Programmation dynamique

Thibaud Lemanissier

PC Lycée Carnot

2026-2027

Calcul de coefficients binomiaux

Méthode naïve

Pour un coefficient binomial $\binom{n}{k}$, on peut procéder récursivement :

- Cas de base : si $k = 0$ ou n, alors $\binom{n}{k} = 1$;
- Réduction à un problème plus petit : si $0 < k < n$, alors $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.

Méthode naïve

Pour un coefficient binomial $\binom{n}{k}$, on peut procéder récursivement :

- Cas de base : si $k = 0$ ou n, alors $\binom{n}{k} = 1$;
- Réduction à un problème plus petit : si $0 < k < n$, alors $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.

La relation de récurrence reliant un problème avec ses sous problèmes sera appelée **équation de Bellman** (en l'honneur de Richard Bellman qui a donné son nom à la programmation dynamique).

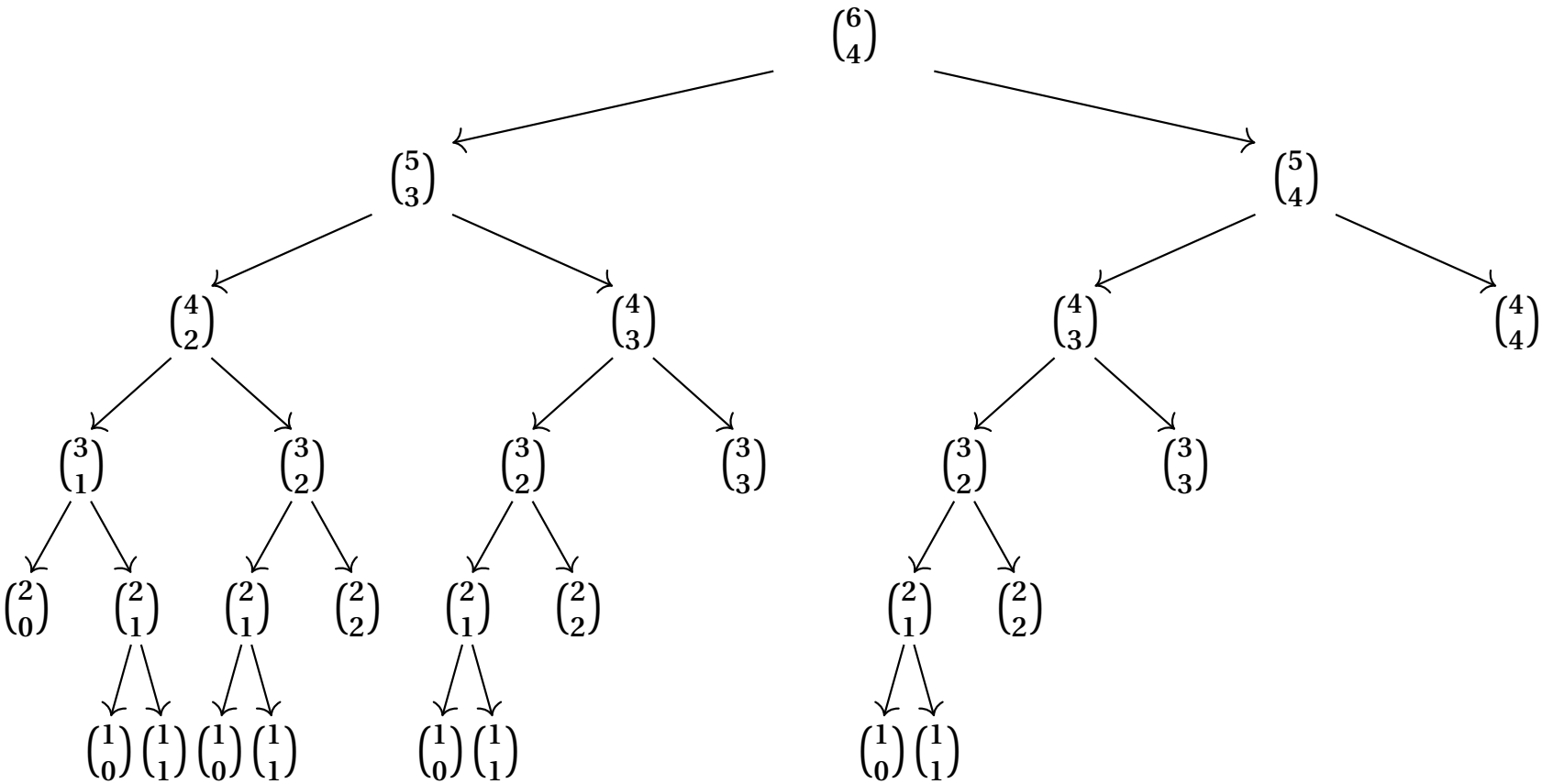
Méthode naïve

On en déduit donc la fonction récursive suivante :

```
1 def binome(n : int, k : int) -> int :
2     if k == 0 or k == n :
3         return 1
4     else :
5         return binome(n-1,k-1)+binome(n-1,k)
```

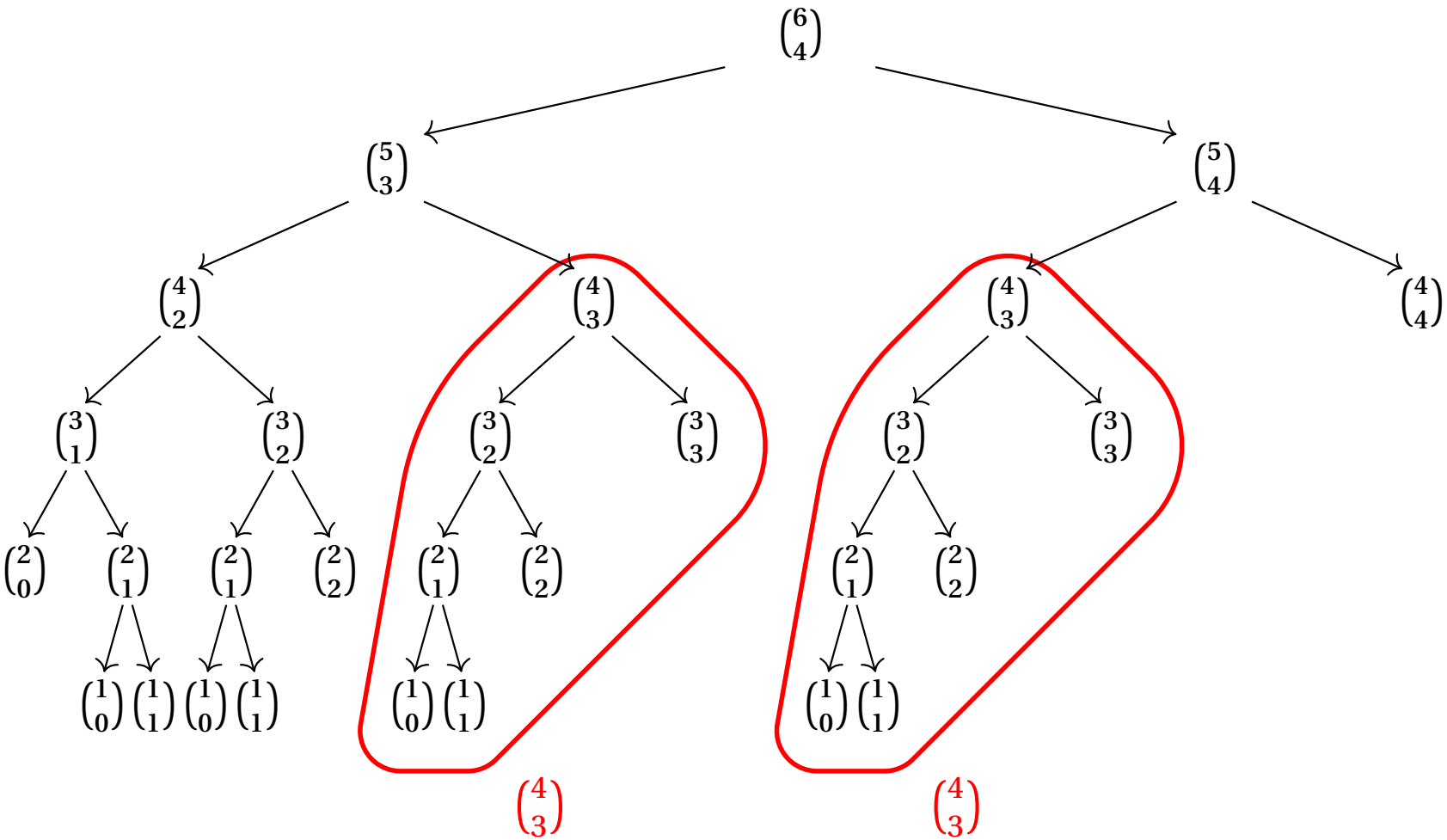
Méthode naïve

Voici le représentation des appelles récursifs dans le cas du calcul de $\binom{6}{4}$:



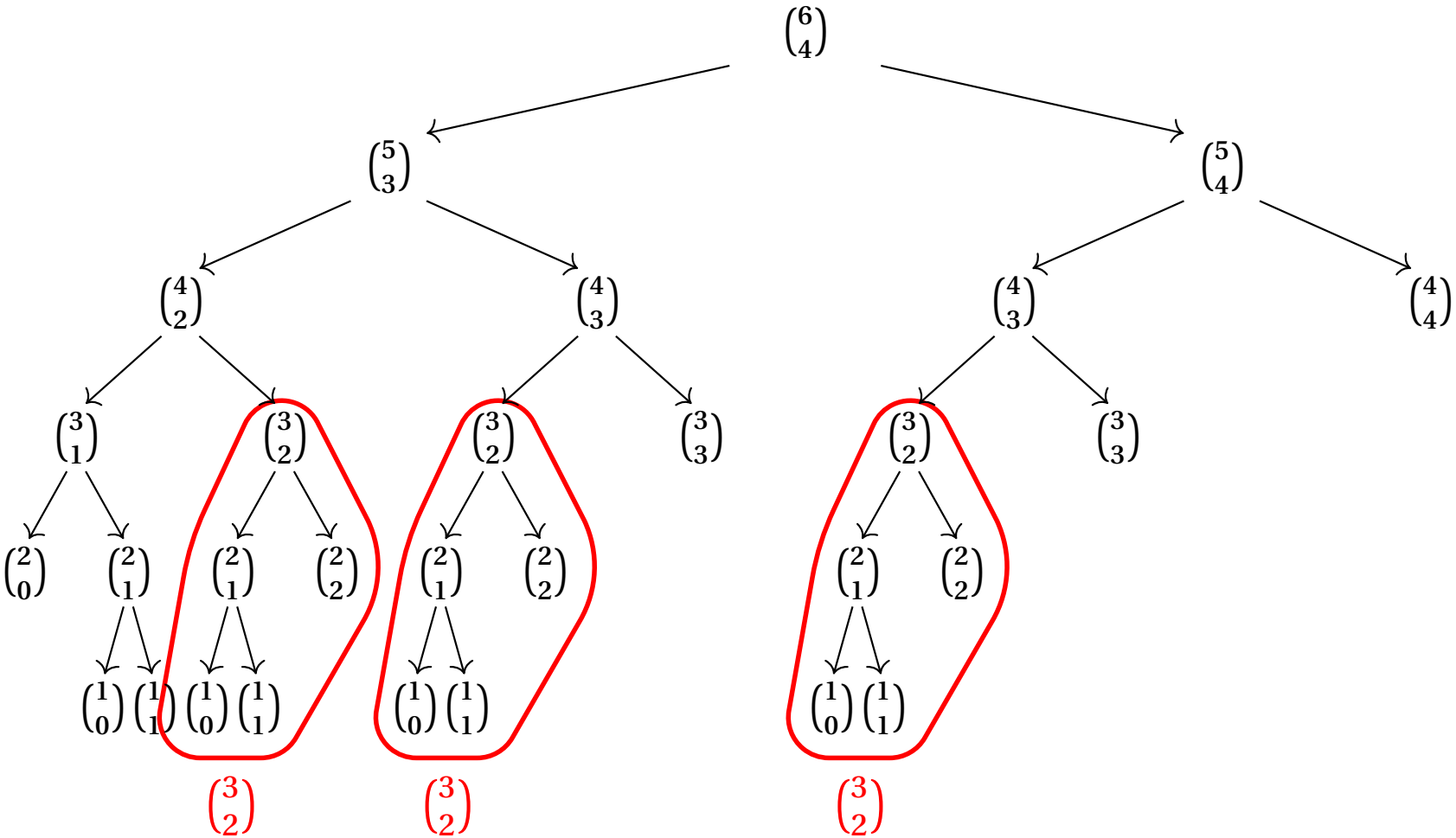
Méthode naïve

Plusieurs étapes se répètent plusieurs fois, on dit qu'il y a **chevauchement** (ou **recouvrement**) des sous-problèmes :



Méthode naïve

Plusieurs étapes se répètent plusieurs fois, on dit qu'il y a **chevauchement** (ou **recouvrement**) des sous-problèmes :



Programmation dynamique

On peut s'inspirer de ce que l'on fait lorsque l'on fait le calcul à la « main ».

Programmation dynamique

On peut s'inspirer de ce que l'on fait lorsque l'on fait le calcul à la « main ».

$n \backslash k$	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	15

Programmation dynamique

On peut s'inspirer de ce que l'on fait lorsque l'on fait le calcul à la « main ».

n \ k	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	15

Nous allons donc utiliser une liste de listes pour stocker les valeurs calculées au fur et à mesure.

Programmation dynamique

On obtient donc le code suivant :

```
def binome_dyn(n, k) :
    L = [[1]+[0]*k for i in range(n+1)]
    for i in range(1, n+1) :
        for j in range(1, k+1) :
            L[i][j] = L[i-1][j-1]+L[i-1][j]
    return L[n][k]
```

Programmation dynamique

On obtient donc le code suivant :

```
def binome_dyn(n, k) :
    L = [[1]+[0]*k for i in range(n+1)]
    for i in range(1, n+1) :
        for j in range(1, k+1) :
            L[i][j] = L[i-1][j-1]+L[i-1][j]
    return L[n][k]
```

Compléxité : $\mathcal{O}(n \times k)$.

Mémoïsation

Avec la fonction précédente, on a besoin de faire 24 étapes de calcul pour obtenir $\binom{6}{4}$.

On remarque que seulement une partie des coefficients du tableau sont utiles pour calculer $\binom{6}{4}$.

n \ k	0	1	2	3	4
0	1	0	0	0	0
1	1	1	0	0	0
2	1	2	1	0	0
3	1	3	3	1	0
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	15

Mémoïsation

Ce problème est dû au fait que l'on commence de problème le plus simple pour aller vers le problème le plus compliqué on dit qu'on procède de **bas en haut** (souvent anglicisé **bottom-up**).

Mémoïsation

Ce problème est dû au fait que l'on commence de problème le plus simple pour aller vers le problème le plus compliqué on dit qu'on procède de **bas en haut** (souvent anglicisé **bottom-up**).

Pour le résoudre, nous allons procéder dans le sens inverse du problème le plus complexe vers le plus simple (**top-down**). On dit qu'on procède par **mémoïsation**.

Mémoïsation

Ce problème est dû au fait que l'on commence de problème le plus simple pour aller vers le problème le plus compliqué on dit qu'on procède de **bas en haut** (souvent anglicisé **bottom-up**).

Pour le résoudre, nous allons procéder dans le sens inverse du problème le plus complexe vers le plus simple (**top-down**). On dit qu'on procède par **mémoïsation**.

Pour cela, nous allons à nouveau procéder par récursivité en utilisant un dictionnaire pour stocker les valeurs déjà calculer. Nous allons donc devoir tester si la valeur a déjà été calculée.

Mémoïsation

On peut soit utiliser une fonction auxiliaire :

```
def bin_memo(n : int, k : int) -> int :
    d = {}
    def aux(n : int, k : int) -> int :
        if (k,n) in d :
            return d[(k,n)]
        if k == 0 or k == n :
            d[(n, k)] = 1
        else :
            d[(n, k)] = aux(n-1, k-1) + aux(n-1, k)
        return d[(n, k)]
    return aux(n, k)
```

Mémoïsation

Soit ajouter un argument :

```
1 def bin_m(n : int, k : int, d : dict) -> int :
2     if (k, n) in d :
3         return d[(k, n)]
4     if k == 0 or k == n :
5         d[(n, k)] = 1
6     else :
7         d[(n, k)] = bin_m(n-1, k-1) + bin_m(n-1, k)
8     return d[(n, k)]
```

Synthèse

Pour faire de la programmation dynamique pour calculer une valeur, on a donc besoin des trois ingrédients suivants :

- Un cas de base où la valeur est facile à calculer ;
- Une relation de récurrence permettant de réduire le calcul d'un cas à un cas plus simple (l'équation de Bellman) ;
- Une variable permettant de stocker les valeurs plus simples déjà calculées.

Complément sur les dictionnaires

Implémentation d'un dictionnaire

En python un dictionnaire est implémenté via une **table de hachage**.

Implémentation d'un dictionnaire

En python un dictionnaire est implémenté via une **table de hachage**.

Table de hachage

Un **table de hachage** est constitué :

- une **tableau** contenant des valeurs ;
- une **fonction de hachage** h telle que si k est une clé, $h(k)$ est l'indice de la valeurs associée à la clé k.

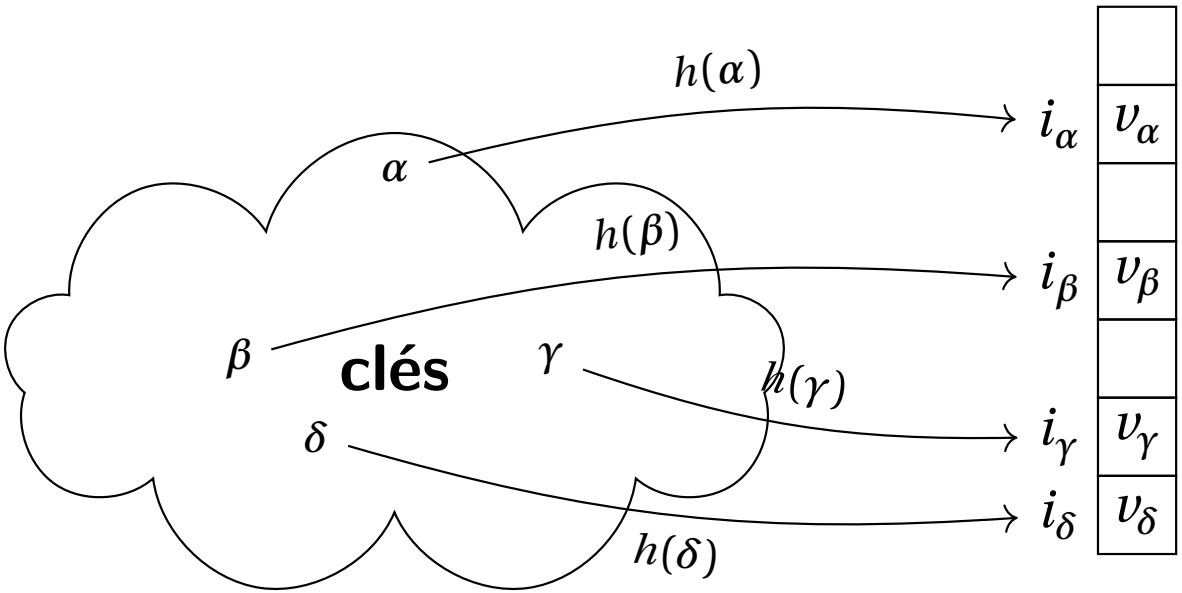
Implémentation d'un dictionnaire

En python un dictionnaire est implémenté via une **table de hachage**.

Table de hachage

Un **table de hachage** est constitué :

- une **tableau** contenant des valeurs ;
- une **fonction de hachage** h telle que si k est une clé, $h(k)$ est l'indice de la valeurs associée à la clé k.



Implémentation d'un dictionnaire

En python un dictionnaire est implémenté via une **table de hachage**.

Table de hachage

Un **table de hachage** est constitué :

- une **tableau** contenant des valeurs ;
- une **fonction de hachage** h telle que si k est une clé, $h(k)$ est l'indice de la valeurs associée à la clé k.

La fonction de hachage doit

- pouvoir prendre en argument une grande gamme d'objet et renvoyer des indices dans un ensemble d'entier petit ;
- être exécutable en cout constant d'opération ($\mathcal{O}(1)$).

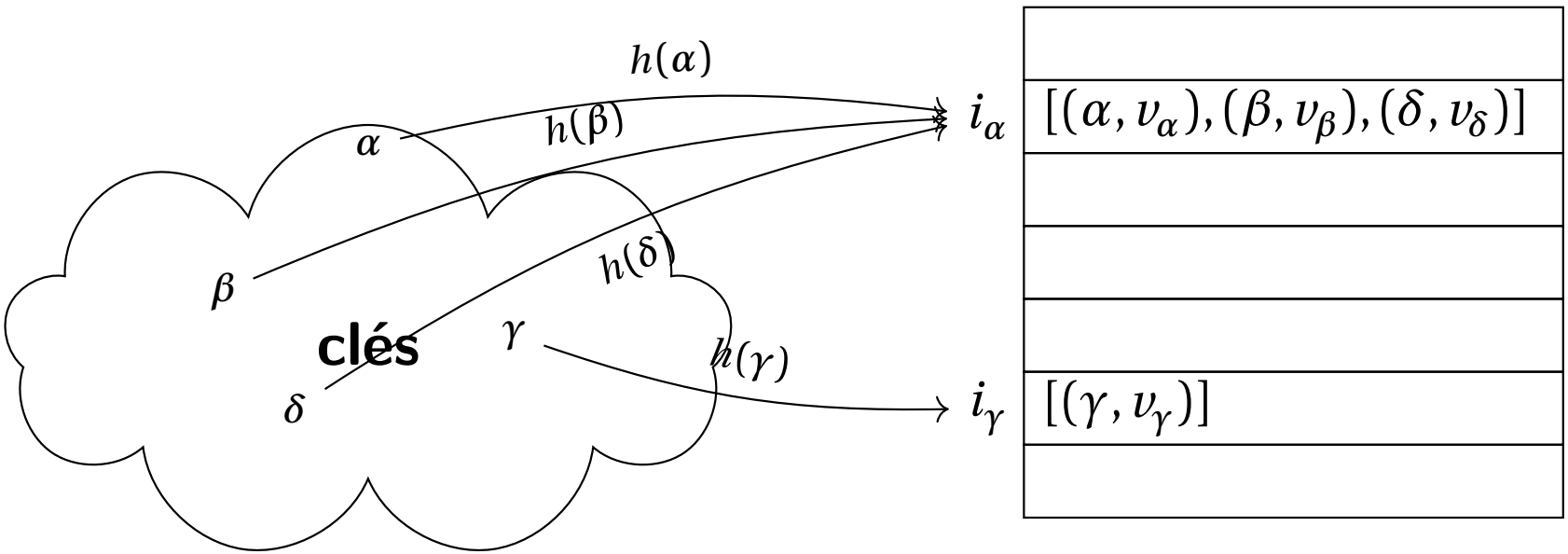
Pour éviter que l'image d'une clé par la fonction puisse changer, les arguments d'une fonction de hachage doivent non mutable. Donc pas de liste ou de table mais par contre, on peut mettre des chaînes de caractères ou des n-uplet.
La fonction de hachage utilisé par python pour les dictionnaires et `hash`.

Collision et chainage

L'ensemble des clés possible est beaucoup plus grand que l'ensemble d'indice obtenu après hachage. Si deux clés correspondent aux mêmes indices, on dit qu'il y a **colision**.

Collision et chainage

L'ensemble des clés possible est beaucoup plus grand que l'ensemble d'indice obtenu après hachage. Si deux clés correspondent aux mêmes indices, on dit qu'il y a **colision**.
Pour résoudre le problème de collision, python procède à **chainage** en stockant une liste à chaque position de la table de hachage.



à retenir sur les dictionnaires

Instruction Python	Opération effectuée	Complexité
`d = {}`	création d'un dictionnaire vide	$\mathcal{O}(1)$
`dico[key]`	accéder à la valeur associée à la clef `key`	$\mathcal{O}(1)$
`dico[key]=value1`	ajouter/modifier un élément du dictionnaire	$\mathcal{O}(1)$
`for x in dico`	parcours d'un dictionnaire	$\mathcal{O}(n)$
`key in dico`	test d'appartenance	$\mathcal{O}(1)$

Problème du sac à dos

Rappel du problème

La programmation dynamique permet aussi de résoudre certain problème d'optimisation comme le problème du sac à dos.

Rappel du problème

La programmation dynamique permet aussi de résoudre certain problème d'optimisation comme le problème du sac à dos.
Problème : On dispose d'une liste d'objets qui ont une masse et une valeur que l'on représente par une liste de couples $[(m_0, v_0), \dots, (m_{n-1}, v_{n-1})]$ (on suppose pour simplifier que $m_i \in \mathbb{N}^*$) et d'un sac à dos qui peut contenir une masse limite à M. On souhaite choisir parmi ces objets une sous-liste telle que :

- La somme des masses des objets choisis soit inférieures à M ;
- La somme des valeurs des objets choisis soit maximal (en respectant la contrainte précédente).

Algorithme glouton

Vous avez vu l'an dernier qu'une possibilité pour résoudre ce problème était de choisir les objet à mettre dans le sac un à un en choisissant parmi les objets de masses inférieures à ce que l'on peut mettre dans le sac celui qui a le ratio v_i/m_i le plus élevé.

Algorithme glouton

Vous avez vu l'an dernier qu'une possibilité pour résoudre ce problème était de choisir les objet à mettre dans le sac un à un en choisissant parmi les objets de masses inférieures à ce que l'on peut mettre dans le sac celui qui a le ratio v_i/m_i le plus élevé. Cet algorithme est donc **glouton** puisque à chaque étape on effectue le choix qui nous semble optimal.

Algorithme glouton

On peut donc le coder de la manière suivante :

```
1  def indice_max(L : list) -> int :
2      i_max = 0 # indice ratio max
3      r_max = L[0][1]/L[0][0] #  ratio max
4      for i in range(len(L)) :
5          r = L[i][1]/L[i][0]
6          if r > r_max :
7              i_max, r_max = i, r
8      return i_max
9
10 def sac_glouton(M_max : int, objets : list) -> list :
11     sac = [] # création du sac
12     objets = [l for l in objets if l[0] <= M_max]    #  →
           création de la liste des objets utiles
13     while len(objs_ut) > 0 :
14         i_max = indice_max(objets)
15         sac.append(objs_ut.pop(i_max))
16         M_max -= sac[-1][0]
17         objets = [l for l in objets if l[0] <= M_max]
18     return sac
```

Algorithme glouton

On peut les trier par ordre décroissant de valeur massique :

```
1  def sac_glouton(M_max : int, objets : list) -> list :
2      sac = [] # création du sac
3      objs_ut = [l for l in objets if l[0] <= M_max]  #  →
           création de la liste des objets utiles
4      def ratio(objet) :
5          return objet[1]/objet[0]
6      objs_ut = sorted(objs_ut, key = ratio, reverse = True) #  →
           tri de la liste par valeur massique décroissante
7      for objet in objs_ut :
8          if objet[0] <= M_max :
9              M_max -= objet[0]
10             sac.append(objet)
11     return sac
```

Algorithme glouton

Cette approche permet d'obtenir une solution en un temps très court, par contre elle ne garanti pas l'optimalité de la solution proposée.

Algorithme glouton

Cette approche permet d'obtenir une solution en un temps très court, par contre elle ne garanti pas l'optimalité de la solution proposée.

Par exemple, si on applique la fonction précédente à la liste d'objets $[(6,15),(25,60),(5,10),(8,7),(20,10)]$ et au sac de masse maximale 30, on obtient comme remplissage de sac la liste $[(6,15),(5,10),(8,7)]$, ce qui correspond à une valeur totale de 32.

Algorithme glouton

Cette approche permet d'obtenir une solution en un temps très court, par contre elle ne garanti pas l'optimalité de la solution proposée.

Par exemple, si on applique la fonction précédente à la liste d'objets $[(6,15),(25,60),(5,10),(8,7),(20,10)]$ et au sac de masse maximale 30, on obtient comme remplissage de sac la liste $[(6,15),(5,10),(8,7)]$, ce qui correspond à une valeur totale de 32.
Or on peut remarque qu'en choisissant $[(25,60),(5,10)]$ à mettre dans le sac, on obtient une valeur totale de 70.

Analyse pour une programmation dynamique

Nous allons d'abord déterminer la valeur maximal que l'on peut emporter par programmation dynamique.

Analyse pour une programmation dynamique

Nous allons d'abord déterminer la valeur maximal que l'on peut emporter par programmation dynamique.

On remarque que si on n'a pas d'objet, le problème est facile. Nous allons procéder par récurrence sur le nombre d'objet que l'on considère. On note $V(m, i)$ la valeur maximal que l'on peut emporter dans un sac de capacité maximal m lorsque l'on dispose des i premiers objets.

Analyse pour une programmation dynamique

Nous allons d'abord déterminer la valeur maximal que l'on peut emporter par programmation dynamique.

On remarque que si on n'a pas d'objet, le problème est facile. Nous allons procéder par récurrence sur le nombre d'objet que l'on considère. On note $V(m,i)$ la valeur maximal que l'on peut emporter dans un sac de capacité maximal m lorsque l'on dispose des i premiers objets.

On remarque que si l'on considère un objet supplémentaire (m_i, v_i), alors on peut calculer la nouvelle valeur maximal en fonction de si on ajoute l'objet ou non grâce à la formule suivante :

$$V(m,i) = \max\{V(m,i-1), V(m-m_i, i-1) + v_i\}$$

Cette égalité est dû au fait que dans ce problème une sous-structure d'une structure optimale est optimale.

Analyse pour une programmation dynamique

Nous avons tous les ingrédents pour procéder par programmation dynamique :

- Cas de base : $V(m,0) = 0$;
- Équation de Bellman : $V(m,i) = \max\{V(m,i-1), V(m-m_i,i-1)+v_i\}$;
- Variable pour stocker : V un tableau de taille $M+1$ par $m+1$.

Algorithme programmation dynamique pour la valeur maximal

On peut donc le coder de la manière suivante :

```
def vals_max_dyn(M_max : int, objets : list) -> list :
    n = len(objets)
    V = [[0]*(n+1) for k in range(M_max+1)]
    for m in range(1, M_max+1) :
        for i in range(1, n+1) :
            m_ob, v_ob = objets[i][0], objets[i][1]
            if m_ob <= m :
                V[m][i] = max(V[m-m_ob][i-1]+v_ob,V[m][i-1])
    return V
```

La valeur maximal est alors contenue dans `V[M_max][len(objets)]`.

Reconstruction de la configuration optimale

On peut alors reconstruire la configuration optimale à partir de `V` :

```
def recon_dyn(M_max : int, objets : list, V : list) -> list :
    sac = []
    for i in range(n, 0, -1) :
        if V[M_max][i] > V[M_max][i-1] :
            sac.append(objets[i-1])
            M_max -= objets[i-1][0]
    return sac
```

Et donc obtenir la configuration optimale :

```
def config_max_dyn(M_max : int, objets : list) -> list :
    V = vals_max_dyn(M_max, objets)
    return recon_dyn(M_max, objets, V)
```

Analyse par mémoïsation

Nous avons tous les ingrédents pour procéder par programmation dynamique :

- Cas de base : $d[(m,0)] = 0$;
- Équation de Bellman : $d[(m,i)] = \max\{d[(m,i-1)], d[(m-m_i,i-1)] + v_i\}$;
- Variable pour stocker : d un dictionnaire.

Algorithme par mémoïsation pour la configuration maximal

On peut donc le coder de la manière suivante :

```
def vals_max_memo(M_max : int, objets : list) -> dict :
    d = {}
    def aux(m : int, i : int) -> float : # m est la masse maximale et i le rang du dernier objet considéré
        if i == 0 :
            d[(m, i)] = 0
        elif (m, i) not in d :
            m_ob, v_ob = objets[i-1][0], objets[i-1][1]
            if m_ob <= m :
                d[(m, i)] = max(aux(m-m_ob, i-1)+v_ob, aux(m, i-1))
            else :
                d[(m, i)] = aux(m, i-1)
        return d[(m, i)]
    aux(M_max, len(objets))
    return d
```

La valeur maximal est alors contenue dans `V[(len(objets), M_max)]`.

Reconstruction de la configuration optimale

On peut alors reconstruire la configuration optimale à partir de `V` :

```
def recon_memo(M_max : int, objets : list, d : dict) -> list :
    sac = []
    for i in range(len(objets), 0, -1) :
        if d[(M_max, i)] > d[(M_max, i-1)] :
            sac.append(objets[i-1])
            M_max -= objets[i-1][0]
    return sac
```

Et donc obtenir la configuration optimale :

```
def config_max_memo(M_max : int, objets : list) -> list :
    d = vals_max_memo(M_max, objets)
    return recon_memo(M_max, objets, d)
```

Synthèse

On utilise la programmation dynamique lorsque l'on s'intéresse à un problème d'optimisation qui vérifie les trois conditions suivante :

- Résoudre un problème se ramène à résoudre des sous-problèmes plus simple ;
- Ces sous-problèmes peuvent se chevaucher (sinon, on utilise plutôt une stratégie « divide and conquert » comme pour le tri fusion par exemple) ;
- Une sous-solution d'une solution optimal est elle même optimale.