POLYTECH MONTPELLIER

Algorithmique Avancée

Programmation Dynamique

DO5 - 2024

Enseignant : Areski Himeur

areski.himeur@umontpellier.fr

Sur la base de la présentation de Marin Bougeret

Sommaire

Objectifs du cours

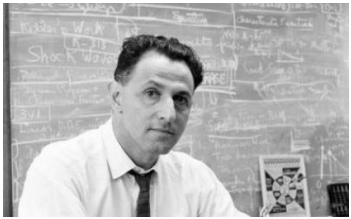
- Identifier les problèmes et algorithmes **qui peuvent ou non utiliser les principes de la programmation dynamique** ;
- Apprendre les méthodes de programmation dynamique utilisées pour des problèmes connus : **Fibonacci, KnapSack, Plus Court Chemin, etc.** ;
- Proposer **pour un nouveau problème donné un algorithme récursif** pouvant utiliser les principes de la programmation dynamique ;
- Transformer **un algorithme récursif en un algorithme de programmation dynamique** ;
- Comprendre et utiliser les liens et différences entre **récursif et itératif** en programmation dynamique ;
- Analyse de la **complexité en temps et en espace** des méthodes proposées.

Sommaire

Présentation

Qu'est-ce que la programmation dynamique ?

La programmation dynamique est une méthode algorithmique pour résoudre des problèmes d'optimisation. Le concept a été introduit au début des années 1950 par Richard Bellman.



Richard Bellman (1920-1984)

Comment fonctionne la programmation dynamique ?

La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes **en stockant les résultats intermédiaires**.

Comment fonctionne la programmation dynamique ?

La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes **en stockant les résultats intermédiaires**.

- Elle s'appuie sur le principe qu'une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes.

Comment fonctionne la programmation dynamique ?

La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes **en stockant les résultats intermédiaires**.

- Elle s'appuie sur le principe qu'une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes.
- Elle évite les calculs redondants des mêmes sous-problèmes, contrairement à la méthode diviser pour régner naïve.

Sommaire

La suite de Fibonacci

La *suite de Fibonacci* est une suite d'entiers naturels définie par la relation de récurrence suivante :

$$F_n = F_{n-1} + F_{n-2}$$

Les deux premiers termes sont $F_0 = 0$ et $F_1 = 1$.

La suite de Fibonacci

La *suite de Fibonacci* est une suite d'entiers naturels définie par la relation de récurrence suivante :

$$F_n = F_{n-1} + F_{n-2}$$

Les deux premiers termes sont $F_0 = 0$ et $F_1 = 1$.

Exercices

- Présenter un algorithme récursif simple pour calculer F_n.

La suite de Fibonacci

La *suite de Fibonacci* est une suite d'entiers naturels définie par la relation de récurrence suivante :

$$F_n = F_{n-1} + F_{n-2}$$

Les deux premiers termes sont $F_0 = 0$ et $F_1 = 1$.

Exercices

- Présenter un algorithme récursif simple pour calculer F_n.
- Dessiner un graphe représentant les appels récursifs pour le calcul de F_4.

La suite de Fibonacci

La *suite de Fibonacci* est une suite d'entiers naturels définie par la relation de récurrence suivante :

$$F_n = F_{n-1} + F_{n-2}$$

Les deux premiers termes sont $F_0 = 0$ et $F_1 = 1$.

Exercices

- Présenter un algorithme récursif simple pour calculer F_n.
- Dessiner un graphe représentant les appels récursifs pour le calcul de F_4.
- Évaluer la complexité de l'algorithme.
- Peut-on faire mieux ? Comment ?

Fibonacci et mémoïsation

La méthode naïve consiste à utiliser la définition récursive et à calculer chaque terme à partir des deux précédents.

Fibonacci et mémoïsation

La méthode naïve consiste à utiliser la définition récursive et à calculer chaque terme à partir des deux précédents. Cette méthode a une complexité temporelle exponentielle, car elle effectue de nombreux **calculs redondants**. Pour calculer F_5, il faut calculer F_4 et F_3, mais pour calculer F_4, il faut aussi calculer F_3 et F_2, etc.

Idée

Mémoriser les résultats intermédiaires pour éviter de refaire des calculs déjà faits.

Exercices

- Proposer un nouvel algorithme qui implémente cette idée de *mémoïsation*.

Fibonacci et mémoïsation

La méthode naïve consiste à utiliser la définition récursive et à calculer chaque terme à partir des deux précédents. Cette méthode a une complexité temporelle exponentielle, car elle effectue de nombreux **calculs redondants**. Pour calculer F_5, il faut calculer F_4 et F_3, mais pour calculer F_4, il faut aussi calculer F_3 et F_2, etc.

Idée

Mémoriser les résultats intermédiaires pour éviter de refaire des calculs déjà faits.

Exercices

- Proposer un nouvel algorithme qui implémente cette idée de *mémoïsation*.
- Évaluer la complexité de l'algorithme.

Complexité d'un algorithme de programmation dynamique

Théorème Complexité en Temps

La complexité d'une DP (qui termine) basée sur un tableau t est

$$\mathcal{O}\Big(\sum_{\text{i case de t}} c(i)\Big)$$

avec $c(i)$ la complexité de l'appel qui remplit la case i et $\mathcal{O}(1)$ pour chaque appel récursif.

Autrement dit, on compte comme si l'on calculait chaque case du tableau une seule fois, et que les appels récursifs comptaient $\mathcal{O}(1)$.

Application à FibDP

```
int FibDP(n){
  if(t[n]==-inf){
    int res;
    if (n<=2) res = 1;
    else res = FibDP(n-1)+FibDP(n-2); //O(1)
    t[n]=res;
  }
  return t[n];
}
```

Complexité en temps de `FibDP(n)`: $taille(t) = \mathcal{O}(n)$, $c = 1 \Rightarrow \mathcal{O}(n)$
Mémoire utilisée : $\mathcal{O}(n)$

Sommaire

Problème du Sac à Dos

Le *problème du sac à dos* est un problème d'optimisation combinatoire qui consiste à choisir un ensemble d'objets de valeur et de poids donnés, de manière à maximiser la valeur totale des objets choisis sans dépasser une capacité.

Formalisation

Étant donné n objets numérotés de 1 à n, chacun ayant une valeur v_i et un poids w_i, et une capacité maximale W, trouver un sous-ensemble d'objets S tel que :

$$i \in S \sum v_i \text{ soit maximal et } i \in S \sum w_i \leq W$$

Exercice

- Comment utiliser la programmation dynamique pour proposer un algorithme efficace ?

Sac à Dos / Knapsack

Définition du problème

- **Entrée :** un entier C, et deux tableaux d'entiers p et v de taille n
  - C représente la taille du sac à dos ;
  - l'objet i occupe une place $p[i]$ dans le sac, mais rapporte une valeur $v[i]$.
- **Sortie :** $S \subseteq [0 : n-1]$ d'objets tenant dans le sac ($p(S) \leq C$, avec $p(S) = \sum_{i \in S} p[i]$)
- **Fonction objectif :** maximiser $v(S)$, avec $v(S) = \sum_{i \in S} v[i]$

Sac à Dos / Knapsack

Définition du problème

- **Entrée :** un entier C, et deux tableaux d'entiers p et v de taille n
  - C représente la taille du sac à dos ;
  - l'objet i occupe une place $p[i]$ dans le sac, mais rapporte une valeur $v[i]$.
- **Sortie :** $S \subseteq [0 : n-1]$ d'objets tenant dans le sac ($p(S) \leq C$, avec $p(S) = \sum_{i \in S} p[i]$)
- **Fonction objectif :** maximiser $v(S)$, avec $v(S) = \sum_{i \in S} v[i]$

- Ce problème est NP-difficile

Sac à Dos / Knapsack

Définition du problème

- **Entrée :** un entier C, et deux tableaux d'entiers p et v de taille n
  - C représente la taille du sac à dos ;
  - l'objet i occupe une place $p[i]$ dans le sac, mais rapporte une valeur $v[i]$.
- **Sortie :** $S \subseteq [0 : n-1]$ d'objets tenant dans le sac ($p(S) \leq C$, avec $p(S) = \sum_{i \in S} p[i]$)
- **Fonction objectif :** maximiser $v(S)$, avec $v(S) = \sum_{i \in S} v[i]$

- Ce problème est NP-difficile
- Brute force (essayer tous les sous-ensembles) : coûterait (au moins) 2^n

Sac à Dos / Knapsack

Définition du problème

- **Entrée :** un entier C, et deux tableaux d'entiers p et v de taille n
  - C représente la taille du sac à dos ;
  - l'objet i occupe une place $p[i]$ dans le sac, mais rapporte une valeur $v[i]$.
- **Sortie :** $S \subseteq [0 : n-1]$ d'objets tenant dans le sac ($p(S) \leq C$, avec $p(S) = \sum_{i \in S} p[i]$)
- **Fonction objectif :** maximiser $v(S)$, avec $v(S) = \sum_{i \in S} v[i]$

- Ce problème est NP-difficile
- Brute force (essayer tous les sous-ensembles) : coûterait (au moins) 2^n
- On va le résoudre en $\mathcal{O}(nC)$: ça n'est que pseudo polynomial (il faudrait $log(C)$ pour être polynomial), mais c'est tout de même mieux !

Utilisation de la Programmation Dynamique

Rappel sur le fonctionnement de la programmation dynamique

La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes **en stockant les résultats intermédiaires**.

- Elle s'appuie sur le principe qu'une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes.
- Elle évite les calculs redondants des mêmes sous-problèmes, contrairement à la méthode diviser pour régner naïve.

Utilisation de la Programmation Dynamique

Rappel sur le fonctionnement de la programmation dynamique

La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes **en stockant les résultats intermédiaires**.

- Elle s'appuie sur le principe qu'une solution optimale d'un problème s'obtient en combinant des solutions optimales à des sous-problèmes.
- Elle évite les calculs redondants des mêmes sous-problèmes, contrairement à la méthode diviser pour régner naïve.

Notre Objectif

$\rightarrow$ Trouver une "bonne" façon récursive de résoudre le problème.

Un Algorithme Récursif pour le Sac à Dos

Prendre ou ne pas prendre, là n'est pas la question...

- On considère le premier objet ;
- S'il est trop gros, on passe au suivant ;

Un Algorithme Récursif pour le Sac à Dos

Prendre ou ne pas prendre, là n'est pas la question...

- On considère le premier objet ;
- S'il est trop gros, on passe au suivant ;
- Sinon, on ne sait pas à priori s'il faut le prendre,

Un Algorithme Récursif pour le Sac à Dos

Prendre ou ne pas prendre, là n'est pas la question...

- On considère le premier objet ;
- S'il est trop gros, on passe au suivant ;
- Sinon, on ne sait pas à priori s'il faut le prendre, **on essaye donc les deux** :
  - Si on le prend, on gagne $v[0]$, mais il reste $c = C - p[0]$, et on passe à l'objet 1 ;
  - Sinon, on passe à l'objet 1.

```
int sac(int c, int i){
// Prerequis : 0 <= c <= C
//             0 <= i <= n (n: nb d'objets)
// Action : calcule la valeur maximale qu'on peut mettre
// dans un sac de taille c avec les objets >= i
```

Algorithme Récursif pour Sac à Dos

```
int sac(int c, int i){
  if(i==n){return 0}
  else
    if (p[i] > c) {
      return sac(c,i+1)}
    else {
      return max(v[i]+sac(c-p[i],i+1), sac(c,i+1))
    }
}
```

```
int sacDP(int c, int i){
  if(t[c,i]==-inf){
    int res;
    if(i==n){res= 0}
    else{
      if(p[i] > c){
        res = sacDP(c, i+1)
      }
      else{
        res = max(v[i]+sacDP(c-p[i],i+1), sacDP(c,i+1))
      }
    }
    t[c,i] = res;
  }
  return t[c,i];
}
```

Preuve d'Optimalité

Correction de sac

sacDP retourne la même valeur que sac. Pourquoi sac retourne bien la valeur annoncée ?

Notations

- Soit E l'ensemble des entrées (c, i) possibles de l'algorithme ;
- Soit $Sol(c, i)$ l'ensemble des S tel que $S \subseteq \{i, \ldots, n\}$ et $p(S) \leq c$.

Exercice

Un exemple simple au tableau :

Poids	5	2	4	2	1
Valeur	2	3	5	3	5

Preuve d'Optimalité

Problème KP-AUX

$\rightarrow$ n (nombre d'objets), C (taille du sac à dos), p (place occupée) et v (valeurs) fixés.

- **Entrée :** $(c, i) \in E$
- **Sortie :** un S avec un sac de taille c avec les objets $\geq i$
- **Objectif :** maximiser $v(S)$

Remarques :

- KP-AUX est bien un problème de maximisation "classique" ;
- $opt(c, i)$ dénote $\max_{S \in Sol(c,i)} v(S)$.

Prouvons par récursion sur i que $\forall (c,i) \in E$, $sac(c,i) = opt(c,i)$.

Cas 1 (le plus dur) : si $p[i] \leq c$

Soit $S^* \in Sol(c,i)$ une solution optimale ($p(S^*) = opt(c,i)$).

Cas 1.1 si $i \in S^*$.

- $v(S^*) = v[i] + v(S^* \setminus \{i\})$
- `sac(c,i)` $\geq v[i] +$ `sac(c-p[i], i+1)` (car on garde le max)

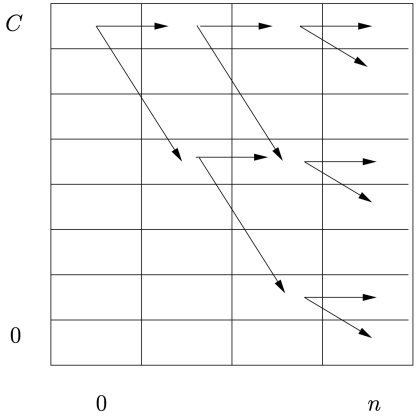
But : `sac(c-p[i],i+1)` $\geq v(S^* \setminus \{i\})$

- Par récursion, `sac(c-p[i], i+1)` $\geq opt(c - p[i], i+1)$
- $S^* \setminus \{i\}$ est une solution faisable de $(c - p[i], i+1)$
- donc $v(S^* \setminus \{i\}) \leq opt(c - p[i], i+1)$ (on a même l'égalité, mais pas utile)

Cas 1.2 si $i \notin S^*$ même idée

D'après le Théorème, `sacDP` s'exécute en temps $(C+1)(n+1) \times \mathcal{O}(1) = \mathcal{O}(Cn)$.

Que se passe-t-il vraiment quand on lance `sacDP(C,0)` avec $C = 7$, $n = 3$, $p[0] = 3, p[1] = 3, p[2] = 1$?



Complexité

- $\mathcal{O}(Cn)$ est donc un pire cas (comme si on avait calculé une fois toutes les cases), en pratique, on en fait moins (mais on a du mal à dire combien exactement) ;
- Si l'on devait exécuter à la main... on utiliserait aussi ce tableau pour éviter de recalculer les appels.

Pourquoi cela fonctionne ?

Remarques

On a l'impression de faire du brut force. Pourquoi a-t-on une complexité faible ?
$\rightarrow$ Car on a réussi à encoder efficacement la situation restante après chaque choix.

Exemple de mauvais encodage : se rappeler tous les objets $j < i$ que l'on a déjà pris :

```
int sac(list l,int i){
    // Prerequis : 0 <= i <= n (n: nb d'objets)
    //             l ne contient que des j < i
    //             le sac contient deja tous les objets de l
    // Action : calcule la valeur maximale que l'on
    // peut mettre dans la place restante avec les objets >= i
}
```

Pourquoi cela fonctionne ?

Remarques

Intuitivement : le “bon” encodage est efficace, car de nombreuses “trajectoires” aboutissent à la même situation. Par exemple, on peut avoir :

```
C->C-p[0]->C-p[0]-p[2]->C-p[0]-p[2]-p[5] = c
C-> ..    -> ..    ->C-p[0]-p[2]-p[3]-p[4] = c
C-> ..  -> ..          ->C-p[0]-p[1]-p[4] = c
  ...
C-> ..  -> ..          ->C-p[3]-p[4]-p[5] = c
```

Comment obtenir une solution ?

Comment obtenir une solution et pas seulement la valeur optimale ?

Réécrire l'algorithme pour obtenir une solution est en général trivial.

```
list sacAvecSolution(int c,int i){
  if(i==n){ return []}
  if(p[i] > c){ return sacV2(c,i+1)}
  else {
    l1 = {i} U sacV2(c-p[i],i+1);
    l2 = sacV2(c,i+1);
    if (v(l1) > v(l2)) {return l1;}
    else {return l2;}
  }
}
```

Dérecursification

On peut déduire de ces algorithmes un algorithme itératif qui remplit le tableau “dans le bon ordre”. On peut ensuite généralement réduire la mémoire utilisée en n’utilisant qu’un tableau plus petit.

Par exemple pour sac avec une complexité en $\mathcal{O}(nC)$ et mémoire utilisée en $\mathcal{O}(nC)$:

```
void sacIteV1(C,p,v){
  declarer t de taille nC
  for (i from n to 0)
    for (c from 0 to n)
      if (i==n) { t[c,i]=0}
      else {
        if (p[i] > c) { t[c,i]=t[c,i+1]}
        else {
          t[c,i]=max(v[i]+t[c-p[i],i+1],t[c,i+1]);
        }
      }
}
```

Ou alors 2 tableaux de taille C seulement. On a besoin que de la colonne courante et de celle juste à sa droite (Complexité en $\mathcal{O}(nC)$ et mémoire utilisée en $\mathcal{O}(C+n)$) :

```
void sacIteV2(C,p,v){
  declarer tprec et tcour de taille C
  init tprec avec 0
  for(i from n-1 to 0)
    for (c from 0 to n)
      if (p[i] > c) { tcour[c,i]=tprec[c,i+1]; }
      else {
          tcour[c,i] =max(v[i]+tprec[c-p[i],i+1],tprec[c,i+1]);
      }
  }
  tprec <- tcour;
}
```

Programmation Dynamique Récursive ou Itérative ?

Itérative

- **Avantage :** on gagne en mémoire ;
- **Inconvénients :** on est sûr de remplir l'équivalent de tout le tableau, et il faut réfléchir pour trouver le bon ordre de remplissage.

Récursive

- **Avantage :** il faut juste écrire l'algorithme récursif (l'ajout du tableau est trivial), on ne calculera que les cases dont on aura besoin ;
- **Inconvénients :** place mémoire, il faut tout de même initialiser le tableau.

Ce qu'il faut retenir : **Programmation Dynamique = Récursif + Mémoïsation**.
$\rightarrow$ La version itérative n'est qu'une petite optimisation.

Programmation Dynamique basée sur une structure de donnée quelconque

Remerciez les IG4 2020-2021 pour leur remarque qui a entrainé l'ajout des prochaines slides !

Remarque

Au lieu de prendre un tableau pour t, on peut prendre n'importe quelle structure de donnée (liste, hashtable, ...).

- **Avantages possibles :**
  - taille de t plus petite (égale au nombre d'entrées réellement calculées)
  - initialisation de t plus rapide
- **Inconvénients possibles :**
  - temps d'ajout/de recherche d'une entrée plus long

Exercice

Traducteur Récursif Dynamique

Comment transformer élégamment n'importe quel algorithme récursif en un algorithme de programmation dynamique si l'algorithme le permet ?

- Proposez l'idée générale.
- Trouvez une implémentation élégante dans le langage de votre choix (Python ?).

Complexité sur une structure de donnée quelconque

Soit

- n_e : nombre d'entrées possibles de l'algo ($n_e = (C+1)(n+1)$ pour Sac à Dos)
- $n_c \leq n_e$: nombre d'entrées réellement calculées
- $tpsInit(n_e)$: temps d'initialisation de t pour pouvoir stocker jusqu'à n_e éléments
- $tpsRech(n_c, n_e)$: temps de recherche quand t contient n_c éléments
- $tpsAjout(n_c, n_e)$: temps de recherche quand t contient n_c éléments
- c la complexité d'un appel en comptant $\mathcal{O}(1)$ pour chaque appel récursif ($c = \mathcal{O}(1)$ pour Sac à Dos)
- $\Delta \leq c$ le nombre maximum d'appels récursifs faits depuis un appel à l'algo ($\Delta = 2$ pour Sac à Dos)

Théorème Complexité en Temps :

La complexité d'un algorithme de programmation dynamique (qui termine) basée sur une structure de donnée quelconque est :

$$tpsInit(n_e) + \mathcal{O}(n_c \times (c + \Delta tpsRech(n_c, n_e) + tpsAjout(n_c, n_e)))$$

Pour un tableau classique :

- $\mathcal{O}(n_e) + \mathcal{O}(n_c \times (c + \Delta\mathcal{O}(1) + \mathcal{O}(1))) = \mathcal{O}(n_e) + \mathcal{O}(n_c c)$
  $\leq \mathcal{O}(n_e c)$ (pour retrouver formule d'avant)

Pour une liste chainée :

- $\mathcal{O}(1) + \mathcal{O}(n_c \times (c + \Delta\mathcal{O}(n_c) + \mathcal{O}(n_c)) \leq \mathcal{O}(n_c c) + \mathcal{O}(n_c^2 \Delta)$
  $\leq \mathcal{O}(n_e^2 c)$ (plus intéressant : on a trop majoré)

Théorème Complexité en Espace :

L'espace mémoire d'un algorithme de programmation dynamique (qui termine) basée sur une structure t est de

$$\mathcal{O}(|t|_{max}) + m$$

où $|t|_{max}$ est la taille maximale atteinte par t et m est l'espace supplémentaire correspondant aux variables locales.

Très souvent :

- $m = \mathcal{O}(|t|_{max})$
- $|t|_{max} \leq n_e$ (ou même n_c avec une liste)

En particulier :

- avec un tableau, mémoire $\mathcal{O}(n_e) + m$
- avec une liste, mémoire $\mathcal{O}(n_c) + m$

Sommaire

Plus Court Chemin dans un Graphe

Soit $G = (V, A)$ un graphe, où V est l'ensemble des sommets et A l'ensemble des arcs.

- Le poids de l'arc a est un entier naturel $l(a)$;
- La longueur d'un chemin P est égale à la somme des longueurs des arcs qui les composent (notée $l(P)$).

Problème SP (Shortest-Path)

- **Entrée :** un graphe G, et deux sommet s et y
- **Sortie :** un s-y chemin P
- **Objectif :** minimiser $l(P)$

Remarque : pour l'instant, on suppose qu'il n'y a pas de cycles négatifs.

Fixons s, et partons de t, en cherchant à reculer vers s :

Fixons s, et partons de t, en cherchant à reculer vers s :

Problème SP-aux

G et s sont fixés.

- **Entrée :** un sommet y
- **Sortie :** un s-y chemin P
- **Objectif :** minimiser $l(P)$

Fixons s, et partons de t, en cherchant à reculer vers s :

Problème SP-aux

G et s sont fixés.

- **Entrée :** un sommet y
- **Sortie :** un s-y chemin P
- **Objectif :** minimiser $l(P)$

- Quel prédécesseur de y devrait-on utiliser ?

Fixons s, et partons de t, en cherchant à reculer vers s :

Problème SP-aux

G et s sont fixés.

- **Entrée :** un sommet y
- **Sortie :** un s-y chemin P
- **Objectif :** minimiser $l(P)$

- Quel prédécesseur de y devrait-on utiliser ?
- On ne sait pas, donc on branche !

```
int A(y){
    //A(y) calcule opt(y)
    if(y==s) return 0;
    else{
      let P(y) = ensemble des predecesseurs de y
      return min_{z \in P(y)}(A(z)+l(zy))
    }
}
```

Quel est le problème de A ?

- Boucle infinie potentielle à cause des cycles ;
- Rien ne devient “plus petit” lors des appels récursifs.

Solution : On impose un nombre maximum k d’arêtes autorisées et c’est ce paramètre qui va diminuer dans les appels récursifs.

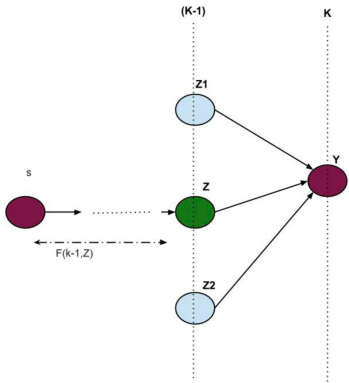
Problème SP-aux-V2

G et s sont fixés.

- **Entrée :** (y, k) avec y un sommet, et k un entier $(0 \leq k < |V(G)|)$
- **Sortie :** un s-y chemin P ayant au plus k arêtes
- **Objectif :** minimiser $l(P)$

Si on sait résoudre *SP-aux-V2*, alors il suffira de demander à le résoudre sur $(y, n-1)$ pour avoir le s-y chemin le plus court car un tel chemin utilise bien au plus $n-1$ arêtes.

```
void A2(k,y){
 //A2(k,y) calcule opt(k,y)
 if(k==0){
   if(y==s) return 0;
   else return infini;
 }
 else{ //k != 0
   if(y==s) return 0 (car pas cycle negatifs)
   else
     let P(y) = ensemble des predecesseurs de
     return min_{z \in P(y)}(A2(k-1,z)+l(zy))
   }
}
```



Complexité (taille du tableau $= n^2$, $c = n$) : $\mathcal{O}(n^3)$.

Conclusion

La programmation dynamique résout un problème en le décomposant en sous-problèmes, puis en résolvant ces derniers en enregistrant les résultats intermédiaires.

La programmation dynamique est un bon outil lorsque :

- la structure dans laquelle on cherche une solution à un ordre naturel pour la parcourir ;
- de nombreuses suites de décisions locales différentes peuvent aboutir au même état.