TNSI Algorithme Glouton vs Programmation Dynamique

1. Algorithme glouton rendu de pièces

Un **algorithme glouton** (*greedy algorithm* en anglais, parfois appelé aussi algorithme gourmand, ou goulu) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local, dans l'espoir d'obtenir un résultat optimum global.

De manière pratique :

1. On prend la plus grande valeur disponible immédiatement inférieure à celle de l'ensemble de départ pour la retrancher à l'ensemble de départ
2. On recommence avec la valeur immédiatement inférieure jusqu'à ce que la somme des valeurs retranchées corresponde à celle de l'ensemble de départ

Exemple le rendu de pièces on désire globalement un nombre de pièces minimal.

- Dans le système de pièces européen (en centimes : 1, 2, 5, 10, 20, 50, 100, 200), on doit rendre 37 centimes
    - alors on effectue le choix d'une pièce de 20 centimes que l'on retranche à 37 il reste 17
    - alors on effectue le choix d'une pièce de 10 centimes que l'on retranche à 17 il reste 7
    - alors on effectue le choix d'une pièce de 5 centimes que l'on retranche à 7 il reste 2
    - alors on effectue le choix d'une pièce de 2 centimes que l'on retranche à 2 il reste 0
    - on peut montrer que l'algorithme glouton donne toujours une solution optimale.

- Dans le système de pièces (1, 3, 4), l'algorithme glouton n'est pas optimal, comme le montre l'exemple simple suivant. Il donne pour 6 : 4+1+1, alors que 3+3 est optimal.

- L'algorithme glouton ne peut pas revenir en arrière ici dès qu'il a trouvé 4 ce résultat est gardé pour rechercher les autres valeurs de pièces

2. Paradigmes algorithmiques

Les algorithmes peuvent être catalogués en fonction de leur principe de fonctionnement :

- *Algorithme glouton* : construit une solution de manière incrémentale, en optimisant un critère de manière locale. Cf plus haut

- *Diviser pour régner* : divise un problème en sous-problèmes indépendants (qui ne se chevauchent pas), résout chaque sous-problème, et combine les solutions des sous-problèmes pour former une solution du problème initial. Cf recherche dichotomique

- *Programmation dynamique* : divise un problème en sous-problèmes qui sont non indépendants (qui se chevauchent), et cherche (et stocke) des solutions de sous-problèmes de plus en plus grands

TNSI Algorithme Glouton vs Programmation Dynamique

➢ *Bref historique :*

Richard Bellman

Programmation dynamique : paradigme développé par Richard Bellman en 1953 chez RAND Corporation.

● « Programmation » = planification

● Technique de conception d'algorithme très générale et performante.

● Permet de résoudre de nombreux problèmes d'optimisation.

3. Programmation dynamique : exemple académique – vocabulaire

L'exemple le plus souvent utilisé est celui de la suite de Fibonacci :

Leonardo da Pisa, dit Fibonacci

Soit Fn = nombre de lapins au mois n

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

On retrouve les nombres de Fibonacci : 1,1,2,3,5,8 ,13,34,55,89 etc …

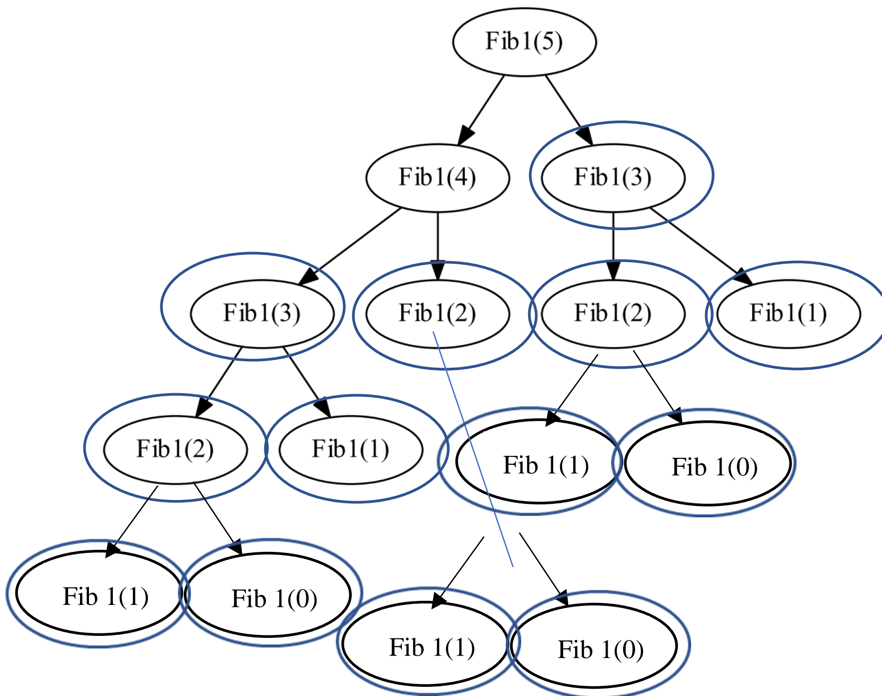
Ils augmentent très rapidement : $F_{30} > 10^6$

Un algorithme récursif la formule s'y prête parfaitement mais pas efficace

```
fonction Fib1(n)
    si n = 0 retourner 0
    si n =1 retourner 1
    retourner Fib1(n-1) + Fib1(n-2)
```

3.1 A faire

- Ecrire un programme **fibonacci_recursif.py** qui calcule la valeur de la suite pour le rang n
- Vérifier les résultats https://fr.wikipedia.org/wiki/Suite_de_Fibonacci
- Faites afficher les appels récursifs pour n = 5 (voir page suivante)



On remarque que l'on calcule plusieurs fois les mêmes valeurs

Un algorithme qui garde en mémoire les résultats intermédiaires en remplissant un tableau

On utilise une liste (ou un tableau suivant le langage) mem[n]

fonction Fib_mem(n)

remplir un tableau de taille n mem sans valeur

Si n< 2 alors

mem[n] = n

Sinon

calculer toutes les valeurs de mem de 2 à n

retourner mem[n]

3.2 A faire

- Implémenter cet algorithme

Un algorithme récursif qui garde en mémoire les différentes valeurs intermédiaires dans un tableau

fonction Fib_dynamique(n) : (permet d'initialiser un tableau)

réserve un tableau sans valeur de taille n

renvoie la valeur pour n de la fonction récursive Fib

fonction Fib(n,tableau) : (fait intervenir la récursion)

si tableau[n] est défini alors retourner tableau[n]

sinon tableau[n] = formule de récursion avec Fib

retourner valeur dans le tableau

Ici Fib(n,tableau) induit des appels récursifs seulement la première fois qu'elle est appelée. Les valeurs intermédiaires sont stockées dans le tableau.

Remarque importante la fonction Fib renvoie sa valeur qui est repris dans le calcul sans cela on ne peut pas utiliser la récursion.

3.3 A faire Compléter le code

- La méthode consiste à utiliser une fonction qui initialise la liste des valeurs à vide

```
def Fib_dynamique(n):
    *****************************
    return  Fib(n, memo)
```

- Et une fonction qui exécute la récursion pour remplir cette liste en utilisant les résultats déjà connus

```
def Fib(n,tableau) :
    if tableau[n]:
        ***************************
    elif ******************************
        tableau[n] = n
    else:
        tableau[n] =  ************************
        print(Fib(n, tableau))
    return tableau[n]
```

- Tester la fonction Fib_dynamique

Cette technique porte un nom :

Mémoïser = conserver à la fin de l'exécution d'une fonction le résultat associé aux arguments d'appels, pour ne pas avoir à recalculer ce résultat lors d'un autre appel récursif.

Idée : mémoïser et réutiliser les solutions de sous-problèmes qui aident à résoudre le problème.

Complexité temporelle :

nombre de sous-problèmes

x

complexité par sous-problème*

* : on ne compte pas les appels récursifs.

On parle de de programmation dynamique de forme Bottom up quand on part des problèmes plus petits pour aller à ceux des tailles les plus grandes et inversement de Bottom Down quand on part des problèmes les plus grands pour arriver aux plus petits (cf Fibonacci).

De manière générale la programmation dynamique nécessite de pouvoir

Diviser pour régner : c'est-à-dire trouver une relation de récurrence

Mémoïser les résultats intermédiaires pour les réutiliser et ne pas les recalculer

4. Programmation dynamique rendu de pièces

Trouvons une relation de récurrence :

On suppose que nous avons à disposition une liste de pièce P_i avec i compris entre 1 et n (n étant le nombre total de pièces différentes)

Si nous sommes capables de rendre la somme S avec un nombre de pièces nb(S) alors quelle somme est-on capable de rendre avec 1+nb(S) ?

Exemple : si je suis capable de rendre 52 cts et que j'ai à ma disposition des pièces de 2 cts, 5 cts, 10 cts, 50 cts et 1 euro, je peux aussi rendre :

52 - 2 = 50 cts

52 - 5 = 47 cts

52 - 10 = 42 cts

52 - 50 = 2 cts

Je ne peux pas utiliser de pièce de 1 euro.

Si nb(S-p_i) est le nombre minimal de pièces à rendre pour le montant S-p_i alors **nb(S) = 1+nb(S-p_i)**

Ex nb(52) = 1 +nb(52-50) soit une pièce de 50 + après une de 2

Soit

Si S = 0 alors nb(S) = 0

Si S > 0 alors nb(S) = 1+min(nb(S - p_i)) avec $1 \leq i < n$ et pi $\leq$ S

Le min correspond à la plus petite combinaison de pièces possible pour rendre la somme S-p_i

A faire

- Compléter le programme de rendu_monnai-recurence.py qui utilise cette fonction de récurrence

```
import math
def rendu_monnaie_recurrence(pieces, s_a_rendre):
    """renvoie le nombre minimal de pièces pour faire
       la somme s avec le système pieces"""
    if s_a_rendre == 0: return 0 # Somme à rendre nulle
    else : # Nb n'a pas été déjà calculer pour ce rendu : on reprend la
formule de recurrence
        mini = float('inf') # Le nombre minimal est mis à la valeur max
        # On explore tous les cas
        for p in pieces:
            #Si il est possible d'utiliser une pièce
            if ***********************:
                #Relation de récurrence
                nb =  **********************************
                #mémorisation du nouveau minimum
                if nb < mini :
                    mini = nb
    # rendu du nombre minimum de pièces
    return mini
```

Essayer avec les listes suivantes pour 11 centimes puis 15, 17 puis 37 à rendre

pieces_euros = [1,2,5,10,20,50,100,200]
pieces_234 = [2,3,4]
pieces_134 = [1,3,4]

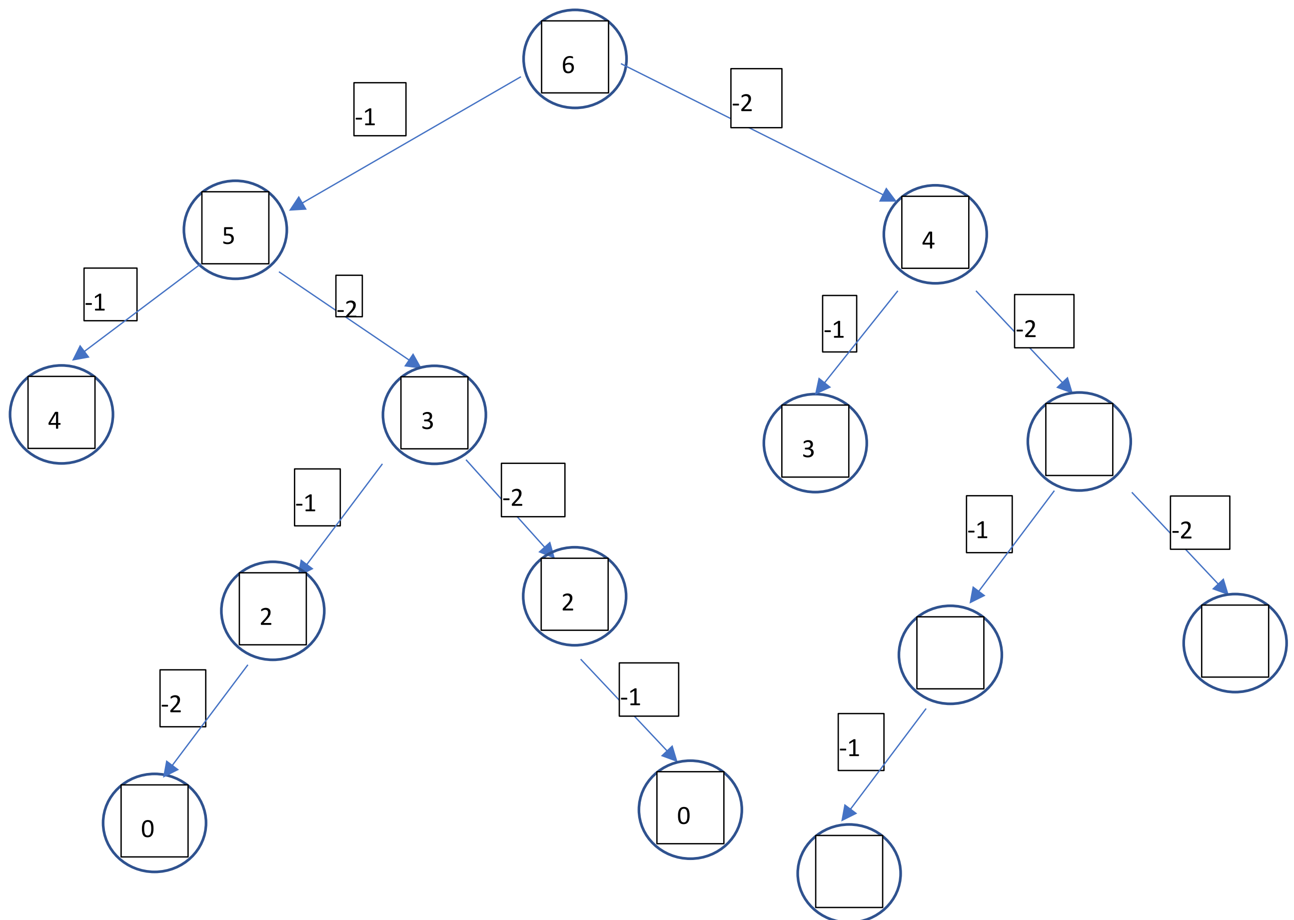
Vos conclusions :

Programmation dynamique :

Pour limiter le nombre de calculs on va mémoïser les résultats optimums pour toutes les valeurs de 1 à la somme à rendre (quand la somme à rendre est nulle on rend 0 pièces)

Compléter l'arbre de rendu pour une somme de 6 est un système de pièces [1,2]

Faites apparaitre les combinaisons minimales de pièces pour chaque somme et sous sommes



TNSI Algorithme Glouton vs Programmation Dynamique

On remarque que certains rendus sont recalculés plusieurs fois. On va stocker ces résultats dans un tableau de la manière suivante

On trouve une solution optimale pour chaque somme d'argent de 0 à s.

En effet, si on connait les façons optimales de rendre toute somme d'argent strictement inférieure à s, alors pour rendre une somme s, on rend une pièce, à choisir dans la liste monnaie, et une somme strictement inférieure à s, ce qui correspond à un problème déjà résolu. Ainsi, on trouve la solution optimale avec une somme s en comparant les n solutions pour chaque pièce de la liste monnaie.

On remplit le tableau par colonne. On peut noter directement les solutions dans le tableau.

Compléter le tableau pour l'exemple précédent

Piéces/Somme	0	1	2	3	4	5	6
1							
2							
Total							

On note pour la programmation :

- n le nombre de pièces de monnaie différentes
- pièces un tableau/liste de taille n contenant les valeurs des pièces (des entiers)
- s la somme à rendre
- T un tableau d'entiers à n + 1 lignes et s + 1 colonnes

L'algorithme permettant de compléter le tableau est le suivant :

```
pour i allant de 0 à n :
    T[i, 0] ← 0
pour j allant de 1 à s :
    res_ j ← +∞
    pour i allant de 0 à n – 1 :
        val ← pieces[i]
        si val <= j alors :
            res_ ij ← 1 + T[n, j – val]
            si res_ ij < res_ j alors :
                res_ j ← res_ ij
                i_opt ← i
        si res_ j ≠ +∞ alors :
            val ← pieces[i_opt]
            pour i allant de 0 à n – 1 :
                T[i, j] ← T[i, j – val]
            T[i_opt, j] ← T[i_opt, j] + 1

    # enregistrement d'une solution optimale

    T[n, j] ← res_ j
```

Suite à l'exécution de cet algorithme, le résultat optimal est dans la dernière colonne. T[n, s] contient le nombre minimal de pièce et T[i, s] pour i allant de 0 à n – 1 le nombre de pièces de chaque type.

Compléter le programme rendu_dynamique_tab_el.

<u>Essayer de nouveau pour 37 centimes</u>