MARO 037 - Competitive Programming

Fabian Lecron - Arnaud Vandaele

Faculté Polytechnique - Services MIT et MARO

UMONS

Année académique 2024-2025

Séance 04 :
Greedy methods et Dynamic Programming

Séance 04 – Greedy methods et Dynamic Programming

Tables des matières

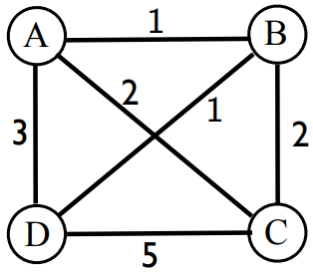
Plan

Greedy methods (algorithme glouton)

Un algorithme est qualifié de *glouton* lorsqu'il implémente l'idée suivante pour résoudre un problème donné :

A chaque étape de la construction de la solution, l'algorithme effectue le choix 'glouton', c'est-à-dire celui qui semble le meilleur tout de suite.

Exemple avec un problème de TSP :



Un algorithme greedy est le suivant :

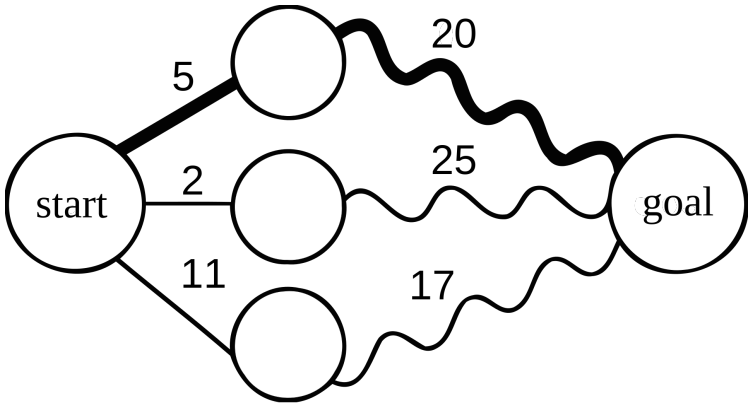
tant qu'on n'a pas un tour, on se dirige vers la ville non visitée la plus proche.

- Algorithme ayant une complexité $\mathcal{O}(n)$
- Solution obtenue avec cet algorithme : A - B - D - C - A avec un coût $= 9$
- Solution optimale : A - C - B - D - A avec un coût $= 8$

Greedy methods - Propriétés

Les algorithmes gloutons produisent les solutions optimales sur certains problèmes, mais pas sur d'autres. La plupart des problèmes pour lesquels ils fonctionnent de façon optimale possèdent deux propriétés :

- **propriété des choix gloutons optimaux :** le choix effectué par un algorithme glouton à une étape k dépend des choix faits lors des étapes $1, ..., k$ mais ne dépend pas des choix futurs. Il fait itérativement un choix glouton l'un après l'autre, réduisant chaque problème donné en un plus petit. En d'autres termes, un algorithme gourmand ne reconsidère jamais ses choix.
- **propriété de sous-structure optimale** : une solution optimale du problème est composée de solutions optimales aux sous-problèmes



Même si ces propriétés ne sont pas satisfaites, l'approche gloutonne conduit souvent à des solutions intéressantes avec une complexité algorithmique faible.

Exemple 1 : Knapsack problem (sac à dos)

Un premier exemple que vous connaissez déjà (normalement)

Soit une valeur $W > 0$ et n objets ayant un poids $w_i > 0$ et une valeur $v_i > 0$.
Variables : $x_i = 1$ si on prend l'objet i, 0 sinon

$$\max_{x_i \in \{0,1\}} \quad \sum_{i=1}^{n} v_i x_i$$
$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq W$$

Exemple : (avec $W = 11$)

i	1	2	3	4	5
v_i	1	6	18	22	28
w_i	1	2	5	6	7

Algorithme glouton, est-ce que ça fonctionne ? Non!

- Algorithme greedy : choisir les objets avec la plus grande valeur v_i
- Solution greedy : $x_5 = 1$, $x_2 = 1$, $x_1 = 1$ et $x_3 = x_4 = 0$
  $\rightarrow$ valeur du sac $= 35$
- Solution optimale : $x_4 = 1$, $x_3 = 1$, et $x_1 = x_2 = x_5 = 0$
  $\rightarrow$ valeur du sac $= 40$

Exemple 1 : fractional knapsack

Soit une valeur $W > 0$ et n objets ayant un poids $w_i > 0$ et une valeur $v_i > 0$.
Variables : $x_i \in [0, 1]$ (on peut donc prendre des fractions d'un objet)

$$\max_{x_i \in [0,1]} \quad \sum_{i=1}^{n} v_i x_i$$
$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq W$$

Exemple : (avec $W = 11$)

i	1	2	3	4	5
v_i	1	6	18	22	28
w_i	1	2	5	6	7
$\frac{v_i}{w_i}$	1	3	3.6	3.7	4

Algorithme greedy, est-ce que ça fonctionne ? Oui!

- Algorithme greedy :
  choisir, tant que c'est possible, les objets avec le plus grand ratio $\frac{v_i}{w_i}$
- Solution greedy : $x_5 = 1$, $x_4 = \frac{2}{3}$ et $x_3 = 0$, $x_2 = 0$, $x_1 = 0$
  $\rightarrow$ valeur du sac $= 42.66$

Exemple 1 : preuve de correction pour le fractional knapsack

Théorème.

Le problème fractionnel du sac à dos possède la propriété des choix gloutons optimaux.

Preuve (par contradiction).

Soit la solution optimale $(x_1, x_2, ..., x_n)$ pour laquelle on a un objet i tel que $x_i = 0$ et un objet j tel que $x_j > 0$ vérifiant

$$\frac{v_i}{w_i} > \frac{v_j}{w_j}.$$

- On transforme cette solution $(x_1, x_2, ..., x_n)$ en $(x'_1, x'_2, ..., x'_n)$:
  - $x'_k = x_k$ pour tout $k \in \{1, ..., n\} \backslash \{i, j\}$,
  - $x'_i = x_i + \frac{\Delta}{w_i}$, et
  - $x'_j = x_j - \frac{\Delta}{w_j}$,

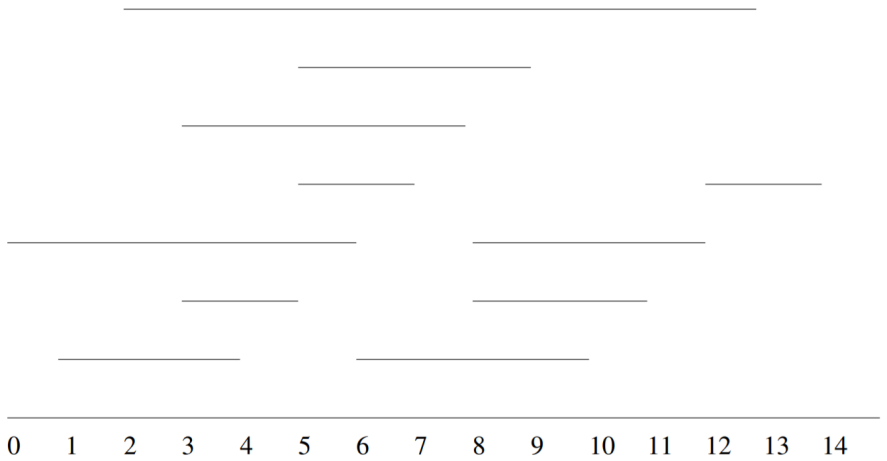
  avec $\Delta = \min(w_i(1 - x_i), w_j x_j)$.
- Cette transformation ne modifie pas le poids total, mais améliore l'objectif!
- On en déduit que $(x_1, x_2, ..., x_n)$ n'était pas optimale et qu'il est toujours avantageux de prendre la fraction maximale de l'objet i possédant le plus grand rapport $\frac{v_i}{w_i}$.

Exemple 2 : sélection d'activités

Entre les slots 0 et 14, un terrain de sport peut être réservé par une seule personne à la fois, voici les demandes (s_i, f_i) de réservation :

$$(1,4),(3,5),(0,6),(5,7),(3,8),(5,9),(6,10),(8,11),(8,12),(2,13),(12,14)$$

Quelles réservations choisir afin de maximiser le nombre de demandes acceptées ?



Algorithme greedy, est-ce que ça fonctionne ? Oui!

- Trier les demandes par valeurs croissantes de f_i et sélectionner les activités compatibles avec celles déjà prises
- Solution greedy : $(1,4)$, $(5,7)$, $(8,11)$, $(12,14)$

Exemple 3 : Huffman Coding

Soit une séquence très longue définie sur base de caractères.
Question : *quelle est la manière la plus efficace de stocker cette séquence ?*

Exemple : Supposons une séquence de taille 10^9 composée uniquement de 6 caractères (a,b,c,d,e et r), une première approche consiste alors à encoder chaque caractère par un mot binaire de longueur fixe :

Symbole	a	b	c	d	e	r
Codage	000	001	010	011	100	101

- 6 caractères nécessitent 3 bits par caractères
- Cela fait, au total $\frac{3.10^9}{8} = 3.75.10^8$ bytes (un peu moins de 400Mb)

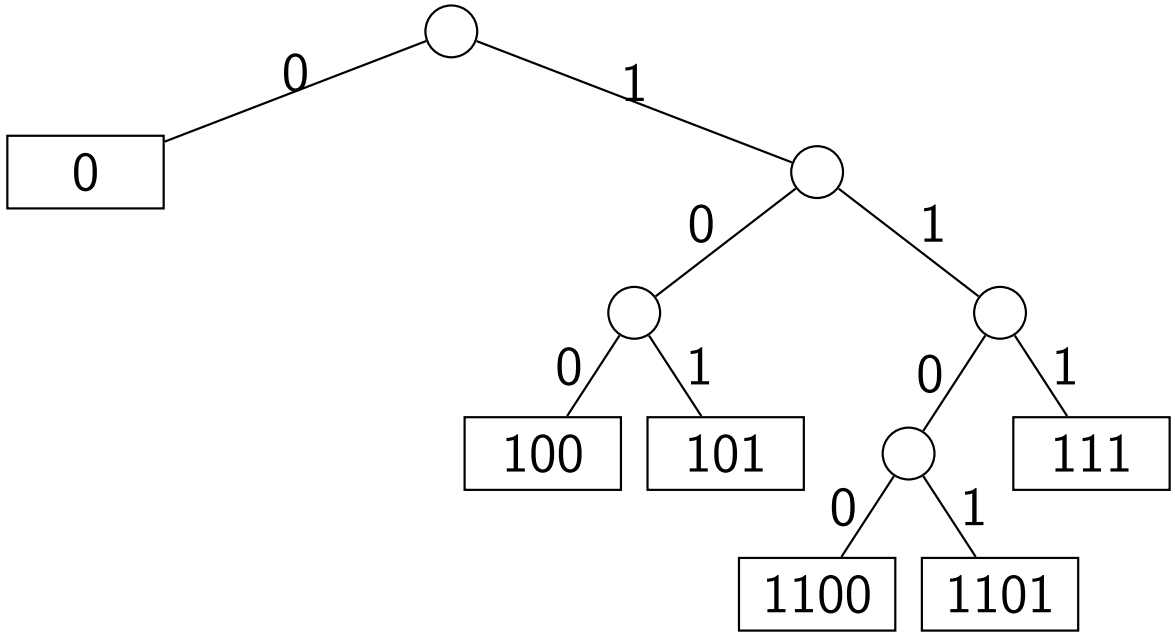
Peut-on faire mieux ?

Attention, le code utilisé doit être un *prefix code* (code préfixe) !
Dans un prefix code, aucun mot du code n'est le début d'un autre mot.

- $[1, 2, 33, 34, 50, 61]$ est un prefix code
- $[1, 2, 21, 34, 50, 61]$ n'est pas un prefix code
- $[10, 01, 000, 110, 001]$ est un prefix code
- $[10, 01, 000, 110, 010]$ n'est pas un prefix code

Exemple 3 : Huffman Coding - Code préfixe et arbre binaire

Un ensemble fini de mots sur $0, 1$ est un code préfixe si et seulement s'il est le code des feuilles d'un arbre binaire.



- Le codage d'un texte par un code préfixe (complet) se fait de la manière suivante : à chaque lettre est associée une feuille de l'arbre.
- Dans le cas de notre exemple, on pourrait avoir :
  $\{a \to 0, b \to 100, c \to 101, d \to 1100, e \to 1101, r \to 111\}$
- Le mot *acadabra* se code 01001110101011000100111O avec 24 bits

Exemple 3 : Huffman Coding

De retour au problème initial : pour un mot donné, comment faire au mieux ?
On peut montrer que le codage optimal peut être obtenu par un algo greedy :

- on construit l'arbre de bas en haut en partant des feuilles
- à chaque étape, on fait le choix glouton de fusionner les deux noeuds les moins fréquents

Exemple pour une phrase avec 45 a, 13 b, 12 c, 16 d, 9 e et 5 r :

Exercice à implémenter en séance : Coin Change

Objectif : *Etant donné un ensemble de type de billets* $C = [1, 2, 5, 10, 20]$ *en dollars, trouver une méthode pour rembourser une somme de* S *dollars en utilisant le moins de billets possible.*

Exemple : $S = 34$ dollars

- 1ère possibilité : $\{1, 1, 2, 5, 5, 20\} \rightarrow 6$ billets
- 2ème possibilité : $\{2, 2, 10, 20\} \rightarrow 4$ billets

Algorithme greedy :

- A chaque étape, on ajoute à la solution un billet de la plus grande valeur qui ne dépasse pas la somme restant à rembourser

Attention,

- cette approche greedy n'est correcte que pour certains choix de valeurs
- si on prend $C = [1, 10, 21, 34, 70, 100]$ et $S = 140$
  - approche greedy : 100, 34, 1, 1, 1, 1, 1, 1
  - solution optimale : 70, 70

Greedy methods - En résumé

- Très efficaces quand ils fonctionnent.
  Simples et faciles à implémenter.
- Ne fonctionnent pas toujours.
  Leur preuve de correction peut être très compliquée.

Etude théorique via le concept de matroïde.

Applications :

- Arbre de couverture minimal (Kruskal, Prim)
- Plus court chemin dans un graphe (algorithme de Dijkstra)
- A^* search
- au Reply Challenge ?

Plan

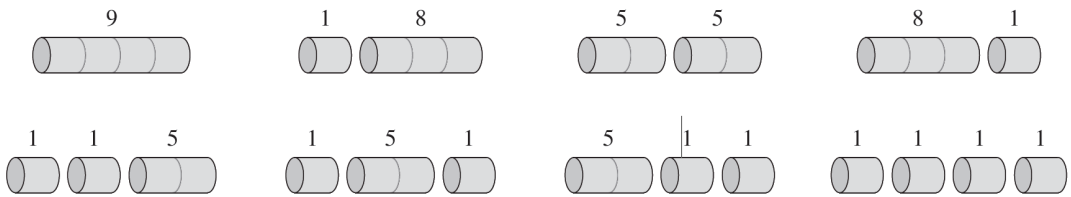
Programmation dynamique - présentation via un exemple

Soit une tige en acier de n centimètres qu'on découpe pour la vendre par morceau

- La découpe ne peut se faire que par nombre entier de centimètres
- Le prix de vente dépend de sa longueur :

Longueur i	0	1	2	3	4
Prix p_i	0	1	5	8	9

- Pour une longueur n et un tableau de prix,
  quel est le revenu maximum qu'on peut obtenir ?
- Dans le cas $n = 4$, les possibilités sont :



  Meilleur revenu : découpe en 2 tiges de 2 centimètres avec un revenu=10
- Approche brute force : 2^{n-1} manières de découper une tige de longueur n

Programmation dynamique - présentation via un exemple

Approche récursive

Soit f_j le revenu optimal pour une tige de longueur j. On a, de manière récursive :

$$f_i = \max_{1 \leq j \leq i}(p_j + f_{i-j}),$$

en supposant $f_0 = 0$.

Exemple : $f_4 = \max(p_4 + f_0, p_3 + f_1, p_2 + f_2, p_1 + f_3)$

```
def rodcutting(p,n):
    if n==0:
        return 0
    fmax=p[n]
    for i in range(n,0,-1):
        f = p[i]+rodcutting(p,n-i)
        if f>fmax:
            fmax=f
    return fmax
```

Mais la complexité de cet algorithme récursif est toujours exponentielle (2^n)...

Programmation dynamique - présentation via un exemple

Approche par programmation dynamique (enfin)

- Au lieu de résoudre les mêmes sous-problèmes plusieurs fois, on s'arrange pour ne les résoudre qu'une seule fois chacun
- Pour faire cela, on sauvegarde les solutions dans un tableau et on s'y réfère à chaque demande de résolution d'un sous-problème déjà rencontré

```
p = [0,1,5,8,9,10,17,17,20]
n = len(p)-1 #n est le nombre de centimetres de la tige
f=[0 for i in range(n+1)]
s=[0 for i in range(n+1)]
for i in range(1,n+1):
    fmax = -1
    for j in range(1,i+1):
        if p[j]+f[i-j]>fmax:
            fmax = p[j]+f[i-j]
            s[i] = j
    f[i]=fmax
print(f[-1],s)
```

Longueur i	0	1	2	3	4	5	6	7	8
p_i	0	1	5	8	9	10	17	17	20
f_i	0	1	5	8	10	13	17	18	22
s_i	0	1	2	3	2	2	6	1	2

Programmation dynamique - Knapsack

L'exemple le plus connu en programmation dynamique

Soit une valeur $W > 0$ et n objets ayant un poids $w_i > 0$ et une valeur $v_i > 0$.
Variables : $x_i = 1$ si on prend l'objet i, 0 sinon

$$\max_{x_i \in \{0,1\}} \quad \sum_{i=1}^{n} v_i x_i$$
$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq W$$

On définit, pour $k = 0, ..., n$ et $j = 0, ..., W$:

$f(k, j) =$ bénéfice optimal avec les objets de 1 à k et un sac de capacité j

Deux cas se présentent :

- Si l'objet k n'est pas sélectionné, alors $f(k, j)$ est le bénéfice optimal sur les $k - 1$ premiers objets
- Si l'objet k est sélectionné, alors $f(k, j)$ vaut la valeur de l'objet k plus le bénéfice optimal sur les $k - 1$ premiers objets

$$f(k,j) = \begin{cases} 0 & \text{si } k = 0 \\ f(k-1,j) & \text{si } w_k > j \\ \max\{f(k-1,j), v_k + f(k-1,j-w_k)\} & \text{sinon} \end{cases}$$

Programmation dynamique - Knapsack : exemple

$$f(k,j) = \begin{cases} 0 & \text{si } k = 0 \\ f(k-1,j) & \text{si } w_k > j \\ \max\{f(k-1,j), v_k + f(k-1,j-w_k)\} & \text{sinon} \end{cases}$$

Exemple : (avec $W = 11$)

k	1	2	3	4	5
v_k	1	6	18	22	28
w_k	1	2	5	6	7

f	0	1	2	3	4	5	6	7	8	9	10	11
	0	0	0	0	0	0	0	0	0	0	0	0
$\{1\}$	0	1	1	1	1	1	1	1	1	1	1	1
$\{1,2\}$	0	1	6	7	7	7	7	7	7	7	7	7
$\{1,2,3\}$	0	1	6	7	7	18	19	24	25	25	25	25
$\{1,2,3,4\}$	0	1	6	7	7	18	22	24	28	29	29	40
$\{1,2,3,4,5\}$	0	1	6	7	7	18	22	28	29	34	35	40

Revenons au dernier exercice du slide 11 du cours03

Enoncé : *si on possède autant de billets de 50, 20, 10, 5 et 1 dollars que l'on souhaite, listez toutes les manières de faire 50 dollars? Et 100 dollars ?*

Implémentez un algorithme faisant appel à la programmation dynamique pour ce problème.

Question à se poser : quels sont les sous-problèmes à considérer ?

Programmation dynamique

Les 2 mots du paradigme possèdent une signification historique :

- **Programmation** : les premières applications étaient en planification
- **Dynamique** : les décisions sont vues séquentiellement

Les problèmes traités par la programmation dynamique sont combinatoires. Une méthode consistant à tester toutes les solutions conduit à un comportement exponentiel.
Cependant, certains problèmes présentent une structure particulière qui est exploitée par la programmation dynamique :

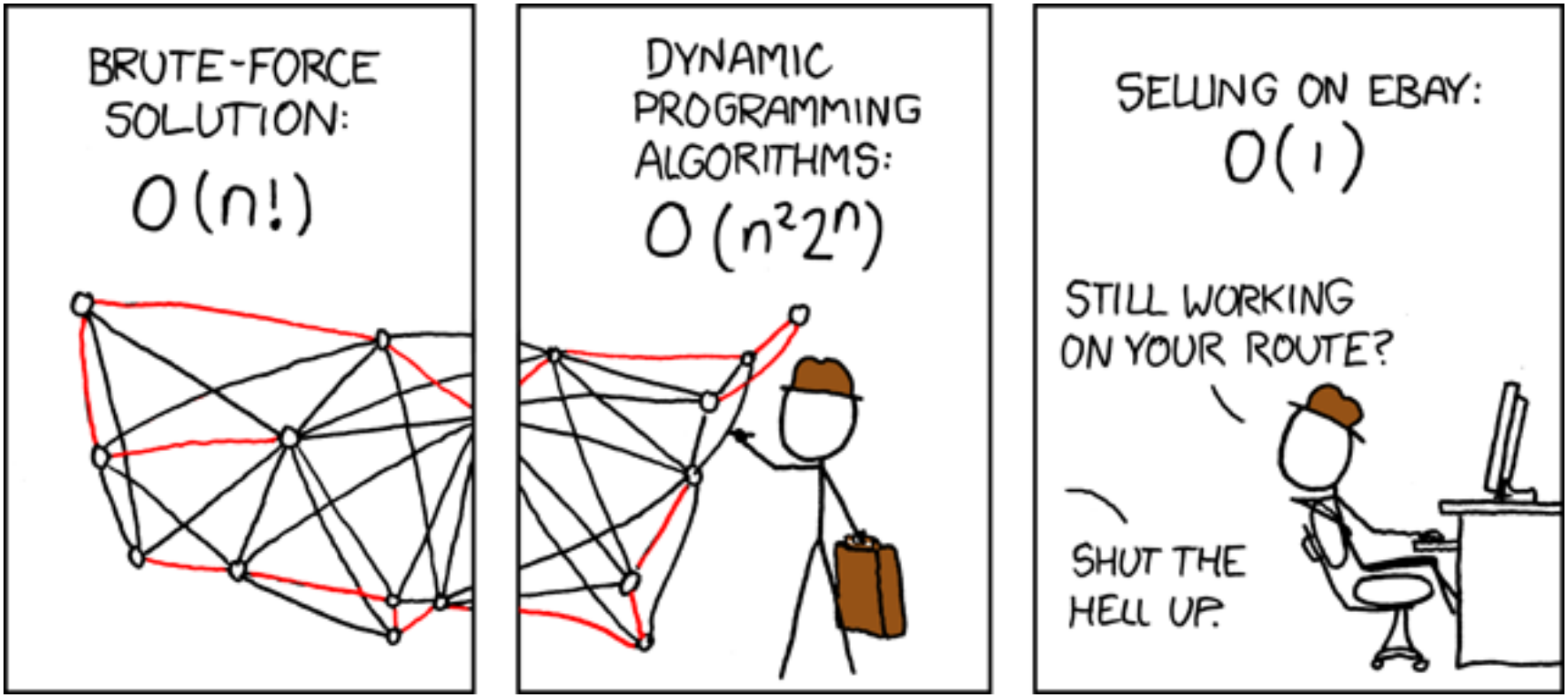
- Une solution optimale du problème considéré contient des solutions optimales de sous-problèmes
- Une solution récursive comporte un petit nombre de sous-problèmes distincts répétés plusieurs fois
- La dernière étape consiste à calculer la solution du problème

La structure des problèmes diffère et, par conséquent, les formules de récursivité permettant de calculer les solutions optimales des sous-problèmes également.

Programmation dynamique vs Méthodes gloutonnes

Les deux paradigmes necessitent la propriété de sous-structure optimale. Cependant, seuls les algorithmes gloutons nécessitent la propriété de choix gloutons optimaux.

	Greedy method	Dynamic programming
Choix	Le meilleur choix est effécuté à chaque étape sans regarder en arrière	La décision à chaque étape est prise en fonction des solutions précédentes
Optimalité	Il n'y a souvent aucune garantie d'optimalité	Souvent, un tel algorithme va générer la solution optimale car tous les cas sont considérés
Mémoire	Plus efficace car les solutions des résultats intermédiaires ne sont pas stockés	Les solutions des problèmes précédents doivent être stockées


BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.