Programmation Dynamique

Prof d'info

Lycée Thiers

Historique

- *Programmation dynamique* : processus de résolution de problèmes où on trouve les meilleures décisions les unes après les autres.
  Le terme était utilisé par le mathématicien Richard Bellman dès les années 40.

Historique

- *Programmation dynamique* : processus de résolution de problèmes où on trouve les meilleures décisions les unes après les autres.
  Le terme était utilisé par le mathématicien Richard Bellman dès les années 40.
- En 1953, Bellman en donne la définition moderne, où les décisions à prendre sont ordonnées par sous-problèmes.
  le domaine a alors été reconnu par l'Institute of Electrical and Electronics Engineers (IEEE) comme un sujet d'analyse de systèmes et d'ingénierie.

Diviser pour régner

- La méthode *Diviser pour régner* est un cas particulier de programmation dynamique.

Diviser pour régner

- La méthode *Diviser pour régner* est un cas particulier de programmation dynamique.
- On décompose encore un problème principal en sous-problèmes. Cependant, les sous-problèmes sont ici indépendants les uns des autres ce qui facilite la tâche du programmeur.

Diviser pour régner

- La méthode *Diviser pour régner* est un cas particulier de programmation dynamique.
- On décompose encore un problème principal en sous-problèmes. Cependant, les sous-problèmes sont ici indépendants les uns des autres ce qui facilite la tâche du programmeur.
- Principe :

Diviser pour régner

- La méthode *Diviser pour régner* est un cas particulier de programmation dynamique.
- On décompose encore un problème principal en sous-problèmes. Cependant, les sous-problèmes sont ici indépendants les uns des autres ce qui facilite la tâche du programmeur.
- Principe :
  - On « divise » en réduisant un problème en sous-problèmes du même type et qui ne se *chevauchent* pas.

Diviser pour régner

- La méthode *Diviser pour régner* est un cas particulier de programmation dynamique.
- On décompose encore un problème principal en sous-problèmes. Cependant, les sous-problèmes sont ici indépendants les uns des autres ce qui facilite la tâche du programmeur.
- Principe :
  - On « divise » en réduisant un problème en sous-problèmes du même type et qui ne se *chevauchent* pas.
  - Puis on règne en résolvant ces sous-problèmes.

Diviser pour régner

- La méthode *Diviser pour régner* est un cas particulier de programmation dynamique.
- On décompose encore un problème principal en sous-problèmes. Cependant, les sous-problèmes sont ici indépendants les uns des autres ce qui facilite la tâche du programmeur.
- Principe :
  - On « divise » en réduisant un problème en sous-problèmes du même type et qui ne se *chevauchent* pas.
  - Puis on règne en résolvant ces sous-problèmes.
  - Il reste à combiner les solutions des sous-problèmes pour obtenir une soution au problème initial.

Cadre d'application

- La programmation dynamique est envisagée si le problème présente la *propriété de sous-structure optimale* et si les *chevauchements de sous-problèmes* doivent être gérés.

Cadre d'application

- La programmation dynamique est envisagée si le problème présente la *propriété de sous-structure optimale* et si les *chevauchements de sous-problèmes* doivent être gérés.
- Vocabulaire :

Cadre d'application

- La programmation dynamique est envisagée si le problème présente la *propriété de sous-structure optimale* et si les *chevauchements de sous-problèmes* doivent être gérés.
- Vocabulaire :

  Sous-structure optimale Se dit d'un problème qu'on peut résoudre en le décomposant en sous-problèmes du même type, eux-mêmes résolubles récursivement.

Cadre d'application

- La programmation dynamique est envisagée si le problème présente la *propriété de sous-structure optimale* et si les *chevauchements de sous-problèmes* doivent être gérés.
- Vocabulaire :

  Sous-structure optimale Se dit d'un problème qu'on peut résoudre en le décomposant en sous-problèmes du même type, eux-mêmes résolubles récursivement.

  Chevauchement de sous-problèmes Se dit si des sous-problèmes ne sont pas indépendants et doivent être résolus plusieurs fois.

Cadre d'application

- La programmation dynamique est envisagée si le problème présente la *propriété de sous-structure optimale* et si les *chevauchements de sous-problèmes* doivent être gérés.
- Vocabulaire :

  Sous-structure optimale Se dit d'un problème qu'on peut résoudre en le décomposant en sous-problèmes du même type, eux-mêmes résolubles récursivement.

  Chevauchement de sous-problèmes Se dit si des sous-problèmes ne sont pas indépendants et doivent être résolus plusieurs fois.
- En général on envisage tous les sous-problèmes comme dans une recherche exhaustive mais on prend ses précautions pour ne pas avoir à les résoudre tous :

Cadre d'application

- La programmation dynamique est envisagée si le problème présente la *propriété de sous-structure optimale* et si les *chevauchements de sous-problèmes* doivent être gérés.
- Vocabulaire :

  Sous-structure optimale Se dit d'un problème qu'on peut résoudre en le décomposant en sous-problèmes du même type, eux-mêmes résolubles récursivement.

  Chevauchement de sous-problèmes Se dit si des sous-problèmes ne sont pas indépendants et doivent être résolus plusieurs fois.
- En général on envisage tous les sous-problèmes comme dans une recherche exhaustive mais on prend ses précautions pour ne pas avoir à les résoudre tous :
  - soit parce que certains sont inutiles (ex : recherche dichotomique)

Cadre d'application

- La programmation dynamique est envisagée si le problème présente la *propriété de sous-structure optimale* et si les *chevauchements de sous-problèmes* doivent être gérés.
- Vocabulaire :

  Sous-structure optimale Se dit d'un problème qu'on peut résoudre en le décomposant en sous-problèmes du même type, eux-mêmes résolubles récursivement.

  Chevauchement de sous-problèmes Se dit si des sous-problèmes ne sont pas indépendants et doivent être résolus plusieurs fois.
- En général on envisage tous les sous-problèmes comme dans une recherche exhaustive mais on prend ses précautions pour ne pas avoir à les résoudre tous :
  - soit parce que certains sont inutiles (ex : recherche dichotomique)
  - soit parce qu'ils ont déjà été rencontrés et résolus (ex : mémoïsation dans le calcul des suites de Fibonacci)

Principe d'optimalité

- La programmation dynamique s'applique à des problèmes d'optimisations : il s'agit souvent d'optimiser le coût d'une suite de décisions.

Principe d'optimalité

- La programmation dynamique s'applique à des problèmes d'optimisations : il s'agit souvent d'optimiser le coût d'une suite de décisions.
- Cette suite de décisions correspond à un découpage du problème en sous-problèmes :

Principe d'optimalité

- La programmation dynamique s'applique à des problèmes d'optimisations : il s'agit souvent d'optimiser le coût d'une suite de décisions.
- Cette suite de décisions correspond à un découpage du problème en sous-problèmes :
  - On calcule les solutions optimales successives comme pour un algorithme glouton à des sous problèmes liés par une relation de récurrence.

Principe d'optimalité

- La programmation dynamique s'applique à des problèmes d'optimisations : il s'agit souvent d'optimiser le coût d'une suite de décisions.
- Cette suite de décisions correspond à un découpage du problème en sous-problèmes :
  - On calcule les solutions optimales successives comme pour un algorithme glouton à des sous problèmes liés par une relation de récurrence.
  - Puis, c'est la combinaison de ces solutions qui produit la solution au problème initial.

Principe d'optimalité

- La programmation dynamique s'applique à des problèmes d'optimisations : il s'agit souvent d'optimiser le coût d'une suite de décisions.
- Cette suite de décisions correspond à un découpage du problème en sous-problèmes :
  - On calcule les solutions optimales successives comme pour un algorithme glouton à des sous problèmes liés par une relation de récurrence.
  - Puis, c'est la combinaison de ces solutions qui produit la solution au problème initial.
- *Principe d'optimalité de Bellman* : une solution optimale pour un problème présentant la propriété de sous-structure optimale est la combinaison de solutions optimales locales pour les sous-problèmes.

Programmation dynamique et graphes

- Un théorème général énonce que tout algorithme de programmation dynamique peut se ramener à la recherche du plus court chemin dans un graphe.

Programmation dynamique et graphes

- Un théorème général énonce que tout algorithme de programmation dynamique peut se ramener à la recherche du plus court chemin dans un graphe.
- Or, les techniques de recherche heuristique basées sur l'algorithme A* permettent d'exploiter les propriétés spécifiques d'un problème pour gagner en temps de calcul.

Programmation dynamique et graphes

- Un théorème général énonce que tout algorithme de programmation dynamique peut se ramener à la recherche du plus court chemin dans un graphe.
- Or, les techniques de recherche heuristique basées sur l'algorithme A* permettent d'exploiter les propriétés spécifiques d'un problème pour gagner en temps de calcul.
- Autrement dit, il est souvent plus avantageux d'exploiter un algorithme A* que d'utiliser la programmation dynamique.

Définition, première implémentation

- On appelle *suite de Fibonacci* toute suite réelle ou complexe $(f_n)_{n\in\mathbb{N}}$ récurrente d'ordre 2 définie par $f_{n+2} = f_{n+1} + f_n$ pour tout $n \in \mathbb{N}$. Souvent $f_0 = 0, f_1 = 1$, c'est ce que nous prendrons par la suite.

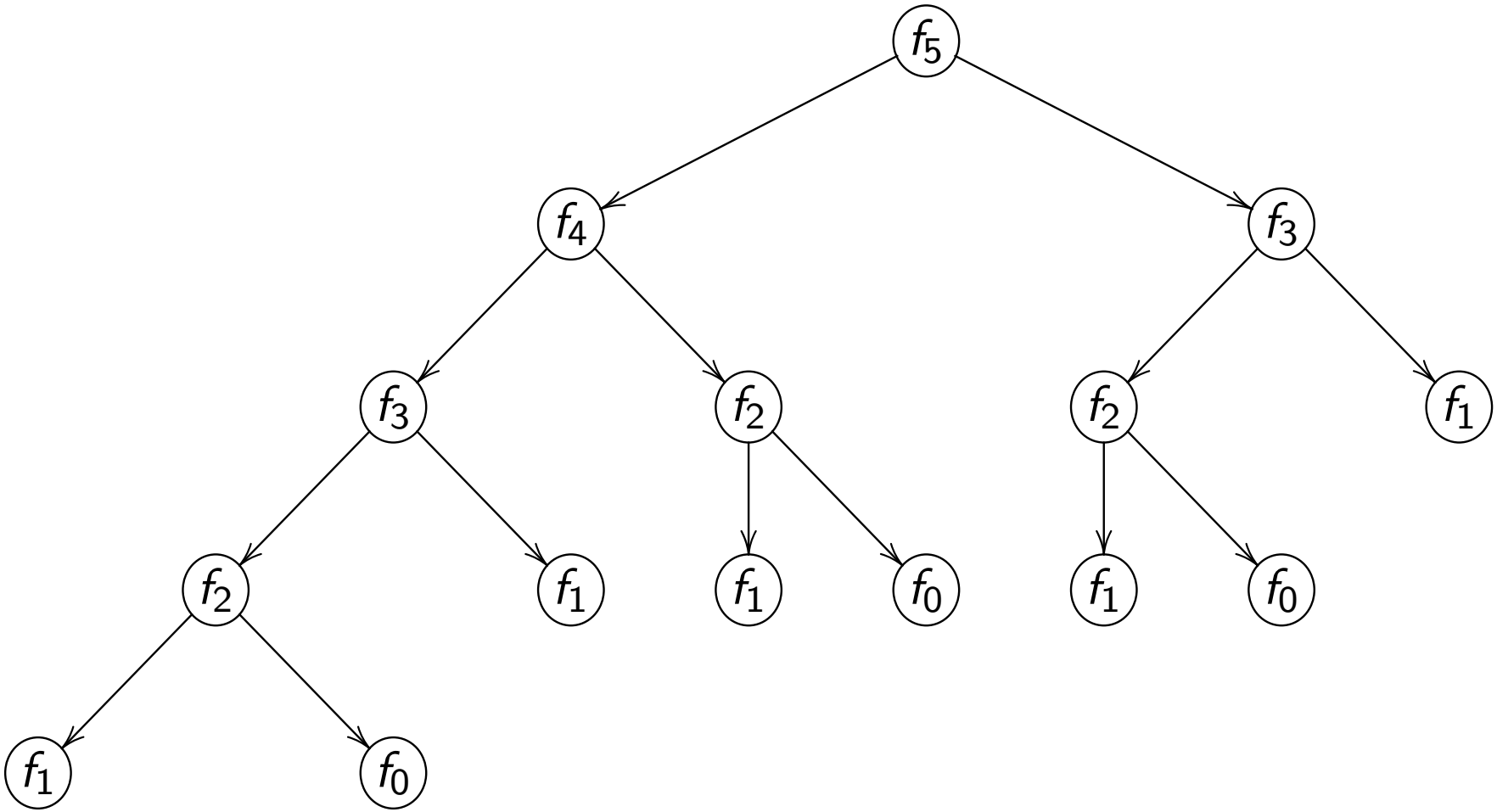
Définition, première implémentation

- On appelle *suite de Fibonacci* toute suite réelle ou complexe $(f_n)_{n\in\mathbb{N}}$ récurrente d'ordre 2 définie par $f_{n+2} = f_{n+1} + f_n$ pour tout $n \in \mathbb{N}$. Souvent $f_0 = 0, f_1 = 1$, c'est ce que nous prendrons par la suite.
- On peut alors proposer le code suivant :

```
1 def fibo_rec(n):
2     if n==0 or n==1:
3         return n
4     return fibo_rec(n-1)+fibo_rec(n-2)
```

Première implémentation

Calcul de f_5



On constate que f_2 est calculé 3 fois.

Première implémentation

Complexité

- Si $C(n)$ est la complexité pour calculer f_n, alors

$$C(n) = C(n-1) + C(n-2) + 1 \geq 2C(n-2)$$

en admettant que la complexité soit croissante

Première implémentation

Complexité

- Si $C(n)$ est la complexité pour calculer f_n, alors

$$C(n) = C(n-1) + C(n-2) + 1 \geq 2C(n-2)$$

en admettant que la complexité soit croissante

- On obtient que

$$C(n) \geq 2^{n/2} \underbrace{\max(C(0), C(1))}_{=1} = (\sqrt{2})^n$$

Complexité exponentielle.

Première implémentation

Complexité

- Si $C(n)$ est la complexité pour calculer f_n, alors

$$C(n) = C(n-1) + C(n-2) + 1 \geq 2C(n-2)$$

en admettant que la complexité soit croissante

- On obtient que

$$C(n) \geq 2^{n/2} \underbrace{\max(C(0), C(1))}_{=1} = (\sqrt{2})^n$$

Complexité exponentielle.

- De la même façon, on peut majorer $C(n)$ par 2^n. La complexité n'est pas « plus » qu'exponentielle.

Fibonacci : mémoïsation

Approche descendante

- **Mémoïsation** :

Fibonacci : mémoïsation

Approche descendante

- **Mémoïsation** :
  - On mémorise une valeur de la suite si c'est la première fois qu'on la rencontre.

Fibonacci : mémoïsation

Approche descendante

- **Mémoïsation** :
  - On mémorise une valeur de la suite si c'est la première fois qu'on la rencontre.
  - Si on a déjà rencontré le calcul courant, on récupère sa valeur par un accès à la structure de stockage en $O(1)$

Fibonacci : mémoïsation

Approche descendante

- **Mémoïsation** :
  - On mémorise une valeur de la suite si c'est la première fois qu'on la rencontre.
  - Si on a déjà rencontré le calcul courant, on récupère sa valeur par un accès à la structure de stockage en $O(1)$
  - On ne lance le calcul que si la valeur voulue n'a pas déjà été calculée.

Fibonacci : mémoïsation

Approche descendante

- **Mémoïsation** :
  - On mémorise une valeur de la suite si c'est la première fois qu'on la rencontre.
  - Si on a déjà rencontré le calcul courant, on récupère sa valeur par un accès à la structure de stockage en $O(1)$
  - On ne lance le calcul que si la valeur voulue n'a pas déjà été calculée.
- *Approche descendante* : On commence par lancer les calculs pour les valeurs de paramètres les plus grands. Ces calculs induisent des appels avec des paramètres plus petits.

Fibonacci : mémoïsation

Approche descendante

```
1 def fibo_mem(n):#fonction principale
2     d={0:0,1:1}#un dictionnaire est créé
3     return fibo_rec2(n,d)
4
5 def fibo_rec2(n,d):
6     """ Fonction auxiliaire : elle fait tout le boulot"""
7     if n in d:# pour éviter un calcul inutile
8         return d[n]
9     d[n] = fibo_rec2(n-1,d)+fibo_rec2(n-2,d)
10    return d[n]
```

- On crée un dictionnaire dans la fonction principale. Il contient les valeurs de la suite.

Fibonacci : mémoïsation

Approche descendante

```
1 def fibo_mem(n):#fonction principale
2     d={0:0,1:1}#un dictionnaire est créé
3     return fibo_rec2(n,d)
4
5 def fibo_rec2(n,d):
6     """ Fonction auxiliaire : elle fait tout le boulot"""
7     if n in d:# pour éviter un calcul inutile
8         return d[n]
9     d[n] = fibo_rec2(n-1,d)+fibo_rec2(n-2,d)
10    return d[n]
```

- On crée un dictionnaire dans la fonction principale. Il contient les valeurs de la suite.
- La fonction auxiliaire est appelée avec ce dictionnaire en paramètre.

Fibonacci : mémoïsation

Approche descendante

```
1 def fibo_mem(n):#fonction principale
2     d={0:0,1:1}#un dictionnaire est créé
3     return fibo_rec2(n,d)
4
5 def fibo_rec2(n,d):
6     """ Fonction auxiliaire : elle fait tout le boulot"""
7     if n in d:# pour éviter un calcul inutile
8         return d[n]
9     d[n] = fibo_rec2(n-1,d)+fibo_rec2(n-2,d)
10    return d[n]
```

- On crée un dictionnaire dans la fonction principale. Il contient les valeurs de la suite.
- La fonction auxiliaire est appelée avec ce dictionnaire en paramètre.
- La complexité devient temporelle linéaire en n (voir slide suivant). La complexité spatiale est aussi en $O(n)$.

Fibonacci : mémoïsation

Approche descendante

```
1 def fibo_mem(n):#fonction principale
2     return fibo_rec2(n,{0:0,1:1})
3
4 def fibo_rec2(n,d):#fonction auxiliaire
5     if n in d: return d[n]
6     d[n] = fibo_rec2(n-1,d)+fibo_rec2(n-2,d)
7     return d[n]
```

La complexité temporelle devient linéaire en n :

- On ne remplit la case `d[n]` qu'une fois ;

Fibonacci : mémoïsation

Approche descendante

```
1 def fibo_mem(n):#fonction principale
2     return fibo_rec2(n,{0:0,1:1})
3
4 def fibo_rec2(n,d):#fonction auxiliaire
5     if n in d: return d[n]
6     d[n] = fibo_rec2(n-1,d)+fibo_rec2(n-2,d)
7     return d[n]
```

La complexité temporelle devient linéaire en n :

- On ne remplit la case `d[n]` qu'une fois ;
- Alors le nombre d'appels à `fibo_rec2(i,d)`, pour une valeur $i \leq n$ est d'au plus 3 : une fois pour renseigner le dictionnaire, une fois quand on calcule `fibo_rec2(i+1,d)` et une fois quand on calcule `fibo_rec2(i+2,d)`.

Fibonacci : mémoïsation

Approche descendante

```
1 def fibo_mem(n):#fonction principale
2     return fibo_rec2(n,{0:0,1:1})
3
4 def fibo_rec2(n,d):#fonction auxiliaire
5     if n in d: return d[n]
6     d[n] = fibo_rec2(n-1,d)+fibo_rec2(n-2,d)
7     return d[n]
```

La complexité temporelle devient linéaire en n :

- On ne remplit la case `d[n]` qu'une fois ;
- Alors le nombre d'appels à `fibo_rec2(i,d)`, pour une valeur $i \leq n$ est d'au plus 3 : une fois pour renseigner le dictionnaire, une fois quand on calcule `fibo_rec2(i+1,d)` et une fois quand on calcule `fibo_rec2(i+2,d)`.
- Complexité spatiale $O(n)$: il faut tout stocker.

Fibonacci : mémoïsation

Approche ascendante

```
1 def fib_asc(n):
2     tab= [0,1]+[0]*(n-1) # rappel [0]*(-1) et [0]*0 valent []
3     for i in range(2,n+1):
4         tab[i]=tab[i-1]+tab[i-2]
5     return tab[n]
```

- On supprime la récursivité en la remplaçant par une boucle.On commence par résoudre les sous-problèmes et on les combine pour résoudre les plus gros problèmes.

Fibonacci : mémoïsation

Approche ascendante

```
1 def fib_asc(n):
2     tab= [0,1]+[0]*(n-1) # rappel [0]*(-1) et [0]*0 valent []
3     for i in range(2,n+1):
4         tab[i]=tab[i-1]+tab[i-2]
5     return tab[n]
```

- On supprime la récursivité en la remplaçant par une boucle.On commence par résoudre les sous-problèmes et on les combine pour résoudre les plus gros problèmes.
- Pas besoin de dictionnaire dans ce cas précis. Une liste suffit. Et on la crée à la bonne longueur (f_n nécessite de stocker n valeurs).

Fibonacci : mémoïsation

Approche ascendante

```
1 def fib_asc(n):
2     tab= [0,1]+[0]*(n-1) # rappel [0]*(-1) et [0]*0 valent []
3     for i in range(2,n+1):
4         tab[i]=tab[i-1]+tab[i-2]
5     return tab[n]
```

- On supprime la récursivité en la remplaçant par une boucle.On commence par résoudre les sous-problèmes et on les combine pour résoudre les plus gros problèmes.
- Pas besoin de dictionnaire dans ce cas précis. Une liste suffit. Et on la crée à la bonne longueur (f_n nécessite de stocker n valeurs).
- Complexité spatiale et temporelle $O(n)$

Fibonacci

Sans mémoïsation

```
1 def fibo(n):
2     u,v=0,1
3     for i in range(n):
4         u,v=v,u+v
5     return u
```

- On peut se demander pourquoi mémoïser dans la méthode ascendante.

Fibonacci

Sans mémoïsation

```
1 def fibo(n):
2     u,v=0,1
3     for i in range(n):
4         u,v=v,u+v
5     return u
```

- On peut se demander pourquoi mémoïser dans la méthode ascendante.
- En effet, si seule la dernière valeur nous intéresse, deux variables suffisent. La complexité spatiale passe en $O(1)$

Fibonacci

Sans mémoïsation

```
1 def fibo(n):
2     u,v=0,1
3     for i in range(n):
4         u,v=v,u+v
5     return u
```

- On peut se demander pourquoi mémoïser dans la méthode ascendante.
- En effet, si seule la dernière valeur nous intéresse, deux variables suffisent. La complexité spatiale passe en $O(1)$
- Dans ce cas précis, la mémoïsation dans la méthode itérative n'est intéressante que si on souhaite connaître toutes les valeurs de la suite jusqu'à f_n.

Présentation du problème

- On dispose d'un (multi) ensemble d'entiers positifs E.

Présentation du problème

- On dispose d'un (multi) ensemble d'entiers positifs E.
- On souhaite déterminer une partition de E en deux sous-ensembles E_1, E_2 tels que

Présentation du problème

- On dispose d'un (multi) ensemble d'entiers positifs E.
- On souhaite déterminer une partition de E en deux sous-ensembles E_1, E_2 tels que
  - $E_1 \cup E_2 = E; E_1 \cap E_2 = \emptyset$ (partition)

Présentation du problème

- On dispose d'un (multi) ensemble d'entiers positifs E.
- On souhaite déterminer une partition de E en deux sous-ensembles E_1, E_2 tels que
  - $E_1 \cup E_2 = E; E_1 \cap E_2 = \emptyset$ (partition)
  - La somme des éléments de E_1 et celle de E_2 sont les les plus proches possibles.

Approche gloutonne

Pour $E = \{e_1, \ldots, e_n\}$:

- On gère deux sous-ensembles E_1, E_2 initialisés resp. en $\{e_1\}, \emptyset$.

Exercice

Implanter cet algorithme. Donner sa complexité. Exhiber un exemple où la solution n'est pas optimale.

Approche gloutonne

Pour $E = \{e_1, \ldots, e_n\}$:

- On gère deux sous-ensembles E_1, E_2 initialisés resp. en $\{e_1\}, \emptyset$.
- On boucle sur $i = 2, \ldots, n$.

Exercice

Implanter cet algorithme. Donner sa complexité. Exhiber un exemple où la solution n'est pas optimale.

Approche gloutonne

Pour $E = \{e_1, \ldots, e_n\}$:

- On gère deux sous-ensembles E_1, E_2 initialisés resp. en $\{e_1\}, \emptyset$.
- On boucle sur $i = 2, \ldots, n$.
- On maintient l'invariant « À la fin du tour i, $\{e_1, \ldots, e_i\} \subset E_1 \cup E_2$ et $E_1 \cap E_2 = \emptyset$ ».
  L'élément courant e est ajouté à E_1 si, en ajoutant e à la somme des éléments de E_1 on obtient quelque chose de plus petit que la somme des éléments de e_2.

Exercice

Implanter cet algorithme. Donner sa complexité. Exhiber un exemple où la solution n'est pas optimale.

Approche gloutonne

Pour $E = \{e_1, \ldots, e_n\}$:

- On gère deux sous-ensembles E_1, E_2 initialisés resp. en $\{e_1\}, \emptyset$.
- On boucle sur $i = 2, \ldots, n$.
- On maintient l'invariant « À la fin du tour i, $\{e_1, \ldots, e_i\} \subset E_1 \cup E_2$ et $E_1 \cap E_2 = \emptyset$ ».
  L'élément courant e est ajouté à E_1 si, en ajoutant e à la somme des éléments de E_1 on obtient quelque chose de plus petit que la somme des éléments de e_2.
- Malheureusement, même en triant les éléments de E, la solution fournie n'est pas toujours optimale.

Exercice

Implanter cet algorithme. Donner sa complexité. Exhiber un exemple où la solution n'est pas optimale.

Algorithme basé sur la demi-somme

- On dispose d'un ensemble d'entiers positifs E.

Algorithme basé sur la demi-somme

- On dispose d'un ensemble d'entiers positifs E.
- Si E_1 et E_2 réalisent une partition équilibrée de E, alors

$$\left|\sum_{x\in E_1} x - \sum_{y\in E_2} y\right| =$$

$$\min\left(\left\{\left|\sum_{x\in E'} x - \sum_{y\in E''} y\right| \mid (E', E'') \text{ est une partition de } E\right\}\right)$$

Algorithme basé sur la demi-somme

- On dispose d'un ensemble d'entiers positifs E.
- Si E_1 et E_2 réalisent une partition équilibrée de E, alors

$$\left|\sum_{x\in E_1} x - \sum_{y\in E_2} y\right| =$$

$$\min\left(\left\{\left|\sum_{x\in E'} x - \sum_{y\in E''} y\right| \mid (E', E'') \text{ est une partition de } E\right\}\right)$$

- Soit S la somme des éléments de E. Si $A \subset E$ est tel que

$$\left|S/2 - \sum_{a\in A} a\right| = \min(\left\{\left|S/2 - \sum_{x\in X} x\right| \mid X \subset E\right\}),$$

alors $(A, E \setminus A)$ réalise une partition équilibrée de E.

Distance à la demi-somme : explication

- On a $S/2 = \lfloor \frac{S}{2} \rfloor$. Soit $A \subset E$ tel que :

$$\left| S/2 - \sum_{a \in A} a \right| = \min(\left\{ \left| S/2 - \sum_{x \in X} x \right| \mid X \subset E \right\}),$$

Distance à la demi-somme : explication

- On a $S/2 = \lfloor \frac{S}{2} \rfloor$. Soit $A \subset E$ tel que :

$$\left| S/2 - \sum_{a \in A} a \right| = \min(\left\{ \left| S/2 - \sum_{x \in X} x \right| \mid X \subset E \right\}),$$

- Supposons sans perte de généralité $S(A) \leq S/2$. Soit (F, G) partition de E telle que $S(F) \leq S(G)$. Alors $S(F) \leq S/2$ et $S(F) \leq S(A) \leq S/2$.

Distance à la demi-somme : explication

- On a $S/2 = \lfloor \frac{S}{2} \rfloor$. Soit $A \subset E$ tel que :

$$\left| S/2 - \sum_{a \in A} a \right| = \min(\left\{ \left| S/2 - \sum_{x \in X} x \right| \mid X \subset E \right\}),$$

- Supposons sans perte de généralité $S(A) \leq S/2$. Soit (F, G) partition de E telle que $S(F) \leq S(G)$. Alors $S(F) \leq S/2$ et $S(F) \leq S(A) \leq S/2$.
- On a

$$S(A) + S(E \setminus A) = S = \lfloor \frac{S}{2} \rfloor + \lceil \frac{S}{2} \rceil = S(F) + S(G)$$

Distance à la demi-somme : explication

- On a $S/2 = \lfloor \frac{S}{2} \rfloor$. Soit $A \subset E$ tel que :

$$\left| S/2 - \sum_{a \in A} a \right| = \min(\left\{ \left| S/2 - \sum_{x \in X} x \right| \mid X \subset E \right\}),$$

- Supposons sans perte de généralité $S(A) \leq S/2$. Soit (F, G) partition de E telle que $S(F) \leq S(G)$. Alors $S(F) \leq S/2$ et $S(F) \leq S(A) \leq S/2$.
- On a

$$S(A) + S(E \setminus A) = S = \lfloor \frac{S}{2} \rfloor + \lceil \frac{S}{2} \rceil = S(F) + S(G)$$

- Il vient :

$$0 \leq \lfloor \frac{S}{2} \rfloor - S(A) = -\lceil \frac{S}{2} \rceil + S(E \setminus A) \leq \lfloor \frac{S}{2} \rfloor - S(F) = -\lceil \frac{S}{2} \rceil + S(G)$$

Distance à la demi-somme : explication

- On a $S/2 = \lfloor \frac{S}{2} \rfloor$. Soit $A \subset E$ tel que :

$$\left| S/2 - \sum_{a \in A} a \right| = \min(\left\{ \left| S/2 - \sum_{x \in X} x \right| \mid X \subset E \right\}),$$

- Supposons sans perte de généralité $S(A) \leq S/2$. Soit (F, G) partition de E telle que $S(F) \leq S(G)$. Alors $S(F) \leq S/2$ et $S(F) \leq S(A) \leq S/2$.
- On a

$$S(A) + S(E \setminus A) = S = \lfloor \frac{S}{2} \rfloor + \lceil \frac{S}{2} \rceil = S(F) + S(G)$$

- Il vient :

$$0 \leq \lfloor \frac{S}{2} \rfloor - S(A) = -\lceil \frac{S}{2} \rceil + S(E \setminus A) \leq \lfloor \frac{S}{2} \rfloor - S(F) = -\lceil \frac{S}{2} \rceil + S(G)$$

- Alors $S(F) \leq S(A) \leq S(E \setminus A) \leq S(G)$. Et donc $|S(E \setminus A) - S(A)| \leq |S(G) - S(F)|$. $(A, E \setminus A)$: partition équilibrée.

Solution par programmation dynamique

Méthode descendante

- On cherche (E_1, E_2), partition équilibrée de E

Solution par programmation dynamique

Méthode descendante

- On cherche (E_1, E_2), partition équilibrée de E
- Le slide précédent suggère de travailler avec a) la demi-somme des éléments de E et b) l'ensemble E_1 (puisqu'on trouve alors E_2 facilement).

Solution par programmation dynamique

Méthode descendante

- On cherche (E_1, E_2), partition équilibrée de E
- Le slide précédent suggère de travailler avec a) la demi-somme des éléments de E et b) l'ensemble E_1 (puisqu'on trouve alors E_2 facilement).
- Agorithme récursif : On gère un ensemble E et la demi-somme $S/2$ des éléments de E. On cherche à construire E_1.
  Prendre $e \in E$ et calculer la distance de la somme des éléments de E_1 à $S/2$ dans 2 cas :

Solution par programmation dynamique

Méthode descendante

- On cherche (E_1, E_2), partition équilibrée de E
- Le slide précédent suggère de travailler avec a) la demi-somme des éléments de E et b) l'ensemble E_1 (puisqu'on trouve alors E_2 facilement).
- Agorithme récursif : On gère un ensemble E et la demi-somme $S/2$ des éléments de E. On cherche à construire E_1.
  Prendre $e \in E$ et calculer la distance de la somme des éléments de E_1 à $S/2$ dans 2 cas :
  - En mettant e dans E_1. Cela revient à ajouter e à la solution au problème obtenue lorsque $E = E \setminus \{e\}$ et $S/2 = S/2 - e$

Solution par programmation dynamique

Méthode descendante

- On cherche (E_1, E_2), partition équilibrée de E
- Le slide précédent suggère de travailler avec a) la demi-somme des éléments de E et b) l'ensemble E_1 (puisqu'on trouve alors E_2 facilement).
- Agorithme récursif : On gère un ensemble E et la demi-somme $S/2$ des éléments de E. On cherche à construire E_1.
  Prendre $e \in E$ et calculer la distance de la somme des éléments de E_1 à $S/2$ dans 2 cas :
  - En mettant e dans E_1. Cela revient à ajouter e à la solution au problème obtenue lorsque $E = E \setminus \{e\}$ et $S/2 = S/2 - e$
  - En ne mettant pas e dans E_1. On calcule la solution au problème obtenue lorsque $E = E \setminus \{e\}$ et $S/2$ est inchangé.

Solution par programmation dynamique

Méthode descendante

- On cherche (E_1, E_2), partition équilibrée de E
- Le slide précédent suggère de travailler avec a) la demi-somme des éléments de E et b) l'ensemble E_1 (puisqu'on trouve alors E_2 facilement).
- Agorithme récursif : On gère un ensemble E et la demi-somme $S/2$ des éléments de E. On cherche à construire E_1.
  Prendre $e \in E$ et calculer la distance de la somme des éléments de E_1 à $S/2$ dans 2 cas :
  - En mettant e dans E_1. Cela revient à ajouter e à la solution au problème obtenue lorsque $E = E \setminus \{e\}$ et $S/2 = S/2 - e$
  - En ne mettant pas e dans E_1. On calcule la solution au problème obtenue lorsque $E = E \setminus \{e\}$ et $S/2$ est inchangé.

  Choisir la meilleure option : celle qui améliore la distance de la somme des éléments de E_1 à la demi-somme $S/2$.

Solution par programmation dynamique

Méthode descendante

Comprenons la présentation précédente :

- On pose $S_{\frac{1}{2}}$ la demi-somme.

Solution par programmation dynamique

Méthode descendante

Comprenons la présentation précédente :

- On pose $S_{\frac{1}{2}}$ la demi-somme.
- On veut rendre minimum

$$A = |(\sum_{x \in E_1} x) - S/2|$$

Solution par programmation dynamique

Méthode descendante

Comprenons la présentation précédente :

- On pose $S_{\frac{1}{2}}$ la demi-somme.
- On veut rendre minimum

$$A = |(\sum_{x \in E_1} x) - S/2|$$

- Si $e \in E_1$ alors

$$A = |e + (\sum_{x \in E_1, x \neq e} x) - S/2| = |(\sum_{x \in E_1, x \neq e} x) - (S/2 - e)|$$

Solution par programmation dynamique

Méthode descendante

Comprenons la présentation précédente :

- On pose $S_{\frac{1}{2}}$ la demi-somme.
- On veut rendre minimum

$$A = |(\sum_{x \in E_1} x) - S/2|$$

- Si $e \in E_1$ alors

$$A = |e + (\sum_{x \in E_1, x \neq e} x) - S/2| = |(\sum_{x \in E_1, x \neq e} x) - (S/2 - e)|$$

- Si $e \notin E_1$ $A = |(\sum_{x \in E_1, x \neq e} x) - S/2|$

Solution par programmation dynamique

Méthode descendante

Écrire la fonction `def partition(E,S)` qui prend en paramètres un multi-ensemble d'entiers positifs E et la demi-somme (notée ici S par commodité). Elle renvoie le premier ensemble d'une partition équilibrée.

Solution par programmation dynamique

Méthode descendante

TODO Insérer le code ici.
À écrire...

Solution par programmation dynamique

Méthode ascendante avec tableau de bouléen

$E = \{e_0, \ldots, e_{n-1}\}$, multi-ensemble de nombres entiers positifs,
$S = \sum_{e \in E} e$.

- On construit une matrice de bouléens T de taille $(n+1) \times (S+1)$

Solution par programmation dynamique

Méthode ascendante avec tableau de bouléen

$E = \{e_0, \ldots, e_{n-1}\}$, multi-ensemble de nombres entiers positifs,
$S = \sum_{e \in E} e$.

- On construit une matrice de bouléens T de taille $(n+1) \times (S+1)$
- On se débrouille pour que le coefficient $T_{i,j}$ $(i \geq 0, j \geq 0)$ soit vrai si et seulement si il existe un sous-ensemble de $\{e_k \mid k \leq i-1\}$ dont la somme des éléments vaut j.

Solution par programmation dynamique

Méthode ascendante avec tableau de bouléen

$E = \{e_0, \ldots, e_{n-1}\}$, multi-ensemble de nombres entiers positifs,
$S = \sum_{e \in E} e$.

- On construit une matrice de bouléens T de taille $(n+1) \times (S+1)$
- On se débrouille pour que le coefficient $T_{i,j}$ $(i \geq 0, j \geq 0)$ soit vrai si et seulement si il existe un sous-ensemble de $\{e_k \mid k \leq i-1\}$ dont la somme des éléments vaut j.
- On cherche une relation de récurrence qui permette de construire $T_{i,j}$ connaissant les $T_{i',j'}$ pour $(i',j') < (i,j)$ au sens lexicographique.

Solution par programmation dynamique

Méthode ascendante avec tableau de bouléen

$E = \{e_0, \ldots, e_{n-1}\}$, multi-ensemble de nombres entiers positifs ($|E| = n$).

- Ligne 0 : Pour $k \geq 0$, $T_{0,k}$ désigne la possibilité pour que la somme des éléments de l'ensemble $\{e_k \mid k \leq 0 - 1\} = \emptyset$ vale k. Ainsi $T_{0,k}$ est faux sauf si $k = 0$.

Solution par programmation dynamique

Méthode ascendante avec tableau de bouléen

$E = \{e_0, \ldots, e_{n-1}\}$, multi-ensemble de nombres entiers positifs ($|E| = n$).

- Ligne 0 : Pour $k \geq 0$, $T_{0,k}$ désigne la possibilité pour que la somme des éléments de l'ensemble $\{e_k \mid k \leq 0 - 1\} = \emptyset$ vale k. Ainsi $T_{0,k}$ est faux sauf si $k = 0$.
- Pour $i \geq 0$, T_{i+1}, j est vrai si et seulement si il existe un sous-ensemble de $\{e_0, \ldots, e_i\}$ dont la somme des éléments vaut j. Ceci se décompose en :

Solution par programmation dynamique

Méthode ascendante avec tableau de bouléen

$E = \{e_0, \ldots, e_{n-1}\}$, multi-ensemble de nombres entiers positifs ($|E| = n$).

- Ligne 0 : Pour $k \geq 0$, $T_{0,k}$ désigne la possibilité pour que la somme des éléments de l'ensemble $\{e_k \mid k \leq 0 - 1\} = \emptyset$ vale k. Ainsi $T_{0,k}$ est faux sauf si $k = 0$.
- Pour $i \geq 0$, T_{i+1}, j est vrai si et seulement si il existe un sous-ensemble de $\{e_0, \ldots, e_i\}$ dont la somme des éléments vaut j. Ceci se décompose en :
  - Ou bien il existe un sous-ensemble de $\{e_0, \ldots, e_{i-1}\}$ dont la somme des éléments vaut j. Ceci est équivalent à « $T_{i,j}$ est vrai ».

Solution par programmation dynamique

Méthode ascendante avec tableau de bouléen

$E = \{e_0, \ldots, e_{n-1}\}$, multi-ensemble de nombres entiers positifs ($|E| = n$).

- Ligne 0 : Pour $k \geq 0$, $T_{0,k}$ désigne la possibilité pour que la somme des éléments de l'ensemble $\{e_k \mid k \leq 0 - 1\} = \emptyset$ vale k. Ainsi $T_{0,k}$ est faux sauf si $k = 0$.
- Pour $i \geq 0$, T_{i+1}, j est vrai si et seulement si il existe un sous-ensemble de $\{e_0, \ldots, e_i\}$ dont la somme des éléments vaut j. Ceci se décompose en :
  - Ou bien il existe un sous-ensemble de $\{e_0, \ldots, e_{i-1}\}$ dont la somme des éléments vaut j. Ceci est équivalent à « $T_{i,j}$ est vrai ».
  - Ou bien, il existe un sous-ensemble de $\{e_0, \ldots, e_{i-1}\}$ dont la somme des éléments vaut $j - e_i$ (chose impossible si $j < e_i$). Ceci est équivalent à « $T_{i,j-e_i}$ est vrai » lorsque $j \geq e_i$.

Solution par programmation dynamique

Méthode ascendante avec tableau de bouléen

$E = \{e_0, \ldots, e_{n-1}\}$, multi-ensemble de nombres entiers positifs ($|E| = n$).

- Ligne 0 : Pour $k \geq 0$, $T_{0,k}$ désigne la possibilité pour que la somme des éléments de l'ensemble $\{e_k \mid k \leq 0 - 1\} = \emptyset$ vale k. Ainsi $T_{0,k}$ est faux sauf si $k = 0$.
- Pour $i \geq 0$, T_{i+1}, j est vrai si et seulement si il existe un sous-ensemble de $\{e_0, \ldots, e_i\}$ dont la somme des éléments vaut j. Ceci se décompose en :
  - Ou bien il existe un sous-ensemble de $\{e_0, \ldots, e_{i-1}\}$ dont la somme des éléments vaut j. Ceci est équivalent à « $T_{i,j}$ est vrai ».
  - Ou bien, il existe un sous-ensemble de $\{e_0, \ldots, e_{i-1}\}$ dont la somme des éléments vaut $j - e_i$ (chose impossible si $j < e_i$). Ceci est équivalent à « $T_{i,j-e_i}$ est vrai » lorsque $j \geq e_i$.
- Relation de récurrence : pour $i \geq 1, j \geq 0$, $T_{i,j}$ est équivalent à :

$$T_{i-1,j} \text{ ou } (j \geq e_{i-1} \text{ et } T_{i-1,j-e_{i-1}})$$

Code : construction du tableau de bouléens

```
1 def tableau(E):
2     S=sum(E) # somme des éléments de $E$.
3     T=[[False for _ in range(S+1)]]# au départ, T a juste une ligne
4     T[0][0]=True #la somme des éléments de ens vide vaut 0
5     for i in range(len(E)):
6         NT = list(T[i]) #ligne i+1; condition 1 OK
7         for j in range(S+1-E[i]):
8             if T[i][j] : NT[j+E[i]]=True # cond.2 OK
9         T.append(NT)
10    m=S//2
11    while not T[len(E)][m]: m=m-1
12    return T,m
```

La valeur `m` retournée avec le tableau est la somme la plus proche de `S//2` inférieure ou égale à `S//2`

Code : explications

- La première partie implémente la relation de récurrence précédente : la ligne $i+1$ contient des `True` aux mêmes positions que la ligne i, mais elle en contient d'autres.

Code : explications

- La première partie implémente la relation de récurrence précédente : la ligne $i+1$ contient des `True` aux mêmes positions que la ligne i, mais elle en contient d'autres.
- `range` Ligne 8 : on s'assure que la somme du numéro de la colonne courante et de l'élément `E[i]` ne dépasse pas le nombre de colonnes. Observons que `T[i+1][E[i]]` est vrai (info : car `T[i][0]` est `True` ; maths : car $\sum_{e\in\{e_i\}} e = e_i$)

Code : explications

- La première partie implémente la relation de récurrence précédente : la ligne $i+1$ contient des `True` aux mêmes positions que la ligne i, mais elle en contient d'autres.
- `range` Ligne 8 : on s'assure que la somme du numéro de la colonne courante et de l'élément `E[i]` ne dépasse pas le nombre de colonnes. Observons que `T[i+1][E[i]]` est vrai (info : car `T[i][0]` est `True` ; maths : car $\sum_{e \in \{e_i\}} e = e_i$)
- Ligne 11 et 12 : trouver $m \leq \lfloor S/2 \rfloor$ tel qu'il existe un sous-ensemble de E dont la somme des éléments soit la plus proche possible de $\lfloor S/2 \rfloor$ (et inférieure à $\lfloor S/2 \rfloor$).
  Pour cela, on se place sur la ligne `len[E]` car les éléments considérés sont $e_0, \ldots, e_{|E|-1}$, c.a.d tous les éléments de E.

Construction de la partition équilibrée

Une fois trouvés le tableau de bouléens T et la somme m, on construit E_1 récursivement en lui ajoutant ou pas l'élément courant.

- On part de $T_{n,m}$ (qui est Vrai) et $E_1 = \emptyset$.

Construction de la partition équilibrée

Une fois trouvés le tableau de bouléens T et la somme m, on construit E_1 récursivement en lui ajoutant ou pas l'élément courant.

- On part de $T_{n,m}$ (qui est Vrai) et $E_1 = \emptyset$.
- On parcourt une suite $(T_{i,m_i})_{i=n,n-1,\dots 1}$ de coefficients avec $m_i \downarrow$ et $m_n = m$.
  C'est donc une suite dont les indices sont positifs et décroissants strictement au sens lexicographique, ce qui assure la terminaison de la récursion.

Construction de la partition équilibrée

Une fois trouvés le tableau de bouléens T et la somme m, on construit E_1 récursivement en lui ajoutant ou pas l'élément courant.

- On part de $T_{n,m}$ (qui est Vrai) et $E_1 = \emptyset$.
- On parcourt une suite $(T_{i,m_i})_{i=n,n-1,\ldots 1}$ de coefficients avec $m_i \downarrow$ et $m_n = m$.
  C'est donc une suite dont les indices sont positifs et décroissants strictement au sens lexicographique, ce qui assure la terminaison de la récursion.
- Invariant « T_{i,m_i} est vrai ». Critère de déplacement dans la matrice :

Construction de la partition équilibrée

Une fois trouvés le tableau de bouléens T et la somme m, on construit E_1 récursivement en lui ajoutant ou pas l'élément courant.

- On part de $T_{n,m}$ (qui est Vrai) et $E_1 = \emptyset$.
- On parcourt une suite $(T_{i,m_i})_{i=n,n-1,\ldots 1}$ de coefficients avec $m_i \downarrow$ et $m_n = m$.
  C'est donc une suite dont les indices sont positifs et décroissants strictement au sens lexicographique, ce qui assure la terminaison de la récursion.
- Invariant « T_{i,m_i} est vrai ». Critère de déplacement dans la matrice :
  - Si T_{i-1,m_i} est vrai, alors on peut trouver un sous-ensemble de $\{e_0, \ldots, e_{i-2}\}$ qui a pour somme m_i. Donc E_1 peut ne pas contenir e_{i-1} : il reste inchangé.

Construction de la partition équilibrée

Une fois trouvés le tableau de bouléens T et la somme m, on construit E_1 récursivement en lui ajoutant ou pas l'élément courant.

- On part de $T_{n,m}$ (qui est Vrai) et $E_1 = \emptyset$.
- On parcourt une suite $(T_{i,m_i})_{i=n,n-1,\ldots 1}$ de coefficients avec $m_i \downarrow$ et $m_n = m$.
  C'est donc une suite dont les indices sont positifs et décroissants strictement au sens lexicographique, ce qui assure la terminaison de la récursion.
- Invariant « T_{i,m_i} est vrai ». Critère de déplacement dans la matrice :
  - Si T_{i-1,m_i} est vrai, alors on peut trouver un sous-ensemble de $\{e_0, \ldots, e_{i-2}\}$ qui a pour somme m_i. Donc E_1 peut ne pas contenir e_{i-1} : il reste inchangé.
  - Sinon c'est que $T_{i-1,m_i-e_{i-1}}$ est vrai. On peut trouver un sous-ensemble de $\{e_0, \ldots, e_{i-2}\}$ qui a pour somme $m - e_{i-1}$. On ajoute donc e_{i-1} à E_1.

Construction de la partition équilibrée

Code

```
1  def partition(E,T,i,s,E1):
2      """
3      E : ensemble étudié
4      T : matrice de bouléens
5      i : ligne courante (numéro d'un élément de E)
6      s : somme courante
7      E1 : ens. construit (la partition équilibrée est E1, E\E1)
8      """
9      if i==0:
10         return E1
11     if T[i-1][s]:
12         return partition(E,T,i-1,s,E1)
13     elif T[i-1][s-E[i-1]]:#elif inutile mais pédagogique
14         E1.append(E[i-1])
15         return partition(E,T,i-1,s-E[i-1],E1)
```

Fin de la récursion : lorsqu'on a atteint la ligne 0 (correspond à $E = \emptyset$).

Initialisation

Initialisation. Connaissant `T,m`, on appelle `partition(E,T,len(E),m,[])`. N'oublions pas que `T[len(E)][m]` est `True`.

```
1 def Partition(E):
2     T,m=tableau(E)
3     return partition(E,T,len(E),m,[])
```