Informatique tronc commun

Programmation dynamique

Poly de cours à compléter

1 Introduction 2
1.1 Le programme 2
1.2 Aux concours 3
2 Rappels : les algorithmes gloutons 5
2.1 L'idée : le meilleur choix local 5
2.2 … qui ne donne que rarement une solution optimale 6
2.3 Un algorithme glouton comme outil de preuve d'existence 7
2.4 Quand un algorithme glouton est optimal : rendre gloutonne une solution optimale 9
3 Un premier exemple : le rendu de monnaie 11
3.1 Le problème 11
3.2 État, sous-problèmes et récurrence 11
3.3 Récursion naïve : l'arbre des appels 11
3.4 De haut en bas : mémoïsation 12
3.5 De bas en haut : le tableau 13
3.6 Reconstruire une solution optimale 14
4 Les ingrédients de la programmation dynamique 15
4.1 À propos des sous-problèmes 15
4.2 De haut en bas ou de bas en haut ? 16
4.3 Calculer une complexité 17
4.4 Enjeux de mémoire 18
5 La plus longue sous-suite croissante (LIS) : choisir ses états 18
5.1 Le problème 18
5.2 Solution très naïve 19
5.3 Algorithme glouton ? 20
5.4 Choisir les états : sous-suites se terminant en i 20
6 Deux préfixes : la plus longue sous-suite commune 22
6.1 Le problème 22
6.2 États et récurrence 24
6.3 Calcul : le tableau et sa complexité 25
6.4 Reconstruction 26
7 Un préfixe et une ressource : le sac à dos 27
7.1 Le problème 27
7.2 États et récurrence 27
7.3 Calcul : le tableau et sa complexité 28
7.4 Reconstruction 29

1 Introduction

1.1 Le programme

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.*

extrait programme ITC, 1re 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.

extrait programme ITC, 2e année

Il y a plusieurs idées générales pour concevoir un algorithme qui détermine une configuration optimale d'un problème :

Méthode	**Idée générale**	**Avantages**	**Inconvénients**	**Exemples**
Solution naïve	Tester exhaustivement toutes les configurations possibles (au besoin par backtracking, en abandonnant une branche sans issue).	Souvent rapide à écrire, donne la meilleure solution.	Une *complexité catastrophique*, souvent exponentielle.	tous les problèmes s'y prêtent
Diviser pour régner	Découper le problème en plusieurs sous-problèmes **indépendants**, les résoudre récursivement et recombiner.	Souvent très efficace, rien à mémoriser, la récursion suffit.	Il faut que le problème se prête à une décomposition « sans interactions ».	tri fusion, dichotomie, exponentiation rapide
Algorithme glouton	Choisir à chaque étape la meilleure solution « à court terme », sans jamais revenir en arrière.	Rapide à écrire, bonne complexité.	Donne rarement une solution optimale : il faut le prouver au cas par cas.	sélection d'activités, Dijkstra

Méthode	Idée générale	Avantages	Inconvénients	Exemples
Programmation dynamique	Essayer tous les choix, mais ne résoudre chaque sous-problème **qu'une fois** grâce à un bon choix d'états.	Donne la meilleure solution, complexité raisonnable quand les états sont peu nombreux.	Il faut trouver les états et la récurrence, mémoire du tableau.	ce chapitre : rendu de monnaie, sous-suites, sac à dos

La programmation dynamique emprunte à **diviser pour régner** l'idée de se ramener à des sous-problèmes, mais ses sous-problèmes se **chevauchent** (le même revient dans plusieurs branches) : elle leur applique donc la **mémoïsation**, ou remplit un tableau. Elle emprunte à la recherche exhaustive l'idée d'**essayer tous les choix**, sans en payer le prix exponentiel.

Programmation dynamique. Résoudre un problème en le ramenant à une famille de **sous-problèmes** dont les solutions se déduisent les unes des autres par une **relation de récurrence**, chaque sous-problème n'étant résolu **qu'une seule fois** (par mémoïsation c'est-à-dire « de haut en bas », ou en remplissant un tableau c'est-à-dire « de bas en haut »).

Concevoir un algorithme de programmation dynamique revient donc à : **résumer, décomposer, mémoriser**.

1. **Résumer** : choisir un **état**, c'est-à-dire le peu d'information qui décrit un sous-problème (un entier k, un indice i, un couple (i, j)…). Les états doivent être assez riches pour que l'étape suivante soit possible, et assez peu nombreux pour que l'algorithme soit rapide.
2. **Décomposer** : écrire une récurrence qui exprime la solution d'un état à partir de celles d'états « plus petits » (l'*équation de Bellman*), et la justifier par la **sous-structure optimale**.
3. **Mémoriser** : calculer chaque état une seule fois, puis, si nécessaire, **reconstruire** une solution à partir des valeurs calculées.

> (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!**
>
> extrait J. Erickson, *Algorithms*, 2019, p. 106 (chapitre 3, en ligne)

Autrement dit : la programmation dynamique, ce n'est pas remplir des tableaux, c'est de la **récursion intelligente**.

1.2 Aux concours

La programmation dynamique est présente dans presque tous les sujets de 2023 à 2026 des deux épreuves communes (Mines-Ponts MP-PC-PSI et X-ENS Informatique B). La construction des sujets est souvent la même :

- un algorithme glouton, qui n'est pas optimal (ou un algorithme de recherche exhaustive, trop coûteux)
- une récurrence, donnée par l'énoncé ou à justifier « brièvement »
- son implémentation (mémoïsation et/ou tableau)
- le calcul de sa complexité
- la reconstruction d'une solution.

Les six sujets de 2023 à 2025 suivent ce chemin, en tout ou en partie. En 2026, Mines ne contient pas de programmation dynamique et X-ENS n'en garde que la mémoïsation d'une récurrence sur un graphe sans cycle :

Sujet	Problème	État	Ce qui est demandé
Mines 2023 Q21–26	justifier un paragraphe	i : premier mot d'une ligne	glouton non optimal, mémoïser une fonction récursive donnée, complexités, reconstruire les lignes
Mines 2024 Q18–25	décodage de Viterbi	(symbole i, position j)	glouton non optimal, remplir un tableau de bas en haut, lire le tableau des prédécesseurs, complexités
Mines 2025 Q7–14	sac à dos	(objets $< k$, capacité r)	glouton non optimal, dérouler le tableau à la main, complexité, reconstruire la solution
X-ENS 2023 Q9–11	distance d'édition	(préfixe i, préfixe j)	justifier la récurrence, tableau de bas en haut, reconstruction en $O(n+m)$
X-ENS 2024 Q10–12	logimages	(case c, bloc b)	tableau de bas en haut en $O(n_c^2)$ (n_c colonnes), cas de bord, reconstruction
X-ENS 2025 Q6–7, 11–15	chemin de pénalité minimale	(case, bonus activés)	récursif naïf, mémoïsation (quelle clé ?), récurrence, ordre de calcul, compléter le code, reconstruction
Mines 2026	—	—	pas de programmation dynamique (jeu à deux joueurs, minimax)
X-ENS 2026 Q10–12	forme normale et nombre de chemins dans un graphe sans cycle	sommet s : $\mathrm{fn}(s)$, $T(s)$	montrer que le récursif naïf est exponentiel, mémoïser (clé : le sommet) en temps linéaire, trouver la récurrence de T

En pratique. Sur les six sujets de 2023 à 2025 :

- la **phrase qui définit l'état** (une case du tableau) est **toujours donnée**, on ne demande pas d'inventer les sous-problèmes du tableau (seule la clé de mémoïsation de X-ENS 2025, Q7, est à trouver : il fallait y mettre les bonus activés), ni de prouver la sous-structure optimale,
- la **récurrence** est donnée telle quelle (les trois Mines), donnée en prose et à formaliser (X-ENS 2024, 2025) ou à trouver (X-ENS 2023, sur un exemple du programme),
- le plus souvent la programmation dynamique suit une construction **de bas en haut** avec seulement deux exemples de mémoïsation,
- le calcul de la **complexité** (six fois) et l'étape de **reconstruction** d'une solution (six fois, quatre fois à coder) sont systématiques,
- quand un **algorithme glouton** précède (quatre fois) il faut montrer qu'il est non optimal, en vérifiant un contre-exemple donné (ou en l'exhibant).

Ce chapitre donne la vision d'ensemble : choisir les états, justifier une récurrence, la calculer, reconstruire, etc. Même si au concours, l'essentiel du travail consiste à **lire** la définition fournie, la **vérifier** sur l'exemple, puis **coder** sans erreur le remplissage, la complexité et la reconstruction, il est important de bien comprendre la logique générale sous-jacente.

Dans tout le chapitre, pour simplifier, `INFINI` désigne une valeur plus grande que toutes les autres, comme la variable globale `INFINI` du sujet X-ENS 2025 ou `float("inf")` de Mines 2023 :

```
INFINI = float("inf")     # +∞ : plus grand que tout entier, INFINI + 1 == INFINI
```

2 Rappels : les algorithmes gloutons

2.1 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**, et sans **jamais revenir** sur un choix effectué.

Le schéma est souvent le même : on ordonne les candidats selon un critère (la plus grande pièce, la tâche qui finit le plus tôt, l'objet de meilleur rapport prix/poids…), puis on les parcourt en retenant chacun s'il est compatible avec les choix déjà faits.

Le rendu de monnaie. On dispose de pièces dont les valeurs, entières et strictement positives, sont données par une liste `pieces` (d'ensemble noté $\mathcal{P}$) triée par ordre croissant et contenant 1 (par exemple $1, 3, 4$ ou $1, 2, 5, 10, 20, 50$), chacune en quantité illimitée, et l'on veut rendre une somme entière n avec **le moins de pièces possible**. La recherche exhaustive essaie toutes les suites de pièces de somme n (l'arbre des choix de la partie 3) : leur nombre croît exponentiellement avec n. Un algorithme glouton de première année rend à chaque étape la plus grande pièce possible.

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

Le sac à dos. On dispose de n objets, l'objet i ayant un poids p_i et un prix v_i (entiers strictement positifs), et d'un sac de capacité b. Il faut choisir des objets de poids total $\leq b$ et de prix total maximal, chaque objet au plus une fois. La recherche exhaustive examine les 2^n sous-ensembles, chacun en $O(n)$ pour calculer son poids et son prix : on a donc une complexité $O(n2^n)$, qui est inutilisable en pratique. Un algorithme glouton naturel trie les objets par rapport prix/poids décroissant et prend chaque objet s'il rentre encore.

Question. 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.

Deux qualités font le succès des algorithmes gloutons :

- ils sont **faciles à écrire** : une boucle, un test, pas de récursion ni de tableau,
- ils sont **rapides** : typiquement un tri en $O(n\ln(n))$ suivi d'un parcours en $O(n)$, là où une recherche exhaustive coûte de l'ordre de 2^n ou $n!$.

2.2 ... qui ne donne que rarement une solution optimale

Néanmoins les algorithmes gloutons ont un défaut majeur : un choix localement excellent peut interdire la meilleure solution globale.

Question. 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 ?

De plus, concevoir un algorithme glouton demande de choisir **ce qu'est** le meilleur choix local, et plusieurs critères sont souvent candidats.

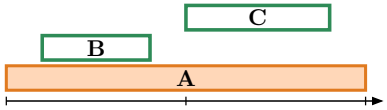
Prenons par exemple la **sélection d'activités** de première année : on dispose d'une seule salle et de n activités, l'activité i occupant l'intervalle de temps $[d_i, f_i[$; on veut en accueillir le plus grand nombre possible, deux activités retenues ne devant pas se chevaucher. Trois critères gloutons viennent à l'esprit :

- prendre l'activité qui commence le plus tôt,
- prendre la plus courte,
- prendre celle qui finit le plus tôt,

puis continuer parmi les activités compatibles avec celles déjà retenues.

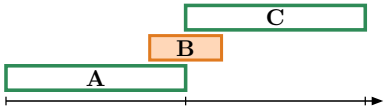
Question. Pour chacun des deux premiers critères, donner un contre-exemple à trois tâches.

début le plus tôt



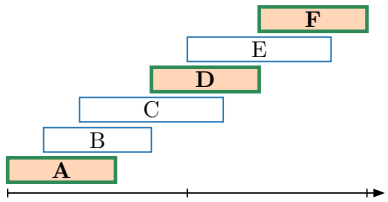
glouton : 1 tâche, optimum : 2

la plus courte



glouton : 1 tâche, optimum : 2

fin la plus tôt



glouton : 3 tâches, optimal

En orange les tâches retenues par un algorithme glouton, en contour vert une solution optimale.

2.3 Un algorithme glouton comme outil de preuve d'existence

Même quand il n'est pas optimal, un algorithme glouton qui **termine** prouve qu'une solution **existe** : c'est un argument constructif fréquent en mathématiques.

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

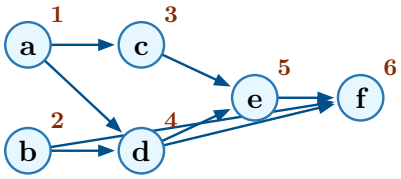
$$\frac{p}{q} = \sum_{k=0}^{m-1} \frac{1}{a_k}$$

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 une fonction `fractions_egyptiennes(p, q)` qui renvoie la liste des dénominateurs.

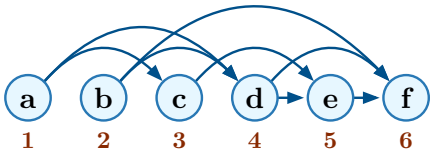
Question. **Théorème de Zeckendorf.** Montrer que tout entier $n \geq 1$ est somme de nombres de Fibonacci distincts et **non consécutifs** ($F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, …). Écrire une fonction `zeckendorf(n)` qui renvoie la liste des termes.

Question. **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



2.4 Quand un algorithme glouton est optimal : rendre gloutonne une solution optimale

Pour la majorité des problèmes on démontre que l'algorithme glouton n'est **pas** optimal, en **exhibant un contre-exemple**.

Pour certains problèmes, on peut **démontrer** qu'un algorithme glouton est optimal. L'argument est presque toujours le même, dit **d'échange** : partir d'une solution optimale quelconque et la transformer, sans la dégrader, en la solution gloutonne. Il suffit de montrer que le **premier** choix glouton est « sûr », c'est-à-dire qu'il existe une solution optimale qui commence par lui, on recommence ensuite sur le problème restant, qui est de même nature.

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

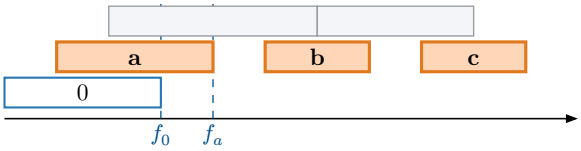
1. Supposer qu'il existe une solution optimale O **différente** de la solution gloutonne G.
2. Repérer la **première différence** : le premier choix de G qui n'est pas dans O.
3. Montrer qu'on peut **échanger** dans O ce choix contre le choix glouton **sans dégrader** O (l'échange ne l'améliore pas forcément, il suffit qu'il ne la détériore pas).
4. Conclure par **récurrence** : une solution optimale contient toute la solution gloutonne.

Le point 3 est le seul qui dépend du problème : c'est lui qu'il faut rédiger soigneusement.

Reprenons par exemple le problème de la sélection d'activités.

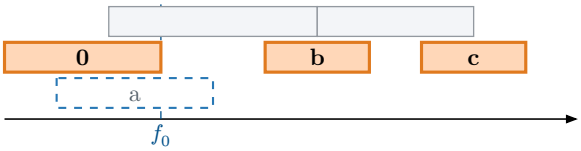
Question. Les activités sont triées par fin croissante, $f_0 \leq f_1 \leq \ldots \leq f_{n-1}$. Montrer qu'un algorithme glouton « fin la plus tôt » retient un nombre maximal d'activités compatibles.

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

L'échange de a contre la tâche 0 ; en gris, des tâches hors de O.

```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
```

Complexité : $O(n)$ après le tri, soit $O(n \ln(n))$ en tout, contre 2^n sous-ensembles à examiner naïvement.

Autres applications du patron : le sac à dos fractionnaire, et le guichet, où l'échange porte sur deux **voisins** (fascicule de compléments).

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.

3 Un premier exemple : le rendu de monnaie

3.1 Le problème

Illustrons l'idée de la programmation dynamique sur le problème de rendu de pièces de monnaie de la partie 2 : des pièces de valeurs données par une liste `pieces` (triée, contenant 1), notée $\mathcal{P}$, et une somme n à rendre avec le moins de pièces possible. Un algorithme glouton `rendu_glouton` de la partie 2 échoue pour certains systèmes de pièces : avec $\mathcal{P} = \{1, 3, 4\}$, il rend $6 = 4 + 1 + 1$ au lieu de $3 + 3$. C'est cet exemple qui servira de fil conducteur.

3.2 État, sous-problèmes et récurrence

L'état est la somme qui reste à rendre.

Un algorithme glouton échoue parce qu'il essaye de **deviner** la première pièce. La programmation dynamique, elle, **essaie toutes les premières pièces** et garde la meilleure. Pour cela il lui faut connaître le meilleur rendu des sommes plus petites : ce sont les sous-problèmes.

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

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

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

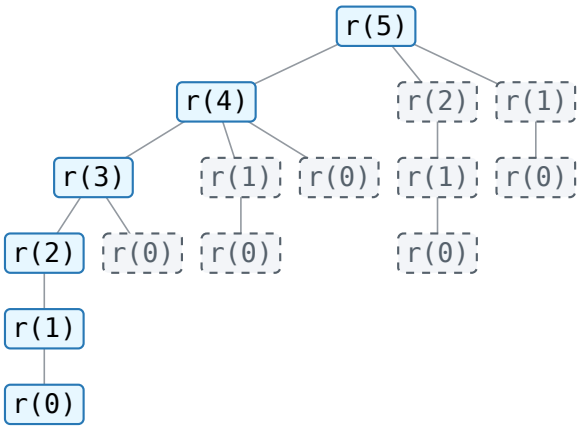
En pratique, si on souhaite la démontrer formellement, le schéma de preuve de l'équation de récurrence est souvent le même :

- « $\leq$ » parce que toute solution d'un sous-problème se complète en une solution du problème,
- « $\geq$ » parce qu'une solution optimale du problème, amputée de son premier choix, est une solution (donc au moins aussi coûteuse que l'optimum) d'un sous-problème.

3.3 Récursion naïve : l'arbre des appels

La récurrence se traduit mot à mot en une fonction récursive :

```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
```



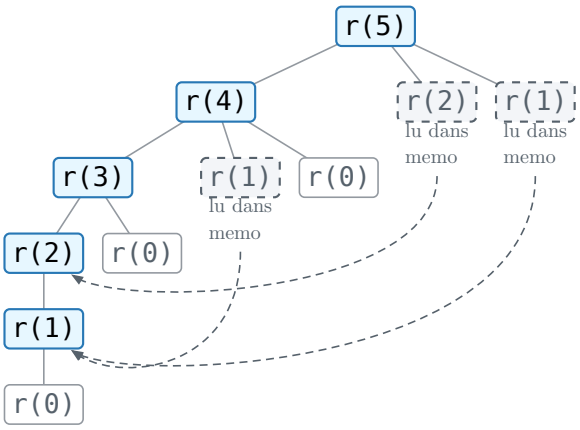
Arbre des appels de `rendu_naif(5, [1, 3, 4])` : 15 appels.

Question. 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 ?

3.4 De haut en bas : mémoïsation

C'est la situation du chapitre précédent : il suffit de retenir chaque $f(k)$ dans un dictionnaire.

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

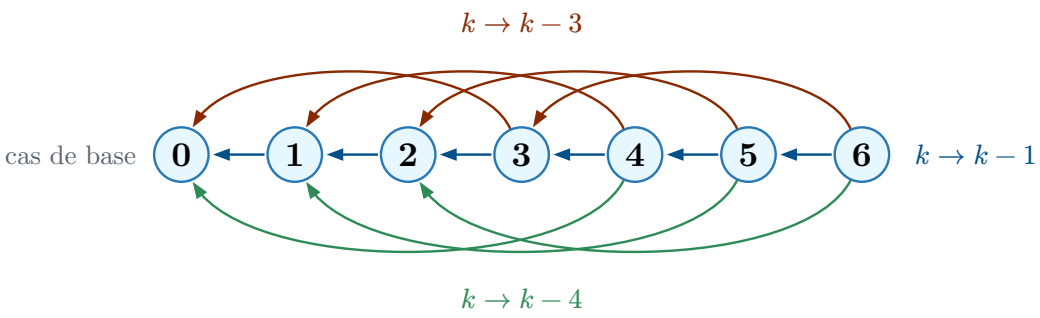


$n = 5$ appels développés, 3 lectures dans le dictionnaire, 3 cas de base : 11 appels au lieu de 15 pour la version naïve

Le même arbre avec `memo` : les appels en pointillé lisent le dictionnaire.

3.5 De bas en haut : le tableau

Plutôt que de laisser la récursion découvrir les sous-problèmes, on peut les calculer **tous**, dans un ordre tel que chaque valeur ne dépende que de valeurs déjà calculées. Pour trouver cet ordre, on dessine le **graphe des sous-problèmes** : une arête de k vers chaque $k - p$ dont $f(k)$ a besoin.



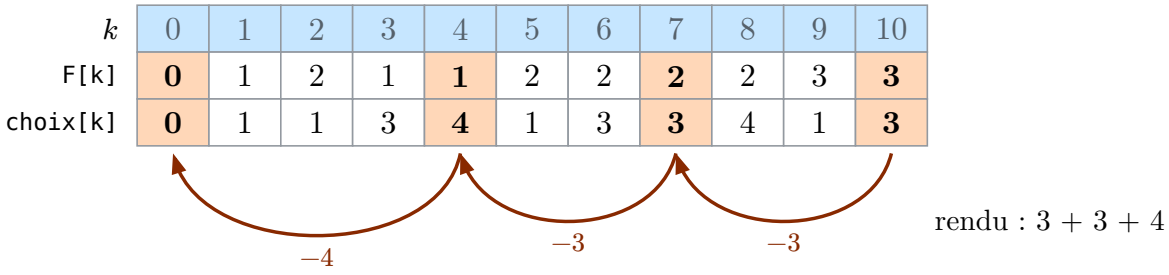
Le graphe des sous-problèmes pour $n = 6$.

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

3.6 Reconstruire une solution optimale

La fonction `rendu` donne le **nombre** de pièces, pas la composition d'une solution. Pour les retrouver, il suffit de retenir, pour chaque k, **un choix** ayant réalisé le minimum.

Question. 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.



`F`, `choix` et la reconstruction pour $n = 10$.

> On peut aussi se passer de `choix` : depuis k, chercher une pièce $p \leq k$ telle que `F[k - p] == F[k] - 1`, c'est-à-dire *refaire le choix* à partir des valeurs calculées (après avoir vérifié que `F[n]` n'est pas `INFINI`, car `INFINI - 1` vaut `INFINI`).
>
> Le coût de la reconstruction est alors celui du calcul d'un état, multiplié par la longueur du chemin.

Méthode. Les ingrédients de la programmation dynamique sont, dans l'ordre,

1. **États.** Définir les sous-problèmes et la valeur cherchée pour chacun par **une phrase en français** (« $f(k)$ est le nombre minimal de pièces pour rendre k »), et dire où se trouve la réponse ($f(n)$). Cette phrase est la *spécification* : sans elle, ni le correcteur ni vous ne pouvez vérifier la récurrence.
2. **Récurrence.** Cas de base, puis relation (équation de Bellman) exprimant f en un état à partir d'états plus petits, justifiée par la sous-structure optimale (pour un problème de décision ou de dénombrement : par une partition des solutions selon le premier choix).
3. **Calcul.** Mémoïsation ou tableau rempli dans un ordre compatible avec la récurrence.
4. **Complexité.** Le temps est le nombre d'états multiplié par le coût du calcul d'un état, hors appels récursifs ou lectures de cases déjà calculées (partie 4.3), la mémoire est le nombre d'états conservés.
5. **Reconstruction.** Mémoriser le choix gagnant de chaque état (ou le retrouver à partir des valeurs), puis remonter de l'état final vers un cas de base.

4 Les ingrédients de la programmation dynamique

4.1 À propos des sous-problèmes

Définition. Un problème d'optimisation a la propriété de **sous-structure optimale** lorsqu'une solution optimale est formée de solutions optimales de sous-problèmes. Autrement dit : si l'on retire à une solution optimale son premier choix, ce qui reste est optimal pour le sous-problème correspondant.

C'est elle qui autorise la récurrence : pour connaître le meilleur premier choix, il suffit de connaître la valeur optimale de chaque sous-problème, **et rien d'autre**. La preuve se fait par l'absurde : si le reste

n'était pas optimal, on le remplacerait par une meilleure solution du sous-problème et on obtiendrait une meilleure solution du problème de départ.

Attention. Tous les problèmes n'ont pas cette propriété.

- Le **plus long chemin simple** (sans sommet répété) dans un graphe : un plus long chemin de s à t passant par v n'est pas forcément formé d'un plus long chemin de s à v et d'un de v à t, car les deux morceaux pourraient partager un sommet.
- Le **sous-tableau de produit maximal** : dans $(-2, 3, -4)$, la meilleure tranche se terminant à l'indice 1 a pour produit 3, mais la meilleure tranche se terminant à l'indice 2, de produit $24 = (-2) \times 3 \times (-4)$, prolonge la tranche de produit **minimal** -6, et non la meilleure.

Le remède est toujours le même : **enrichir l'état**. Ici, le couple (produit maximal, produit minimal) suffit ; pour le chemin, il faudrait l'ensemble des sommets déjà visités, soit 2^n fois plus d'états.

> L'idée de la programmation dynamique ne sert pas qu'à optimiser. On l'utilise pour **dénombrer** (escaliers, chemins dans une grille, partitions d'un entier : chapitre et TP précédents), pour **décider** (peut-on découper ce texte en mots du dictionnaire ?) ou pour calculer une **position** (où placer ce bloc au plus tôt ? X-ENS 2024). La récurrence est alors une somme, un « ou » ou un cas par cas au lieu d'un minimum, et il n'y a pas de sous-structure **optimale** à invoquer, seulement une décomposition en 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 : typiquement polynomial contre exponentiel. Rien n'interdit qu'il soit lui-même exponentiel (plus long chemin simple ci-dessus) : la mémoïsation reste correcte, mais ne garantit plus un temps polynomial.

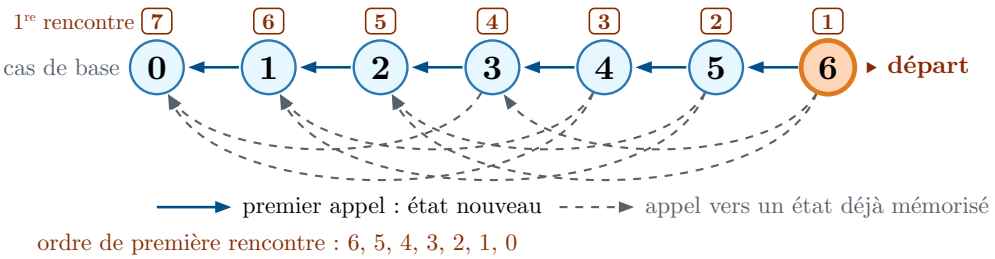
4.2 De haut en bas ou de bas en haut ?

Une fois la récurrence écrite, deux chemins mènent au résultat :

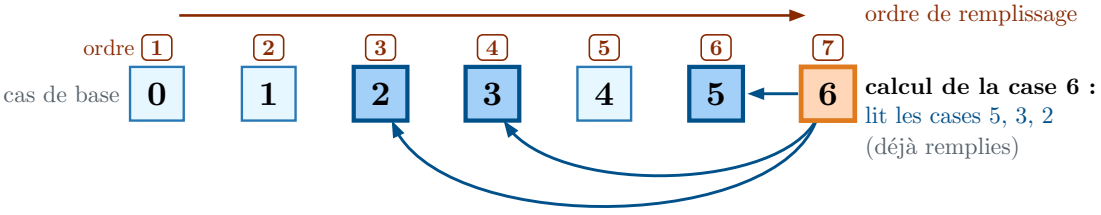
- **De haut en bas**, on part de la question posée ($f(n)$) et la récursion mémoïsée descend vers les cas de base en ne visitant que les états dont elle a besoin. On utilise le plus souvent dans ce cas une programmation récursive et un dictionnaire.
- **De bas en haut**, on part des cas de base et l'on remplit un tableau dans un ordre tel que chaque case ne dépende que de cases déjà remplies, jusqu'à atteindre $f(n)$. On utilise le plus souvent un tableau (ou une liste) et une programmation itérative.

La figure ci-dessous montre les deux parcours sur le graphe des sous-problèmes du rendu de monnaie.

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



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



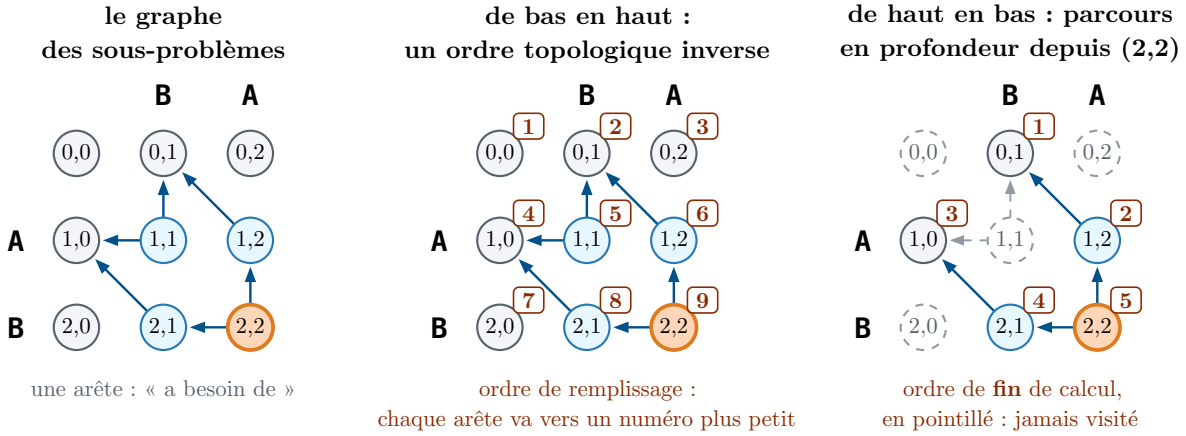
Les deux parcours pour $n = 6$ et $\mathcal{P} = \{1, 3, 4\}$.

	De haut en bas (mémoïsation)	**De bas en haut** (tableau)
écriture	se déduit directement de la récurrence : c'est la fonction naïve plus un dictionnaire	il faut trouver un **ordre de calcul** compatible avec les dépendances
états calculés	seulement ceux **atteints** depuis l'état initial	tous, y compris les inutiles
mémoire	dictionnaire + **pile d'appels** (profondeur = longueur du plus long chemin de dépendances : `RecursionError` possible)	tableau, dont on peut souvent ne garder que la dernière ligne
vitesse	accès au dictionnaire (hachage) et appels de fonction	boucles sur des listes : plus rapide d'un facteur constant
aux concours	Mines 2023 Q23 (« introduire une mémoïsation »), X-ENS 2025 Q7	Mines 2023 (fourni), 2024, 2025, X-ENS 2023, 2024, 2025 : toujours présent

Remarque théorique. Pour que chaque état se calcule à partir d'états déjà résolus (mémoïsation ou tableau), le graphe des sous-problèmes (une arête de chaque état vers les états dont il dépend) doit être **sans cycle**.

- Un calcul de bas en haut (remplissage de tableau) est un parcours de ce graphe dans l'ordre **inverse** d'un ordre topologique (partie 2.3) : on choisit cet ordre à la main, en regardant où vont les arêtes.
- Un calcul de haut en bas (mémoïsation) est un parcours en profondeur qui découvre cet ordre tout seul : un état est terminé après tous ceux dont il a besoin, et seuls les états atteints depuis la question posée sont visités.

Quand chaque état ne dépend que d'**un** sous-état à la fois (rendu de monnaie, sac à dos, distance d'édition), une optimisation revient souvent à chercher un **plus court chemin** (ou un plus long, pour un maximum) dans ce graphe.



Les états (i, j) de la PLSC de `AB` et `BA` (partie 6) : diagonale si les dernières lettres sont égales, dessus et gauche sinon.

4.3 Calculer une complexité

Une question de complexité accompagne chaque programmation dynamique des sujets. Le calcul suit toujours le même gabarit, et les erreurs relevées par les jurys sont toujours les mêmes.

Méthode.

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

Chaque état est calculé une seule fois. Son coût est le travail fait **dans** son propre calcul : boucles, comparaisons, fonctions auxiliaires appelées. Lire un autre état (une case du tableau, ou un appel récursif dont le résultat est déjà dans `memo`) compte pour $O(1)$: le travail de cet autre état est compté dans son propre coût, pas deux fois. Le temps total est la somme des coûts de tous les états, d'où le produit quand ils ont tous le même coût.

- **Tous les paramètres.** Le rendu de monnaie a $n+1$ états à $O(|\mathcal{P}|)$ chacun : $O(n\ |\mathcal{P}|)$. Le sac à dos a $(n+1)(b+1)$ états : $O(nb)$, et rien ne permet d'écrire $O(n^2)$, b n'étant pas de l'ordre de n. On garde n et m pour deux mots.
- **Les fonctions appelées comptent.** Si le calcul d'un état appelle une fonction auxiliaire, son coût multiplie le nombre d'états : deux boucles imbriquées ne font pas $O(n^2)$ si le corps de la boucle n'est pas en $O(1)$.
- **Mémoïsation.** Même gabarit, avec le nombre de clés **atteintes** au lieu du nombre de cases : au pire le même, parfois bien moins.
- **Reconstruction.** Longueur du chemin remonté × coût pour retrouver un choix : $O(1)$ par étape avec un tableau de choix, le coût d'un état sinon. Ce coût s'ajoute et doit être annoncé.
- **Mémoire.** Le nombre d'états pour un tableau (moins si l'on ne garde que la ligne précédente), pour la mémoïsation, le dictionnaire **et** la pile d'appels.

Exemple : le rendu de monnaie de la partie 3. Les états sont les $n+1$ entiers $k \in [\![0, n]\!]$. Le calcul de l'état k parcourt les pièces et lit une case `F[k - p]` par pièce : $O(|\mathcal{P}|)$. Temps : $(n+1) \times O(|\mathcal{P}|) = O(n\ |\mathcal{P}|)$, pour le tableau comme pour la mémoïsation (au plus $n+1$ états atteints, chacun développé une fois, les autres appels sont des lectures en $O(1)$). Reconstruction : au plus n étapes en $O(1)$ grâce à `choix`, soit $O(n)$, qui s'ajoute sans changer l'ordre. Mémoire : $O(n)$ pour `F` et `choix` ; en mémoïsation, le dictionnaire **et** la pile d'appels, de profondeur jusqu'à n.

Un exemple où le coût d'un état n'est pas constant, tiré du sujet Mines 2023 (justification d'un paragraphe), est traité dans le fascicule de compléments.

4.4 Enjeux de mémoire

Le programme évoque la complexité **en mémoire**, souvent du même ordre que le nombre d'états, ce qui peut être rédhibitoire pour un tableau à deux dimensions de grande taille.

- Lorsque la récurrence ne fait intervenir que la ligne précédente, on peut ne conserver que **deux lignes** (voire une seule, parcourue dans le bon sens) : mémoire $O(m)$ au lieu de $O(nm)$. On perd alors la possibilité de reconstruire la solution à partir du tableau (exemple de la plus longue sous-suite commune dans le fascicule de compléments).
- En mémoïsation, seuls les états atteints sont stockés : pour le sac à dos avec des poids tous grands, bien moins que les $(n+1)(b+1)$ cases du tableau.
- La profondeur de récursion est une consommation de mémoire à part entière (pile d'appels) : pour la plus longue sous-suite commune mémoïsée de deux mots de 600 lettres sans lettre commune, la profondeur atteint $n+m = 1200$ (elle descend d'abord en i, puis en j), au-delà de la limite par défaut de Python (1000).

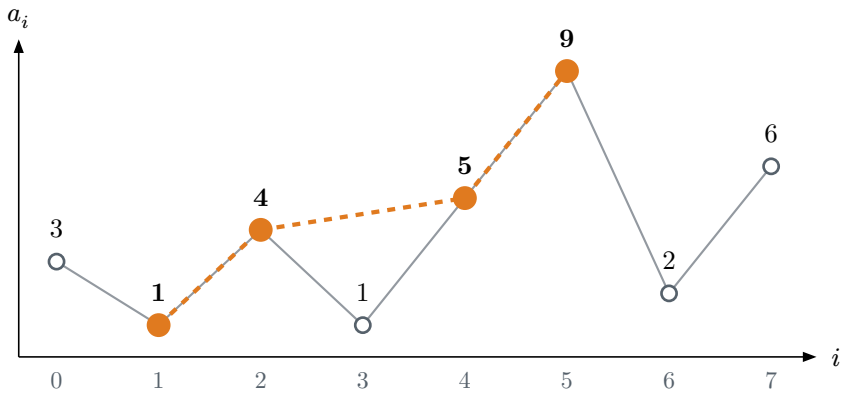
5 La plus longue sous-suite croissante (LIS) : choisir ses états

Dans le rendu de monnaie, la récurrence naturelle donnait directement les bons états. Ce n'est pas toujours le cas : **le choix des états est le vrai enjeu de la programmation dynamique**.

5.1 Le problème

Une *sous-suite* d'une suite $(a_0, ..., a_{n-1})$ est une suite de la forme $\left(a_{\varphi(0)}, ..., a_{\varphi(p-1)}\right)$ où $\varphi : [\![0, p-1]\!] \to [\![0, n-1]\!]$ est strictement croissante : on garde certains termes, pas forcément consécutifs, dans leur

ordre. On cherche la longueur maximale p d'une sous-suite **strictement croissante**. Par exemple, pour $(3, 1, 4, 1, 5, 9, 2, 6)$, la sous-suite $(1, 4, 5, 9)$ est croissante, de longueur 4, et c'est le maximum (atteint aussi par $(3, 4, 5, 9)$ ou $(1, 4, 5, 6)$). On représente une suite par la ligne brisée des points (i, a_i) : une sous-suite croissante est un choix de points qui « montent » de gauche à droite.



La suite $(3, 1, 4, 1, 5, 9, 2, 6)$ et la sous-suite $(1, 4, 5, 9)$.

5.2 Solution très naïve

La solution la plus naïve engendre **toutes** les sous-suites, garde celles qui sont strictement croissantes et retient la plus longue.

Question. 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 ?

5.3 Algorithme glouton ?

Question. Proposer un algorithme glouton pour ce problème. Est-il optimal ?

En pratique, un algorithme glouton plus subtil, qui garde non pas *une* sous-suite mais la meilleure sous-suite *de chaque longueur*, est optimal, en $O(n\ln(n))$ (fascicule de compléments : non exigible, mais la dichotomie qu'il utilise est à connaître). Commençons par la méthode générale.

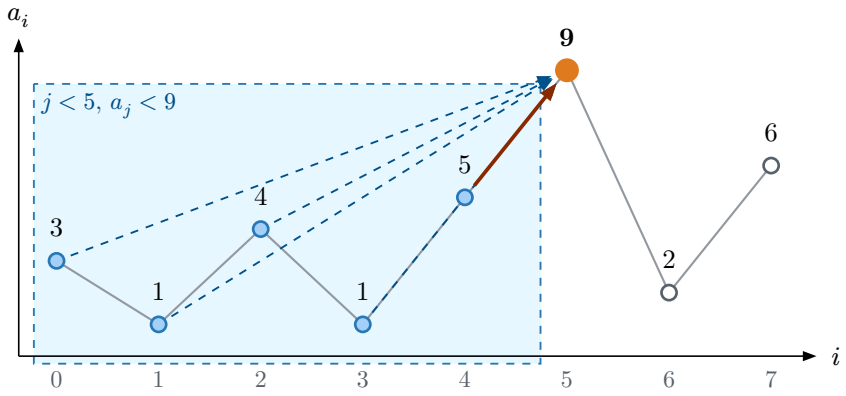
5.4 Choisir les états : sous-suites se terminant en i

L'état naturel serait la position i : la meilleure sous-suite de $a_i, ..., a_{n-1}$. Il ne suffit pas : pour savoir si l'on peut prendre a_i, il faut aussi connaître le dernier terme pris (comme `dernier` dans `lis_glouton`). Plutôt que de l'ajouter à l'état (fascicule de compléments), on change de point de vue : on fixe le **dernier** terme de la sous-suite.

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

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

Question. Justifier que $L(i) = 1 + \max\big(\{L(j),\ j < i \text{ tel que } a_j < a_i\}\big)$, avec la convention $\max(\varnothing) = 0$. Où se trouve la réponse au problème ?

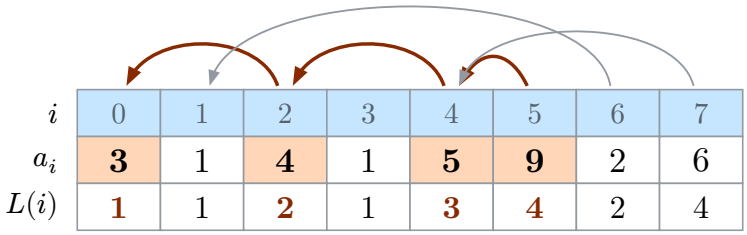


Les candidats pour $L(5)$: les points $j < 5$ avec $a_j < a_5$, dans le rectangle.

Voici `lis(a)`, qui calcule les $L(i)$ de bas en haut puis reconstruit une sous-suite strictement croissante de longueur maximale :

```
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
    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
```

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



Les $L(i)$, les prédécesseurs et la reconstruction depuis $i = 5$.

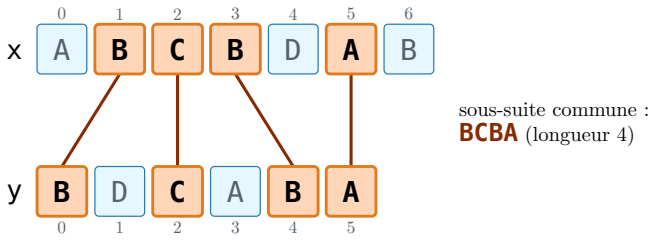
Un exercice à faire seul sur le même modèle, le **sous-tableau de somme maximale**, est proposé dans le fascicule de compléments.

6 Deux préfixes : la plus longue sous-suite commune

6.1 Le problème

Une *sous-suite commune* à deux mots $x = x_0 \ldots x_{n-1}$ et $y = y_0 \ldots y_{m-1}$ est un mot qui est sous-suite de l'un et de l'autre. La longueur maximale d'une telle sous-suite (PLSC) mesure la ressemblance entre x et y.

Pour $x =$ `ABCBDAB` et $y =$ `BDCABA`, le mot `BCBA` est une sous-suite commune de longueur 4, maximale.



Une plus longue sous-suite commune à `ABCBDAB` et `BDCABA`.

Voici `est_sous_suite(z, y)`, qui teste si le mot `z` est une sous-suite de `y` en appariant chaque lettre de `z` **au plus tôt** dans `y` (un algorithme glouton correct : apparier au plus tôt ne peut pas nuire aux lettres suivantes), et `plsc_exhaustif(x, y)`, qui essaie toutes les sous-suites de `x` (fonction `sous_suites` de la partie 5) :

```
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
```

Question. Quelle est la complexité de `plsc_exhaustif` ?

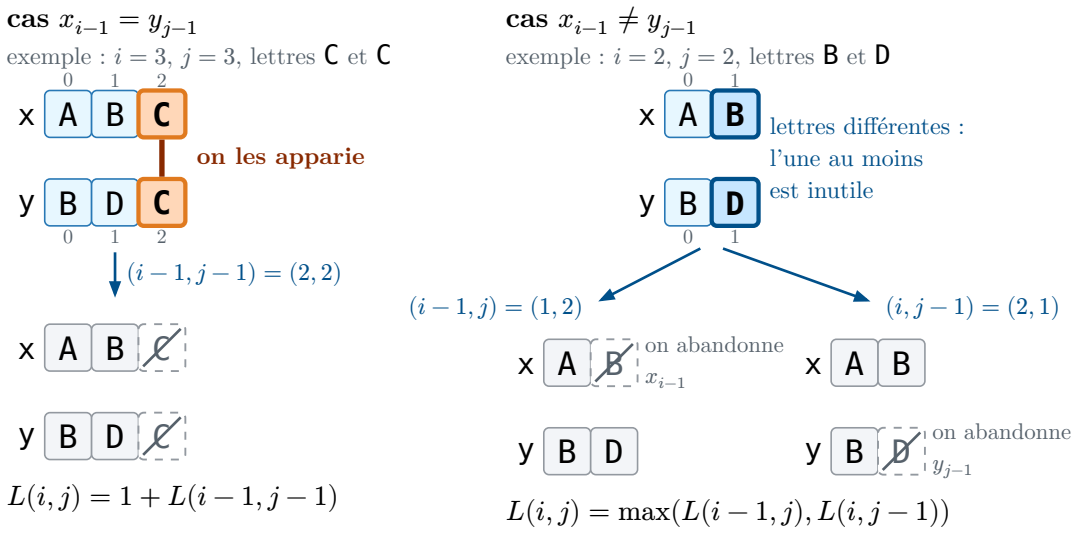
Question. Un algorithme glouton : parcourir `x` et apparier chaque lettre à sa **première** occurrence encore disponible dans `y`. L'écrire et montrer qu'il n'est pas optimal.

6.2 États et récurrence

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

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

Question. 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 $L(i,j)$.



Les deux cas selon les dernières lettres des préfixes.

6.3 Calcul : le tableau et sa complexité

Les états forment un tableau de $(n+1) \times (m+1)$ cases : la ligne 0 et la colonne 0 sont les cas de base. La case (i, j) peut dépendre des cases du dessus, de gauche et en diagonale : un remplissage ligne par ligne, de gauche à droite, convient.

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						

Les trois cases dont peut dépendre une case (i, j), ici $(3, 4)$: la diagonale ne sert que si les lettres sont égales.

Question. 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 ?

Attention. Ne jamais écrire `[[0] * (m + 1)] * (n + 1)` : les $n+1$ lignes seraient la **même** liste (chapitre précédent). La compréhension `[[0 for j in range(m + 1)] for i in range(n + 1)]` crée $n+1$ listes distinctes, X-ENS 2023 fournit d'ailleurs cette instruction dans l'énoncé.

6.4 Reconstruction

Question. 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 ?

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

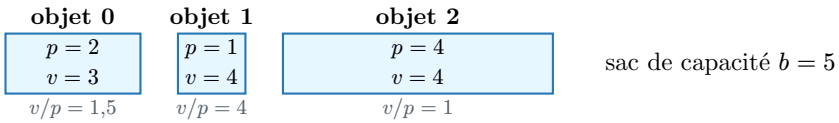
Le tableau et la remontée depuis $(7,6)$: les pas en diagonale donnent `BCBA`.

La **distance d'édition** (ou de Levenshtein), au programme, se traite avec les mêmes états, deux préfixes : c'est l'objet du TP, et sa récurrence, avec la variante du sujet X-ENS 2023, est rappelée dans le fascicule de compléments.

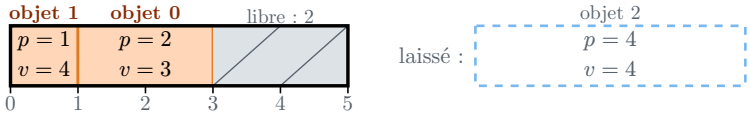
7 Un préfixe et une ressource : le sac à dos

7.1 Le problème

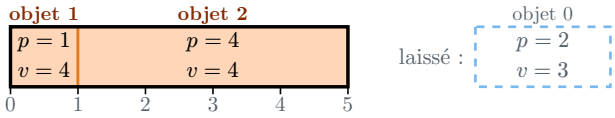
Reprenons le sac à dos de la partie 2 : n objets de poids p_i et de prix v_i (entiers strictement positifs), une capacité b, chaque objet au plus une fois.



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



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



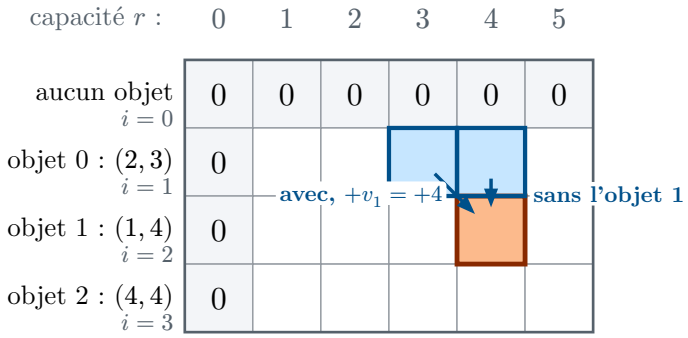
7.2 États et récurrence

L'état naturel est le préfixe i (seuls les objets $0, \dots, i-1$ sont autorisés). Il ne suffit pas : pour décider de l'objet $i-1$, il faut savoir quelle **capacité** il reste pour les autres.

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

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

Question. Donner et justifier la récurrence vérifiée par P.



Les deux dépendances de la case $(2, 4)$.

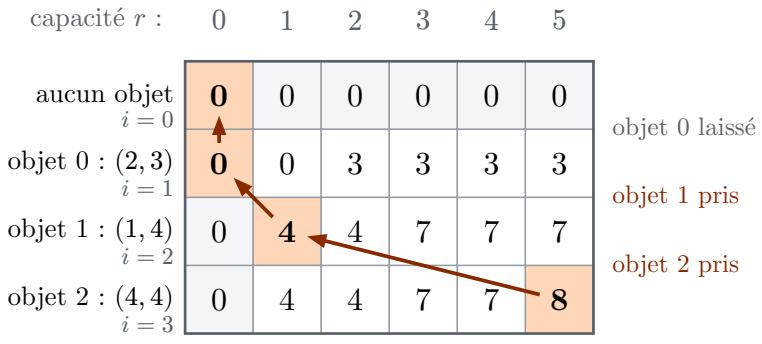
7.3 Calcul : le tableau et sa complexité

Question. 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					

7.4 Reconstruction

Question. 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.



Le tableau et la remontée depuis $(3, 5)$: objets 2 et 1, prix 8.

Le même tableau, rempli de **booléens**, résout un problème de décision du programme, la partition équilibrée (fascicule de compléments).