TD : PROGRAMMATION DYNAMIQUE
== PARTITION ÉQUILIBRÉE D'UN TABLEAU D'ENTIERS POSITIFS ==

Remarque : les rappels théoriques sont en dernière page de ce sujet.

Le fichier source à utiliser pour ce TD est : « TD1 – Partition.py »

Une entreprise de logistique doit répartir des colis entre deux camions de livraison. Chaque colis a un poids donné, et l'objectif est de répartir les colis de sorte que la différence de charge entre les deux camions soit la plus faible possible. Cela permet d'équilibrer l'usure des véhicules et d'optimiser la consommation de carburant.

Vous disposez de n = 6 colis avec les poids suivants (en kg) :

Colis	**1**	**2**	**3**	**4**	**5**	**6**
Poids	10	10	10	1	5	10

La somme totale des poids est S = 46 kg. L'objectif est de trouver un sous-ensemble de colis dont la somme des poids est la plus proche possible de ⌊S/2⌋ = 23 kg.

L'objectif de ce TD est d'implémenter les algorithmes de programmation dynamique (approches top-down et bottom-up) pour résoudre ce problème, puis reconstruire la solution optimale. Vous utiliserez des dictionnaires Python pour mémoriser les résultats des sous-problèmes.

I) APPROCHE BOTTOM-UP (TABULATION)

Dans cette partie, vous allez implémenter l'approche bottom-up qui remplit une table de tous les sous-problèmes, des plus petits aux plus grands.

La liste des poids est déjà définie dans le fichier source : `poids = [10,10,10,1,5,10]`

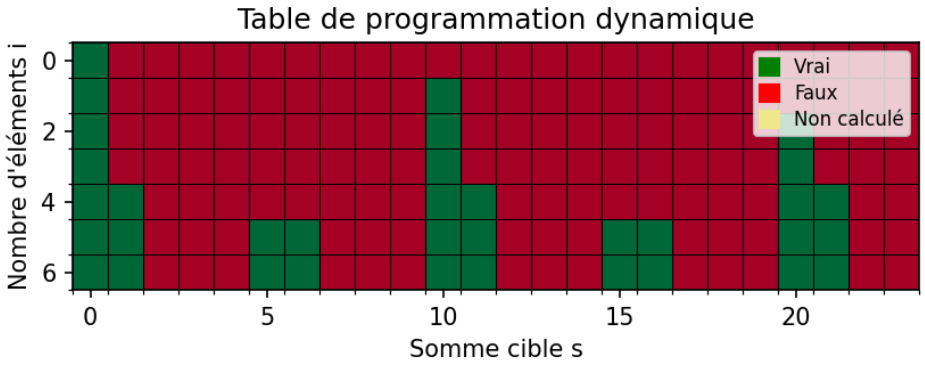
1. Écrire une fonction `initialiser_donnees(poids)` qui retourne :
   - La somme totale S
   - La cible idéale S_cible = S // 2

2. Écrire une fonction `initialiser_table(cible)` qui crée et retourne un dictionnaire représentant la table P. La fonction doit uniquement initialiser les cas de base pour le moment :
   - P[(0, 0)] = True (on peut atteindre 0 avec 0 éléments)
   - P[(0, s)] = False pour tout s de 1 à cible

3. Écrire une fonction `remplir_table(P, poids, cible)` qui remplit entièrement la table P en utilisant les équations de récurrence. Attention à l'ordre de parcours : on doit calculer P[(i, s)] pour i allant de 1 à n, et pour chaque i, s allant de 0 à cible.

4. Écrire une fonction `trouver_meilleure_somme_bottomup(P, cible, poids)` qui parcourt la dernière ligne de la table (i = n) pour trouver la plus grande somme s telle que P[(n, s)] = True.

```
Vérifier :  >>> trouver_meilleure_somme_bottomup(P,S_cible, poids)
            21
            >>> AfficheTable(P,S_cible, poids)
```



5. Combien de sous-problèmes sont calculés dans l'approche bottom-up ?

II) APPROCHE TOP-DOWN AVEC MÉMOÏSATION

Dans cette partie, vous allez implémenter l'algorithme récursif avec mémoïsation. L'idée est de partir du problème principal et de le décomposer en sous-problèmes, en mémorisant les résultats pour éviter les calculs redondants.

La somme cible `S_cible` et la somme totale `S` ont déjà été définies dans la première partie.

On utilisera un dictionnaire définit dans le programme général pour la mémoïsation : `P = {}`

1. Écrire une fonction récursive `rec_opt_val(i, s)` qui implémente la récurrence qui est rappelée à la fin du sujet.

```
Tester :    >>> rec_opt_val(6,23)          >>> rec_opt_val(6,21)
            False                          True
            >>> rec_opt_val(2,10)          >>> rec_opt_val(2,11)
            True                           False
```

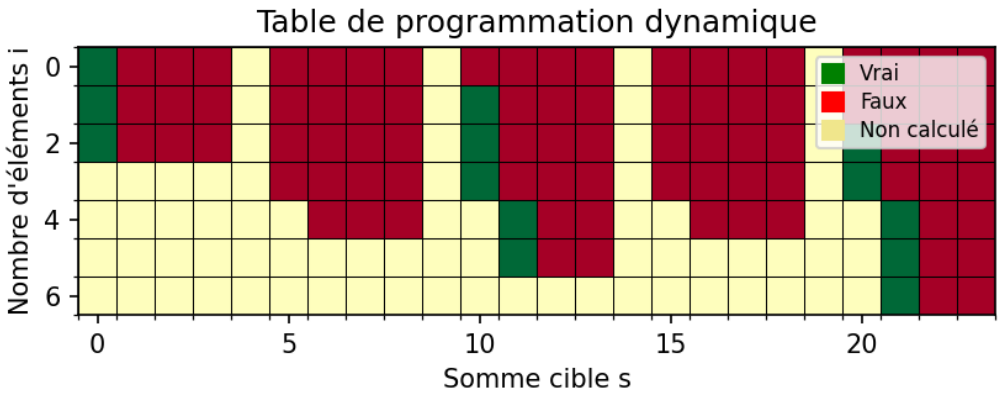
2. Écrire une fonction `trouver_meilleure_somme_topdown(cible, poids)` qui :
   - Teste d'abord si la cible idéale est atteignable
   - Si oui, retourne cette cible
   - Sinon, cherche la plus grande somme atteignable inférieure à la cible et la retourne

```
Tester :    >>> P = {}
            >>> trouver_meilleure_somme_topdown(S_cible,poids)
            21
```

3. Quelle est la complexité temporelle de l'algorithme top-down avec mémoïsation ? Justifiez votre réponse en considérant le nombre de sous-problèmes distincts et le coût de chaque sous-problème.

4. Quelle est la complexité spatiale de cet algorithme ? Prenez en compte à la fois le dictionnaire de mémoïsation et la pile d'appels récursifs.

Vérifier :
```
>>> P = {}
>>> trouver_meilleure_somme_topdown(S_cible,poids)
21
>>> AfficheTable(P,S_cible, poids)
```



III) RECONSTRUCTION DE LA SOLUTION

Maintenant que nous savons quelle est la meilleure somme atteignable, nous devons déterminer quels colis mettre dans chaque camion. Cette étape s'appelle la reconstruction de la solution.

La reconstruction consiste à « remonter » dans la table P pour déterminer, à chaque étape, si l'élément i a été pris ou non. On part de P[(n, s_opt)] et on remonte jusqu'à i = 0.

1. Écrire une fonction `element_pris(P, poids, i, s)` qui retourne `True` si l'élément i a été pris pour atteindre la somme s, `False` sinon.

2. Écrire une fonction `reconstruire_ensemble1(P, poids, n, s_opt)` qui retourne la liste des indices des colis à mettre dans le camion 1 (celui qui doit atteindre la somme s_opt).

Tester :
```
>>> camion1 = reconstruire_ensemble1(P,poids,S_opt)
>>> print (camion1)
[6, 4, 3]
```

3. Écrire une fonction `construire_ensemble2(poids, ensemble1)` qui retourne la liste des indices des colis à mettre dans le camion 2 (ceux qui ne sont pas dans le camion 1).

Tester :
```
>>> construire_ensemble2(poids, camion1)
[1, 2, 5]
```

4. Quelle est la complexité temporelle de l'algorithme de reconstruction ?

RAPPELS THÉORIQUES

Formulation du problème

Soit un tableau A = [a_1, a_2, ..., a_n] de n entiers positifs. On note S la somme totale des éléments. L'objectif est de partitionner ce tableau en deux sous-ensembles dont la différence des sommes est minimale.

Pour cela, on cherche un sous-ensemble dont la somme est la plus proche possible de ⌊S/2⌋. Si on trouve un sous-ensemble de somme exactement ⌊S/2⌋, la partition est parfaite (différence nulle ou 1).

Sous-problèmes et notation

On définit le sous-problème $P_{i,s}$ comme suit :

- $P_{i,s}$ = Vrai si on peut atteindre exactement la somme s en utilisant uniquement les i premiers éléments du tableau.
- $P_{i,s}$ = Faux sinon.

Relation de récurrence

Pour calculer $P_{i,s}$, on distingue deux cas selon que l'on prend ou non l'élément i :

- Cas n°1 (on ne prend pas l'élément i) : la somme s doit être atteignable avec les (i-1) premiers éléments.
- Cas n°2 (on prend l'élément i) : la somme ($s - a_i$) doit être atteignable avec les (i-1) premiers éléments.

La récurrence s'écrit donc (avec l'opérateur OU logique) :

Pour tout i = 1, 2, …, n et tout s = 0, 1, 2, …, ⌊S/2⌋ :

$$P_{i,s} = \begin{cases} P_{i-1,s} & a_i > s \\ P_{i-1,s} \; OU \; P_{i-1,s-a_i} & a_i \leq s \end{cases}$$

Cas de base

Les cas de base sont les suivants :

- $P_{0,0}$ = Vrai : on peut atteindre la somme 0 avec 0 éléments (en ne prenant rien).
- $P_{0,s}$ = Faux pour tout s > 0 : on ne peut pas atteindre une somme positive sans aucun élément.

Algorithme de reconstruction

Une fois la table P remplie et la meilleure somme atteignable s_opt trouvée, on reconstruit la solution en « remontant » dans la table depuis P_{n,s_opt} jusqu'à i = 0.

Principe : Pour chaque élément i (de n à 1), on détermine s'il a été pris ou non :

- Si $a_i \leq s$ ET $P_{i-1,s-a_i}$ = Vrai, alors l'élément i a été pris. On l'ajoute à l'ensemble et on met à jour s := s - a_i.
- Sinon, l'élément i n'a pas été pris. On passe à l'élément suivant sans modifier s.