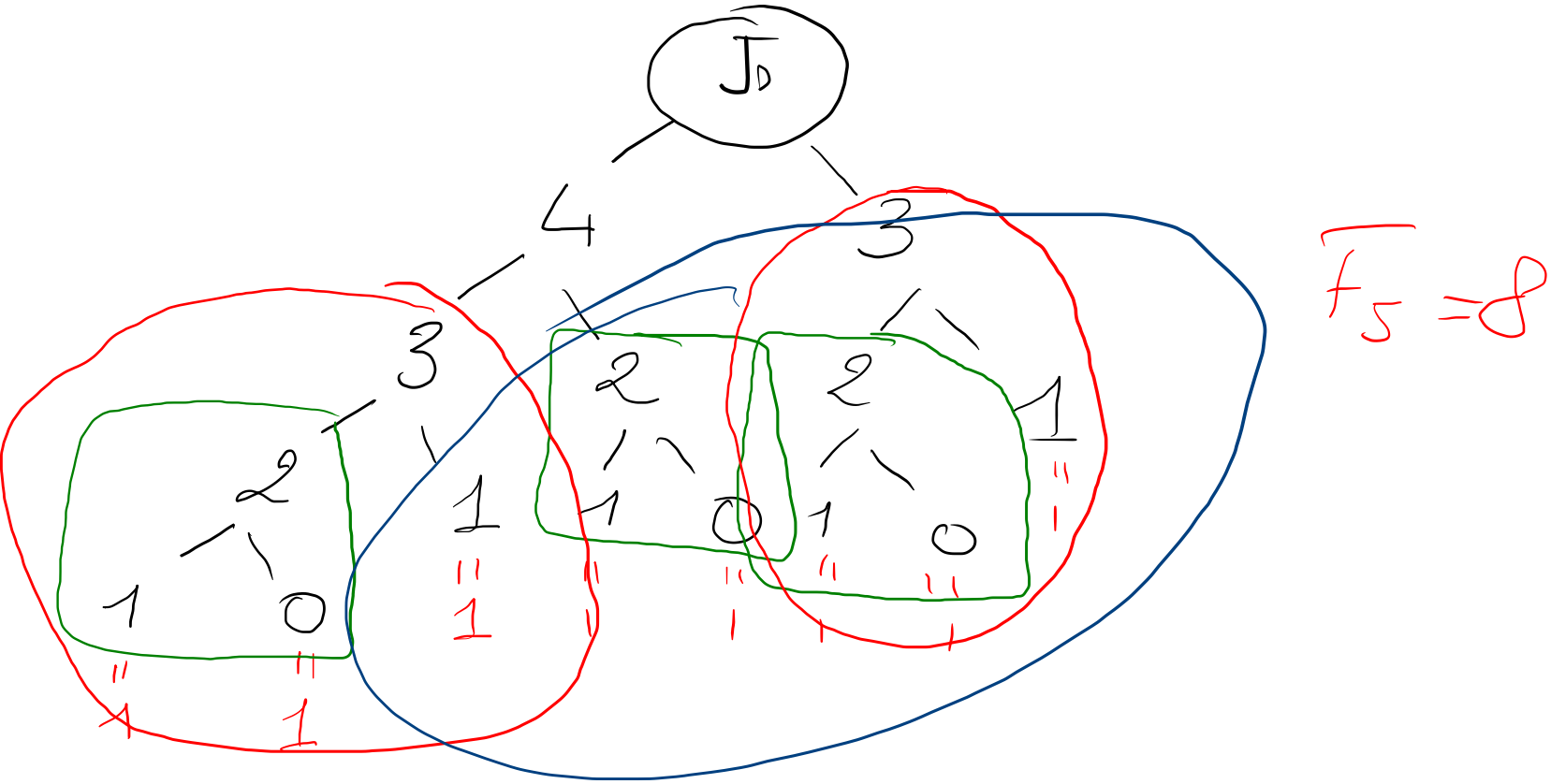
Programmation dynamique

Mardi 23 Juin

Pourquoi la solution récursive est mauvaise ?

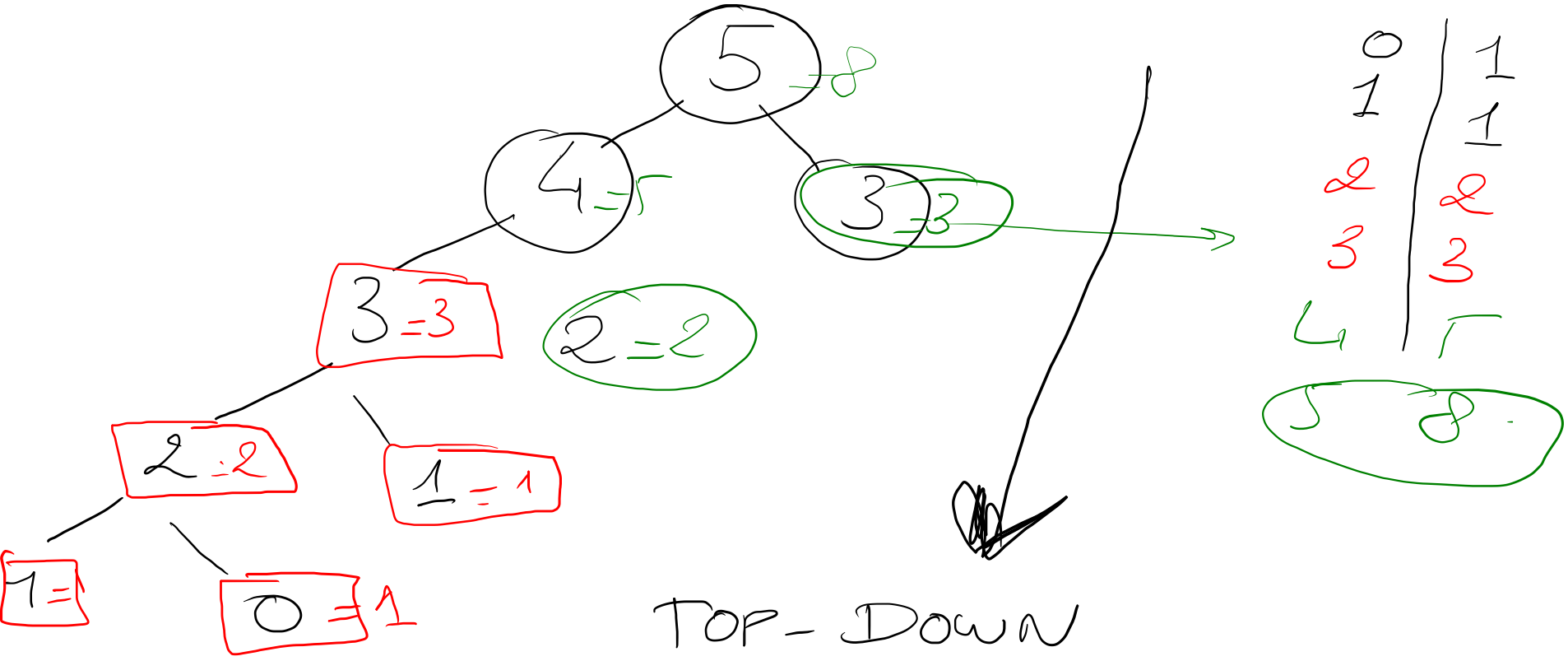
```
def fibo2(n) :
    if n < 2 :
        return 1
    else :
        return fibo2(n-1)+fibo2(n-2)
```



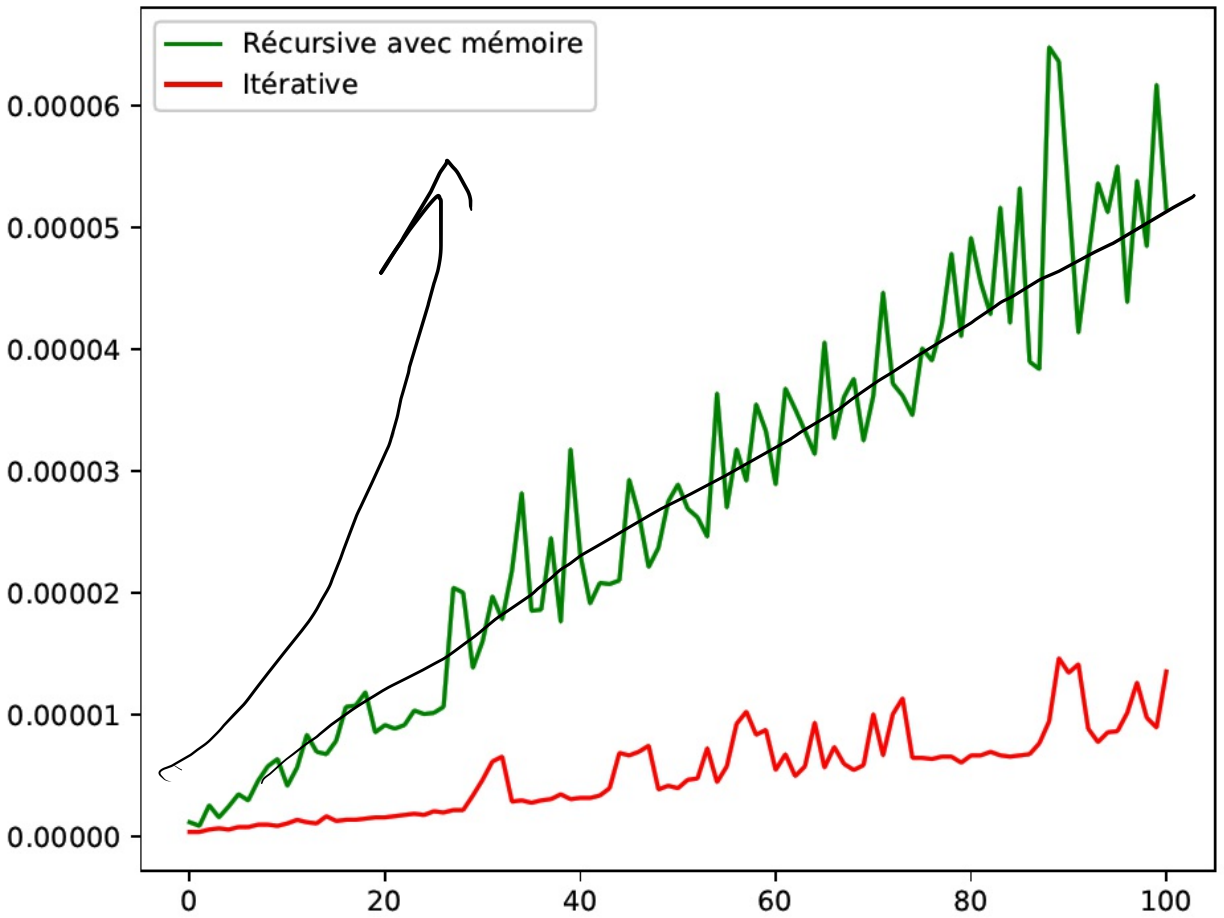
$\overline{f_5} = 8$

Une première alternative est de conserver l'idée de programmation récursive, mais de mémoriser les résultats déjà calculés :

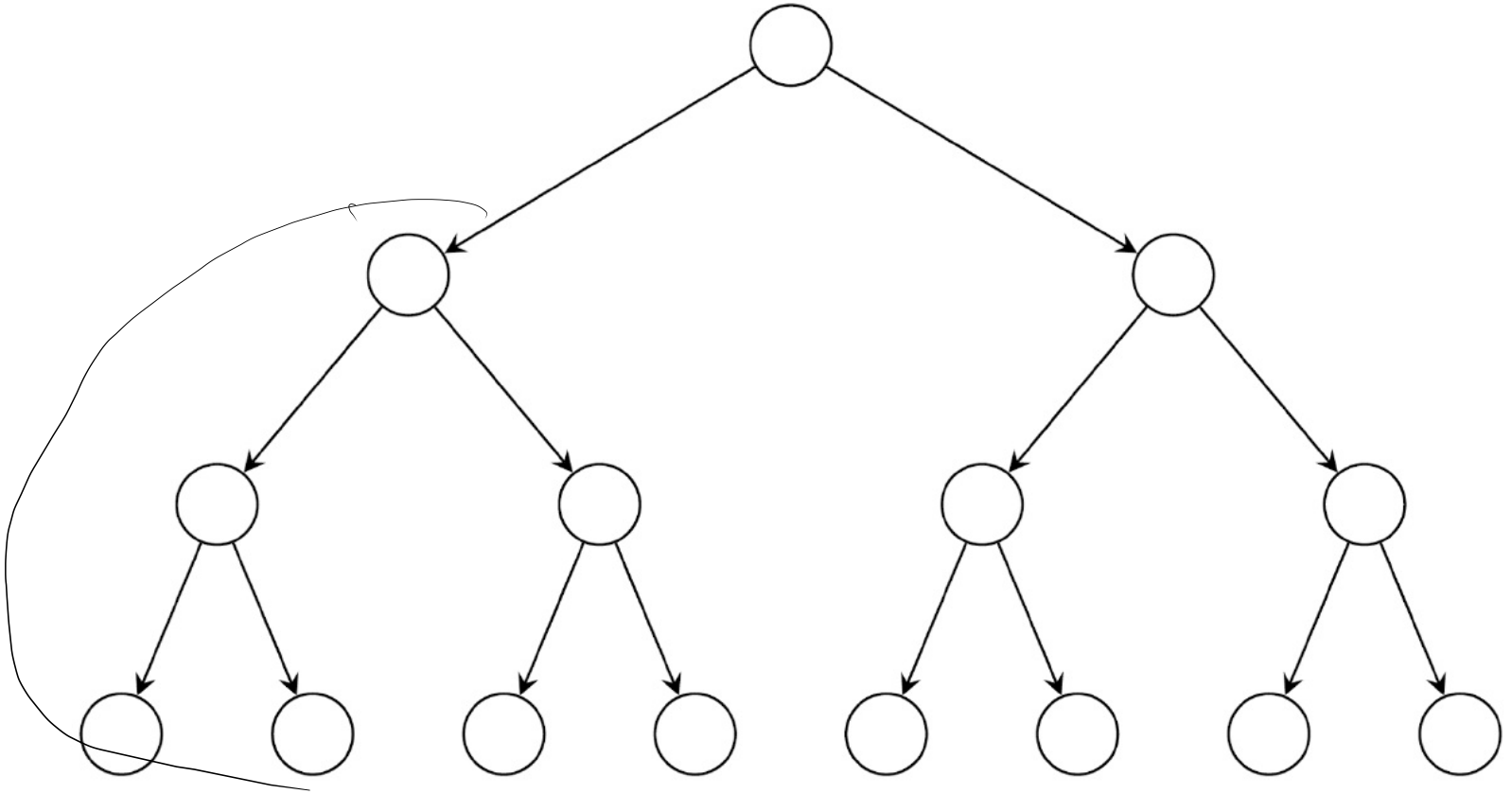
```
def fibo(n) :
    valeurs = {0 : 1, 1 : 1}
    def fibo_aux(n) :
        if n in valeurs :
            return valeurs[n]
        else :
            v = fibo_aux(n-1)+fibo_aux(n-2)
            valeurs[n] = v
            return v
    return fibo_aux(n)
```



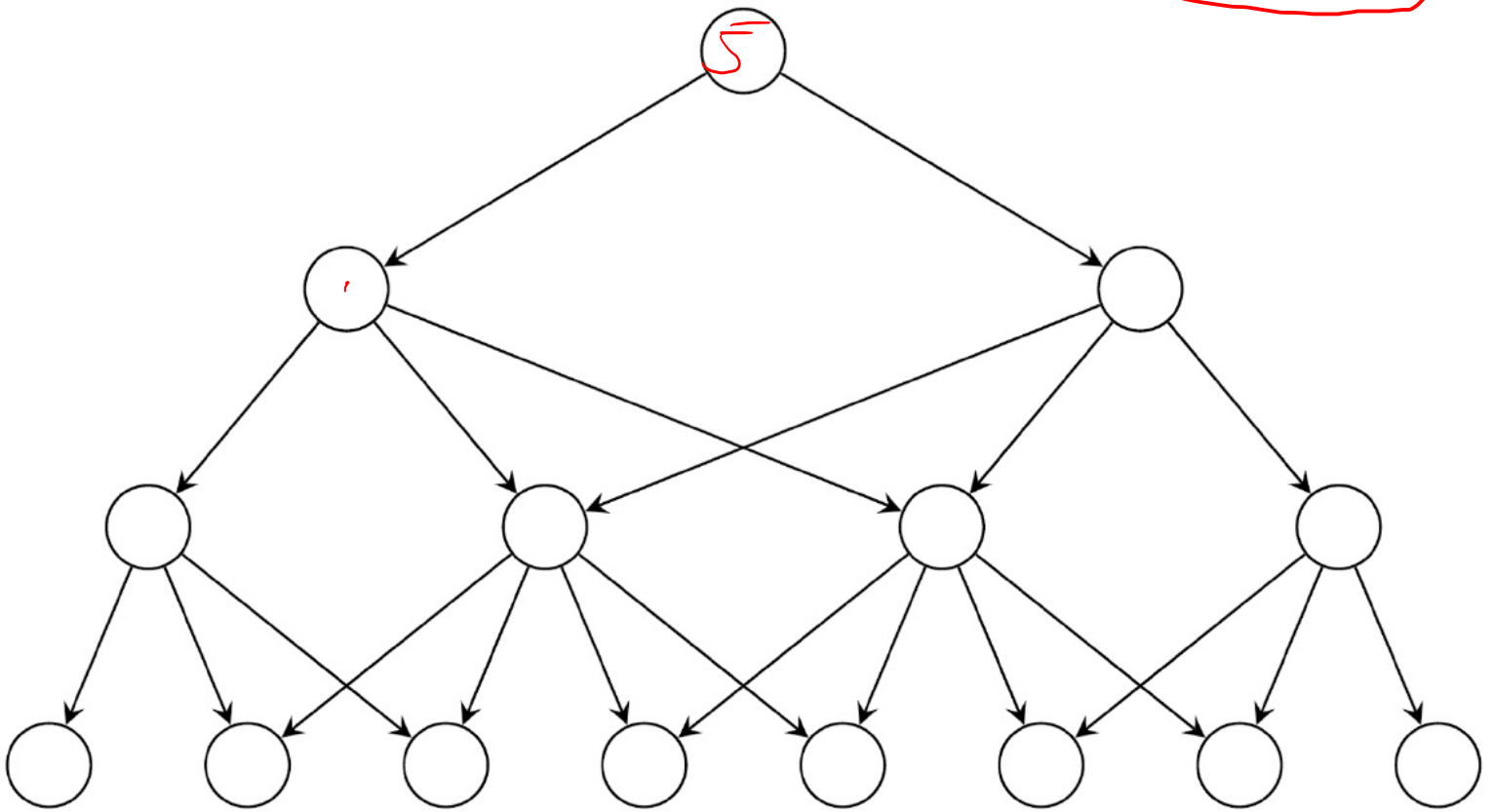
TOP - DOWN


Récursive avec mémoire
Itérative
0.00006
0.00005
0.00004
0.00003
0.00002
0.00001
0.00000
0
20
40
60
80
100

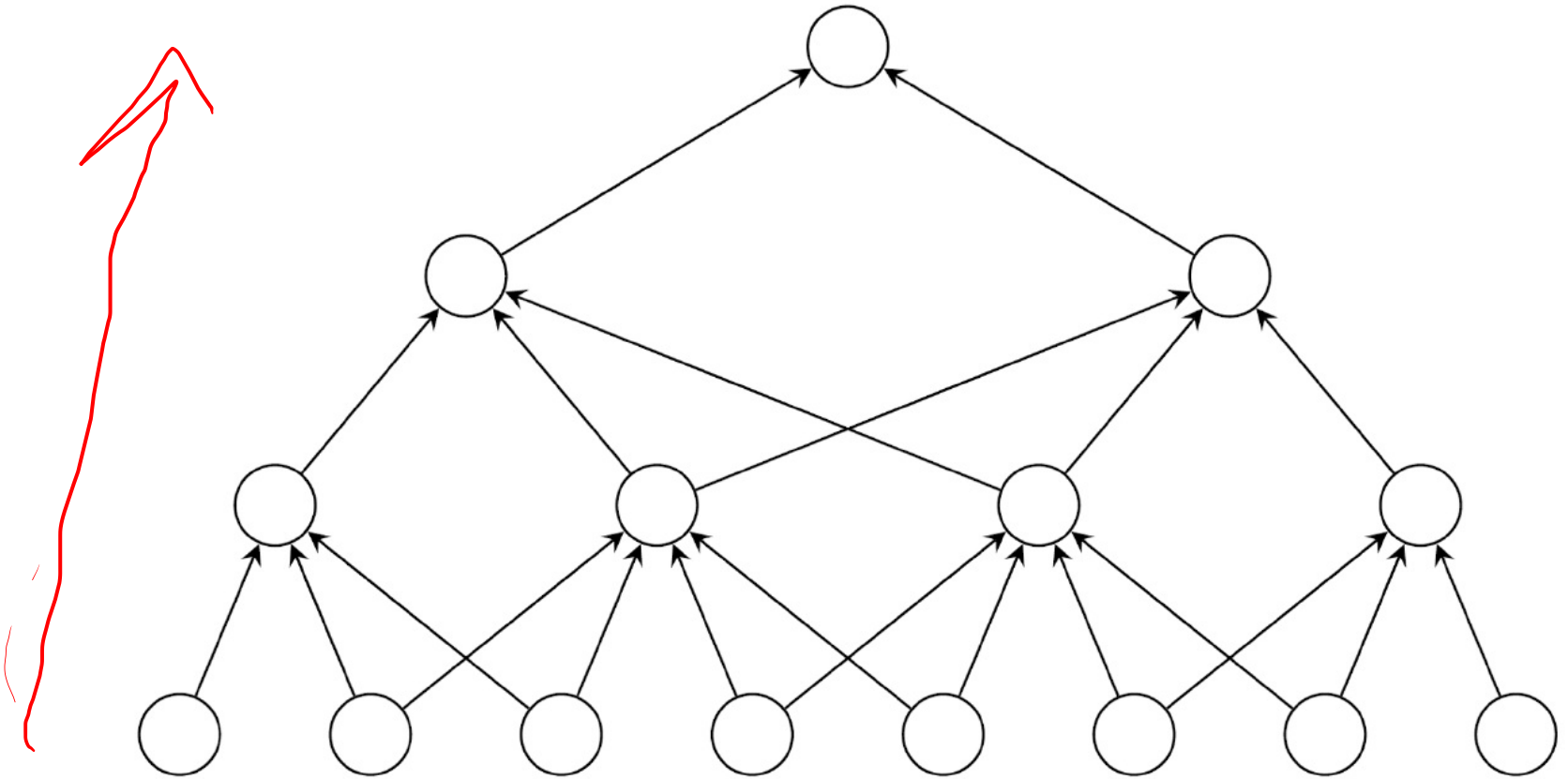
Situation avec des sous-problèmes indépendants
Exemple : **tri par partition-fusion**.

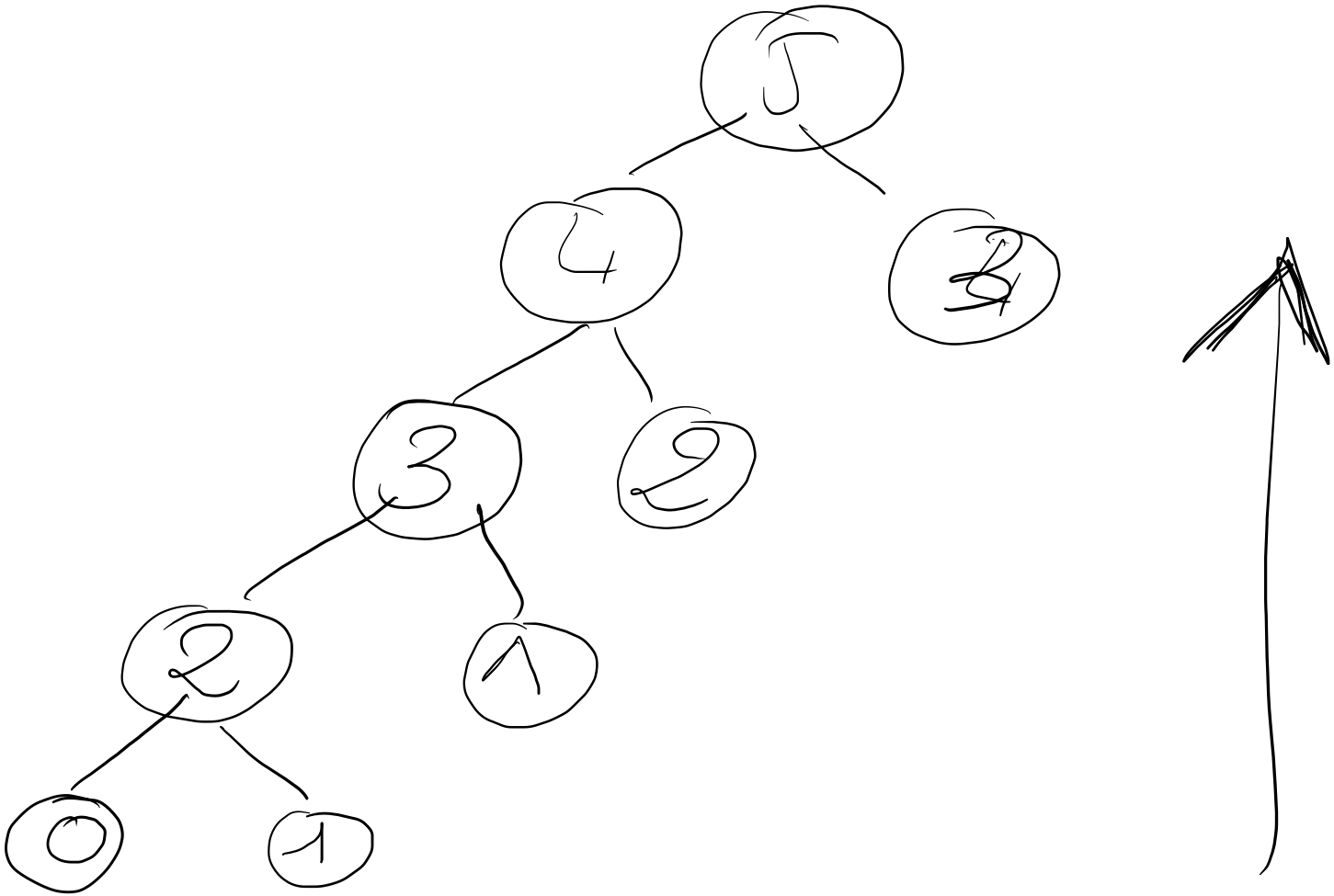


Situation avec des sous-problèmes dépendants
Solution **récursive Top-Down** avec **mémoïsation**.



Situation avec des sous-problèmes dépendants
Solution **dynamique Down-Top**.




5
4
3
3
2
2
1
0
1

Vocabulaire

La **programmation dynamique** est un paradigme de conception itératif adapté aux fonctions récursives qui permet d'améliorer leur complexité, lorsque les sous-problèmes sont **dépendants**.

Vocabulaire

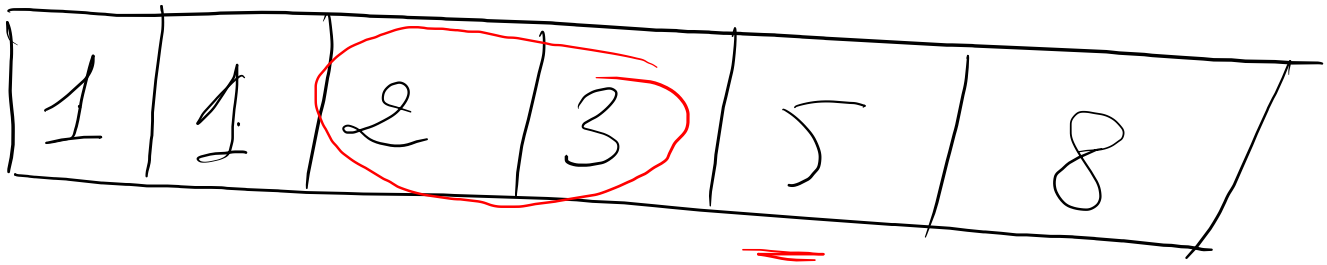
La **programmation dynamique** est un paradigme de conception itératif adapté aux fonctions récursives qui permet d'améliorer leur complexité, lorsque les sous-problèmes sont **dépendants**.

Ce concept a été introduit par Bellman, dans les années 50, pour résoudre typiquement des problèmes d'optimisation. De nos jours, l'application de cette méthode ne se limite plus à ce type de problèmes.

- Comme dans le cas d'un algorithme récursif, la programmation dynamique s'applique lorsque la solution d'un problème peut s'exprimer en fonction des solutions de sous-problèmes.

- Comme dans le cas d'un algorithme récursif, la programmation dynamique s'applique lorsque la solution d'un problème peut s'exprimer en fonction des solutions de sous-problèmes.
- Au lieu de passer par une programmation récursive, la programmation dynamique est itérative.

- Comme dans le cas d'un algorithme récursif, la programmation dynamique s'applique lorsque la solution d'un problème peut s'exprimer en fonction des solutions de sous-problèmes.
- Au lieu de passer par une programmation récursive, la programmation dynamique est itérative.
- La programmation dynamique évite, contrairement aux programmes récursifs, la répétition éventuelle de calculs identiques. Il en résulte un gain significatif en terme de complexité temporelle.

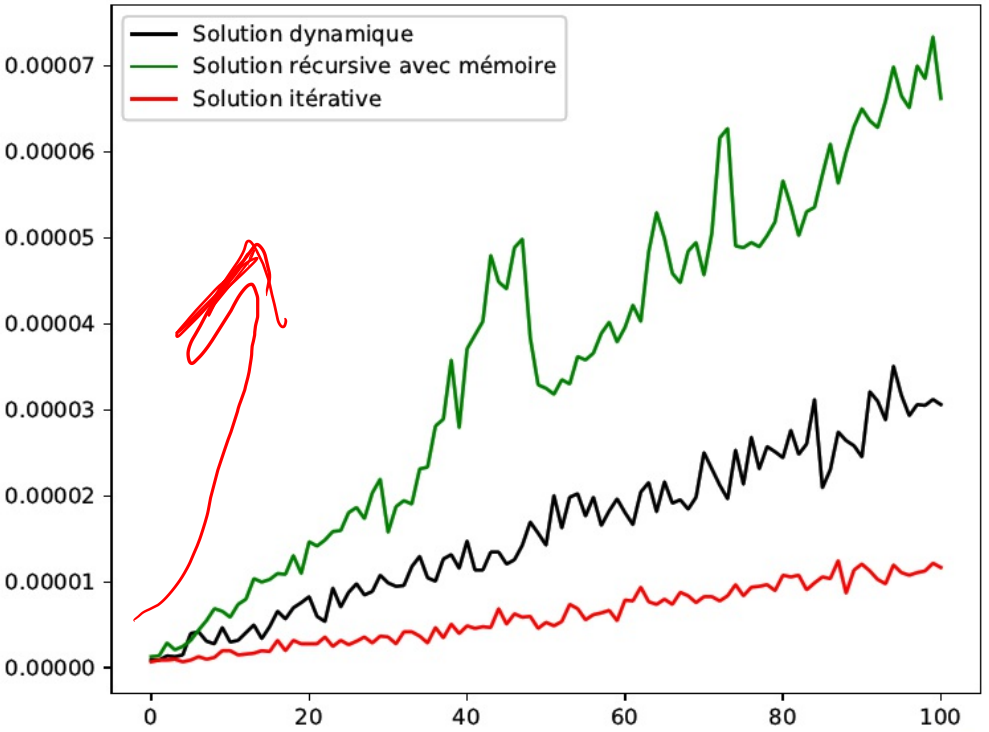

1
1
2
3
5
8

- Les calculs dans un programme récursif se font de haut en bas, tandis qu'en programmation dynamique ils s'effectuent de bas en haut

- Les calculs dans un programme récursif se font de haut en bas, tandis qu'en programmation dynamique ils s'effectuent de bas en haut
- On commence par résoudre les plus petits sous-problèmes. En combinant leurs solutions, on obtient les solutions des sous-problèmes de plus en plus grands.

- Les calculs dans un programme récursif se font de haut en bas, tandis qu'en programmation dynamique ils s'effectuent de bas en haut
- On commence par résoudre les plus petits sous-problèmes. En combinant leurs solutions, on obtient les solutions des sous-problèmes de plus en plus grands.
- Cela est rendu possible en sauvegardant tous les résultats obtenus pour les sous-problèmes dans un tableau à une ou plusieurs dimensions. C'est le principe de **mémoïsation**. On gagne en complexité temporelle, mais on perd en complexité spatiale.

```python
def fibo_dyn(n) :
    valeurs = [1]*(n+1)
    for i in range(2, n+1) :
        valeurs[i] = valeurs[i-1] + valeurs[i-2]
    return valeurs[n]
```



C'est un exemple qui ne présente **aucun autre intérêt** que celui de comprendre le principe puisque :

- Le problème au rang n ne dépend que des 2 problèmes précédents (pas d'intérêt de conserver en mémoire tous les résultats précédents).
- Il existe une solution de calcul en temps constant : F_n est une suite linéaire récurrente d'ordre 2, on peut donc exprimer F_n en fonction de n tout simplement...

Voyons quelques exemples plus pertinents...

Problème du rendu de monnaie.

Dans cette partie, on cherche à réaliser un programme de rendu de monnaie. On souhaite que l'automate rende l'appoint de manière optimale, dans le sens où il minimise le nombre de pièces rendues (ou billets).

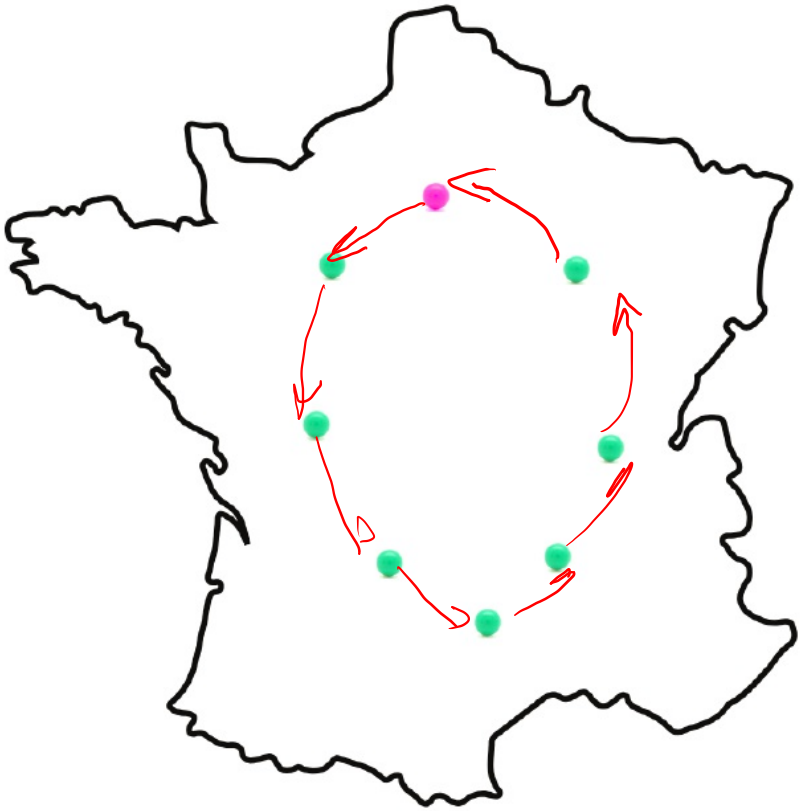
Dans cette partie, on cherche à réaliser un programme de rendu de monnaie. On souhaite que l'automate rende l'appoint de manière optimale, dans le sens où il minimise le nombre de pièces rendues (ou billets).

On se place pour le moment avec le système européen

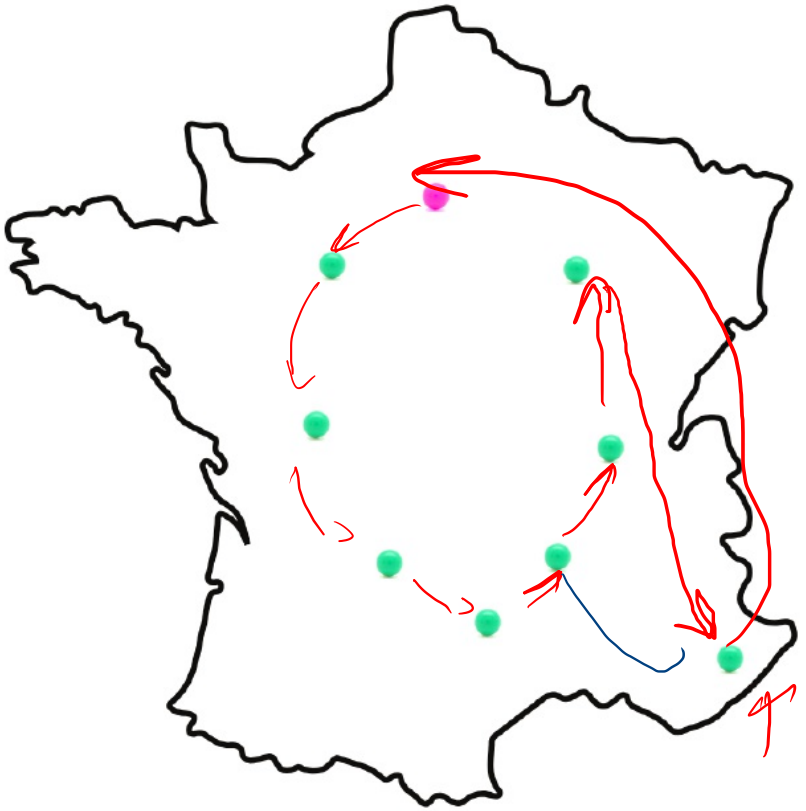
$$\mathcal{P} = \{1; 2; 5; 10; 20; 50; 100; 200\}$$

Un algorithme **glouton** est un algorithme localement optimal. Il ne donne pas toujours le meilleur résultat, mais « parfois » un résultat satisfaisant.

Un algorithme **glouton** est un algorithme localement optimal. Il ne donne pas toujours le meilleur résultat, mais « parfois » un résultat satisfaisant.



Un algorithme **glouton** est un algorithme localement optimal. Il ne donne pas toujours le meilleur résultat, mais « parfois » un résultat satisfaisant.



Pourquoi la solution gloutone utilisée par les commerçants est optimale ?

Pourquoi la solution gloutone utilisée par les commerçants est optimale ?

- On peut toujours rendre la monnaie avec cette technique puisqu'il y a des pièces de 1.

Pourquoi la solution gloutone utilisée par les commerçants est optimale ?

- On peut toujours rendre la monnaie avec cette technique puisqu'il y a des pièces de 1.
- Le système de monnaie $\mathcal{P} = \{1; 2; 5; 10; 20; 50; 100; 200\}$ est super croissant (c'est à dire que chaque élément est strictement supérieur à la somme des valeurs précédentes).

Pourquoi la solution gloutone utilisée par les commerçants est optimale ?

- On peut toujours rendre la monnaie avec cette technique puisqu'il y a des pièces de 1.
- Le système de monnaie $\mathcal{P} = \{1; 2; 5; 10; 20; 50; 100; 200\}$ est super croissant (c'est à dire que chaque élément est strictement supérieur à la somme des valeurs précédentes).

Remarque : le problème a toujours une solution si et seulement si $1 \in \mathcal{P}$

- le système britannique d'avant 1971 utilisait les multiples suivants du penny : $\mathcal{P} = \{1; 3; 6; 12; 24; \mathbf{30}\}$....

$48 = 30 + 12 + 6 \longrightarrow 3p$

$48 = 24 + 24 \longrightarrow 2p$

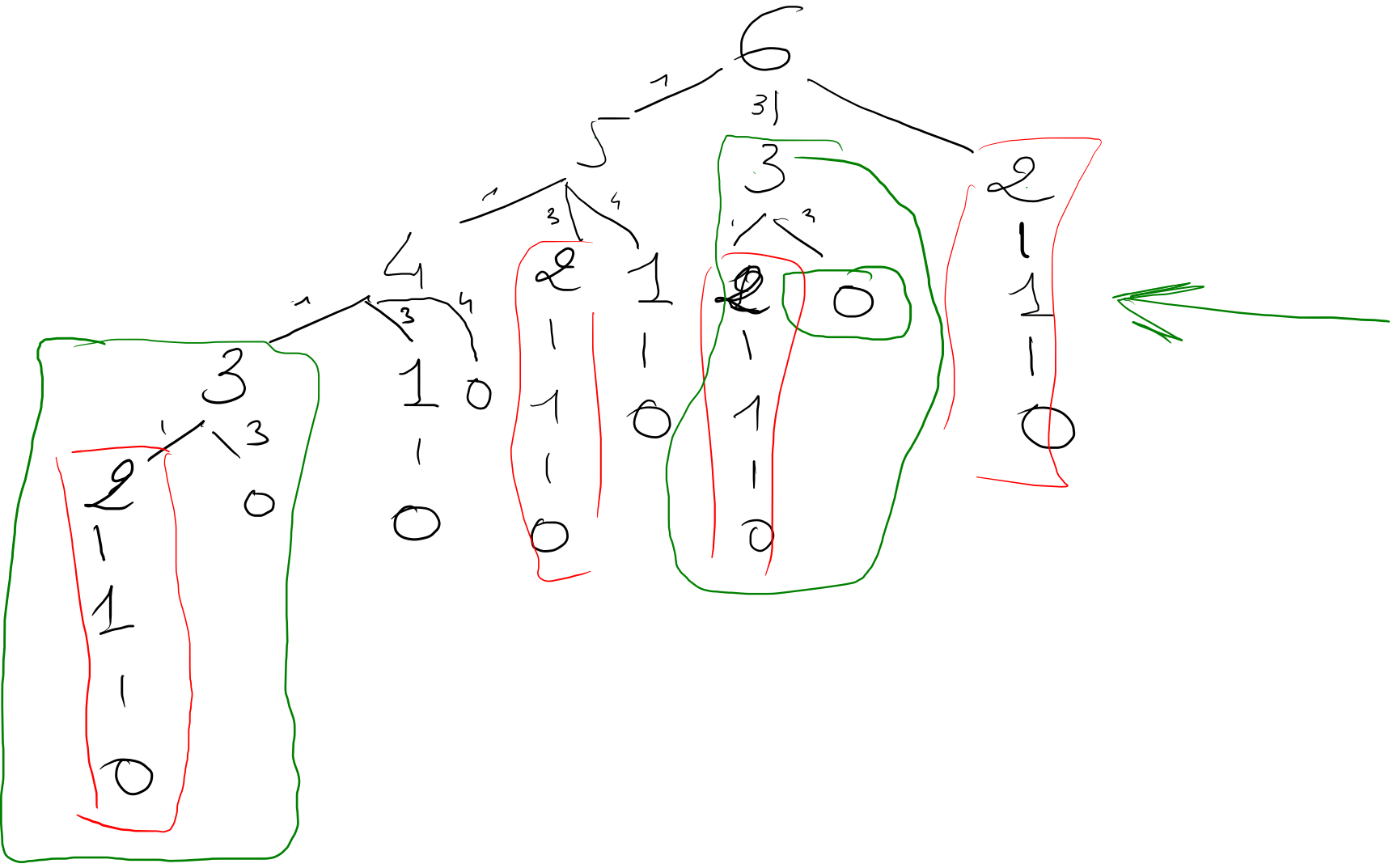
- le système britannique d'avant 1971 utilisait les multiples suivants du penny : $\mathcal{P} = \{1; 3; 6; 12; 24; \mathbf{30}\}$....
- Avec ce système, l'algorithme glouton décompose
  - 48 pennies en : 48 = 30 + 12 + 6
  - alors que la décomposition optimale est : 48 = 24 + 24.

- le système britannique d'avant 1971 utilisait les multiples suivants du penny : $\mathcal{P} = \{1; 3; 6; 12; 24; \mathbf{30}\}$....

- Avec ce système, l'algorithme glouton décompose
  - 48 pennies en : 48 = 30 + 12 + 6
  - alors que la décomposition optimale est : 48 = 24 + 24.

Une nouvelle stratégie s'impose...

$P = \{1, 3, 4\}$ 6



Si on note $N(s)$ le nombre minimal de pièces nécessaires pour obtenir la somme s :

Si on note $N(s)$ le nombre minimal de pièces nécessaires pour obtenir la somme s :

- $N(0) = 0$

Si on note $N(s)$ le nombre minimal de pièces nécessaires pour obtenir la somme s :

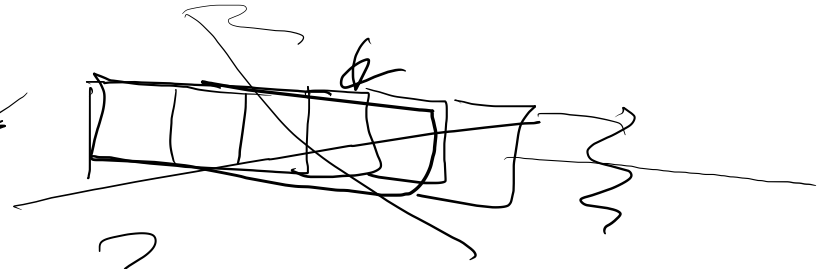
- $N(0) = 0$
- $N(s) = 1 + \min_{p \in \mathcal{P} \mid p \leqslant s} N(s - p)$

Si on note $N(s)$ le nombre minimal de pièces nécessaires pour obtenir la somme s :

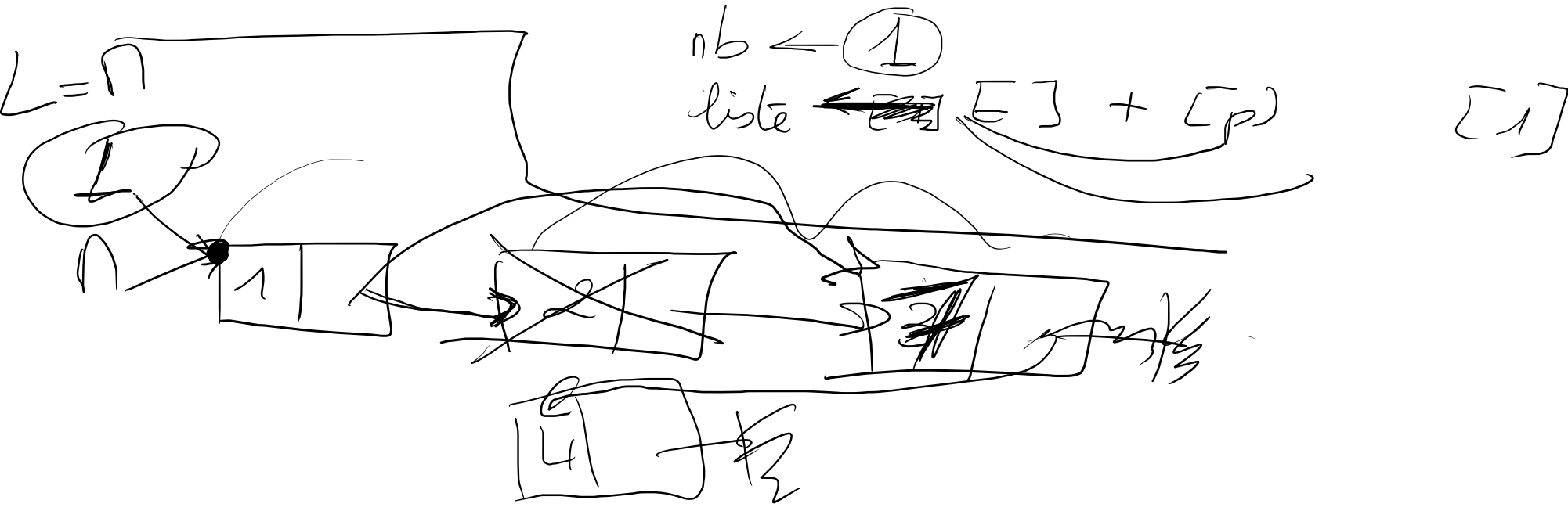
- $N(0) = 0$
- $N(s) = 1 + \min_{p \in \mathcal{P} \mid p \leqslant s} N(s-p)$

1. Écrire cette fonction récursive.
2. L'améliorer pour obtenir en plus la répartition de pièces.
3. Proposer une solution Top-Down avec mémoïsation.

N(0) = 0, [ ]



N(1) ⟶ 3



6

1, 3, 4

- p_0, p_1, p_{k-1} les éléments de $\mathcal{P}$ toujours classés par ordre croissant.
- $N_{m,s}$ le nombre minimal de pièces pour obtenir la somme s, mais en n'utilisant que des pièces du sous-ensemble $\{p_0, p_1, \ldots, p_m\}$

on cherche : $N(\text{somme}) = N_{k-1,\ \text{somme}}$

$N(2) = 2$

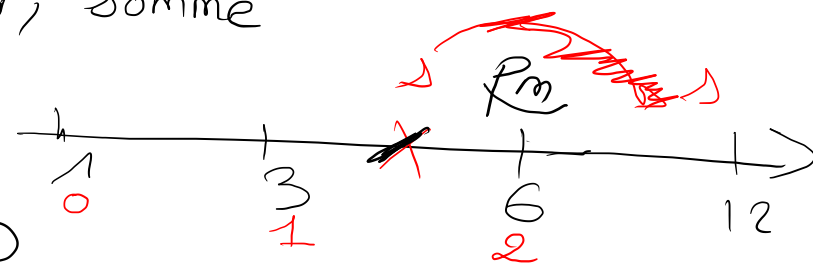
$N(3) = 1$

$N(4) = 1$

$N(5) = 2$

$N(6) = 3$

$s < p_2$

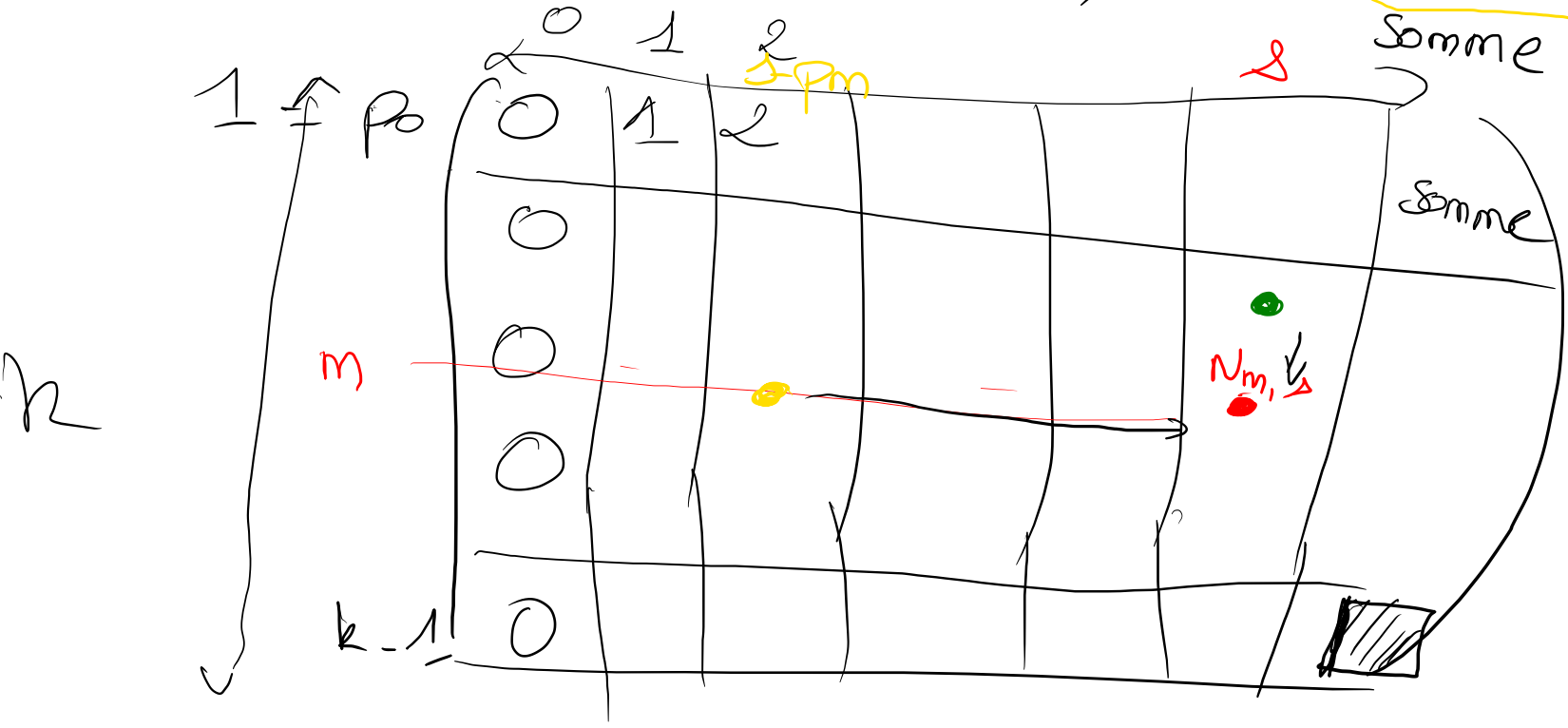


$$N_{m,s} = \begin{cases} 0 & \text{si } s = 0 \\ N_{m-1,s} & s < p_m \\ \min\left(N_{m-1,s}\ ;\ 1 + N_{m,s-p_m}\right) & \text{prends } p_m \end{cases}$$

4 → 5

$1+1+1+1+1 = 6$

$$N_{m,s} = \begin{cases} 0 & \text{si } s = 0 \\ N_{m-1,s} & \text{si } s < p_m \\ \min_{y=0} \left(N_{m-1,s} \; ; \; 1 + N_{m,s-p_m} \right) \end{cases}$$



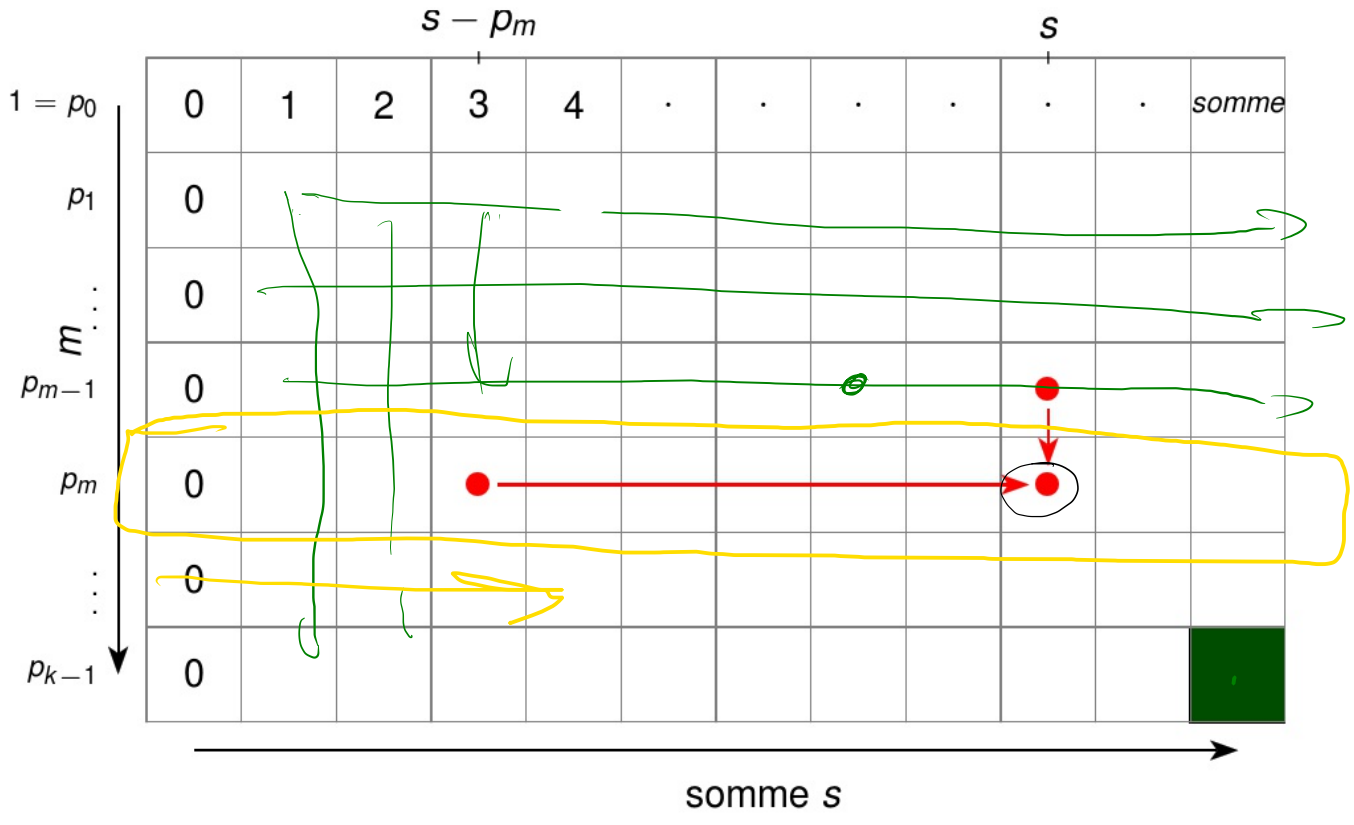
Notons,

- p_0, p_1, p_{k-1} les éléments de $\mathcal{P}$ toujours classés par ordre croissant.
- $N_{m,s}$ le nombre minimal de pièces pour obtenir la somme s, mais en n'utilisant que des pièces du sous-ensemble $\{p_0, p_1, ..., p_m\}$

On cherche $N(s) = N_{k-1,somme}$ où k est le nombre de pièces différentes et *somme* la somme souhaitée. On a alors :

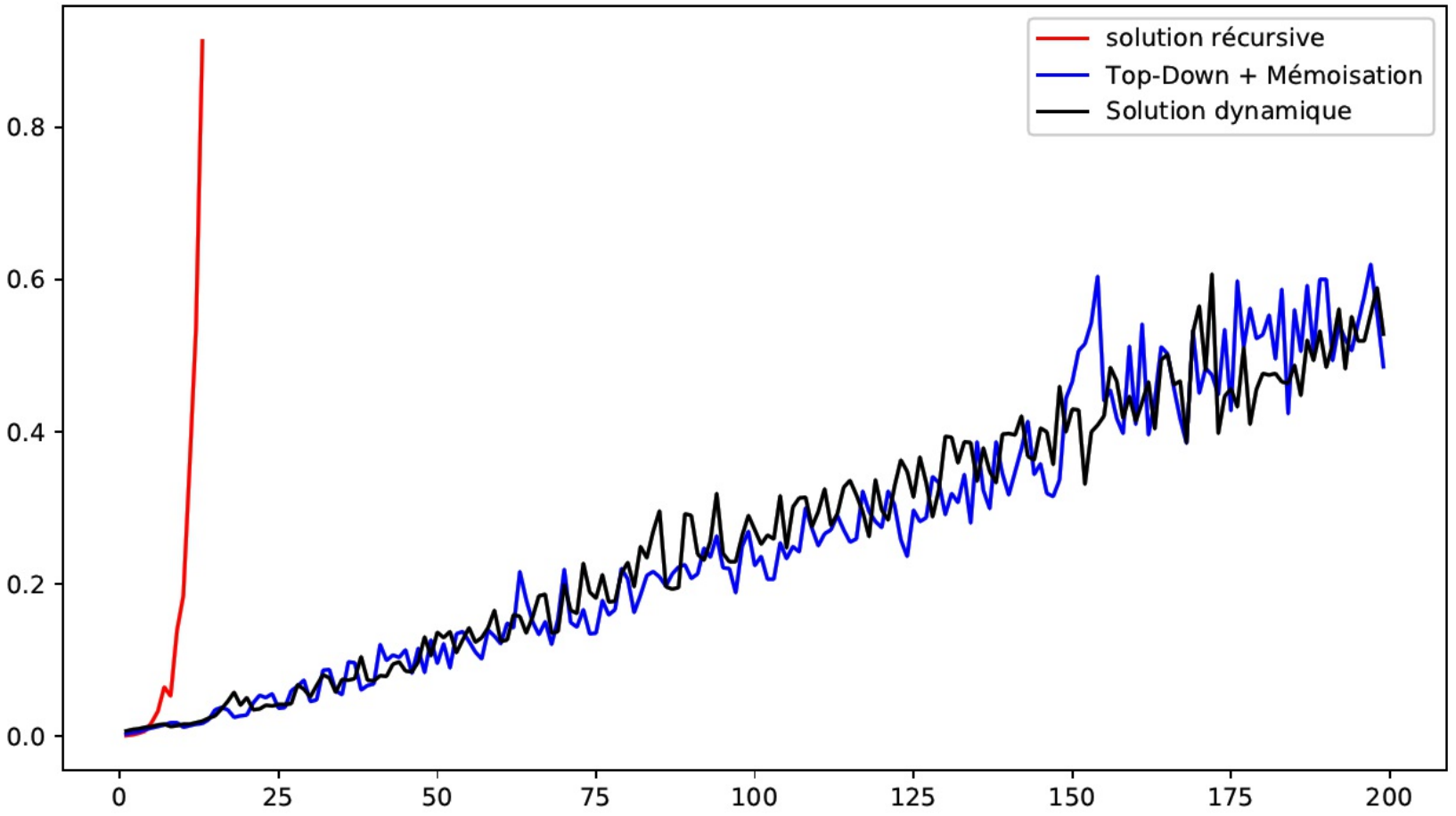
$$N_{m,s} = \begin{cases} 0 \text{ si } s = 0 \\ N_{m-1,s} \text{ si } s < p_m \\ \min \left\{ \underbrace{N_{m-1,s}}_{\text{On ne prend pas la pièce } p_m} ; \underbrace{1 + N_{m,s-p_m}}_{\text{On prend la pièce } p_m} \right\} \end{cases}$$

diagramme des dépendances : $N_{m,s} = \begin{cases} 0 \text{ si } s = 0 \\ N_{m-1,s} \text{ si } s < p_m \\ \min\{N_{m-1,s} \; ; \; 1 + N_{m,s-p_m}\} \end{cases}$



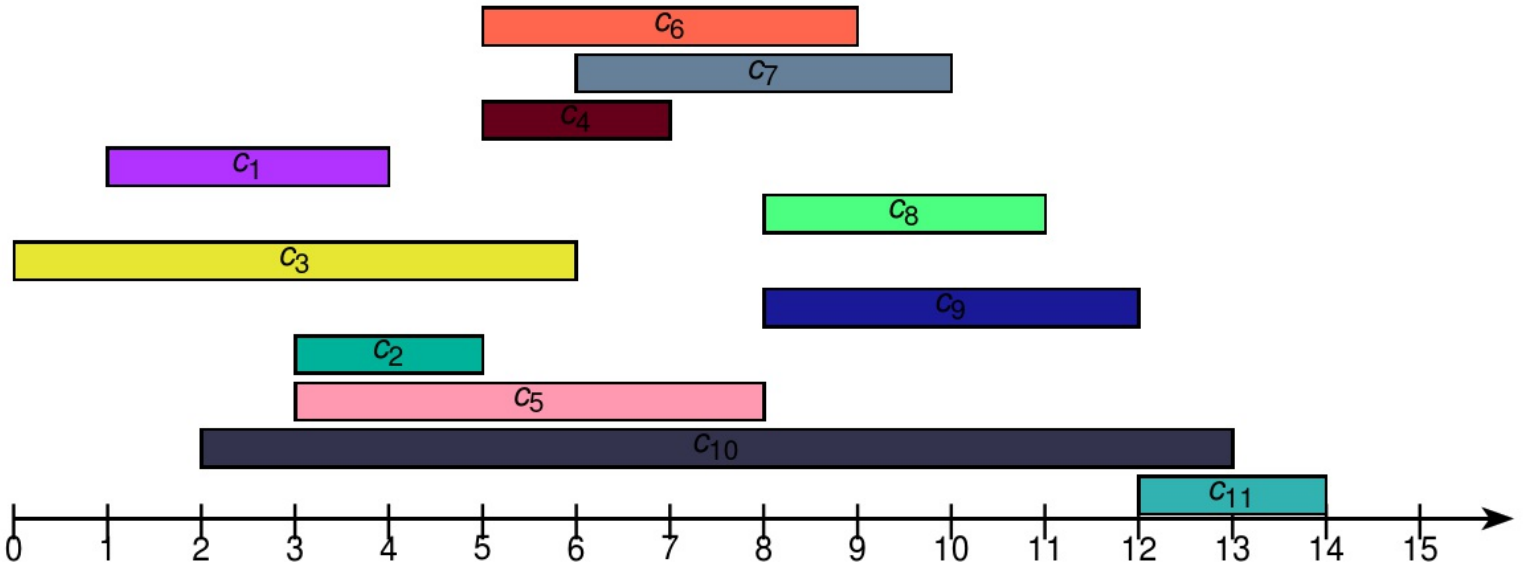
On peut gagner en complexité spatiale. Quelle est la complexité temporelle de ce nouvel algo ?

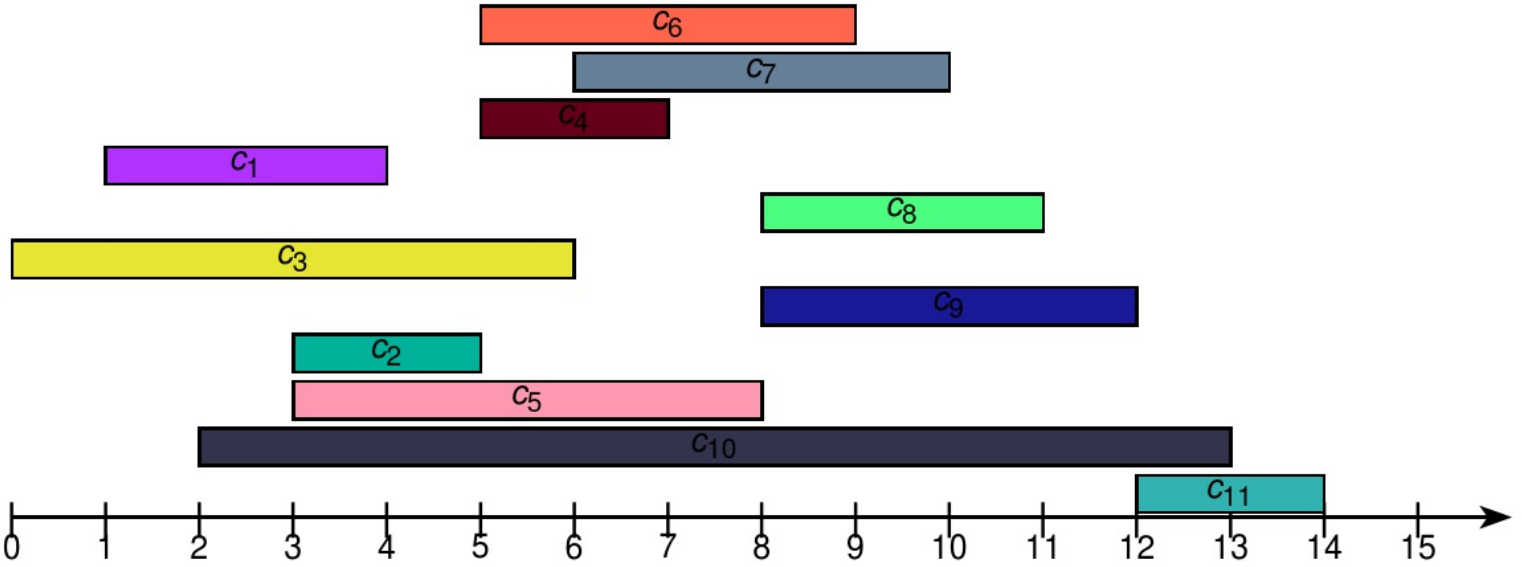
On peut gagner en complexité spatiale. Quelle est la complexité temporelle de ce nouvel algo ?



Comment choisir ses concerts pour en voir un maximum ?

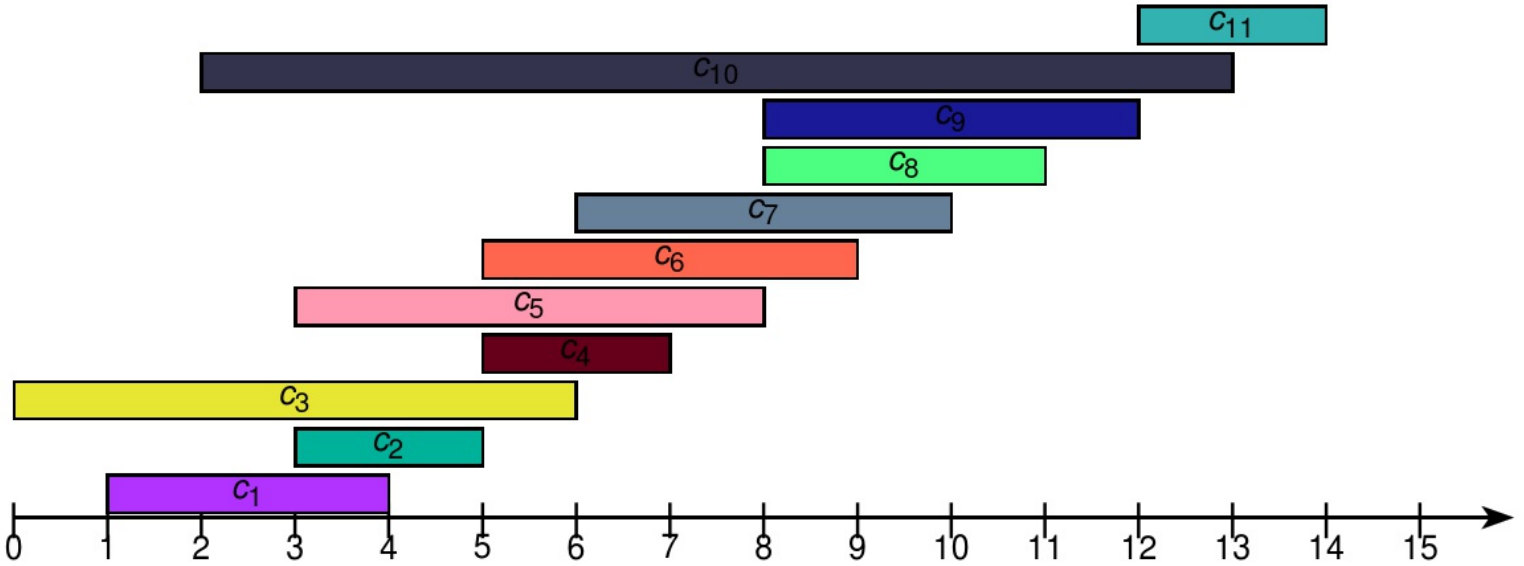
Lors d'un festival, n concerts c_1, c_2, ... c_n sont organisés. Chaque concert c_i est défini par son heure de début d_i et de fin f_i. On convient qu'il est possible d'assister aux concerts c_i et c_j dès lors que $[d_j; f_i[\cap [d_j; f_j[= \emptyset$, c'est à dire que les intervalles ne se chevauchent pas.





1. Comment traduire que l'on peut assister à c_i et c_j ?
2. Comment savoir si on peut voir c_1, c_8 et c_5 ?

Et ainsi ? On voit mieux ? !



1. Comment traduire que l'on peut assister à c_i et c_j ?
2. Comment savoir si on peut voir c_1, c_8 et c_5 ?

Python propose un module `itertools` qui permet de générer facilement des **itérateurs** pour donner une liste exhaustive des cas répondant à certaines situations.

- Permutations (tous les mélanges possibles)

```
import itertools

E = ['A', 'B', 'C']
for cas in itertools.permutations(E) :
    print (cas,end=' ; ')
```

Affiche :

```
('A', 'B', 'C') ; ('A', 'C', 'B') ; ('B', 'A', 'C') ;
('B', 'C', 'A') ; ('C', 'A', 'B') ; ('C', 'B', 'A') ;
```

- *p*-listes : (tirages avec remise)

```
import itertools

E = [1, 2, 3, 4, 5]
for cas in itertools.product(E,repeat=2) :
    print (cas,end=' ; ')
```

Affiche :

```
(1, 1) ; (1, 2) ; (1, 3) ; (1, 4) ; (1, 5) ; (2, 1) ; (2, 2) ;
(2, 3) ; (2, 4) ; (2, 5) ; (3, 1) ; (3, 2) ; (3, 3) ; (3, 4) ;
(3, 5) ; (4, 1) ; (4, 2) ; (4, 3) ; (4, 4) ; (4, 5) ; (5, 1) ;
(5, 2) ; (5, 3) ; (5, 4) ; (5, 5) ;
```

- Arrangements : (tirage sans remise)

```
import itertools

E = [1, 2, 3, 4]
for cas in itertools.permutations(E,3) :
    print (cas,end=' ; ')
```

Affiche :

```
(1, 2, 3) ; (1, 2, 4) ; (1, 3, 2) ; (1, 3, 4) ; (1, 4, 2) ; (1, 4, 3)
(2, 1, 3) ; (2, 1, 4) ; (2, 3, 1) ; (2, 3, 4) ; (2, 4, 1) ; (2, 4, 3)
(3, 1, 2) ; (3, 1, 4) ; (3, 2, 1) ; (3, 2, 4) ; (3, 4, 1) ; (3, 4, 2)
(4, 1, 2) ; (4, 1, 3) ; (4, 2, 1) ; (4, 2, 3) ; (4, 3, 1) ; (4, 3, 2)
```

- Combinaisons : (tirages simultanés, ordre non pris en compte)

```
import itertools

E = [1, 2, 3, 4, 5]
for cas in itertools.combinations(E,3) :
    print (cas,end=' ; ')
```

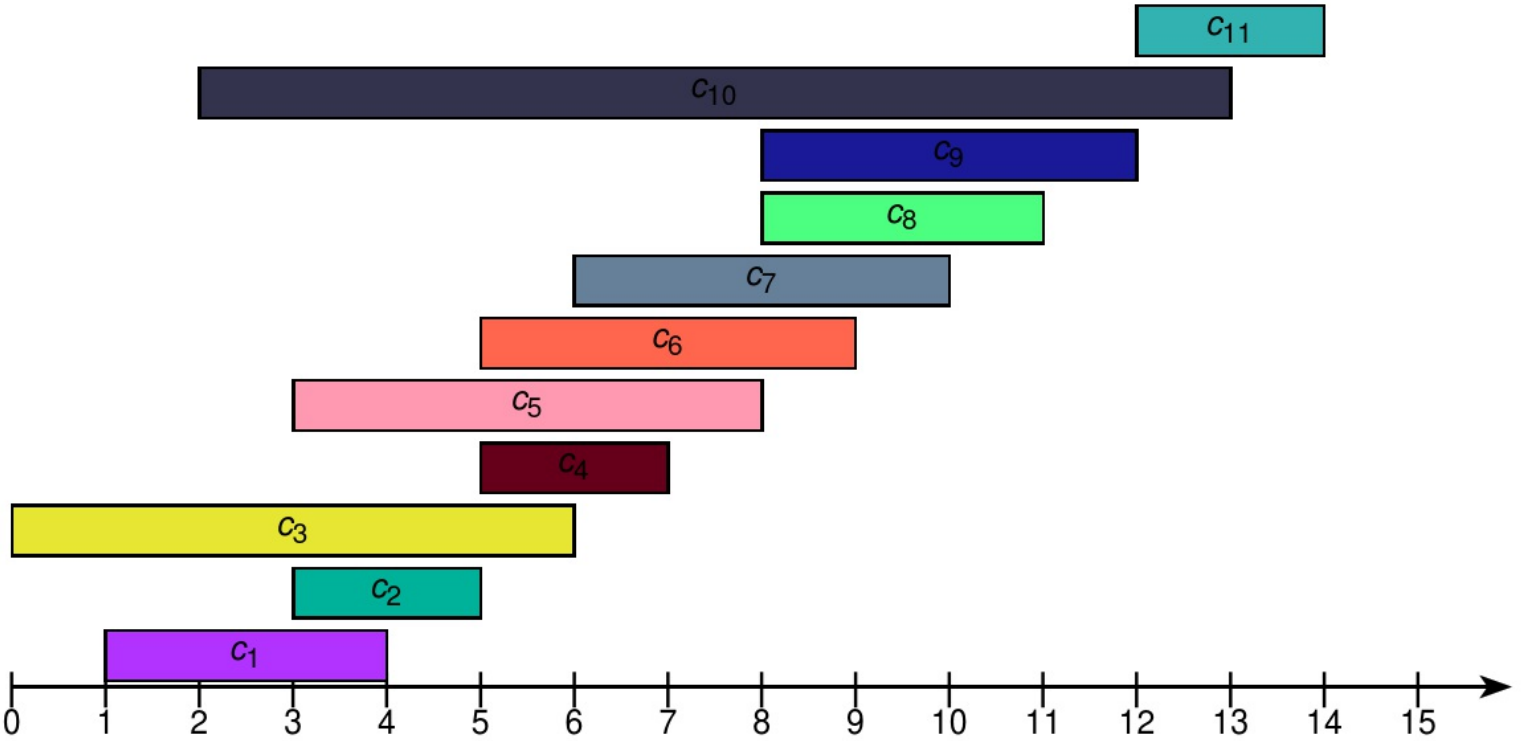
Affiche :

```
(1, 2, 3) ; (1, 2, 4) ; (1, 2, 5) ; (1, 3, 4) ; (1, 3, 5) ;
(1, 4, 5) ; (2, 3, 4) ; (2, 3, 5) ; (2, 4, 5) ; (3, 4, 5) ;
```

A partir de ces informations, on peut se lancer dans un algorithme **brute force** pour répondre à la question. Que dire de sa complexité ?

- On suppose comme précédemment que les concerts sont rangés par ordre croissant d'heure de fin : $f_0 \leqslant f_1 \leqslant .... \leqslant f_n$.
- On note C_{ij} l'ensemble des concerts se déroulant (intégralement) entre les heures f_i et d_j.
- Enfin, on note m_{ij} le nombre maximal de concerts auxquels on peut assister parmi ceux de C_{ij}
- Pour simplifier les problèmes aux bords, on ajoute deux concerts virtuels :
  - c_0 qui commence et finit au temps $-\infty$;
  - c_{n+1} au temps $+\infty$

On cherche donc $m_{0,n+1}$



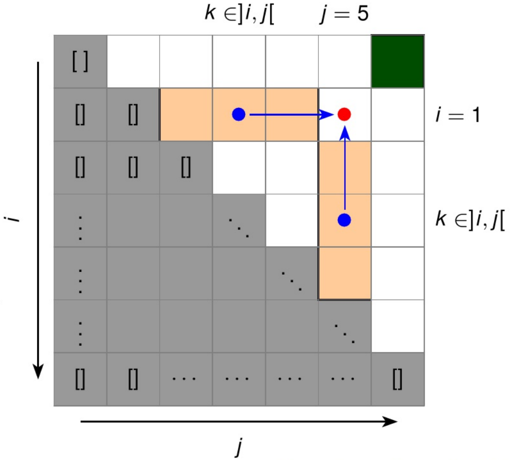
1. Que vaut S_{ij} lorsque $i \geqslant j$?
2. Si $i < j$, exprimer m_{ij} en fonction des m_{ik} et m_{kj} $(i < k < j)$

$$m_{ij} = \begin{cases} 0 \text{ si } C_{ij} = \emptyset \\ \\ \max\limits_{c_k \in C_{ij}} \left\{ m_{ik} + 1 + m_{kj} \right\} \end{cases}$$

Nous allons donc créer un tableau A de taille $(n+2) \times (n+2)$ destiné à contenir des listes de concerts de C_{ij} tels que $|A_{ij}| = m_{ij}$.

Diagramme des dépendances

$$m_{ij} = \begin{cases} 0 \text{ si } C_{ij} = \emptyset \\ \max\limits_{c_k \in C_{ij}} \{ m_{ik} + 1 + m_{kj} \} \end{cases}$$



A vous de jouer !

Sources :

https://www.supinfo.com/cours/2ADS/chapitres/05-programmation-dynamique.

Algorithmique - Thomas H. Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein.

Cours d'informatique commune et option de CPGE de Jean-Pierre Becirspahic (Lycée Louis Le Grand).

https://omnilogie.fr/O

Merci à mes sponsors !