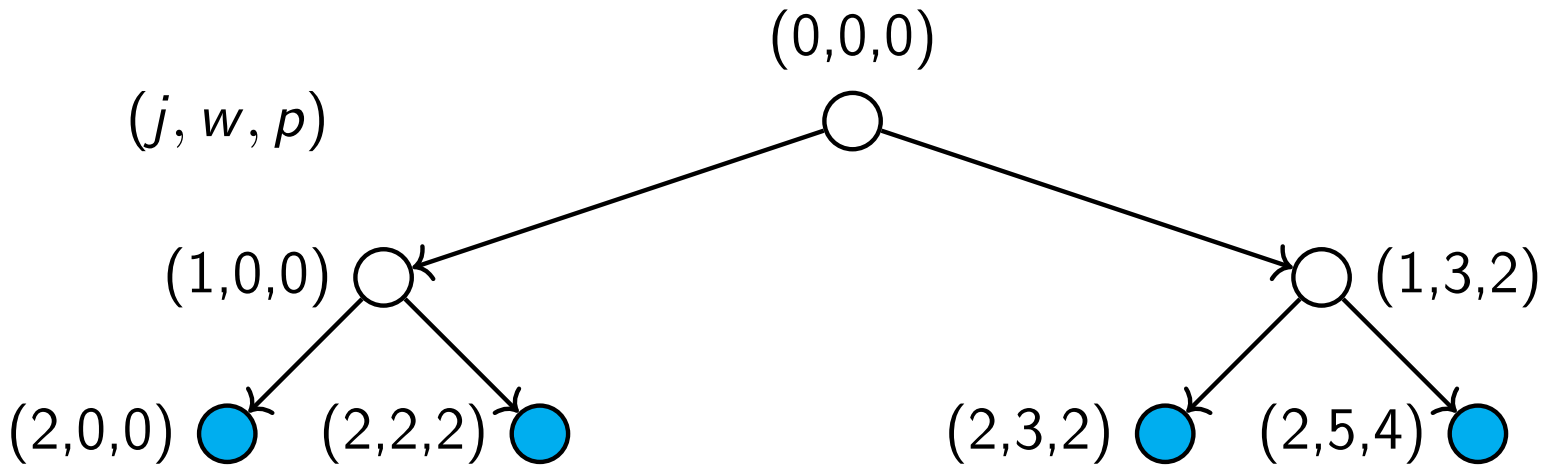
Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2



Breadth first search: example (knapsack problem)

Example:
$C = 5$

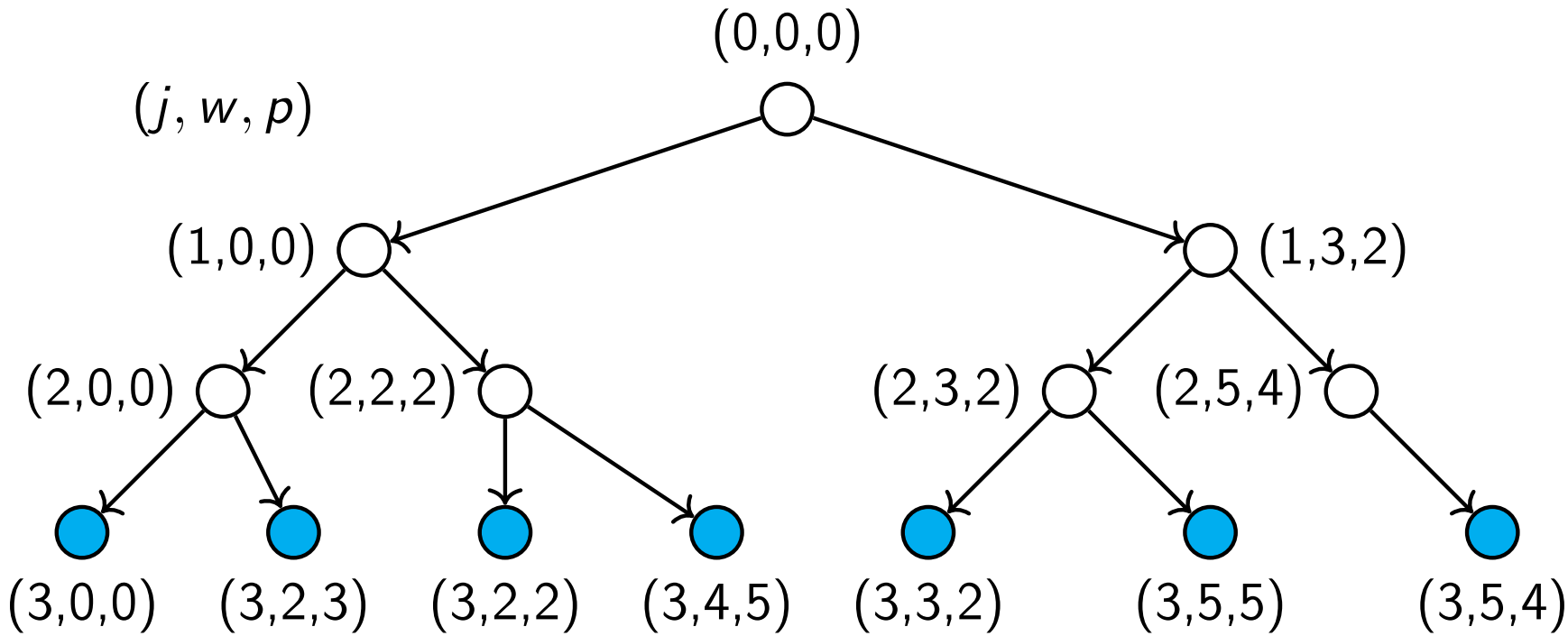
Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2



Breadth first search: example (knapsack problem)

Example:
$C = 5$

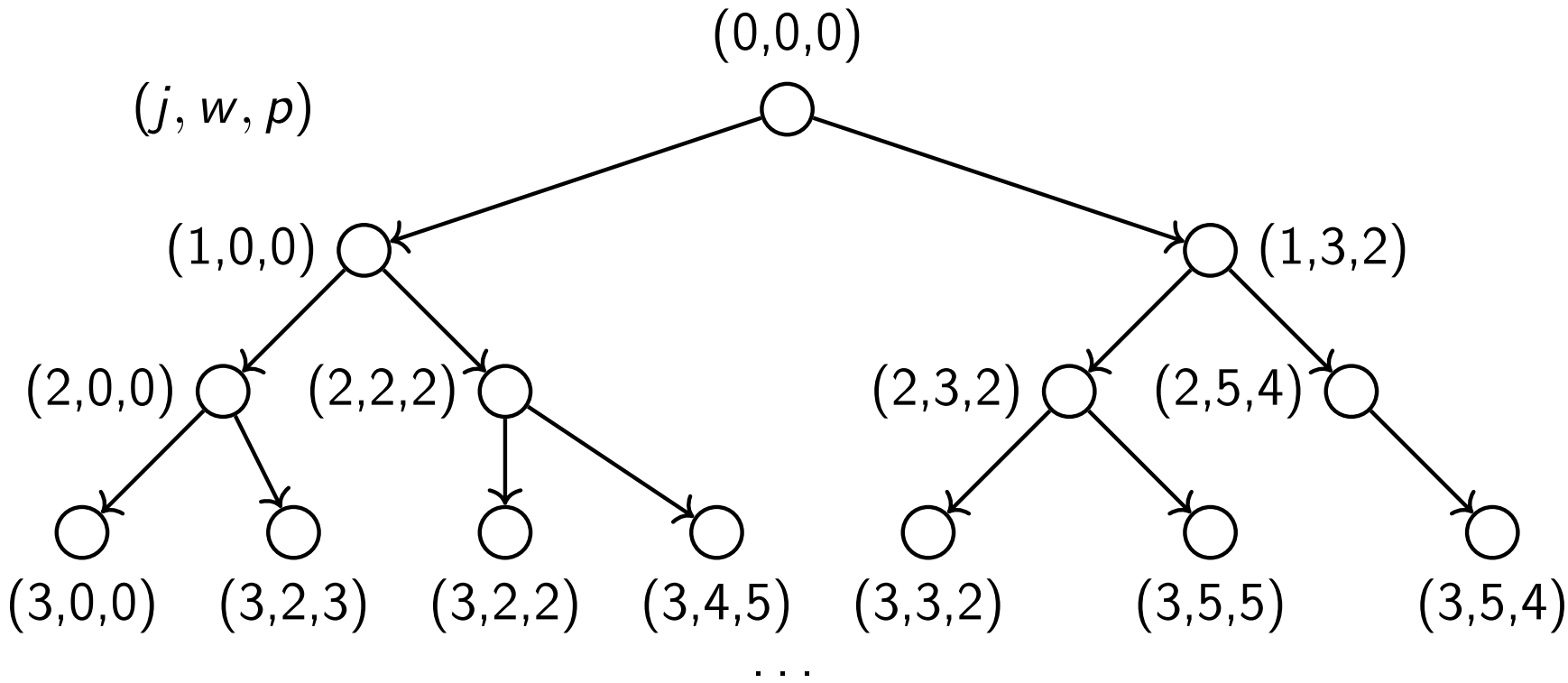
Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2



Breadth first search: example (knapsack problem)

Example:
$C = 5$

Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2



Breadth first search (knapsack problem)

procedure BreadthFirstSearch(w, C)
    $L_0 \leftarrow ((0, 0, 0))$
    for $k = 1, \ldots, n$ **do**
        for $(j, w, p) \in L_{k-1}$ **do**
            $L_k \leftarrow L_k \cup ((j + 1, w, p))$
            if $w + w_{j+1} \leq C$ **then**
                $L_k \leftarrow L_k \cup ((j + 1, w + w_{j+1}, p + p_{j+1}))$

Breadth first search (knapsack problem)

procedure BreadthFirstSearch(w, C)
    $L_0 \leftarrow ((0, 0, 0))$
    for $k = 1, \ldots, n$ **do**
        for $(j, w, p) \in L_{k-1}$ **do**
            $L_k \leftarrow L_k \cup ((j + 1, w, p))$
            if $w + w_{j+1} \leq C$ **then**
                $L_k \leftarrow L_k \cup ((j + 1, w + w_{j+1}, p + p_{j+1}))$

- Time complexity?

Breadth first search (knapsack problem)

procedure BreadthFirstSearch(w, C)
    $L_0 \leftarrow ((0, 0, 0))$
    for $k = 1, \ldots, n$ **do**
        for $(j, w, p) \in L_{k-1}$ **do**
            $L_k \leftarrow L_k \cup ((j + 1, w, p))$
            if $w + w_{j+1} \leq C$ **then**
                $L_k \leftarrow L_k \cup ((j + 1, w + w_{j+1}, p + p_{j+1}))$

- Time complexity? $O(2^n)$

Breadth first search (knapsack problem)

procedure BreadthFirstSearch(w, C)
    $L_0 \leftarrow ((0, 0, 0))$
    for $k = 1, \ldots, n$ **do**
        for $(j, w, p) \in L_{k-1}$ **do**
            $L_k \leftarrow L_k \cup ((j + 1, w, p))$
            if $w + w_{j+1} \leq C$ **then**
                $L_k \leftarrow L_k \cup ((j + 1, w + w_{j+1}, p + p_{j+1}))$

- Time complexity? $O(2^n)$
- Space complexity?

Breadth first search (knapsack problem)

procedure BreadthFirstSearch(w, C)
    $L_0 \leftarrow ((0, 0, 0))$
    for $k = 1, \ldots, n$ **do**
        for $(j, w, p) \in L_{k-1}$ **do**
            $L_k \leftarrow L_k \cup ((j + 1, w, p))$
            if $w + w_{j+1} \leq C$ **then**
                $L_k \leftarrow L_k \cup ((j + 1, w + w_{j+1}, p + p_{j+1}))$

- Time complexity? $O(2^n)$
- Space complexity? $O(2^n)$

Dominance rule (knapsack problem)

We express dynamic programming as a dominance rule: Consider two nodes $n_1 = (j_1, w_1, p_1)$ and $n_2 = (j_2, w_2, p_2)$. If

$$j_1 \leq j_2 \quad \text{and} \quad w_1 \leq w_2 \quad \text{and} \quad p_1 \geq p_2$$

then node n_1 dominates node n_2 and therefore node n_2 can be safely pruned.

Breadth first search + dominance rule: example (knapsack problem)

Example:
$C = 5$

Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2

Breadth first search + dominance rule: example (knapsack problem)

Example:
$C = 5$

Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2

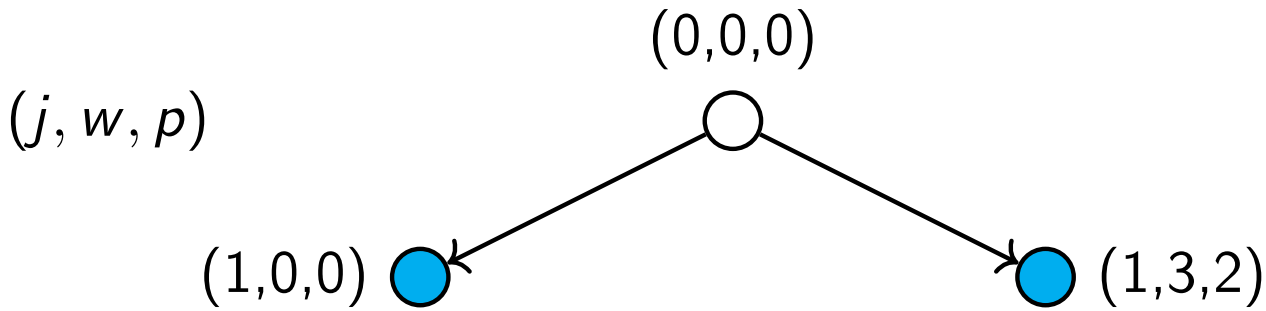
(j, w, p)

(0,0,0)

Breadth first search + dominance rule: example (knapsack problem)

Example:
$C = 5$

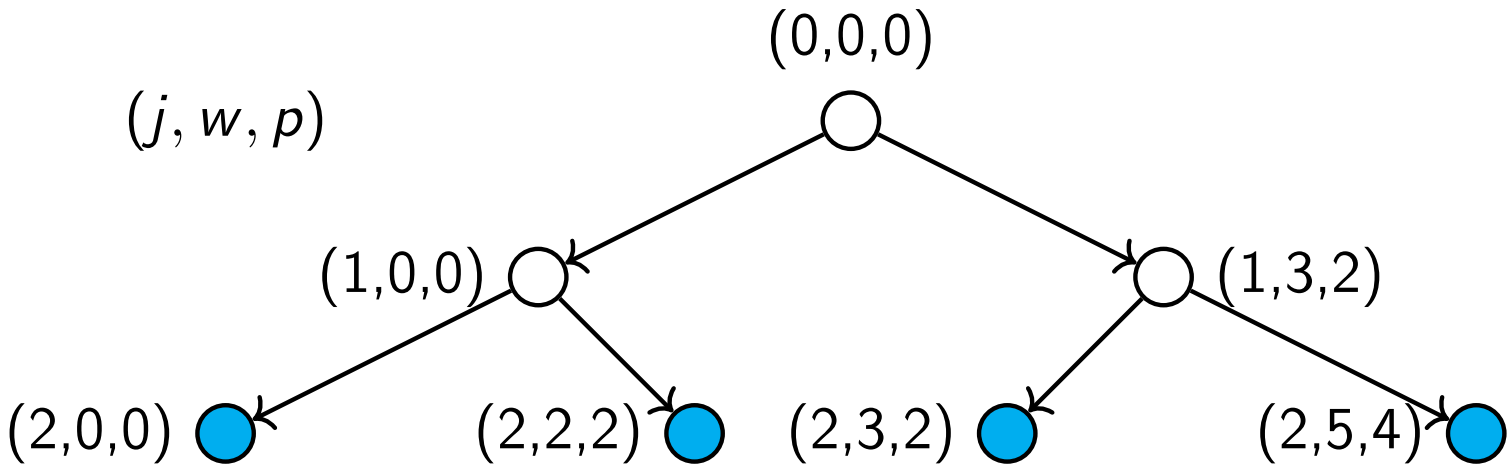
Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2



Breadth first search + dominance rule: example (knapsack problem)

Example:
$C = 5$

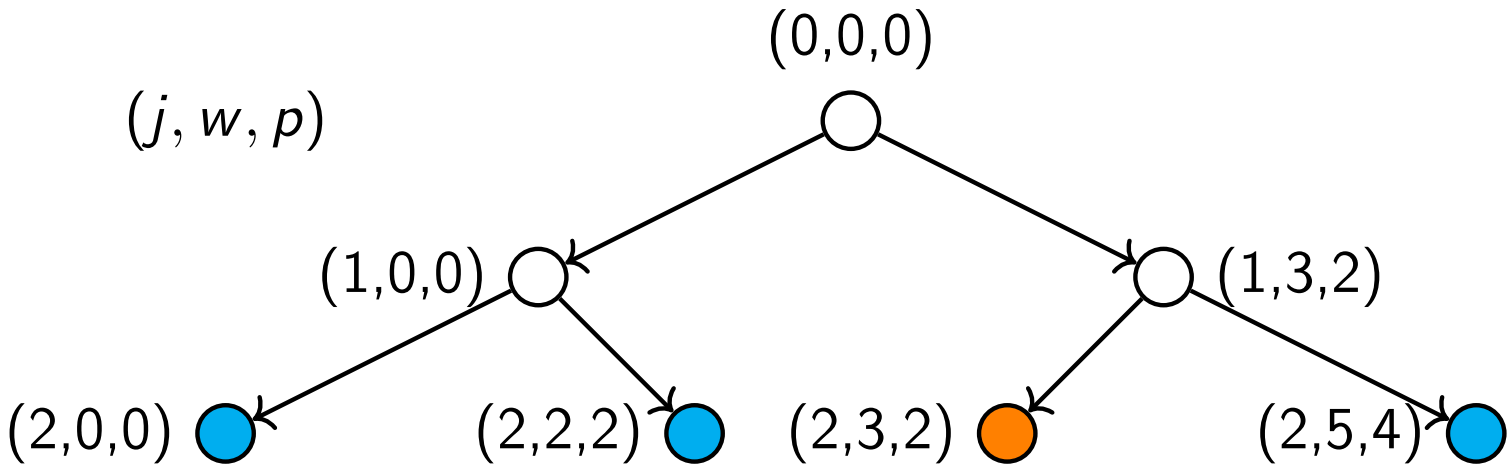
Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2



Breadth first search + dominance rule: example (knapsack problem)

Example:
$C = 5$

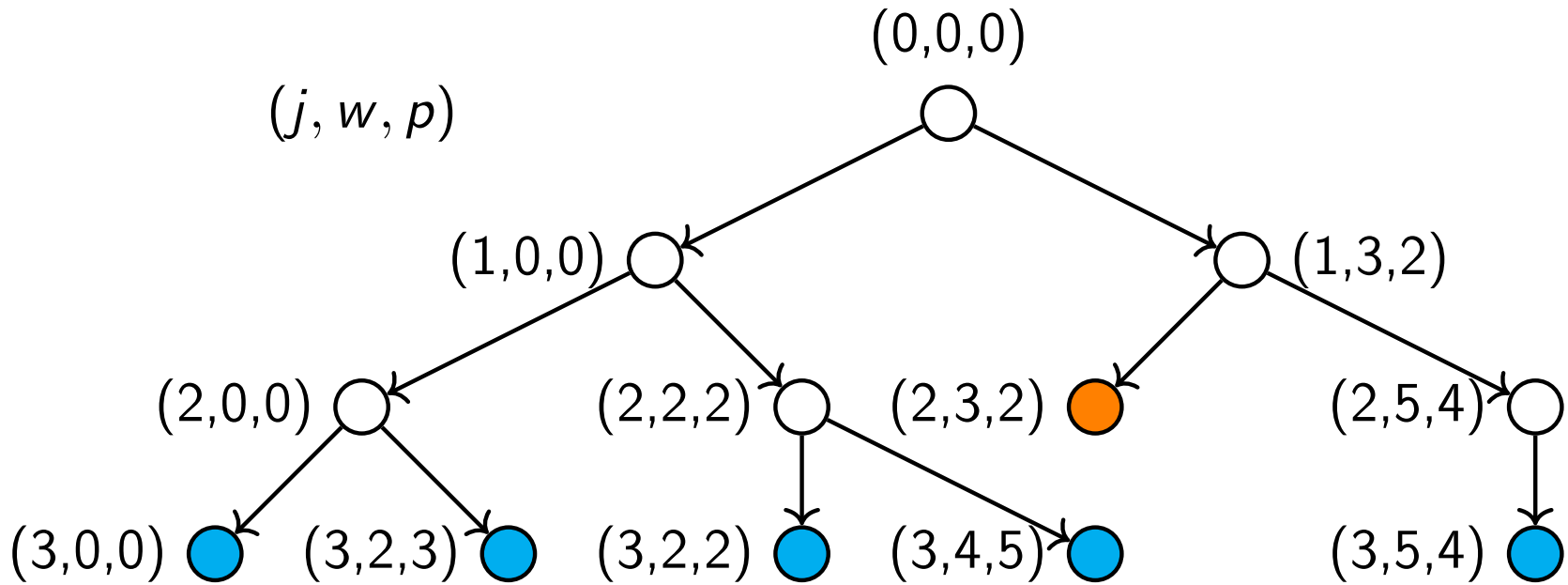
Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2



Breadth first search + dominance rule: example (knapsack problem)

Example:
$C = 5$

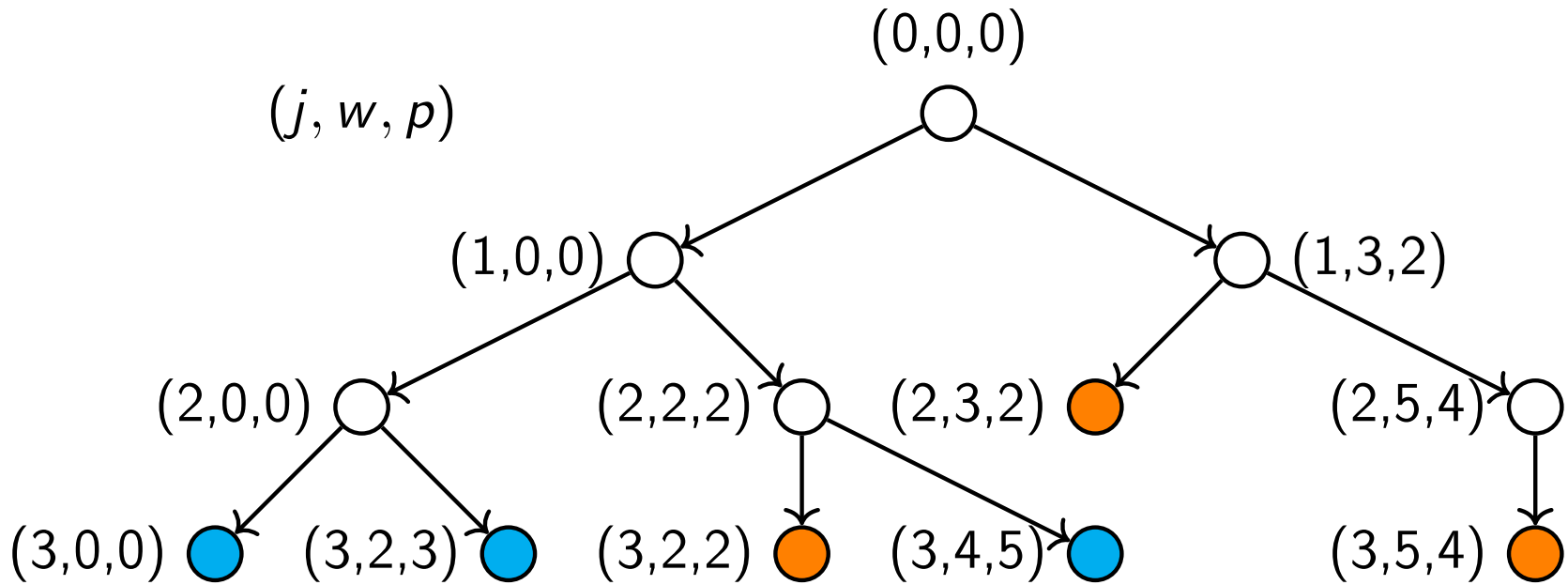
Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2



Breadth first search + dominance rule: example (knapsack problem)

Example:
$C = 5$

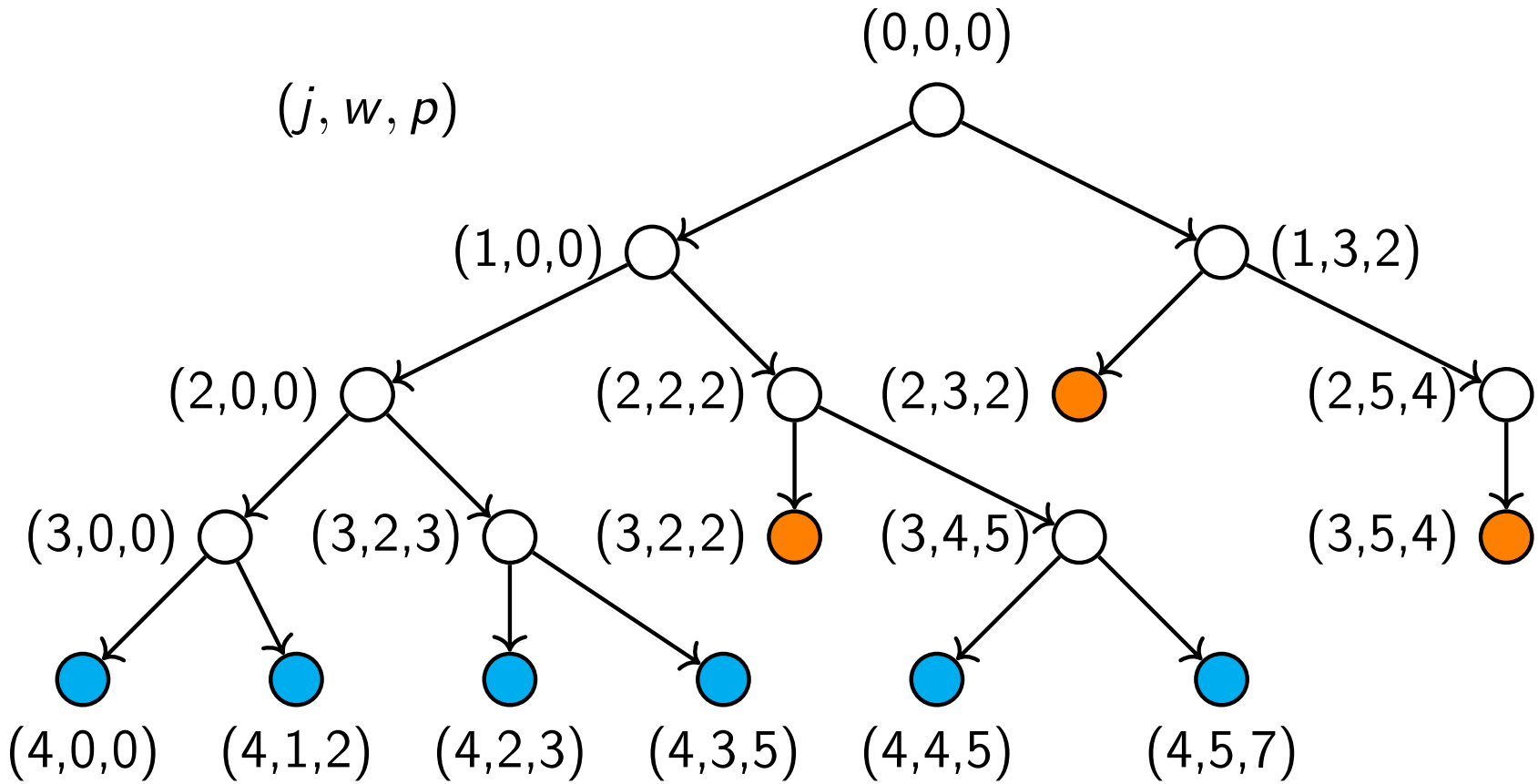
Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2



Breadth first search + dominance rule: example (knapsack problem)

Example:
$C = 5$

Item	Weight	Profit
1	3	2
2	2	2
3	2	3
4	1	2



Breadth first search + dominance rule (knapsack problem)

procedure BreadthFirstSearch(w, C)
$L_0 \leftarrow ((0, 0, 0))$
for $k = 1, \ldots, n$ **do**
for $(j, w, p) \in L_{k-1}$ **do**
$L_k \leftarrow L_k \cup ((j + 1, w, p))$
if $w + w_{j+1} \leq C$ **then**
$L_k \leftarrow L_k \cup ((j + 1, w + w_{j+1}, p + p_{j+1}))$
Remove all dominated nodes from L_j

Breadth first search + dominance rule (knapsack problem)

procedure BreadthFirstSearch(w, C)
$L_0 \leftarrow ((0, 0, 0))$
for $k = 1, \ldots, n$ **do**
for $(j, w, p) \in L_{k-1}$ **do**
$L_k \leftarrow L_k \cup ((j + 1, w, p))$
if $w + w_{j+1} \leq C$ **then**
$L_k \leftarrow L_k \cup ((j + 1, w + w_{j+1}, p + p_{j+1}))$
Remove all dominated nodes from L_j

- Time complexity?

Breadth first search + dominance rule (knapsack problem)

procedure BreadthFirstSearch(w, C)
$L_0 \leftarrow ((0, 0, 0))$
for $k = 1, \ldots, n$ **do**
for $(j, w, p) \in L_{k-1}$ **do**
$L_k \leftarrow L_k \cup ((j + 1, w, p))$
if $w + w_{j+1} \leq C$ **then**
$L_k \leftarrow L_k \cup ((j + 1, w + w_{j+1}, p + p_{j+1}))$
Remove all dominated nodes from L_j

- Time complexity? The complexity depends on the complexity of applying the dominance rule!
  In this case, it is possible to implement it in $O(C)$ and keep the complexity of the whole algorithm to $O(nC)$ as for the iterative implementation. But in some cases, the complexity might increase.

Table of contents

Introduction

The partition problem and the subset sum problem

The knapsack problem

The single-night star observation scheduling problem

Dynamic programming as a tree search

Conclusion

Conclusion

- Dynamic programming: solving a problem recursively and storing the results of the subproblems to avoid recomputing them multiple times.
- It requires the problem to have a specific structure. It might not be applicable to all problems.
- Sometimes, a bit of work is needed in order to exhibit this structure
- Multiple possible implementations with their advantages and drawbacks (recursive, iterative, tree search)
- "Knapsack Problems" (Kellerer, Pferschy et Pisinger, 2004)
- Dynamic programming through path problems: `https://moodle.caseine.org/mod/page/view.php?id=30723`

Advanced Models and Methods in Operations Research
Dynamic programming

Florian Fontan

November 7, 2023