Programmation Dynamique

I - Introduction

1. Exemple du problème du Sac à dos

Écrivons un algorithme qui résout le problème par backtracking

```ocaml
(* obj : (int*int) array : tablau de poids/valeur *)
let knapsack obj poids_max =
    (* On souhaite calculer la valeur optimale sans conserver
    la façon de remplir x *)
    (* p : poids_max courant *)
    let rec aux i p =
        if i = 0 || p = 0 then 0
        else begin
            if fst obj.(i-1) > p then aux (i-1) p
            else
                max (aux (i-1) p)
                    (snd obj.(i-1) + aux (i-1) (p-(fst obj.(i-1))))
        end
    in
    aux (Array.length obj) poids_max;;
```

$\Longrightarrow$ On fait apparaître la notion de sous-problèmes. La valeur renvoyée par `aux i p` est la valeur optimale du sac avec les objets $obj[0 : i]$ et le poids max p.

Prenons un exemple : $\text{obj} = [\![(1,3); (3,10); (5,7); (8,12)]\!]$ et $\text{poids\_max}(\text{max}) = 10$

On dessine l'arbre des appels récursifs à la fonction aux.

```toml
aux 4 10
├── oui
|   └── aux 3 2
|       └── non
|           └── aux 2 2
|               └── non
|                   └── aux 1 2
└── non
    └── aux 3 10
        ├── oui
        |   └── aux 2 5
        |       ├── oui
        |       |   └── aux 1 2
        |       └── non
        |           └── ...
        └── non
            └── aux 2 10
                └── ...
```

$\Longrightarrow$ Le calcul `aux 1 2` va être effectué 2 fois !

2. Mémoïsation

La mémoïsation est une “technique” en informatique qui consiste à mémoriser (stocker en mémoire) des résultats de “calculs” qui risquent d’être réutilisés plus tard.

De manière générale, on peut mémoïser une fonction $f : 'a \to 'b$

$\Rightarrow$ On utilise un dictionnaire dont les clés sont les arguments de f (de type ‘a) et les valeurs associées sont les images de f : $\{x : f(x), \ldots\}$

Appliquons cette idée :

```ocaml
let knapsack obj poids_max =
    (* Création de la table *)
    let d = Hashtbl.create () in

    (* Cas de base *)
    for i = 0 to Array.length obj do
        Hashtbl.add d (i,0) 0
    done;
    for p = 0 to poids_max do
        Hashtbl.add d (0,p) 0
    done;

    (* Fonction aux *)
    let rec aux i p = match Hashtbl.find_opt d (i,p) with
        | None ->
            if fst obj.(i-1) > p then begin
                let v = aux (i-1) p in
                Hashtbl.add d (i,p) v;
                v
            end else begin
                let v = max (aux (i-1) p)
                    (snd obj.(i-1) (p- fst obj.(i-1))) in
                Hashtbl.add d (i,p) v;
                v
        | Some v -> v
    in
    aux (Array.length obj) poids_max;;
```

Exemple tiré de wikipédia

```ocaml
let memo f =
    let h = Hashtbl.create 97 in
    fun x ->
      try Hashtbl.find h x
      with Not_found ->
        let y = f x in
        Hashtbl.add h x y ;
        y ;;
```

```

let ma_fonction_efficace = memo ma_fonction;;
```

```ocaml
(* Pour une fonction récursive comme la suite de Fibonacci *)
let memo_rec yf =
    let h = Hashtbl.create 97 in
    let rec f x =
      try Hashtbl.find h x
      with Not_found ->
        let y = yf f x in
        Hashtbl.add h x y ;
        y
    in f ;;

let fib = memo_rec (fun fib n -> if n<2 then n else fib (n-1)+fib (n-2));;
```

II - Principe de la programmation Dynamique

C'est un concept aux contours assez flous. A notre niveau, les exercices liés à la programmation dynamique auront (presque) toujours la forme suivante :

1. Établir une équation de récurrence qui décrit le problème concret : $u_n = u_{n...,p...} + max\{u_{n...,p...}\}$
2. Écrire un programme qui calcule $u_{n,p}$ (sans refaire 2 fois le même calcul).

Reprise de l'exemple du sac à dos :

$\Longrightarrow$ On pose $v_{i,p}$ la valeur optimale du sac à dos pour le sous-problème $obj[0 : i]$ aux poids maximal p.

$\forall i \in [0, n], v_{i,0} = 0, \forall p \in [0, \text{poids}_{\text{max}}], v_{0,p} = 0$

$v_{i,p} = v_{i-1,p}$ si fst obj.$(i-1) > p$

$v_{i,p} = \max\big(v_{i-1,p}, \text{snd obj}.(i-1) + v_{i-1} + v_{i-1}, p - \text{fst obj}.(i-1)\big)$ sinon

Les suites récurrentes qui décrivent le problème correspondent souvent à découper le problème en sous-problèmes. Pour expliquer que certains sous-problèmes seront considérés plusieurs fois dans l'arbre des appels récursifs on dit que les sous-problèmes se chevauchent.

III - Première étape, un exemple

Vous êtes consultant pour une entreprise qui vend des barres de fer. Une étude de marché vient fixer des prix pour chaque longueur de barre de fer :

Longueur	0	1	2	3	4	...	K
prix	0	5	8	16	16	...	Prices[K]

Pour accéder au prix de la barre de longueur K on écrit prices$[K]$.

> Problème : l'usine livre une barre de taille N. Quel est le découpage optimal de la barre, c'est-à-dire celui qui maximise le prix de vente.

On note pour $K \in [0, N], p_K$ le prix de vente optimal d'une barre de longueur K.

Établir une équation de récurrence sur p_K :

- $p_0 = 0$
- Pour $K > 0, p_K = \max_{l \in [1,K]}(\text{prices}[l] + p_{K-l})$

Preuve

$\text{prices}[k]$: prix d'une barre de longueur k

$p_0 = 0$

$p_K = max_{l \in [1,K]}(\text{prices}[l] + p_{K-l})$

On montre par récurrence sur K que p_K est le prix de vente maximal d'une barre de longueur K.

Initialisation : Trivial

Hérédité : Soit $K > 0$ tel que l'hypothèse de récurrence HR soit vrai pour tout i

Soit $l \in [1, K]$, par HR p_{K-l} est le prix de vente optimal d'une barre de longueur K-l, donc il existe un découpage $K - l = n_0 + \ldots + n_P$ tel que $\sum_{i=0}^{p} \text{prices}[n_i] = p_{K-l}$.

Alors clairement le découpage $K = l + n_0 + \ldots + n_P$ réalise le prix de vente $\text{prices}[l] + p_{K-l}$. Donc $p_K \leq prix_{opti}$ pour K.

Réciproquement, soit $K = n_0 + \ldots + n_P$ un découpage optimal pour une barre de longueur K (existe car possibilités finies).

Alors le découpage $K - n_0 = n_1 + \ldots n_P$ est optimal. Si ce n'était pas le cas, en prenant un meilleur découpage $K - n_0 = {n_1}' + \ldots + {n_P}'$ on obtient un meilleur découpage pour K.

Donc $\sum_{i=1}^{P} \text{prices}[n_i] = p_K - n_0$.

Donc $prix_{opti} = \text{prices}[n_0] + p_K - n_0 \leq p_K$.

Mot-clé : Propriété de sous-problème optimal = une solution qui se construit en combinant des solutions optimales pour des sous-problèmes.

IV - Seconde étape

1. Version Descendante

Il s'agit de la version récursive, c'est la mémoïsation.

Illustration sur la vente de barres de fer :

L'équation de récurrence est donc connue.

```ocaml
open Hastbl;;

let price_opti prices n =
    (* 1. Création de la table *)
    let t = create () in

    (* 2. Cas de base *)
    add t 0 0;

    (* 3. Fonction aux *)
    let rec aux K = (* aux K = pK *)
```

```ocaml
        match find_opt t K with
        | None ->
            let p =
                max_list (List.init K (fun l->prices.(l+1) + aux (K-l-1)))
            in add t K p;
            p
        | Some p -> p
    in

    (* 4. On retourne la valeur souhaitée *)
    aux n;;

let max_list = List.fold_left max 0;;
```

```ocaml
let rec fold_left op acc = function
  | []   -> acc
  | h :: t -> fold_left op (op acc h) t
```

La fonction `fold_left` permet d'abréger toute fonction de cette forme :

```ocaml
let s = ref e in
for i = 0 to Array.length a - 1 do
    s := s f a[i];;
```

Squelette Générique

Écrit en `python`, traduit du `camlython`

```python
def version_desc(arg):
    # 1. On crée la table
    T = dict()
    # 2. Cas de base
    T[cas_de_base] = ...
    # 3. Fonction aux
    def aux (arg_sspb):
        if arg_sspb in T:
            return T[arg_sspb]
        else
            res = equation_de_recurrence(arg_sspb)
            T[arg_sspb] = res
            return res
    # 4. Valeur souhaitée
    aux (arg)
```

Variantes :

- Type de table : dictionnaire ou tableau de dimension N.
- Cas de base : Si table de dimension N > 1, il y a plusieurs cas de base.

- Possibilité de traiter tous les cas de base dans la fonction aux.

2. Version Ascendante = Impérative
Au lieu de vérifier si un calcul a déjà été mené (/sous-problème déjà résolu), on remplit toute la table dans le bon ordre systématiquement.

Le bon ordre est celui qui assure que pour remplir la case courante, on a déjà remplit les cases utilisées dans l'équation de récurrence.

On reprend l'exemple des barres de fer une fois de plus :

```ocaml
let barre_de_fer prices n =
    (* 1. Création de la table *)
    let t = Array.make (n+1) 0 in
    (* 2. Cas de base *)
    t.(0) <- 0;
    (* 3. On remplit la table dans l'ordre montant *)
    for k = 1 to n do
        t.(k) <- max_list (List.init (fun l -> prices.(l+1) + t.(k-l-1)))
    done;
    (* 4 Valeur souhaitée *)
    t.(n);;
```

desc : $p_k \to \text{aux } k$
asc : $p_k \to t.(k)$

Squelette Générique

```ocaml
let version_asc arg =
    (* 1. Création de la table *)
    let t = Array.make_matrix .... in
    (* 2. Cas de base *)
    t.(cas_de_base) <- val_init;
    (* 3. On remplit dans le bon ordre ascendant *)
    for i = 1 to .... do
        for j = 1 to .... do
            t.(i).(j) <- ....t.(k).(l)....
        done
    done
    (* 4. Valeur souhaitée *)
    t.(arg);;
```

Difficultés :

- Taille de la table : souvent (n+1) (p+1) à n et p sont les variables du problèmes
- Les cas de base : ne pas en oublier
- Trouver le bon ordre : $t.(k).(l)$ doit avoir déjà été remplit lorsqu'il est utilisé.

Version ascendante

```ocaml
let knapsack obj poids_max =
    let n = Array.length obj in

    (* Création de la table *)
    let t = Hashtbl.create () in

    (* Cas de base *)
    for i = 0 to n do
        for p = 0 to poids_max do
            Hashtbl.add t (i,p) 0
        done;
    done;

    let rec aux i p = match Hashtbl.find_opt t (i,p) with
        | None ->
            let res =
                max (Hashtbl.find t (i-1,p))
                    (snd obj.(i-1) + (Hashtbl.find (i-1) (p-(fst obj.(i-1)))));
            in Hashtbl.add t res;
            res;
        | Some v -> v
```

V - Optimisations Mémoires

1. Fibonacci

$u_0 = u_1 = 1$

$u_{n+2} = u_{n+1} - u_n$

Version descendante

```python
def fibo(n):
    T = {}
    T[0] = 1
    T[1] = 1
    def aux(k):
        if k not in T:
            T[k] = aux(k-1) - aux(k-2)
        return T[k]
    return aux(n)
```

Ici on remplit le dictionnaire à la demande, on ne remplit que ce dont on a besoin.

Version ascendante

```python
def fibo(n):
    T = [0]*(n+1)
    T[0] = 1
    T[1] = 1
```

```python
    for i in range(2, n+1):
        T[i] = T[i-1] - T[i-2]
    return T[n]
```

On remarque qu'on aurait pu utiliser 2 variables au lieu de toute une liste.

```python
def fibo(n):
    u_prec = 1
    u = 1
    for i in range(2, n+1):
        tmp = u_prec
        u_prec = u
        u = u + tmp
    return u
```

On a ainsi l'invariant suivant : $u = fibo(i-1)$ et $uprec = fibo(i-2)$.

On a ainsi un coût d'espace constant bien que l'on reste on coût temporel linéaire.

$\Rightarrow$ La version ascendante peut permettre de gagner en espace.

2. Sac à dos
Version impératif TODO

Sur un exemple : $\{(1,5),(3,5),(5,8),(8,12)\}$ avec $p_{max} = 10$

Table des $v_{i,p}$

p 0	1	2	3	4	
0	X	X	X	X	X
1	0				
2	0	5	5	5	
3	X				
4	0				
5	0	5	10		
6	0				
7	0	5			
8	1				
9	0				
10	0	5	10	18	18

Version ascendante : On remplit $(poids_{max} + 1) \times (n+1)$ cases

Version descendante : Potentiellement beaucoup moins

$\Rightarrow$ gain en temps (difficile à mesurer dans le pire des cas)

$\Rightarrow$ gain en espace ?? $\rightarrow$ Cela dépend de l'implémentation des dictionnaires, ce n'est pas si évident.

VI - Reconstruction de la Solution

Les programmes écrits pour le problème du sac à dos donnent la valeur optimale du sac à dos mais pas comment l'atteindre.

L'arbre de décision se lit dans la table obtenue à la fin de l'algorithme.

Pour reconstruire la solution on conserve la table et on la parcourt "à l'envers".

> Ici le "18" de la case $t.(10).(4)$ a été obtenu comme $max(t.(10).(3), t.(2).(3) + 12)$. Comme c'est $t.(10).(3)$ qui donne sa valeur au max, l'objet 4 n'est pas choisi dans la solution.

> Puis on continue.

VII - TD

1. Optimisation mémoire

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

Version ascendante triangle de Pascal

```ocaml
let pascal k n =
    let t = Hashtbl.create 1 in

    let rec aux k n =
        if k > n || n < 0 then 0
        if k = 0 then 1
        if k = 1 then n
        match Hashtbl.find_opt t (k,n) with
        | Some v -> v
        | None -> begin
            let res = (aux (n-1) (k-1)) + (aux (n-1) k) in
            Hashtbl.add t (k,n) res;
            res
        end;
    in aux k n;;
```

Version ascendante

Un transforme notre parallélogramme en rectangle :

$mi, j = m_{i(j-1)} + m_{(i-1),j}$

$m_{0,j} = m_{i,0} = 1$

```ocaml
let pascal k n =
    let t = Array.make_matrix (k+1) (n-k) 0 in
    for i = 0 to n do
        t.(0).(i) <- 1
```

On peut ainsi se ramener à un problème plus classique que nous savons déjà implémenter.

Amélioration de la version ascendante pour être en O(k)

On applique l'algorithme sur un tableau de taille k.

```ocaml
let pascal k n =
    let t = Array.make (k+1) 1 in
    (* Cas de base déjà fait *)

    for i = 0 to n-1 do (* Lignes *)
        (* Invariant: t.(j) = j parmi i pour j dans [0,i] *)
        for j = min k (i-1) downto 1 do
            t.(j) <- t.(j) + t.(j-1)
        done;
    done;
    t.(k);;
```

Pour le sac à dos

La même astuce permet d'obtenir un coût linéaire de mémoire.

2. Trouver et Prouver des Formules de Récurrences

2.1 Vente de Barres de Fer

Preuve dans la partie **III** de cours.

2.2 Distance d'édition : Levenshtein

Formule de récurrence

$d_{i,0} = i$

$d_{0,j} = j$

$d_{i,j} = d_{i-1,j-1}$ si $t1[i-1] = t2[j-1]$

$d_{i,j} = min(d_{i-1,j} + 1, 1 + d_{i,j-1}) = 1 + min(d_{i-1,j}, d_{i,j-1})$ sinon

Le minimum fait intervenir d'un côté la suppression du caractère suivi de l'application de l'algorithme avec le reste du mot. De l'autre côté il fait intervenir l'ajout du caractère avant de continuer.

Avec remplacement

$d_{i,j} = 1 + min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1})$