Informatique tronc commun — MP*

Programmation dynamique

Introduction

Le programme : première année

Algorithmes gloutons.	Rendu de monnaie. Allocation de salles pour des cours. Sélection d'activité. *On peut montrer par des exemples qu'un algorithme glouton ne fournit pas toujours une solution exacte ou optimale.*

- **algorithmes gloutons** : « meilleure » solution locale
- « ne fournit pas toujours une solution exacte ou optimale »

Le programme : deuxième année

3.2 Dictionnaires et programmation dynamique

Les dictionnaires sont utilisés en boîte noire dès la première année ; les principes de leur fonctionnement sont présentés en deuxième année. Ils peuvent être utilisés afin de mettre en mémoire des résultats intermédiaires quand on implémente une stratégie d'optimisation par programmation dynamique.

Notions	**Commentaires**
Dictionnaires, clés et valeurs.	On présente les principes du hachage, et les limitations qui en découlent sur le domaine des clés utilisables.
Usage des dictionnaires en programmation Python.	Syntaxe pour l'écriture des dictionnaires. Parcours d'un dictionnaire.
Programmation dynamique. Propriété de sous-structure optimale. Chevauchement de sous-problèmes. Calcul de bas en haut ou par mémoïsation. Reconstruction d'une solution optimale à partir de l'information calculée.	La mémoïsation peut être implémentée à l'aide d'un dictionnaire. On souligne les enjeux de complexité en mémoire. Exemples : partition équilibrée d'un tableau d'entiers positifs, ordonnancement de tâches pondérées, plus longue sous-suite commune, distance d'édition (Levenshtein), distances dans un graphe (Floyd-Warshall).
Mise en œuvre	
Les exemples proposés ne forment une liste ni limitative ni impérative. Les cas les plus complexes de situations où la programmation dynamique peut être utilisée sont guidés. On met en rapport le statut de la propriété de sous-structure optimale en programmation dynamique avec sa situation en stratégie gloutonne vue en première année.	

Quatre stratégies pour trouver une configuration optimale

Méthode	Idée générale	Exemples
Solution naïve	tester **toutes** les configurations (backtracking) : exponentiel	tous les problèmes
Diviser pour régner	sous-problèmes **indépendants**, rien à mémoriser	tri fusion, dichotomie
Algorithme glouton	le meilleur choix **à court terme**, jamais remis en cause	sélection d'activités, Dijkstra
Programmation dynamique	**tous les choix**, chaque sous-problème résolu **une fois** (ils se **chevauchent**)	rendu de monnaie, sous-suites, sac à dos

Programmation dynamique : définition

Programmation dynamique. ramener un problème à une famille de **sous-problèmes** liés par une **relation de récurrence**, chaque sous-problème n'étant résolu **qu'une seule fois** (mémoïsation, « de haut en bas », ou tableau, « de bas en haut »).

1. **Résumer** : choisir un **état** (k, i, (i, j)…), assez riche, assez peu nombreux
2. **Décomposer** : une récurrence (équation de Bellman), justifiée par la **sous-structure optimale**
3. **Mémoriser** : chaque état une seule fois, puis **reconstruire**

« Smart recursion »

> (and instructors, and textbooks) make the mistake of focusing on the table—because tables are easy and familiar—instead of the *much* more important (and difficult) task of finding a correct recurrence. As long as we memoize the correct recurrence, an explicit table isn't really necessary, but if the recurrence is incorrect, we are well and truly hosed.

Dynamic programming is *not* about filling in tables.
It's about smart recursion!

- J. Erickson, *Algorithms* : la programmation dynamique, ce n'est pas remplir des tableaux, c'est de la **récursion intelligente**

Aux concours : la structure des sujets (souvent)

1. un **algorithme glouton** non optimal (ou une recherche exhaustive trop coûteuse)
2. une **récurrence**, donnée ou à justifier « brièvement »
3. son **implémentation** : mémoïsation et/ou tableau
4. sa **complexité**
5. la **reconstruction** d'une solution

Aux concours : les sujets 2023-2026

Sujet	Problème	État
Mines 2023 Q21-26	justifier un paragraphe	i : premier mot d'une ligne
Mines 2024 Q18-25	décodage de Viterbi	(symbole i, position j)
Mines 2025 Q7-14	sac à dos	(objets $< k$, capacité r)
X-ENS 2023 Q9-11	distance d'édition	(préfixe i, préfixe j)
X-ENS 2024 Q10-12	logimages	(case c, bloc b)
X-ENS 2025 Q6-7, 11-15	chemin de pénalité minimale	(case, bonus activés)
Mines 2026	—	pas de programmation dynamique
X-ENS 2026 Q10-12	nombre de chemins d'un graphe sans cycle	sommet s

- la phrase qui **définit l'état** est toujours donnée, la récurrence : donnée, ou à formaliser d'après une phrase (à trouver : X-ENS 2023)
- **bas en haut**, **complexité** et **reconstruction**

Pour faciliter les programmes

```
INFINI = float("inf")
```

$+\infty$: plus grand que tout entier, `INFINI + 1 == INFINI`

Rappels : les algorithmes gloutons

L'idée : le meilleur choix local

Définition. Un algorithme **glouton** construit une solution pas à pas en faisant, à chaque étape, le choix qui paraît **le meilleur localement**, sans **jamais revenir** sur un choix effectué.

Schéma : **trier** les candidats selon un critère, puis les **parcourir** en retenant chacun s'il est compatible

- **rendu de monnaie** : pièces `pieces` (triées, contenant 1), rendre n avec le moins de pièces, glouton : la plus grande pièce possible
- **sac à dos** : n objets (p_i, v_i), capacité b, prix total maximal ; exhaustif : $O(n2^n)$; glouton : rapport prix/poids décroissant

Exemple: rendu de monnaie

On dispose de pièces de valeurs entières, parmi lesquelles 1.

On cherche à rendre une somme n avec le moins de pièces possible.

La liste des pièces (une liste d'entiers `pieces`) est supposée triée par valeur croissante.

pièces : 

en quantité illimitée, somme à rendre : $n = 6$

Exemple: sac à dos

On dispose de n objets de poids $(p_i)_{0 \leq i < n}$ et de prix $(v_i)_{0 \leq i < n}$.

On cherche à remplir un sac de capacité b avec un sous-ensemble d'objets de prix total maximal.

Pour l'algorithme glouton, la liste des objets (une liste de tuples `(poids, prix)`) sera triée par rapport prix/poids décroissant.

objet 0	**objet 1**	**objet 2**
$p = 2$	$p = 1$	$p = 4$
$v = 3$	$v = 4$	$v = 4$
$v/p = 1{,}5$	$v/p = 4$	$v/p = 1$

sac de capacité $b = 5$

Question : `rendu_glouton`

Question (poly § 2.1) Écrire `rendu_glouton(n, pieces)` qui renvoie la liste des pièces rendues par un algorithme glouton.

Question : rendu_glouton (correction)

```python
def rendu_glouton(n, pieces):
    """Rend n avec la plus grande pièce possible à chaque étape."""
    rendu, reste = [], n
    i = len(pieces) - 1
    while reste > 0:
        if pieces[i] <= reste:
            rendu.append(pieces[i])
            reste = reste - pieces[i]
        else:
            i = i - 1  # On s'arrête forcément à la pièce 1
    return rendu
```

Question : `rendu_glouton` (correction)

```python
def rendu_glouton(n, pieces):
    """Rend n avec la plus grande pièce possible à chaque étape."""
    rendu, reste = [], n
    i = len(pieces) - 1
    while reste > 0:
        if pieces[i] <= reste:
            rendu.append(pieces[i])
            reste = reste - pieces[i]
        else:
            i = i - 1  # On s'arrête forcément à la pièce 1
    return rendu
```

- terminaison (**variant**): `reste + i` décroît strictement
- correction (**invariant**): `reste`+ somme des pièces rendues = `n`
- complexité: $O(|\mathcal{P}| + n)$

Question : `sac_glouton`

Question (poly § 2.1) Les objets sont donnés par une liste de couples `(poids, prix)` **déjà triée** par rapport prix/poids décroissant. Écrire `sac_glouton(objets, b)` qui renvoie la liste des indices des objets pris et leur prix total.

Question : sac_glouton (correction)

```python
def sac_glouton(objets, b):
    """Indices des objets pris et prix total."""
    choisis, prix, reste = [], 0, b  # reste : capacité restante
    for i in range(len(objets)):
        poids_i, prix_i = objets[i]
        if poids_i <= reste:
            choisis.append(i)
            reste = reste - poids_i
            prix = prix + prix_i
    return choisis, prix
```

Question : sac_glouton (correction)

```python
def sac_glouton(objets, b):
    """Indices des objets pris et prix total."""
    choisis, prix, reste = [], 0, b  # reste : capacité restante
    for i in range(len(objets)):
        poids_i, prix_i = objets[i]
        if poids_i <= reste:
            choisis.append(i)
            reste = reste - poids_i
            prix = prix + prix_i
    return choisis, prix
```

- terminaison: boucle `for`
- correction (**invariant**): `reste` + somme des poids choisis $= b$, `reste` ≥ 0
- complexité: $O(n)$ **après un tri** en $O(n \ln(n))$ (au mieux)

Question : … qui ne donne que rarement une solution optimale

Question (poly § 2.2)

- Que donne `rendu_glouton(6, [1, 3, 4])` ?
- Et un algorithme glouton du sac à dos sur les objets `obj = [(2, 3), (1, 4), (4, 4)]` avec $b = 5$? On triera d'abord les objets par rapport prix/poids décroissant.
- Ces résultats sont-ils optimaux ?

Question : … qui ne donne que rarement une solution optimale (correction, 1/2)

glouton : 4 + 1 + 1 = 6, trois pièces

Question : … qui ne donne que rarement une solution optimale (correction, 1/2)

glouton : 4 + 1 + 1 = 6, trois pièces

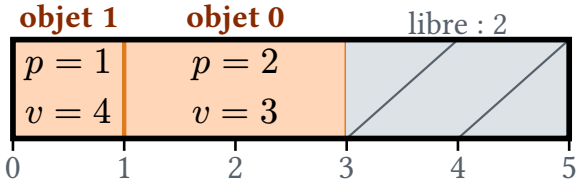
optimal : 3 + 3 = 6, deux pièces

Question : … qui ne donne que rarement une solution optimale (correction, 2/2)

objet 0	objet 1	objet 2
$p = 2$	$p = 1$	$p = 4$
$v = 3$	$v = 4$	$v = 4$
$v/p = 1{,}5$	$v/p = 4$	$v/p = 1$

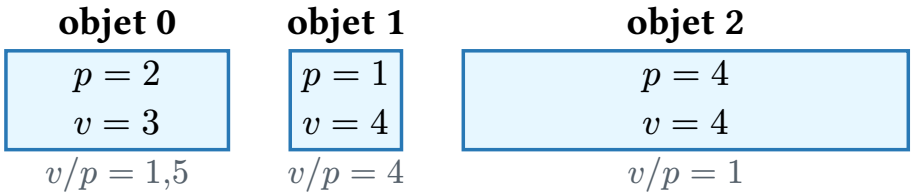
sac de capacité $b = 5$

glouton (rapport v/p décroissant) : objets 1 puis 0, poids 3, prix **7**



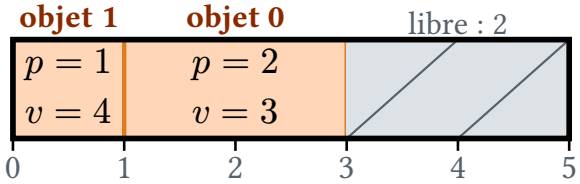
laissé : objet 2 ($p = 4$, $v = 4$)

Question : … qui ne donne que rarement une solution optimale (correction, 2/2)

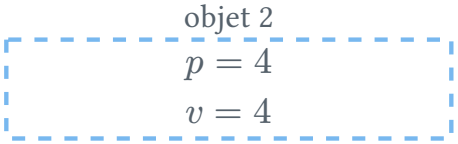


sac de capacité $b = 5$

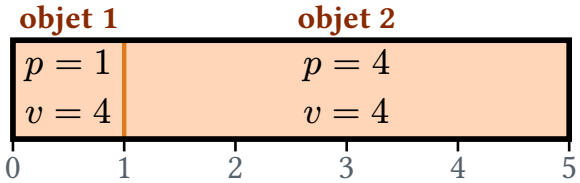
glouton (rapport v/p décroissant) : objets 1 puis 0, poids 3, prix **7**



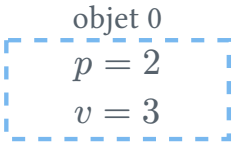
laissé :



optimum : objets 1 et 2, poids 5, prix **8**



laissé :



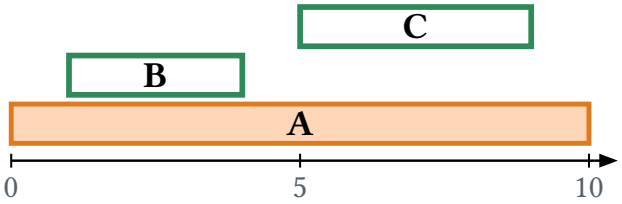
Sélection d'activités : trois critères gloutons

- une salle, n activités $[d_i, f_i[$, en accueillir le plus possible sans chevauchement
- trois critères possibles: **commence le plus tôt**, **la plus courte**, **finit le plus tôt**

Question (poly § 2.2) Pour chacun des deux premiers critères, donner un contre-exemple à trois tâches.

Question : trois critères gloutons (correction)

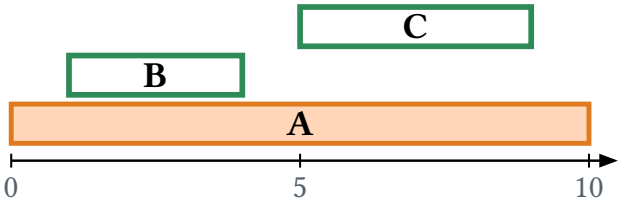
début le plus tôt



glouton : 1 tâche, optimum : 2

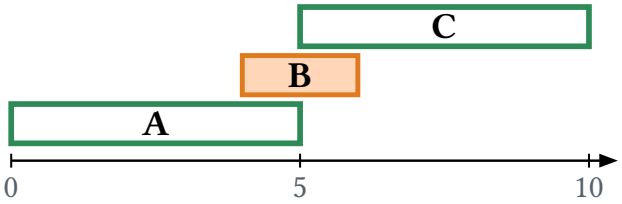
Question : trois critères gloutons (correction)

début le plus tôt



glouton : 1 tâche, optimum : 2

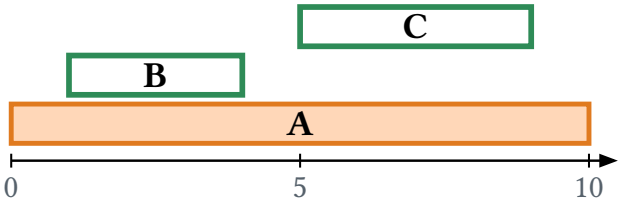
la plus courte



glouton : 1 tâche, optimum : 2

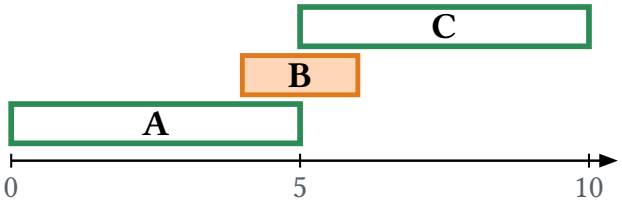
Question : trois critères gloutons (correction)

début le plus tôt



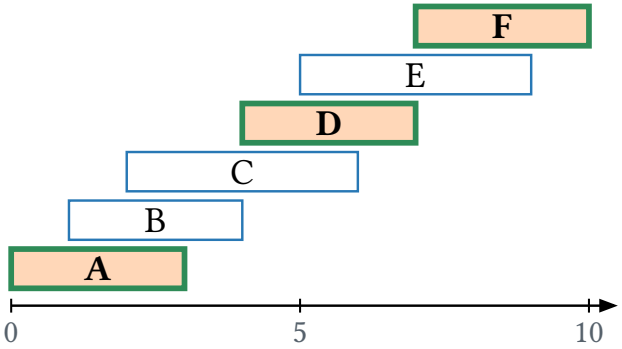
glouton : 1 tâche, optimum : 2

la plus courte



glouton : 1 tâche, optimum : 2

fin la plus tôt



glouton : 3 tâches, optimal

Un algorithme glouton comme outil de preuve d'existence

- un algorithme glouton qui **termine** prouve qu'une solution **existe** : argument constructif

Question (poly § 2.3) **Fractions égyptiennes.** Montrer que tout rationnel p/q avec $0 < p < q$ peut s'écrire sous la forme

$$\frac{p}{q} = \sum_{k=0}^{m-1} \frac{1}{a_k}$$

où $a_0, \ldots, a_{m-1}$ sont des entiers *distincts*, en étudiant un algorithme glouton « retirer la plus grande fraction $1/n \leq p/q$ ». Écrire `fractions_egyptiennes(p, q)` qui renvoie la liste des dénominateurs.

Question : fractions égyptiennes (correction)

```
def fractions_egyptiennes(p, q):
    denominateurs = []
    num, den = p, q                    # fraction restante num/den
    while num > 0:
        n = den // num                 # partie entière inférieure de den/num
        if den % num != 0:             # partie entière supérieure
            n = n + 1
        denominateurs.append(n)
        num, den = num * n - den, den * n      # num/den - 1/n
    return denominateurs
```

Question : fractions égyptiennes (correction)

```
def fractions_egyptiennes(p, q):
    denominateurs = []
    num, den = p, q                    # fraction restante num/den
    while num > 0:
        n = den // num                 # partie entière inférieure de den/num
        if den % num != 0:             # partie entière supérieure
            n = n + 1
        denominateurs.append(n)
        num, den = num * n - den, den * n      # num/den - 1/n
    return denominateurs
```

- glouton: n plus petit entier tel que $\frac{1}{n} \leq \frac{p}{q}$
- terminaison (**variant**): `num` décroît strictement
- correction (**invariant**): somme des fractions + fraction restante = $\frac{p}{q}$

Question : théorème de Zeckendorf

Question (poly § 2.3) **Théorème de Zeckendorf.** Montrer que tout entier $n \geq 1$ est somme de nombres de Fibonacci distincts et **non consécutifs** ($F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, \ldots$). Écrire une fonction `zeckendorf(n)` qui renvoie la liste des termes.

Question : théorème de Zeckendorf

Question (poly § 2.3) **Théorème de Zeckendorf.** Montrer que tout entier $n \geq 1$ est somme de nombres de Fibonacci distincts et **non consécutifs** ($F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, \ldots$). Écrire une fonction `zeckendorf(n)` qui renvoie la liste des termes.

- glouton: plus grand Fibonacci $F_k \leq n$
- terminaison (**variant**): `k` décroît strictement (dans le code)
- correction (**invariant**): la somme est correcte et $n - F_k < F_{k-1}$ donc non consécutifs

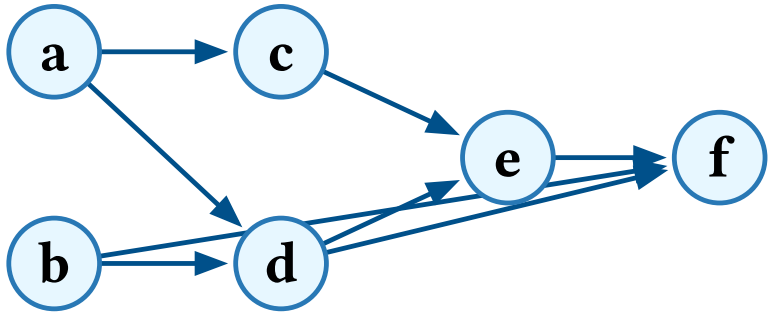
Question : théorème de Zeckendorf (correction)

```
def zeckendorf(n):
    fib = [1, 2]                        # calculée une seule fois
    while fib[len(fib) - 1] <= n:
        fib.append(fib[len(fib) - 1] + fib[len(fib) - 2])
    termes = []
    reste = n
    k = len(fib) - 1
    while reste > 0:
        if fib[k] <= reste:
            termes.append(fib[k])
            reste = reste - fib[k]
        k = k - 1
    return termes
```

Question : tri topologique

Question (poly § 2.3) **Tri topologique.** Montrer qu'un graphe **orienté sans cycle** admet un **ordre topologique** c'est-à-dire une numérotation de ses sommets telle que toutes les arêtes soient « croissantes ».

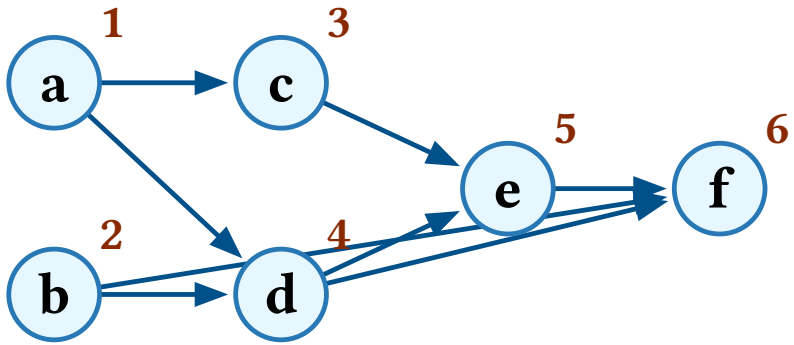
le graphe et l'ordre d'extraction



Question : tri topologique

Question (poly § 2.3) **Tri topologique.** Montrer qu'un graphe **orienté sans cycle** admet un **ordre topologique** c'est-à-dire une numérotation de ses sommets telle que toutes les arêtes soient « croissantes ».

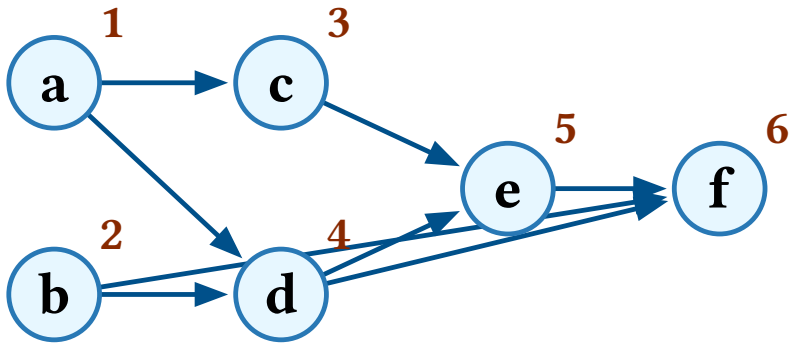
le graphe et l'ordre d'extraction



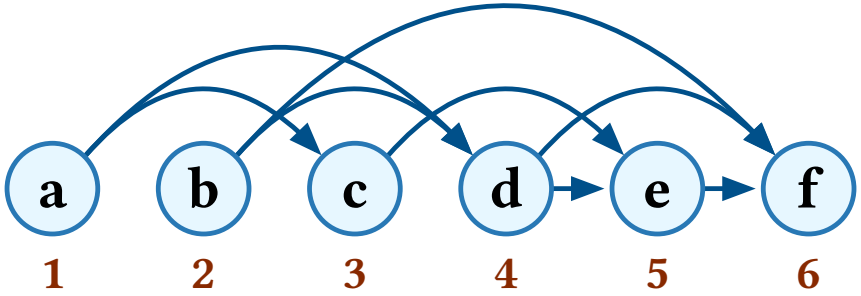
Question : tri topologique

Question (poly § 2.3) **Tri topologique.** Montrer qu'un graphe **orienté sans cycle** admet un **ordre topologique** c'est-à-dire une numérotation de ses sommets telle que toutes les arêtes soient « croissantes ».

le graphe et l'ordre d'extraction



les sommets dans l'ordre topologique



Question : tri topologique (correction)

- il existe un sommet **sans prédécesseur** (remonter les prédécesseurs s'arrête)
- on le numérote
- on le retire
- on recommence

Quand un algorithme glouton est optimal

Patron d'une preuve gloutonne (argument d'échange).

1. une solution optimale O **différente** de la solution gloutonne G
2. la **première différence** : le premier choix de G absent de O
3. **échanger** dans O ce choix contre le choix glouton **sans dégrader** O
4. conclure par **récurrence**

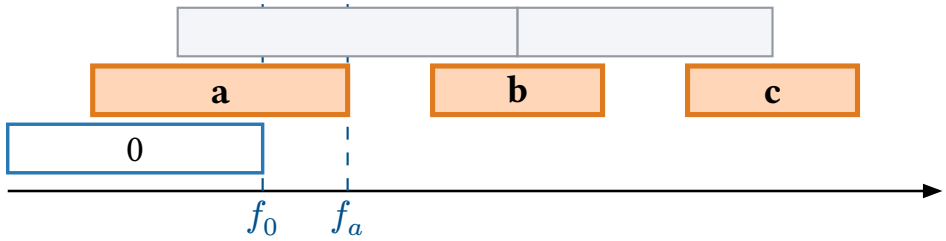
- le point 3 est le seul qui dépend du problème : le premier choix glouton est **sûr**
- sinon : **exhiber un contre-exemple**

Question : « fin la plus tôt » est optimal

Question (poly § 2.4) Les activités sont triées par fin croissante, $f_0 \leq f_1 \leq \ldots \leq f_{n-1}$. Montrer qu'un algorithme glouton « fin la plus tôt » retient un nombre maximal d'activités compatibles.

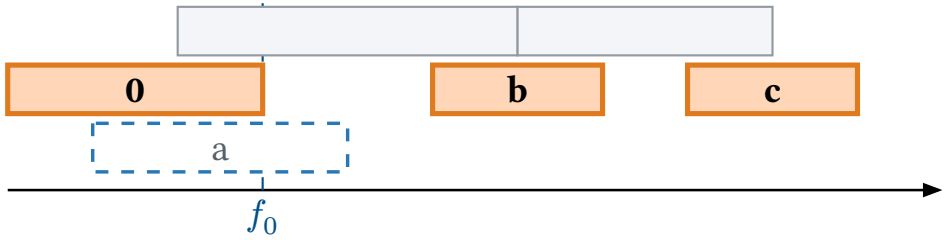
Question : « fin la plus tôt » est optimal (correction)

une solution optimale O (en orange) et la tâche 0, qui finit la première



$f_0 \leq f_a$: ce qui commence après f_a commence après f_0

O où a est remplacée par 0 : encore trois tâches compatibles



même taille : O n'était pas meilleure que le choix glouton

- a : première fin dans O, $f_0 \leq f_a$: échanger a contre 0 garde O compatible, de même taille
- puis **récurrence** (ou par l'absurde, avec un contre-exemple minimal)

▷ Démo 2

`selection_activites` (Python)

```python
def selection_activites(taches):
    """taches : liste de couples (debut, fin), triée par fin croissante."""
    choisies = []
    derniere_fin = -INFINI
    for (d, f) in taches:
        if d >= derniere_fin:            # compatible avec la dernière retenue
            choisies.append((d, f))
            derniere_fin = f
    return choisies
```

- $O(n)$ après le tri, $O(n \ln(n))$ en tout, contre 2^n sous-ensembles

À retenir : glouton et programmation dynamique

À retenir. Glouton et programmation dynamique reposent tous deux sur la **sous-structure optimale** : après le premier choix, on résout le reste de façon optimale. Un algorithme glouton suppose de plus qu'un **premier choix** se reconnaît sans résoudre les sous-problèmes. Quand ce n'est pas le cas, la programmation dynamique **essaie tous les premiers choix** et garde le meilleur.

Un premier exemple : le rendu de monnaie

État, sous-problèmes et récurrence

L'état est la somme qui reste à rendre.

- un algorithme glouton **devine** la première pièce
- la programmation dynamique **les essaie toutes**
- il lui faut le meilleur rendu des sommes plus petites : les **sous-problèmes**

Notation. pour $k \in [\![0, n]\!]$, $f(k)$ est le nombre minimal de pièces de $\mathcal{P}$ nécessaires pour rendre k (l'état est l'entier k).

Question : la récurrence

Question (poly § 3.2) Justifier que $f(0) = 0$ et que, pour $k \geq 1$,

$$f(k) = 1 + \min(\{f(k-p), p \in \mathcal{P} \text{ t.q } p \leq k\}).$$

Question : la récurrence

Question (poly § 3.2) Justifier que $f(0) = 0$ et que, pour $k \geq 1$,

$$f(k) = 1 + \min(\{f(k-p), p \in \mathcal{P} \text{ t.q } p \leq k\}).$$

- « $\leq$ » : un rendu optimal de $k - p$, complété par p, rend k
- « $\geq$ » : dans un rendu optimal de k, retirer une pièce p laisse un rendu de $k - p$, donc $f(k-p) \leq f(k) - 1$: **sous-structure optimale**

Récursion naïve (Python)

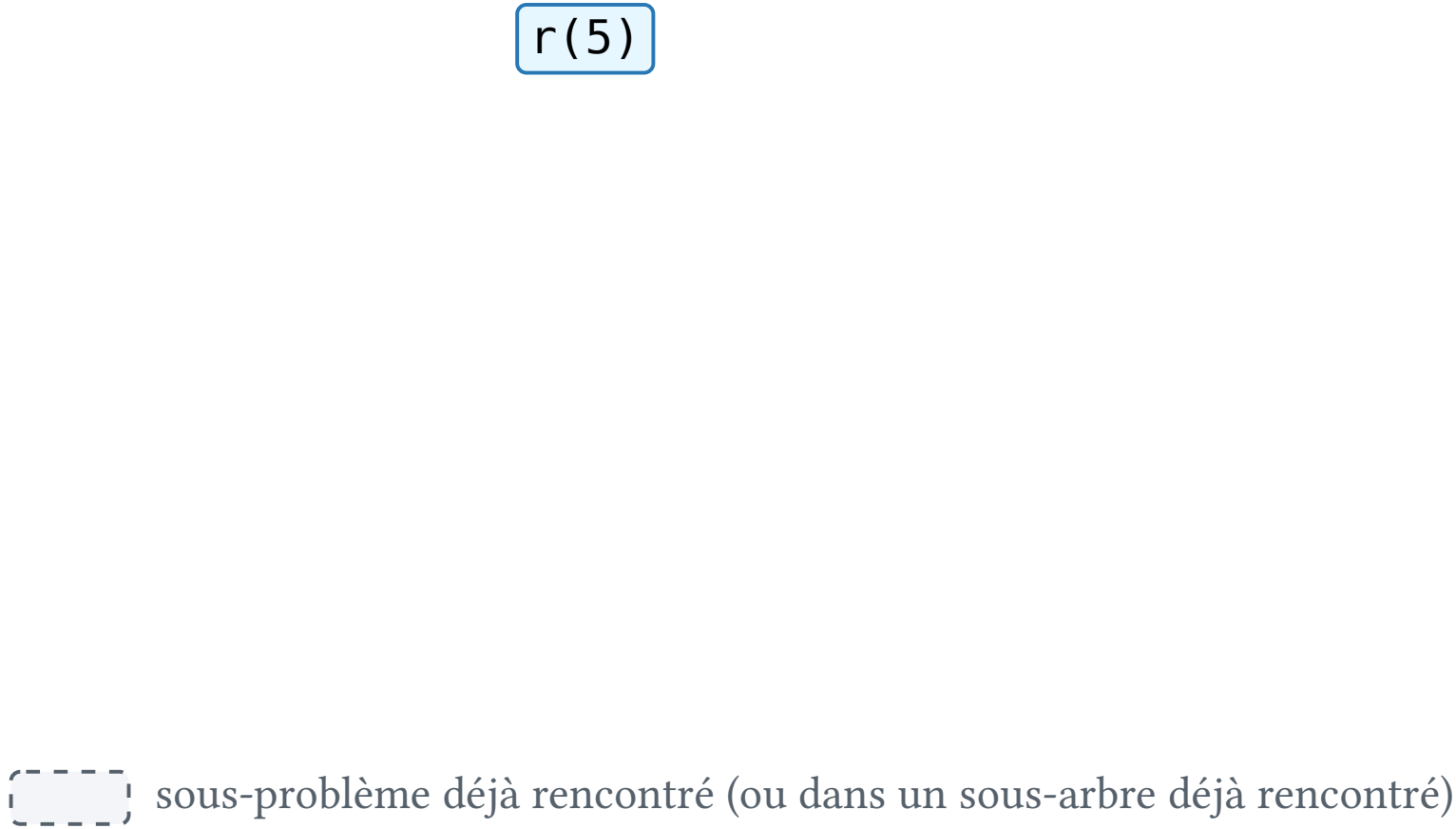
```python
def rendu_naif(n, pieces):
    """Nombre minimal de pièces de `pieces` pour rendre n."""
    if n == 0:
        return 0
    mini = INFINI
    for p in pieces:
        if p <= n:
            r = 1 + rendu_naif(n - p, pieces)
            if r < mini:
                mini = r
    return mini
```

Récursion naïve (Python)

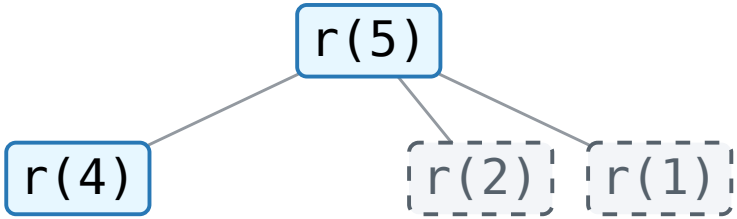
```python
def rendu_naif(n, pieces):
    """Nombre minimal de pièces de `pieces` pour rendre n."""
    if n == 0:
        return 0
    mini = INFINI
    for p in pieces:
        if p <= n:
            r = 1 + rendu_naif(n - p, pieces)
            if r < mini:
                mini = r
    return mini
```

Pour les exemples on prendra `pieces = [1, 3, 4]`.

Récursion naïve : l'arbre des appels

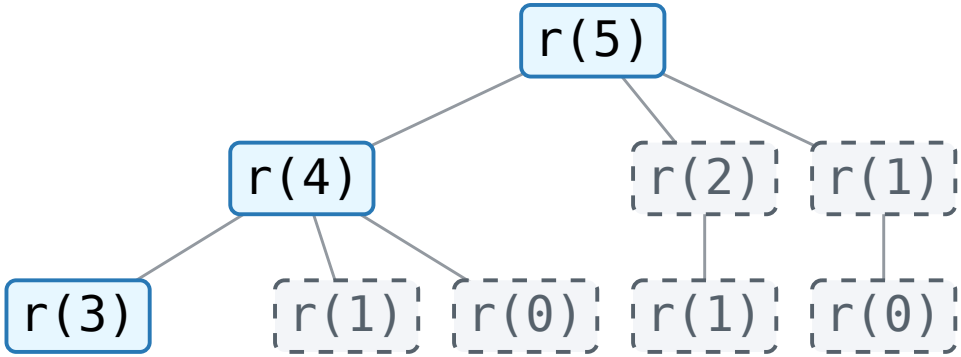


Récursion naïve : l'arbre des appels



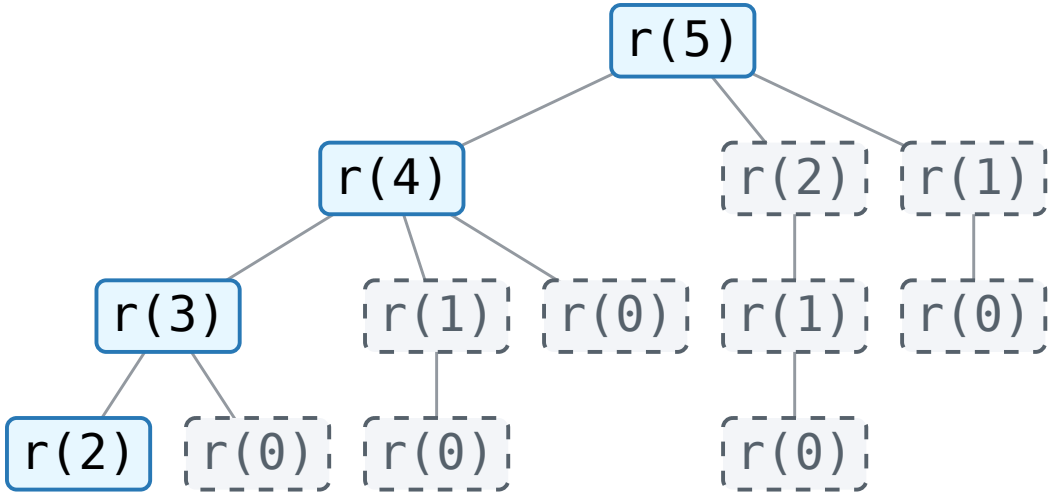
sous-problème déjà rencontré (ou dans un sous-arbre déjà rencontré)

Récursion naïve : l'arbre des appels



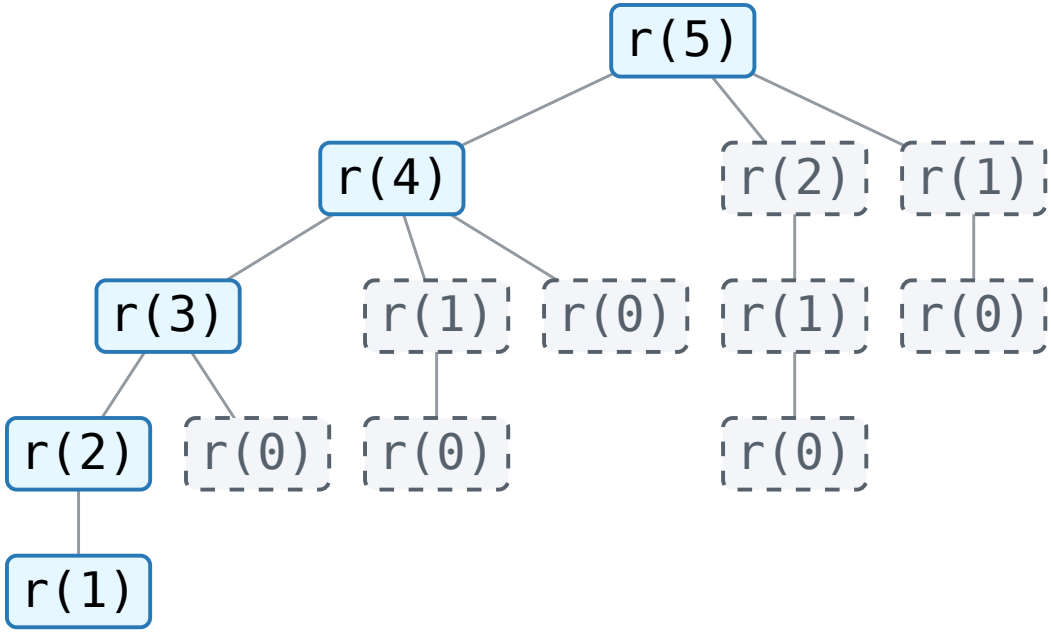
sous-problème déjà rencontré (ou dans un sous-arbre déjà rencontré)

Récursion naïve : l'arbre des appels



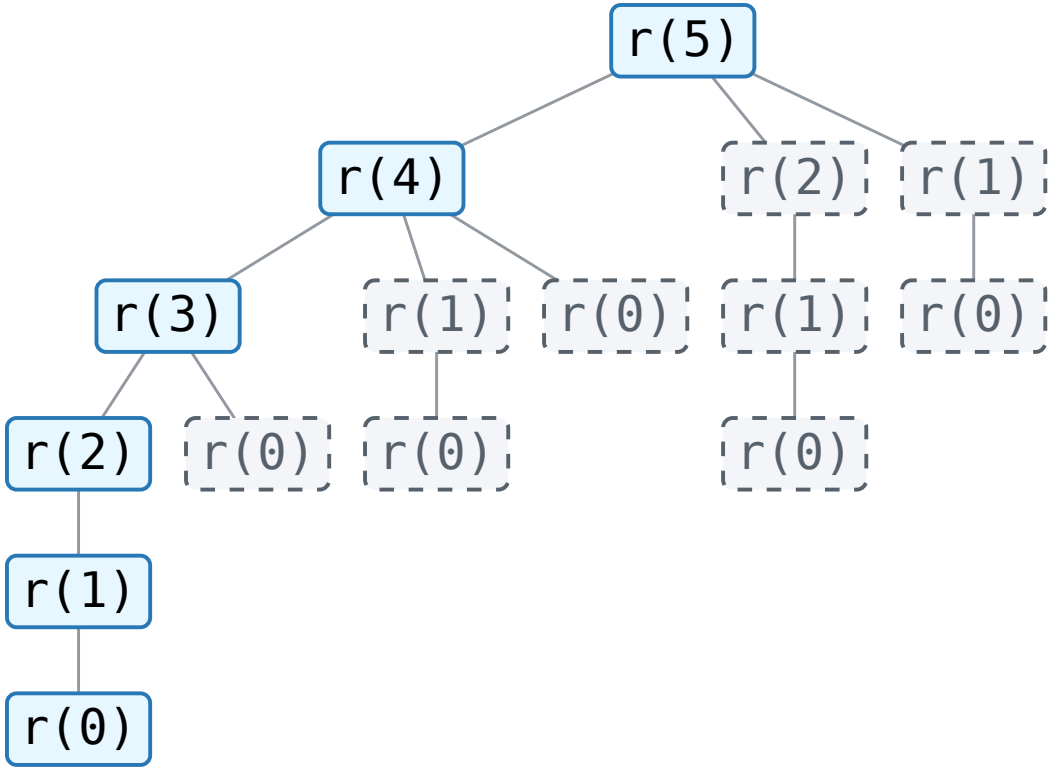
sous-problème déjà rencontré (ou dans un sous-arbre déjà rencontré)

Récursion naïve : l'arbre des appels



sous-problème déjà rencontré (ou dans un sous-arbre déjà rencontré)

Récursion naïve : l'arbre des appels



sous-problème déjà rencontré (ou dans un sous-arbre déjà rencontré)

▷ Démo 3

Question : complexité de `rendu_naif`

Question (poly § 3.3) On note $C(k)$ le nombre d'appels de `rendu_naif(k, [1, 3, 4])`. Donner une récurrence sur C et montrer que la complexité est exponentielle. Combien y a-t-il de sous-problèmes distincts ?

Question : complexité de `rendu_naif`

Question (poly § 3.3) On note $C(k)$ le nombre d'appels de `rendu_naif(k, [1, 3, 4])`. Donner une récurrence sur C et montrer que la complexité est exponentielle. Combien y a-t-il de sous-problèmes distincts ?

- $C(0) = 1, C(k) = 1 + C(k-1) + C(k-3) + C(k-4)$ ($C(k) = 0$ si $k < 0$)
- C croissante : $C(k) \geq 3C(k-4)$, donc $C(k) \geq 3^{\lfloor k/4 \rfloor}$; $C(10) = 168$, $C(40) \approx 3{,}1 \cdot 10^8$
- seulement **$n+1$ sous-problèmes distincts** : $f(0), \ldots, f(n)$, **chevauchement**

Question : mémoïsation

Question (poly § 3.4) Écrire `rendu_memo(n, pieces)`, version mémoïsée de `rendu_naif`. Quelle est sa complexité ?

Question : mémoïsation (correction, 1/2 : la fonction auxiliaire)

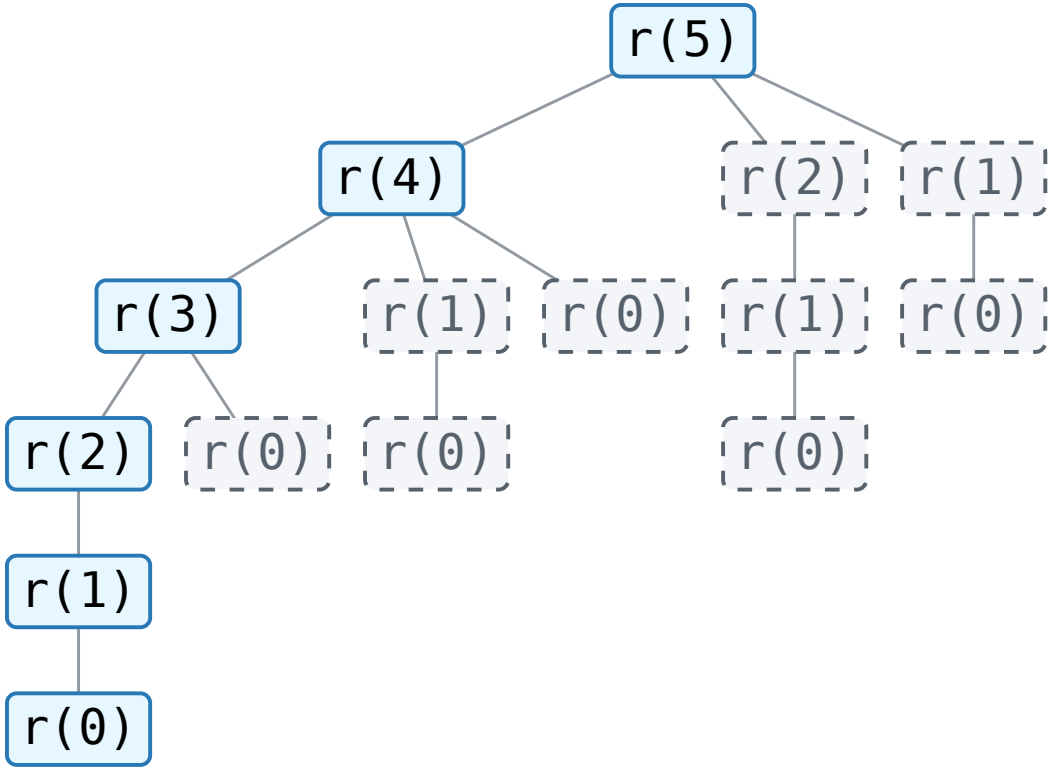
```
def rendu_memo_aux(k, pieces, memo):
    """memo : dictionnaire memo[k] = f(k), rempli au fil des appels."""
    if k == 0:
        memo[0] = 0
    if k in memo:
        return memo[k]
    mini = INFINI
    for p in pieces:
        if p <= k:
            r = 1 + rendu_memo_aux(k - p, pieces, memo)
            if r < mini:
                mini = r
    memo[k] = mini
    return memo[k]
```

▷ Démo 3

Question : mémoïsation (correction, 2/2)

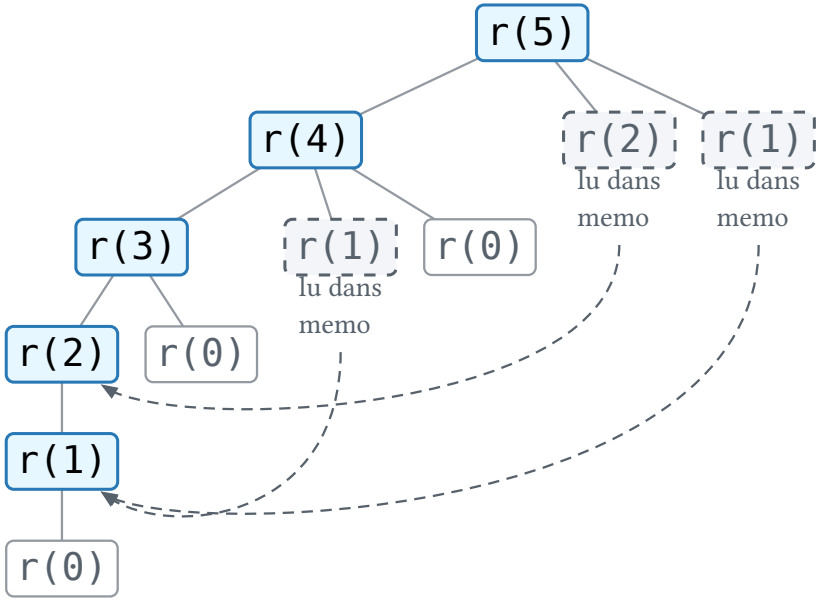
```
def rendu_memo(n, pieces):
    return rendu_memo_aux(n, pieces, {})
```

L'arbre des appels avec `memo`



sous-problème déjà rencontré (ou dans un sous-arbre déjà rencontré)

L'arbre des appels avec `memo`



état déjà mémorisé : valeur lue dans `memo`, sans appel récursif

cas de base `k == 0` : renvoie 0 sans appel récursif

$n = 5$ appels développés, 3 lectures dans le dictionnaire, 3 cas de base : 11 appels au lieu de 15 pour la version naïve

De bas en haut : le graphe des sous-problèmes



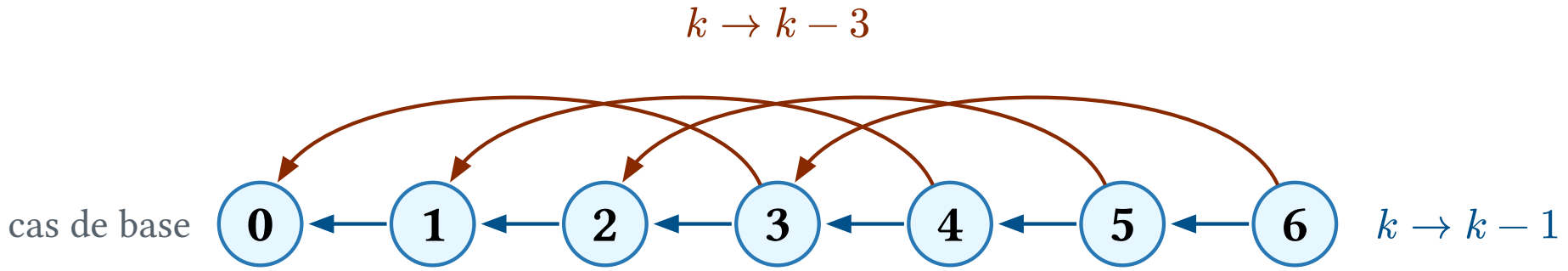
- une arête de k vers chaque $k - p$ dont $f(k)$ a besoin
- un ordre **compatible** : k croissant, chaque **valeur ne dépend que de valeurs déjà calculées**

De bas en haut : le graphe des sous-problèmes



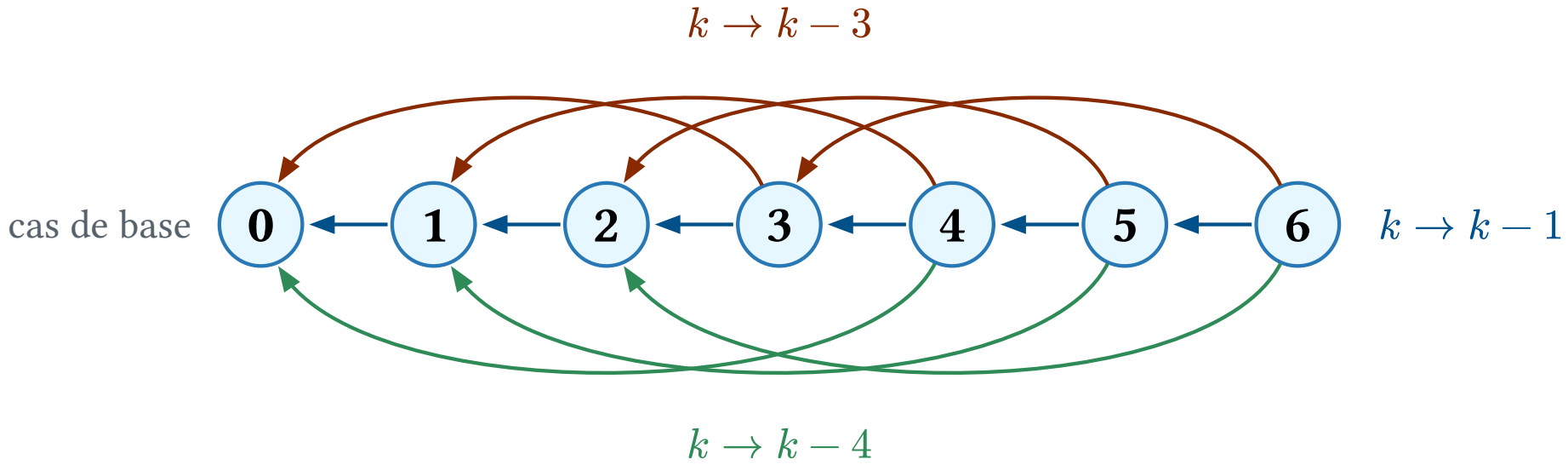
- une arête de k vers chaque $k - p$ dont $f(k)$ a besoin
- un ordre **compatible** : k croissant, chaque **valeur ne dépend que de valeurs déjà calculées**

De bas en haut : le graphe des sous-problèmes



- une arête de k vers chaque $k-p$ dont $f(k)$ a besoin
- un ordre **compatible** : k croissant, chaque **valeur ne dépend que de valeurs déjà calculées**

De bas en haut : le graphe des sous-problèmes



- une arête de k vers chaque $k-p$ dont $f(k)$ a besoin
- un ordre **compatible** : k croissant, chaque **valeur ne dépend que de valeurs déjà calculées**

Question : le tableau

Question (poly § 3.5) Écrire `rendu(n, pieces)` qui calcule $f(n)$ en remplissant une liste `F` telle que `F[k]` vaille $f(k)$, pour k croissant.

Question : le tableau (correction)

```
def rendu(n, pieces):
    F = [INFINI for k in range(n + 1)]
    F[0] = 0
    for k in range(1, n + 1):
        for p in pieces:
            if p <= k and 1 + F[k - p] < F[k]:
                F[k] = 1 + F[k - p]
    return F[n]
```

Question : reconstruire une solution optimale

Question (poly § 3.6) Modifier `rendu` en une fonction `rendu_pieces` pour qu'elle renvoie la liste des pièces d'un rendu optimal.

On remplira une seconde liste `choix` telle que `choix[k]` soit une pièce par laquelle commence un rendu optimal de k.

Question : reconstruire une solution optimale (correction)

```
def rendu_pieces(n, pieces):
    F = [INFINI for k in range(n + 1)]
    choix = [0 for k in range(n + 1)]
    F[0] = 0
    for k in range(1, n + 1):
        for p in pieces:
            if p <= k and 1 + F[k - p] < F[k]:
                F[k] = 1 + F[k - p]
                choix[k] = p                    # la pièce qui réalise le minimum
    rendu = []                                  # reconstruction, de n vers 0
    k = n
    while k > 0:
        rendu.append(choix[k])
        k = k - choix[k]
    return rendu
```

- `choix[k]` : une pièce par laquelle commence un rendu optimal de k, on la retire et on recommence depuis `k - choix[k]`

▷ Démo 3

Reconstruction : `F`, `choix` et la remontée

k	0	1	2	3	4	5	6	7	8	9	10
`F[k]`	0	1	2	1	1						
`choix[k]`	0	1	1	3	4						

▷ Démo 3

Reconstruction : `F`, `choix` et la remontée

k	0	1	2	3	4	5	6	7	8	9	10
`F[k]`	0	1	2	1	1	2	2	2			
`choix[k]`	0	1	1	3	4	1	3	3			

▷ Démo 3

Reconstruction : `F`, `choix` et la remontée

k	0	1	2	3	4	5	6	7	8	9	10
`F[k]`	0	1	2	1	1	2	2	2	2	3	3
`choix[k]`	0	1	1	3	4	1	3	3	4	1	3

▷ Démo 3

Reconstruction : `F`, `choix` et la remontée

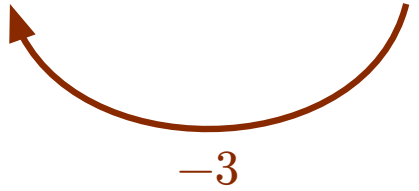
k	0	1	2	3	4	5	6	7	8	9	10
`F[k]`	0	1	2	1	1	2	2	2	2	3	**3**
`choix[k]`	0	1	1	3	4	1	3	3	4	1	**3**

- à rebours, de n vers 0, en $O(n)$: complexité inchangée
- pour chaque k, on stocke une seule pièce (la première d'un rendu optimal), pas la liste complète
- sans `choix` : chercher $p \leq k$ tel que `F[k - p] == F[k] - 1`

▷ Démo 3

Reconstruction : `F`, `choix` et la remontée

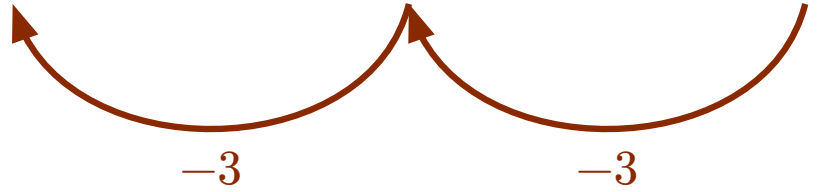
k	0	1	2	3	4	5	6	7	8	9	10
`F[k]`	0	1	2	1	1	2	2	**2**	2	3	**3**
`choix[k]`	0	1	1	3	4	1	3	**3**	4	1	**3**



- à rebours, de n vers 0, en $O(n)$: complexité inchangée
- pour chaque k, on stocke une seule pièce (la première d'un rendu optimal), pas la liste complète
- sans `choix` : chercher $p \leq k$ tel que `F[k - p] == F[k] - 1`

Reconstruction : `F`, `choix` et la remontée

k	0	1	2	3	4	5	6	7	8	9	10
`F[k]`	0	1	2	1	**1**	2	2	**2**	2	3	**3**
`choix[k]`	0	1	1	3	**4**	1	3	**3**	4	1	**3**

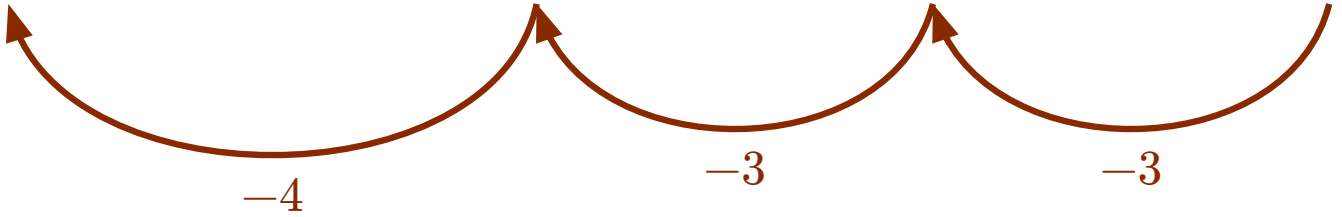


- à rebours, de n vers 0, en $O(n)$: complexité inchangée
- pour chaque k, on stocke une seule pièce (la première d'un rendu optimal), pas la liste complète
- sans `choix` : chercher $p \leq k$ tel que `F[k - p] == F[k] - 1`

▷ Démo 3

Reconstruction : `F`, `choix` et la remontée

k	0	1	2	3	4	5	6	7	8	9	10
`F[k]`	**0**	1	2	1	**1**	2	2	**2**	2	3	**3**
`choix[k]`	**0**	1	1	3	**4**	1	3	**3**	4	1	**3**



rendu : 3 + 3 + 4

- à rebours, de n vers 0, en $O(n)$: complexité inchangée
- pour chaque k, on stocke une seule pièce (la première d'un rendu optimal), pas la liste complète
- sans `choix` : chercher $p \leq k$ tel que `F[k - p] == F[k] - 1`

Méthode : les cinq temps

Méthode.

1. **États.** Une **phrase en français** : « $f(k)$ est le nombre minimal de pièces pour rendre k », et où est la réponse ($f(n)$).
2. **Récurrence.** Cas de base, puis équation de Bellman, justifiée par la sous-structure optimale.
3. **Calcul.** Mémoïsation ou tableau rempli dans un ordre compatible.
4. **Complexité.** Temps = nombre d'états $\times$ coût d'un état, mémoire = états conservés.
5. **Reconstruction.** Mémoriser le choix gagnant (ou le retrouver), remonter de l'état final vers un cas de base.

Les ingrédients de la programmation dynamique

Sous-structure optimale

Définition. Un problème d'optimisation a la propriété de **sous-structure optimale** lorsqu'une solution optimale est formée de solutions optimales de sous-problèmes : retirer à une solution optimale son premier choix laisse une solution optimale du sous-problème.

- pour choisir le premier choix, il suffit de connaître la **valeur optimale** de chaque sous-problème, **et rien d'autre**
- preuve par l'absurde : un reste non optimal s'améliorerait, et la solution avec

Attention : tous les problèmes n'ont pas cette propriété

- **plus long chemin simple** : un plus long chemin $s \to v \to t$ n'est pas formé de deux plus longs chemins (sommets partagés)
- **sous-tableau de produit maximal** : $(-2, 3, -4)$, la meilleure tranche finissant en 2 prolonge la tranche de produit **minimal** -6

- la programmation dynamique sert aussi à **dénombrer** (escaliers, grilles), **décider** (découper un texte en mots), **calculer une position**

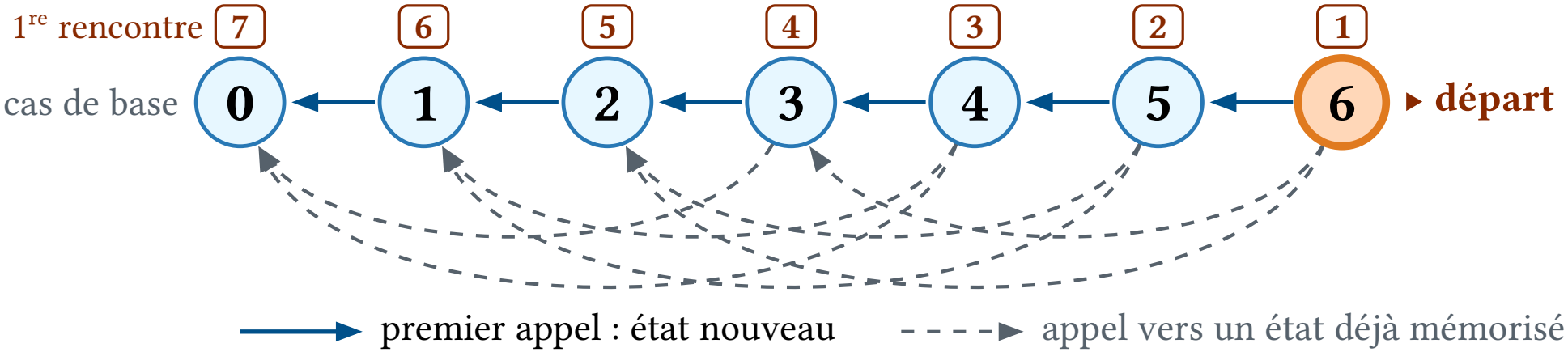
Chevauchement des sous-problèmes

Définition. Il y a **chevauchement des sous-problèmes** lorsque la récursion naïve résout plusieurs fois les mêmes sous-problèmes. La mémorisation est rentable quand le nombre de sous-problèmes **distincts** est bien plus petit que le nombre d'appels : polynomial contre exponentiel.

- de bas en haut (tableau, parcours des états dans l'ordre inverse d'un ordre topologique)
- de haut en bas (mémoïsation, dictionnaire, parcours des états en profondeur)

De haut en bas ou de bas en haut ?

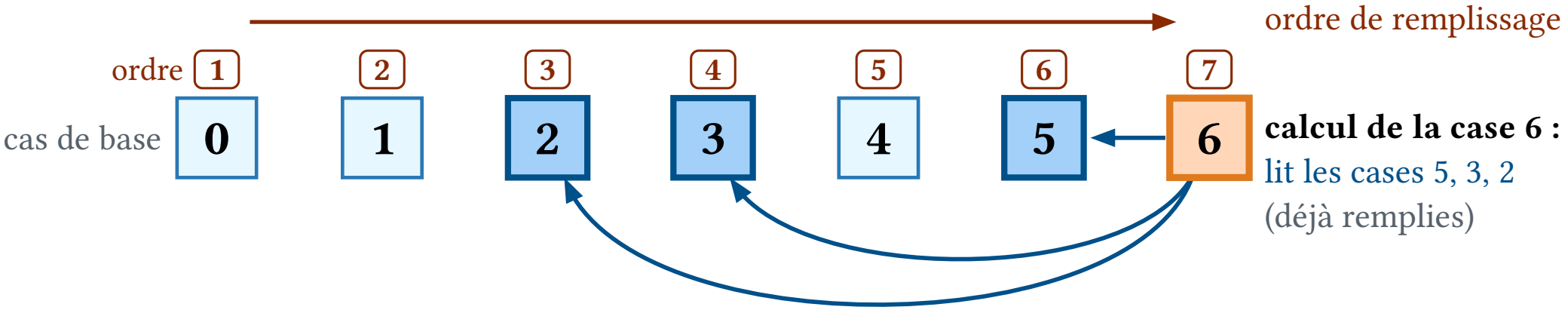
de haut en bas (mémoïsation) : on part de la question



ordre de première rencontre : 6, 5, 4, 3, 2, 1, 0

De haut en bas ou de bas en haut ?

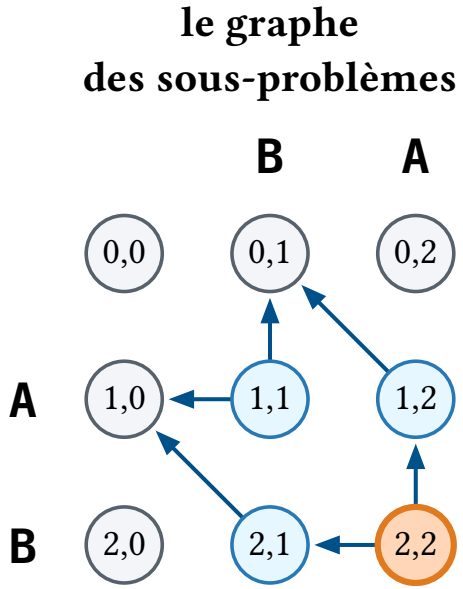
de bas en haut (tableau) : on part des cas de base



De haut en bas ou de bas en haut ? Comparaison

	De haut en bas (mémoïsation)	**De bas en haut** (tableau)
écriture	la fonction naïve plus un dictionnaire	il faut trouver un **ordre de calcul**
états calculés	seulement ceux **atteints**	tous, y compris les inutiles
mémoire	dictionnaire + **pile d'appels** (`RecursionError`)	tableau, souvent réductible à une ligne
vitesse	hachage, appels de fonction	boucles sur des listes : plus rapide

Remarque théorique : un graphe sans cycle

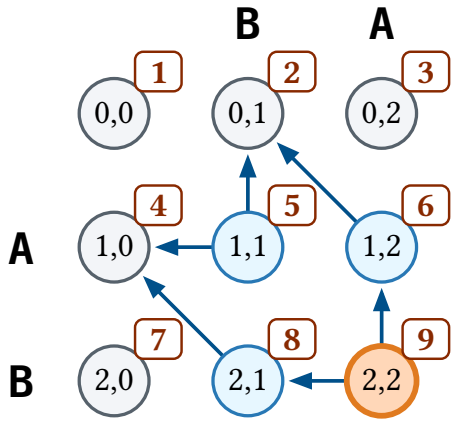


une arête : « a besoin de »

- bas en haut = ordre topologique **inverse**, choisi à la main
- haut en bas = **parcours en profondeur**, ordre de fin
- un sous-état par choix : un **plus court** (ou long) **chemin**

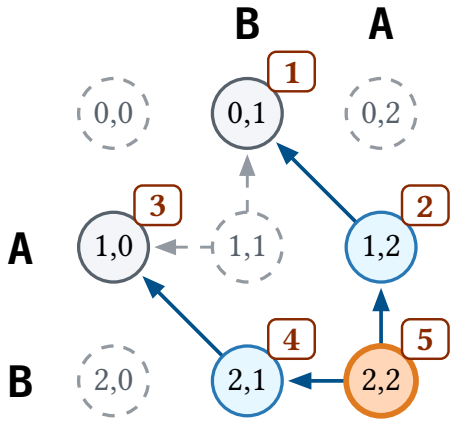
Remarque théorique : un graphe sans cycle

de bas en haut :
un ordre topologique inverse



ordre de remplissage :
chaque arête va vers un numéro plus petit

de haut en bas : parcours
en profondeur depuis (2,2)



ordre de **fin** de calcul,
en pointillé : jamais visité

- bas en haut = ordre topologique **inverse**, choisi à la main
- haut en bas = **parcours en profondeur**, ordre de fin
- un sous-état par choix : un **plus court** (ou long) **chemin**

Calculer une complexité

Méthode.

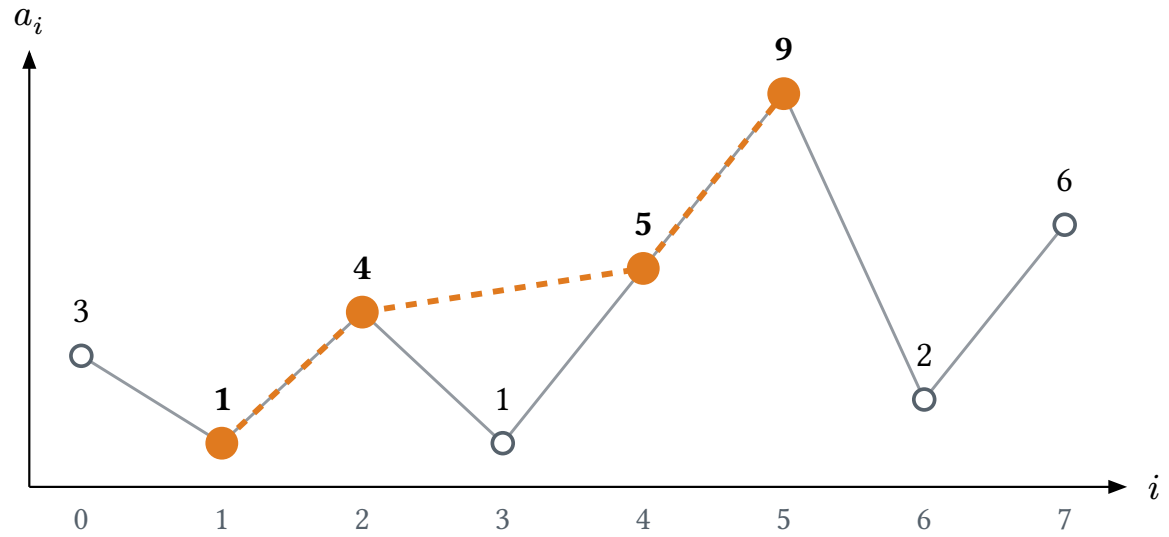
$$\text{temps} = (\textbf{nombre d'états}) \times (\textbf{coût du calcul d'un état})$$

lire un autre état (case, appel déjà dans `memo`) compte pour $O(1)$.

- **tous les paramètres** : sac à dos $O(nb)$
- **les fonctions appelées comptent** : deux boucles ne font pas $O(n^2)$ si le corps n'est pas $O(1)$
- **mémoïsation** : nombre de clés **atteintes**
- **reconstruction** : longueur du chemin $\times$ coût d'un choix, à annoncer
- **mémoire** : états conservés, en mémoïsation, dictionnaire **et** pile

La plus longue sous-suite croissante (LIS) : choisir ses états

Le problème



- longueur maximale d'une sous-suite **strictement croissante**
- enjeu: **le choix des états**

Question : solution très naïve

Question (poly § 5.2) Écrire `sous_suites(a)` qui renvoie la liste de toutes les sous-suites de `a` (comme listes), puis `lis_tres_naif(a)` qui renvoie la longueur maximale d'une sous-suite strictement croissante. Quelle est la complexité ?

Question : solution très naïve (correction, 1/2)

```
def sous_suites(a):
    if len(a) == 0:
        return [[]]
    sans = sous_suites(a[1:])
    avec = [[a[0]] + s for s in sans]
    return sans + avec

def est_croissante(s):
    for k in range(len(s) - 1):
        if s[k] >= s[k + 1]:
            return False
    return True
```

- sous-suites de `a` = celles de `a[1:]`, avec ou sans `a[0]` devant

Question : solution très naïve (correction, 2/2)

```
def lis_tres_naif(a):
    meilleure = 0
    for s in sous_suites(a):
        if est_croissante(s) and len(s) > meilleure:
            meilleure = len(s)
    return meilleure
```

Question : solution très naïve (correction, 2/2)

```
def lis_tres_naif(a):
    meilleure = 0
    for s in sous_suites(a):
        if est_croissante(s) and len(s) > meilleure:
            meilleure = len(s)
    return meilleure
```

2^n sous-suites de longueur $\leq n$: $O(n2^n)$ en temps et en mémoire, inutilisable en pratique (même optimisé)

Question : algorithme glouton ?

Question (poly § 5.3) Proposer un algorithme glouton pour ce problème. Est-il optimal ?

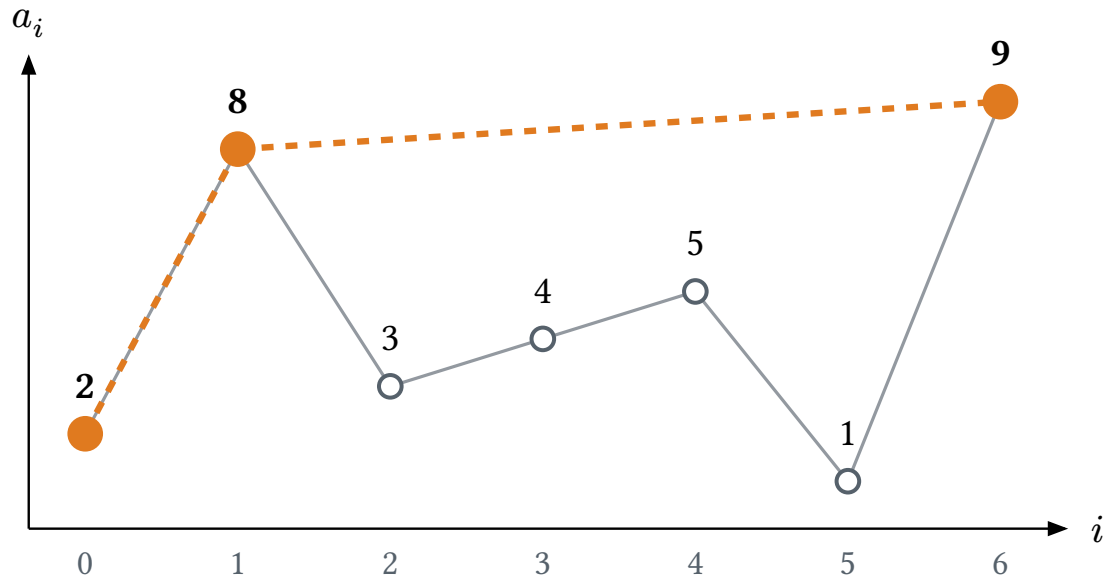
▷ Démo 5

Question : algorithme glouton ? (correction)

```
def lis_glouton(a):
    sous_suite = []
    dernier = -INFINI
    for x in a:
        if x > dernier:
            sous_suite.append(x)
            dernier = x
    return sous_suite
```

- glouton: prendre le premier terme dépassant le dernier pris
- complexité: $O(n)$

Algorithme glouton : un contre-exemple



- $(2, 8, 9)$ au lieu de $(2, 3, 4, 5, 9)$: prendre 8 ferme la porte à $3, 4, 5$

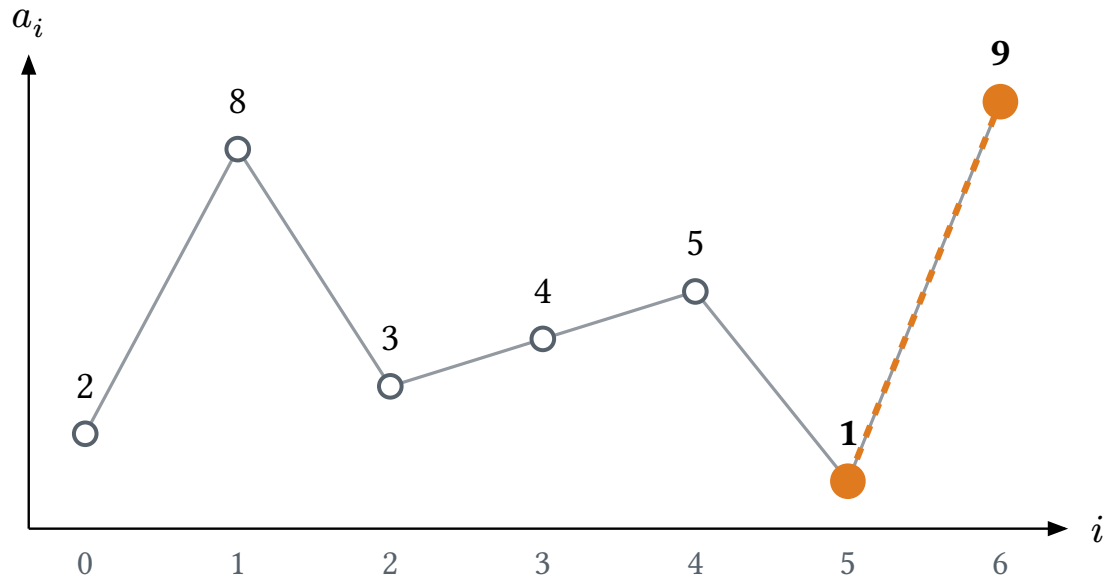
▷ Démo 5

Un autre algorithme glouton (Python)

```python
def lis_glouton_min(a):
    sous_suite = []
    dernier = -INFINI
    i = 0                              # les candidats sont dans a[i:]
    while i < len(a):
        m = -1                         # indice du plus petit candidat
        for j in range(i, len(a)):
            if a[j] > dernier and (m == -1 or a[j] < a[m]):
                m = j
        if m == -1:                    # plus aucun candidat
            return sous_suite
        sous_suite.append(a[m])
        dernier = a[m]
        i = m + 1
    return sous_suite
```

- glouton: prendre le plus petit terme possible
- complexité: $O(n^2)$

Autre algorithme glouton : un contre-exemple



- $(1, 9)$ au lieu de $(2, 3, 4, 5, 9)$: le minimum 1 est vers la fin

Choisir les états : sous-suites se terminant en i

L'état est l'indice i du dernier terme de la sous-suite.

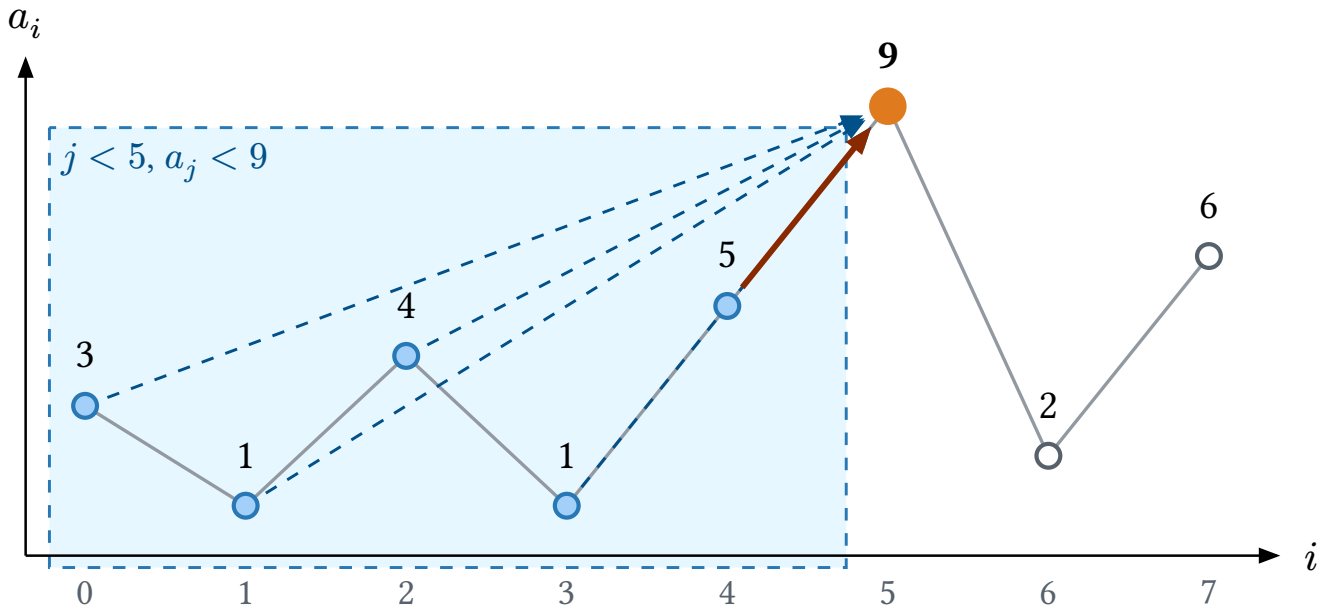
Notation. $L(i)$ est la longueur maximale d'une sous-suite strictement croissante **se terminant exactement** par le terme a_i.

Question (poly § 5.4) Justifier que

$$L(i) = 1 + \max\big(\{L(j),\ j < i \text{ t.q } a_j < a_i\}\big),$$

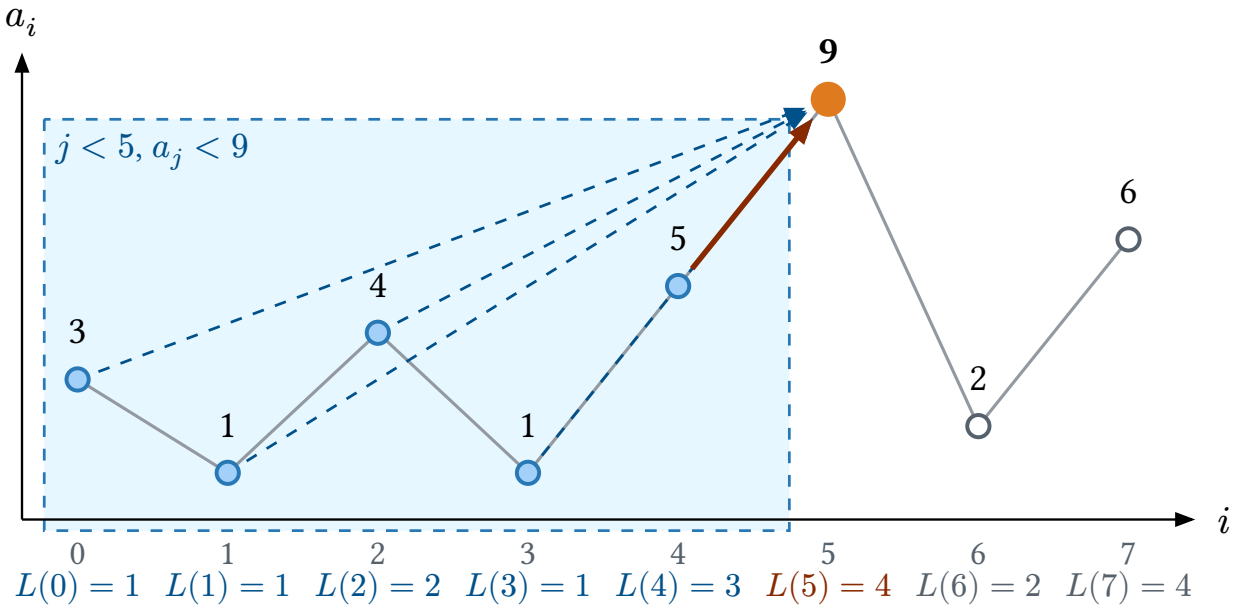
avec la convention $\max(\varnothing) = 0$. Où se trouve la réponse au problème ?

Question : la récurrence sur $L(i)$ (correction)



- une sous-suite optimale finissant en i : (a_i) seul, ou une sous-suite optimale finissant en $j < i$, $a_j < a_i$, puis a_i
- réponse : $\max_i L(i)$ (0 si la suite est vide)

Question : la récurrence sur $L(i)$ (correction)



- une sous-suite optimale finissant en i : (a_i) seul, ou une sous-suite optimale finissant en $j < i$, $a_j < a_i$, puis a_i
- réponse : $\max_i L(i)$ (0 si la suite est vide)

`lis(a)` (Python, 1/2 : le calcul des $L(i)$)

```
def lis(a):
    n = len(a)
    if n == 0:
        return []
    L = [1 for i in range(n)]            # L[i] : longueur max. terminant en i
    pred = [-1 for i in range(n)]        # pred[i] : prédécesseur de i
    for i in range(n):
        for j in range(i):
            if a[j] < a[i] and L[j] + 1 > L[i]:
                L[i] = L[j] + 1
                pred[i] = j
```

- code donné dans le poly ; n états dans l'ordre des indices, chacun en $O(i)$: temps $O(n^2)$, mémoire $O(n)$

`lis(a)` (Python, 2/2 : la reconstruction)

```
    fin = 0                                 # fin d'une sous-suite optimale
    for i in range(n):
        if L[i] > L[fin]:
            fin = i
    sous_suite = []                         # reconstruction à rebours
    i = fin
    while i != -1:
        sous_suite.append(a[i])
        i = pred[i]
    return sous_suite[::-1]                 # construite à l'envers
```

▷ Démo 5

Les $L(i)$, les prédécesseurs et la reconstruction

`pred[i]` : indice précédent dans une sous-suite optimale terminant en i

i	0	1	2	3	4	5	6	7
a_i	3	1	4	1	5	9	2	6
$L(i)$	1							

▷ Démo 5

Les $L(i)$, les prédécesseurs et la reconstruction

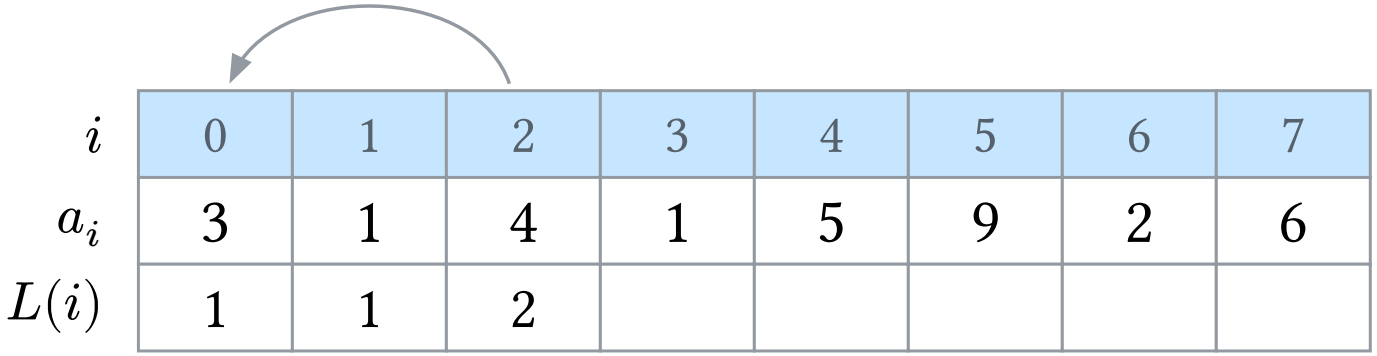
`pred[i]` : indice précédent dans une sous-suite optimale terminant en i

i	0	1	2	3	4	5	6	7
a_i	3	1	4	1	5	9	2	6
$L(i)$	1	1						

▷ Démo 5

Les $L(i)$, les prédécesseurs et la reconstruction

`pred[i]` : indice précédent dans une sous-suite optimale terminant en i

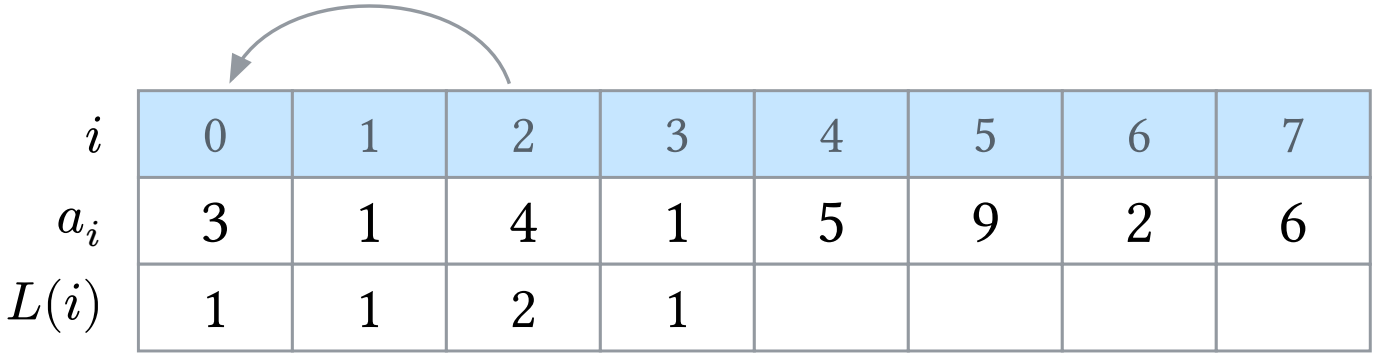


i	0	1	2	3	4	5	6	7
a_i	3	1	4	1	5	9	2	6
$L(i)$	1	1	2					

▷ Démo 5

Les $L(i)$, les prédécesseurs et la reconstruction

`pred[i]` : indice précédent dans une sous-suite optimale terminant en i

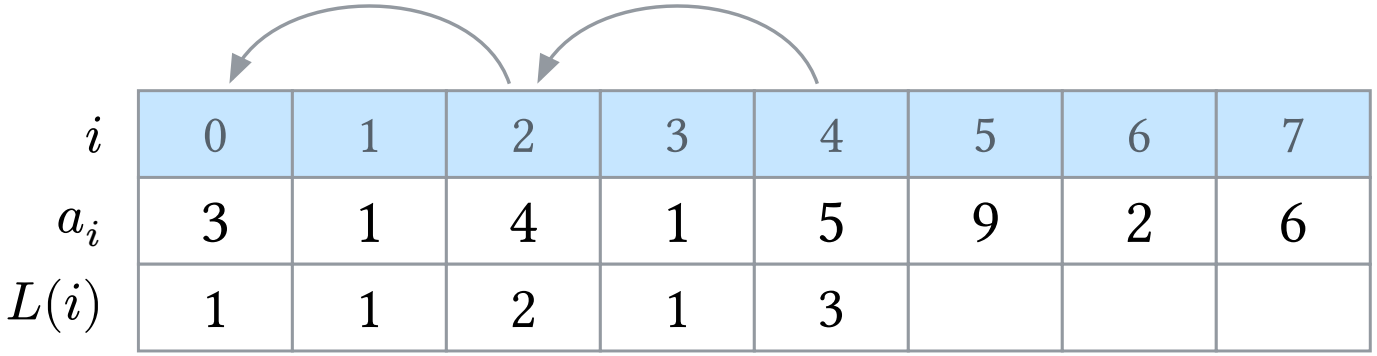


i	0	1	2	3	4	5	6	7
a_i	3	1	4	1	5	9	2	6
$L(i)$	1	1	2	1				

▷ Démo 5

Les $L(i)$, les prédécesseurs et la reconstruction

`pred[i]` : indice précédent dans une sous-suite optimale terminant en i

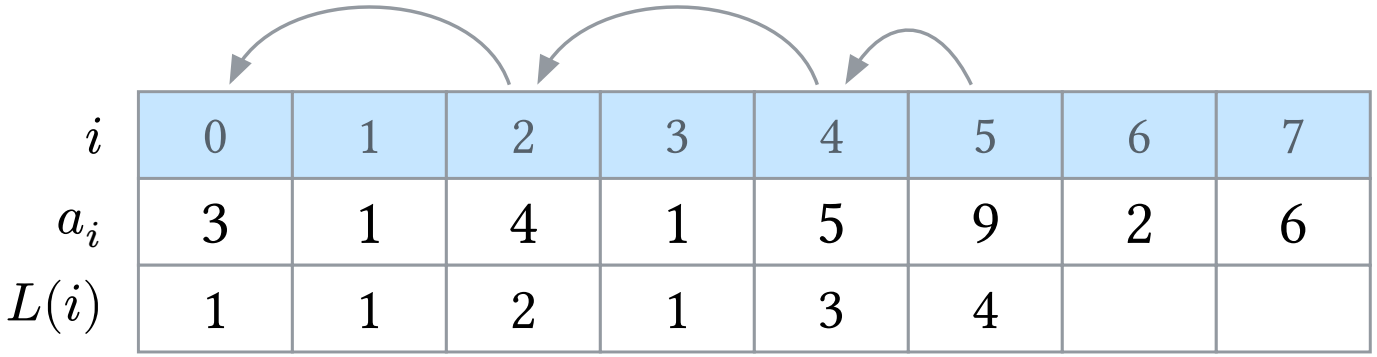


i	0	1	2	3	4	5	6	7
a_i	3	1	4	1	5	9	2	6
$L(i)$	1	1	2	1	3			

▷ Démo 5

Les $L(i)$, les prédécesseurs et la reconstruction

`pred[i]` : indice précédent dans une sous-suite optimale terminant en i

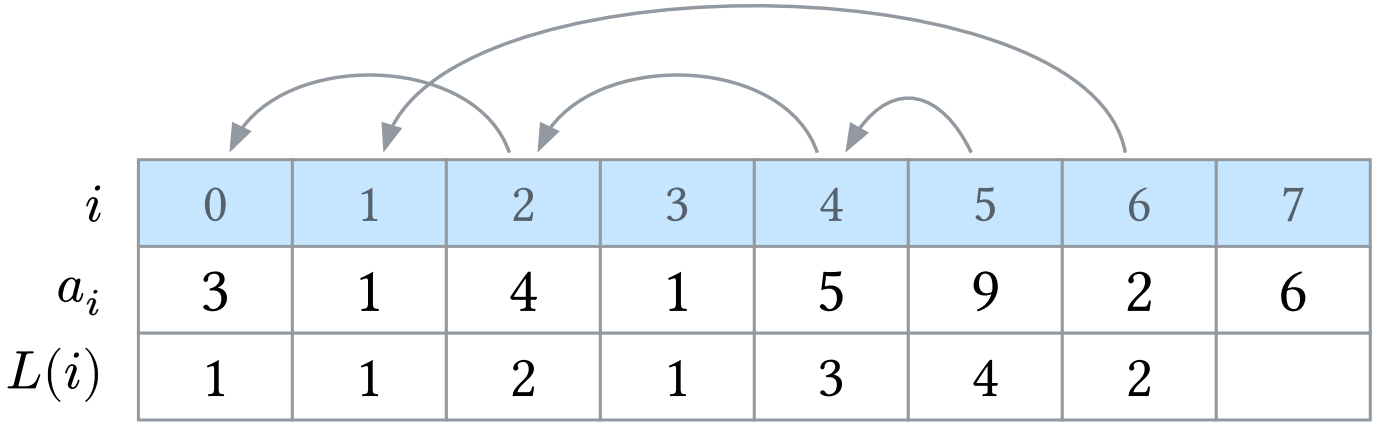


i	0	1	2	3	4	5	6	7
a_i	3	1	4	1	5	9	2	6
$L(i)$	1	1	2	1	3	4		

▷ Démo 5

Les $L(i)$, les prédécesseurs et la reconstruction

`pred[i]` : indice précédent dans une sous-suite optimale terminant en i

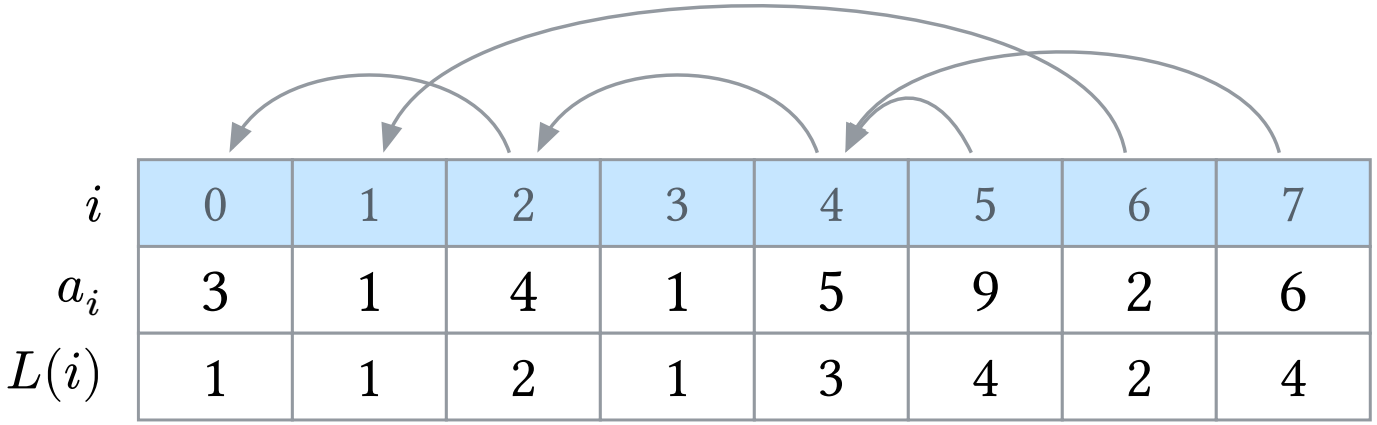


i	0	1	2	3	4	5	6	7
a_i	3	1	4	1	5	9	2	6
$L(i)$	1	1	2	1	3	4	2	

▷ Démo 5

Les $L(i)$, les prédécesseurs et la reconstruction

`pred[i]` : indice précédent dans une sous-suite optimale terminant en i

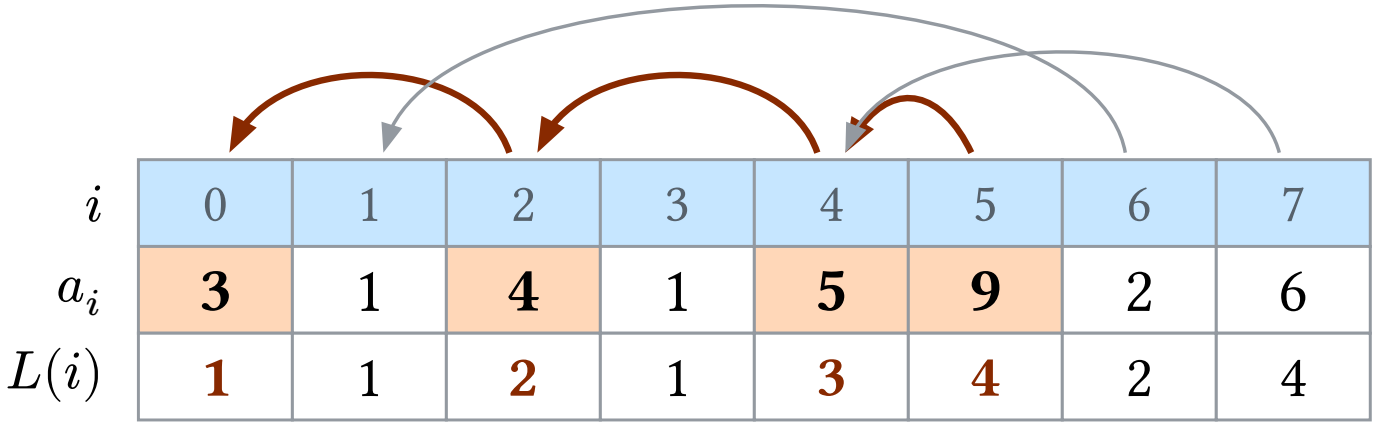


i	0	1	2	3	4	5	6	7
a_i	3	1	4	1	5	9	2	6
$L(i)$	1	1	2	1	3	4	2	4

▷ Démo 5

Les $L(i)$, les prédécesseurs et la reconstruction

`pred[i]` : indice précédent dans une sous-suite optimale terminant en i

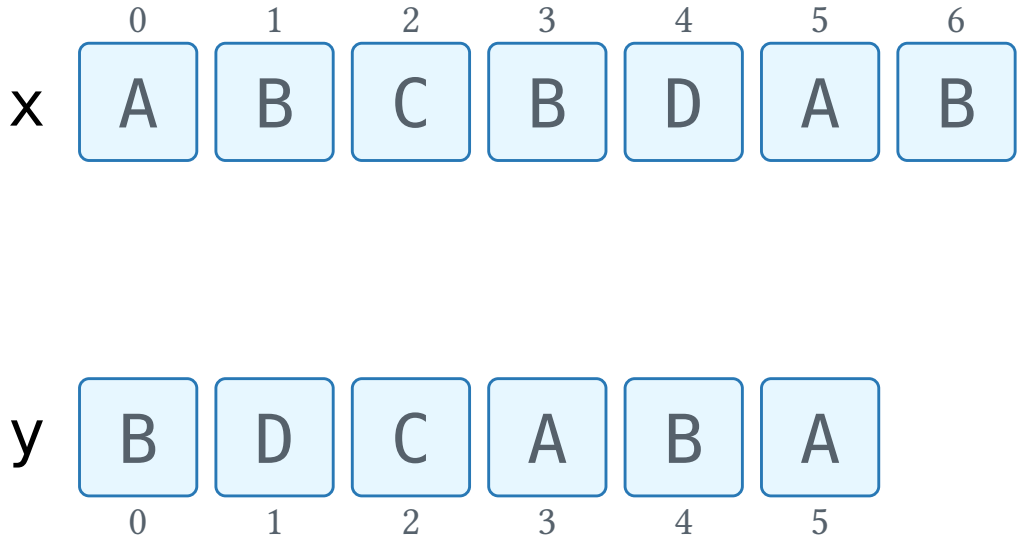


i	0	1	2	3	4	5	6	7
a_i	**3**	1	**4**	1	**5**	**9**	2	6
$L(i)$	**1**	1	**2**	1	**3**	**4**	2	4

- remontée depuis le **maximum** des $L(i)$, ici $i = 5$, en suivant `pred`

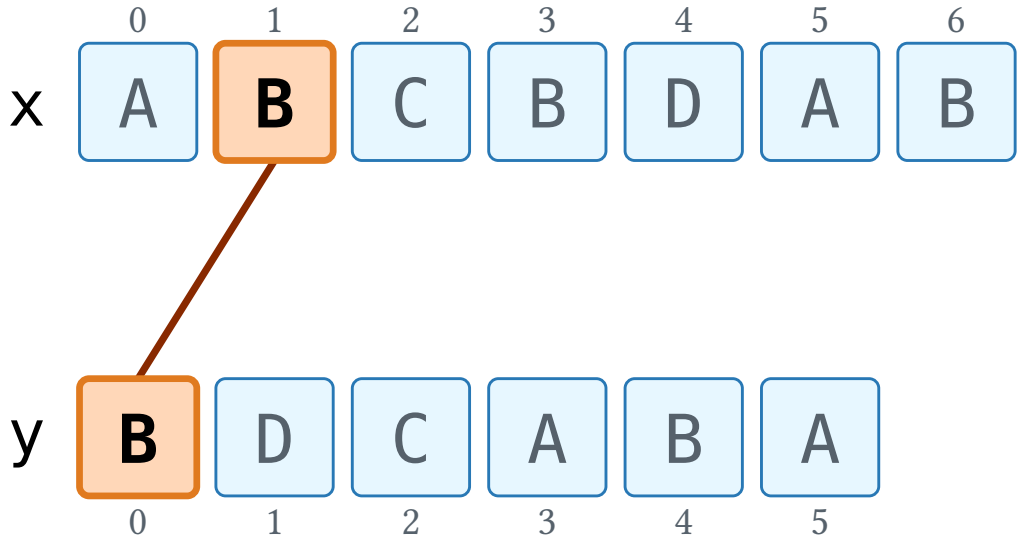
Deux préfixes : la plus longue sous-suite commune

Le problème



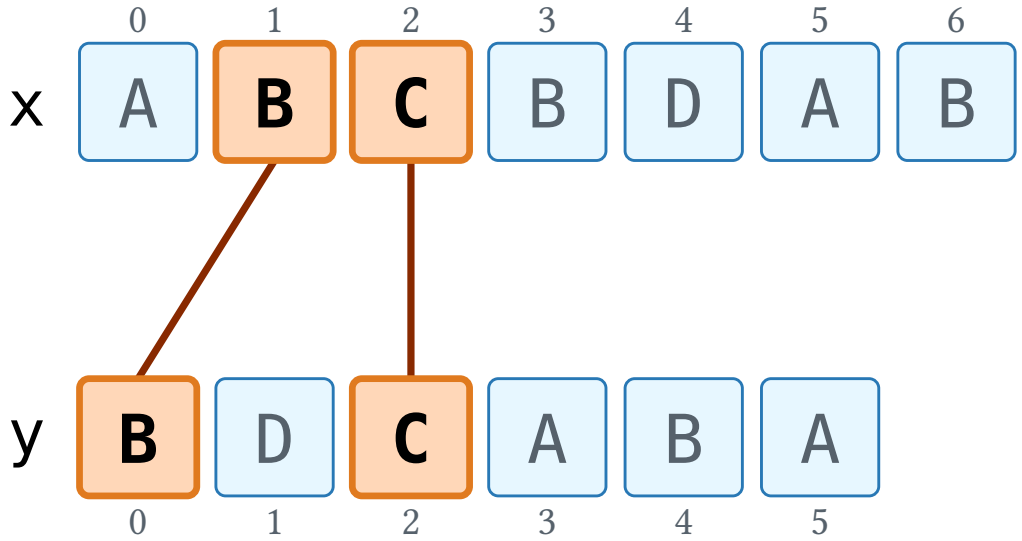
- sous-suite **commune** à $x = x_0 \ldots x_{n-1}$ et $y = y_0 \ldots y_{m-1}$; longueur maximale (PLSC) : ressemblance de x et y
- `ABCBDAB` et `BDCABA` : `BCBA`, longueur 4

Le problème



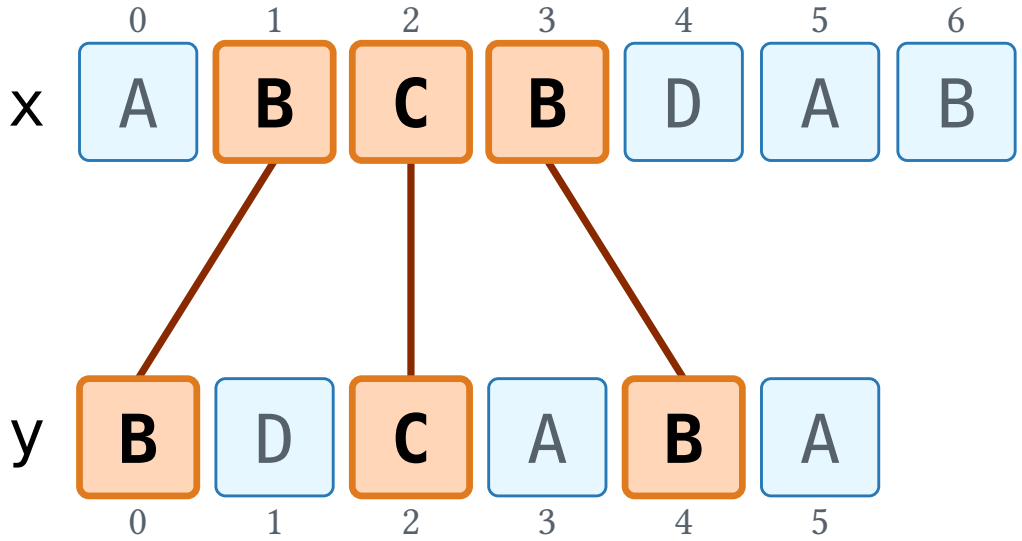
- sous-suite **commune** à $x = x_0 \ldots x_{n-1}$ et $y = y_0 \ldots y_{m-1}$; longueur maximale (PLSC) : ressemblance de x et y
- `ABCBDAB` et `BDCABA` : `BCBA`, longueur 4

Le problème



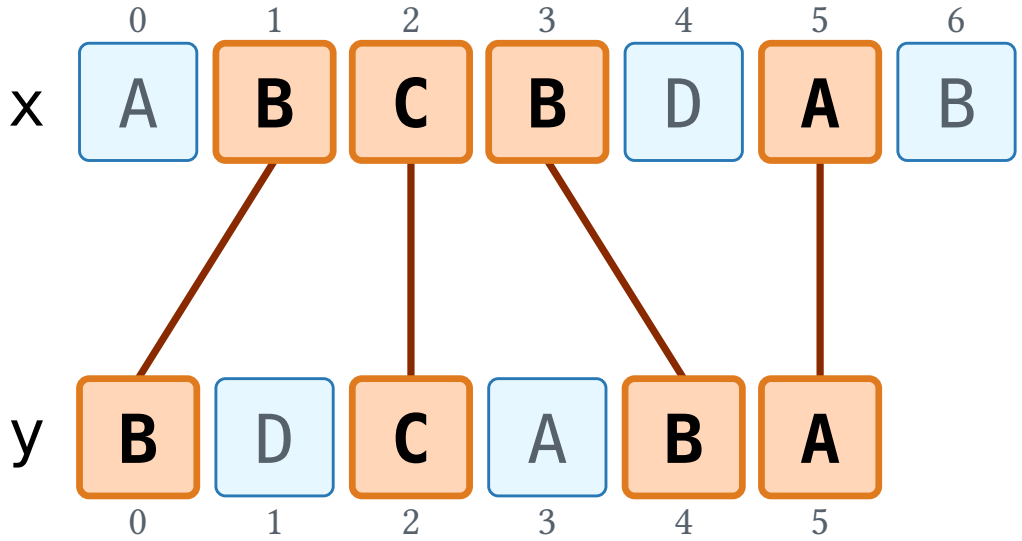
- sous-suite **commune** à $x = x_0 \ldots x_{n-1}$ et $y = y_0 \ldots y_{m-1}$; longueur maximale (PLSC) : ressemblance de x et y
- `ABCBDAB` et `BDCABA` : `BCBA`, longueur 4

Le problème



- sous-suite **commune** à $x = x_0 \ldots x_{n-1}$ et $y = y_0 \ldots y_{m-1}$; longueur maximale (PLSC) : ressemblance de x et y
- `ABCBDAB` et `BDCABA` : `BCBA`, longueur 4

Le problème



sous-suite commune :
BCBA (longueur 4)

- sous-suite **commune** à $x = x_0 \ldots x_{n-1}$ et $y = y_0 \ldots y_{m-1}$; longueur maximale (PLSC) : ressemblance de x et y
- `ABCBDAB` et `BDCABA` : `BCBA`, longueur 4

Recherche exhaustive (Python)

```
def est_sous_suite(z, y):
    k = 0                                  # prochaine lettre de z à trouver
    for lettre in y:
        if k < len(z) and lettre == z[k]:
            k = k + 1
    return k == len(z)

def plsc_exhaustif(x, y):
    meilleure = 0
    for z in sous_suites(x):
        if len(z) > meilleure and est_sous_suite(z, y):
            meilleure = len(z)
    return meilleure
```

- code donné dans le poly : apparier chaque lettre de z **au plus tôt** dans y

Question : complexité de la recherche exhaustive

Question (poly § 6.1) Quelle est la complexité de `plsc_exhaustif` ?

Question : complexité de la recherche exhaustive

Question (poly § 6.1) Quelle est la complexité de `plsc_exhaustif` ?

2^n sous-suites de x, test en $O(m)$: $O((n+m)2^n)$, mémoire $O(n2^n)$

Question : un algorithme glouton

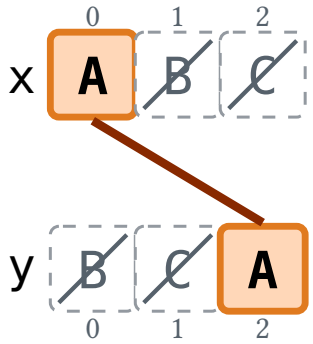
Question (poly § 6.1) Un algorithme glouton : parcourir x et apparier chaque lettre à sa **première** occurrence encore disponible dans y. L'écrire et montrer qu'il n'est pas optimal.

Question : un algorithme glouton (correction)

```
def plsc_glouton(x, y):
    commune = []
    k = 0                                  # position courante dans y
    for lettre in x:
        j = k
        while j < len(y) and y[j] != lettre:
            j = j + 1
        if j < len(y):                     # trouvée : on l'apparie
            commune.append(lettre)
            k = j + 1
    return commune
```

Question : un algorithme glouton (contre-exemple)

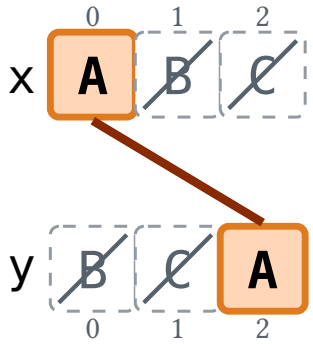
glouton : chaque lettre de x au plus tôt dans y



A apparié au plus tôt, les lettres d'avant sont sautées :
B et **C** ne trouvent plus rien
longueur **1**

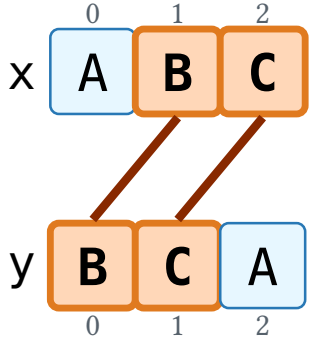
Question : un algorithme glouton (contre-exemple)

glouton : chaque lettre de x au plus tôt dans y



A apparié au plus tôt, les lettres d'avant sont sautées :
B et **C** ne trouvent plus rien
longueur **1**

optimum



BC est commune
longueur **2**

États et récurrence

L'état est un couple de préfixes, un de chaque mot.

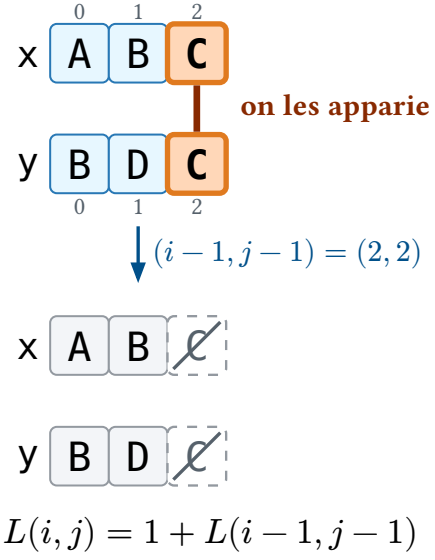
Notation. pour $0 \leq i \leq n$ et $0 \leq j \leq m$, $L(i, j)$ est la longueur d'une PLSC de $x_0 \ldots x_{i-1}$ et $y_0 \ldots y_{j-1}$ (`x[:i]` et `y[:j]`). La réponse est $L(n, m)$.

Question (poly § 6.2) Donner les cas de base. Puis, en regardant les dernières lettres x_{i-1} et y_{j-1} des deux préfixes, déterminer et démontrer une récurrence sur $L(i, j)$.

Question : la récurrence de la PLSC (correction)

cas $x_{i-1} = y_{j-1}$

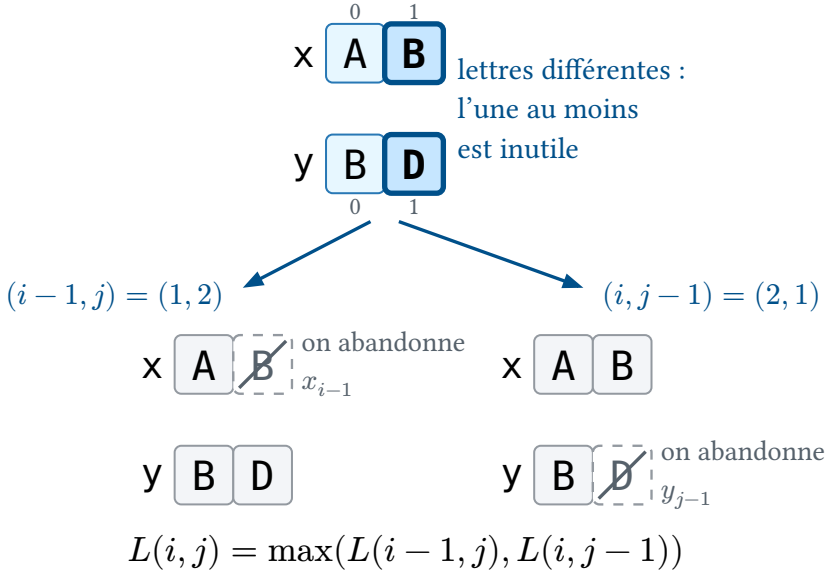
exemple : $i = 3, j = 3$, lettres C et C



$L(i, j) = 1 + L(i-1, j-1)$

cas $x_{i-1} \neq y_{j-1}$

exemple : $i = 2, j = 2$, lettres B et D



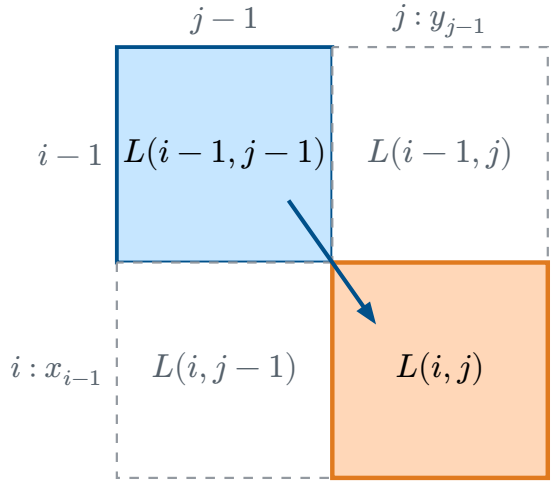
$L(i, j) = \max(L(i-1, j), L(i, j-1))$

$$L(i, 0) = L(0, j) = 0, \quad L(i, j) = \begin{cases} 1 + L(i-1, j-1) & \text{si } x_{i-1} = y_{j-1}, \\ \max(L(i-1, j), L(i, j-1)) & \text{sinon.} \end{cases}$$

Calcul : le tableau

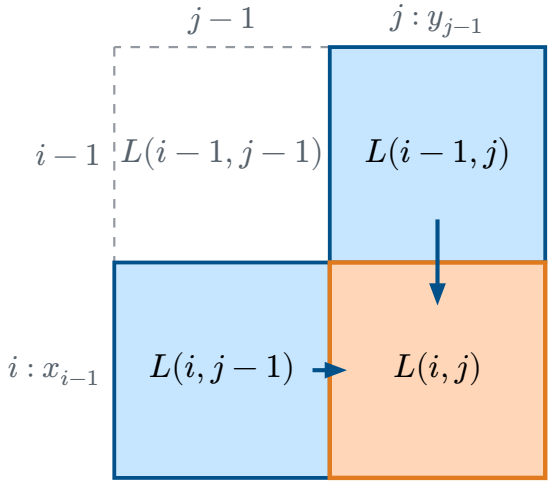
i \ j		0	1	2	3	4	5	6
			B	**D**	**C**	**A**	**B**	**A**
0		0	0	0	0	0	0	0
1	**A**	0						
2	**B**	0						
3	**C**	0						
4	**B**	0						
5	**D**	0						
6	**A**	0						
7	**B**	0						

si $x_{i-1} = y_{j-1}$



$L(i,j) = 1 + L(i-1,j-1)$

si $x_{i-1} \neq y_{j-1}$



$L(i,j) = \max(L(i-1,j), L(i,j-1))$

- $(n+1) \times (m+1)$ cases, ligne 0 et colonne 0 : cas de base ; (i,j) peut dépendre du dessus, de la gauche, de la diagonale : **ligne par ligne**, de gauche à droite

Question : `table_plsc`

Question (poly § 6.3) Écrire `table_plsc(x, y)` qui renvoie le tableau L, puis `plsc(x, y)` qui renvoie la longueur $L(n, m)$. Quelle est sa complexité ?

Question : `table_plsc` (correction)

```python
def table_plsc(x, y):
    n, m = len(x), len(y)
    L = [[0 for j in range(m + 1)] for i in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if x[i - 1] == y[j - 1]:
                L[i][j] = 1 + L[i - 1][j - 1]
            elif L[i - 1][j] >= L[i][j - 1]:
                L[i][j] = L[i - 1][j]
            else:
                L[i][j] = L[i][j - 1]
    return L

def plsc(x, y):
    return table_plsc(x, y)[len(x)][len(y)]
```

Le tableau, ligne par ligne

i \ j		0	1 **B**	2 **D**	3 **C**	4 **A**	5 **B**	6 **A**
0		0	0	0	0	0	0	0
1	**A**	0						
2	**B**	0						
3	**C**	0						
4	**B**	0						
5	**D**	0						
6	**A**	0						
7	**B**	0						

ligne 0 : préfixe vide

x A B C B D A B

y **B** **D** **C** **A** **B** **A**

Le tableau, ligne par ligne

i \ j		0	1 **B**	2 **D**	3 **C**	4 **A**	5 **B**	6 **A**
0		0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1	1
2	**B**	0						
3	**C**	0						
4	**B**	0						
5	**D**	0						
6	**A**	0						
7	**B**	0						

ligne 1 : on ajoute **A**

x **A** B C B D A B

y **B** **D** **C** **A** **B** **A**

encadrées : les lettres égales (diagonale)

Le tableau, ligne par ligne

i \ j		0	1 **B**	2 **D**	3 **C**	4 **A**	5 **B**	6 **A**
0		0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1	1
2	**B**	0	1	1	1	1	2	2
3	**C**	0						
4	**B**	0						
5	**D**	0						
6	**A**	0						
7	**B**	0						

ligne 2 : on ajoute **B**

x **A** **B** C B D A B

y **B** **D** **C** **A** **B** **A**

encadrées : les lettres égales (diagonale)

Le tableau, ligne par ligne

i \ j		0	1 **B**	2 **D**	3 **C**	4 **A**	5 **B**	6 **A**
0		0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1	1
2	**B**	0	1	1	1	1	2	2
3	**C**	0	1	1	2	2	2	2
4	**B**	0						
5	**D**	0						
6	**A**	0						
7	**B**	0						

ligne 3 : on ajoute **C**

x **A** **B** **C** B D A B

y **B** **D** **C** **A** **B** **A**

encadrées : les lettres égales (diagonale)

▷ Démo 6

Le tableau, ligne par ligne

i \ j		0	1 **B**	2 **D**	3 **C**	4 **A**	5 **B**	6 **A**
0		0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1	1
2	**B**	0	1	1	1	1	2	2
3	**C**	0	1	1	2	2	2	2
4	**B**	0	1	1	2	2	3	3
5	**D**	0						
6	**A**	0						
7	**B**	0						

ligne 4 : on ajoute **B**

x **A** **B** **C** **B** D A B

y **B** **D** **C** **A** **B** **A**

encadrées : les lettres égales (diagonale)

Le tableau, ligne par ligne

i \ j		0	1 **B**	2 **D**	3 **C**	4 **A**	5 **B**	6 **A**
0		0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1	1
2	**B**	0	1	1	1	1	2	2
3	**C**	0	1	1	2	2	2	2
4	**B**	0	1	1	2	2	3	3
5	**D**	0	1	2	2	2	3	3
6	**A**	0						
7	**B**	0						

ligne 5 : on ajoute **D**

x **A** **B** **C** **B** **D** A B

y **B** **D** **C** **A** **B** **A**

encadrées : les lettres égales (diagonale)

▷ Démo 6

Le tableau, ligne par ligne

i \ j		0	1 **B**	2 **D**	3 **C**	4 **A**	5 **B**	6 **A**
0		0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1	1
2	**B**	0	1	1	1	1	2	2
3	**C**	0	1	1	2	2	2	2
4	**B**	0	1	1	2	2	3	3
5	**D**	0	1	2	2	2	3	3
6	**A**	0	1	2	2	3	3	4
7	**B**	0						

ligne 6 : on ajoute **A**

x **A** **B** **C** **B** **D** **A** B

y **B** **D** **C** **A** **B** **A**

encadrées : les lettres égales (diagonale)

Le tableau, ligne par ligne

i \ j		0	1 **B**	2 **D**	3 **C**	4 **A**	5 **B**	6 **A**
0		0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1	1
2	**B**	0	1	1	1	1	2	2
3	**C**	0	1	1	2	2	2	2
4	**B**	0	1	1	2	2	3	3
5	**D**	0	1	2	2	2	3	3
6	**A**	0	1	2	2	3	3	4
7	**B**	0	1	2	2	3	4	4

ligne 7 : on ajoute **B**

x **A** **B** **C** **B** **D** **A** **B**

y **B** **D** **C** **A** **B** **A**

encadrées : les lettres égales (diagonale)

Question : reconstruction

Question (poly § 6.4) Écrire `une_plsc(x, y)` qui renvoie une plus longue sous-suite commune, en remontant dans le tableau depuis la case (n, m). Complexité de la remontée ?

Question : reconstruction (correction)

```
def une_plsc(x, y):
    n, m = len(x), len(y)
    L = table_plsc(x, y)                    # le tableau rempli
    i, j = n, m
    commune = []
    while i > 0 and j > 0:
        if x[i - 1] == y[j - 1]:            # caractère commun : on le garde
            commune.append(x[i - 1])
            i, j = i - 1, j - 1
        elif L[i - 1][j] >= L[i][j - 1]:    # on refait le choix du maximum
            i = i - 1
        else:
            j = j - 1
    return commune[::-1]                    # construite à l'envers
```

Question : reconstruction (correction)

```
def une_plsc(x, y):
    n, m = len(x), len(y)
    L = table_plsc(x, y)                    # le tableau rempli
    i, j = n, m
    commune = []
    while i > 0 and j > 0:
        if x[i - 1] == y[j - 1]:            # caractère commun : on le garde
            commune.append(x[i - 1])
            i, j = i - 1, j - 1
        elif L[i - 1][j] >= L[i][j - 1]:    # on refait le choix du maximum
            i = i - 1
        else:
            j = j - 1
    return commune[::-1]                    # construite à l'envers
```

- terminaison et complexité (**variant**): $i + j$ est strictement décroissant, $O(n + m)$, et $O(nm)$ en mémoire pour le tableau.

La remontée depuis $(7, 6)$

i \ j		0	1 **B**	2 **D**	3 **C**	4 **A**	5 **B**	6 **A**
0		0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1	1
2	**B**	0	1	1	1	1	2	2
3	**C**	0	1	1	2	2	2	2
4	**B**	0	1	1	2	2	3	3
5	**D**	0	1	2	2	2	3	3
6	**A**	0	1	2	2	3	3	4
7	**B**	0	1	2	2	3	4	4

case $(7, 6)$: préfixes en bleu

x **A** **B** **C** **B** **D** **A** **B**

y **B** **D** **C** **A** **B** **A**

gardées : **—**

La remontée depuis $(7, 6)$

i \ j		0	1 B	2 D	3 C	4 A	5 B	6 A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

case $(6, 6)$: préfixes en bleu

x **A** **B** **C** **B** **D** **A** B

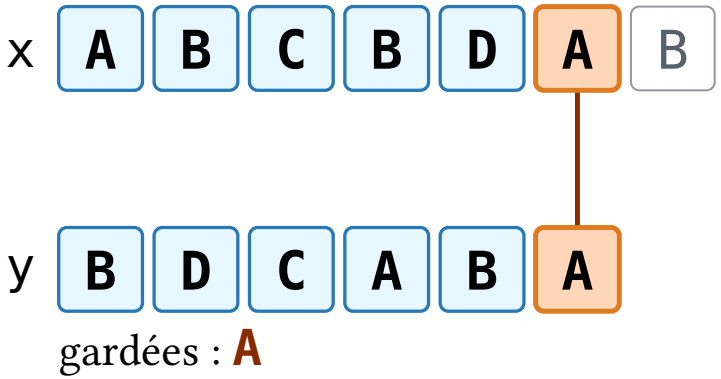
y **B** **D** **C** **A** **B** **A**

gardées : **—**

La remontée depuis $(7, 6)$

i \ j		0	1 B	2 D	3 C	4 A	5 B	6 A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

case $(5, 5)$: préfixes en bleu

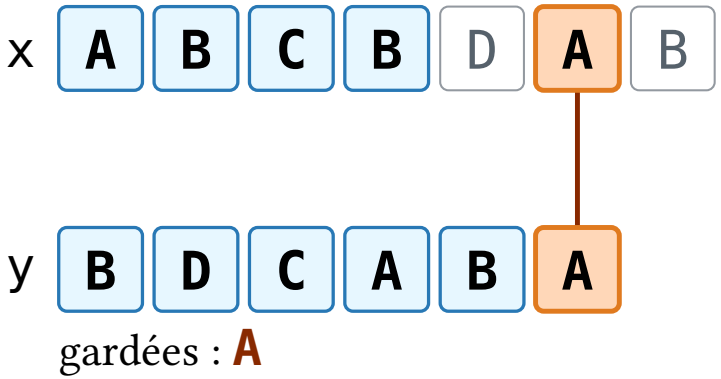


gardées : **A**

La remontée depuis $(7, 6)$

i \ j		0	1 B	2 D	3 C	4 A	5 B	6 A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

case $(4, 5)$: préfixes en bleu

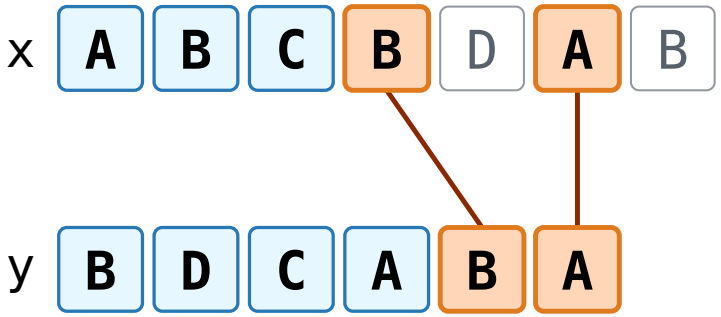


gardées : **A**

La remontée depuis $(7, 6)$

i \ j		0	1 **B**	2 **D**	3 **C**	4 **A**	5 **B**	6 **A**
0		0	0	0	0	0	0	0
1	**A**	0	0	0	0	1	1	1
2	**B**	0	1	1	1	1	2	2
3	**C**	0	1	1	2	2	2	2
4	**B**	0	1	1	2	2	3	3
5	**D**	0	1	2	2	2	3	3
6	**A**	0	1	2	2	3	3	4
7	**B**	0	1	2	2	3	4	4

case $(3, 4)$: préfixes en bleu



gardées : **BA**

La remontée depuis $(7, 6)$

i \ j		0	1 B	2 D	3 C	4 A	5 B	6 A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	**2**	2	2	2
4	B	0	1	1	2	2	**3**	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	**4**
7	B	0	1	2	2	3	4	4

case $(3, 3)$: préfixes en bleu

x **A** **B** **C** **B** D **A** B

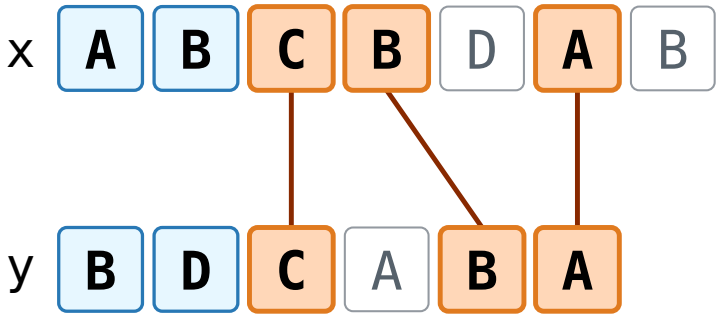
y **B** **D** **C** A **B** **A**

gardées : **BA**

La remontée depuis $(7, 6)$

i \ j	0	1 B	2 D	3 C	4 A	5 B	6 A
0	0	0	0	0	0	0	0
1 A	0	0	0	0	1	1	1
2 B	0	1	1	1	1	2	2
3 C	0	1	1	2	2	2	2
4 B	0	1	1	2	2	3	3
5 D	0	1	2	2	2	3	3
6 A	0	1	2	2	3	3	4
7 B	0	1	2	2	3	4	4

case $(2, 2)$: préfixes en bleu

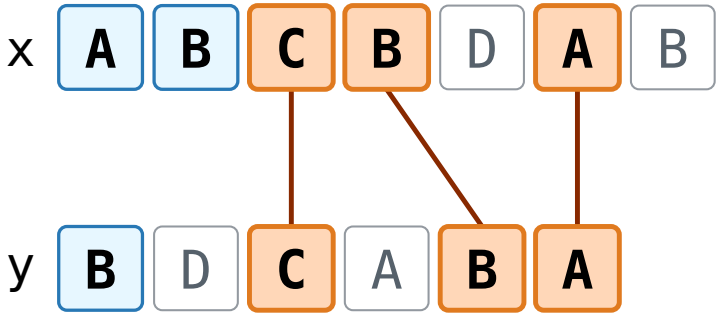


gardées : **CBA**

La remontée depuis $(7, 6)$

i \ j		0	1 B	2 D	3 C	4 A	5 B	6 A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

case $(2, 1)$: préfixes en bleu

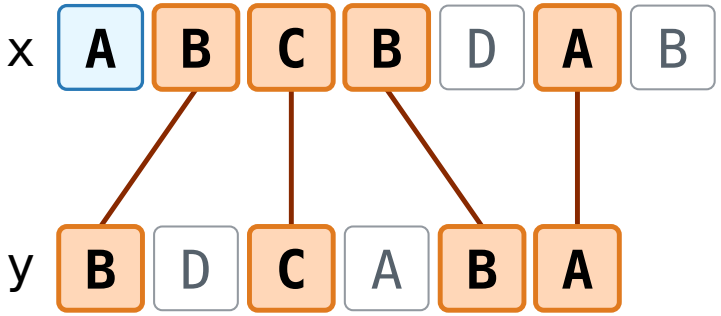


gardées : **CBA**

La remontée depuis $(7, 6)$

i \ j		0	1 B	2 D	3 C	4 A	5 B	6 A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

case $(1, 0)$: préfixes en bleu



gardées : **BCBA**

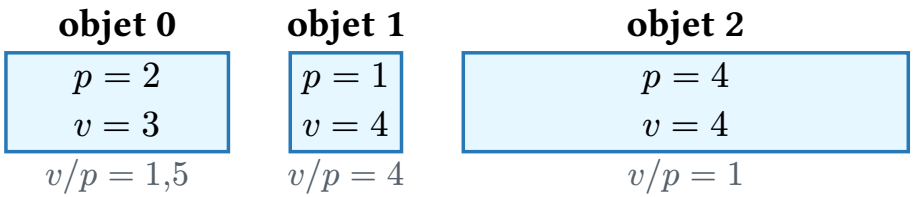
Un préfixe et une ressource : le sac à dos

Le problème

objet 0	objet 1	objet 2
$p = 2$	$p = 1$	$p = 4$
$v = 3$	$v = 4$	$v = 4$
$v/p = 1{,}5$	$v/p = 4$	$v/p = 1$

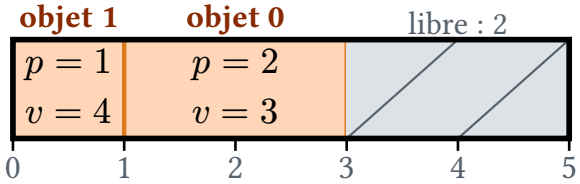
sac de capacité $b = 5$

Le problème

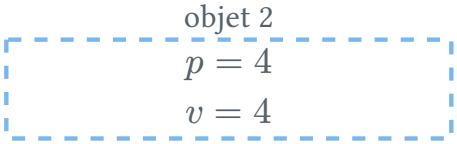


sac de capacité $b = 5$

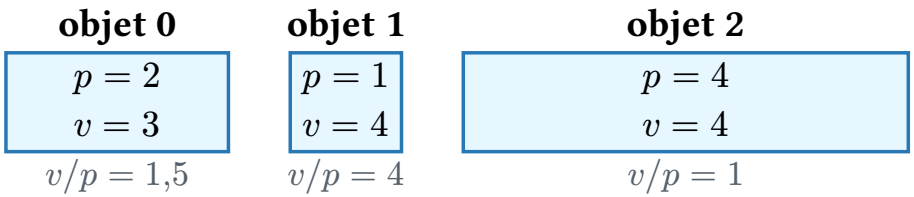
glouton (rapport v/p décroissant) : objets 1 puis 0, poids 3, prix **7**



laissé :

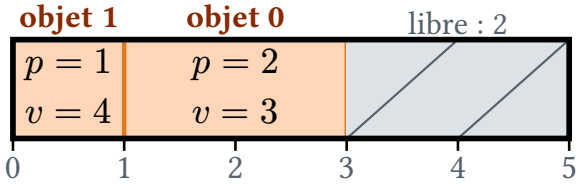


Le problème

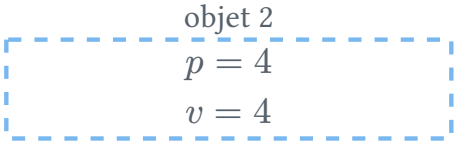


sac de capacité $b = 5$

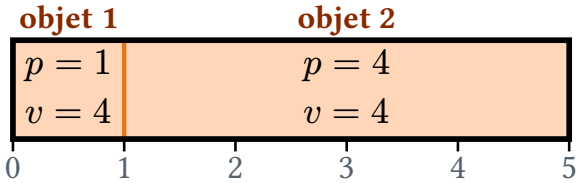
glouton (rapport v/p décroissant) : objets 1 puis 0, poids 3, prix **7**



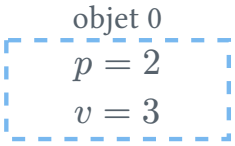
laissé :



optimum : objets 1 et 2, poids 5, prix **8**



laissé :



États et récurrence

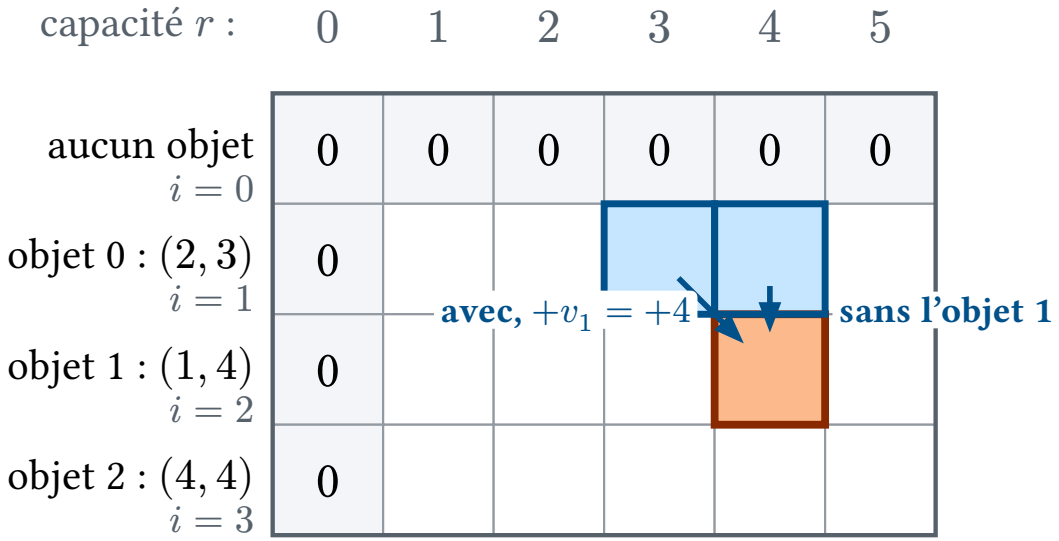
- le préfixe i ne suffit pas : pour décider de l'objet $i-1$, il faut la **capacité** qui reste pour les autres

L'état est le couple (i, r) : les objets autorisés (indices $< i$) et la capacité disponible.

Notation. $P(i, r)$ est le prix maximal réalisable avec les objets d'indices $< i$ et une capacité r. La réponse est $P(n, b)$.

Question (poly § 7.2) Donner et justifier la récurrence vérifiée par P.

Question : la récurrence du sac à dos (correction)



$$P(0, r) = 0, \quad P(i, r) = \begin{cases} P(i-1, r) & \text{si } p_{i-1} > r, \\ \max(P(i-1, r), v_{i-1} + P(i-1, r - p_{i-1})) & \text{sinon.} \end{cases}$$

- une solution optimale pour (i, r) prend ou ne prend pas l'objet $i - 1$

Question : remplir le tableau

Question (poly § 7.3) Remplir le tableau ci-dessous (ligne i : objets d'indices $< i$, colonne r : capacité), puis écrire `tableau_sac(objets, b)` qui le calcule et `sac(objets, b)` qui renvoie $P(n, b)$. Quelle est sa complexité ?

capacité r :	0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0					
objet 1 : $(1, 4)$ $i = 2$	0					
objet 2 : $(4, 4)$ $i = 3$	0					

▷ Démo 7

Question : remplir le tableau (correction)

capacité r :	0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0	0	3	3	3	3
objet 1 : $(1, 4)$ $i = 2$	0					
objet 2 : $(4, 4)$ $i = 3$	0					

▷ Démo 7

Question : remplir le tableau (correction)

capacité r :	0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0	0	3	3	3	3
objet 1 : $(1, 4)$ $i = 2$	0					
objet 2 : $(4, 4)$ $i = 3$	0					

▷ Démo 7

Question : remplir le tableau (correction)

capacité r :	0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0	0	3	3	3	3
objet 1 : $(1, 4)$ $i = 2$	0	4				
objet 2 : $(4, 4)$ $i = 3$	0					

▷ Démo 7

Question : remplir le tableau (correction)

capacité r :	0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0	0	3	3	3	3
objet 1 : $(1, 4)$ $i = 2$	0	4	4			
objet 2 : $(4, 4)$ $i = 3$	0					

▷ Démo 7

Question : remplir le tableau (correction)

capacité r :	0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0	0	3	3	3	3
objet 1 : $(1, 4)$ $i = 2$	0	4	4	7		
objet 2 : $(4, 4)$ $i = 3$	0					

▷ Démo 7

Question : remplir le tableau (correction)

capacité r :	0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0	0	3	3	3	3
objet 1 : $(1, 4)$ $i = 2$	0	4	4	7	7	
objet 2 : $(4, 4)$ $i = 3$	0					

▷ Démo 7

Question : remplir le tableau (correction)

capacité r :	0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0	0	3	3	3	3
objet 1 : $(1, 4)$ $i = 2$	0	4	4	7	7	7
objet 2 : $(4, 4)$ $i = 3$	0					

Question : remplir le tableau (correction)

capacité r :	0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0	0	3	3	3	3
objet 1 : $(1, 4)$ $i = 2$	0	4	4	7	7	7
objet 2 : $(4, 4)$ $i = 3$	0	4	4	7	7	8

▷ Démo 7

Question : remplir le tableau (correction)

capacité r :	0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0	0	3	3	3	3
objet 1 : $(1, 4)$ $i = 2$	0	4	4	7	7	7
objet 2 : $(4, 4)$ $i = 3$	0	4	4	7	7	8

- `P[n][b]` $= 8$: le prix maximal

Question : tableau_sac (correction)

```
def tableau_sac(objets, b):
    n = len(objets)
    P = [[0 for r in range(b + 1)] for i in range(n + 1)]
    for i in range(1, n + 1):
        poids, prix = objets[i - 1]
        for r in range(b + 1):
            P[i][r] = P[i - 1][r]                          # sans l'objet i - 1
            if poids <= r and prix + P[i - 1][r - poids] > P[i][r]:
                P[i][r] = prix + P[i - 1][r - poids]     # avec
    return P
```

```
def sac(objets, b):
    return tableau_sac(objets, b)[len(objets)][b]
```

Question : reconstruction

Question (poly § 7.4) Écrire `sac_objets(objets, b)` qui renvoie la liste des indices des objets d'une solution optimale, à partir du tableau `P`, sans tableau de choix supplémentaire.

Question : reconstruction (correction)

```
def sac_objets(objets, b):
    n = len(objets)
    P = tableau_sac(objets, b)
    choisis = []
    r = b
    i = n
    while i > 0:                          # de la dernière ligne vers la première
        if P[i][r] != P[i - 1][r]:        # l'objet i - 1 a été pris
            choisis.append(i - 1)
            r = r - objets[i - 1][0]
        i = i - 1
    return choisis[::-1]                  # dans l'ordre des indices
```

Question : reconstruction (correction)

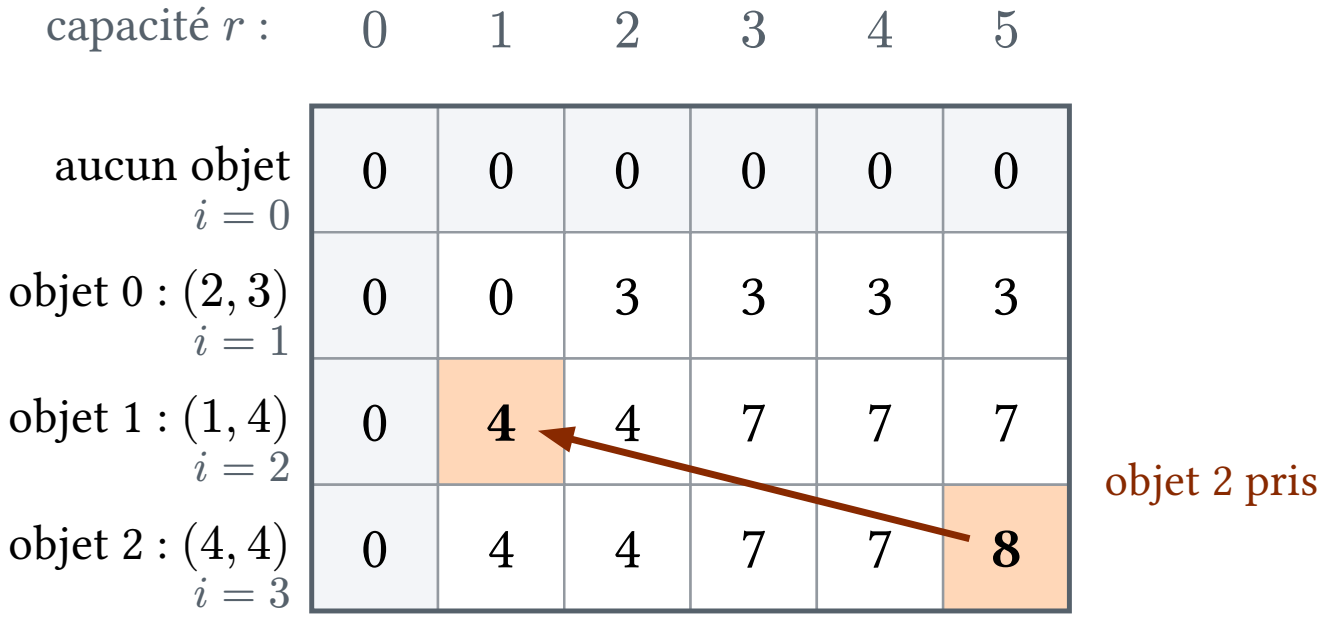
```
def sac_objets(objets, b):
    n = len(objets)
    P = tableau_sac(objets, b)
    choisis = []
    r = b
    i = n
    while i > 0:                          # de la dernière ligne vers la première
        if P[i][r] != P[i - 1][r]:        # l'objet i - 1 a été pris
            choisis.append(i - 1)
            r = r - objets[i - 1][0]
        i = i - 1
    return choisis[::-1]                  # dans l'ordre des indices
```

si `P[i][r] == P[i - 1][r]` alors on ne prend pas l'objet $i - 1$

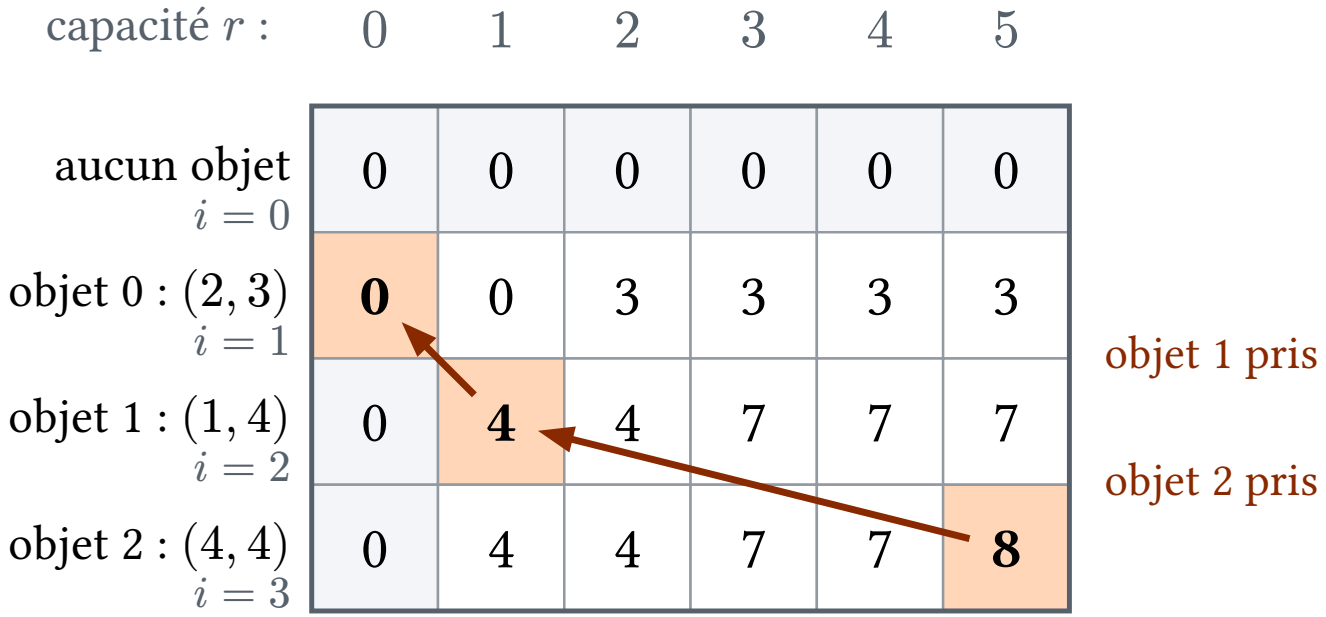
La remontée depuis $(3, 5)$

	capacité r : 0	1	2	3	4	5
aucun objet $i = 0$	0	0	0	0	0	0
objet 0 : $(2, 3)$ $i = 1$	0	0	3	3	3	3
objet 1 : $(1, 4)$ $i = 2$	0	4	4	7	7	7
objet 2 : $(4, 4)$ $i = 3$	0	4	4	7	7	**8**

La remontée depuis $(3, 5)$



La remontée depuis $(3, 5)$



La remontée depuis $(3, 5)$

