Activité de cours

PROGRAMMATION DYNAMIQUE

Terminale NSI
Lycée Émile Combes, Pons
Année 2023 – 2024
M. Rouquan

Exemples de problèmes d'optimisation

Rendu de monnaie

Un caissier souhaite rendre une certaine **somme** à un client. Il dispose d'une quantité illimitée de **pièces de différentes valeurs**. Son objectif est de rembourser cette somme en rendant le **minimum** de pièce.

Donner une solution si le caissier doit rendre **25€** avec les pièces de **1€, 5€, 6€ et 7€**. Quelle est son coût ?

- Solution :
- Coût :

Votre solution est-elle optimale ?

Même question si le caissier doit rendre 172€ avec les pièces de **1€, 5€, 8€, 11€ et 14€.**

Découpe de ruban

Dans la cour de récréation, le marché du *Hubba Bubba*, un chewing-gum en forme de ruban, connaît une forte demande. Chaque **longueur** de ruban a un **prix** spécifique. Un élève se pose la question sur la manière optimale de découper son ruban d'une certaine **longueur** afin de **maximiser** son bénéfice. Les morceaux découpés doivent être de longueur entière.

Soit le prix des différentes longueurs :

Longueur	1	2	3	4	5	6	7
Prix	1€	5€	8€	9€	10€	17€	17€

Quelle est la solution optimale pour découper un ruban de longueur initiale **4** ? Énumérer toutes les solutions possibles.

De même pour un ruban de longueur initiale **5**.

Et **17** ?

Problème du sac à dos

On dispose d'objets de poids et de valeurs différents et d'un sac avec une certaine capacité en poids. Quels objets affecter au sac pour maximiser la valeur totale du sac sans que le poids total dépasse la capacité ?

Soit l'instance de ce problème suivante :

Objet n°	1	2	3	4	5	6	7	8	9
Poids kg	28	5	15	9	24	7	21	25	1
Prix €	15	25	65	10	75	30	80	1	100

Déterminer la solution optimale !

Point vocabulaire

- Un **problème d'optimisation** est un problème où l'on souhaite **minimiser** ou **maximiser** une certaine quantité numérique. Cette quantité est appelée **objectif**.
- Une **instance** d'un problème d'optimisation définit concrètement les **données** de celui-ci.
- Le **coût** d'une solution est la valeur de l'objectif de cette solution.
- Une solution est **optimale** si son coût est **optimal**, s'il est le plus petit/grand possible suivant si on minimise/maximise.

Une séquence de choix

Le **graphe des états** qui permet de visualiser toutes les solutions possibles. Dans ce graphe :

- Un **sommet** représente un **ÉTAT**, càd les informations nécessaires pour effectuer un choix
- Un **arc** représente un **CHOIX** possible

Rendu de monnaie

Pour le problème du rendu de monnaie, à chaque étape, on choisit une pièce à rendre :

Pièce ➔ Pièce ➔ Pièce ➔ Pièce ➔ etc.

Le caissier doit rendre une somme de 6€ avec des pièces de 1€, 3€, 4€. Pour cette instance, donner le graphe des états et mettre en évidence la solution optimale et la solution gloutonne.

- **Sommet / État :** Somme restante à rendre
- **Arc / Choix :** Une pièce rendue

Existe-t-il une autre définition possible des états/choix pour ce problème ? Proposer une autre séquence de choix. Comment cela change-t-il l'état ?

✂ Découpe de ruban

Pour le problème de découpe, à chaque étape, on choisit une longueur de ruban à découper :

Longueur ➔ Longueur ➔ Longueur ➔ etc.

On reprend l'instance précédente. On découpe un ruban de longueur initiale **5** :

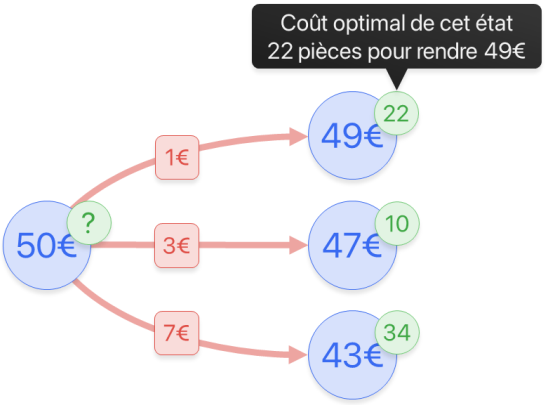
Longueur	1	2	3	4	5	6	7
Prix	1€	5€	8€	9€	10€	17€	17€

Donner le graphe des états pour cette instance. Proposer une pondération des arcs (choix). Mettre en évidence la solution optimale.

- **Sommet / État :** Longueur de ruban restante
- **Arc / Choix :** Une longueur découpée du ruban

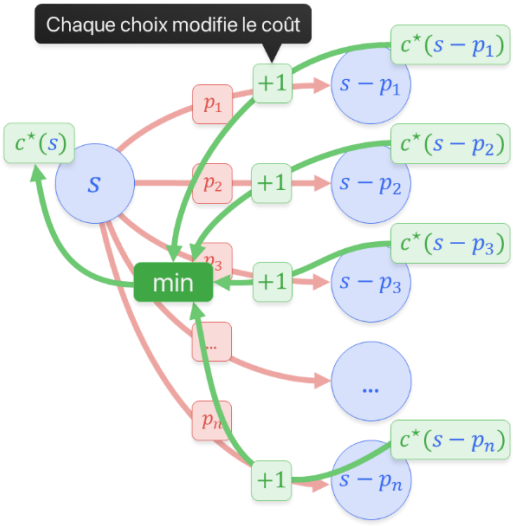
Définition récursive du coût optimal

On reprend le problème de rendu de monnaie. On souhaite rendre 50€ avec les pièces 1€, 3€ et 7€. On construit le début du graphe des états :



On suppose qu'on a résolu les états enfants (les sous-problèmes) ; quel est le coût optimal de l'état parent (50€) ici ? Donner le calcul mathématique effectué :

On généralise pour un **état quelconque** s. On pose **$c^\star(s)$ le coût optimal** d'un état ; il faut rendre au minimum $c^\star(s)$ pièces pour rendre la somme s. Le graphe « motif » :



> **Mais ce n'est pas tout le graphe des états ça !**
>
> Ce morceau de graphe définit **totalement** le graphe des états. En effet, il définit le voisinage pour n'importe quel état, et donc tous les états ! C'est un motif que l'on peut répéter aux états enfants etc.

Exprimer alors mathématiquement le coût optimal de l'état parent $c^\star(s)$ en fonction des coût optimaux des états enfants $c^\star(s-p_1), c^\star(s-p_2), c^\star(s-p_3), \cdots$:

Cette définition est **récursive** ! On pourra calculer $c^\star(s-p_1), c^\star(s-p_2)$ etc. de la même manière… mais que manque-t-il à notre récursion ?

Résoudre notre problème initial revient donc à calculer : $c^\star(\ldots\ldots\ldots)$

La programmation dynamique

Bravo ! Vous venez de maitriser le cœur de la programmation dynamique : **définir récursivement le coût optimal d'un problème (état) en fonction des coûts optimaux de ses sous-problèmes (états enfants).**

Proposer une implémentation naïve en Python de cette fonction :

```
def rendu(somme, pieces):
    def c(s):
        if s == 0:
            return ……
        else:
            return ……
    return c(……)
```

Une fonction dans une fonction ?

La fonction interne `c` a accès aux paramètres `somme` et `pieces` de la fonction `rendu`. Cela simplifie le code en évitant de passer en paramètres `somme` et `pieces` à la fonction `c`.

Découpe de ruban

En suivant la même **démarche**, résoudre le problème de découpe de ruban :

1. Définir les **données** d'une instance
2. Définir **l'objectif** : c'est-à-dire la quantité numérique que l'on souhaite minimiser / maximiser
3. Identifier une **séquence de choix** à effectuer
4. Définir un **état** et les **choix possibles** à partir de cet état
5. Dessiner le graphe « motif » : ajouter les coûts pour chaque choix
6. Définir récursivement le **coût optimal** d'un état en fonction du coût optimal de ses états enfants.
   Ne pas oublier les cas de base et de pondérer $\pm\infty$ les états irréalisables.

Question ouverte

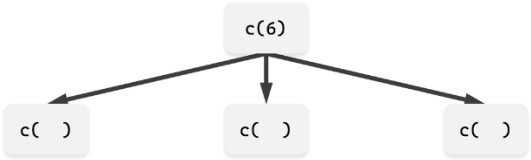
Pour le problème du sac à dos, il faut être un peu malin pour définir l'état. On peut considérer la séquence de choix suivante :

Objet N°1 (affecter ou ne pas affecter ?) → Objet N°2 (affecter ou ne pas affecter ?) → etc.

Je vous laisse en exercice la définition récursive du coût optimal.

La cerise sur le gâteau : le principe de mémoïsation

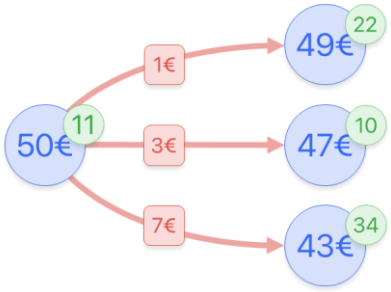
Dessiner **l'arbre d'appels** de la fonction **c** définit précédemment pour le problème de rendu de monnaie où l'on rend 6€ avec les pièces 1€, 3€ et 7€ :



Que remarque-t-on ? Quelle solution pour résoudre ce problème ?

- On remarque :
- Une solution serait de :

Pour mémoriser le coût optimal calculé pour chaque, on utilisera un **dictionnaire** dans un premier temps.



⇔

```
cache = {
    50 : 11,
    49 : 22,
    47 : 10,
    43 : 34
}
```

Compléter alors le programme suivant :

```
def rendu(somme, pieces):
    cache = {}  # état <-> coût optimal ; cache[état] = cout opt. de l'état
    def c(s):
        if s not in cache:  # si on a pas déjà calculé le coût opt. de cet état, on le calcule
            if s == 0:
                ..... = 0
            else:
                ..... = min(c(s - p) + 1 for p in pieces if p <= s)
        return .....

    return c(somme)
```

Le principe de mémoïsation

Le principe de mémoïsation consiste à **mémoriser** les résultats des calculs effectués (ici le calcul du coût optimal pour un état), afin d'éviter de les recalculer inutilement si on retombe dessus.

Quelle est la **complexité en temps** et **en espace** de notre algorithme final de rendu ? En fonction de S la somme à rendre et P l'ensemble des valeurs des pièces.

Aller plus loin

- Au lieu d'utiliser un dictionnaire, on peut ici utiliser une simple liste.
- Au lieu de faire une récursion à partir de l'état initial du problème, on peut commencer à partir des états finaux (feuilles) puis remonter : on obtient alors un algorithme itératif plus performant ! C'est l'approche « *bottom-up* », au contraire de l'approche « *top-bottom* »).
- Notre algorithme ne renvoie que le coût optimal, mais pas la solution optimale. Pour récupérer cette dernière, il suffit de mémoriser pour chaque état le meilleur choix effectué ; on utilisera par exemple un autre cache.

Synthèse

La démarche générale de la programmation dynamique

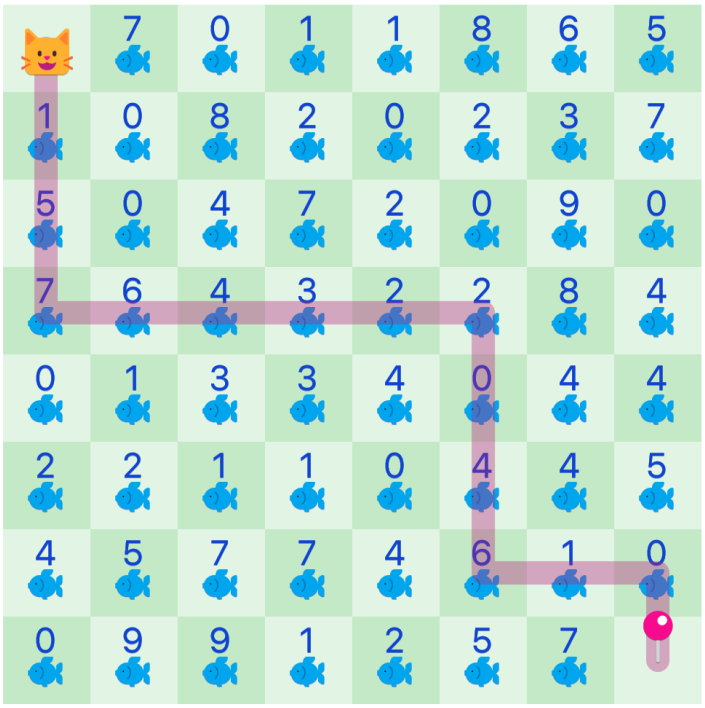
Soit un problème d'optimisation :

1. Définir les **données** d'une instance
2. Définir **l'objectif** : c'est-à-dire la quantité numérique que l'on souhaite minimiser / maximiser
3. Identifier une **séquence de choix** à effectuer
4. Définir un **état** et les **choix possibles** à partir de cet état
5. Dessiner le graphe « motif » : ajouter les coûts pour chaque choix
6. Définir récursivement le **coût optimal** d'un état en fonction du coût optimal de ses états enfants. Ne pas oublier les cas de base et de pondérer $\pm\infty$ les états irréalisables.
7. Implémenter naïvement la récursion en Python.
8. Ajouter un **cache** pour mémoriser le coût optimal calculé pour chaque état.
9. **Bonus :** Ajouter un second cache pour mémoriser le meilleur choix à effectuer pour chaque état : ceci permettra de retrouver la solution.
10. **Bonus :** Proposer une approche itérative, une approche « *bottom-up* » (on part des racines pour remonter au nœud initial) en remplissant le cache intelligemment.

Projet

En binôme, résoudre grâce à la programmation dynamique un des deux problèmes suivants :

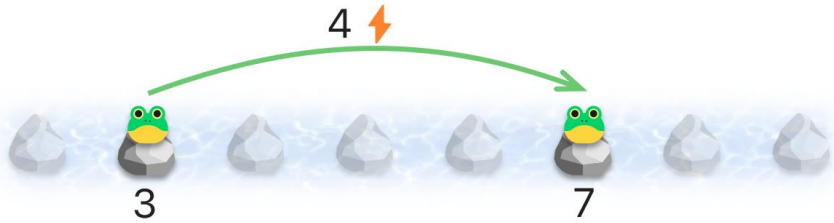
PROBLÈME DU CHAT



- Un chat se retrouve en haut à gauche d'un échiquier. Lorsque le chat se déplace, il ne peut qu'aller vers la case de droite ou vers la case du bas. Son but est d'atteindre la case en bas à droite.
- Sur chaque case de l'échiquier (sauf le point de départ et d'arrivé) est déposée une certaine quantité de poisson.
- Lorsque le chat prend un chemin, il mange tous les poissons sur les cases qu'il emprunte !
- Par exemple, si le chat suit le chemin rose, il mange 1 + 5 + 7 + ... = 39 poissons.
- Quel chemin doit prendre le chat pour manger le plus poisson ?
- L'algorithme doit s'adapter à toute taille d'échiquier carré et à toute quantité de poissons sur les cases.

PROBLÈME DE LA GRENOUILLE

- Une grenouille désire de traverser un étang dont la surface est ponctuée par n pierres uniformément espacées. Les pierres sont numérotées de 0 à $n - 1$.
- Posé initialement sur la première pierre, notre intrépide batracien devra exécuter **une séquence de sauts en avant** afin d'atteindre la dernière pierre.



- Si la grenouille saute d'une pierre i vers une pierre j (avec $i < j$ car elle ne peut sauter qu'en avant), elle saute d'une longueur égale à $l = j - i$.
- La longueur maximale d'un saut de la grenouille dépend de son **niveau d'énergie** (≥ 0). Un saut de longueur l coûte à la grenouille une quantité d'énergie égale à l.

Sur certaines pierres, il peut y avoir des **vers**, qui ajoutent de l'énergie à la grenouille. Gourmande, si la grenouille saute sur une pierre avec des vers, elle les mangera tous !

La liste $E = [e_0, e_1, \ldots, e_{n-1}]$ contient les valeurs d'énergie des vers sur les pierres correspondantes. S'il n'y a pas de ver sur la pierre, cela équivaut à une valeur de 0.

La grenouille démarre avec 0 d'énergie, mais il est garanti qu'il y a au moins un ver sur la première pierre.

Étant donnée une liste E quelconque, quel est le nombre minimal de sauts pour que la grenouille atteigne la dernière pierre ?

Rédigez un rapport comprenant les points suivants :

1. Définition des données du problème et de l'objectif (1 point)
2. Définition d'un état, des choix possibles à partir de cet état (1 point)
3. Schéma du graphe « motif » (2 points)
4. Formulation récursive du coût optimal (6 points)
5. Implémentation en Python avec mémoïsation (7 points)
6. Analyse de la complexité temporelle et spatiale de l'algorithme (1 point)
7. Bonus : Adaptation de l'algorithme pour qu'il renvoie également la solution (bonus de 2 points)

La mise en forme du rapport et du code (incluant des commentaires, des tests et des noms de variables explicites) sera sur 2 points.