# Paradigmes algorithmiques #3

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

# Table des matières

# Objectifs

- Identifier des cadres de résolution de problèmes algorithmiques

## Déjà vu

- Diviser pour régner
- Backtracking

## À venir

- Algorithmes gloutons
  - « Toujours » efficaces
  - Rarement corrects/optimaux
- Programmation dynamique
  - La version toujours correcte des gloutons
  - Plus-ou-moins efficace
- Rencontre au milieu
  - Un peu niche

# Table des matières

# Algorithmes gloutons

## Contexte

Généralement des problèmes d'optimisation (« quelle est la meilleure façon de … »)

## Principe

On construit une solution incrémentalement (itérativement/récursivement) :

- en faisant le choix localement meilleur à chaque fois
  - si l'on va voir plus loin, c'est plutôt de la prog. dyn.
- que l'on ne remet jamais en question
  - sinon c'est plutôt du backtracking

# Caractéristiques

- Approche généralement intuitive (mais c'est une intuition à combattre/remettre en question !)
- Généralement facile à implémenter
- Généralement efficace
  - L'exploration & évaluation des choix découle de la représentation du problème
- Généralement faux/non-optimal
  - Parfois optimal seulement en présence d'une structure suffisante
- Quand optimal, pas toujours simple de prouver que c'est le cas
- Quand non-optimal, parfois de bons algorithmes d'approximation (*cf.* l'année prochaine)

# Exemples de problèmes

## Coloriage de graphe

*Cf.* D.S.

- Simple à implémenter mais **pas optimal en général**
- Optimal pour certains ordres d'énumération
  - **et** efficace si un tel ordre se calcule efficacement

## Arbre couvrant de poids minimum

*Cf.* l'année prochaine

- Simple, **optimal**, preuve d'optimalité pas si simple

## Compression de symboles

*Cf.* dans quelques semaines (?)

- Simple, **optimal** (pour le problème précis considéré)

# Table des matières

# Programmation dynamique

## Contexte

Généralement des problèmes d'optimisation

## Principe

On construit récursivement une solution (optimale) à partir de solutions (optimales) de **sous-problèmes** qui (contrairement à une approche DPR) sont :

- En nombre possiblement non constant
  - Peut dépendre d'un paramètre d'entrée, de la taille de l'instance…
- Pas forcément indépendants
  - Gros enjeu d'efficacité

Pour que l'approche soit possible, les solutions doivent posséder une propriété de **sous-structure optimale** : une solution au problème « contient » des solutions à des sous-problèmes

# Exemple illustratif : FW point-à-point, vision prog. dyn.

On cherche (la longueur d') un plus-court-chemin (que l'on sait être élémentaire) $v \rightsquigarrow w$ dans un graphe orienté pondéré à $n$ sommets

## Recherche exhaustive

Solution évidente en temps $O(\sum_{k=0}^{n-2} A_k^n)$ *bof*

## Sous-structure optimale

Un plus-court-chemin élémentaire $v \to [\![k+1]\!] \to w$ contient (comme sous-chemins) :

- Un plus-court-chemin élémentaire $v \to [\![k]\!] \to w$, ou
- Des plus-court-chemins élémentaires $v \to [\![k]\!] \to k$ et $k \to [\![k]\!] \to w$

*e poi basta!*

## Conséquence algorithmique

Résolution récursive immédiate : le problème initial revient à résoudre $v \to [\![n]\!] \to w$ et :

- On résout $\bullet \to [\![k+1]\!] \to \bullet$ à partir de trois résolutions de $\bullet \to [\![k]\!] \to \bullet$
- Coût (hors cas de base) : donné par la récurrence $T(k+1) = 3T(k)$ *Uh-oh Oo*

# FW prog. dyn. #2 : non-indépendance des sous-problèmes

$v \to [\![k+1]\!] \to w$ demande de résoudre :

- $v \to [\![k]\!] \to w$
  - $v \to [\![k-1]\!] \to w$
  - $v \to [\![k-1]\!] \to (k-1)$
  - $(k-1) \to [\![k-1]\!] \to w$
- $v \to [\![k]\!] \to k$
  - $v \to [\![k-1]\!] \to k$
  - $v \to [\![k-1]\!] \to (k-1)$
  - $(k-1) \to [\![k-1]\!] \to k$
- $k \to [\![k]\!] \to w$
  - $k \to [\![k-1]\!] \to w$
  - $k \to [\![k-1]\!] \to (k-1)$
  - $(k-1) \to [\![k-1]\!] \to w$

On passe notre temps à résoudre **récursivement** les mêmes problèmes !

# FW prog. dyn. #3 : chasser la redondance

La **non-indépendance** des sous-problèmes est **très fréquente** en prog. dyn.

Deux grandes techniques de résolution :

- Par « **mémoïsation** » de l'algorithme récursif naturel
  - Approche générique facile à mettre en œuvre, mais possiblement plus gourmande que nécessaire
- On **dérécursifie**, en résolvant les sous-problèmes dans l'ordre de dépendance (« **calcul de bas en haut** »)
  - Il y a bien nécessairement un ordre (pourquoi ?) que l'on peut par exemple représenter par un DAG
  - Mais souvent (quand on applique cette approche) très structuré (généralement pas de représentation explicite)

# Mémoïsation

## Principe

On écrit l'algorithme récursivement, mais l'on maintient une structure globale $M$ (typiquement implémentée par un tableau associatif) et :

- quand on a calculé la solution à un sous-problème, on la stocke dans $M$
- au début d'un appel (récursif), on vérifie d'abord dans $M$ si l'on ne connaît pas déjà la solution

## Coût en temps

Hors coût d'opération de $M$, la résolution d'un problème coûte seulement **une fois** le **coût marginal** de résolution de **tous ses sous-problèmes**

## Coût en mémoire

En général, il faut stocker les solutions de **tous les sous-problèmes**

- Potentiellement important, parfois la ressource limitante

# Mémoïsation : structure typique d'une fonction mémoïsée

```
let rec solve a b c =
    (* ...  *)
    res
```

devient

```
let rec solve' m a b c =
    if mem m (a, b, c) then find m (a, b, c) else
    (* ... *)
    add m (a, b, c) res ;
    res
```

On pourrait même *mécaniser* une telle transformation

si vous n'avez toujours pas de sujet de TIPE…

# FW prog. dyn. #4

## Coût d'une version récursive mémoïsée

- Le coût marginal de la résolution de $\bullet \to [\![\bullet]\!] \to \bullet$ est constant
  - Si l'on suppose un coût d'opération constant pour la structure de mémoïsation
- Les (sous-)problèmes sont donnés par des triplets quelconques de sommets $v, k, w$ : $n^3$ sous-problèmes pour un graphe à $n$ sommets
- Coût en temps et mémoire $O(n^3)$
  - Gain en temps exponentiel par rapport à une version naïve
  - (À comparer avec les fausses/vrai exponentiation rapides ; le dénombrement de lagomorphes…)

# Reconstruction « de bas-en-haut »

## Principe

Plutôt que de partir du « haut » du problème pour déterminer les sous-problèmes à résoudre (récursivement), on part du « bas » et l'on construit progressivement la solution

- En clair, on dérécursifie
- À réserver aux cas simples (de la même façon qu'en général on ne dérécursifie pas tous nos algorithmes)

## Comparaison avec la mémoïsation

- Moins automatique : demande de trouver (à la main, algorithmiquement) la structure de dépendance des sous-problèmes
- Simple quand les dépendances sont simples (par ex. FW)
  - Alors souvent possible de stocker les solutions des sous-problèmes dans un tableau pas-associatif
- Coût *généralement* facile à analyser (pas de récursion)
- Possiblement plus économe en mémoire (on peut oublier les solutions des sous-problèmes que l'on n'utilisera plus)

# FW prog. dyn. #5

## Construction d'une version de bas-en-haut

- Une résolution du problème $\bullet \to [\![k]\!] \to \bullet$ dépend de résolutions de sous-problèmes $\bullet \to [\![k-1]\!] \to \bullet$
- Ordre de résolution évident : par $k$ croissant
- Si l'on ne veut pas réfléchir, on résout **tous** les problèmes $\bullet \to [\![0]\!] \to \bullet$, puis de là **tous** les problèmes $\bullet \to [\![1]\!] \to \bullet$, puis…, jusqu'à $\bullet \to [\![n]\!] \to \bullet$
- Il y a $n^2$ sous-problèmes à $k$ fixé : coût **total** $O(n \times n^2)$ en temps mais seulement $O(n^2)$ en mémoire
  - Une fois les problèmes $\bullet \to [\![k]\!] \to \bullet$ résolus, plus besoin de garder en mémoire les solutions de $\bullet \to [\![k-1]\!] \to \bullet$
  - Une analyse plus fine montre même que l'on peut être en coût mémoire constant, *cf.* TP précédent

On retrouve l'algorithme de FW original

# Calcul d'optimum v. construction de solution

- L'algorithme de FW « de base » calcule seulement la **longueur** d'un plus-court-chemin, mais pas les chemins eux-mêmes
- C'est cependant facile de stocker des **informations supplémentaires** permettant une **reconstruction** d'une solution optimale
  - On stocke un tableau donnant en $(i, j)$ l'identifiant d'un sommet se trouvant sur le plus-court-chemin $i \rightsquigarrow j$ (s'il y en a un), *cf.* TP

C'est une caractéristique commune des algorithmes de programmation dynamique (pour problèmes d'optimisation)

# Bilan sur la prog. dyn.

- Une approche générique pour résoudre des problèmes (d'optimisation) qui possèdent une propriété de sous-structure optimale
  - Résolution récursive naturelle, avec (potentiellement) **beaucoup de sous-problèmes non indépendants**
  - À mémoïser ou dérécursifier (peut parfois se voir comme une optimisation d'une recherche par backtracking)
- Souvent plus efficace qu'une recherche exhaustive
  - Des fois **beaucoup**, des fois moins…
- Souvent l'alternative correcte à une approche gloutonne
- Souvent gourmand en mémoire

# Table des matières

# Rencontre au milieu

## Contexte

Généralement des problèmes de recherche

## Principe

On cherche à exprimer la solution $x \in \mathcal{S}$ d'un problème comme une collision entre deux fonctions $f, g$ à image dans $\mathcal{S}$

- $F$ images de $f$ & $G$ images de $g$ peuvent être appariées de $F \times G$ façons différentes
- Quand $F \times G = \#\mathcal{S} =: N$, on peut *espérer* (pour des fonctions uniformes : paradoxe des anniversaires) ou (parfois) garantir une collision, et donc une solution (candidate)
  - Pour $F = G$, l'espérance du **nombre** de collisions **croît quadratiquement** en $F$
- **Compromis temps mémoire** possible pour une recherche naïve de collision : temps (T) = $O(\max(F, G))$, mémoire (M) = $O(\min(F, G))$
- Courbe de compromis temps-mémoire $T \times M = N$
  - Contrainte : $M \leq T$

# Exemple 1 : un peu de crypto

## Chiffre par bloc

Un *chiffre par bloc* (en anglais : *block cipher*) est une **famille de permutation paramétrée par une clef** : une fonction $E : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$ t.q. $\forall k \in \mathcal{K}$, $E(k, \bullet)$ est une permutation (une bijection) $\mathcal{M} \to \mathcal{M}$

## Problème de recherche de clef

Soit $p$, $c = E(k, p)$, trouver $k'$ t.q. $c = E(k', p)$

- Coût pire cas d'une recherche exhaustive évidente : $O(\#\mathcal{K})$ (évaluations de $E$) en temps, $O(1)$ mémoire

## Chiffrement double

Soit $E$ un chiffre par bloc, on peut définir $E^2 : (\mathcal{K} \times \mathcal{K}) \to \mathcal{M}$ comme
$E^2((k_1, k_2), p) = E(k_2, E(k_1, p))$

## Résolution d'une recherche de clef pour chiffrement double ?

# Exemple 1 *bis* : MITM pour le chiffrement double

On suppose qu'un couple $(p, c)$ détermine uniquement $(k_1, k_2)$ (ce qui implique notamment $\#\mathcal{M} \geq \#\mathcal{K}^2$)

## Recherche exhaustive

Coût pire cas $O(\#\mathcal{K}^2)$ en temps, $O(1)$ mémoire

## *Meet-In-The-Middle*

- On calcule $\mathcal{F} := \{E(k, p), k \in \mathcal{K}\}$
- On calcule $\mathcal{G} := \{E^{-1}(k, c), k \in \mathcal{K}\}$
  - $E(k, \bullet)$ est une permutation pour tout $k$

La solution est révélée par la collision (unique par notre hypothèse) entre $\mathcal{F}$ et $\mathcal{G}$

- Coût $O(\#\mathcal{K})$ en temps **et mémoire**
  - En utilisant par ex. un tableau associatif pour la détection de collision, ou un tri linéaire + fusion, ou…
- **Même coût temporel** que pour le chiffrement *pas double*
- Compromis temps-mémoire possible

# Exemple 2 : un peu de ~~crypto~~ algèbre computationnelle

*baby-step/giant-step* pour le calcul de logarithme discret

# Exemple 3 : un peu de ~~crypto~~ théorie des codes

## Recherche de mot de faible poids

Soit $\mathcal{C}$ un sous-espace vectoriel de $(\mathbb{Z}/2\mathbb{Z})^n$ de dimension $k$, trouver $\mathbf{x} \in \mathcal{C}$ de *poids* faible (par exemple minimum), où le poids de $\mathbf{x}$ est donné par son nombre de coordonnées à 0 dans une représentation canonique

## Recherche exhaustive

Coût pire cas : $O(2^k)$ calculs dans $\mathcal{C}$

## Amélioration par le paradoxe des anniversaires

Il existe toujours $k$ coordonnées sur lesquelles on peut *choisir* la valeur de tout élément de $\mathcal{C}$ ; on les divise en trois sous-ensembles $\mathcal{I}_1, \mathcal{I}_2, \mathcal{P}$ de taille $k/2 - p$, $k/2 - p$, $2p$, et l'on génère deux ensembles de vecteurs :

- $\mathcal{C}_1$ : « faible » poids en $\mathcal{I}_1$, poids nul en $\mathcal{I}_2$
- $\mathcal{C}_2$ : « faible » poids en $\mathcal{I}_2$, poids nul en $\mathcal{I}_1$

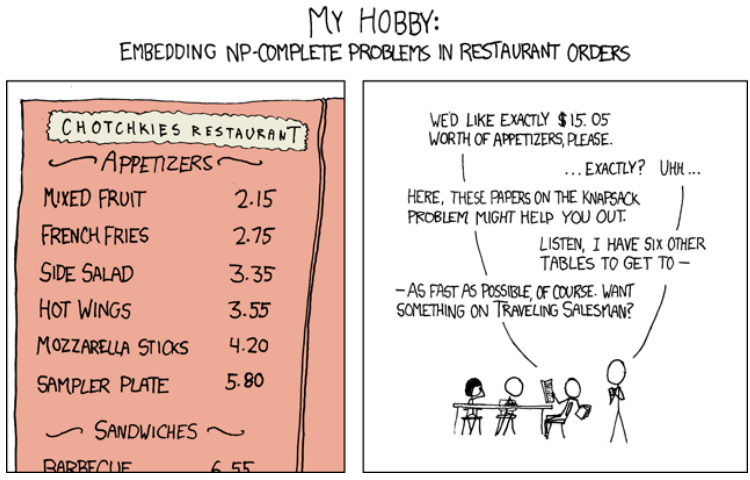
Et l'on cherche des mots de faible poids parmi ceux de $\mathcal{C}_1 \bigoplus \mathcal{C}_2$ **qui s'annulent sur** $\mathcal{P}$

- Ce sont des collisions : leur nombre augmente **quadratiquement** avec $\#\mathcal{C}_1, \#\mathcal{C}_2$

Coût pire cas bien meilleur qu'une recherche exhaustive (mais reste exponentiel en $k$, $n$)

# Table des matières

# Illustrations avec un sac à dos




https://xkcd.com/287/

# Problèmes du sac à dos (*knapsack problem*)

## Une version multiple

Soit $\mathcal{O} = \{(w_i, v_i\}$ un ensemble de types d'objets de poids $w_i$ et de valeur $v_i$ tous deux **entiers positifs** et $C$ la capacité en poids d'un sac à dos, on souhaite remplir le sac à dos d'objets de $\mathcal{O}$ dans la limite de sa capacité, en maximisant la valeur des objets

- Soit $s_i$ le nombre (naturel) de fois que l'objet $i$ est pris dans le sac, on veut :
  - maximiser $\sum_i s_i \times v_i$
  - sous la contrainte $\sum_i s_i \times w_i \leq C$

## Variante duale

Pour une cible $t$, l'on cherche la **capacité minimale** d'un sac de valeur $\geq t$

- Pour $w_i = 1$ pour tout $i$ et $= t$, c'est le problème du « rendu de monnaie »

## Une version unique (*subset sum*)

Soit $\mathcal{S}$ un ensemble d'entiers naturels, $t \in \mathbb{N}$, on cherche un sous-ensemble de $\mathcal{S}$ (s'il existe) dont la somme vaut $t$

- On cherche $\mathcal{T} \in 2^{\mathcal{S}}$ tel que $\sum_{x \in \mathcal{T}} = t$

# Résolution de sac à dos : approche exhaustive

## Version unique

Trivialement exponentiel en le nombre d'objets

## Version multiple

Peut se réduire à une version unique : exponentiel en le nombre d'objets à multiplicité max

# Résolution de sac à dos : approche gloutonne

On choisit autant de fois que possible l'objet maximisant $v/w$ et l'on répète jusqu'à atteindre la cible

- Efficace
  - Typiquement linéaire en la taille du problème
- Généralement non optimal
  - Contre-exemple en version « rendu de monnaie » : cible 10 pour des valeurs $7, 6, 4, 1$
- Possiblement optimal pour certaines instances
  - Exemple en version « rendu de monnaie » : des valeurs $1, b, b^2, \ldots, b^k$ pour un certain $b$ entier $> 1$ (*pourquoi ?*)

# Résolution de sac à dos : approche prog. dyn.

## Sous-structure optimale

Soit $C$ une capacité et $\mathcal{O}_C \subseteq \mathcal{O}$ les objets de poids $\leq C$

- Si $\mathcal{O}_C \neq \varnothing$, une solution optimale inclut forcément un objet $o = (w, \_) \in \mathcal{O}_C$ et sinon il n'y a pas de solution non triviale
- Et alors elle contient forcément une solution optimale du sous-problème de capacité $C - w$ (obtenue en retirant une fois $o$)

## Résolution récursive

Si l'on se contente de calculer la valeur, par implémentation immédiate de la formule :

$$V_C = \max_{(w,v)\in\mathcal{O}_C} V_{C-w} + v$$

# Résolution par prog. dyn. : analyse de coût

- Au plus $C$ sous-problèmes majoration brutale
- Coût marginal de résolution d'un sous-problème : au plus $\#\mathcal{O} =: n$
- Coût pire cas avec mémoïsation : $O(nC)$ temps, $O(C)$ mémoire
  - Dépendance **linéaire** en le nombre d'objets
  - « Efficace si $C$ est petit »
  - Mais tout de même **exponentiel en la taille de l'entrée** qui est un $\Omega(n + \log C)$

# Résolution de sac à dos / subset sum : approche MITM

## Esquisse de l'algorithme

Soit $\mathcal{S}$ l'ensemble de $n$ entiers naturels, $t$ la cible :

- On partitionne arbitrairement $\mathcal{S}$ en $\mathcal{S}_1$, $\mathcal{S}_2$ de taille $n/2$ on suppose $n$ pair sans perte de généralité
- On calcule $\mathcal{T}_1$ ensemble des sommes des éléments de $2^{\mathcal{S}_1}$
  - $2^{n/2}$ éléments
- On calcule $\mathcal{T}_2$ ensemble des sommes des éléments de $2^{\mathcal{S}_2}$ **que l'on soustrait à** $t$
  - *ditto*
- Les solutions au problème sont exactement révélées par les collisions entre éléments de $\mathcal{T}_1$ et $\mathcal{T}_2$
  - soit $\sigma_1 \in 2^{\mathcal{S}_1}$, $\sigma_2 \in 2^{\mathcal{S}_2}$ t.q. $\sum_{x\in\sigma_1} x = t - \sum_{x\in\sigma_2} x$, alors $\sum_{x\in\sigma_1\cup\sigma_2} x = t$

# Résolution par MITM : analyse de coût

- Coût en temps $O(2^{n/2})$ pour les constructions de $\mathcal{T}_1$ et $\mathcal{T}_2$ et pour la recherche de collision (si fait efficacement)
- Coût en mémoire $O(2^{n/2})$ (il faut au moins stocker l'un des $\mathcal{T}_i$)
- Équilibre temps/mémoire peu attractif, mais compromis temps-mémoire évident

Coût exponentiel en $n$, mais gain (jusqu'à) quadratique par rapport à la recherche exhaustive !

## Amélioration possible

On peut réduire le coût **mémoire** à un $O(2^{n/4})$ (pour le coût en temps en $O(2^{n/2})$) en utilisant une représentation compressée des $\mathcal{T}_i$ (algorithme de Schroeppel-Shamir)

- Très attractif !