Programmation dynamique

Capacités attendues

- ✓ Utiliser la programmation dynamique pour écrire un algorithme.
- ✓ Les exemples de l'alignement de séquences ou du rendu de monnaie peuvent être présentés.
- ✓ La discussion sur le coût en mémoire peut être développée.

Nous avons vu cette année que des problèmes pouvaient être résolus efficacement en les divisant en plus petits problèmes, dont on combine les solutions pour obtenir une solution à notre problème initial : c'est l'approche **diviser pour régner**. Cette méthode algorithmique est puissante mais pas toujours optimisée en temps. Nous allons nous intéresser à une manière qui permet d'obtenir des algorithmes plus rapides à s'exécuter : la **programmation dynamique**.

L'exemple choisi pour ce cours est celui, classique, du calcul des nombres de la suite de **Fibonacci**. Cette suite u_n est définie pour tout entier n par :

$$u_n = \begin{cases} n & \text{si } n \leq 1 \\ u_{n-1} + u_{n-2} & \text{sinon} \end{cases}$$

Les 10 premiers termes de la suite sont :

Chaque terme est calculé en faisant la .

→ On souhaite programmer la fonction `fibo`, qui, pour tout entier `n`, renvoie le terme d'indice `n` de la suite.

*Le cours se déroule sous la forme d'une **activité** qui comporte plusieurs questions numérotées ci-dessous.*

1 Approche récursive

La définition mathématique de la suite, avec la récurrence, guide vers une solution récursive.

→ Télécharger sur notre site le fichier `fibo_dyn.py`

1.1 Approche naïve

1. Écrire une fonction récursive `fibo_rec(n)` où `n` est un entier positif, et qui renvoie le `n`-ième terme de la suite de Fibonacci.

2. Dessiner l'arbre binaire représentant les appels récursifs effectués pour le calcul de `fibo_rec(5)` :

3. Quels appels sont faits plusieurs fois, pour la même valeur de `n` ?

4. Dans quel ordre les appels récursifs sont-ils faits ?

La complexité en temps de cet algorithme exponentielle, c'est-à-dire très mauvaise : plus il y a de calculs à faire, plus le temps passé à les faire augmente de manière critique. La raison de cette inefficacité se résume en fait à la **multiplicité de calcul d'un même nombre**.

1.2 Améliorations

La clef d'une solution plus efficace serait de s'affranchir de la multiplicité des résolutions du même sous-problème. On améliore de loin la complexité temporelle si, une fois calculé, on sauvegarde un résultat puis que, au besoin, on le reprend depuis ce tableau.

Ici, pour éviter de faire plusieurs fois les mêmes calculs, nous allons donc les termes de la suite déjà calculés dans une variable `memo` (pour mémoire) de type `List` Python.

5. En se basant sur l'arbre des appels pour `fibo_rec(3)`, représenter cette liste `memo` au fur et à mesure qu'elle se remplit :

memo[0]	memo[1]	memo[2]	memo[3]	memo[4]	memo[5]

Ce genre de tableau est appelé tableau de **mémoïsation**.

6. Écrire en Python la façon dont on doit initialiser ce tableau.

7. Écrire une nouvelle fonction `fibo_rec_memo(n, memo=[])` qui :
   - initialise le tableau de mémoïsation s'il est vide ;
   - récupère la valeur `memo[n]` si elle existe dans le tableau de mémoïsation et la renvoie ;
   - et qui, si elle n'existe pas dans le tableau, la calcule et la stocke dans `memo[n]`.

C'est le principe de la programmation dynamique : mémoriser les résultats des calculs au fur et à mesure, pour ne pas avoir à les recalculer après. L'avantage est un gain de temps puisqu'on n'a pas à refaire les calculs à chaque fois. Cette approche de résolution est connue sous le nom de **fonction à mémoire**.

2 Approche itérative

On utilise souvent cette méthode algorithmique pour améliorer les performances en temps des programmes récursifs, mais son principe s'applique aussi sur des programmes itératifs.

8. À partir de l'observation du tableau rempli dans la partie précédemment, proposer une version itérative de cette fonction à mémoire `fibo_ite_memo(n)` (elle utilisera donc le principe de la programmation dynamique).

De cette façon, la **complexité** de notre algorithme est à présent **linéaire** : c'est un énorme progrès.

3 Synthèse

Tester les fonctions écrites précédemment pour différentes valeurs de `n` et reporter dans le tableau ci-dessous l'**ordre de grandeur** du temps de calcul nécessaire pour obtenir le résultat parmi :

imm pour immédiat – *sec* pour secondes – *min* pour minutes – *indét* pour indéterminé – *stacko* pour Stack Overflow.

→ Décommenter dans le fichier Python tout ce qui est après le commentaire `# Mesures des temps d'exécution` pour mesurer les temps.

n	5	20	30	50	100	1000	10000
fibo_rec							
fibo_rec_memo							
fibo_ite_memo							

La programmation dynamique est une amélioration de la méthode algorithmique *diviser pour régner*. Elle reprend son principe, mais ajoute le stockage des résultats intermédiaires (des sous-problèmes) pour éviter de les recalculer. Elle est très utile lorsque beaucoup des sous-problèmes sont identiques.

Travaux dirigés : le problème du rendu de monnaie

Étant donné un système de monnaie comportant des pièces de valeurs différentes, on veut écrire un programme renvoyant le nombre minimal de pièces à utiliser pour rendre la monnaie sur une somme due.

On suppose qu'il est possible de rendre la somme due avec le système de monnaie que l'on utilise.

1 L'approche gloutonne

Au programme de 1ère NSI, on étudie qu'il est possible de résoudre ce problème avec un **algorithme glouton**.

Son principe est de choisir **la pièce de valeur maximale**, inférieure à la somme à rendre. On actualise la somme à rendre à chaque fois qu'on sélectionne une nouvelle pièce, et on réitère l'algorithme.

1. Quelles pièces vont être rendues avec le système de monnaie `euro = [50, 20, 10, 5, 2, 1]` pour payer la somme `48` ?

2. Quelles pièces vont être rendues avec le système de monnaie `imperial = [30, 24, 12, 6, 3, 1]` pour payer la somme `48` ? Est-ce une solution optimale ?

→ Télécharger sur notre site le fichier `monnaie_dyn.py`

Voici un implémentation possible de cet algorithme glouton en Python.

`rendu_glouton` prend en paramètres un entier `somme` représentant la somme à rendre et une liste `pieces` représentant les pièces dont on dispose pour le faire, rangées par valeurs décroissantes comme décrit ci-dessus.

```
def rendu_glouton(somme, pieces):
    nb_pieces = 0
    i = 0
    while somme > 0 and i < len(pieces):
        if pieces[i] > somme:
            i = i + 1
        else:
            somme = somme - pieces[i]
            nb_pieces = nb_pieces + 1
    return nb_pieces
```

(TSVP)

3. Vérifier qu'on obtient le même résultat que précédemment pour la monnaie impériale en déroulant l'algorithme à la main à l'aide du tableau suivant.

	somme	nb_pieces	i	pieces[i]
init	48	0	0	30

L'algorithme glouton ne renvoie donc pas toujours la solution optimale !

2 Solution optimale avec une approche récursive

Voici un algorithme explorant toutes les solutions et sélectionnant celle optimale, en utilisant la récursivité.

```
def rendu_recursif(somme, pieces):
    if somme == 0:
        return 0

    mini = somme
    for p in pieces:
        if p <= somme:
            nb = 1 + rendu_recursif(somme - p, pieces)
            if nb < mini:
                mini = nb
    return mini
```

Il est possible de construire un **algorithme optimal de manière récursive**.

Il faut pour cela faire les observations suivantes :

- Pour rappel, le rendu est toujours possible : dans le pire des cas, le nombre de pièces à rendre est égal à la somme de départ (rendu effectué en pièces de 1) ;
- Si `p` est une pièce de `pieces`, le nombre minimal de pièces nécessaires pour rendre la somme somme est égal à 1 + le nombre minimal de pièces nécessaires (contenant `p`) pour rendre la somme `somme - p`.

Cette dernière observation est cruciale. Elle repose sur le fait qu'il suffit de ajouter 1 pièce (la pièce de valeur `p`) à la meilleure combinaison qui rend `somme - p` pour avoir la meilleure combinaison qui rend `somme` (meilleure combinaison parmi celles contenant `p`).

On va donc passer en revue toutes les pièces `p` et mettre à jour à chaque fois le nombre minimal de pièces `min`.

L'algorithme ne se laisse pas piéger comme l'algorithme glouton et rend bien en 2 pièces la somme 48 avec la monnaie impériale.

(TSVP)

4. (a) Tracer l'arbre représentant les appels récursifs fait pour l'appel :

   `rendu_recursif(6, [50, 20, 10, 5, 2, 1])`.

   Ses nœuds contiennent la valeur de la somme restante à rendre, et on associe à chaque branche la pièce choisie correspondante.

   (b) Identifier les appels effectués plusieurs fois (en les entourant avec des couleurs par exemple).

   (c) Entourer la solution optimale.

Malheureusement, le nombre d'appels récursifs de notre algorithme augmente **exponentiellement** avec la valeur de la somme à rendre : on se retrouve très rapidement avec des milliards d'appels récursifs, ce qui n'est pas gérable (`RecursionError` en Python). D'ailleurs, si vous faites différents essais sur votre ordinateur, vous observerez vite les limites de cet algorithme. L'exécution ne se terminera probablement pas.

Ces appels récursifs ont lieu sur un nombre limité de valeurs mais les calculs sont très nombreux : si la somme à rendre est 100, il y aura beaucoup (beaucoup) d'appels vers 99, vers 98, vers 97... jusqu'à 0.

On peut donc légitimement penser à **mémoïser notre algorithme**, en stockant les valeurs pour éviter de les recalculer.

5. Compléter la fonction `rendu_recursif_dyn` qui prend en paramètres une liste de pièces `pieces` et la somme à rendre `somme` et qui renvoie le nombre minimal de pièces qu'il faut rendre.

   On utilisera le dictionnaire `memo_rendu` dans lequel on associera à chaque somme `somme` son nombre de pièces minimal.

   On procèdera de manière classique :

   - Soit la somme est disponible dans le dictionnaire, et on se contente de renvoyer la valeur associée ;
   - Soit on la calcule (comme dans l'algorithme classique), puis on stocke le résultat dans le dictionnaire avant de le renvoyer.

```
def rendu_recursif_dyn(somme, pieces, memo = []):
    if memo == []:
        memo = [None for _ in range(somme+1)]

    if somme == 0:
        return 0

    if ..... != .....:
        return .....

    mini = somme
    for p in pieces:
        if p <= somme:
            nb = .....
            if nb < mini:
                mini = .....
                ..... = nb
    return .....
```

6. Tester les cas précédents avec cette fonction à mémoire. Qu'observe-t-on ?

7. Enfin, compléter la version itérative avec mémoïsation de ce rendu de monnaie.

```
def rendu_ite_dyn(somme, pieces):
    memo = [0]
    for s in range(1, somme+1):
        ........
        for p in pieces:
            if p <= s:
                ........
    return ........
```

8. Tester aussi cette dernière fonction pour vérifier qu'elle donne bien le résultat optimal pour les valeurs précédemment étudiées.

Notre algorithme itératif est de complexité (par rapport à la variable `somme`).