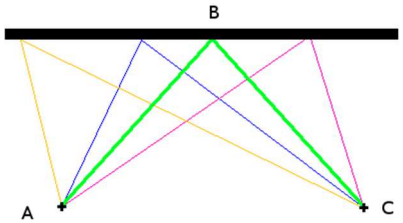
Programmation dynamique

Les problèmes d'optimisation

Quel est le plus court chemin (i.e. le plus rapide en physique) pour aller de A à B en rebondissant sur le mur ?



Réponse : il faut que l'angle d'incidence soit égal à l'angle de réflexion (loi de la réflexion de Snell-Descartes).
Cette question est un ***problème d'optimisation*** du temps de parcours (la loi de la réfraction est également un problème d'optimisation du temps de parcours).

L'optimisation est une branche des mathématiques concernant les problèmes qui consistent à ***minimiser ou maximiser une fonction sur un ensemble***.

Les premiers problèmes d'optimisation auraient été formulés par Euclide, au III^e^ siècle avant notre ère, dans son ouvrage historique *Éléments*.

D'après Wikipédia

Il existe plusieurs stratégies pour tenter de résoudre les ***problèmes d'optimisation*** :

- ✓ les ***algorithmes gloutons*** (pcsi) ;
- ✓ la ***programmation dynamique*** (pc) ;
- ✓ … (domaine très vaste)

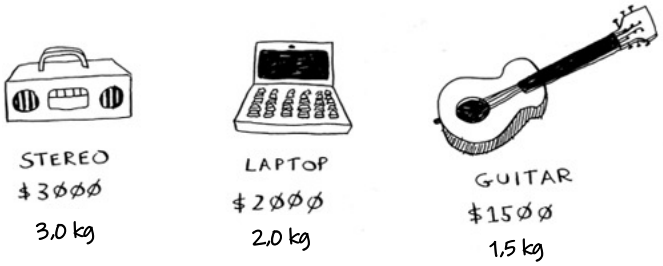
The knapsack problem

D'après grokking algorithms Aditya Y. Bhargava. Ed. Manning

Le problème du sac à dos - Stratégie gloutonne

Suppose you're a greedy thief. You're in a store with a knapsack, and there are all these items you can steal.
But you can only take what you can fit in tour knapsack. The knapsack can hold 3,5 kg.

You're trying to maximize the value of the items you put in your knapsack.
What algorithm do you use?



The greedy strategy is pretty simple:

1. Pick the most expensive thing that will fit in your knapsack.
2. Pick the next most expensive thing that will fit in your knapsack. And so on.

Exercice 1 :

L'algorithme glouton donne-t-il la solution optimale ?

Conclusion

La stratégie gloutonne ne fournit pas toujours la solution optimale mais elle peut fournir une solution proche de la solution optimale dont on peut parfois se contenter parce que c'est un algorithme simple et rapide.
La solution optimale est fournie par la programmation dynamique.

Un ***algorithme glouton*** (ou gourmand ou goulu) est un algorithme qui suit le principe de réaliser, ***étape par étape, un choix optimum local, afin d'obtenir un résultat optimum global***.
La stratégie gloutonne consiste à effectuer *un* choix local pour résoudre le problème global.

Dans les cas où l'algorithme ne fournit pas systématiquement la solution optimale, il est appelé une ***heuristique*** gloutonne.

Par exemple, dans le problème du rendu de monnaie (donner une somme avec le moins possible de pièces, *problème d'optimisation*), l'algorithme consistant à répéter le choix de la pièce de plus grande valeur (*optimum local*) qui ne dépasse pas la somme restante est un algorithme glouton.
Suivant le système de pièces, l'algorithme glouton est optimal ou pas (cf. exercice 2).

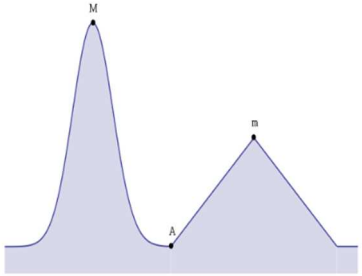
***Exercice* 2-1** - Rendu de monnaie

1. Dans le système de pièces européen (en centimes : 1, 2, 5, 10, 20, 50, 100, 200), quelle somme l'algorithme glouton donne-t-il pour 37 centimes ? (i.e. quelles pièces pour former cette somme ?)

Dans ce système de pièces, on peut montrer que l'algorithme glouton donne toujours une solution optimale.

2. Justifier que, dans le système de pièces (1, 3, 4), l'algorithme glouton n'est pas optimal pour 6 centimes.

L'illustration ci-contre montre également un cas où le choix optimum local ne conduit pas au résultat optimum global.



***Exercice* 2-2** – Recherche de maximum

En partant du point A et en cherchant à monter selon la plus forte pente (*optimum local*), quel maximum un algorithme glouton trouvera-t-il ?

Une ***heuristique*** est une méthode de calcul qui fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte, pour un problème d'optimisation difficile.
Heuristique vient du grec trouver, trouver par hasard, découvrir, imaginer qui a donné eurêka : j'ai trouvé !

C'est un concept utilisé entre autres en optimisation combinatoire, en théorie des graphes, en théorie de la complexité des algorithmes, en intelligence artificielle, dans la programmation des jeux (comme les échecs ou go), dans la primalité des nombres entiers et dans la démonstration de théorèmes.
Une heuristique s'impose quand les algorithmes de résolution exacte sont impraticables, à savoir de complexité polynomiale de haut degré, exponentielle ou plus.
Généralement, une heuristique est conçue pour un problème particulier, en s'appuyant sur sa structure propre, mais il existe des approches fondées sur des principes généraux.

D'après Wikipédia

The knapsack problem

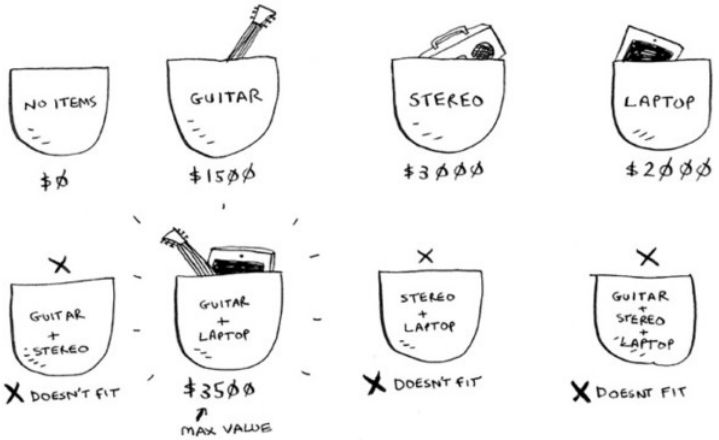
D'après grokking algorithms Aditya Y. Bhargava. Ed. Manning

Le problème du sac à dos – Introduction à la programmation dynamique

You are a thief with a knapsack that can carry 4 lb of goods.
You have three items that you can put into the knapsack.

What items should you steal so that you steal the maximum money's worth of goods?

The simplest algorithm is this: you try every possible set of goods and find the set that gives you the most value.



Cette stratégie, dite de ***force brute***, fonctionne mais elle est lente. Pour seulement 3 objets, il faut envisager 8 configurations possibles. Pour 4 objets, le nombre de configurations est 16. Le nombre de configurations double si on ajoute un objet, la complexité est $O(2^n)$ avec n objets.

La programmation dynamique va permettre de résoudre le problème plus rapidement.

La stratégie consiste à résoudre des sous-problèmes de taille croissante jusqu'à résoudre le problème global.

***Un tableau permet d'illustrer un algorithme de programmation dynamique*.**

Colonnes = poids du sac à dos de 1 à 4 lbs

Poids max du sac 4 lbs

Une ligne par objet à choisir parmi

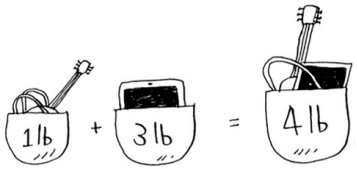
	1 lbs	2 lbs	3 lbs	4 lbs
Guitar 1 lbs $ 1500	G $1500			
Stereo 4 lbs $ 3000				
Laptop 3 lbs $ 2000				

***Exercice* 3-1** :

Remplir les 3 premières lignes du tableau pas à pas : ligne par ligne et de gauche à droite, sur le modèle de la première cellule, en gardant à l'esprit qu'à chaque ligne, on peut voler l'objet de cette ligne ou/et un objet des lignes précédentes ; explications ci-dessous :

- ✓ Ligne de la guitare
  Pour chaque cellule de la ligne, la seule question à se poser est : est-ce que je vole la guitare ou pas ? Dans cette ligne, il n'y a rien d'autre à voler…
- ✓ Ligne de la chaîne stéréo
  Pour les cellules de cette ligne, il est possible de voler la guitare **ou/et** la chaîne (en respectant la contrainte sur le poids total).
- ✓ Ligne de l'ordinateur portable
  Il est possible de voler une combinaison quelconque d'objets qui respecte la contrainte sur le poids total, la dernière cellule donne la réponse.

Le cas de la dernière cellule est particulièrement intéressant : la solution pour un sac de 4 livres = 1 + 3 livres combine ***le meilleur choix pour un sac de 1 livre*** (la guitare) et ***le meilleur choix pour un sac de 3 livres*** (l'ordinateur).



A ce stade, il peut sembler que le raisonnement utilisé pour compléter la dernière cellule diffère du raisonnement utilisé pour remplir les autres cellules : reprendre le meilleur résultat précédent (flèches vertes) ou choisir le nouvel objet (texte en bleu).

En réalité, il existe une formule qui permet de remplir toutes les cases à condition d'ajouter une première ligne et une première colonne remplies de 0 (cf. tableau ci-dessous).
Notons :

- v_i et w_i la valeur et le poids de l'objet à choisir, ou non, dans la ligne i ;
- W_j la capacité maximale du sac dans la colonne j (en réalité, j et W_j désignent la même chose : à la fois l'index de la colonne du tableau et la capacité du sac pour cette colonne) ;
- $v_{max}[i][j]$ la valeur optimale calculée pour la cellule ligne i colonne j.

$$v_{max}[i][W_j] = \begin{cases} \max\left(\underset{v_{max}\text{ pour objets précédents}}{v_{max}[i-1][W_j]} \;,\; v_i + \underset{v_{max}\text{ pour le poids disponible restant}}{v_{max}[i-1][W_j - w_i]} \right) & \text{si } w_i \le W_j \\ v_{max}[i-1][W_j] & \text{si } w_i > W_j \end{cases}$$

***Exercice* 3-2** :

Vérifier cet algorithme.

		1 lbs	2 lbs	3 lbs	4 lbs
∅	0	0	0	0	0
Guitar 1 lbs $ 1500	0 $0	G $1500			
Stereo 4 lbs $ 3000	0 $0				
Laptop 3 lbs $ 2000	0 $0				
	0 $0				

***Exercice* 4** :

Que se passe-t-il si on ajoute un objet (iphone 1lb $2000) ?
Remplir la dernière ligne du tableau ci-dessus.
Réponse : il suffit d'ajouter une ligne au tableau précédent (seule la dernière ligne est à recalculer) → $ 4000 = Laptop + iphone

***Exercice* 5** :

Vérifier que modifier l'ordre des choix (l'inversion de deux lignes) ne modifie pas le résultat final.

Remarque

On constate sur l'algorithme de cet exemple que la solution pour la cellule $[i][j]$ nécessite de connaître la solution pour les cellules $[i\text{-}1][j]$ et $[i\text{-}1][j\text{-}p]$ (p étant ici le poids de l'objet exprimé en livres).
Mais le calcul pour la cellule $[i\text{-}1][j\text{-}p]$ a elle aussi nécessité le calcul de la cellule $[i\text{-}1][j]$: on dit que ***les problèmes se chevauchent*** (le même problème est résolu plusieurs fois sauf si on stocke les résultats précédents comme dans le tableau utilisé).
Le ***tableau*** construit permet de mettre à jour un ***graphe de dépendance*** permettant d'identifier les sous-problèmes liés entre eux.
Cette situation a été rencontrée en première année dans le calcul des termes de la suite de Fibonacci (cf. ci-dessous, page 4).

Programmation dynamique – Dynamic programming

Programmation dynamique

En informatique, la ***programmation dynamique*** est une méthode algorithmique pour résoudre des ***problèmes d'optimisation***.
La programmation dynamique consiste à résoudre un problème en le ***décomposant en sous-problèmes***, puis à résoudre les sous-problèmes, ***des plus petits aux plus grands en stockant les résultats intermédiaires***.
La programmation dynamique se caractérise par la résolution, par taille croissante, de *tous* les problèmes locaux alors que la programmation gloutonne effectue *un* choix local.

Le concept a été introduit au début des années 1950 par Richard Bellman. À l'époque, le terme « programmation » signifie *planification* et *ordonnancement*.

Elle a d'emblée connu un grand succès, car de nombreuses fonctions économiques de l'industrie étaient de ce type, comme la conduite et l'optimisation de procédés chimiques, ou la gestion de stocks.

D'après Wikipédia

Solution optimale

Cette méthode garantit d'obtenir la ***solution optimale*** mais sa complexité temporelle est parfois trop importante pour pouvoir être utilisée en pratique. Dans ce cas, on se résout à utiliser la programmation gloutonne.

Exemple : la méthode utilisée dans l'exercice 3 permet de trouver la solution optimale.

Sous-structure optimale

On dit qu'un problème possède la propriété de ***sous-structure optimale*** si une solution optimale peut être trouvée en le découpant en sous-problèmes dont on obtient *récursivement* les solutions optimales.
On parle du ***principe d'optimalité de Bellman***.

Exemple : algorithme précédent (cf. explications sur la détermination de la valeur optimale dans la dernière case du tableau).

Chevauchement des sous-problèmes

On dit que ***les sous-problèmes se chevauchent*** si, au cours de l'algorithme, certains sous-problèmes ne sont pas indépendants et doivent être résolus plusieurs fois (la récursivité les résout plusieurs fois).

Afin d'éviter ces répétitions différentes stratégies sont utilisées (cf. paragraphe suivant).

Programmation dynamique vs méthode dichotomique

Les méthodes « ***diviser pour régner*** », ***méthode dichotomique*** et ***programmation dynamique,*** consistent à se ramener à la résolution de ***sous-problèmes de tailles inférieures***.
Mais à la différence de la méthode dichotomique qui nécessite que les sous-problèmes soient indépendants, en programmation dynamique ces sous-problèmes ne sont pas indépendants, on parle de ***chevauchement de sous-problèmes***.

Stratégies en programmation dynamique

Cette partie décrit les principes de la programmation dynamique à l'aide d'un exemple.

- Détermination des sous-problèmes et de leurs relations.
- La ***mémoïsation*** (de l'anglais memoïzation) consiste à stocker les solutions des sous-problèmes (amélioration de la complexité temporelle au détriment de la complexité spatiale). Cette mémorisation utilise parfois un dictionnaire.
- ***Approche descendante*** (top-down) ***récursive*** avec mémoïsation.
- ***Approche ascendante*** (bottom-up) ***itérative*** avec stockage des résultats.

Exemple – Suite de Fibonacci

L'exemple utilisé est le calcul des termes de la suite de Fibonacci ; il s'agit d'une illustration car ces techniques ne sont pas nécessaires pour calculer efficacement les termes de cette suite.

On considère la suite de Fibonacci définie par $\begin{cases} F_0 = 1 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad n \geq 2 \end{cases}$

Sous-problèmes

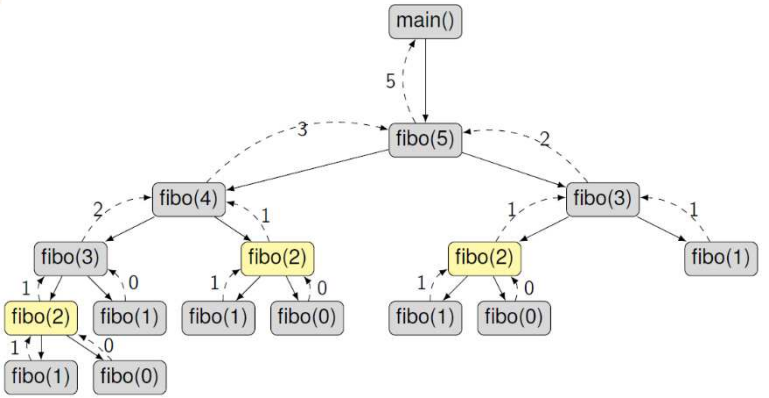
Le problème se ramène aux calculs de F_{n-1} et F_{n-2} mais ces deux sous-problèmes ne sont pas indépendants.

Exercice 6 :

Ecrire une fonction récursive *fibo*(*n*) « naïve » découlant directement de la définition de la suite.

```
1  def fibo(n):
2      if n <= 1: return n
3      return fibo(n-1) + fibo(n-2)
```

L'arbre des appels pour calculer F_5 illustre le chevauchement des sous-problèmes :



La fonction *fibo*(*n*) ainsi écrite calculerait 2 fois *fibo*(*3*), 3 fois *fibo*(*2*).
Si C(*n*) est le coût en temps pour calculer F_n alors C(*n*) = C(*n*-1) + C(*n*-2) + O(1).
D'où C(*n*) ≥ 2 C(*n*-2) : la complexité est exponentielle.

Approche descendante récursive avec mémoïsation (à l'aide d'un dictionnaire)

Préliminaires

Exercice 7-1 : utilisation d'un objet mutable comme paramètre par défaut

Faire tourner la fonction suivante « à la main » : indiquer à chaque étape le contenu de la liste L si on exécute *test*(5) en complétant le tableau ci-dessous.

```
1  def test(n, L=[]):
2      if n==0: return L
3      L += [n]
4      return test(n-1)   # Ou test(n-1, L)
```

n	Exécution	L à l'étape *n*
5	test(5)	
4		
3		
2		
1		
0		

La liste L, utilisée ici pour stocker des informations (les valeurs successives de *n*), est transmise à chaque appel à la fonction *test* : cette liste est disponible et mise à jour d'appel en appel.

Il s'agit d'une ***technique de mémoïsation***.

Exercice 7-2 - ***Récursion avec mémoïsation à l'aide d'un dictionnaire***

Compléter le code de la fonction récursive *fibo_mem*(*n*, *memo*={0: 1, 1: 1}) renvoyant le terme d'ordre *n* de la suite de Fibonacci ; l'algorithme est le suivant : la fonction teste la présence d'un terme déjà calculé dans le dictionnaire avant d'entamer un nouveau calcul puis le dictionnaire *memo* est mis à jour si nécessaire.

Rappel : les clés d'un dictionnaire peuvent être des entiers, des flottants… Ici, le dictionnaire utilise des clés entières (les valeurs de *n*).

```
1  def fibo_mem(n, memo={0:1, 1:1}):
2      if
3          return
4      memo[n] =
5      return
```

Exercice 8 : complexité

1. Rappeler la complexité d'une opération élémentaire quelconque sur un dictionnaire.
2. En déduire la complexité de la fonction *fibo_mem*.

Approche ascendante itérative avec stockage dans un dictionnaire

Exercice 9 - ***Itération et stockage dans un dictionnaire***

Compléter le code de la fonction itérative *fibo_asc*(*n*) renvoyant le terme d'ordre *n* de la suite de Fibonacci : les valeurs successives sont calculées dans l'ordre croissant et sont stockées dans un dictionnaire.

```
1  def fibo_asc(n):
2      memo = {0:1, 1:1}
3      for k in range(  ,    ):
4
5      return
```

Exercice 10 : itération et stockage dans une liste

Ecrire une fonction analogue *fibo_asc2*(*n*) utilisant une liste au lieu d'un dictionnaire.

```
1  def fibo_asc2(n):
2      L =
3      for k in range(  ,    ):
4
5      return
```

Remarque – Algorithme le plus économique

Comme annoncé, dans le cas de la suite de Fibonacci, les algorithmes précédents ne sont pas les plus efficaces mais ils permettent d'illustrer les concepts de la programmation dynamique.

Il suffit en réalité de mémoriser les deux derniers termes de la suite et non tous les termes.

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

Conclusion – Méthodes dynamiques sur l'exemple de la suite de Fibonacci

- ✓ Sous-problèmes : F_{k-1} et F_{k-2}
- ✓ Mise en évidence du chevauchement des sous-problèmes avec un tableau ou un arbre et mise en évidence des relations entre solutions : $F_k = F_{k-1} + F_{k-2}$
- ✓ Approche descendante récursive avec mémoïsation ou approche ascendante itérative avec stockage des solutions intermédiaires au cours de la boucle.

Approche descendante récursive avec mémoïsation – Critiques et perfectionnements

Dans l'exercice 7, l'utilisation d'un paramètre mutable est ***peu intuitive*** (code rappelé ci-dessous).

```
1  def fibo_mem(n, memo={0:1, 1:1}):
2      if n in memo:
3          return memo[n]
4      memo[n] = fibo_mem(n-1) + fibo_mem(n-2)
5      return memo[n]
```

Il peut sembler préférable d'utiliser une ***variable*** et non un ***paramètre*** cependant cette variable ne peut pas être interne à la fonction (une variable interne n'existe que dans la fonction), elle doit donc être externe.

```
1  memo = {0: 1, 1: 1}      # Création du dictionnaire
2  def fibo_mem(n):         # Fonction avec un seul paramètre
3      global memo          # Déclaration facultative
4      if not n in memo:
5          memo[n] = fibo_mem(n - 1) + fibo_mem(n - 2)
6      return memo[n]
```

Toutefois, l'utilisation d'une variable globale pour le fonctionnement ***interne*** d'une fonction est à proscrire : une fonction doit pouvoir être exécutée sans nécessiter de variables à initialiser « à la main ».
Il suffit d'***encapsuler*** le code précédent à l'intérieur d'une nouvelle fonction : ci-dessous, la fonction principale fibo_mem se contente de créer le dictionnaire nécessaire à la mémoïsation et d'appeler la fonction fibo_mem_aux qui effectue le travail.

```
1  def fibo_mem(n):
2      memo = {0: 1, 1: 1}      # Dictionnaire pour mémoïsation
3
4      def fibo_mem_aux(n):     # Fonction effectuant le travail
5          if not n in memo:
6              memo[n] = fibo_mem_aux(n-1) + fibo_mem_aux(n-2)
7          return memo[n]       # Renvoie résultat à fct ppale
8
9      return fibo_mem_aux(n)   # Appel de la fonction auxiliaire
```

L'inconvénient est un code ***un peu plus long mais plus clair*** (et qui permet d'éviter des problèmes subtils liés à l'utilisation de paramètres mutables).