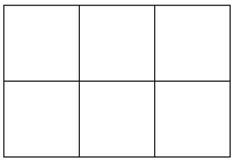
Chapitre 17 - Programmation dynamique

Objectifs :

- ▷ Utiliser la programmation dynamique pour écrire un algorithme.
- ▷ Comprendre les exemples du rendu de monnaie et de l'alignement de séquences.
- ▷ Comparer un algorithme de force brute, un algorithme glouton et un algorithme en programmation dynamique sur le problème du rendu de monnaie.
- ▷ Montrer les conséquences sur le coût en mémoire des algorithmes de programmation dynamique

1 Introduction

On dispose de la grille 2×3 ci-dessous.



Question : Combien de chemins mènent du coin supérieur gauche au coin inférieur droit, en se déplaçant uniquement le long des traits horizontaux vers la droite et le long des traits verticaux vers le bas ?

Et pour une grille 10×10 ?

2 Principe de la programmation dynamique

La programmation dynamique est une technique dûe à *Richard Bellman* dans les années 1950. À l'origine, cette méthode algorithmique était utilisée pour résoudre des problèmes d'**optimisation**.

Repères historiques

Richard Bellman est un mathématicien américain, travaillant principalement dans la branche des mathématiques appliquées. Il est l'inventeur de la **programmation dynamique**, qui résolut à son époque de façon inespérée l'optimisation des sommes de fonctions monotones croissantes sous contraintes.

Le terme programmation désigne la **planification**, et n'a pas de rapport avec les langages de programmation.

A retenir !

L'idée générale est de déterminer un résultat sur la base de calculs précédents.

Plus précisément, la programmation dynamique consiste à résoudre un problème :

— en le **décomposant en sous-problèmes**,

— puis à **résoudre les sous-problèmes** des plus petits au plus grands

— **en stockant les résultats intermédiaires**.

3 La suite de Fibonacci

3.1 Rappels

On a déjà abordé cette suite lorsque nous avons parlé de la programmation récursive (Chapitre 1).

La **suite de Fibonnacci** est une suite de nombres dont chacun est la somme des deux précédents. Le premier et le second nombres sont égaux à 0 et 1 respectivement.

On obtient la suite de nombres : `0 - 1 - 1 - 2 - 3 - 5 - 8 - 13 - 21` ..., *etc.*

Mathématiquement, cette suite notée F_n est définie par :

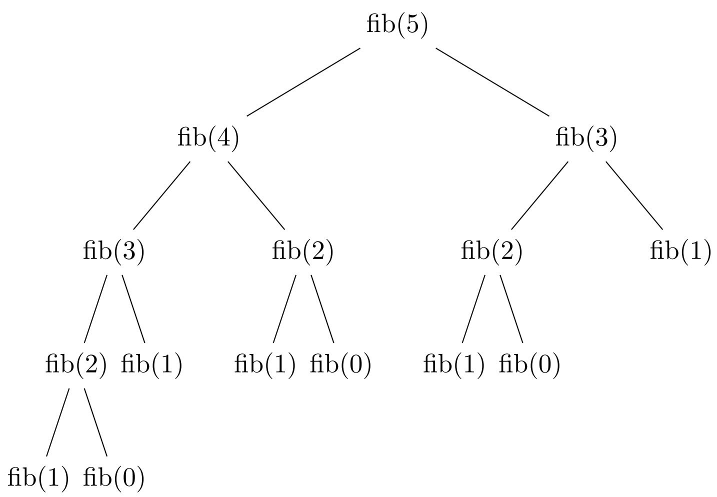
$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ pour tout entier} \geq 2 \end{cases}$$

3.2 Version récursive naïve (et inefficace)

Nous avons déjà programmé une version récursive qui renvoie le terme de rang n de cette suite.

```python
def fib(n):
    """Version recursive naive"""
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Par exemple, voici l'arbre des appels récursifs si on lance `fibo(5)`.



On se rend compte qu'il y a beaucoup d'**appels redondants** :

- `fib(1)` a été lancé 5 fois,
- `fib(2)` a été lancé 3 fois,
- *etc.*

Ces **redondances** entraînent un nombre d'appels récursifs qui explose rapidement dès que n est élevé. Par conséquent, les temps de calcul deviennent vite très élevés. Pire, dès que n est trop grand, l'algorithme ne donnera jamais la réponse.

Par exemple, en utilisant un programme principal qui calcule et affiche le temps d'exécution :

```python
start_time = time.time()                    # Debut du chronometre
fib(25)                                     # Code a mesurer
end_time = time.time()                      # Fin du chronometre

execution_time = end_time - start_time       # Calcul du temps ecoule
print("Temps d'execution :", execution_time, "secondes")
```

On obtient :

- si on exécute `fibo(25)`, temps d'exécution : `30 ms`
- si on exécute `fibo(35)`, temps d'exécution : `3.62 s`
- si on exécute `fibo(40)`, temps d'exécution : `36.77 s`

A retenir !

Il est possible de faire mieux, en évitant de refaire les calculs déjà effectués.

Pour cela, il faut **stocker les résultats intermédiaires** !

3.3 Version récursive avec mémoïsation

Une première approche est d'adapter l'algorithme récursif **en stockant les résultats calculés dans un tableau ou un dictionnaire**.

Lors d'un appel, on commence par vérifier si on ne connaît pas déjà la réponse, auquel cas on la renvoie directement, ce qui évite d'effectuer des **calculs redondants**.

Cela donne la fonction `fibo_memo` suivante qui prend en paramètres un entier `n` et un dictionnaire `memo` que l'on met à jour en stockant les résultats intermédiaires au fur et à mesure.

```
1 def fibo_memo(n,memo):
2     if n in memo:                # si calcul deja effectue
3         return memo[n]           # on renvoie directement sa valeur
4     elif n <= 1:
5         memo[n] = n
6         return memo[n]
7     else:
8         memo[n] = fibo_memo(n-1,memo) + fibo_memo(n-2,memo)
9         return memo[n]
```

Explications :

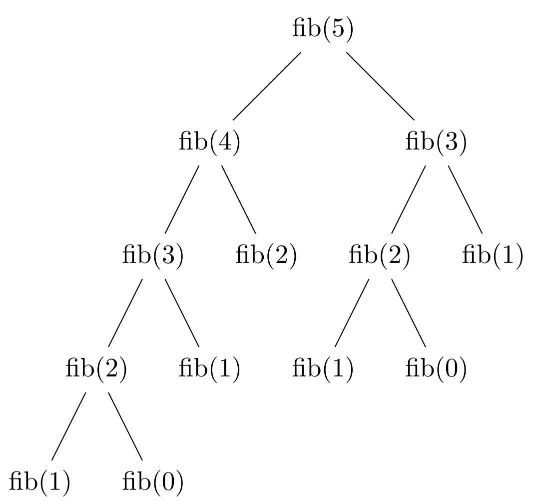
- **Lignes 2 et 3 :** si la valeur n est déjà dans le dictionnaire, c'est qu'on a déjà calculé F_n et il suffit alors de renvoyer sa valeur `mem[n]` (la valeur associée à `n`).
- **Lignes 4 à 9 :** quasiment identiques à la version récursive naïve à ceci près que l'on mémorise la valeur dans le dictionnaire `memo` avant de la renvoyer.
- De cette façon, dès qu'une valeur F_n a été calculée, elle est ajoutée dans le dictionnaire comme la valeur associée à `n`, ce qui permet de la réutiliser directement dès qu'on en a besoin.

Il n'y a plus qu'à lancer le premier appel avec un dictionnaire vide, c'est ce que fait la fonction fibo suivante.

```
def fibo(n):
    """Version recursive avec memoisation"""
    F = {}
    return fibo_memo(n,F)
```

Avec ce **procédé de mémoïsation**, l'arbre des appels est considérablement réduit puisqu'il n'y a plus aucun appel redondant.

Par exemple, l'arbre des appels récursifs en lançant `fibo(5)` se réduit à :



On constate alors qu'avec cette version, les valeurs F_n sont calculées quasiment instantanément et que l'on peut obtenir les valeurs F_n pour des grandes valeurs de `n`.

On obtient :

— si on exécute `fibo(25)`, temps d'exécution : `0 ns`
— si on exécute `fibo(100)`, temps d'exécution : `3.62 s`
— si on exécute `fibo(850)`, temps d'exécution : `2.04 ms`

La méthode descendante

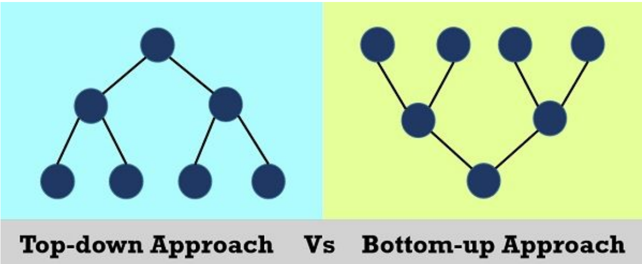
La version récursive avec mémoïsation correspond à une **approche descendante**, aussi appelée **haut-bas** (ou *top-down* en anglais).

En effet, pour connaître F_n , on lance l'appel `fibo(n)` qui déclenche la **descente d'appels récursifs** jusqu'aux cas de base pour lesquels on mémorise les résultats.

Dans un second temps, on remonte les appels tout en mémorisant leurs résultats pour ne pas résoudre plusieurs fois le même problème.

Finalement, avec cette méthode, c'est lors de la **remontée des appels** que leurs résultats sont mémorisés puis réutilisés sur les problèmes plus grands. On peut alors se demander si on ne peut pas procéder directement du plus petit sous-problème au plus grand (celui que l'on veut résoudre).

La réponse est oui ! Une autre manière de résoudre le problème est d'utiliser une **approche ascendante**.



3.4 Version itérative ascendante

On parle aussi de **méthode bas-haut**, ou *bottom-up* en anglais.

Il s'agit d'une méthode itérative dans laquelle on commence par calculer des solutions pour les sous-problèmes les plus petis puis, de proche en proche, on arrivera à la taille voulue. Comme précédemment, on utilise le principe de la mémoïsation pour stocker les résultats partiels.

Le calcul du terme F_n de la suite de la Fibonacci n'est pas un problème d'optimisation, ainsi le calcul d'une solution d'un problème à partir des solutions connues des sous-problèmes est simple puisqu'il n'y a aucun choix à faire.

De manière générale, on utilise un tableau pour stocker les résultats au fur et à mesure.

La méthode descendante

Voici les étapes habituelles :

1. **Création et initialisation du tableau :**
   - On a besoin d'un tableau `F` de taille *n+1* qui va contenir les valeurs F_0, F_1, ...,F_n dans cet ordre.
   - Pour cela on crée le tableau F avec *n+1* zéros initialement.
   - On peut stocker les valeurs déjà connues (F0 et F1 dans notre cas)
2. **Utilisation de la formule de récurrence pour remplir le reste du tableau :**
   - La formule de récurrence donne la solution d'un sous-problème à partir de celles de sous-problèmes plus petits et donc déjà traités ! Ici on a pour $2 \leq i \leq n$: $F_i = F_{i-1} + F_{i-2}$
   - On peut donc remplir le tableau F en parcourant les indices **par ordre croissant** : on va mettre dans `F[i]` la valeur `F[i-1]` + `F[i-2]` que l'on connaît puisque ces deux valeurs ont été calculés précédemment.
3. **Le résultat est dans la dernière case du tableau : on la renvoie !**

Cela donne la fonction suivante :

```
def fibo(n):
    """Version iterative ascendante"""
    F = [0]*(n+1)
    F[0] = 0  # pas indispensable car deja initialise a 0
    F[1] = 1
    for i in range(2,n+1):
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

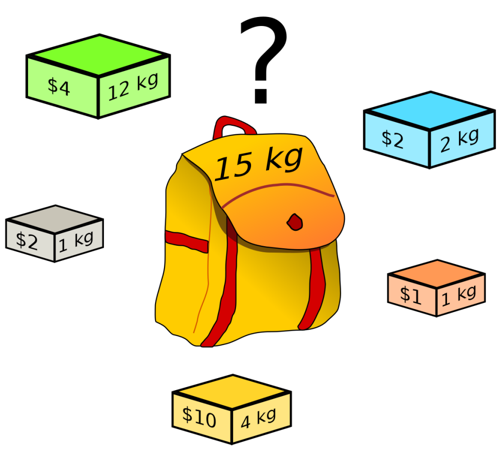
Les performances sont semblables à la version récursive avec mémoïsation

On obtient, si on exécute `fibo(850)`, un temps d'exécution : `0 ns`

3.5 Autres problèmes

Il existe de nombreux problèmes pouvant être résolus avec le **paradigme de la programmation dynamique**, dont beaucoup de problèmes d'optimisation :

- ▷ problème du rendu de monnaie
- ▷ problème du sac-à-dos
- ▷ alignement de séquences
- ▷ problème du plus court chemin (algorithme de *Bellman-Ford* utilisé par le protocole RIP)
- ▷ problèmes d'ordonnancement d'intervalles pondérés
- ▷ toutes sortes de problème d'affectation des ressources, ..., *etc.*



4 Conclusion

- La programmation dynamique est une technique permettant d'**améliorer l'efficacité d'un algorithme** en évitant les **calculs redondants**.
- Pour cela, on utilise un tableau (ou un dictionnaire) pour **stocker les résultats intermédiaires** et pouvoir les réutiliser sans les recalculer.
- Comme la méthode *« diviser pour régner »*, la programmation dynamique permet résoudre un problème à partir des solutions de sous-problèmes. Si ces derniers se « chevauchent » (s'ils sont non indépendants) alors la programmation dynamique permettra d'éviter que les appels récursifs ne soient effectués plusieurs fois. Ainsi, la programmation dynamique permet souvent d'améliorer des algorithmes récursifs.

A retenir !

Pour utiliser la programmation dynamique, on procède généralement ainsi :

1. définition des sous-problèmes
2. identification d'une **relation de récurrence** entre les solutions des sous-problèmes
3. mise en place d'un **algorithme récursif avec mémoïsation** ou d'un **algorithme itératif ascendant**
4. résolution du problème original à partir des solutions des sous-problèmes

La programmation dynamique permet de résoudre de manière efficace de nombreux problèmes d'optimisation, comme le rendu de monnaie ou l'alignement de séquences, pour lesquels une solution récursive classique est inefficace.

5 Exercices

Exercice 1 : *L'alignement de séquences*

1. Regarder la vidéo suivante sur la programmation dynamique et l'alignement de séquences et répondre aux questions ci-dessous dans le *Google Doc*.
2. Donner le principe de base de la programmation dynamique.
3. Dans quel domaine scientifique l'utilisation de la programmation dynamique a connu un grand essor ?
4. Quel type de fonction utilise régulièrement la programmation dynamique ?
5. Détailler le principe de la recherche de l'alignement de séquences.
6. Reprendre l'algorithme de calcul de la **distance de *Levenshtein*** et le tester avec les séquences `information` et `informatique`.
7. Relancer le calcul en modifiant la matrice de *Levenshtein* en prenant un poids de 4 pour un bon alignement et -2 pour une insertion, suppression, modification. Que constatez-vous ?

Exercice 2 : *Le rendu de monnaie*

1. Regarder la vidéo suivante sur la programmation dynamique et le problème de rendu de monnaie et répondre aux questions ci-dessous dans le *Google Doc*.
2. Écrire l'algorithme en Python du rendu de monnaie force brute (et fonction récursive).
3. Dessiner le début de l'arbre des possibilités de rendue de monnaie pour 76 cts. Montrer sur cet arbre qu'il y a des redondances.
4. Écrire l'algorithme en Python du rendue de monnaie en programmation dynamique.
5. A l'aide de la fonction `time()` de la bibliothèque time de Python, évaluer le temps nécessaire pour votre ordinateur pour calculer le nombre minimal de pièces à rendre pour la somme de 76 cts à l'aide des deux algorithmes (questions 2 et 4)
6. Même chose avec 177 cts. Commenter.