Programmation dynamique

1 Introduction

En algorithmique, le principe du **diviser pour régner** repose sur une idée simple :

- découper un problème compliqué en sous-problèmes plus simples (de manière récursive)
- résoudre les sous-problèmes et combiner leurs solutions pour résoudre le problème initial

Cette méthode excelle dans de nombreux domaines (dichotomie, tri fusion, tri rapide) mais elle atteint rapidement ses limites lorsque les sous-problèmes rencontrés ne sont pas uniques (chevauchements de sous-problèmes). Dans ce cas, on en vient à recalculer les solutions de sous-problèmes déjà rencontrés ce qui rend notre programme très lent, voire inutilisable.

Le paradigme de la **programmation dynamique** apporte un moyen de pallier ce problème d'efficacité en stockant les solutions intermédiaires afin de pouvoir les réutiliser sans les recalculer. C'est un exemple parfait de compromis entre complexité temporelle et complexité spaciale.

La programmation dynamique est fréquemment employée pour résoudre des problèmes d'optimisation : elle s'applique dès lors que la solution optimale peut être déduite des solutions optimales des sous-problèmes (c'est le principe d'optimalité de Bellman). Dans les années 1950, Richard Bellman introduit le concept de *dynamic programmation*. À l'époque, « programming » signifie « optimisation » ou « ordonnancement ».

2 Exemple introductif

2.1 Algorithme récursif naïf

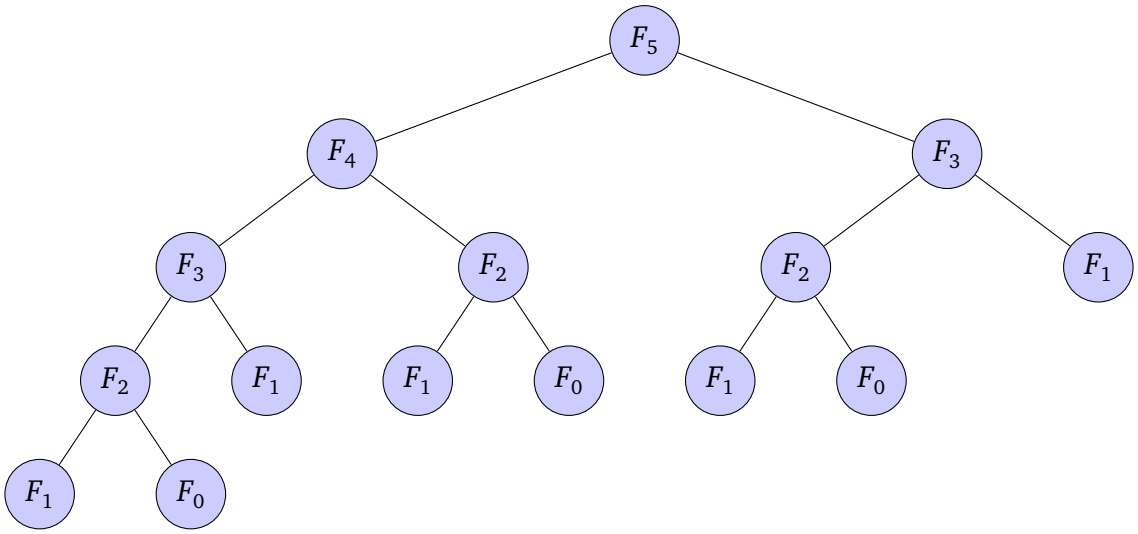
La suite de Fibonacci est la suite $(F_n)_{n\in\mathbb{N}}$ définie par

$$F_0 = F_1 = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n$$

Pour calculer les termes de la suite de Fibonacci, une solution naïve consiste à utiliser la programmation récursive :

```
def Fibonacci(n):
    if n == 0 or n == 1:
        return 1
    return Fibonacci(n-1) + Fibonacci(n-2)
```

Le problème est que ce type de fonction est très inefficace comme le montre l'arbre des appels récursifs nécessaires pour calculer F_5 :



On constate que de nombreux noeuds (et leur sous-arbre engendré) sont répétés : F_3 apparait deux fois, F_2 apparait trois fois. La complexité de cet algorithme récursif est exponentielle.

Rappel : calcul de complexité pour une fonction récursive

On note $C(n)$ la complexité pour un appel sur un problème de taille n.

- Si $C(n) = C(n-1) + O(1)$, alors $C(n) = \dots$
- Si $C(n) = C(n-1) + O(n)$, alors $C(n) = \dots$
- Si $C(n) = C(n-1) + C(n-2) + O(1)$, alors $C(n) = \dots$
- Si $C(n) = C(n//2) + O(1)$, alors $C(n) = \dots$
- Si $C(n) = 2C(n//2) + O(1)$, alors $C(n) = \dots$
- Si $C(n) = 2C(n//2) + O(n)$, alors $C(n) = \dots$

2.2 Programmation dynamique : méthode descendante par mémoïsation

Définition

La **mémoïsation** consiste à améliorer un algorithme récursif naïf en sauvegardant les solutions des sous-problèmes déjà résolus en vue de les réutiliser plus tard si besoin.

Pour calculer les termes de la suite de Fibonacci, on construit un dictionnaire dont les clés sont les nombres de 0 à n et les valeurs sont les termes de la suite correspondants :

```
M = {}

def Fibonacci(n):
    if n in M:
        return M[n]
    elif n == 0 or n == 1:
        f = 1
    else:
        f = Fibonacci(n-1) + Fibonacci(n-2)
    M[n] = f   # On enregistre chaque résultat calculé (mémoïsation)
    return f
```

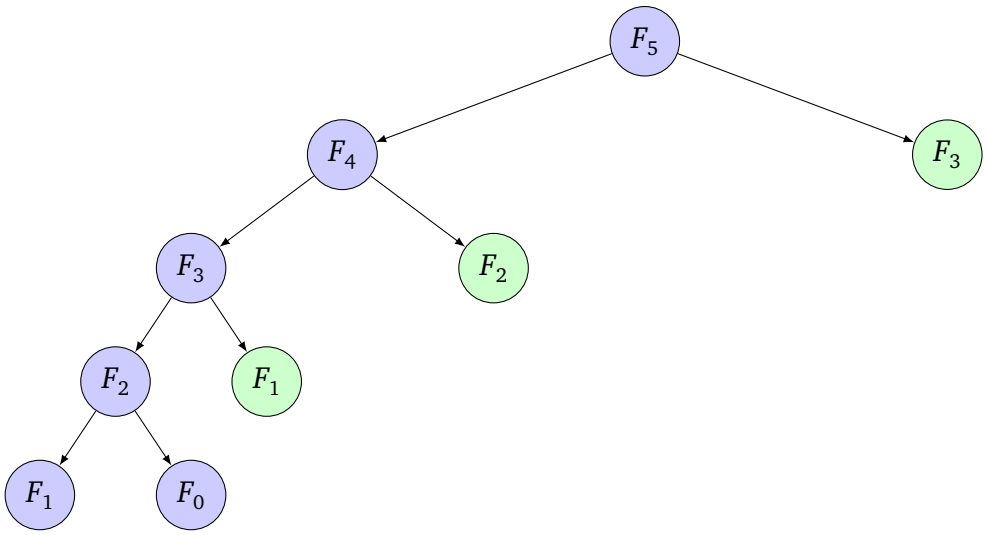
Calcul de la complexité

Soit $C(n)$ le nombre d'opérations pour calculer `Fibonacci(n)`. On a

$$C(n) = C(n-1) + 1 + \mathrm{O}(1)$$

car F_{n-2} aura été mémoïsé lors du calcul de F_{n-1} et la recherche d'un élément d'un dictionnaire se fait en $\mathrm{O}(1)$.

On a donc $\boxed{C(n) = \mathrm{O}(n)}$



Pour chaque noeud en vert, il n'y a pas d'appel récursif car le problème a déjà été résolu avant (le parcours du graphe se fait en profondeur).

Exercice

Écrire une fonction Python `binomial_memo(n, p)`, basée sur le principe de la programmation dynamique et de la mémoïsation, calculant le coefficient binomial $\binom{n}{p}$ à l'aide de la formule de Pascal

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

Quelle est sa complexité ?

2.3 Programmation dynamique : méthode ascendante (calcul de bas en haut)

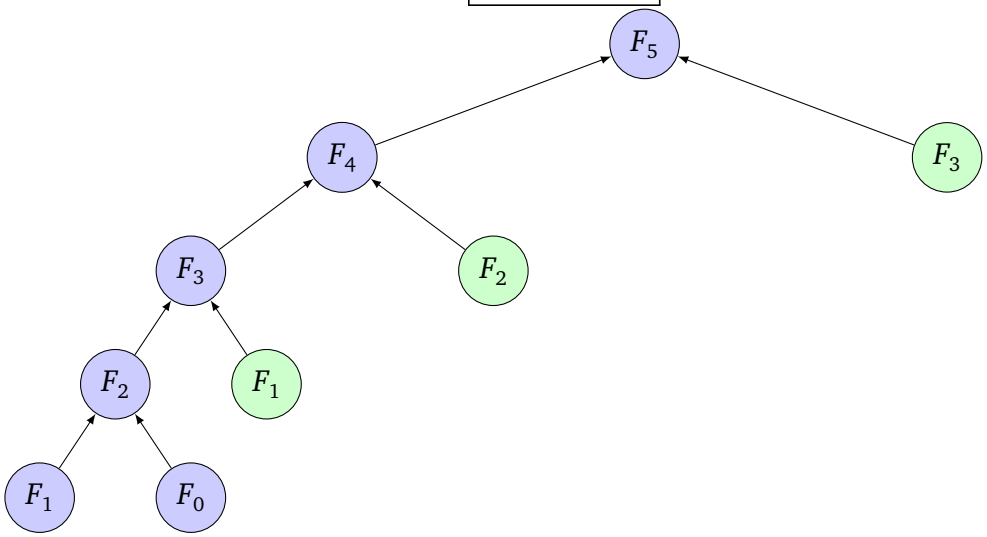
Définition

La **méthode ascendante (bottum-up en anglais)** consiste à résoudre d'abord les plus petits sous-problèmes puis de combiner leurs solutions pour résoudre des sous-problèmes de plus en plus grands jusqu'à atteindre le problème initial. L'algorithme est alors itératif.

```
def Fibonacci(n):
    F = [1, 1]
    for i in range(2, n):
        F.append(F[-1] + F[-2])
    return F[n]
```

Remarques

- La complexité de la méthode ascendante est plus facile à calculer que celle de la mémoïsation car l'algorithme n'est pas récursif : ici encore, on a $\boxed{C(n) = O(n)}$



- Dans certains cas, on peut encore optimiser la méthode ascendante pour diminuer la complexité spaciale. Par exemple ici nous pourrions garder seulement les deux derniers termes calculés car nous n'aurons plus besoin des précédents :

```python
def Fibonacci(n):
    a, b = 1, 1
    for i in range(2, n):
        a, b = b, a + b
    return b
```

- L'inconvénient de la méthode ascendante (par rapport à la mémoïsation) est qu'il faut s'assurer que les calculs sont fait dans le bon ordre et de ne pas faire trop de calculs inutiles : pour l'exemple de la suite de Fibonacci c'est assez simple mais dans le cas général il faut étudier les dépendances des différents sous-problèmes.

Exercice

Écrire une fonction Python `binomial_asc(n, p)`, basée sur le principe de la programmation dynamique et de la méthode ascendante, calculant le coefficient binomial $\binom{n}{p}$ à l'aide de la formule de Pascal

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

Quelle est sa complexité ?

3 Problème du sac à dos

3.1 Énoncé du problème

On dispose de N objets numérotés de 1 à $N \in \mathbb{N}^*$. Chaque objet i a un poids $w_i \in \mathbb{N}^*$ et une valeur $v_i \in \mathbb{N}^*$. On souhaite remplir un sac à dos de capacité maximale W_{max} avec ces objets.

Comment choisir les objets de sorte à maximiser la valeur emportée $\sum_{i \in I} v_i$ tout en respectant la contrainte $\sum_{i \in I} w_i \leqslant W_{max}$?

Ce type de problème fait partie des problèmes dits d'optimisation.

3.2 Solution récursive

Pour avoir un problème plus simple à résoudre, on peut limiter le nombre d'objets disponibles. Pour tous $i \in [\![0, N]\!]$ et $W \in [\![0, W_{max}]\!]$, notons $V(i, W)$ la valeur maximale d'un sac à dos de capacité W en n'utilisant que les objets de numéros 1 à i.

- Pour $i = 0$, il n'y a aucun objet disponible donc $V(0, W) = 0$ pour tout $W \in [\![0, W_{max}]\!]$.
- Pour $W = 0$, aucun objet ne peut être choisi (les poids sont tous strictement positifs) donc $V(i, 0) = 0$ pour tout $i \in [\![0, N]\!]$.
- Étant donné $i \in [\![1, N]\!]$:
  - ⋆ Si l'objet i est dans une solution optimale de ce sous-problème, alors $w_i \leqslant W$ et la valeur du sac à dos sera
  $$V(i, W) = v_i + V(i-1, W-w_i)$$
  - ⋆ Sinon, la valeur du sac à dos sera
  $$V(i, W) = V(i-1, W)$$

  Comme on cherche à maximiser la valeur du sac à dos, on a

$$V(i, W) = \begin{cases} \max(v_i + V(i-1, W-w_i); V(i-1, W)) & \text{si } w_i \leqslant W \\ V(i-1, W) & \text{sinon} \end{cases}$$

3.3 Conditions générale pour un problème de programmation dynamique

Pour qu'un problème soit solvable par programmation dynamique, il doit impérativement respecter deux propriétés :

- **Chevauchement des tâches** : l'algorithme doit être amené à résoudre plusieurs fois le même sous-problème.
- **Sous-structure optimale** : un problème dispose d'une sous-structure optimale lorsque les solutions des sous-problèmes qui composent la solution optimale sont elles-mêmes optimales. Pour le problème du sac à dos, si par exemple l'objet numéro N se trouve dans la solution optimale, alors en enlevant cet objet du sac on doit obtenir une solution optimale au sous-problème avec les objets de 1 à $N-1$ et le poids maximal $W_{max} - w_N$.

3.4 Programmation dynamique par mémoïsation

```python
def memoisation(v: list, w: list, W_max: int):
    D = {}

    def V(i, W):
        if (i, W) in D:
            return D[(i, W)]
        elif i == 0 or W == 0:
            r = 0
        elif w[i-1] <= W:
            r = max(v[i-1] + V(i-1, W - w[i-1]), V(i-1, W))
        else:
            r = V(i-1, W)
        D[(i, W)] = r
        return r

    return V(len(v), W_max)
```

3.5 Programmation dynamique par calcul de bas en haut

Ici, il faut commencer par réfléchir aux dépendances des différents $V(i,W)$: pour calculer $V(i,W)$, il est nécessaire de connaître $V(i-1,W-w_i)$ et $V(i-1,W)$. Comme w_i peut prendre n'importe quelle valeur selon le problème, on va devoir calculer $V(i-1,0), V(i-1,1), \ldots, V(i-1,W)$ avant de pouvoir calculer $V(i,W)$.

```
def bottom_up(v: list, w: list, W_max: int):
    N = len(v)
    D = {}

    for W in range(W_max + 1):
        D[(0, W)] = 0

    for i in range(1, N + 1):
        D[(i, 0)] = 0
        for W in range(1, W_max + 1): # L'ordre des deux boucles est important !
            if w[i-1] > W:
                D[(i, W)] = D[(i-1, W)]
            else:
                D[(i, W)] = max(v[i-1] + D[(i-1, W-w[i-1])], D[(i-1, W)])

    return D[(N, W_max)]
```

3.6 Récupérer une liste optimale

Il est facile ensuite de récupérer une liste optimale des objets à mettre dans le sac à dos : pour cela, il suffit de modifier la fonction précédente pour qu'elle renvoie le dictionnaire complet au lieu de la valeur de celui-ci sur le couple (N, W_{max}).
Ensuite, la fonction suivante reconstitue la liste des objets à emporter :

```
def sac_a_dos(v: list, w: list, W_max: int):
    D = bottom_up(v, w, W_max)
    sac = []
    N = len(v)
    W = W_max

    while N > 0:
        if D[(N, W)] > D[(N-1, W)]:
            sac.append((v[N-1], w[N-1]))
            W = W - w[N-1]
        N = N - 1

    return sac
```

3.7 Programmation dynamique et algorithmes gloutons

La programmation dynamique garantit d'obtenir la meilleure solution au problème étudié, mais dans un certain nombre de cas sa complexité temporelle reste trop importante pour pouvoir être utilisée dans la pratique.

Dans ce type de situation, on se résout à utiliser un autre paradigme de programmation, la **programmation gloutonne**. Alors que la programmation dynamique se caractérise par la résolution par taille croissante de tous les problèmes locaux, la stratégie gloutonne consiste à choisir à partir du problème global un problème local et un seul en suivant une *heuristique* (c'est à dire une stratégie permettant de faire un choix rapide mais pas nécessairement optimal). On ne peut en général garantir que la stratégie gloutonne détermine la solution optimale, mais lorsque l'heuristique est bien choisie on peut espérer obtenir une solution proche de celle-ci.

Pour le problème du sac à dos, il faut définir une heuristique, c'est-à-dire une méthode arbitraire (mais réfléchie) pour évaluer quel objet est prioritaire par rapport à un autre : par exemple, nous pouvons choisir en priorité les objets dont le rapport $\frac{v_i}{w_i}$ est maximal et remplir le sac autant que possible :

```python
def glouton(v, w, W_max):
    W = W_max
    priorites = [(v[i] / w[i], i) for i in range(len(v))]
    priorites.sort(reverse=True)

    V, i = 0, 0
    while W >= 0 and i < len(v):
        objet = priorites[i][1]
        if w[objet] <= W:
            W = W - w[objet]
            V = V + v[objet]
        i = i + 1

    return V
```

Comparaison des résultats

En exécutant les deux fonctions précédentes sur différents jeux de données, on constate que dans environ 40% des cas, l'algorithme glouton donne la même solution que l'algorithme par programmation dynamique (qui donne toujours la solution optimale).

Lorsque l'algorithme glouton ne donne pas la solution optimale, celui-ci donne une solution au moins égale à 98% de la solution optimale.