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

Notre deuxième étude de cas concerne le problème de partitionnement d'un tableau d'entiers positifs. Nous allons construire la solution du problème par programmation dynamique.

I) DÉFINITION DU PROBLÈME

Une instance du problème de la partition équilibrée est spécifiée par n entiers positifs, où n est le nombre d'éléments du tableau (qui sont indexés de 1 à n) : une valeur a_i pour chaque élément i.

On note la somme totale $S = a_1 + a_2 + \dots + a_n$ et $\lfloor S/2 \rfloor$ la partie entière de S/2.

La tâche de l'algorithme est de diviser l'ensemble des éléments en deux groupes de telle sorte que leurs sommes soient aussi proches que possible. Autrement dit, on cherche deux sous-ensembles disjoints P et $\overline{P}$ formant une partition de {1, 2, …, n} :

- $P \subseteq \{1, \dots, n\}$,
- $\overline{P} = \{1, \dots, n\} \setminus P$,

… tels que la différence de somme $\left|\sum_{i \in P} a_i - \sum_{i \notin P} a_i\right|$ soit minimale.

Le problème est donc de trouver un sous-ensemble $P \subseteq \{1, \dots, n\}$ tel que $\sum_{i \in P} a_i \leq \lfloor S/2 \rfloor$ et que $\sum_{i \in P} a_i$ soit maximale.

Problème : partition équilibrée d'un tableau d'entiers positifs

Entrée : un tableau d'entiers positifs $a_1, a_2, \dots, a_n$.

Sortie : un sous-ensemble $P \subseteq \{1, \dots, n\}$ tel que $\sum_{i \in P} a_i \leq \lfloor S/2 \rfloor$ et $\sum_{i \in P} a_i$ soit maximal, ou de manière équivalente tel que la différence $\left|\sum_{i \in P} a_i - \sum_{i \notin P} a_i\right|$ soit minimale.

Exemple : soit un problème de partition équilibrée avec les quatre entiers [3, 1, 4, 2]. La somme totale des objets vaut S = 3 + 1 + 4 + 2 = 10. La demi-somme vaut $\lfloor S/2 \rfloor = 5$. On cherche donc un sous-ensemble d'indices $P \subseteq \{1, 2, 3, 4\}$ dont la somme est au plus 5 et aussi grande que possible.

Indice	a_i
1	3
2	1
3	4
4	2

Parmi les partitions possibles, on a par exemple :

- P = {3} : somme(P) = 4 ; $\overline{P}$ = {1, 2, 4} : somme($\overline{P}$) = 3 + 1 + 2 = 6, différence = 2
- P = {1, 2} : somme(P) = 3 + 1 = 4 ; $\overline{P}$ = {3, 4} : somme($\overline{P}$) = 4 + 2 = 6, différence = 2
- P = {1, 4} : somme(P) = 3 + 2 = 5 ; $\overline{P}$ = {2, 3} : somme($\overline{P}$) = 1 + 4 = 5, différence = 0

La meilleure partition ici est par exemple (P = {1, 4}, $\overline{P}$ = {2, 3}), qui donne deux paquets de somme 5 chacun. La différence est nulle : on a une partition parfaitement équilibrée.

Ce problème apparaît dans de nombreux domaines de la vie réelle :

- Répartition de tâches entre deux machines : on dispose d'un ensemble de tâches, chacune avec une durée estimée (ou un coût CPU). On souhaite les répartir sur deux processeurs de façon à minimiser le temps de fin global.
- Découpage d'un groupe en deux équipes équilibrées : on veut diviser des élèves ou des joueurs en deux équipes de niveaux aussi proches que possible, à partir de scores ou d'indices de performance.
- Répartition de fichiers sur deux disques : chaque fichier a une taille ; on souhaite les placer sur deux disques de façon à équilibrer l'espace utilisé.

Pour résoudre ce problème de manière exhaustive (par brute force), il faut :

- Lister tous les sous-ensembles possibles $P \subseteq \{1, \ldots, n\}$;
- Pour chacun :
  - calculer $\sum_{i \in P} a_i$,
  - en déduire la somme de l'autre paquet $\sum_{i \notin P} a_i = S - \sum_{i \in P} a_i$
  - calculer la différence $\left|\sum_{i \in P} a_i - \sum_{i \notin P} a_i\right|$
- Garder le sous-ensemble P qui donne la plus petite différence.

Pour n éléments, il y a 2^n sous-ensembles possibles (chaque élément est soit dans P, soit dans $\overline{P}$). Pour chaque sous-ensemble, calculer la somme coûte O(n) dans une version naïve (on additionne les éléments un par un).

Au total, la complexité de cette méthode brute force est donc de $O(n \cdot 2^n)$ en temps et de O(n) en espace mémoire pour stocker le sous-ensemble courant et la meilleure solution trouvée.

C'est un algorithme exponentiel en n, impossible à utiliser sur de grandes instances.

II) SOUS-STRUCTURE OPTIMALE ET RELATION DE RÉCURRENCE

Nous pouvons déterminer une collection de sous-problèmes en raisonnant sur la structure des solutions optimales et en identifiant les différentes façons dont elles peuvent être construites à partir de solutions optimales de sous-problèmes plus petits.

II.1. Sous-structure optimale

Considérons une instance du problème de partition équilibrée avec les entiers positifs $a_1, a_2, \ldots, a_n$ et la demi-somme $\lfloor S/2 \rfloor$. Supposons que quelqu'un nous donne, sur un plateau, une solution optimale $P \subseteq \{1, 2, \ldots, n\}$ de valeur totale $S_{max} = \sum_{i \in P} a_i$, avec $S_{max} \leq \lfloor S/2 \rfloor$.

On peut se demander : soit P contient le dernier élément (l'élément n), soit elle ne le contient pas :

Cas n°1 : $n \notin P$: supposons que la solution optimale P ne contient pas le dernier élément n. Dans ce cas, P est entièrement constitué d'indices dans $\{1, 2, \ldots, n-1\}$. On peut la considérer comme une solution réalisable (valeur totale $S_{max} \leq \lfloor S/2 \rfloor$) du problème plus petit ne comportant que les $(n - 1)$ premiers éléments, avec la même contrainte de somme maximale S_{max}.

Cas n°2 : $n \in P$: supposons que la solution optimale P contient le dernier élément n.

Ce cas ne peut se produire que si $a_n \leq S_{max}$. Dans ce cas, si l'on retire l'élément n de P, on obtient le sous-ensemble $P - \{n\}$, qui est une solution à un sous-problème plus petit : il ne reste plus que les $(n - 1)$ premiers éléments, et la somme maximale autorisée est réduite à $S_{max} - a_n$. La valeur totale de $P - \{n\}$ est alors $(S_{max} - a_n)$, et $P - \{n\}$ est une solution optimale au sous-problème utilisant seulement les $(n - 1)$ premiers éléments et une somme maximale $S_{max} - a_n$.

II.2. Équation de récurrence sur les valeurs optimales

Même si nous avons formulé le problème sous la forme d'une sous-structure optimale (maximiser une somme $S_{max} \leq \lfloor S/2 \rfloor$), nous n'allons pas exploiter directement cette fonction de valeur optimale dans notre implémentation.

En pratique, pour construire une partition équilibrée, il suffit de savoir, pour chaque préfixe d'éléments et chaque somme s, si cette somme est réalisable ou non. Nous allons donc raisonner sur la faisabilité des sommes et définir une valeur booléenne qui prendra la valeur « Vrai » s'il existe un sous-ensemble réalisant exactement une certaine somme et « Faux » si aucune combinaison ne permet d'atteindre cette somme.

Notons $T_{i,s}$ un booléen définissant :

- $T_{i,s}$ = « Vrai » s'il existe un sous-ensemble des i premiers éléments {1, …, i} dont la somme est exactement s ;
- $T_{i,s}$ = « Faux » sinon.

Les cas de base sont :

- $T_{0,0}$ = « Vrai » (on peut faire la somme 0 en ne prenant aucun élément) ;
- $T_{0,s}$ = « Faux » pour tout s > 0 (avec 0 élément, aucune somme positive n'est réalisable).

Pour réaliser la somme s avec les i premiers éléments, il n'y a que deux solutions possibles :

Cas n°1 : Ne pas prendre l'élément i (si $s < a_i$).
Dans ce cas si $T_{i-1,s}$ = « Faux » alors $T_{i,s}$ = « Faux » et $T_{i,s}$ = « Vrai » dans les autres cas.

Cas n°2 : Prendre l'élément i (si $s \geq a_i$).
Dans ce cas, pour pouvoir faire la somme s en prenant l'élément i, il faut pouvoir faire la somme $(s - a_i)$ avec les (i - 1) premiers éléments (et on ajoute a_i) ou soit pouvoir faire la somme s avec les (i – 1) premiers éléments (et on n'ajoute pas a_i).

On obtient finalement la table de vérité suivante :

$s \geq a_i$ (on prend l'élément i de valeur a_i)	$T_{i-1,s-ai}$	$T_{i-1,s}$	$T_{i,s}$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

On obtient l'équation logique :

$$T_{i,s} = T_{i-1,s}\ OU\ \left(s \geq a_i\ ET\ T_{i-1,s-a_i}\right)$$

Cela conduit à la relation de récurrence suivante :

Récurrence sur la valeur de la solution optimale

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

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

Avec les conditions de base :
- $T_{0,0}$ = « Vrai »
- $T_{0,s}$ = « Faux » pour tout $s \in \{1, \ldots, \lfloor S/2 \rfloor\}$

Une fois que l'on a calculé $T_{i,s}$ pour tous les s de 0 à $S_{max} = \lfloor S/2 \rfloor$, on peut choisir la meilleure somme atteignable s* :

$$s^* = max\{s \in \{0, \ldots, \lfloor S/2 \rfloor\} \mid T_{n,s} = "Vrai"\}$$

Remarque : dans ce qui précède, nous avons supposé que toutes les valeurs $T_{i,s}$ sont disponibles afin de choisir la meilleure somme atteignable s^*. Cette hypothèse est naturellement vérifiée dans l'approche bottom-up, qui remplit systématiquement toute la table $T_{i,s}$.

En top-down avec mémoïsation, en revanche, seuls les sous-problèmes effectivement visités par la récursion sont calculés : un appel initial sur (n, ⌊S/2⌋) ne suffit donc pas à connaître toutes les valeurs $T_{i,s}$. Si la somme ⌊S/2⌋ n'est pas atteignable, il faut alors lancer des appels récursifs supplémentaires sur (n, s) pour s = ⌊S/2⌋−1, ⌊S/2⌋−2, … jusqu'à trouver une somme atteignable s^*. Ces appels calculent au passage, via la mémoïsation, toutes les entrées $T_{i,s}$ nécessaires pour la reconstruction, sans pour autant remplir toute la table comme en bottom-up.

III) SOUS-PROBLÈMES ET COMPLEXITÉ

III.1. Définition des sous-problèmes

L'étape suivante consiste à définir la collection de sous-problèmes pertinents et à les résoudre systématiquement en utilisant la relation de récurrence.

Pour l'instant, nous nous concentrons sur le calcul de la table booléenne $T_{i,s}$ qui indique si la somme s est réalisable avec les i premiers éléments. La reconstruction de la partition équilibrée sera faite plus tard à partir de ces informations.

Pour le problème de la partition équilibrée, les sous-problèmes sont paramétrés par deux indices : i (longueur du préfixe des éléments disponibles) et s (somme cible de 0 à ⌊S/2⌋). En faisant varier ces deux paramètres sur toutes les valeurs pertinentes, nous obtenons la famille de sous-problèmes.

Sous-problèmes du partitionnement équilibré d'un tableau d'entiers positifs

Calculer $T_{i,s}$, la valeur booléenne indiquant si la somme s est réalisable par un sous-ensemble des i premiers éléments.

(Pour chaque i = 0, 1, 2, …, n et s = 0, 1, 2, …, ⌊S/2⌋)

Le plus grand sous-problème (i = n, s = ⌊S/2⌋) joue ici un rôle un peu particulier : on ne sait pas encore si la meilleure valeur de s sera exactement ⌊S/2⌋ ou un peu plus petite, mais c'est à partir de la ligne i = n de la table que l'on sélectionnera cette meilleure valeur.

Comme toutes les valeurs a_i des éléments sont des entiers positifs, les seules sommes qui peuvent apparaître sont les entiers compris entre 0 et ⌊S/2⌋. Comme on s'intéresse seulement à des sommes ≤ ⌊S/2⌋, nous pouvons limiter s à l'intervalle [0, ⌊S/2⌋].

III.2. Schéma de récursion

On peut représenter les appels récursifs de la fonction $T_{i,s}$ par un arbre de récursion. Dans ce schéma :

- La notation $[a_1,a_2,a_3][3]$ signifie qu'on cherche à savoir si on peut réaliser la somme 3 avec les trois premiers éléments $[a_1,a_2,a_3]$;
- La notation T[3][3] = V signifie que la case correspondante dans la table booléenne vaut « Vrai » (la somme 3 est réalisable) ;
- Dans les cas n°1 (branches de gauche), à partir des cas de base du bas, on remonte la valeur $T_{i-1,s}$;
- Dans les cas n°2 (branches de droite), on remonte la valeur $T_{i-1,s-a_i}$.
- Les valeurs T[·][·] prennent le résultat du OU logique entre les deux valeurs remontées.

https://www.informatique-f1.fr/dp/partition/

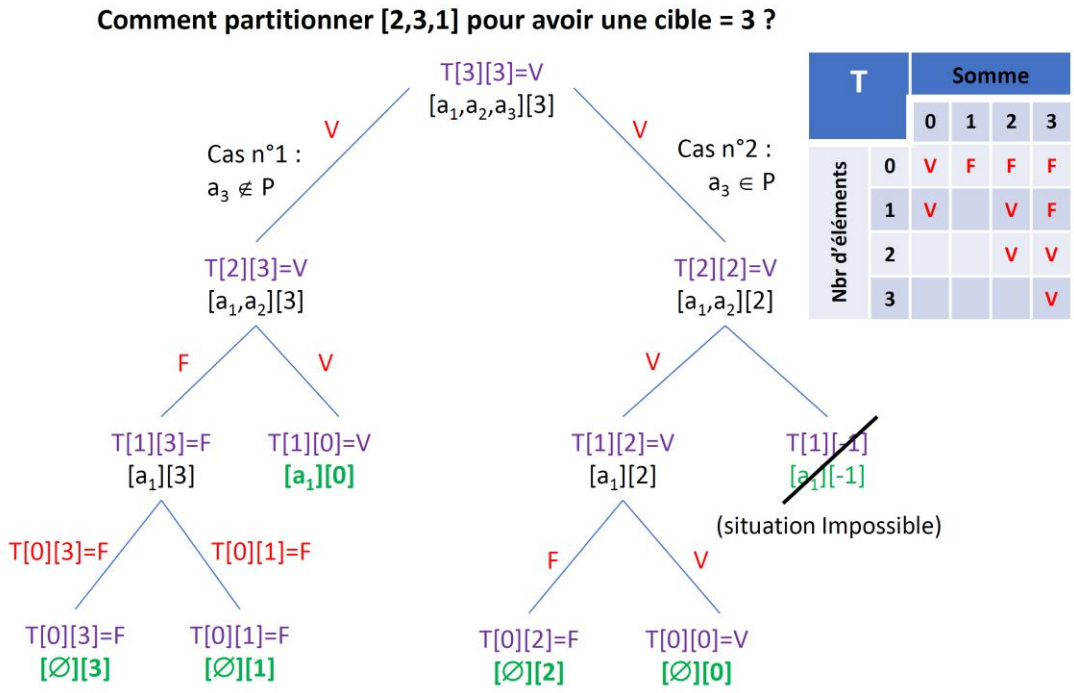

Figure 1 : Schéma de récursion du problème

III.3. Complexité sans mémoïsation

Chaque niveau de récursivité ne peut enlever qu'un seul élément. Il faut donc descendre jusqu'au niveau n pour avoir des cas de base. Tous les nœuds jusqu'au niveau (n – 1) sont donc des nœuds internes qui se ramifient encore, avec un facteur de branchement égal à 2 (si on ne tient pas compte des cas où $a_i > s$). Le nombre de nœuds peut donc aller jusqu'à 2^n au niveau n. Le maximum de nœuds est donc de $1 + 2 + 4 + \ldots + 2^n = 2^{n+1} – 1$.

À chaque nœud de l'arbre de récursion, le travail local (hors appels récursifs) se fait en temps O(1) : on effectue seulement un nombre constant d'opérations (comparaisons, opération OU logique). Comme l'arbre de récursion est binaire et peut contenir jusqu'à $O(2^n)$ nœuds dans le pire des cas, le temps d'exécution de cet algorithme récursif sans mémoïsation est exponentiel, en $O(2^n)$.

Remarquons qu'en pratique, l'algorithme top-down ne résout que les sous-problèmes qui sont réellement atteints en partant de l'état initial (n, ⌊S/2⌋) et en suivant la récurrence. Certains couples (i, s) ne sont jamais visités : par exemple parce que certaines sommes ne peuvent pas apparaître, ou parce que des branches sont coupées quand un élément est trop grand ($a_i > s$).

IV) ALGORITHMES DE PROGRAMMATION DYNAMIQUE

IV.1. Algorithme top-down

Étant donnés les sous-problèmes et la relation de récurrence, on peut mettre en place un algorithme top-down (avec mémoïsation) de programmation dynamique pour le problème de la partition équilibrée.

Particularité de notre approche : l'algorithme effectue d'abord une recherche récursive pour savoir si la somme cible ⌊S/2⌋ est exactement atteignable. Si ce n'est pas le cas, il cherche récursivement la plus grande somme atteignable inférieure à ⌊S/2⌋, ce qui minimise la différence entre les deux parties de la partition.

Cette approche est conforme au comportement top-down qui ne calcule que les sous-problèmes réellement visités. Lors de la première passe (test de la cible ⌊S/2⌋), seuls les sous-problèmes sur le chemin de récurrence sont mémoïsés. Si la cible n'est pas atteinte, les passes suivantes bénéficient de la mémoïsation : beaucoup de sous-problèmes sont déjà calculés, ce qui accélère la recherche de la meilleure somme atteignable.

L'algorithme est donné en page suivante.

Algorithme top-down pour le calcul des valeurs optimales

Entrée : a[1, …, n] : valeurs des éléments

Sortie : la plus grande somme atteignable $s \leq \lfloor S/2 \rfloor$

```
T := {}                          # Dictionnaire de mémoïsation
S := somme(a[1..n])              # Somme totale
cible := S // 2                  # Cible idéale (division entière)

rec_opt_val_partition (i,s) :
    # i : nombre d'éléments considérés (les i premiers)
    # s : somme cible à atteindre

    # Utilise la mémoïsation
    Si (i, s) est dans T :
    |   Retourner T[(i, s)]

    # Cas de base
    Si s == 0 :
    |   T[(i, s)] := Vrai
    |   Retourner T[(i, s)]
    Si i == 0 :
    |   T[(i, s)] := Faux
    |   Retourner T[(i, s)]

    # Récursion cas n°1 : on ne prend pas l'élément i
    resultat := rec_opt_val_partition(i – 1, s)

    # Cas n°2 : on prend l'élément i (si possible)
    Si a[i] <= s :
    |   resultat := resultat OU  rec_opt_val_partition(i – 1, s – a[i])

    # Sauvegarde et retourne la valeur optimale
    T[(i, s)] := resultat
    Retourner resultat

# Appel principal
Si rec_opt_val_partition(n, cible) == Vrai :
|   Retourner cible
Sinon :
|   Pour s allant de (cible – 1) à 0 :
|   |   Si rec_opt_val_partition(n, s) == Vrai :
|   |   |   Retourner s
```

IV.2. Complexité de l'algorithme top-down

Chaque sous-problème est défini par deux paramètres : le nombre d'éléments considérés (0 à n) et la somme cible (0 à ⌊S/2⌋). Les états possibles sont donc les couples (i, s) avec $0 \le i \le n$ et $0 \le s \le \lfloor S/2 \rfloor$. Le nombre maximal de sous-problèmes distincts est donc de $(n+1)\cdot(\lfloor S/2 \rfloor+1) = O(n\cdot S)$.

Avec la mémoïsation, chaque couple (i, s) est calculé au plus une fois et les appels suivants sur les mêmes couples font uniquement un accès dans le dictionnaire des valeurs en O(1).

Lors de la résolution d'un sous-problème (i, s) non mémorisé, l'algorithme effectue un travail local en O(1) (comparaisons, opération OU logique), ainsi qu'au plus deux appels récursifs vers des sous-problèmes comme (i−1, s) et $(i-1, s-a_i)$.

Comme chaque sous-problème est résolu au plus une fois, le nombre total d'appels « réels » est en O(n·S), et la complexité en temps est donc O(n·S).

L'espace mémoire utilisé par le dictionnaire de mémoïsation est en O(n·S) et la profondeur de la pile d'appels récursifs est au maximum de n, soit O(n). Le total de l'espace mémoire est donc dominé par le dictionnaire et est de O(n·S).

IV.3. Algorithme bottom-up

L'algorithme bottom-up consiste à remplir progressivement la table des solutions des sous-problèmes en utilisant la relation de récurrence, en partant des cas de base.

Les tables construites par les algorithmes top-down et bottom-up sont données ci-dessous pour l'exemple [2,3,1] avec cible = 3 :

Comment partitionner [2,3,1] pour avoir S = 3 ?

Top-down

T	Somme			
	0	1	2	3
Nbr d' éléments 0	V	F	F	F
1	V		V	F
2			V	V
3				V

Bottom-up

T	Somme			
	0	1	2	3
Nbr d' éléments 0	V	F	F	F
1	V	F	V	F
2	V	F	V	V
3	V	V	V	V

Figure 2 : Tables des valeurs optimales top-down (gauche) et bottom-up (droite)

IV.4. Complexité de l'algorithme bottom-up

À l'inverse de l'algorithme top-down, l'algorithme bottom-up parcourt systématiquement toute la table T[i, s] pour i = 0..n et s = 0..⌊S/2⌋, même pour des états qui ne seront jamais « utiles » pour la solution finale. Il effectue donc toujours exactement $(n + 1)\cdot(\lfloor S/2 \rfloor + 1)$ calculs, indépendamment de la structure de l'instance.

On a donc la même complexité asymptotique O(n·S) pour les deux approches, mais en pratique le top-down peut faire moins de travail effectif sur certaines instances, alors que le bottom-up remplit la table de manière uniforme, quitte à calculer des sous-problèmes inutiles.

Algorithme bottom-up pour le calcul des valeurs optimales

Entrée : a[1, …, n] : valeurs des éléments
Sortie : la plus grande somme atteignable $s \leq \lfloor S/2 \rfloor$

```
T := {}                          # Dictionnaire de mémoïsation
S := somme(a[1..n])              # Somme totale
cible := S // 2                  # Cible idéale (division entière)

opt_val_partition (a) :
    # Cas de base : avec 0 élément, seule la somme 0 est atteignable
    T[(0, 0)] := Vrai
    Pour s allant de 1 à cible :
    |   T[(0, s)] := Faux

    # Résout l'ensemble des sous-problèmes
    Pour i allant de 1 à n :
    |   Pour s allant de 0 à cible :
    |   |   # Utilise l'équation de récurrence
    |   |   Si a[i] > s :
    |   |   |   T[(i, s)] := T[(i-1, s)]
    |   |   Sinon :
    |   |   |   T[(i, s)] := T[(i-1, s)] OU T[(i-1, s-a[i])]

    # Cherche la plus grande somme atteignable
    Pour s allant de cible à 0 (par pas de -1) :
    |   Si T[(n, s)] == Vrai :
    |   |   Retourner s
```

V) ALGORITHME DE RECONSTRUCTION

V.1. Principe et algorithme de reconstruction

On peut reconstruire une solution optimale en retraçant le chemin dans le tableau T une fois rempli.

En partant du plus grand sous-problème, l'algorithme de reconstruction vérifie quel cas de la récurrence a été utilisé pour calculer T[n][s*] (où s* est la meilleure somme atteignable trouvée) :

- Si $s \geq a_i$ et T[i-1][s-a_i] == Vrai, alors c'est le cas n°2 : on prend l'élément i et on reprend la construction à partir de l'entrée T[i-1][s-a_i] ;
- Sinon, c'est le cas n°1 : on ne prend pas l'élément i et on reprend la reconstruction à partir de l'entrée T[i-1][s] ;

L'algorithme de reconstruction est le suivant :

Algorithme de reconstruction (méthode 1)

Entrée : a[1, …, n] : valeurs des éléments
T : dictionnaire des booléens
s_opt : somme optimale atteinte

Sortie : P : solution optimale du problème (indices des éléments de la première partie)

Reconstruction (a, T, s_opt) :
P := ∅ # Éléments de la solution
s := s_opt # Somme restante à atteindre

Pour i allant de n à 1 :
On vérifie si on peut prendre l'élément i (cas n°2)
Si s >= a[i] ET T[(i-1, s-a[i])] == Vrai :
P := P ∪ {i}
s := s - a[i]
Sinon : cas n°1, on n'inclut pas i, s reste inchangé

Retourner P

La figure ci-dessous illustre ce principe de reconstruction :

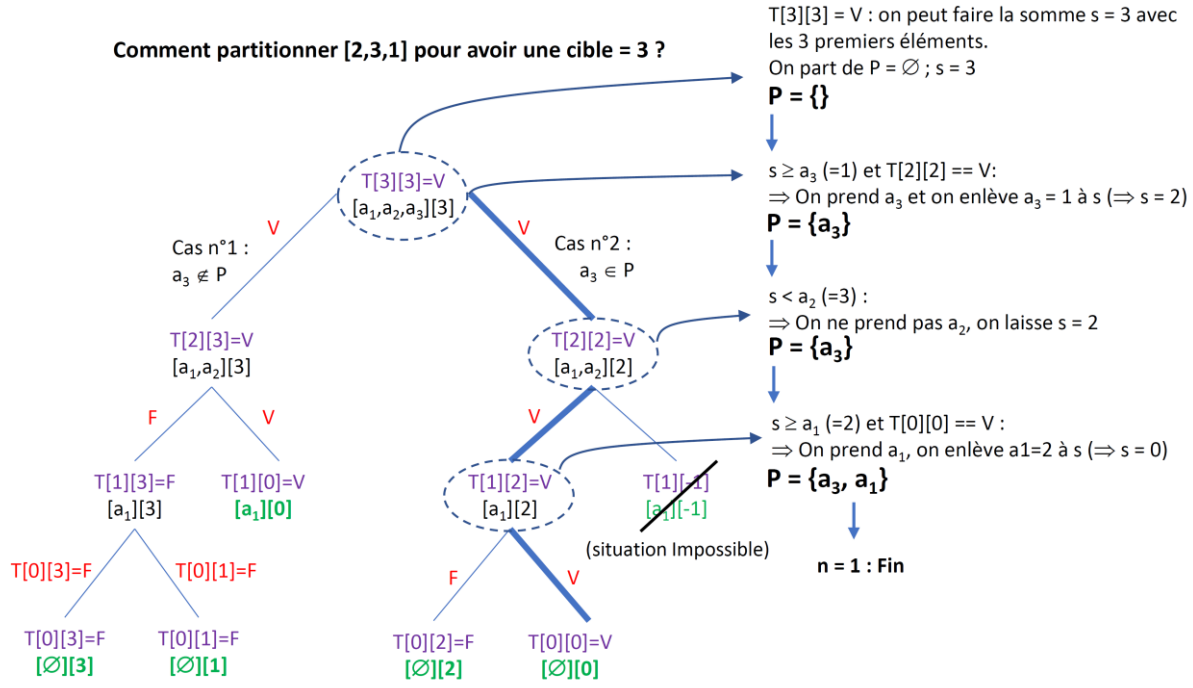

Figure 3 : Principe de reconstruction de la solution optimale (méthode 1)

L'algorithme de reconstruction présenté ci-dessus privilégie le cas n°2 (prendre l'élément) lorsque les deux choix sont possibles. On pourrait tout aussi bien privilégier le cas n°1 (ne pas prendre l'élément) en testant d'abord si T[i-1][s] est Vrai : si c'est le cas, on passe directement à l'élément suivant sans modifier la somme cible ; sinon, on prend l'élément i.

Les deux approches sont équivalentes en termes de complexité et produisent des partitions optimales, mais elles peuvent conduire à des solutions différentes lorsque plusieurs partitions optimales existent.

Algorithme de reconstruction (méthode 2)

Entrée : a[1, …, n] : valeurs des éléments
T : dictionnaire des booléens
s_opt : somme optimale atteinte

Sortie : P : solution optimale du problème (indices des éléments de la première partie)

Reconstruction (a, T, s_opt) :
```
    P := ∅          # Éléments de la solution
    s := s_opt      # Somme restante à atteindre

    Pour i allant de n à 1 :
        # On vérifie d'abord si on peut NE PAS prendre l'élément i (cas n°1)
        Si T[(i-1, s)] == Vrai :
            # Cas n°1 : on ne prend pas l'élément i
            # (on ne fait rien, on passe à i-1 avec la même somme s)
            Continuer
        Sinon :
            # Cas n°2 : on doit prendre l'élément i
            P := P ∪ {i}
            s := s - a[i]
    Retourner P
```

La figure ci-dessous illustre ce principe de reconstruction :

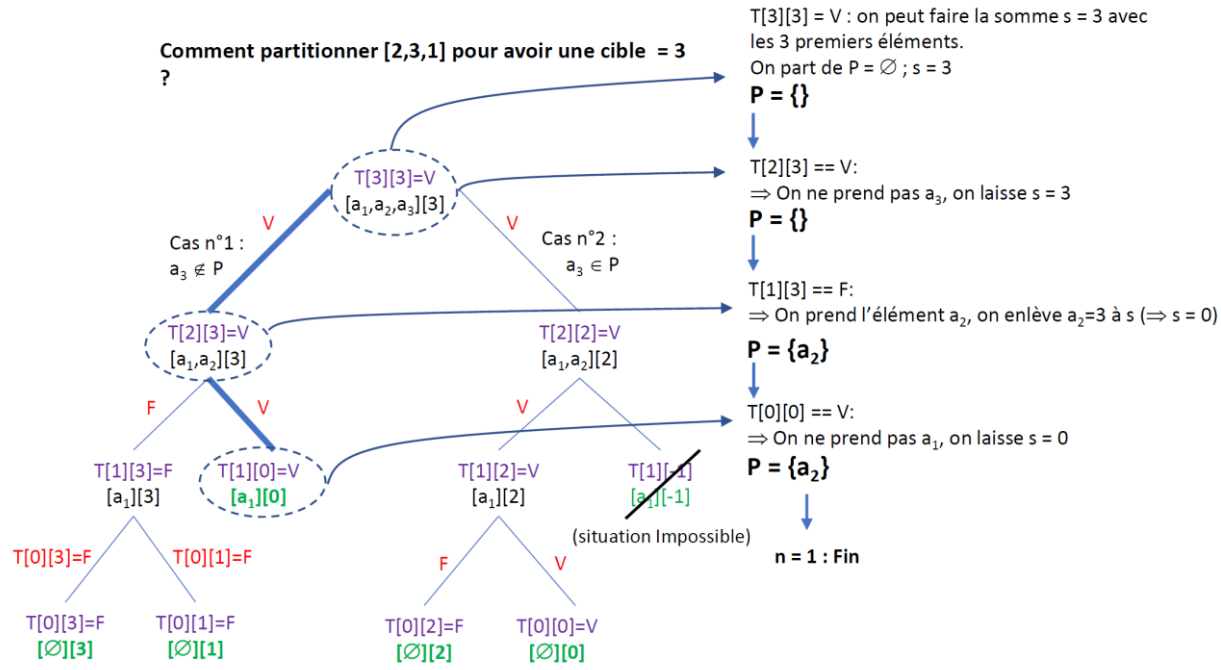

Figure 4 : Principe de reconstruction de la solution optimale (méthode 2)

V.2. Complexité finale

L'étape de reconstruction s'exécute en temps O(n) (avec un travail en O(1) par itération de la boucle principale), ce qui est beaucoup plus rapide que le temps O(n·S) nécessaire pour remplir le tableau des booléens.

Le problème de la partition équilibrée peut donc être résolu par programmation dynamique en temps O(n·S).